User Manual | Version 5.7 | 22 July 2020

HVR Software, Inc.
135 Main Street, Suite 850
San Francisco, CA, 94105

https://www.hvr-software.com

Table of Contents

Introduction

HVR is a powerful software product that enables real-time homogeneous and heterogeneous data replication. HVR uses various CDC (Change Data Capture) methods to replicate changes between databases, directories (file locations), as well as between databases and directories, that HVR calls 'locations'. Locations can either be a source or a target. Each change in a source location is captured by HVR, transmitted and then applied to a target location. Database CDC technology is also referred to as a 'log mining process' that reads a database transaction log for relevant transactions. HVR uses its own internal log mining technology along with certain database vendor APIs. The CDC method that HVR uses during a replication depends on various settings defined/configured within HVR.

HVR has a built-in compare feature that allows real-time verification to ensure that the source and target locations are in sync. In addition, HVR has a replication monitoring feature allowing users to actively monitor the status of replication, including viewing real-time data flow statistics. All actions can be securely monitored using the event audit feature to ensure that all actions taken are logged.

Advantages

- Log based CDC has minimal impact on a source databases
- Low latency gathering of changes made on a source database
- Changes keep transactional integrity
- Changing source applications is not required
- Flexibility allows for trade-offs in remote versus local capture, as well as capture once, deliver to multiple, scenarios
- Resiliency against failures allows for recovery without data loss
- Setup is available both via graphical user interface and CLI

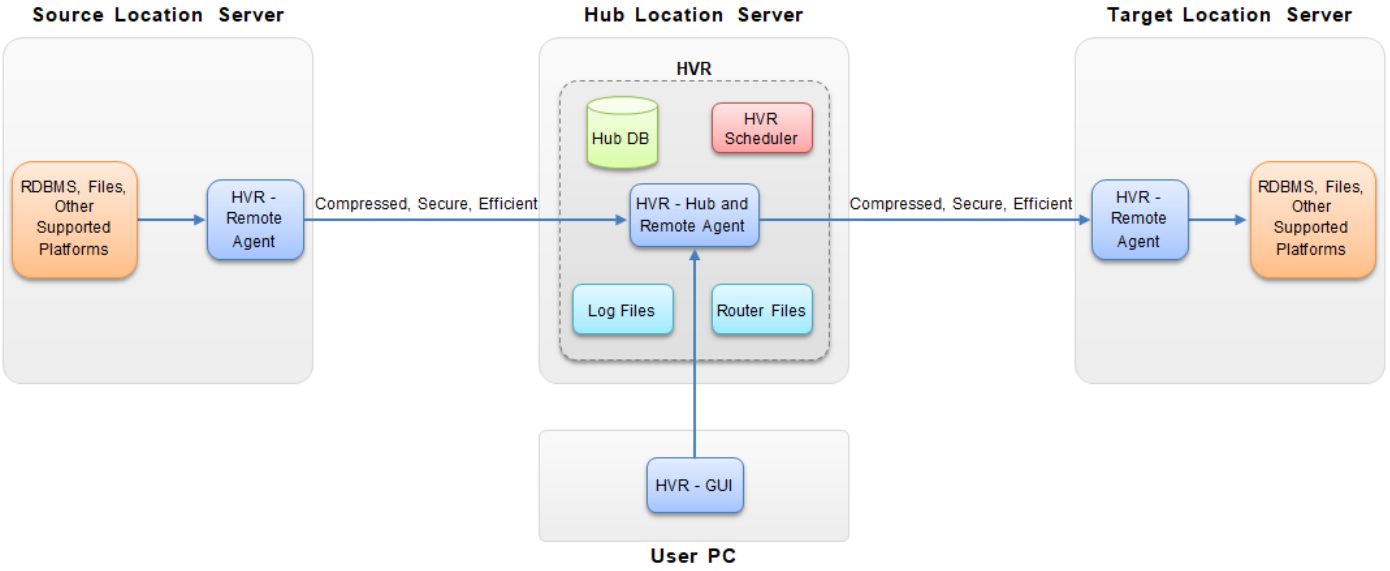
Capabilities

- Feed a reporting database
- Populate a data warehouse or data lake
- Feed Kafka or other streaming platforms
- Migrate from on-premises to cloud e.g. move from on-premises Oracle to AWS Oracle RDS with little or no downtime
- Move data from one cloud vendor to another cloud vendor supporting intra-cloud, inter-cloud and hybrid cloud deployments
- Consolidate multiple databases
- Keep multiple geographically distributed databases in sync
- Migrate from one hardware platform to another, e.g. move from an AIX platform to a Linux platform with little or no downtime
- Migrate from an older database version to the latest supported version
- Migrate from one database technology to another, e.g. from Oracle to PostgreSQL

Architecture Overview

HVR supports a distributed architecture for database and file replication. HVR is a comprehensive software system that has all of the needed modules to run replication. This includes a mechanism called **HVR Refresh** for the initial loading of the database, a continuous **capture** process to acquire all the changes in the source location, an **integrate** (or apply) process that applies the changes to the target location, and a **compare** feature that compares the source and target locations to ensure that the data is the same on both sides. A **location** is a storage place (for example, database or file storage) from where HVR captures (source location) or integrates (target location) changes. Locations can be either local (i.e. residing on the same machine as the HVR hub) or remote (residing on a remote machine, other than the hub).

HVR software may be installed on the most commonly used operating systems. HVR reads the transaction logs of the source location(s) in real time. That data is then compressed, optionally encrypted, and sent to a 'hub machine'. The hub then routes the data and then integrates (applies) the data into the target location(s).



HVR Hub

The HVR hub is an installation of HVR on a server machine (hub server). The HVR hub orchestrates replication in logical entities called channels. A **channel** groups together locations and tables that are involved in the replication. It also contains actions that control the replication. The channel must contain at least two locations. The channel also contains location groups - one for the source location(s) and one for the target location(s). **Location groups** are used for defining actions on the locations. Each location group can contain multiple locations.

The hub machine contains the HVR hub database, Scheduler, Log Files, and Router Files.

- **Hub Database**
  This is a database which HVR uses to control replication between source and target locations. For the list of databases that HVR supports as a hub database, see section Hub Database in Capabilities. The hub database contains HVR catalog tables that hold all specifications of replication such as the names of replicated databases, replication direction and the list of tables to be replicated.
- **HVR Scheduler**
  The hub runs an **HVR Scheduler** service to manage replication jobs (**Capture** jobs, **Integrate** jobs, **Refresh** jobs, **Compare** jobs) that move data flow between source location(s) and target location(s). To either capture or apply (**integrate**) changes, the **HVR Scheduler** on the hub machine starts the capture and integrate jobs that connect out to source and target locations.
- **Log Files**
  Log files are files that HVR creates internally to store information from scheduled jobs (like **Capture** jobs, **Integrate** jobs, **Refresh** jobs, **Compare** jobs) containing a record of all events such as transport, routing and integration.
- **Router Files**
  Router files are files that HVR creates internally on the hub machine to store a history of what HVR captured and submitted for integration, including information about timestamps and states of the capture and integrate jobs, transactions, channel locations and tables, instructions for a replication job, etc.

HVR Remote Agent

Any installation of the HVR software can play the role of the HVR hub and/or an HVR remote agent. The HVR remote agent is an installation of the HVR software on a remote source and/or target machine that allows to implement a distributed setup. The HVR hub can also be configured to work as a HVR remote agent to enable the HVR GUI on a user's PC to connect to the HVR hub.

The HVR remote agent is quite passive and acts as a child process for the hub machine. Replication is entirely controlled by the hub machine.

⚠ Even though HVR recommends using HVR remote agent with the distributed setup, HVR can also support an agent-less architecture.

To access a remote location, the HVR hub normally connects to the HVR remote agent using a special TCP/IP port number.

- If the remote machine is Unix or Linux, then the system process (daemon) is configured on the remote machine to listen on this TCP/IP port. For more information, refer to section Configuring Remote Installation of HVR on Unix or Linux.
- If the remote machine is a Windows machine, then HVR listens using **HVR Remote Listener** (a Windows service). For more information, refer to section Configuring Remote Installation of HVR on Windows.

Alternatively, HVR can connect to a remote database location using a DBMS protocol such as Oracle TNS.

HVR GUI

HVR can be managed using a Graphical User Interface (GUI). The **HVR GUI** can run directly on the hub machine if the hub machine is Windows or Linux.

Otherwise, it should be run on the user's PC and connect to the remote hub machine. In this case, the HVR installation on the hub machine will play dual role:

- Works as an HVR hub which will connect to the HVR remote agent available on the source and target locations.
- Works as an HVR remote agent to enable the HVR GUI on the user's PC to connect to the HVR hub.

HVR Refresh

The **HVR Refresh** feature allows users to initially load data directly from source tables to target tables or files. To perform the initial materialization of tables, users simply create target tables based on the source layouts, and then move the data from the source tables to the target. HVR provides built-in performance options to help reduce the time it takes to initially load the tables. HVR can run jobs in parallel, either by table or location. For large tables, you can instruct HVR to slice the data into data ranges for improved parallelism. Behind the scenes, HVR further improves initial load performance by using the native bulk load capabilities of the database vendor, which typically offer the most efficient way to load the data without requiring HVR users to configure utilities or write scripts.

HVR Compare

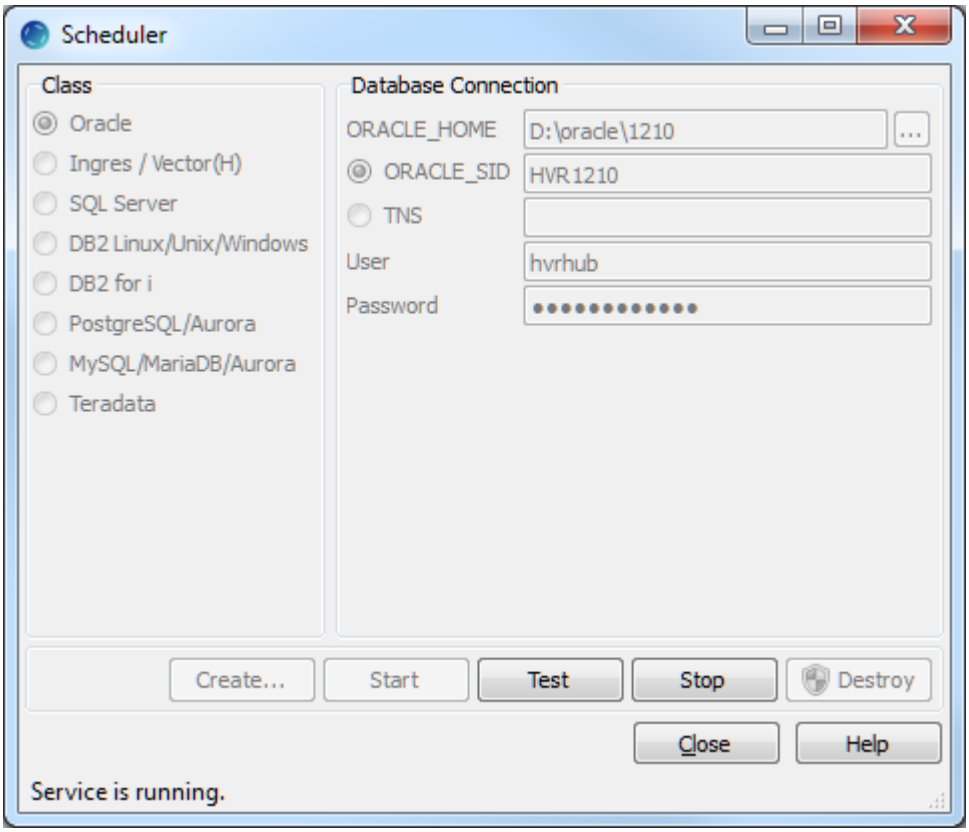
The **HVR Compare** feature ensures that the source and target locations are both 100% the same. HVR has two methods of comparing data: bulk and row by row. During the bulk compare, HVR calculates the checksum for each table in the channel and compares these checksums to report whether the replicated tables are identical. During the row by row compare, HVR extracts data from a source (read) location, compresses it, and transfers the data to a target (write) location(s) to perform the row by row compare. Each individual row is compared to produce a 'diff' result. HVR also has a repair feature. For each difference detected, an SQL statement is written: an insert, update or delete. This SQL statement can then be executed to repair so that the source and targets are the same. For large tables, you can instruct HVR to slice the data into data ranges for improved parallelism.

Replication Overview

The runtime replication system is generated by command **HVR Initialize** in the GUI or **hvrinit** from the command line on the hub machine. **HVR Initialize** checks the channel and then creates the objects needed for replication, plus replication jobs in the **HVR Scheduler**. Also for trigger based capture (as opposed to log based capture), HVR creates database objects such as triggers (or 'rules') for capturing changes. Once **HVR Initialize** has been performed, the process of replicating changes from source to target location occurs in the following steps:

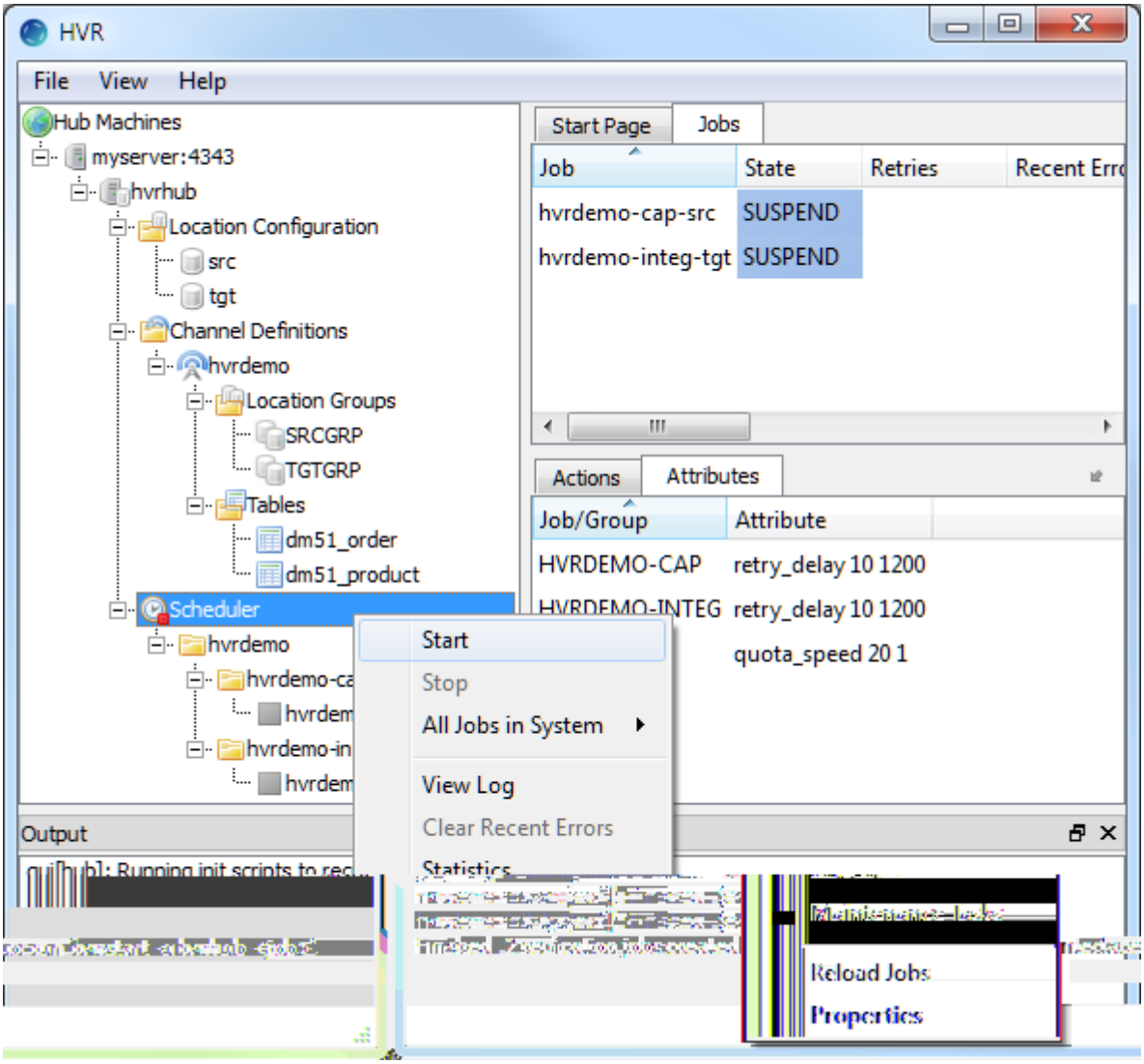
1. Changes made by a user are captured. In case of log based capture these are automatically recorded by the DBMS logging system. For trigger based capture, this is done by HVR triggers inserting rows into capture tables during the user's transaction.
2. When the 'capture job' runs, it transports changes from the source location to router transaction files on the hub machine. Note that when the capture job is suspended, changes will continue being captured (step 1).
3. When the 'integrate job' runs, it reads from the router transaction files and insert, update and delete statements on the target location to mimic the original change made by the user.

Runtime replication requires that the HVR Scheduler is running. Right click on the hub database to create and start the **HVR Scheduler**.



HVR Initialize creates jobs in suspended state. These can be activated using the GUI by right clicking on a channel and select **Start**. Like other operations in the HVR GUI, starting jobs can also be done from the command line. See command **hvrstart**.

The **HVR Scheduler** collects output and errors from all its jobs in several log files in directory **$HVR_CONFIG/log/hubdb** . Each replication job has a log file for its own output and errors, and there are also log files containing all output and only errors for each channel and for the whole of HVR.

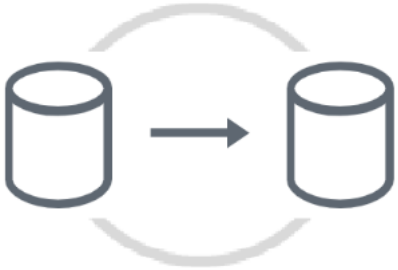


To view errors, right-click the job underneath the **Scheduler** node in the navigation tree pane and select **View Log**. These log files are named **chn_cap_loc.out** or **chn_integ_loc.out** and can be found in directory **$HVR_CONFIG/log/hubdb**.

Replication Topologies

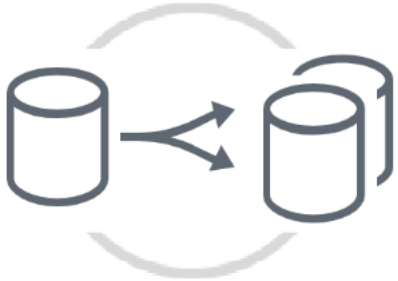
HVR supports data replication across multiple heterogeneous systems. Below are basic replication topologies that allow you to construct any type of replication scenario to meet your business requirements.

Uni-directional (one-to-one)



This topology involves unidirectional data replication from a source location to a target location. This type of topology is common when customers look to offload reporting, feeding data lakes, and populating data warehouses.

Broadcast (one-to-many)

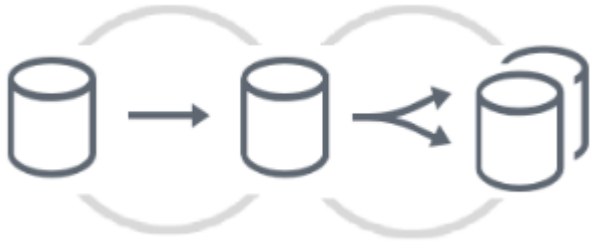


This topology involves one source location, from which data is distributed to multiple target locations. The one-to-many topology is often used to distribute the load across multiple identical systems, or capture once and deliver to multiple destinations. For example, this may be adopted for cloud technologies targeting both a file-based data lake using a distributed storage systems such as S3 or ADLS, as well as a relational database for analytics such as Snowflake, Redshift, or Azure Synapse Analytics.

Consolidation (many-to-one)

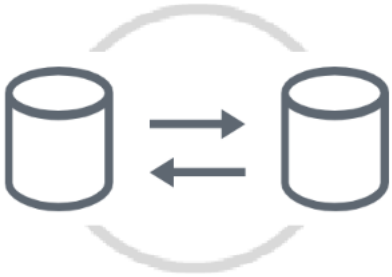
This topology involves multiple source locations consolidating data into one target location. The many-to-one is the topology for data warehouse or data lake consolidation projects, with multiple data sources feeding into a single destination, or for the use case of multiple distributed systems, typically containing a subset of data (e.g. local branches), all feeding into a central database for analytics and reporting.

Cascading



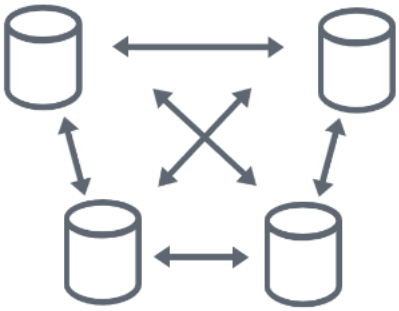
This topology involves a source location pushing data to a target location, whereas the target also acts as a source distributing the data out to multiple target locations. This is typically used to replicate data into a data warehouse and building individual data marts.

Bi-directional (active/active)



A bi-directional topology assumes that data is replicated in both directions, and end users (applications) modify data on both sides. It is also referred to as an active/active scenario to keep two systems in sync, giving the ability to share the replication load across the systems, as well as their high availability. This is typical in a geographically distributed setup, where the data should always be local to the application, or in a high availability setup. HVR provides a built-in loop-back detection mechanism to protect the systems from boomerang loop-back issues, as well as collision detection and resolution mechanism. For steps to configure a bi-directional replication, see section Configuring Multi-Directional Replication.

Multi-directional



A multi-directional active/active replication involves more than two locations that stay synced up. This is a typical scenario for geographically distributed systems, where any changes made to any node will be propagated to all the other nodes within this network. This type of replication generally introduces a number of additional challenges beyond regular active/active replication: network latency, bandwidth reliability, or a combination of these. HVR provides technology to address all these challenges, including automatic recovery via the built-in scheduler, optimized network communication with high data compression, etc. For steps to configure a multi-directional replication, see section Configuring Multi-Directional Replication.

HVR Release Life Cycle

Contents

Support Duration for HVR Releases

- Major HVR versions released since 2019 (starting from HVR 5.6) are supported for 3 years from their General Availability (GA) release date.
- A minor release (e.g. HVR 5.3.0) or an HVR patch (e.g. HVR 5.3.0/1 or HVR 5.3.0/1.1) does not prolong the support duration. For example, if HVR 5.3 will expire at date X then that date will not be extended if HVR 5.3.2 is released a year later.
- HVR offers "extended" support beyond these end of support dates. The duration of extended support is determined on a case-by-case basis.

The following table shows the release and support dates for the most recent major HVR releases:

HVR Release	GA Release Date	End of Support	Support Status
HVR 5.7	2020-07-22	2023-12-31	Regular
HVR 5.6	2019-07-08	2023-12-31	Regular
HVR 5.5	2018-10-23	2023-10-23	Regular
HVR 5.3	2018-01-02	2023-01-02	Regular
HVR 5.2	2017-07-19	2022-07-19	Regular
HVR 5.1	2016-10-18	2021-10-18	Regular
HVR 5.0	2016-04-20	2021-04-20	Regular
HVR 4.7	2015-03-24	2020-03-24	Support Ended

Compatibility Between HVR Versions

- HVR versions are 'network compatible' with the two previous HVR major versions, but not with HVR versions that have a different initial number. For example HVR 5.6 is network compatible with 5.5 and 5.3, but not with HVR 5.2 nor with (future) HVR 6.0.
- When using a mix of HVR versions, note that each bug fix or feature is only effective if the correct installation is upgraded. For example, if a new HVR release fixes an integrate bug, then that release must be installed on the installation(s) which do(es) integrate. So if only the GUI and/or hub installation(s) is(are) upgraded, then there will be no benefit. To decide whether each installation needs to be upgraded, see the descriptions in the release notes (available in **HVR_HOME/hvr.rel**). Each description ends with a line saying which installation(s) must be upgraded for that specific bug fix or feature to be effective.

Supported Operating System (OS) Releases

- The HVR release notes (available in **HVR_HOME/hvr.rel**) lists the supported OS versions (such as "Solaris for SPARC, version 10 and higher") for each HVR release.
- HVR only specifies the initial OS version for Unix systems; certification for subsequent OS versions (e.g. Solaris version 10) by HVR is automatic due to OS supplier's own forward compatibility guarantees.

- Once an OS version is supported by HVR, support for that platform will continue until the OS supplier ends its "mainstream support". After that date HVR support goes from "regular" to "Sunset" support, which means that HVR support continues (including new HVR versions for it), but eventually it may be withdrawn without notice.
- A customer may request that HVR makes support for an OS version "extended" instead of "Sunset", which means that support will not be withdrawn as long as the customer continues to share information about the production status of that OS version for the customer and HVR continues to have the reasonable ability to support the platform.
- A consequence of the end of HVR support is that HVR is no longer able to supply patches for that OS version.
- HVR supports Linux based on a minimal glibc version of 2.12 which is not limited to the Enterprise products below. As long as the version of Linux is compatible with glibc it should work.

To see which OS releases are supported by HVR, see Platform Compatibility Matrix.

The following table shows the OS suppliers support status of specific OS versions as known by HVR.

OS Version	Release Date from Supplier	End of Mainstream Support from Supplier	HVR Support Status
AIX 7.2	2015-12-01	2022-11-30	Regular
AIX 7.1	2010-19-10	2023-04-20	Regular
AIX 6.1	2008-09-12	2017-04-30	Sunset by IBM
HPUX Itanium 11.31	2007-01-02	2014-01-12	Sunset by HP
HPUX Itanium 11.23	2003-01-10	2010-01-12	Sunset by HP
Red Hat 8	2019-05-07		Regular
Red Hat 7	2014-06-10	2024-06-10	Regular
Red Hat 6	2010-10-11	2020-11-30	Regular
Red Hat 5	2007-03-15	2017-03-31	Sunset by RedHat
SUSE ES 15	2018-07-16		Regular
SUSE ES 12	2014-10-27	2024-10-31	Regular
SUSE ES 11	2009-03-24	2019-03-31	Sunset by SUSE
Solaris 11	2011-01-11	2021-01-11	Regular
Solaris 10	2005-01-01	2018-01-01	Sunset by Oracle
Windows Server 2016	2016-09-26	2022-01-11	Regular
Windows 10	2015-07-29	2019-04-09	Sunset by Microsoft
Windows Server 2012	2012-10-30	2018-09-10	Sunset by Microsoft
Windows Server 2008	2008-12-29	2015-01-13	Sunset by Microsoft

Supported DBMS Releases

- The HVR release notes (in **$HVR_HOME/hvr.rel**) lists the supported DBMS versions (such as "Oracle: version 12.1, 12.2, 18c and 19c") for each HVR release. The list of supported DBMS versions is different for HVR on each OS platform.
- Subsequent major versions of the DBMS (such as "Oracle 20") are not automatically supported, but may be recertified by HVR after QA testing. But patches to a supported DBMS version are automatically supported by HVR (no recertification is done at patch level).
- Once a DBMS version is supported by HVR on a given OS platform, support for that DBMS version will continue at least until the DBMS supplier ends its "mainstream support". After that date HVR support goes from "regular" to "Sunset" support, which means that HVR support continues (including new HVR versions for it), but eventually it will be withdrawn without notice.

- A customer may request that HVR makes support for a DBMS version "extended" instead of "Sunset", which means that support will not be withdrawn as long as the customer continues to share information about the production status of that DBMS version for the customer.
- A consequence of the end of HVR support is that HVR is no longer able to supply patches for that DBMS.

To see which DBMS releases are supported by HVR, see Platform Compatibility Matrix.

The following table shows the DBMS suppliers support status of specific DBMS versions as known by HVR.

DBMS Version	Release Date from Supplier	End of Mainstream Support from Supplier	HVR Support Status
Oracle 19	2019-02-13	2026-03-31	Regular
Oracle 18	2018-02-16	2021-06-08	Regular
Oracle 12.2	2017-03-01	2022-03-31	Regular
Oracle 12.1	2013-01-06	2018-01-07	Sunset by Oracle
Oracle 11.2	2009-01-09	2015-01-01	Sunset by Oracle
Ingres 11.1	2019-04-17	2023-04-30	Regular
Ingres 11.0	2017-03-21	2022-06-30	Regular
Ingres 10.2	2014-09-04	2019-09-01	Sunset by Actian
Ingres 10.1	2012-05-01	2017-05-31	Sunset by Actian
Ingres 10.0	2010-09-01	2016-12-31	Sunset by Actian
SQL Server 2019	2019-11-04	2025-01-17	Regular
SQL Server 2017	2017-10-02	2022-10-11	Regular
SQL Server-2016	2016-06-01	2021-07-13	Regular
SQL Server 2014	2014-06-05	2019-06-05	Sunset by Microsoft
SQL Server 2012	2012-05-20	2017-11-07	Sunset by Microsoft
SQL Server 2008	2008-11-06	2014-08-07	Sunset by Microsoft
DB2 for i 7.4	2019-04-15	Not Announced	Regular
DB2 for i 7.3	2016-04-15	Not Announced	Regular
DB2 for i 7.2	2014-05-02	2021-04-30	Regular
DB2 for i 7.1	2010-04-23	2018-04-30	Sunset by IBM
DB2 for LUW 11.5	2019-06-25	Not Announced	Regular
DB2 for LUW 11.1	2016-06-01	Not Announced	Regular
DB2 for LUW 10.5	2013-07-26	Not Announced	Regular
DB2 for LUW 10.1	2012-06-11	2017-09-30	Sunset by IBM
DB2 for LUW 9.8	2010-01-15	2015-01-05	Sunset by IBM
DB2 for LUW 9.7	2010-11-19	2015-01-05	Sunset by IBM

Analytic DBMS Version	Release Date from Supplier	End of Mainstream Support from Supplier	HVR Support Status
Actian Vector (Vectorwise) 5.x	2016-06-01	2020-06-30	Regular
Actian Vector (Vectorwise) 4.x	2015-03-01	2018-12-31	Sunset by Actian

Greenplum 5.x	2017-09-14	2020-09-30	Regular
Greenplum 4.3.x	2013-12-05	2020-02-29	Sunset
Teradata 16 & 16.10			
Teradata 15 & 15.10			Regular

Document Conventions

Contents

This section explains the conventions used in HVR documentation.

Convention	Description
bold	Indicates computer 'words' that are fixed like actions, action parameters, commands, command parameters, file names, directory paths, or SQL command in running paragraphs, such as '**grant execute permission**'.
italics	Indicate computer words that are variable or placeholder text. For example, in a directory path such as **$HVR_CONFIG/log/***hubdb* the word 'hubdb' is in italics indicating that it is a placeholder text and should be replaced by the actual word that is applicable to you.
`hvrsuspend -u hubdb channel`	Code examples, syntax, or commands recognized by the system are displayed in code block with mono spaced font.
[ ]	Square brackets indicate optional arguments/entries. There can be multiple arguments in square brackets.
|	The vertical line or pipe sign (|) separates a mutually exclusive set of options /arguments.

Menu

Menu selection sequences are displayed in bold and each selection is divided by the '' sign. For example, select **Tools Data Entity Organization**. This means select the menu option Tools in the menu bar, then select menu option **Data** in the **Tools** menu, followed by selecting menu option **Entity** in the **Data** sub-menu and finally click on menu option **Organization** in the **Entity** sub-menu.

File or Directory Path

Many path-names are shown using Unix conventions, e.g. using a forward slash '/' as file or directory path delimiter.

For the Microsoft Windows platform this can be understood as a backward slash '\'.

Generally HVR converts between forward and backward slashes as appropriate, so the two can be used interchangeably.

Info

Indicates this is an informational text.

Note

Indicates this is a note.

Platform Specific Functionality

Oracle	Indicates functionality only supported on Oracle
Ingres	Indicates functionality only supported on Ingres or Vectorwise
SQL Server	Indicates functionality only supported on SQL Server
DB2	Indicates functionality only supported on DB2
Salesforce	Indicates functionality only supported on Salesforce.com
Unix & Linux	Indicates functionality only supported on Unix and Linux
Windows	Indicates functionality only supported on Microsoft Windows

Available Since

Since vX.X.X	Indicates the version of HVR that the item/feature was introduced.

Installing and Upgrading HVR

This section provides information on installing, upgrading, and configuring HVR software on various platforms. Before proceeding with the installation, ensure that the following are available.

HVR Installation Distribution

Download an HVR installation distribution for your specific operating system at https://www.hvr-software.com/account/. To request a trial version, please visit https://www.hvr-software.com/free-trial/. Read the "COMPATIBILITY" section of the accompanying release notes (**hvr.rel** available in **HVR_HOME** directory) to ensure the version is compatible with the operating system and DBMS or refer to the relevant section in the HVR documentation - Platform Compatibility Matrix.

Hub Database

Install a database that will serve as a hub database. This database can be Oracle, Ingres, Microsoft SQL Server, DB2 for Linux, UNIX and Windows, DB2 for I, PostgreSQL, Hana or Teradata. Initially, the database may be empty. The hub database will be used as a repository for the HVR channels. It will contain the list of tables to be replicated, actions defined, etc. For Oracle, the hub database is normally just an empty schema within the capture database or the target database. For more information about the grants for hub database, refer to the respective Location Class Requirements section.

HVR License File

After purchasing an HVR license, the HVR technical support will send you an email with an HVR license file (**hvr.lic**). For more information on the license file and how to install it, refer to the HVR Licensing section.

The following topics provide essential information needed to install the HVR software on various platforms.

HVR Licensing

Contents

HVR License File

After installing the HVR software, you need to obtain an HVR license file (**hvr.lic**) which is normally delivered separately to each customer by the HVR technical support when purchasing an HVR license. A license file may have a date attached (e.g. **hvr20201010.lic**). Ensure to rename it as **hvr.lic** when you install the file. The expiration date is often added so that you are aware of the date the license expires.

After receiving the license file, save it to the **$HVR_HOME/lib** directory.

An HVR license file contains the comments section on what the license mandates. For example, the contents of the file may be as follows:

```
&2Z932nc0hAb018rSJWY/54NdPWCD7omsgZ7VdTQx742uS3j3I1G70pCq5TScCuBIWU8cBCPwnFoZjuVV4yNXF#&yPNHE.ITtNb/hVpwQkPxzPLyXGOvfW
# HVR license issued 2020-03-30
#
# Hub platform: *
# Expiry date:  2025-04-01
# HVR features: repl|cmp|refr
# Capture locations:    classes=file|sqlserver
# Integrate locations:  classes=sqlserver|file
```

Multi-Licensing

HVR supports multi-licensing scenario when multiple license files are supplied for a system, i.e. one for a certain number of source/targets, and another for a specific feature like SapXForm. In this case, all the license files should stored in the **$HVR_HOME/lib** directory, but with different names (e.g. **hvr.lic** and **hvrsapx.lic**). They must all end with a **.lic** suffix. Their effect is "additive", that is, if one allows 3 sources and the other allows 2 sources, then 5 sources are allowed for that system.

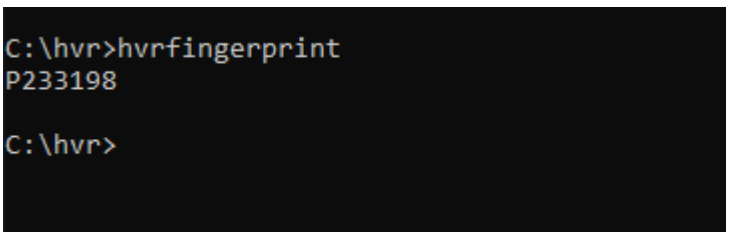
Fingerprint Restricted License

Some HVR licenses require a fingerprint, which is a unique identifier of the machine, on which the HVR hub is installed. You will need to provide the fingerprint to the HVR technical support in order to obtain the fingerprint restricted license.

Prerequisites: The HVR hub should be installed on the machine.

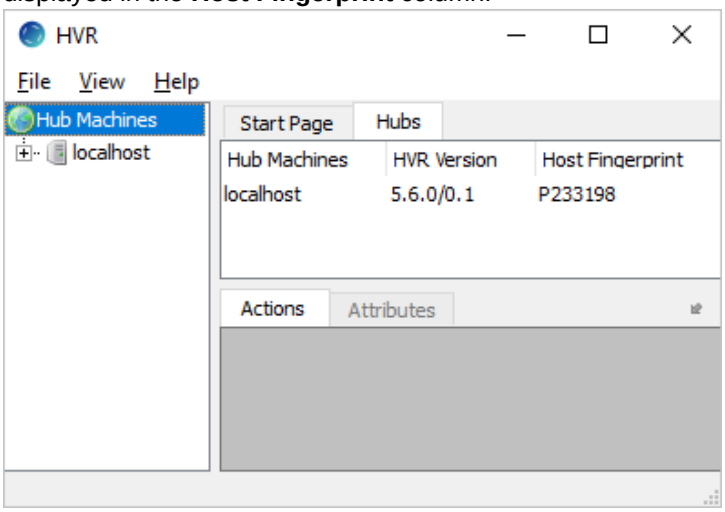
There are two ways to determine the hub fingerprint:

- Execute the **hvrfingerprint** command in the command prompt utility on the machine, on which the HVR hub is installed.

  or

- Register the HVR hub in the **HVR GUI** and click the **Hubs** tab in the right pane. The relevant hub fingerprint is displayed in the **Host Fingerprint** column.



⚠ When you first connect to a hub database without a license, you may receive a warning message requesting to obtain a regular license file from the HVR technical support and displaying the fingerprint of the hub machine. Alternatively, for locations with on-demand licensing, such as Amazon Marketplace or Azure, you need to configure the on-demand licensed location using the **LocationProperties** **/CloudLicense** parameter.

An HVR license file with a fingerprint may be as follows:

```
&2Z932nc0hAb018rSJWY/54NdPWCD7omsgZ7VdTQx742uS3j3I1G70pCq5TScCuBIWU8cBCPwnFoZjuVV4yNXF#&yPNHE.ITtNb/hVpwQkPxzPLyXGOvfW
# HVR license issued 2020-03-30
#
# Hub platform: *
# Expiry date:  2025-04-01
# HVR features: repl|cmp|refr
# Capture locations:    classes=file|sqlserver
# Integrate locations:  classes=sqlserver|file
# Hub fingerprint: P233198
```

System Requirements

This section provides information on the hardware and software requirements for HVR. It contains the following sections:

HVR Installation Location

The HVR software may be installed as an HVR hub and as a remote agent.

For using HVR as HVR hub, the HVR software must be installed on the hub machine. If a database involved in replication is not located on the hub machine, HVR can connect to it in two ways: either using the network database's own protocol (e.g. SQL*Net or Ingres/Net) or using an HVR remote connection (recommended).

HVR remote agent is an additional installation of HVR (within the HVR distributed architecture) that does not play the role of the HVR hub and does not run the **Scheduler** service. The HVR remote agent is installed on a remote machine of the source or target location for optimized communication with the HVR hub. For the HVR remote agent location, the HVR software can also be installed on the remote machine to connect to the hub machine. If the DBMS's own protocol is used for remote connection, then no HVR software needs be installed on the remote agent machine.

If the HVR replication configuration is altered, all the reconfiguration steps are performed on the hub machine; installation files on the remote machine are unchanged.

It is recommended that HVR is installed under its operating system account (e.g. **hvr**) on the hub machine and each remote location containing the replicated database or directory. However, HVR can also use the account that owns the replicated tables (e.g. the DBA's account) or it can use the DBMS owner account (e.g. **Oracle** or **Ingres**). Any login shell is sufficient.

HVR makes no assumptions about there being only one installation per machine, and locations sharing a single machine need only have HVR installed once and HVR can easily replicate data between these locations.

Disk Requirements

The full HVR distribution occupies approximately 90 MB. The following illustrates the disk usage on the hub machine:

HVR_HOME	200 MB
HVR_CONFIG	
diff	Contains differences identified by event-based compare. These are compressed files and could amount to a lot of data unless cleaned up.
files	Smaller than 1 MB
intermediate	For event-based file compare with direct connection to a local file target. This scenario is rare but could lead to a lot of data in the directory.
job	Smaller than 1 MB
jobstate	For the event-based compare. There will not be much data in the directory.
jnl	Compressed data files, grows every time data is replicated. Only used if **Integrate /JournalRouterFiles** is enabled, can then be cleaned up/maintained through a maintenance task.
log	Output from scheduled jobs containing a record of all events such as transport, routing and integration.
logarchive	Copies of log files from **/log** directory created by command **hvrmaint –archive_keep_days**.
router	Compressed data files, grows if replication has a backlog.
tmp	Temporary files if **$HVR_TMP** is not defined.
HVR_TMP	Temporary files and working directory of scheduled jobs.

The following illustrates the disk usage on a remote location:

HVR_HOME	200 MB. For a HVR remote location, this space can be reduced to only the commands required for the HVR remote agent using command **hvrstrip -r**.
HVR_CONFIG	
files	Smaller than 1 MB
intermediate	For event-based file compare with direct connection to a local file target. This scenario is rare but could lead to a lot of data in the directory.
tmp	Temporary files if **$HVR_TMP** is not defined.
HVR_TMP	Temporary files.

For replicating files, HVR also requires disk space in its 'state directory'. This is normally located in sub-directory **_hvr_state** which is inside the file location's top directory, but this can be changed using parameter **LocationProperties /StateDirectory**. When capturing changes, the files in this directory will be less than 1 MB, but when integrating changes, HVR can make temporary files containing data being moved at that moment.

Network Requirements

HVR's network communication initiated through the **HVR Remote Listener** is optimized for high latency, low bandwidth network configurations, so it is recommended to use it, especially in Wide Area Network (WAN) scenarios. HVR is extremely efficient over WAN and uses the minimum possible bandwidth. This is because HVR sends only changes made to a database, not the entire database. It also bundles many rows into a single packet and it does very few network roundtrips. Its compression ratio is typically higher than 90 percent. Compression ratios vary depending on the data types used but may be as high as 10 to 1. This compression ratio is reported by **Capture** jobs and can be used to accurately determine the amount of bandwidth used. The ratio is shown in the HVR GUI by clicking on **Statistics**.

⚠ To measure HVR's network bandwidth usage, we recommend using command **netstat -i**, which runs on Linux, Unix, and Windows.

Refresh and row-by-row compare may use a lot of bandwidth in a short period of time, because a lot of data is transferred between systems. If the HVR protocol is used, then network communication is optimized relative to using database connectivity across the network.

HVR will always try to transport the data as quickly as possible and may be using all of the available network bandwidth. Action **LocationProperties** with options **/ThrottleKbytes** and **/ThrottleMillisecs** can be used to prevent HVR from using all available network resources if the available bandwidth and one point in time would not be sufficient to keep up with the transaction volume.

Server Requirements

Contents

This section describes the requirements for the HVR Hub server machine, as well as the servers running HVR remote agent on the source and target locations.

Hub Server

The HVR hub is an installation of HVR on a server machine (hub server). The HVR hub orchestrates replication in logical entities called **Channels**. The hub runs a **Scheduler** service to manage jobs that move data flow between source location(s) and target location(s) (**Capture** jobs, **Integrate** jobs, **Refresh** jobs, **Compare** jobs).

For the list of databases that HVR supports as a hub database, see section Hub Database in Capabilities.

In order to operate, the **HVR Scheduler** must connect to a repository database consisting of a number of relational tables. By design HVR hub processing is limited to job orchestration, recording system state information and temporary storage of router files and transaction files. For the **Refresh** process, no data is persisted on the HVR hub so the hub acts as a simple pass through. Therefore, the HVR hub needs storage to hold the following:

1. HVR state information (**$HVR_CONFIG**)
2. Repository database (including statistics retention, location information, etc)
3. HVR installation (<100 MB standard)

Resource Consumption

HVR is designed to distribute work between HVR remote agents. As a result, resource-intensive processing rests on the HVR remote agents, with the HVR hub machine performing as little processing as possible. The HVR hub machine controls all the jobs that move data between sources and targets, and stores the system's state to enable recovery without any loss of changes. All data transferred between sources and targets pass through the HVR hub machine, including data from a one-time load (**hvrrefresh**) and detailed row-wise comparison (**hvrcompare**).

The HVR hub machine needs resources to:

- Run the **HVR Scheduler** .
- Spawn jobs to perform one-time data movement (**Compare** and **Refresh**) and continuous replication (**Capture** and **Integrate**). In all cases, the resource-intensive part of data processing is implemented on an HVR remote agent machine, including data compression, with the HVR hub machine simply passing the data from source to target. For data refresh or compare, the data is simply transferred skipping the disk. During normal capture activity, data is temporarily stored on the disk to allow the quickest possible recovery, with capture(s) and integrate (s) running asynchronously for optimal efficiency. If the data transfer is encrypted, the HVR hub machine decrypts the data and encrypts it again (typically using different encryption certificates) as needed to deliver it to the target.
- Transfer compressed data from source to target. Since the amount of data transferred is reduced by 5-10 times, large amounts of data can be transferred without the need for very high network bandwidth.

- Collect metrics from the log files to be stored in the repository database.
- Provide real-time process metrics to any Graphical User Interfaces (GUIs) connected to the HVR hub machine. HVR runs as a service, regardless of whether any GUI is connected, and real-time metrics are provided for monitoring purposes.
- Allow configuration changes in the design environment.

CPU: every HVR job spawns a process – i.e. one for every **Capture**, one for every **Integrate**. The CPU utilization for each of these processes on the hub machine is generally very low unless some heavy transformations are processed on the hub machine (i.e. depending on the channel design). Besides, **Refresh** or **Compare** may spawn multiple processes when running. A lot of CPU can be used when performing a row-by-row refresh/compare.

Memory: memory consumption is slightly higher on the hub machine than on the source, but still fairly modest. Some customers run dozens of channels on a dedicated hub machine with a fairly modest configuration. Row-by-row refresh and compare may use a lot of memory but are not run on an ongoing basis.

Storage space: storage utilization on the hub machine can be high. If **Capture** is running but **Integrate** is not into at least one destination, then HVR will accumulate transaction files on the hub machine. These files are compressed, but depending on the activity on the source database and the amount of time it takes until the target starts processing transactions, a lot of storage space may be used. Start with at least 10 GB, but possibly more if the hub machine manages multiple channels and network connectivity is unreliable. Large row-by-row refresh or compare can also use a lot of storage space.

I/O: if HVR is running **Capture** and keeping up with the transaction log generation on a busy system that processes many small transactions, then transaction files will be created at a rapid pace. Make sure that the file system can handle frequent I/O operations. Typically, a storage system cache or file system cache or SSD (or a combination of these) can take care of this.

Sizing Guidelines for Hub Server

The most important factor impacting the HVR hub size is whether the hub also performs the role of a source and/or a target HVR agent. General recommendations include:

- Co-locate the HVR hub with a production source database only if the server(s) hosting the production database has (have) sufficient available resources (CPU, memory, storage space, and I/O capacity) to support the HVR hub for your setup.
- **HVR Capture** may run against a physical standby of the source database with no direct impact on the source production database. In this case, consider CPU utilization of the capture process(es) running on the source database. For the Oracle RAC production database, there is one log parser per node in the source database, irrespective of the standby database configuration.
- Sorting data to coalesce changes for burst mode and to perform row-wise data compare (also part of the row-wise refresh) are CPU, memory and (temporary) storage space intensive.
- Utilities to populate the database target like TPT (for Teradata) and gpfdist (for Greenplum) can be very resource-intensive.

Storage for HVR_CONFIG

The most important resource for the HVR hub machine to function well is fast I/O operations (in terms of IOPS), especially for the **$HVR_CONFIG** directory, where runtime data and state are written to. To support capture on a busy source system, transaction files can be written to the disk every second or two, with updates to the (tiny) capture state file at the same rate, as well as very frequent updates to the log files that keep track of the activity. With multiple channels running, there will be many small I/O operations into the **$HVR_CONFIG** directory every second. The disk subsystem with a sizable cache and preferably Solid-State Drives (SSDs) is a good choice for HVR hub storage.

Hub Database

The HVR hub stores channels metadata, a very limited amount of runtime data, as well as aggregated process metrics (statistics) in its repository database. The most important resource for the hub database is storage, with even quite modest needs in order to support a single hub (up to 20 GB of disk space allocated for the repository database can

support virtually all hub setups). Traditionally, repository database are stored locally to the HVR hub, but there are cases when a database service is used to host the repository database away from the HVR hub. The main advantage of a local repository database is a lower likelihood that the database connection fails (resulting in all data flows to stop because the **Scheduler** fails in such a case) versus offloading any resources the repository requires with a database elsewhere.

The statistics stored in the repository database (**hvr_stats**) can take up a large amount of storage space.

Sizing Guidelines for Hub Database

Review the guidelines and decide based on your situation what is the best HVR hub configuration. For example:

- Your HVR hub may capture changes for one of multiple sources, using HVR remote agents for the other sources.
- One of your sources may be a heavily-loaded 8-node Oracle Exadata database that requires far more resources to perform CDC than a single mid-size SQL Server database.
- You may plan to run very frequent (resource-intensive) CDC jobs, etc.

The change rate mentioned in the sizing guideline below is the volume of transaction log changes produced by the database (irrespective of whether or not HVR captures all table changes from the source or only a subset).

Hub Size	Resources	Standalone Hub	With Capture, no Integrate	With Integrate, no Capture	With Capture and Integrate
Small	CPU cores: 4-8 Memory: 16-32 GB Disk: 50-500 GB SSD Network: 10GigE HBA (or equivalent)	5 channels with average change rate up to 20 GB /hour	2 channels with average change rate up to 20 GB /hour	2 channels with average change rate up to 20 GB /hour	1 channel processing up to 20 GB/hour
Medium	CPU cores: 8-32 Memory: 32-128 GB Disk: 300 GB - 1 TB SSD Network: 2x10 GigE HBA	20 channels, up to 5 with high average change rate of 100 GB/hour	8 channels, up to 2 with high average change rate of 100 GB/hour	6 channels, up to 2 with high average change rate of 100 GB/hour	4 channels, up to 2 with high average change rate of 100 GB/hour
Large	CPU cores: 32+ Memory: 128 GB+ Disk: 1 TB+ SSD Network: 4+ x10 GigE HBA	50+channels	15+ channels	12+ channels	8+ channels

Monitoring Disk Space on HVR Hub

Even though the HVR hub uses limited storage on the HVR hub, a shortage of free disk space can significantly impact the repository database performance and therefore the HVR performance. Standard database monitoring tools can be employed to verify the amount of disk space left on the HVR hub server – considering the type of repository database that has been installed. Since every database has unique requirements in terms of optimum storage required for operational health, it is important to set these alerting thresholds accordingly. These are to be used as guidelines only and not as reference architectures. In most cases, the disk alerts must be set for 80%, 85% and 90% capacity. Any higher than 90% is considered as a production support call to immediately add disk or free up storage. Standard database monitoring solutions can be helpful to monitor the disk usage from the hub database perspective.

Source Location Server

The HVR remote agent machine on the capture location needs resources to perform the following functions:

- For one-time data loads (refresh) and row-wise compare, HVR remote agent machine retrieves data from a source database, compresses it, optionally encrypts it and sends to the HVR hub. For optimum efficiency, data is not written to the disk during such operations. Matching source database session(s) may use a fair amount of database (and system) resources. Resource consumption for **Refresh** and **Compare** is only intermittent.
- For bulk compare jobs, the HVR remote agent machine computes a checksum for all the data.
- To set up CDC during **Initialize**, HVR remote agent machine retrieves metadata from the database and adds table-level supplemental logging as needed.
- During CDC, resources are needed to read the logs, parse them, and store information about in-flight transactions in memory (until a threshold is reached and additional change data is written to disk). The amount of resources required for this task varies from one system to another, depending on numerous factors, including:
- the log read method (direct or through an SQL interface),
- data storage for the logs (on disk or in, for example, Oracle Automatic Storage Manager),
- whether the system is clustered or not,
- the number of tables in the replication and data types for columns in these tables, and
- the transaction mix (ratio of insert versus updates versus deletes, and whether there are many small, short-running transactions versus larger, longer-running transactions).

Log parsing is generally the most CPU-intensive operation that can use up to 100% of a single CPU core when capture is running behind. HVR uses one log parser per database thread, and every database node in an Oracle cluster constitutes one thread.

For a real-world workload with the HVR agent running on the source database server, it is extremely rare to see more than 10% of total system resource utilization going to the HVR remote agent machine during CDC, with typical resource consumption well below 5% of system resources.

For an Oracle source database, HVR will periodically write the memory state to disk to limit the need to re-read archived log files to capture long-running transactions. Consider storage utilization for this if the system often processes large, long-running transactions.

Resource Consumption

- **CPU:** every channel will use up to 1 CPU core in the system. If HVR runs behind and there is no bottleneck accessing the transaction logs or using memory, then HVR can use up to the full CPU core per channel. In a running system with HVR reading the tail end of the log, the CPU consumption per channel is typically much lower than 100% of the CPU core. Most of the CPU is used to compress transaction files. Compression can be disabled to lower CPU utilization. However, this will increase network utilization (between source HVR remote agent machine and the HVR hub and between the HVR hub and any target HVR remote agent machine). **Refresh** and **Compare** operations that are not run on an ongoing basis will add as many processes as the number of tables refreshed/compared in parallel. In general, the HVR process uses relatively few resources, but the associated database job uses a lot of resources to retrieve data (if parallel select operations run against a database, then the **Refresh** or **Compare** operations can use up to 100% of the CPU on the source database).
- **Memory:** memory consumption is up to 64 MB per transaction per channel. Generally, 64 MB per transaction is not reached and much less memory is used but this depends on the size of the transactions and what portion of it

is against tables that are part of a channel. Note that the 64 MB threshold can be adjusted (upwards and downwards).

- **Storage space:** the HVR installation is about 100 MB in size and while running **Capture**, it uses no additional disk space until the 64 MB memory threshold is exceeded and HVR starts spilling transactions to disk. HVR will write compressed files but in rare cases, with large batch jobs modifying tables in the channel that only commit at the end. HVR may be writing a fair amount of data to disk starting with at least 5 GB for **$HVR_CONFIG**. Please note that **HVR Compare** may also spill to disk which would also go into this area. If one aggressively backs up the transaction logs so that they become unavailable to the source database, then you may consider **hvrlogrelease** to take copies of the transaction logs until HVR does not need them anymore. This can add a lot of storage space to the requirements depending on the log generation volume of the database and how long transactions may run (whether they are idle or active does not make a difference for this).
- **I/O**: every channel will perform frequent I/O operations to the transaction logs. If HVR is current, then each of these I/O operations is on the tail end of the log, which could be a source of contention in older systems (especially if there are many channels). Modern systems have a file system or storage cache, and frequent I/O operations should barely be noticeable.

Target Location Server

The HVR remote agent machine on the integrate location needs resources to perform the following functions:

- Apply data to the target system, both during a one-time load (refresh) and during continuous integration. The resource utilization for this task varies a lot from one system to another, mainly depending whether changes are applied in so-called burst mode or using continuous integration. The burst mode requires HVR to perform a single net change per row per cycle so that a single batch insert, update or delete results in the correct end state for the row. For example, when, in a single cycle, a row is first inserted followed by two updates then the net operation is an insert with the two updates merged with the initial data from the insert. This so-called coalesce process is both CPU and (even more so) memory intensive, with HVR spilling data to disk if memory thresholds are exceeded.
- Some MPP databases like Teradata and Greenplum use a resource-intensive client utility (TPT and gpfdist respectively) to distribute the data directly to the nodes for maximum load performance. Though resource consumption for these utilities is not directly attributed to the HVR remote agent machine, you must consider the extra load when sizing the configuration.
- For data compare, the HVR integrate agent machine retrieves the data from a target system to either compute a checksum (bulk compare) or to perform a row-wise comparison. Depending on the technologies involved, HVR may, in order to perform the row-wise comparison, sort the data, which is memory intensive and will likely spill significant amounts of data to disk (up to the total data set size).
- Depending on the replication setup, the HVR integrate agent machine may perform extra tasks like decoding SAP cluster and pool tables using the SAP Transform or encrypt data using client-side AWS KMS encryption.

With multiple sources sending data to a target, a lot of data has to be delivered by a single HVR integrate agent machine. Load balancers (both physical and software-based like AWS's Elastic Load Balancer (ELB)) can be used to manage integration performance from many sources into a single target by scaling out the HVR integrate agent machines.

Resource Consumption

- **CPU:** on the target, HVR typically does not use a lot of CPU resources, but the database session it initiates does (also depends on whether any transformations are run as part of the channel definition). A single **Integrate** process will have a single database process that can easily use the full CPU core. Multiple channels into the same target will each add one process (unless specifically configured to split into more than one **Integrate** process). **Compare**/**Refresh** can use more cores depending on the parallelism in HVR. Associated database processes may use more than one core each depending on parallelism settings at the database level.

- **Memory:** the memory consumption for HVR on the target is very modest unless large transactions have to be processed. Typically, less than 1 GB per **Integrate** is used. Row-by-row refresh and compare can use gigabytes of memory but are not run on an ongoing basis.

- **Storage space: $HVR_CONFIG** on the target may be storing temporary files for row-by-row compare or refresh, and if tables are large, a significant amount of space may be required. Start with 5 GB.

- **I/O:** the I/O performance for HVR on the target is generally not critical.

Monitoring Integrate Agent Machine Resources

Because replication between heterogeneous source and target heavily depends on the available computing on the integrate agent machine for data type conversions during refresh and CDC, coalescing operations in a burst cycle, computing checksum for compares, in cases of HVR utilizing SAP Xform for declustering and depooling tables, decryption of data received from the hub, and the like, its imperative to determine scale-out integrate agent strategies pre-deployment.

See Also

- Scaling and Monitoring HVR Hub and Agent Resources on AWS

Supported Operating Systems

Contents

This section lists the operating system platforms that HVR supports.

Supported Windows Platforms

HVR can be installed on any of the following Windows platforms:

- Windows (64-bit): 8, 10
- Windows Server (64-bit): 2008, 2012 R2, 2016, 2019

- Windows (32-bit): 8, 10

For information about the HVR platforms and versions that support these operating systems, see **Platform Compatibility Matrix**.

Supported Linux Platforms

HVR can be installed on any of the following Linux platforms:

- Linux (x86-64 bit) based on GLIBC 2.5 and higher. This includes:
  - Red Hat Enterprise Linux Server: 5.x, 6.x, 7.x
  - SUSE Linux Enterprise Server: 10.x, 11.x, 12.x, 15.x
  - Machines imaged from Amazon Linux AMI or Amazon Linux 2
- Linux (x86-64 bit) based on GLIBC 2.4 and higher. This includes:
  - SUSE Linux Enterprise Server: 10.x, 11.x, 12.x
- Linux (x86-32 bit) based on GLIBC 2.3.4 and higher. This includes:
  - Red Hat Enterprise Linux ES: 3.x, 4.x, 5.x, 6.x
  - SUSE Linux Enterprise Server: 9.0,10.x
- Linux (ppc64) based on GLIBC 2.27 and higher.

For information about the HVR platforms and versions that support these operating systems, see **Platform Compatibility Matrix**.

Supported Unix Platforms

HVR can be installed on any of the following Unix platforms:

- AIX (64-bit): 6.1, 7.1, 7.2
- Solaris for SPARC (64-bit): 9, 10, 11.x
- Solaris for Intel/AMD (64-bit): 10.x, 11.x
- HP-UX 11i (64-bit): v1.5, v1.6, v2, v3

- AIX (32-bit): 6.1, 7.1, 7.2
- Solaris for SPARC (32-bit): 9, 10, 11.x

For information about the HVR platforms and versions that support these operating systems, see **Platform Compatibility Matrix**.

Supported macOS Platforms

HVR can be installed on the following macOS platform:

- macOS Sierra: 10.12.x

⚠ The HVR installation on macOS can only be used as HVR GUI to connect to remote hubs or for file replication. HVR does not support hub, capture or integrate databases on macOS.

For information about the HVR platforms and versions that support these operating systems, see **Platform Compatibility Matrix**.

Installing HVR on AWS

This section provides information on the requirements for HVR on AWS and the steps for installing HVR on AWS.

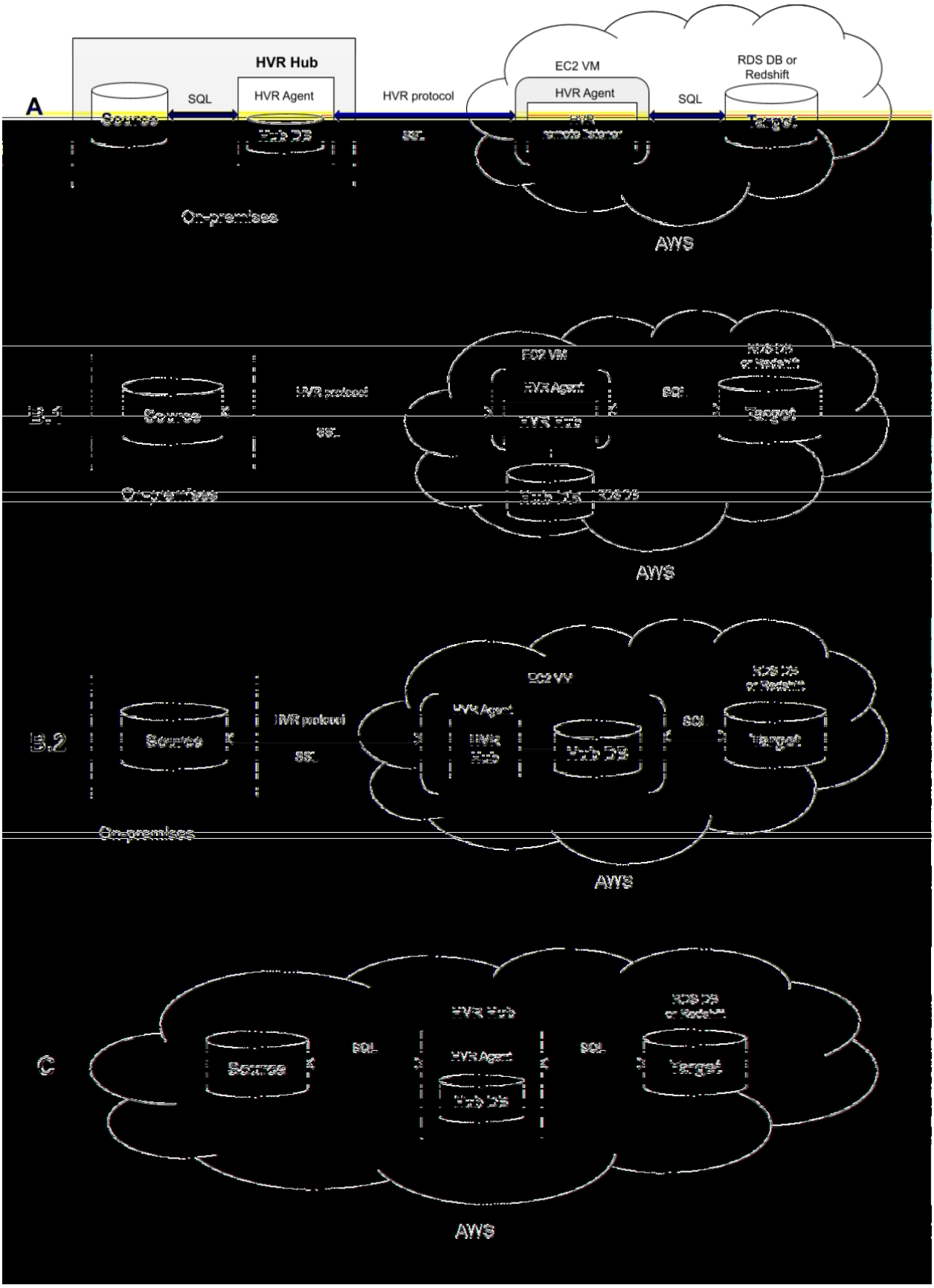
Requirements for AWS

AWS (Amazon Web Services) is Amazon's cloud platform providing the following services relevant for HVR:

- EC2 Elastic Cloud Computing instances are Virtual Machines in the AWS cloud. These VMs can be either Linux or Windows-based. This is "Infrastructure as a Service" (IaaS). HVR can run on an EC2 Instance provided the OS is supported by HVR (Linux, Windows server). This scenario is identical to running HVR in a data center for an on-premises scenario.
- Amazon Redshift is Amazon's highly scalable clustered data warehouse service. HVR supports Redshift as a target database, both for initial load/refresh and in Change Data Capture mode. For more information, see Requirements for Redshift.
- Amazon RDS is Amazon's Relational Database Service. HVR supports MariaDB, MySQL, Aurora, Oracle, PostgreSQL, and Microsoft SQL Server running on Amazon RDS. Note that log-based capture is not supported for Microsoft SQL Server on Amazon RDS.
- Amazon EMR (Elastic Map Reduce) is Amazon's implementation of Hadoop. It can be accessed by using HVR's generic Hadoop connector. For more information, see Requirements for HDFS.
- Amazon S3 storage buckets are available as staging area to load data into Redshift, can be used as a file location target (optional with Hive external tables on top), or for staging for other databases (Hive Acid, Snowflake).

Architecture

There are different types of configuration topologies supported by HVR when working with AWS. The following ones are most commonly used:



- **A**: Connecting to an AWS resource with the HVR hub installed on-premises. To avoid poor performance due to low bandwidth and/or high latency on the network, the HVR Agent should be installed in AWS. Any size instance will be sufficient for such use case, including the smallest type available (**T2.Micro**).
- **B.1**: Hosting the HVR hub in AWS to pull data from an on-premises source into AWS. For this use case, the hub database can be a separate RDS database supported as a hub by HVR. The HVR Agent may be installed on an AWS EC2 instance and be configured to connect to the hub database. For this topology (**B.1**), using the HVR

Agent on EC2 is optional. However, it may provide a better performance, as opposed to remotely connecting the HVR to RDS over the Internet. If the HVR Agent is used on EC2 to connect to RDS, then communication with the HVR hub over the HVR protocol is fast and is not affected by network latency that much.

- **B.2**: Alternatively the hub database can be installed on the EC2 VM.
- **C**: Performing cloud-based real-time integration. HVR can connect to only cloud-based resources, either from AWS or from other cloud providers.

AMI Agent Configuration Notes

HVR provides a special packaged edition **HVR Image for AWS**: a pre-configured Linux AMI (Amazon Machine Image) containing HVR's remote listener agent including Redshift and Oracle drivers. For more information, see Installing HVR on AWS using HVR Image.
Alternatively, an Agent or Hub can be set up by doing a manual installation as described in Installing HVR on UNIX or Linux, with the following notes:

- An instance **t2.micro** is sufficient to run HVR as an agent. HVR running as a hub requires at least instance type **T2.medium** for more memory.
- Open the firewall to allow remote TCP/IP HVR connections to the HVR installation in AWS (e.g. topology A), by default on port 4343. Restrict the port access to the originator's public IP address. If the instance has to connect to an on-premises installation of HVR (topology B), then (a) add the HVR TCP/IP protocol to the on-premises firewall and DMZ port forwarding to be able to connect from AWS to on-premises, or (b) configure a VPN.
- When HVR is running as a hub, it needs temporary storage to store replication data. Add this when creating the instance. 10 GB is normally sufficient.
- Install the appropriate database drivers (e.g. Redshift, Oracle, SQL Server). For Redshift, follow the instructions in Requirements for Redshift. Download the Oracle Client installation on Oracle's website.
- The hub database can be an RDS database service or a local installation of a supported database in the VM. Install the appropriate database drivers in the HVR hub instance to connect to the database.
- An HVR HUB machine running in an AWS Linux instance can be remotely managed by a Windows PC registering the remote hub.
- File replication is supported in AWS.
- By default, network traffic is not encrypted. For production purposes we strongly advise to use SSL encryption to securely transport data over public infrastructure to the cloud. For more information, see Encrypted Network Connection.

Installing HVR on AWS using HVR Image

HVR Image for AWS is a pre-configured Linux-based AMI (Amazon Machine Image) to run HVR's remote listener (as a daemon) including Redshift drivers. HVR Image for AWS supports the use as both a remote HVR agent (for capture and /or integration) as well use as a hub machine using either a PostgreSQL database on the machine, an RDS database, or a database running on a different machine.

HVR Image for AWS is available from the AWS Marketplace and includes all necessary components to connect to Amazon Redshift, Oracle and PostgreSQL on RDS, S3 or any HVR supported target database on EC2, enabling replication from/into all supported on-premises platforms.

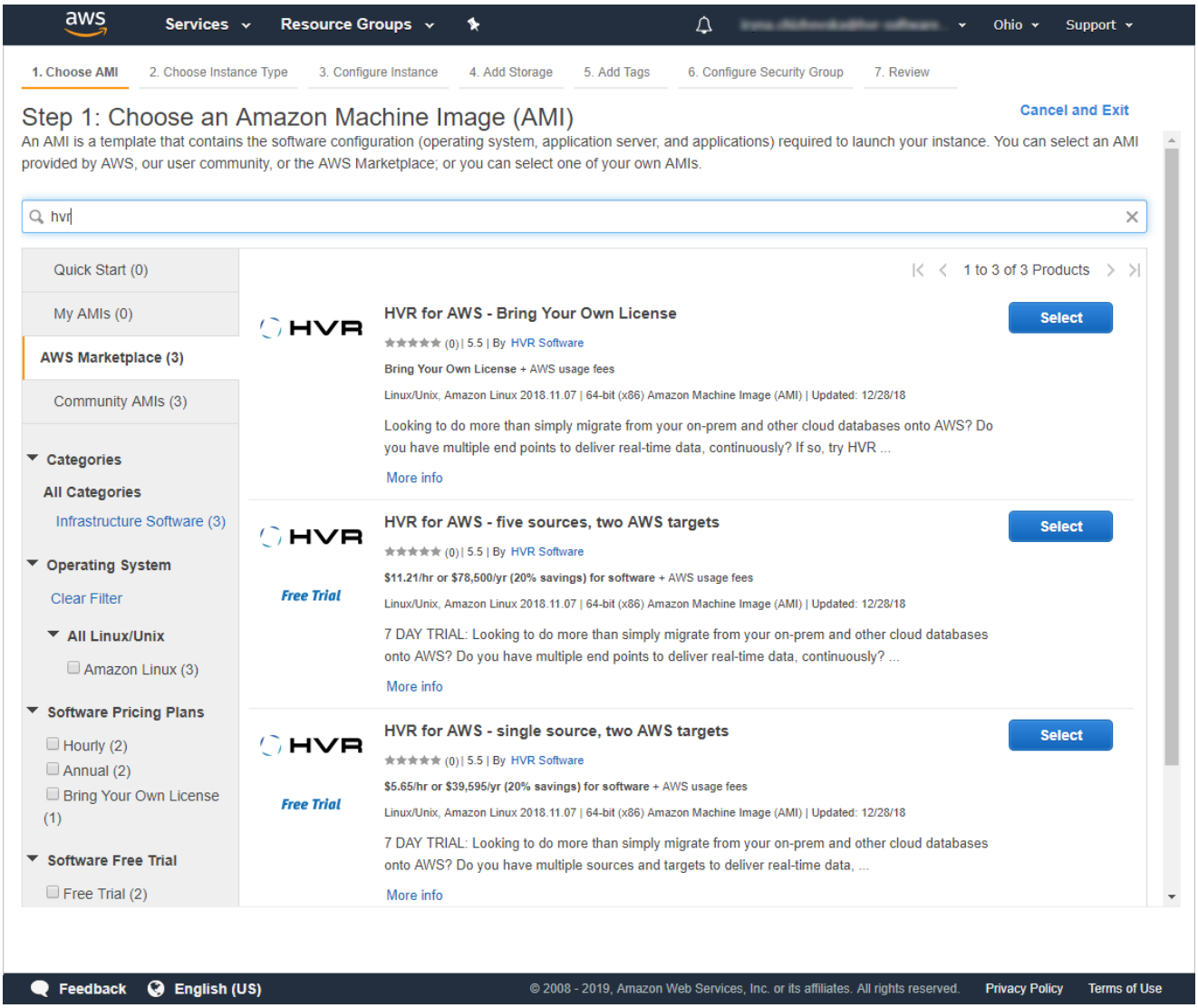
⚠ HVR Image for AWS is currently available only on Linux. Connectivity to SQL Server requires an HVR installation on Windows (optionally running as an agent).

Using HVR Image for AWS to create a remote listener agent

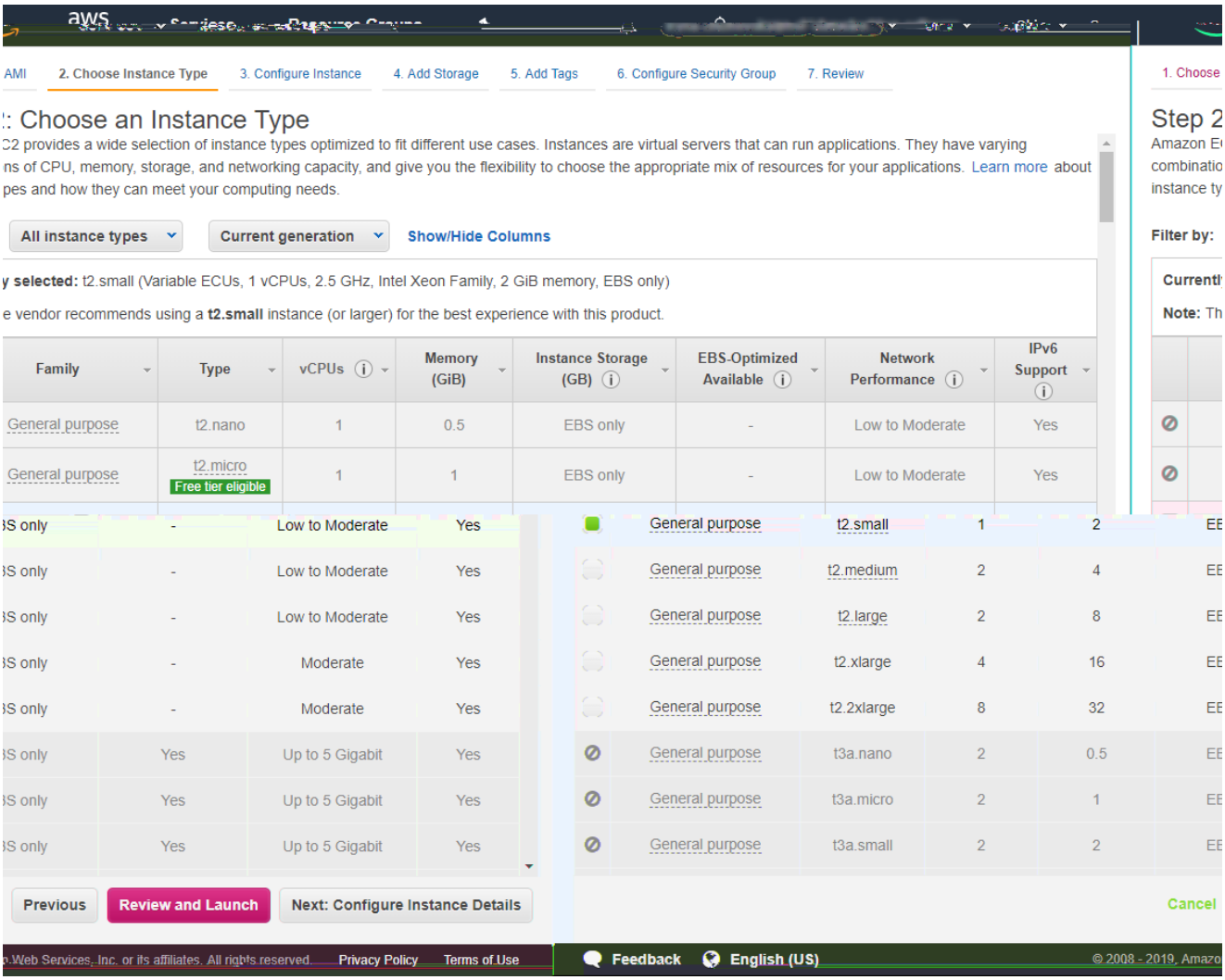
To use the HVR Image for AWS:

1. Sign in to the AWS portal and select **EC2** under the **Services** menu.
2. Click the **Launch Instance** button.
3. On the right side menu bar, click **AWS Marketplace** and type 'HVR' in the search field.
4. Select the HVR for AWS offering suitable for you.

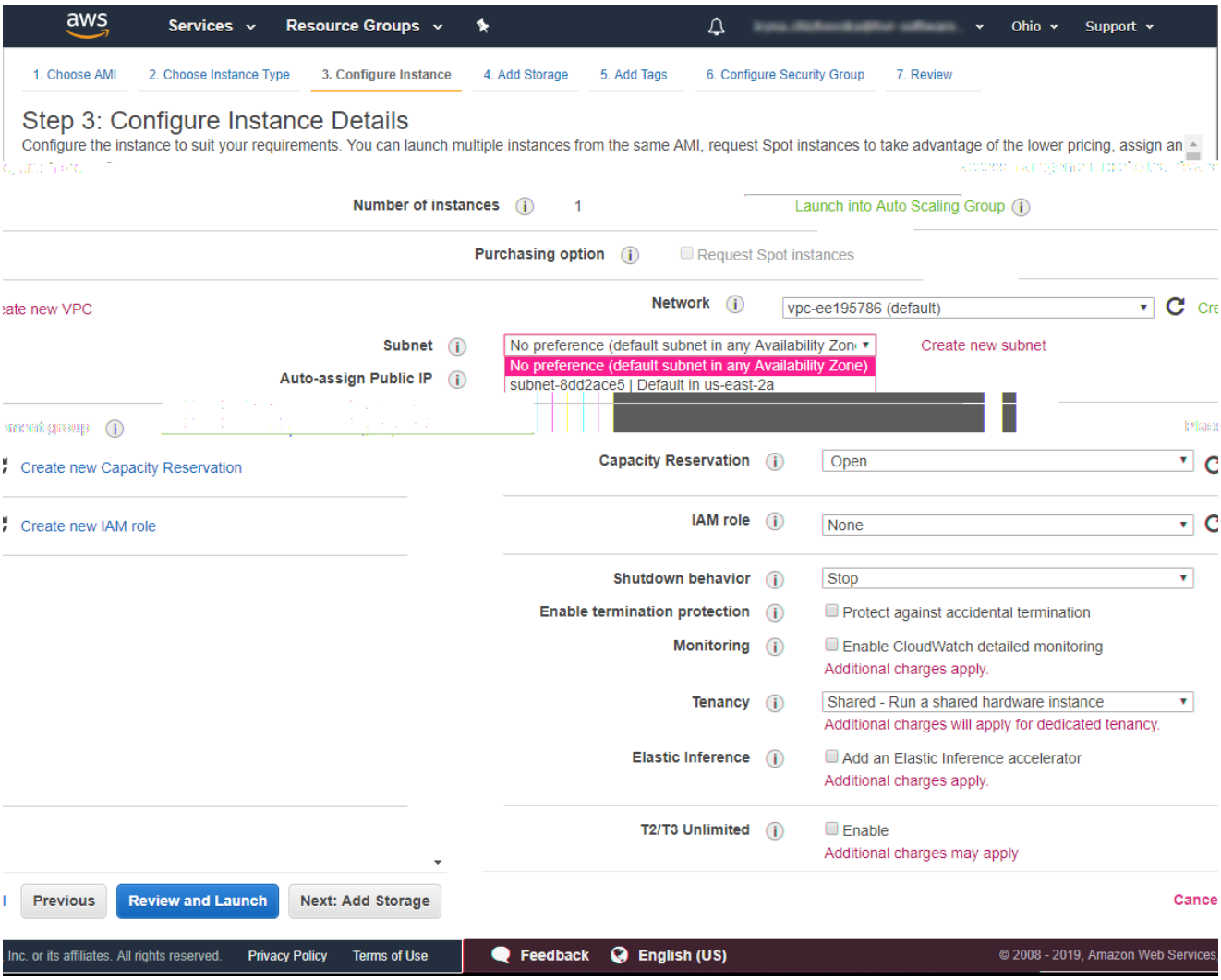
⚠ For optimum efficiency make sure HVR in AMI runs in the same region as the database or service taking part in real-time data movement.



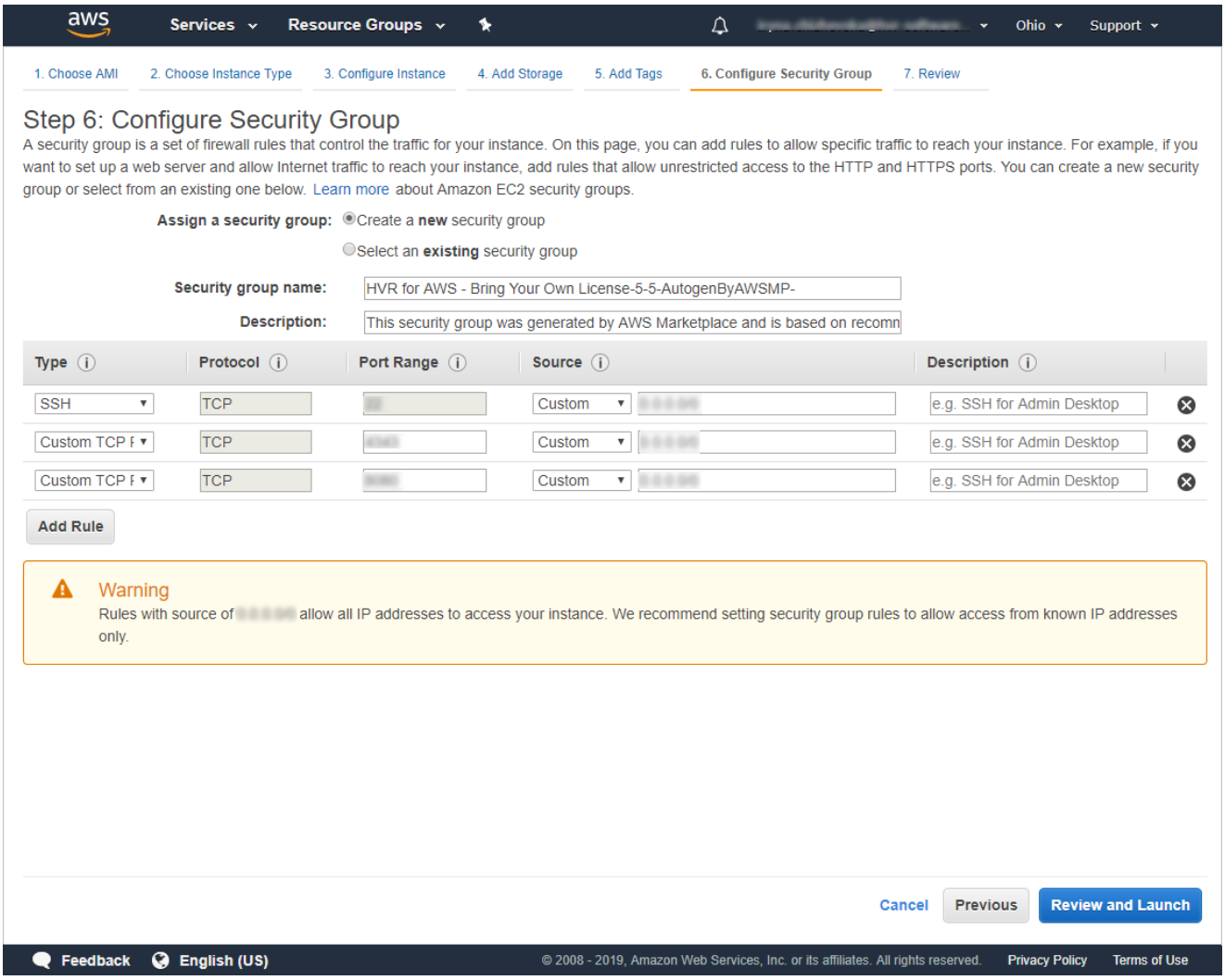
5. HVR will show you the product details, click **Continue**.
6. Under the **Choose an Instance Type** tab, select a required one and click **Next: Configure Instance Details**.



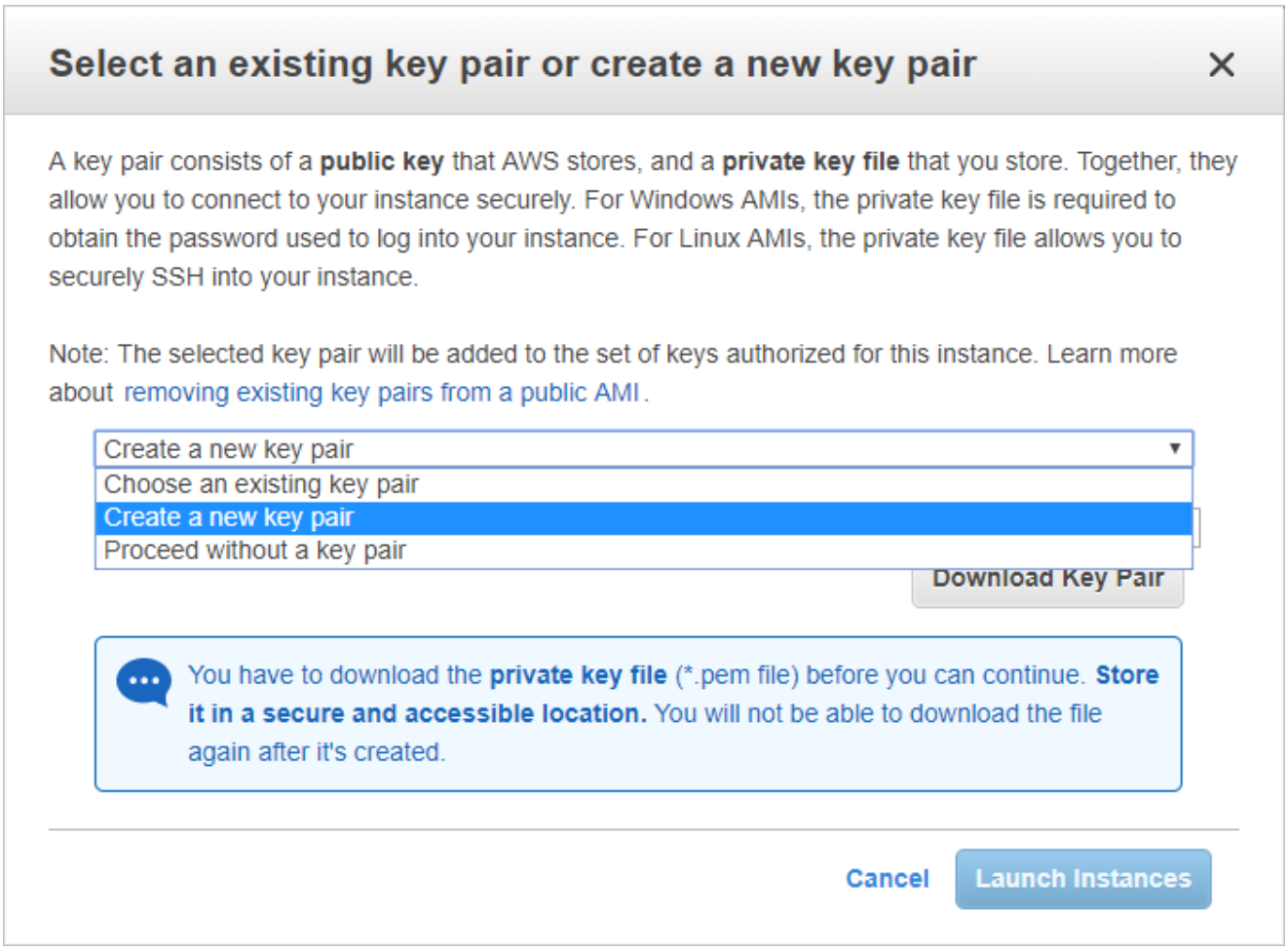
7. If appropriate, under the **Configure Instance** tab, select a preferred network (VPC) subnet or use the default one.



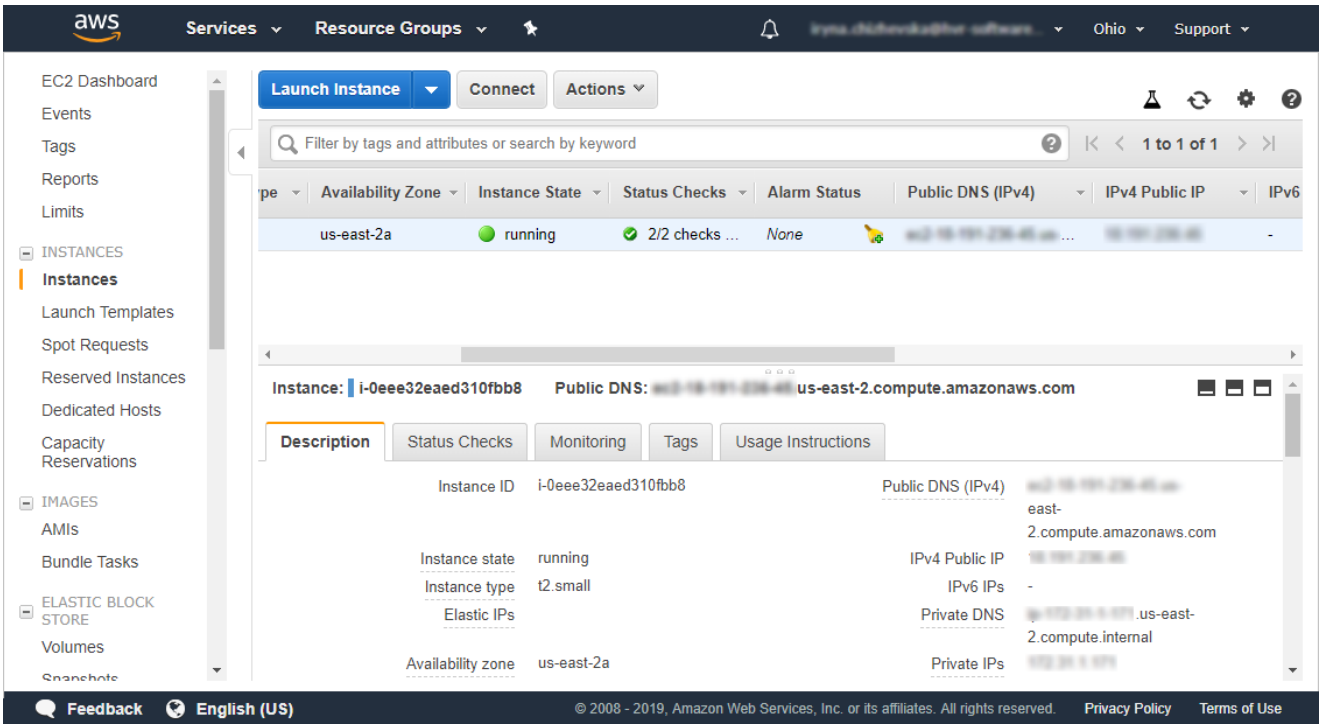
8. Under the **Add Storage** and **Add Tags** tabs, proceed with default settings.
9. Under the **Configure Security Group** tab, set up traffic rules for your instance. HVR uses TPC/IP connectivity on port 4343. Configure an incoming connection for the HVR listener daemon on this port and limit the IP range as narrow as possible. Click **Review and Launch**.



10. Under the **Review** tab, review your instance launch details and click **Launch** to assign a key pair to your instance and complete the launch process. An EC2 key pair will be used to securely access the AMI. If you have an existing EC2 key pair defined, you can use that. If you don't have a key pair defined, you can create one. Once created, the associated private key file will be downloaded through your web browser to your local computer.



11. You will now see the AMI being deployed in the Marketplace. This may take a few minutes. The details of the created AMI can be accessed in the AWS EC2 console. Note down the Public DNS name or IP address of the AMI as this will be required when connecting to the AMI.



12. Once the AMI has been deployed and is running you can start an ssh session to the AMI instance to obtain information on how to proceed. To connect, use the **pem** key that was used when the image was created. Connect as a user named **ec2-user**:

Installing HVR on AWS Manually

Following are the steps to create HVR instance for AWS:

1. In the AWS portal, create a VM in the AWS console **AWS -> EC2 --> Launch Instance** type **T2.micro – Red Hat 64bit** for the agent. In the advanced configuration, have it created in the same **VPC** as your Redshift cluster(Step 3) and create /use a security group allowing connections on port e.g. **4343** from your on-premise environment (Step 6) (let Amazon auto detect your IP range).
2. Then install the HVR software on that VM as an agent by following the installation steps 1,2 and 5 in section Installing HVR on Unix or Linux on the same port (eg **4343**) as you just opened in the secuity group.
3. Also in the VM, install the Redshift ODBC driver. This is actually the Postgres 8 ODBC driver for Linux. First use **yum install** to install packages **unixODBC**, **unixODBC-devel** and **postgresql-libs** automatically and then download (with **wget**) and install package **postgres-odbc v8.04** manually to overwrite the Postgres9 components with Postgres 8 components.
4. Finally, in the AWS portal, check your Redshift and EC2 instances are in the same **VPC** and in a security group allowing port e.g. **4343**, check **AWS -> VPC -> security group -> inbound rules -> 4343** or add it.

Scaling and Monitoring HVR Hub and Agent Resources on AWS

Contents

This section provides information on scaling resources available to HVR Hub and/or integrate agent on AWS and monitoring HVR Hub disk space and integrate agent resources utilization on AWS.

Scaling Resources Available to HVR Hub and/or Integrate Agent on AWS

The HVR hub requires storage for **$HVR_CONFIG**, a repository database and an HVR installation. With proper configuration of the repository database in terms of frequent purging of the **hvr_stats** tables and maintaining the **hvr_users** tables, not much variability can be expected in the HVR hub storage needs. In the case of an EC2 instance on which an HVR Agent (integrate) is running, storage utilization can be impacted by the size of the burst tables and the number of operations per burst cycle.

Like most stateless services that run on Amazon EC2 instances, scaling can be achieved using Amazon Elastic Load Balancer. This allows the HVR hub to automatically connect to a different stateless agent should the agent or server on which it runs become unavailable. In case the HVR hub or HVR agent installed on an Amazon EC2 instance shows high disk usage either through a third-party monitoring solution or using Amazon CloudWatch (see below), uninterrupted replication can be achieved through elastic scaling of EBS volumes as described here. After increasing the volume, you need to extend the volume's file system to make use of the new storage capacity. For more information, see Extending a Linux File System After Resizing a Volume or Extending a Windows File System After Resizing a Volume.

⚠ If your EBS volumes were attached to the EC2 instance, on which the HVR hub is installed, before November 3, 2016, 23:40 UTC, please note that there is no real way to achieve uninterrupted on-demand scaling. See AWS documentation corresponding to this scenario.

In HVR, the data replication location is identified by a DNS entry/IP address. When this location points to an ELB, then multiple EC2 edge nodes, or edge nodes of variable sizes, can be allocated without any change to the definitions in HVR. This provides the ability to dynamically adjust resources available on the edge nodes.

⚠ Note that the scaling is not currently automated out-of-the-box.

Because the HVR integrate agents are stateless and one agent can handle multiple connections to one or more target locations, load balancers can be used to help scale parallel processing for bulk loads and continuous data streaming. For example, if you are planning to onboard new source systems feeding a data lake in AWS, you can register new target instances to your Amazon Elastic Load Balancer. In addition, Amazon Auto Scaling Groups could be added to use new EC2 instances running an HVR agent based on CloudWatch Agent alarms detecting CPU or memory at 90% capacity.

Monitoring HVR Hub Disk Space Utilization on AWS

When the HVR hub is deployed in AWS on an EC2 instance, Amazon Elastic Block Store (Amazon EBS) sends data points to CloudWatch for several metrics. Amazon EBS General Purpose SSD (gp2), Throughput Optimized HDD (st1) , Cold HDD (sc1), and Magnetic (standard) volumes automatically send five-minute metrics to CloudWatch. Provisioned IOPS SSD (io1) volumes automatically send one-minute metrics to CloudWatch. Therefore, using monitoring services

like CloudWatch (Viewing Information about an Amazon EBS Volume) and integrating with the Amazon SNS messaging allows the users to work under optimal storage conditions.

Monitoring Integrate Agent Resources on AWS

Integration with services like Amazon CloudWatch for the EC2 instance on which the HVR agent is deployed can provide guidance to DBAs. Services like Amazon CloudWatch can monitor both on-premise and cloud-based servers in integrated monitoring modules. Refer to the list of available metrics to monitor on servers.

Since the resource usage is highly variable for the integrate agent, monitoring and analyzing patterns in CPU/memory utilization should help determine if a larger EC2 instance is required or if ELB are better choices to keep up the real-time replication needs of the AWS customers.

Installing HVR on Azure

This section provides information on the requirements for HVR on Azure and the steps for installing HVR on Azure.

Requirements for Azure

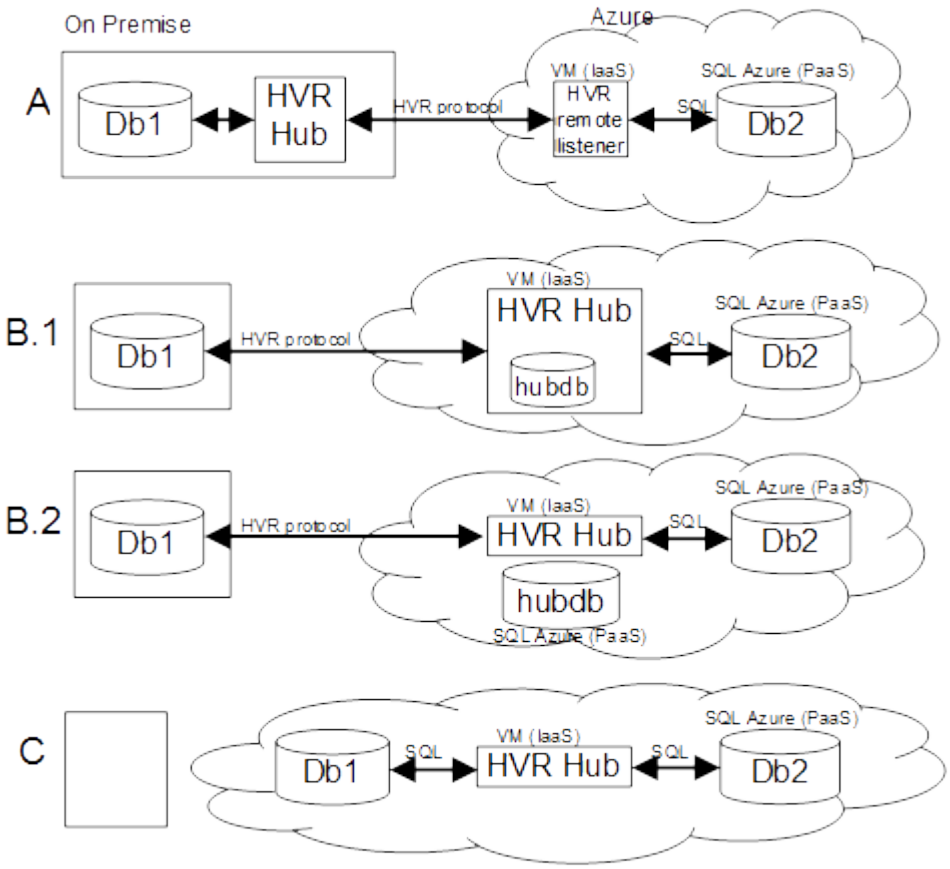
Contents

Azure is Microsoft's cloud platform providing the following components relevant for HVR:

- Virtual Machines (VMs) inside Microsoft's cloud. These VMs can be either Windows or Linux-based. This is "Infrastructure as a Service" (IaaS). HVR can run on a VM inside Azure (IaaS) provided the OS is supported by HVR (Windows server, Linux). This scenario is identical to running HVR in a data center for an on-premises scenario.
- HVR supports connecting to regular databases running in an Azure VM (like SQL Server, Oracle, DB2....).
- Azure data services supported as a source or a target for HVR. HVR supports three Azure data services:
    1. Azure SQL Database as a subset of a full SQL Server database. This is "Platform as a Service" (PaaS). HVR can connect to Azure SQL Database as a source, as a target and as hub database. For more information, see Requirements for Azure SQL Database.
    2. Azure Synapse Analytics, also PaaS. HVR can connect to Azure Synapse Analytics as a target only. For more information, see Requirements for Azure Synapse Analytics.
    3. Azure HDInsight, HDFS on Azure. For more information on Hadoop, see Requirements for HDFS.

Architecture

The following topologies can be considered when working with HVR in Azure:



- A: Connecting to an Azure resource from an on-premises HVR installation. To avoid poor performance due to low bandwidth and/or high latency on the network HVR should be installed as an agent in Azure running on a VM. Any size VM will be sufficient for such use case, including the smallest type available (an **A1 instance**).

- B: Hosting the HVR hub in Azure to pull data from on-premises systems into Azure. For this use case HVR must be installed on an Azure VM and be configured to connect to a hub database. The hub database can be a database on the hub's VM (topology B1) or it can be a SQL Azure database (topology B2).
- C: Performing cloud-based real-time integration. HVR can connect to only cloud-based resources, either from Azure or from other cloud providers.

Configuration Notes

HVR provides **HVR for Azure** image in the Azure marketplace to automatically setup and provision an HVR remote listener agent on a Windows VM in Azure (topology A). This addresses all the notes described below when setting up an agent in Azure. See Installing HVR on Azure using HVR Image for further details. **HVR for Azure** can also be used as a starting point for an Agent or Hub set up by doing a manual installation as described in Installing HVR on Windows, with the following notes:

- HVR running as a hub requires at least an **A2 instance** for more memory. Ensure to allocate sufficient storage space to store compressed transaction files if the destination system is temporarily unreachable. Depending on the transaction volume captured and the expected maximum time of disruption allocate multiple GB on a separate shared disk to store HVR's configuration location.
- A manually configured Azure VM must open the firewall to the remote listener port for hvrremotelistener.exe to allow the on-premises hub to connect (compare topology A). The **HVR for Azure** image already contains this setting.For an Azure-based hub connecting to on-premises systems (topologies B1 and B2) add the HVR port to the on-premises firewall and DMZ port forwarding.
- The HVR user must be granted **log in as a server** privileges in order to run the remotelistener service. Configure the Windows service to start automatically and to retry starting on failure to ensure the service always starts.
- Install the appropriate database drivers to connect to hub, source and/or target databases from this environment. Th **HVR for Azure** image contains SQL and Oracle drivers.
- To use the instance as a hub install perl (Strawberry Perl or ActivePerl). This is already done in the **HVR for Azure** image.
- The hub database can be an **Azure SQL** database service or an instance of any one of the other supported databases that must be separately licensed.
- Use **Remote Desktop Services** to connect to the server and manage the environment.
- File replication is supported in Azure.
- By default, network traffic is not encrypted if you install HVR yourself. For production purposes we strongly advise to use SSL encryption to securely transport data over public infrastructure to the cloud. For more information, see Encrypted Network Connection. If you use the **HVR for Azure** image from the marketplace, network traffic is encrypted.

Installing HVR on Azure using HVR Image

Contents

This section describes the steps to install an HVR Agent on Azure virtual machine using the **HVR for Azure** image. Another way is to manually install an HVR Agent on Azure virtual machine when using Azure SQL Database for replication.

In the Azure Marketplace, HVR Software provides **HVR for Azure**, an Azure virtual machine (VM) image pre-configured to be used as a HVR remote agent (for Capture or Integrate) containing the HVR remote listener service and the necessary drivers for connecting to Oracle, SQL Server, Azure SQL Database, and Azure Synapse Analytics. It can also be used as the first step for manually installing the HVR hub on Azure as described in section Installing HVR on Windows . **HVR for Azure** automatically opens the Azure firewall to allow incoming connections on port **4343** (the default port for HVR communications).

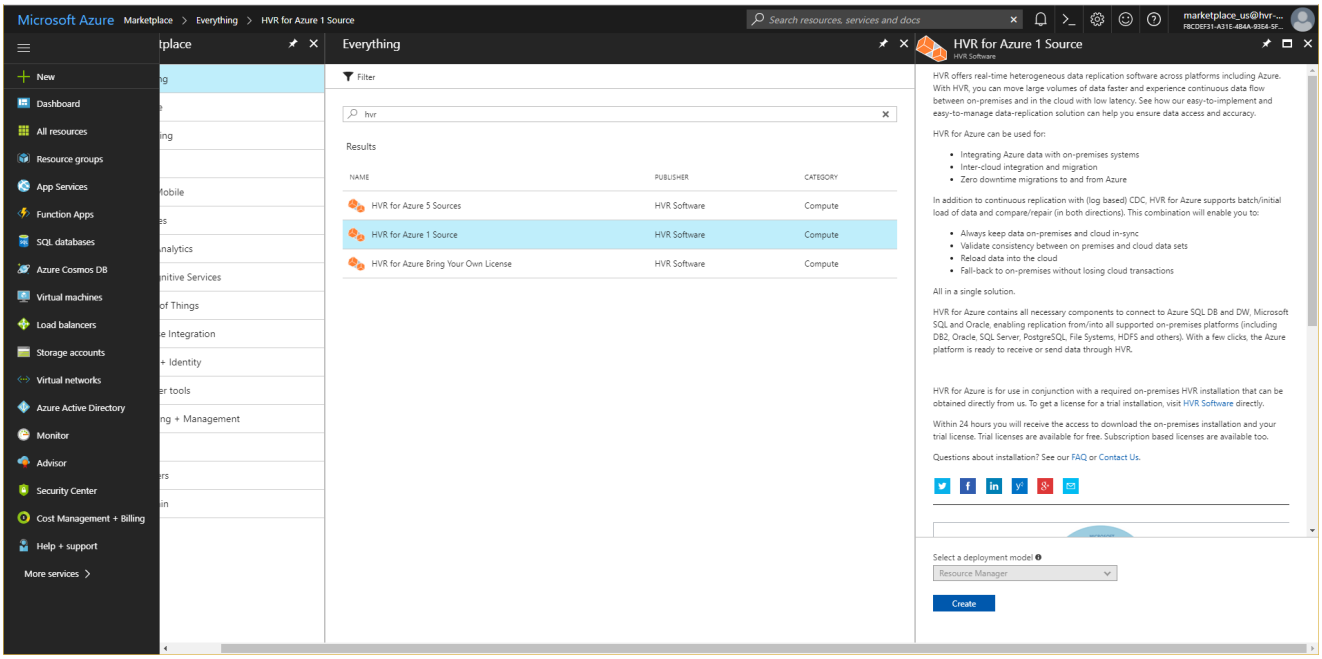
HVR for Azure is available in the BYOL (Bring Your Own License) variant without a license and various licensed variants (depending on the number of sources). The BYOL variant is intended for trial use and for customers already licensed. When **HVR for Azure** is used as an HVR remote agent, this license can be used by the HVR hub residing in a different location by setting the **/CloudLicense** parameter of the **LocationProperties** action when configuring the Azure location.

Last-minute configuration details can be found by connecting to the VM using the remote desktop connection and opening the Getting Started web link on the Desktop.

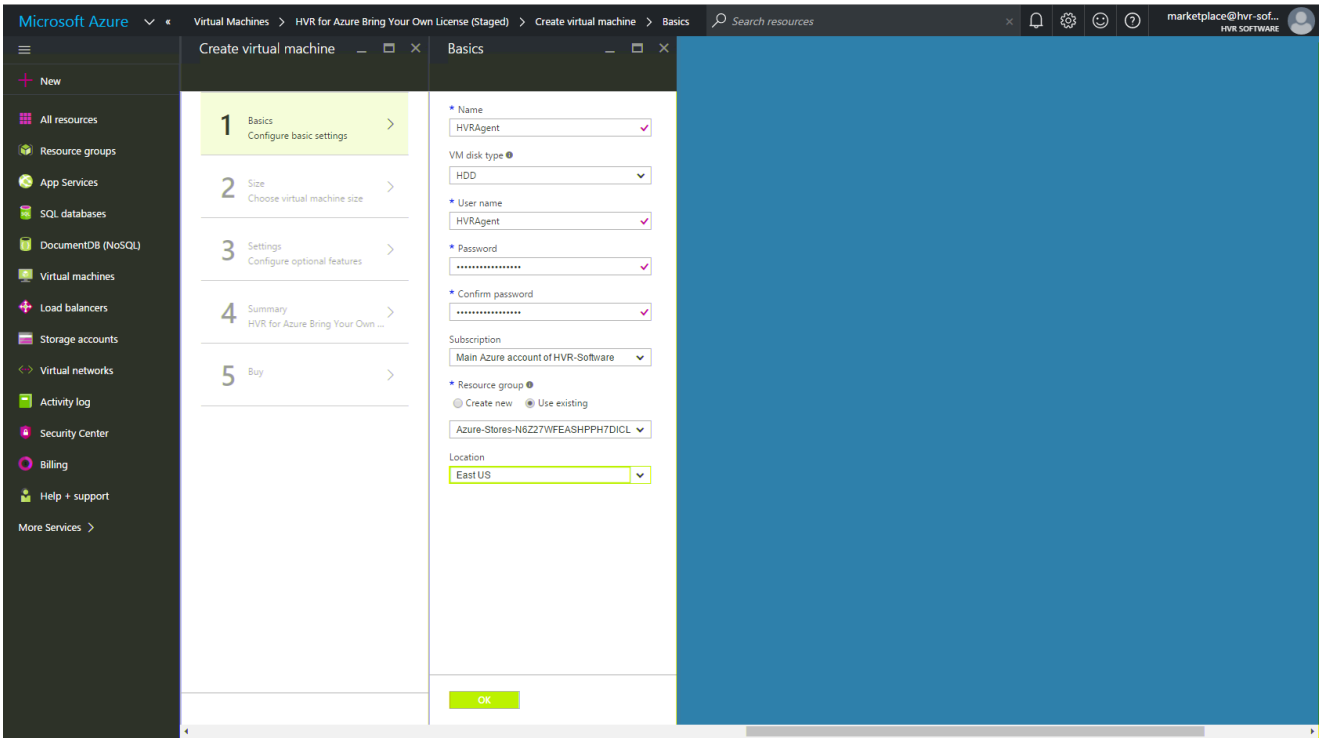
Using HVR for Azure Image to Install HVR Remote Agent

To create a new VM using **HVR for Azure** from the Azure Marketplace, perform the following steps:

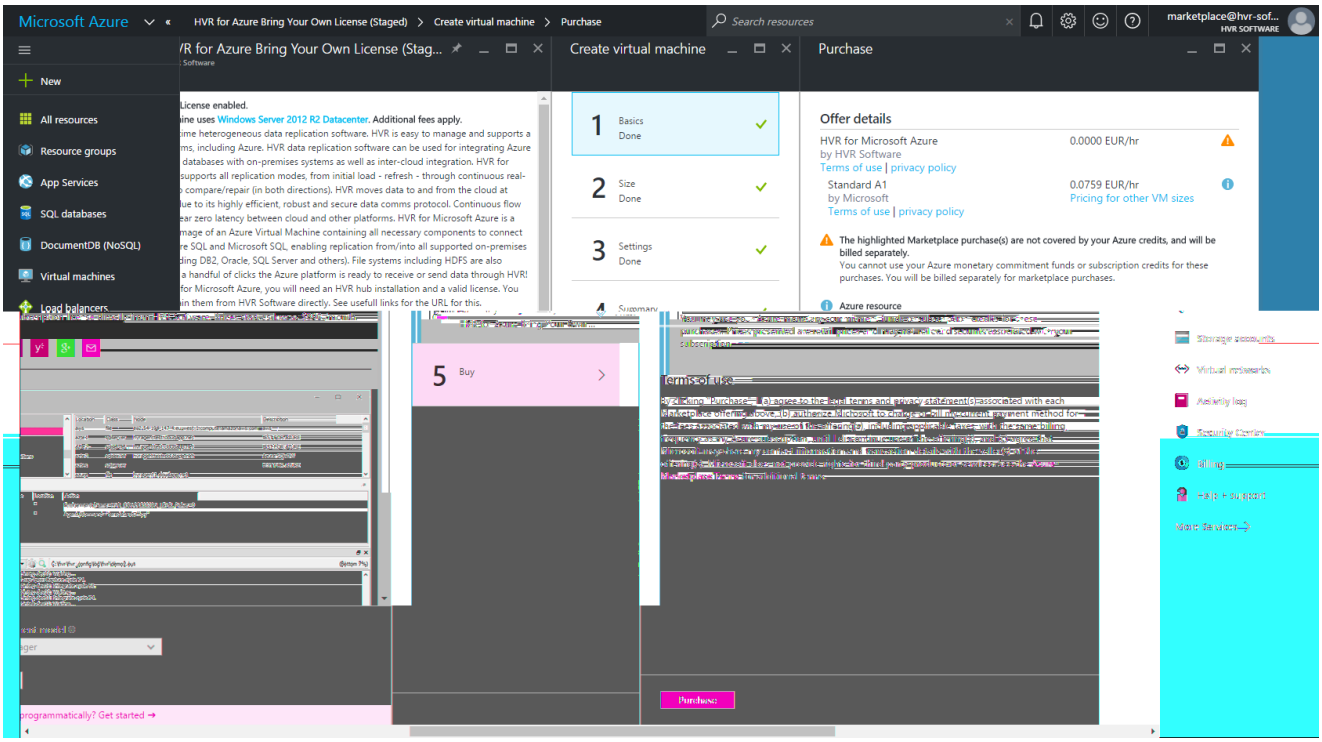
1. In the Azure portal, go to the Marketplace and select HVR for Azure Choose the version that most suitable for you: "**Bring Your Own License**", "**1 source**" or "**5 sources**". Note that the "**1 source**" and "**5 sources**" variants include an HVR license and are available at an additional cost. Click **Create**.



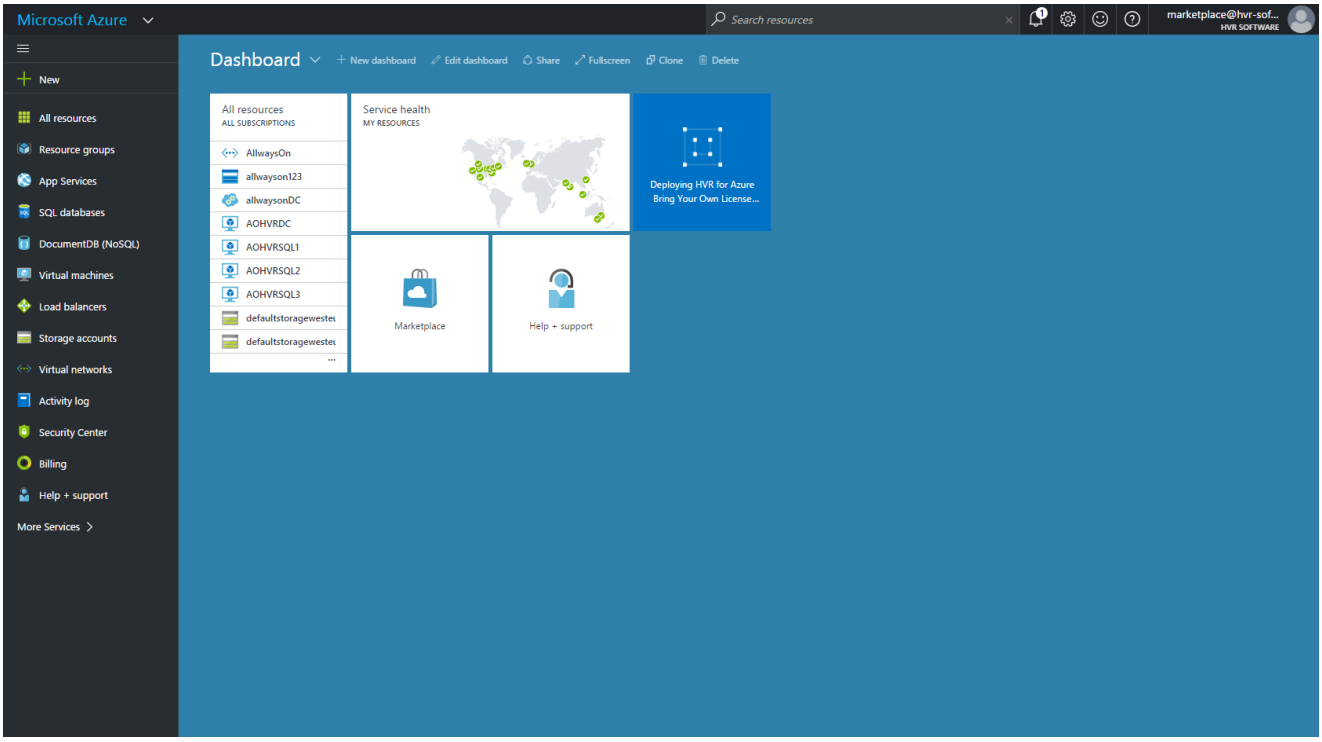
2. In the wizard, follow the sequential steps: **Basics**, **Size**, **Settings**, **Summary** and **Buy**. In each step, you can abort or return.



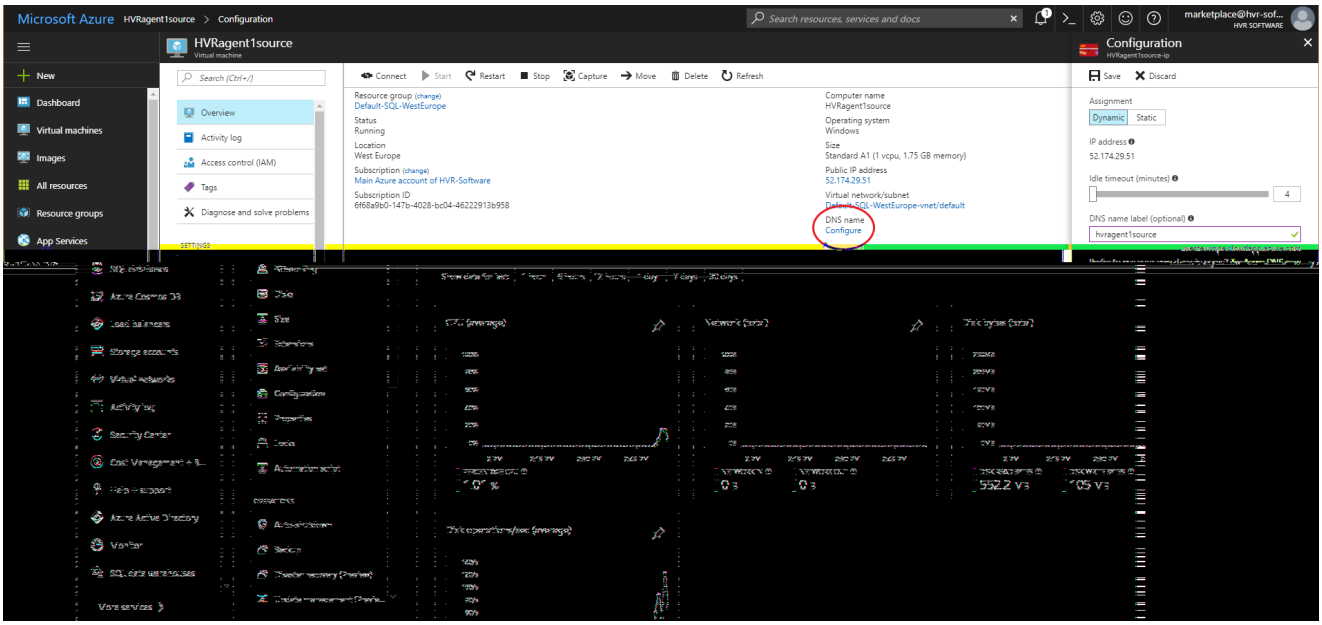
3. On the **Basics** tab, select **VM Disk type** to be **HDD** so as to be able to select the entry level machine types (A) in the next step. Enter the credentials you want to use, select or create a resource group and determine the location you want HVR to run in (typically the same as your databases). Click **OK**.
4. On the **Size** tab, select the appropriate machine type. Azure will suggest machine type **A1**, which is perfectly suitable to run the **HVR for Azure** image when used as a remote listener agent only. Click **Select**.
5. On the **Settings** tab, select the storage account and network settings. By default, Azure will create new ones. If this is your first time, the default settings are fine. Experienced users may want to reuse the existing storage accounts and virtual networks. Click **OK**. In the **Summary** panel, all your choices are listed for your reference. Click **OK**.
6. The **Buy** tab shows the regular costs of the VM you are going to deploy, as well as the cost of the HVR license. The BYOL version costs €0.000 because the image does not contain a license. Click **Purchase**.



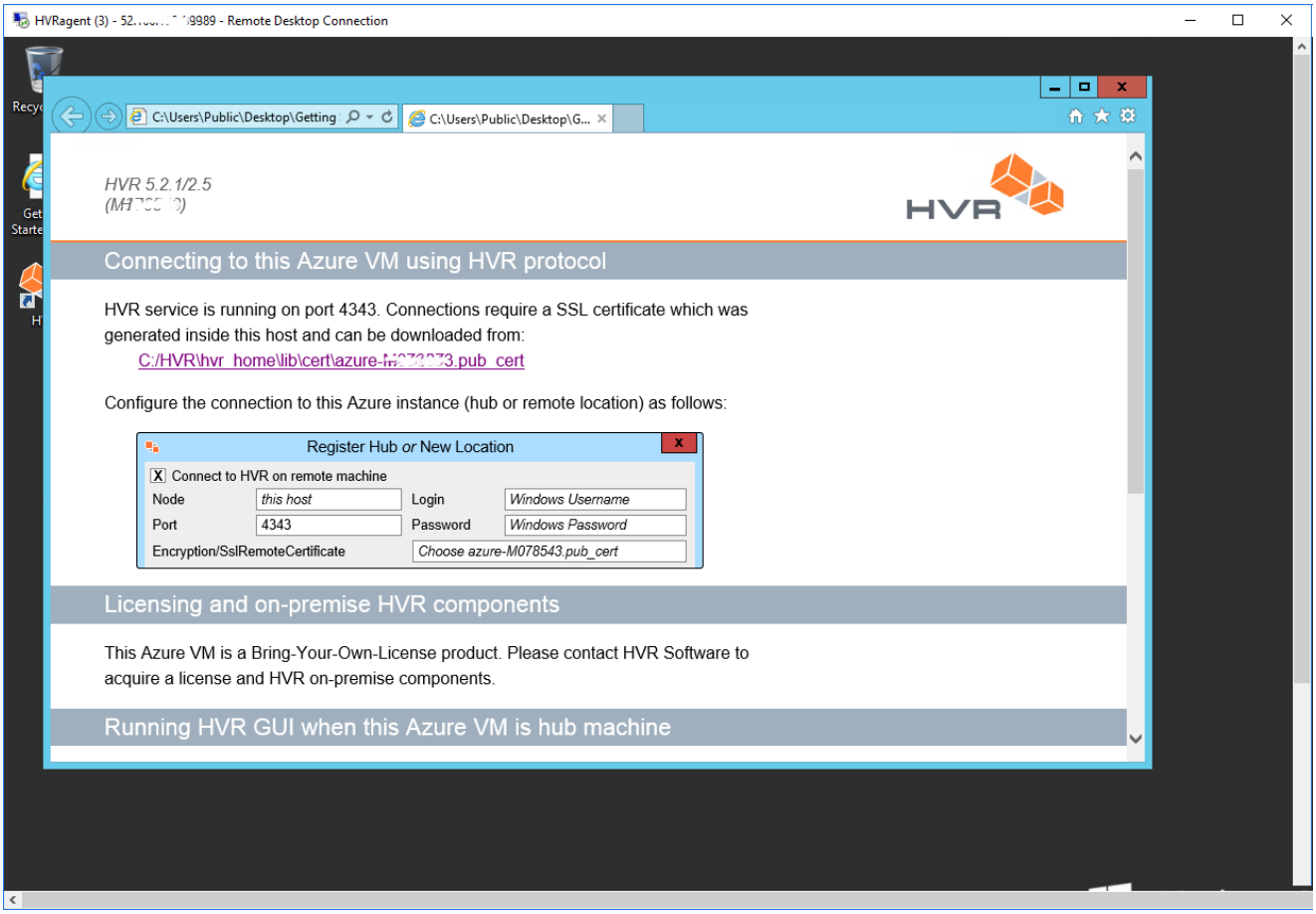
7. You will now see the VM image being deployed in the Marketplace. Once it is completed, the details of the created VM will be available under the **Virtual Machines** tab.



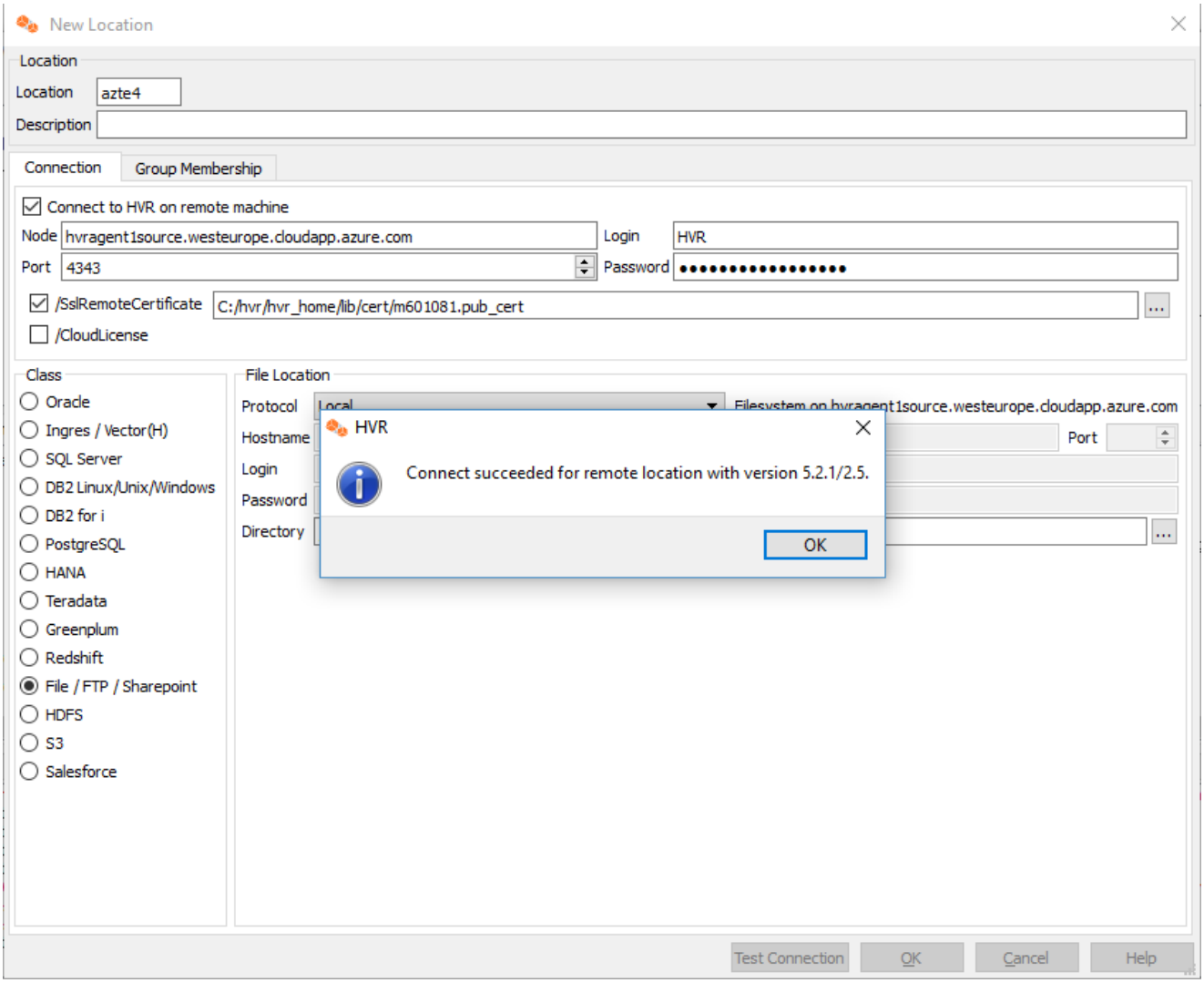
8. Optional step: By default, no DNS name is created for the IP address of the VM. To create a DNS name, click **Configure** under the **DNS name** of the created VM.



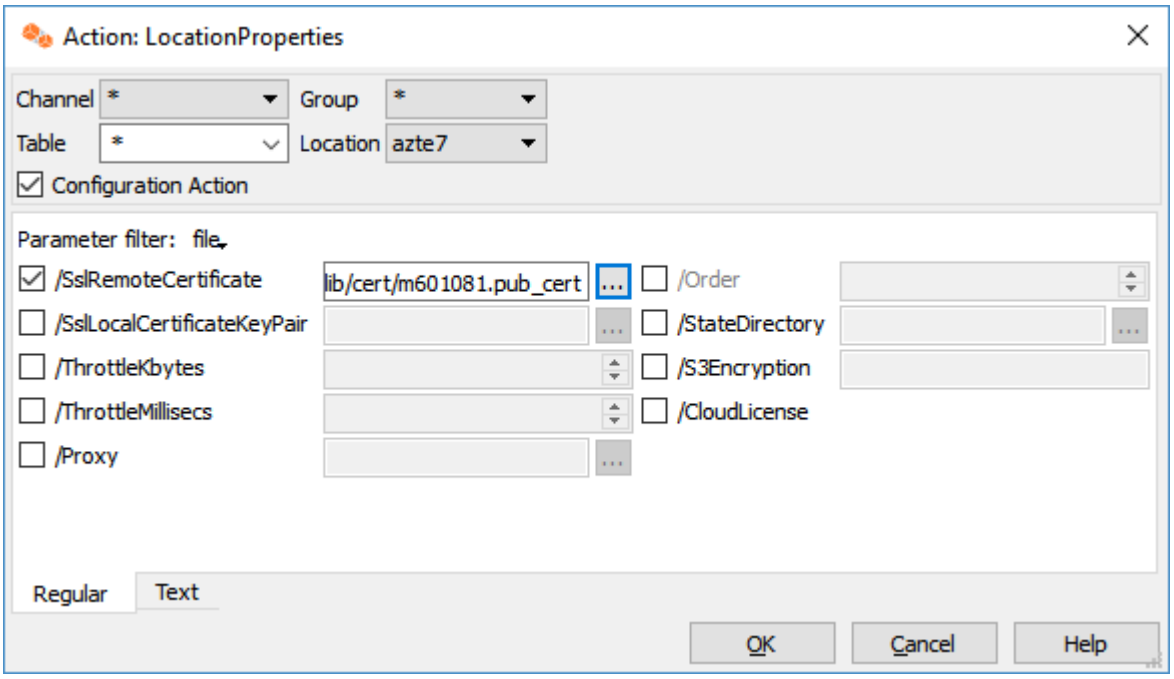
9. The HVR for Azure image will create a remote listener agent with network encryption enabled by default. A certificate is needed to establish a connection. Connection details on the hub should be configured to force all data to be encrypted. For this, download the necessary certificate from the VM agent. To do this, log in to your Azure VM using remote desktop and open the Getting Started web page on the home page and click the certificate link.



10. Copy the contents of the certificate and save them into a certificate file (.pub_cert) on your hub machine in *%HVR_HOME%\lib\cert* folder.
11. Exit the Remote Desktop session to the Azure VM. Create a new location on the hub. Select **/SslRemoteCertificate** and enter the name of the certificate file you just created.



12. If you want to update an existing Azure location, add action **LocationProperties** **/SslRemoteCertificate** <*filename*> for your Azure location (or group) on the hub instead.



Using Licensed HVR for Azure Image

A licensed **HVR for Azure** image contains a license that can be used only for the image just created. If you use the image as an HVR remote listener agent, the hub (residing elsewhere, e.g. on-premises) should be instructed to use this license.

1. Install and start the hub as described in section Installing HVR on Windows or Installing HVR on Unix or Linux. Ignore messages warning that there is no valid license file.
2. Create a new location on the hub. Select **Connect to HVR on remote machine** and enter the credentials of the Azure VM you just created. Then select **LocationProperties /CloudLicense**. You will also have to follow the steps above to obtain the certificate and enter it in **/SslRemoteCertificate**.
3. If you want to update the existing Azure location, add Action **LocationProperties /CloudLicense** for your Azure location (or group) on the hub instead. Again, **LocationProperties /SslRemoteCertificate** <*filename*> should also be set.

Using HVR for Azure Image for Hub Installation

Follow the steps above to instantiate the **HVR for Azure** image, but select at least the **A2** VM type due to the memory requirements of the HVR Hub (up to step 7). Then log in to the created VM and complete the installation steps as described in section Installing HVR on Windows, skipping the installation steps for HVR Distribution and Perl.

If you have acquired a licensed variant of the **HVR for Azure** image, a valid license will be included in the HVR installation on the VM and you can skip the step copying the license file. This license only works on this particular Azure VM and cannot be migrated to another VM or environment. Instead, delete the VM and create a new one. Deleting will automatically stop the charging of the license fee.

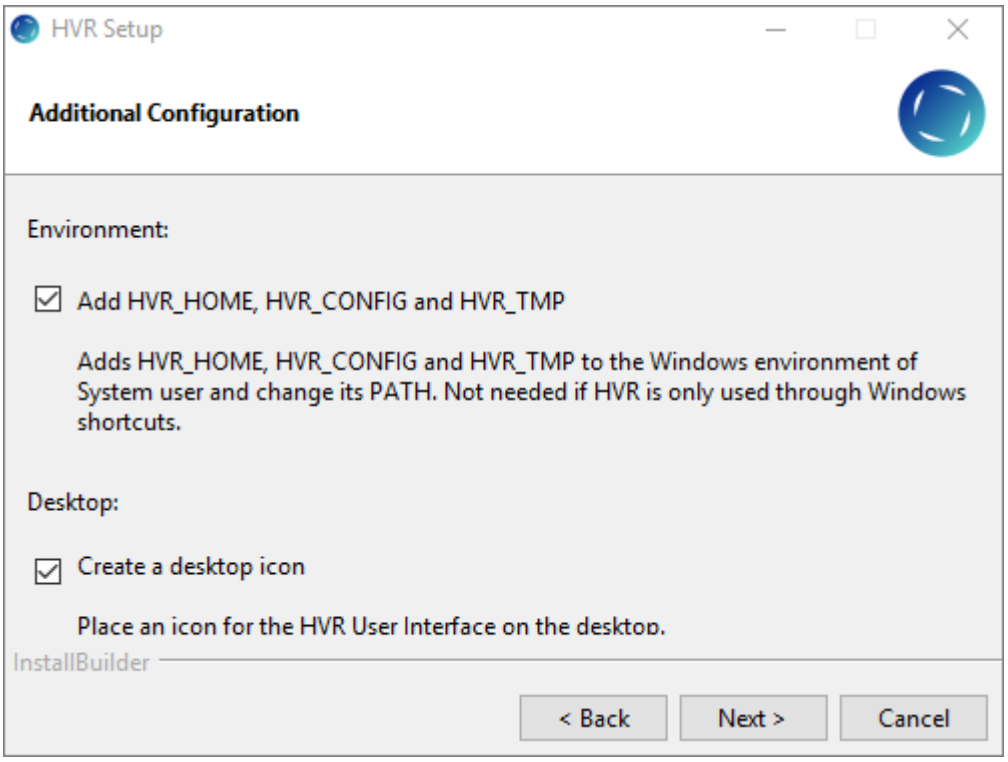
To use the Azure SQL Database service (as location or hub), see section Requirements for Azure SQL Database.

Installing HVR on Azure Manually

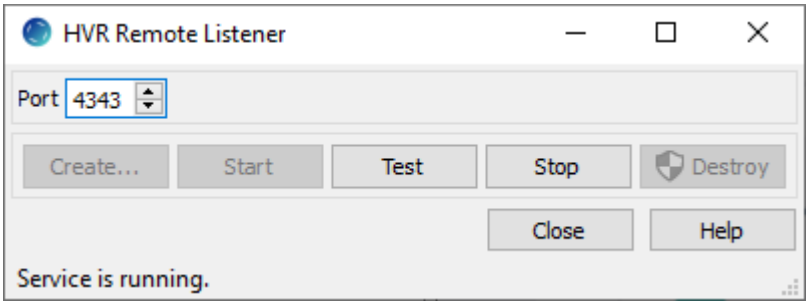
This section describes the steps to manually install an HVR Agent on Azure virtual machine when using Azure SQL Database for replication. Another way to install an HVR Agent on Azure virtual machine is to use a pre-configured **HVR for Azure** image available on the Azure Marketplace.

Follow the steps to create HVR instance on Azure virtual machine (VM):

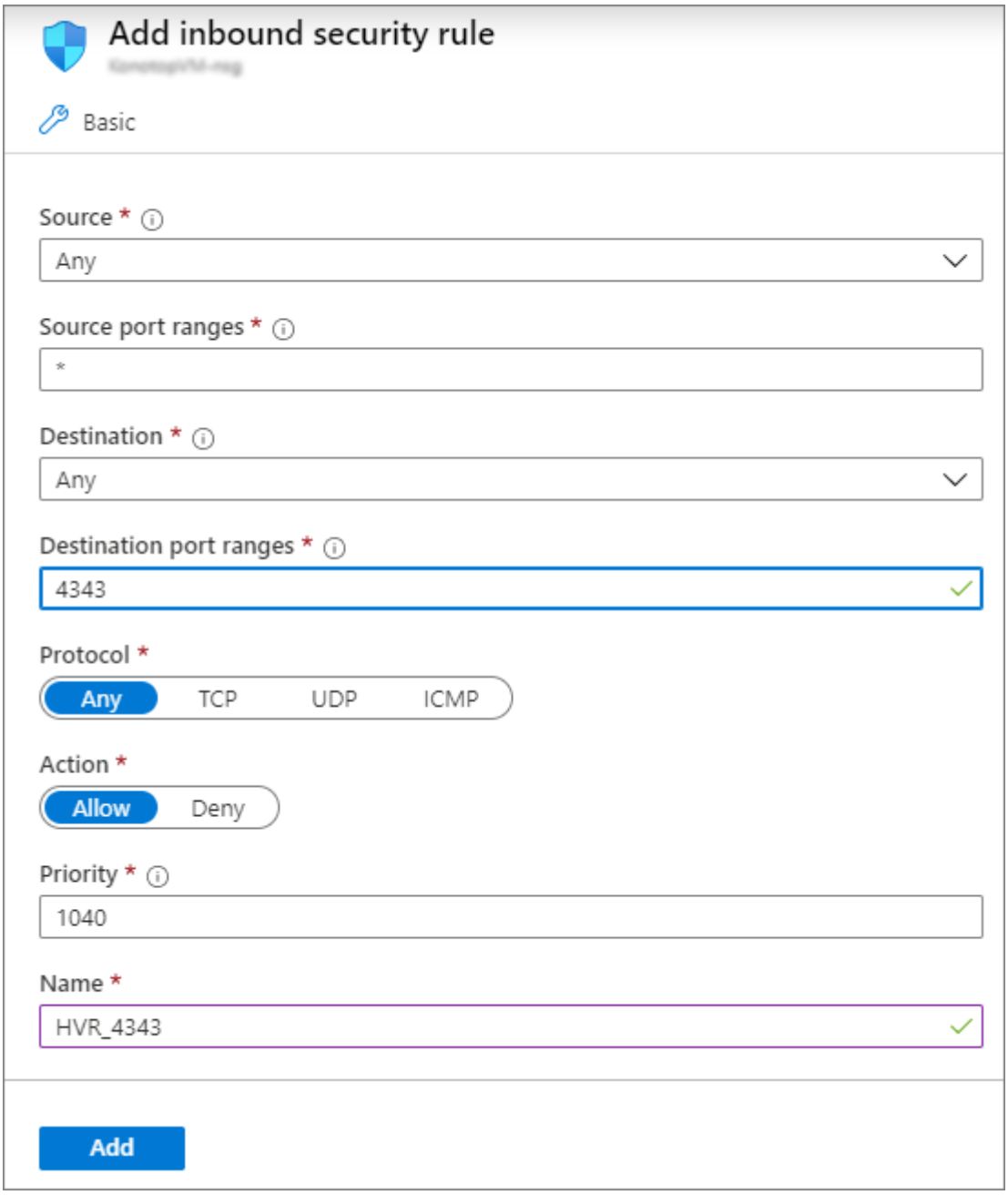
1. On the Azure portal home page, click **Virtual Machines** and then click **Add** to create a new virtual machine. Complete the corresponding steps to configure the virtual machine as required (**Basics**, **Disks**, **Networking**, **Management**, **Advanced**, **Tags**). For example, under the **Basics** tab, select **Windows Server 2012** as a VM image and **A1** for the VM size. Please note that the size of the VM depends on the workload that you want to run.
2. Install the HVR software on the virtual machine following the steps of the HVR Setup wizard. On step **Additional Configuration**, select **Add HVR_HOME, HVR_CONFIG and HVR_TMP** option. Note that the HVR user needs the privilege **Log on as a service** on Windows.



3. Create and start the HVR Remote Listener service as described in section Creating and Starting HVR Remote Listener Service from GUI.



4. In the virtual machine, install the SQL Server 2012 Native Client (**sqlncli.msi**) and configure Windows Firewall for allowing connections both **local** and **remote** for **hvrremotelistener.exe**.
5. Finally, in the Azure portal, create an endpoint for the HVR Remote Listener with the same port number you have specified in step 3. Go to the **Networking** settings of your virtual machine, click **Add inbound security rule** and type the port number (e.g. 4343) in the **Destination port ranges**.



6. Optional step - when the Azure SQL database is not accessible from the virtual machine (e.g. in different subscriptions or zones), add its IP address to the Azure SQL firewall. On the Azure Portal, go to **SQL databases** and select the database you want to give access to. In the database, select **Set server firewall**, this will open the **Firewall settings** dialog, where the IP address can be added by clicking **Add client IP**.

⚠ Note that the VM in Azure can also be a Linux server. For the steps to install HVR on Linux, refer to section Installing HVR on Unix or Linux. Additionally, a relevant ODBC driver for the database you use need to be installed on the Linux VM.

Installing HVR on macOS

Contents

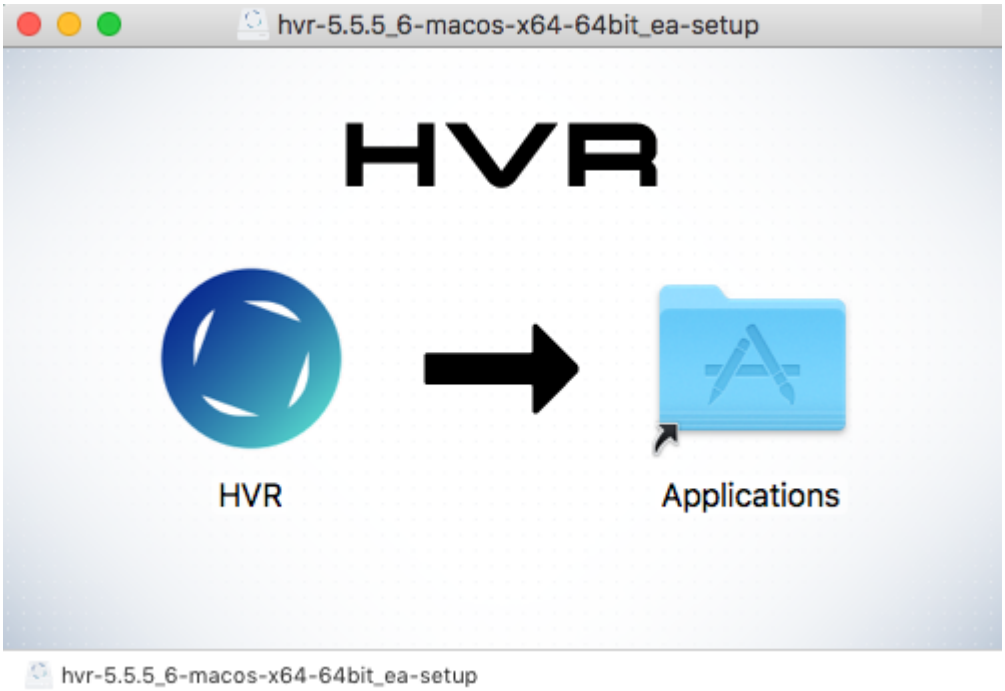
This section provides a step-by-step instruction on how to install HVR on macOS.

⚠ HVR on macOS can only be used as **HVR GUI** to connect to remote hubs or for file replication. HVR does not support hub, capture or integrate databases on macOS.

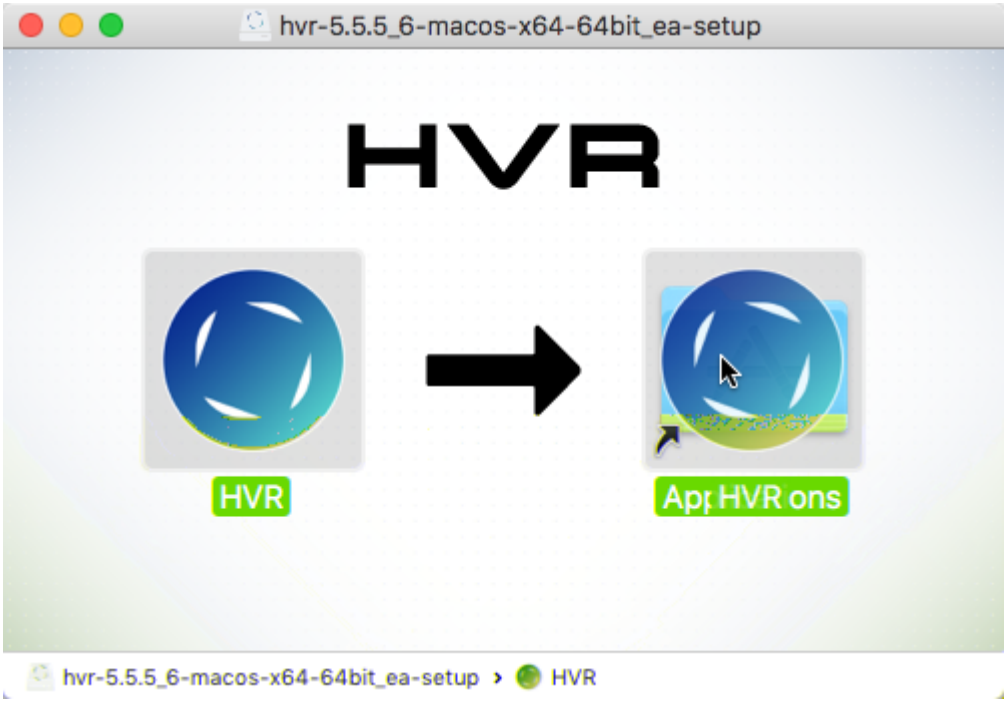
Install HVR on macOS

To install HVR on macOS, perform the following steps:

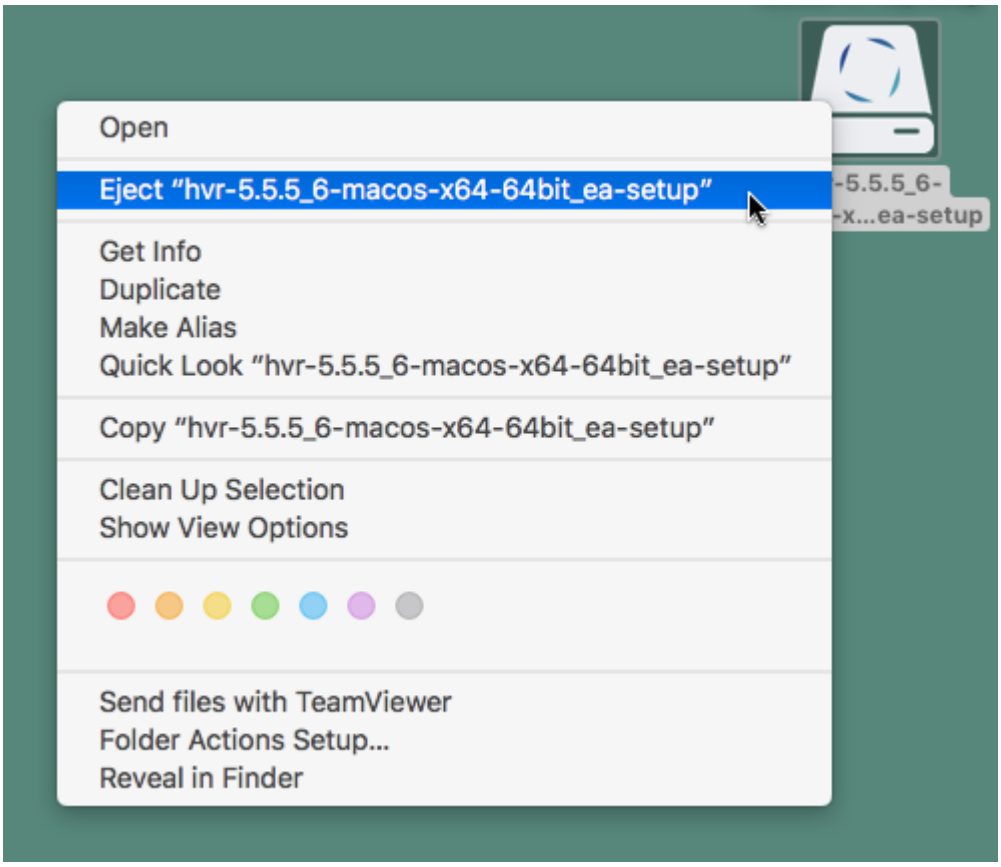
1. Log in with the standard user credentials.
2. Double-click the HVR distribution file for macOS, **hvr-***hvr_ver***-macos-x64-64bit-setup.dmg**. Then read and accept the license terms. A file system named **hvr-***hvr_ver***-macos-x64-64bit-setup** will be mounted on the desktop and the **Finder** window will open.



3. Drag & drop the HVR logo to the **Applications** folder.



4. After all the files are copied to the **Applications** folder, the installation is complete.
5. Eject the **hvr-***hvr_ver***-macos-x64-64bit-setup** file to remove it from the desktop.

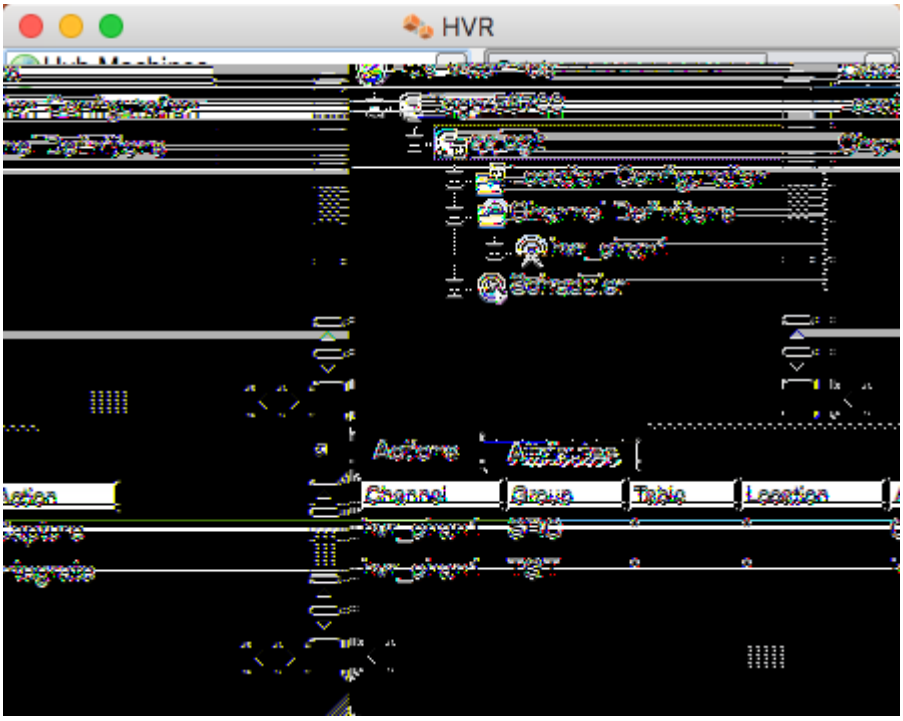


Upgrading HVR on macOS

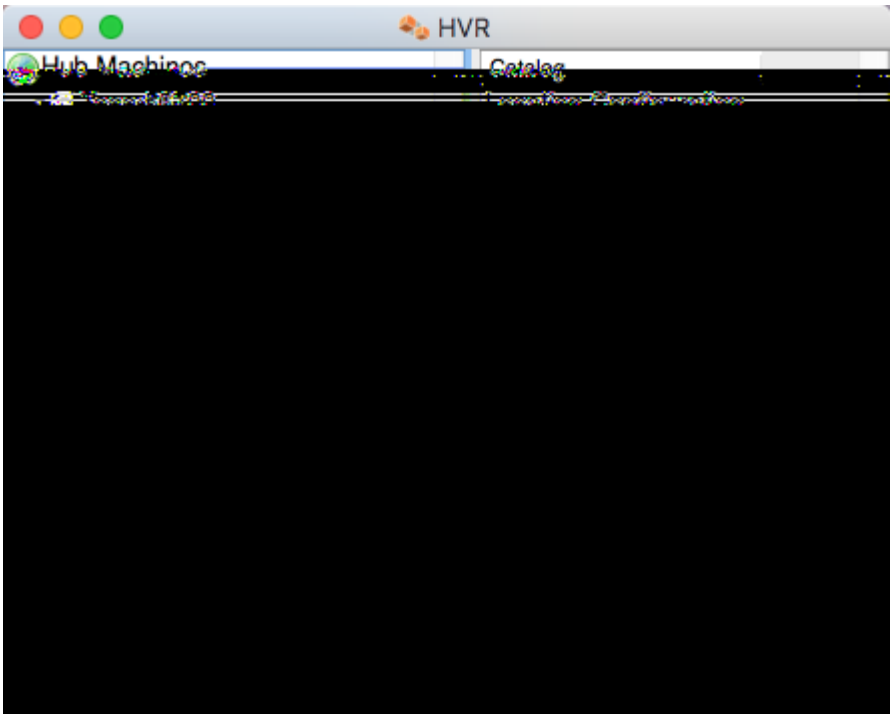
To upgrade HVR on macOS, perform the installation steps mentioned in section Installing HVR on macOS. After step 4 (Drag & drop the HVR logo to the **Applications** folder), click **Replace** in the system dialog to replace the old installation.

Changing Look and Feel in macOS

By default, HVR's GUI on macOS has the same look and feel as the GUI on Unix/Linux.



This can be changed to a beta version of the native macOS look and feel by clicking **View Look and Feel Mac (beta)**.



Installing HVR on Unix or Linux

Contents

This section provides information about the requirements and a step-by-step instruction on how to install HVR on Unix or Linux. The installation procedure described here is applicable (and also same) for installing - HVR as Hub or HVR as remote agent.

Requirements for Unix or Linux

Installation of HVR's own protocol can require **root** permission. HVR's protocol is needed when connecting from the HVR GUI to the **hub** machine and also when connecting from the hub machine to a remote HVR location. The **root** permission is needed to edit the **inetd**, **xinetd** or **systemd** configuration files. An alternative configuration method for HVR's protocol is to use command **Hvrremotelistener** instead of **inetd**, **xinetd** or **systemd**; this alternative does not necessarily need **root** permission.

Install HVR on Unix or Linux

To install HVR on Unix or Linux, perform the following steps as the user which will operate the HVR application:

> It is recommended to create a non-root account for installing and operating HVR. We suggest creating a separate user account (e.g, **hvruser**) for this purpose.

1. Configure the environment variables **HVR_HOME**, **HVR_CONFIG**, and **HVR_TMP** for your operating system. Each of these environment variables should be pointed to the HVR installation directories - **hvr_home**, **hvr_config**, and **hvr_tmp**.

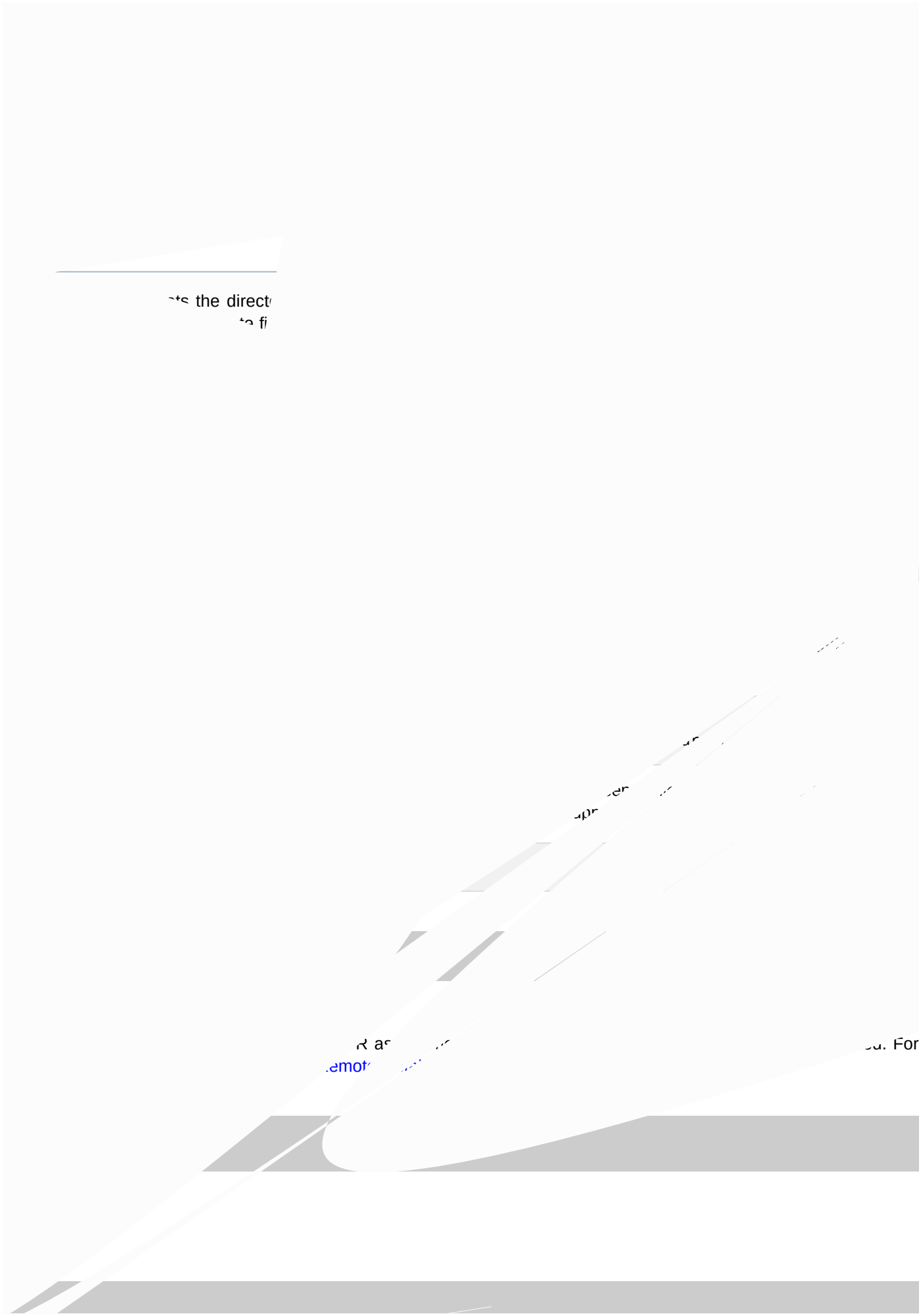
   ```
   $ export HVR_HOME=/home/hvruser/hvr/hvr_home
   $ export HVR_CONFIG=/home/hvruser/hvr/hvr_config
   $ export HVR_TMP=/home/hvruser/hvr/hvr_tmp
   ```

   > The commands to set the environment variables depend on the shell you use to interface with the operating system. This procedure lists examples that can be used in Bourne Shell (sh) and KornShell (ksh).

2. Add the HVR executable directory path to the environment variable **PATH**.

   ```
   $ PATH=$PATH:$HVR_HOME/bin
   ```

3. Add the HVR executable directory path into the startup file (e.g. **.profile**).



After the installation, HVR can be controlled using HVR's graphical user interface (**HVR GUI**).

For HVR on Linux, **HVR GUI** can be executed directly on the hub server. However, an 'X Window System' application must be installed to execute **HVR GUI** directly on Linux. To control HVR remotely from your PC, install HVR on the PC (with Windows or MacOS) and configure HVR Remote Listener on hub server.

For HVR on Unix, **HVR GUI** should be typically executed remotely from a PC to control HVR installed on hub server. To do this, install HVR on the PC (with Windows or MacOS) and configure HVR Remote Listener on hub server.

Notes for Oracle

If HVR must perform log based capture from Oracle, then the Unix username that HVR uses must be able to read the redo files and archive files that Oracle generates. This can either be done by adding HVR user to the **oinstall** or **dba** group in **/etc/group**. The permission to read these files can also be given by creating special access control lists (ACLs). For more information, see Requirements for Oracle.

Notes for Ingres

To perform log-based capture from Ingres a trusted executable must be created so that HVR can read from the internal DBMS logging files. For more information, see section Creating Trusted Executable in Requirements for Ingres and Vector.

See Also

For information about configuring HVR after the installation, see section Configuring HVR with the following topics:

- Auto-Starting HVR Scheduler after Unix or Linux Boot
- Auto-Starting HVR after Windows Boot
- Configuring Remote Installation of HVR on Unix or Linux
- Configuring Remote Installation of HVR on Windows
- Authentication and Access Control
- Encrypted Network Connection
- Hub Wallet and Encryption
- Network Protocols, Port Numbers and Firewalls
- Regular Maintenance and Monitoring for HVR
- HVR High Availability

Installing HVR on Windows

Contents

This section provides information about the requirements and a step-by-step instruction on how to install HVR on Microsoft Windows. The installation procedure described here is applicable (and also same) for installing - HVR as Hub or HVR as remote agent.

Requirements for Windows

- Perl (version 5.6 or higher) must be installed if this is a hub server or if this is a server where HVR will perform log-based capture and will use command **Hvrlogrelease**. If Perl is installed after the installation of HVR, then the **HVR Scheduler** service must be recreated.
- When creating the **HVR Scheduler** or **HVR Remote Listener** service, HVR needs elevated privileges on Windows 2008 and Windows 7. Normally this means the user must supply an **Administrator** password interactively.
- On the hub machine, HVR's user needs the privilege **Log on as a service**.
- The**HVR Remote Listener**service can be installed to run in one of two ways:
  - As a **Local System**.
  - As a normal Windows user. In this case the user must have privilege **Log on as a service**. But this service can only handle connections to its own username/password, unless it is a member of the **Administrator** group (Windows 2003 and XP) or has **Replace a process level token** privilege (Windows 2008 and Windows 7).

To start the Windows tool for managing privileges, use the command **secpol.msc**.

Install HVR on Windows

HVR's distribution for Windows is available for download in **.exe** and **.zip** format. The compressed file (**.zip**) distribution is normally used as an alternative for the Windows executable based (**.exe**) distribution. The steps to install HVR are also different on either formats of the distribution.

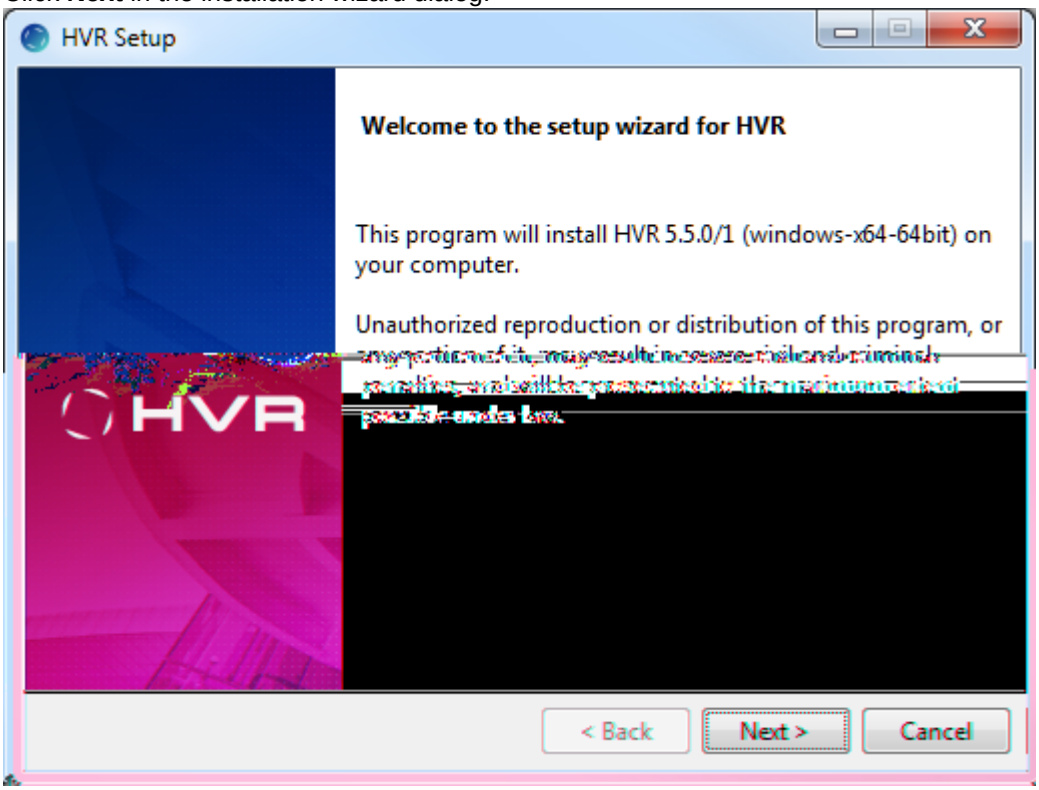
- Install Using Executable (.exe) File
- Install Using Compressed (.zip) File

Install Using Executable (.exe) File

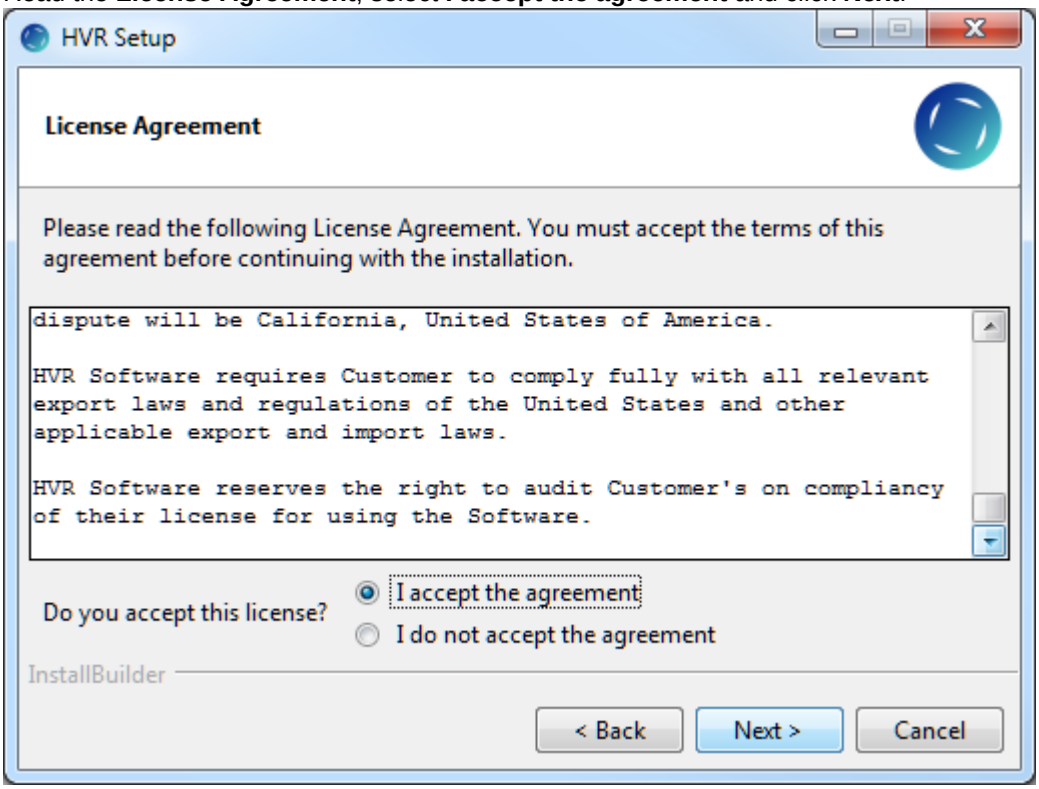
To install HVR on Microsoft Windows using the **.exe** file, perform the following steps:

1. Log in to Microsoft Windows under a normal user account.
2. Run the HVR distribution file **hvr-***hvr_ver***-windows-x64-64bit-setup.exe**.

3. Click **Next** in the installation wizard dialog.

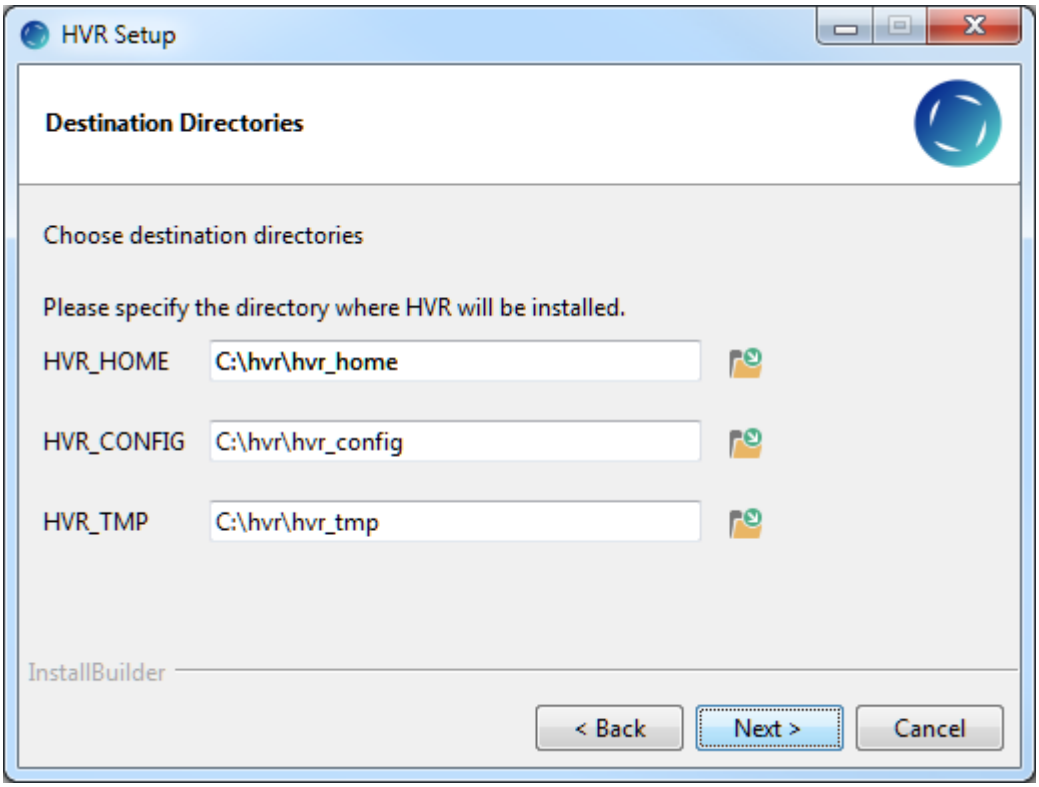


4. Read the **License Agreement**, select **I accept the agreement** and click **Next**.

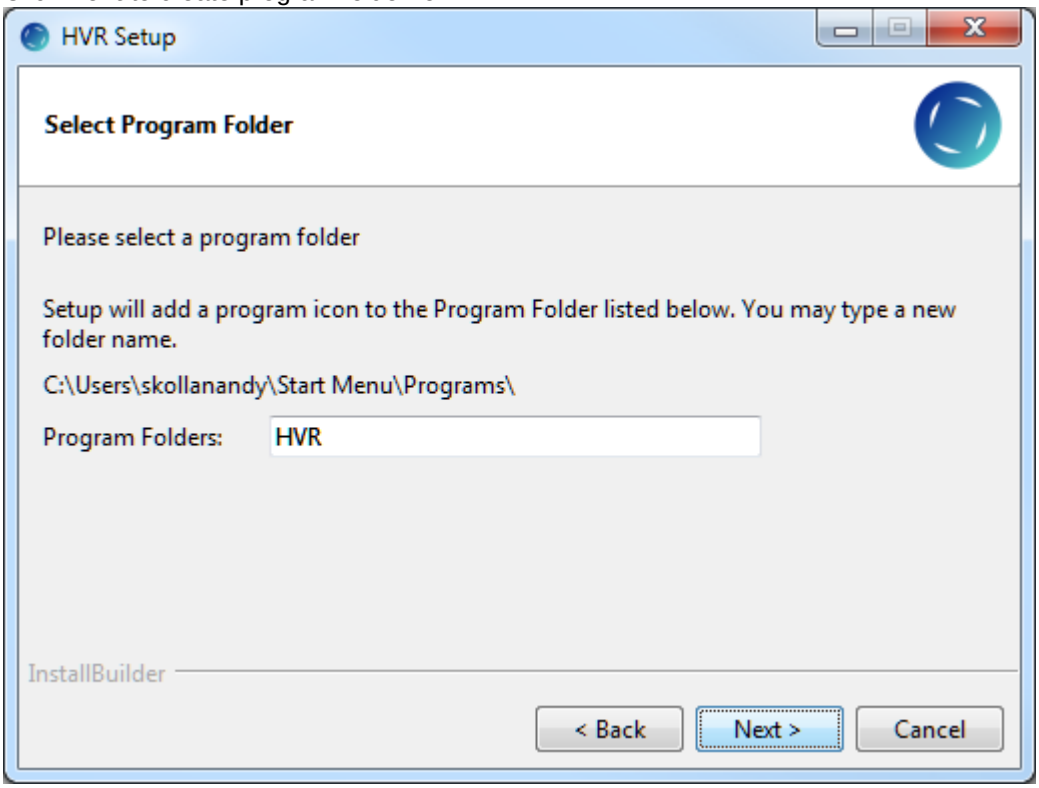


5. Specify the HVR installation directory for **HVR_HOME**, **HVR_CONFIG**, and **HVR_TMP** and click **Next**.

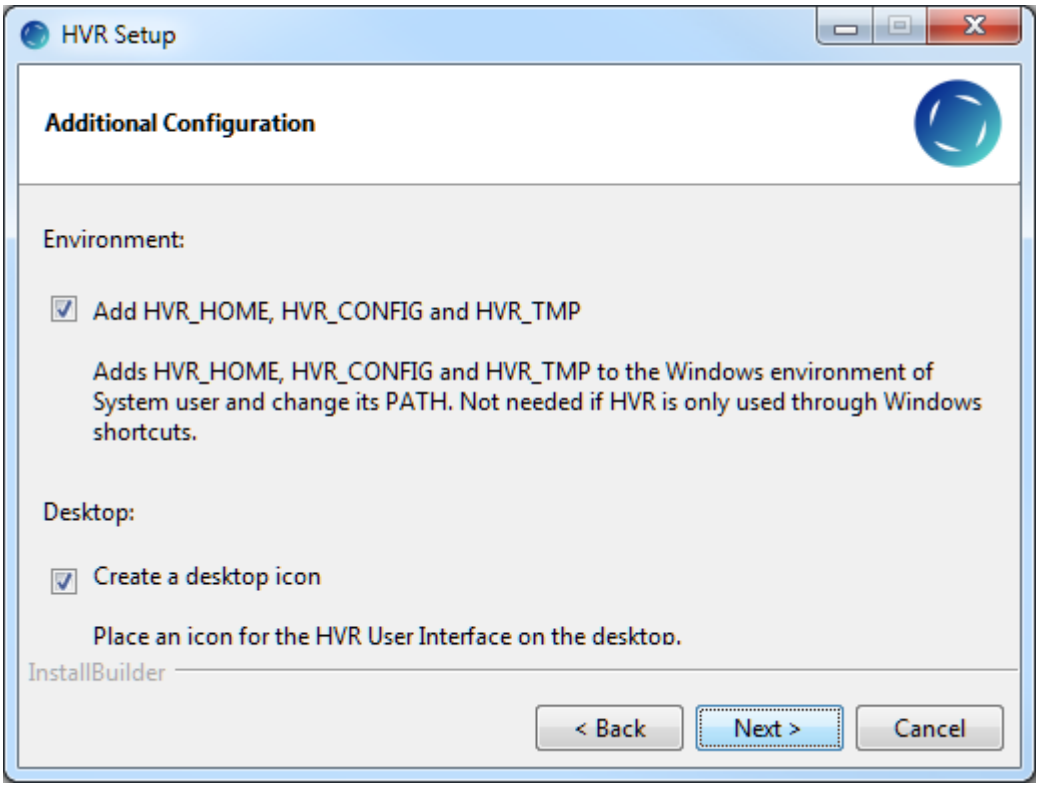
> **HVR_HOME** is regarded a read-only directory. The user files saved in this directory will be moved to a backup directory when executing HVR for the first time or after an HVR upgrade.



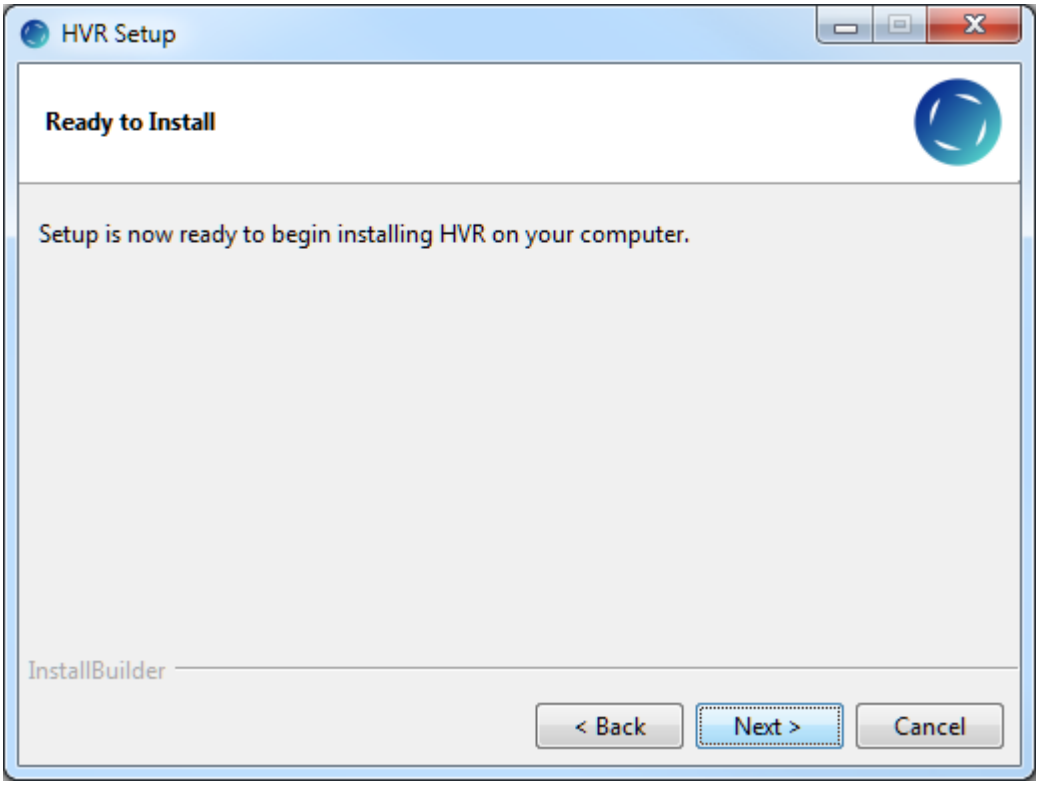
6. Click **Next** to create program folder for HVR.



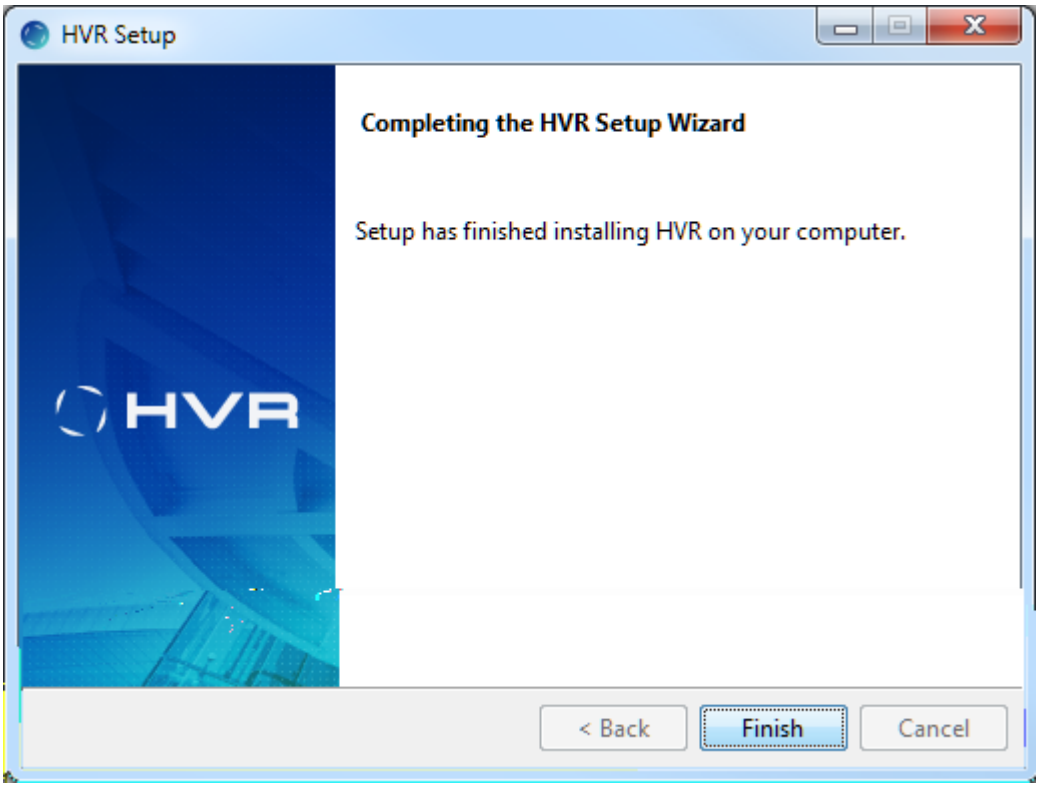
7. Select **Add HVR_HOME, HVR_CONFIG and HVR_TMP**, if required. This is to set the environment variables **HVR_HOME**, **HVR_CONFIG**, and **HVR_TMP** for your operating system. Each of these environment variables should be pointed to the respective HVR installation directories - **hvr_home**, **hvr_config**, and **hvr_tmp**.



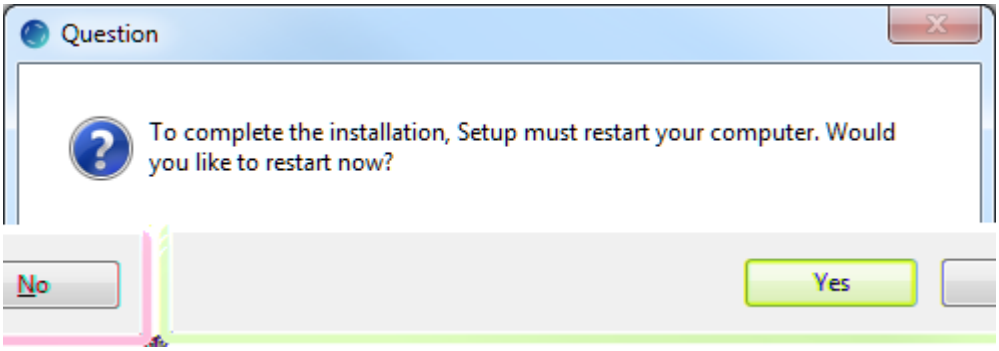
8. Click **Next** to initiate the installation.



9. After the installation is complete, click **Finish**.



10. The computer must be restarted to complete the installation. Click **Yes** to restart the computer immediately or **No** to restart later.



11. If this installation is done for using 'HVR as a hub' then copy the HVR license file (**hvr.lic**) into **%HVR_HOME%\lib** folder. The license file is normally delivered by HVR Technical Support.
12. If this installation is done for using 'HVR as a remote agent' then an HVR listener port must be configured. For more information, see Configuring Remote Installation of HVR on Windows.

For information about configuring HVR after installation, see section **Configuring HVR**.

Install Using Compressed (.zip) File

To install HVR on Microsoft Windows using the **.zip** file, perform the following steps:

1. Log in to Microsoft Windows under a standard user account.
2. Create the directories **hvr_home**, **hvr_config**, and **hvr_tmp** required for installing HVR.
   Example:

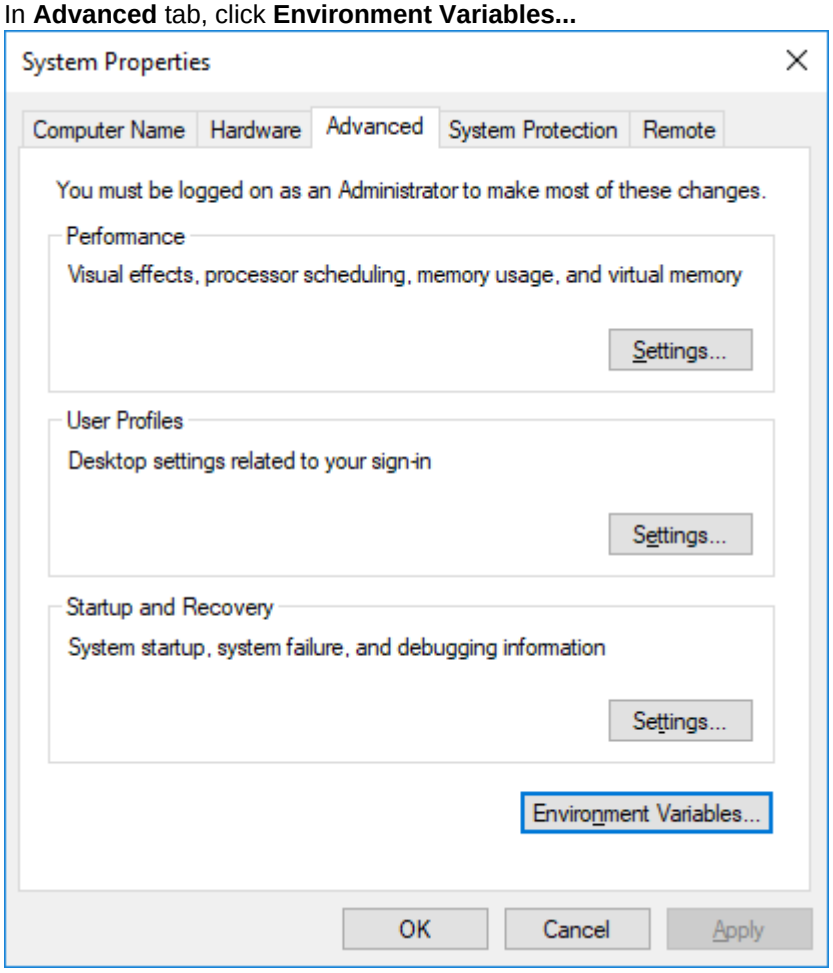
```
C:\>mkdir hvr
C:\>cd hvr
C:\hvr>mkdir hvr_home hvr_config hvr_tmp
```

> **hvr_home** is regarded a read-only directory. The user files saved in this directory will be moved to a backup directory when executing HVR for the first time or after an HVR upgrade.

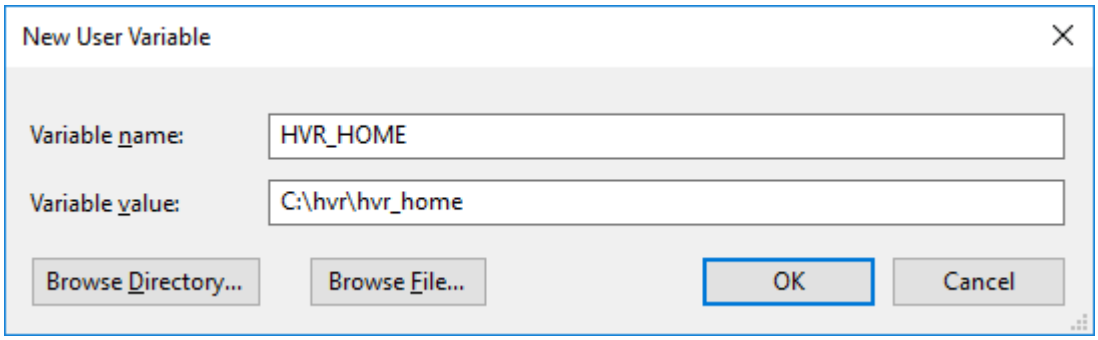
3. Extract (unzip) HVR's compressed distribution file **hvr-*hvr_ver*-windows-x64-64bit_ga.zip** into **hvr_home** directory.
4. Configure the environment variables **HVR_HOME**, **HVR_CONFIG**, and **HVR_TMP** for your operating system. Each of these environment variables should be pointed to the respective HVR installation directories - **hvr_home**, **hvr_config**, and **hvr_tmp**.
5. To set the environment variables:
   a. Execute the command **sysdm.cpl** to access the Windows **System Properties**.

> **System Properties** can also be accessed from **Control Panel System and Security System Advanced system settings**. The shortcut for this is **Windows Key+Pause**.

   b. In **Advanced** tab, click **Environment Variables...**



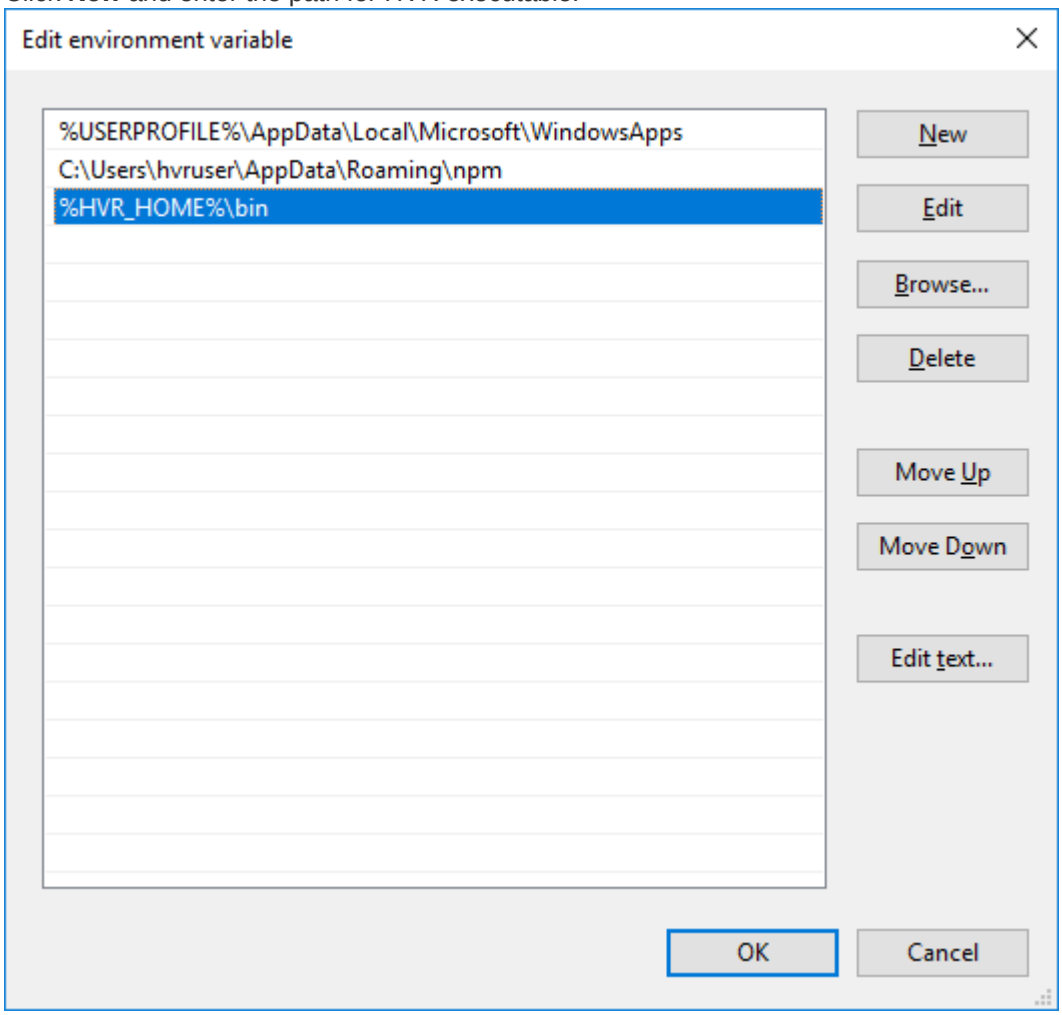
   c. In the section **System variables** or **User Variables for** *user name*, click **New**.
   d. Enter Variable name (e.g, **HVR_HOME**) and Variable value (e.g, **C:\hvr\hvr_home**).



e. Click **OK**.
f. These steps should be repeated for each environment variables.

6. Add the HVR executable directory path (**%HVR_HOME%\bin**).

a. In the section **System variables** or **User Variables for** *user name*, from the list of Variables, select **Path** and click **Edit...**.
b. Click **New** and enter the path for HVR executable.



c. Click **OK**.

7. Click **OK** in **System Properties** dialog.
8. If this installation is done for using 'HVR as a hub' then copy the HVR license file (**hvr.lic**) into **%HVR_HOME%\lib** directory. The license file is normally delivered by the HVR Technical Support.
9. If this installation is done for using 'HVR as a remote agent' then an HVR listener port must be configured. For more information, see Configuring Remote Installation of HVR on Windows.

If the HVR installation is done for using 'HVR as a remote agent' (and not as a hub), then to save space, files not required for the HVR remote agent can be removed using the command **hvrstrip -r**.

For information about configuring HVR after installation, see section **Configuring HVR**.

Notes for SQL Server

- For trigger-based capture from SQL Server (not Microsoft Azure SQL Database), a special extended stored procedure can be created called **hvrevent**. This procedure is not necessary for a capture machine if it is log-based capture or certain capture parameters are defined (**/ToggleFrequency** or **Scheduling/CaptureStartTimes** or **/CaptureOnceOnStart**). This DLL must be registered in the master database with the username under which HVR will run. This step must be performed by a user which is a member of the **system administration** role. Use the following SQL command:

```
C:\>osql -d master -U <user> -P <password>
> exec sp_addextendedproc 'hvrevent', 'c:\hvr\hvr_home\bin\hvrevent.dll'
> go
> grant execute on hvrevent to public
> go
```

- Replace pathname **c:\hvr\hvr_home** with the correct value of **HVR_HOME** on that machine.
- To remove this extended stored procedure use the following SQL statement:

```
exec sp_dropextendedproc 'hvrevent'
```

- If HVR does not actually run on the machine that contains the database (either the hub database is not on the hub machine or HVR is capturing from a database without running on the machine that contains this database) then this step should be performed on the database machine, not the HVR machine. If the HVR machine is 32 bit Windows and the other database machine is 64 bit Windows then copy file **hvrevent64.dll** instead of file **hvrevent.dll**. If both machines are 32 bit Windows or both are 64 bit Windows, then the file is just named **hvrevent.dll**.

Notes for Azure

- For installations on Windows VMs in Azure, HVR Image for Azure can be used. For agent only installations, after deploying the image the installation is finished. For using it as a first step in a Hub installation (use at least an *A2* VM type), you can skip the HVR Distribution and Perl installation steps. Oracle and MS SQL database drivers are preinstalled, so direct connections to these databases (including usage as hub database) can be made.
- Azure SQL Databases can be accessed as SQL Server locations. To connect to the Azure SQL Database service from the HVR hub on an Azure VM, make sure to set the database server configuration's firewall to enable **Allow access to Azure services**. Using Azure SQL Database as the hub database is fully supported. For more information, see Requirements for Azure SQL Database.

See Also

For information about configuring HVR after the installation, see section Configuring HVR with the following topics:

- Auto-Starting HVR Scheduler after Unix or Linux Boot
- Auto-Starting HVR after Windows Boot
- Configuring Remote Installation of HVR on Unix or Linux

- Configuring Remote Installation of HVR on Windows
- Authentication and Access Control
- Encrypted Network Connection
- Hub Wallet and Encryption
- Network Protocols, Port Numbers and Firewalls
- Regular Maintenance and Monitoring for HVR
- HVR High Availability

Installing HVR in a Cluster

A cluster is a group of nodes which share a 'virtual' IP address and also have a shared file system. Both an HVR hub and an HVR location can be put in cluster package. And if an HVR location is in a cluster, then there are different ways for the HVR hub to connect to it. The three common scenarios are:

- HVR Hub in a Cluster
- Location with HVR 'Inside the Cluster'
- Location with HVR 'Outside the Cluster'

HVR Hub in a Cluster

When the HVR hub runs inside the cluster, make sure that only one HVR Scheduler is running at a time (i.e. active /passive not active/active) and ensure that the **$HVR_CONFIG** directory is shared between all nodes. **$HVR_HOME** should either be shared between all nodes, or an identical copy should be reachable with the same path from all nodes. Directory **$HVR_TMP** can be local. The DBMS of the hub database must also be inside the same 'cluster resource' or at least be reachable with the same name from each cluster node.

On Windows, the commands **hvrscheduler** and **hvrremotelistener** can be enrolled in Windows cluster services using option **–c**. These services must be recreated on each node. Once these services are enrolled in the cluster, then they should only be controlled by stopping and starting the cluster group (instead of using option **–as**).

Command **hvrmaint** should be scheduled so it runs on whichever is machine active.

If the hub database is inside an Oracle RAC, then enroll the HVR services in the Oracle RAC cluster using command **crs_profile** for script **hvr_boot**.

Location with HVR 'Inside the Cluster'

This means HVR will connect with its own protocol to a remote listener (configured using **inetd** on Unix) which is configured to run inside all cluster nodes simultaneously. The hub then connects to the remote location using its relocateable virtual IP address.

If this remote location is a file location, then these nodes must share the same file location top directory and state directory.

Log based capture from an Oracle RAC requires this approach with a single capture location for the Oracle RAC. This location should be defined using the relocatable IP address of the Oracle RAC cluster.

On Windows, the command **hvrremotelistener** can be enrolled in Windows cluster services using option **–c**. This service must be recreated on each node. Once the service is enrolled in the cluster, then it should only be controlled by stopping and starting the cluster group (instead of using option **–as**).

Directory **$HVR_HOME** and **$HVR_CONFIG** should exist on both machines, but does not normally need to be shared. But for log based capture, if command **hvrlogrelease** is used, then **$HVR_CONFIG** must be shared between all nodes. If **$HVR_TMP** is defined, then it should not be shared. Command **hvrlogrelease** should then be scheduled to run on both machines, but this scheduling should be 'interleaved' so it does not run simultaneously. For example, on one machine it could run at '0, 20, 40' past each hour and on the other machine it could run at '10, 30, 50' past each hour.

Location with HVR 'Outside the Cluster'

This means that HVR connects using the DBMS's network protocol (e.g. a TNS alias) as if it were a normal SQL client. This means that it is unaware that the database is really in a cluster.
This is not supported for file locations. A benefit is that HVR does not need to be installed inside the cluster at all; a disadvantage is that the DBMS protocol is not as efficient as HVR's protocol because it does not have compression. HVR log based capture does not work in this situation; it must be installed 'inside' the cluster.

Upgrading HVR

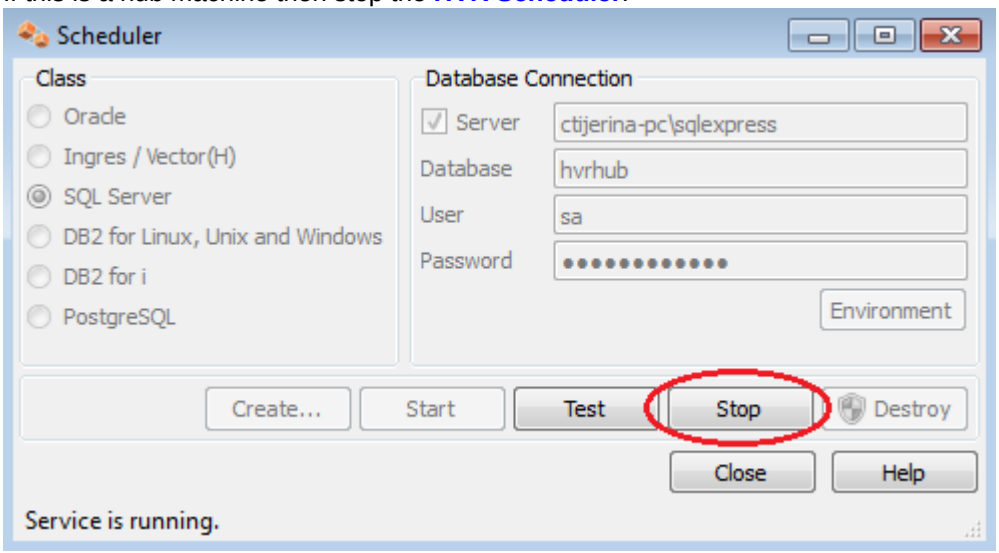
Upgrading installations can be a large undertaking, especially when it involves downtime or lots of machines. For this reason different HVR versions are typically fully compatible with each other; it is not necessary to upgrade all machines in a channel at once. When upgrading from HVR version 4 to HVR 5 additional steps are necessary. Fore more information, see Upgrading from HVR Version 4 to 5

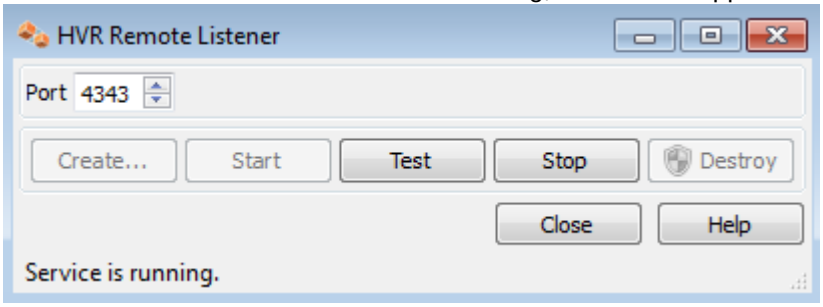
In many environments all machines will just be upgraded at the same time.

It is also possible to only upgrade certain machines (e.g. only the hub machine, GUI, remote source or target). If this is done, it should be understood that each HVR release fixes some bugs and or contains new features. Each fix or feature is only effective if the correct machine is upgraded. For example, if a new HVR release fixes an integrate bug, then that release must be installed on the machine(s) which do integrate. If only the GUI and/or hub machine is upgraded, then there will be no benefit. Read the HVR Release Notes (in **$HVR_HOME/hvr.rel**) for a description of which features and fixes have been added, and which machine must be upgraded for each feature and fix to be effective. New features should not be used until all machines that are specified for that feature are upgraded, otherwise errors can occur.

Before upgrading HVR it is recommended to take a full backup of the following:

- the **HVR_HOME** and **HVR_CONFIG** directories.
- the HVR hub database using the DBMS native backup option.
  Alternatively, use the command **hvrcatalogexport** to create a backup of the hub database. However, this command does not support creating backup of HVR **Events** and **Statistics** related information. To backup the hub database using this method in **HVRGUI**, right-click the hub and select **Export Catalogs**. Ensure that all Catalogs (**Locations**, **Channel Definitions**, **Group Membership**, and **Configuration Actions**) are selected in the **Export Catalogs...** dialog. Click **Browse** and specify the name for the backup file to be created. Click **OK**.

1. If this is a hub machine then stop the **HVR Scheduler**:


2. If the **HVR Remote Listener** service is running, it must be stopped.


3. Stop all **HVR GUI** processes.

4. On AIX, uninstall the old HVR runtime executables using command (only for Unix & Linux):

   Unix & Linux

   ```
   $ rm $HVR_HOME/lib/*
   ```

   Alternatively, remove all cached shared libraries from the kernel by executing the command **slibclean** as **root** user. This step is not needed for other flavors of Unix or Linux.
5. Install the new version of HVR downloaded from https://www.hvr-software.com/account/.

   a. For Unix or Linux, read and uncompress the distribution file into the **HVR_HOME** directory.

   ```
   $ cd $HVR_HOME
   $ umask 0
   $ gzip -dc /tmp/hvr-5.0.3-linux_glibc2.5-x64-64bit.tar.gz | tar xf -
   ```

   b. For Windows, run the HVR installation file (**.exe**) or extract the compressed file (**.zip**) into the **HVR_HOME** directory.
6. If HVR must perform log–based capture from Ingres, it needs a trusted executable to read from the DBMS logging system.

   Click here to see the steps...
   Execute the following commands while logged in as the DBMS owner (**ingres**):

   ```
   $ cd /usr/hvr/hvr_home
   $ cp bin/hvr sbin/hvr_ingres
   $ chmod 4755 sbin/hvr_ingres
   ```

   It is not required to create a trusted executable when either of the following are true:
   - capture is trigger-based
   - capture will be from another machine
   - HVR is running as the DBMS owner (**ingres**)

   Additionally, on Linux, the trusted executable should be patched using the following command:

   ```
   $ /usr/hvr/hvr_home/lib/patchelf --set-rpath /usr/hvr/hvr_home/lib --force-rpath /usr
   /hvr/hvr_home/sbin/hvr_ingres
   ```

   If HVR and **ingres** share the same Unix group, then the permissions can be tightened from 4755 to 4750. Permissions on directories **$HVR_HOME** and **$HVR_CONFIG** may need to be loosened so that user **Ingres** can access them;

   ```
   $ chmod g+rX $HVR_HOME
   $ chmod -R g+rwX $HVR_CONFIG
   ```
7. Launch **HVR GUI** and connect to the hub.

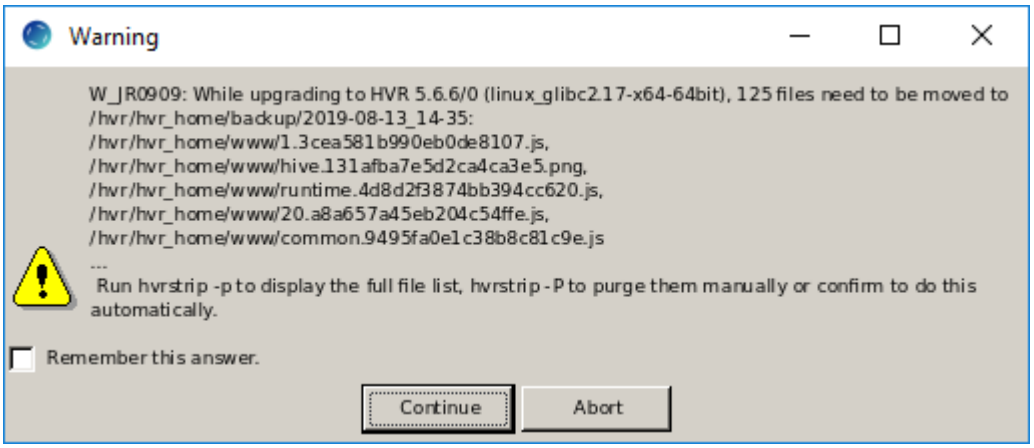
   When connecting to the HVR hub for the first time after the upgrade, a warning dialog (**Warning: W_JR0909**) may be displayed. This dialog lists all files that will be moved to a backup directory (**/hvr_home/backup/***currentdate*) since they are not required or could be potentially dangerous in the upgraded version of HVR.

These files can include the old HVR installation files like shared **libs** or **scripts** and also the scripts/files added by the user.

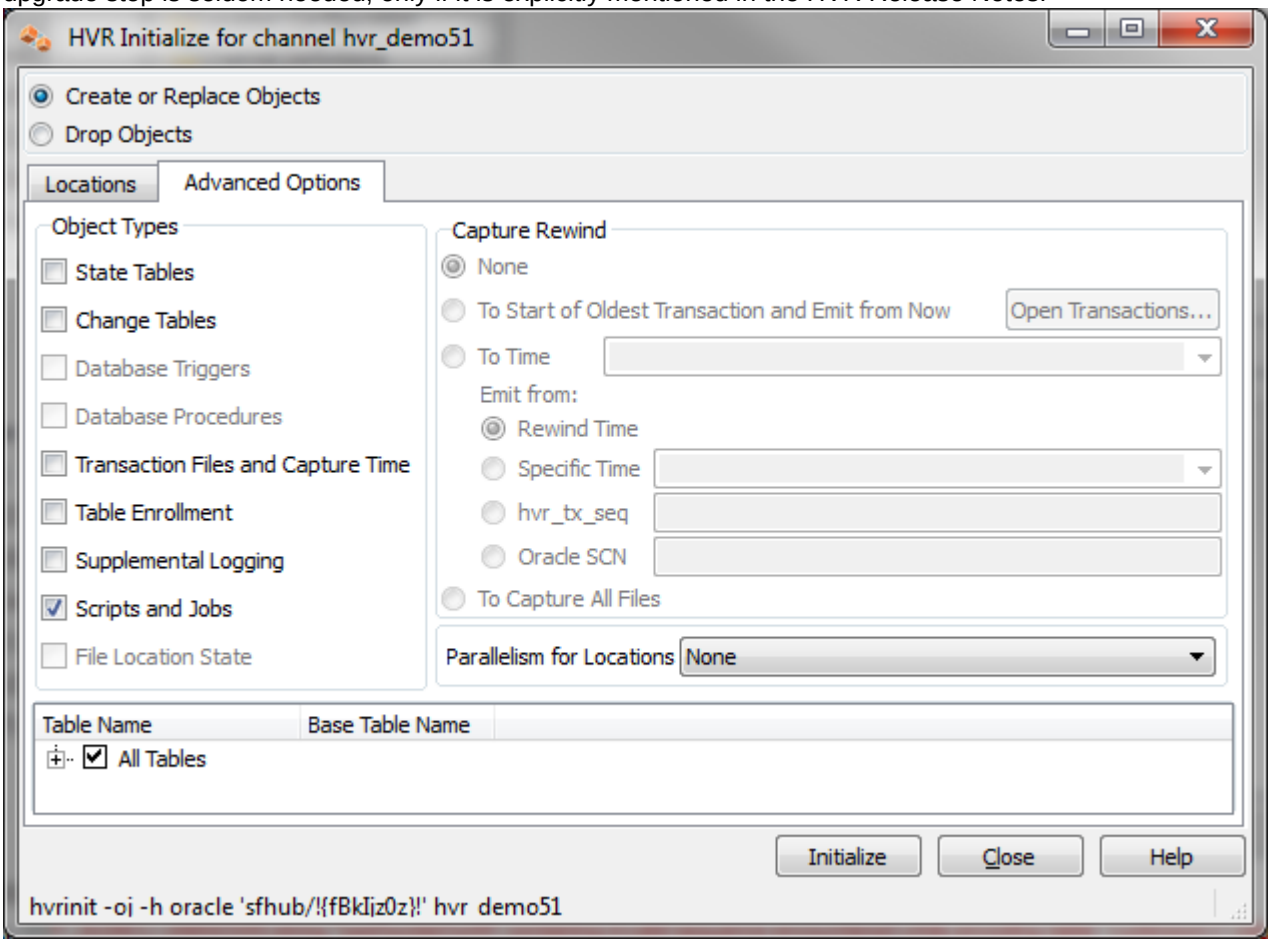
> HVR regards **HVR_HOME** as its private directory which should not contain user files.

In the warning dialog, click **Continue** to move the files to a backup directory (**/hvr_home/backup/***currentdate*).
If **Abort** is clicked, these files will not be moved to the backup directory. However, these files can be purged at a later time using the command **hvrstrip -P**. Note that executing **hvrstrip -P** does not create backup of the files being purged.

Following is a sample screenshot of the warning dialog:



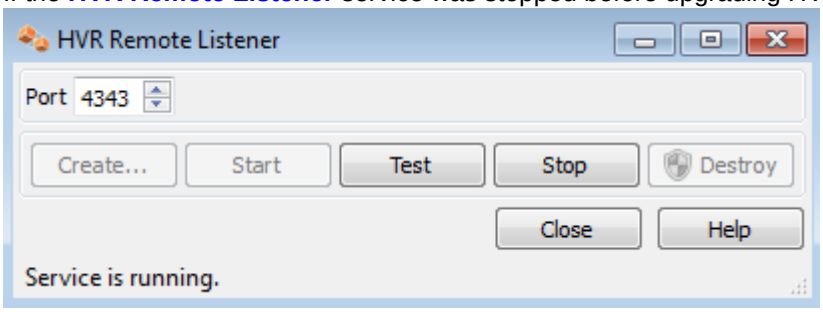
8. For certain bug fixes it may be necessary to regenerate the **Scripts and Jobs** for each channel. Note that this upgrade step is seldom needed; only if it is explicitly mentioned in the HVR Release Notes.



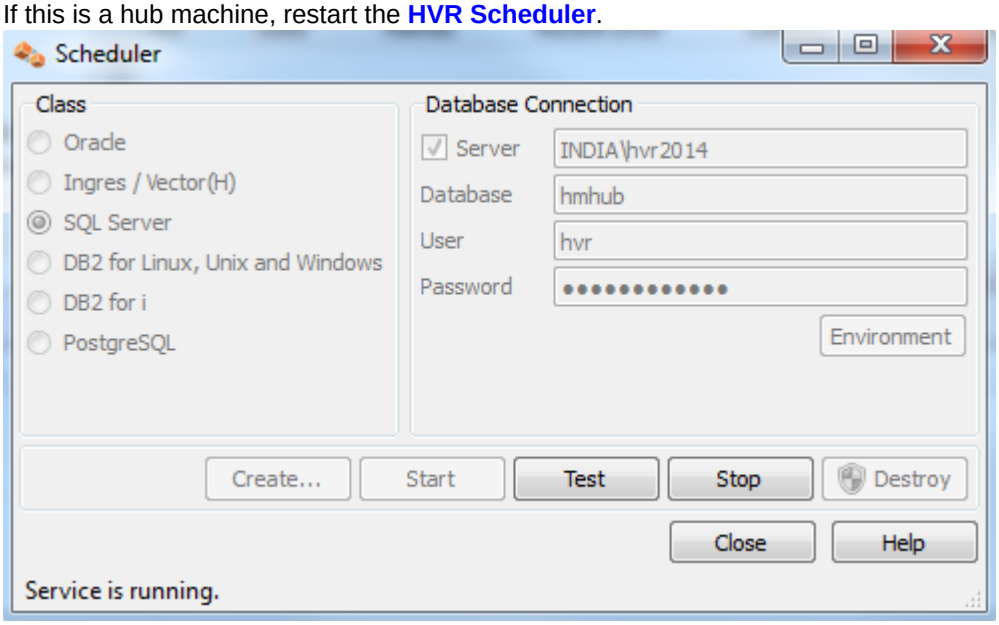
This can be done on the command line as follows:

```
hvrinit -oj hubdb/password channelname
```

9. If the **HVR Remote Listener** service was stopped before upgrading HVR, it must be restarted.



10. If this is a hub machine, restart the **HVR Scheduler**.



Downgrading HVR

Contents

This section provides a step-by-step instruction on how to downgrade HVR to an earlier version from which it was last upgraded. The primary goal of the downgrade procedure is to restore HVR quickly and safely to a known configuration in the previous version.

The procedure here assumes that:

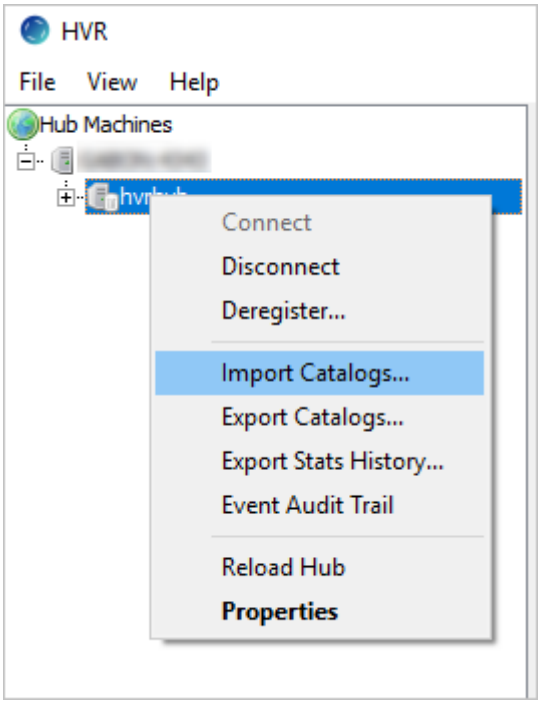
- a backup of the Hub database and the **HVR_HOME**, **HVR_CONFIG** directories were created before performing the HVR upgrade and
- in the upgraded HVR version, no capture or integrate jobs had successfully completed a replication cycle and **HVR Initialize** had not been run or had only been run with options **Table Enrollment** (**-oe**) and **Scripts and Jobs** (**-oj**) selected.

⚠ The procedure mentioned in this section is applicable only to immediately revert an HVR upgrade performed. Contact HVR Technical Support for the downgrade procedure in case HVR has processed changes after the upgrade or if an error is encountered during/after the downgrade.

Downgrading HVR Hub

To downgrade the HVR Hub, perform the following steps:

1. Stop **HVR Scheduler** and **HVR Remotelistener**.
2. Install the previous version of HVR.
3. Restore **HVR_HOME** and **HVR_CONFIG** directories from the backup.
4. Restore the Hub database from the backup. One of the following methods can be followed for restoring the hub database backup:
    - Restore the Hub database using the DBMS native restore option. This step is possible only if the backup was created using the DBMS native backup option. After restoring the Hub database, launch HVR GUI and connect to the Hub database.
    - Restore the Hub database (HVR catalogs) using the command **hvrcatalogimport**. This step is possible only if the backup was created using the command **hvrcatalogexport**.
      In HVR GUI,
        a. Launch HVR GUI and connect to the Hub database.
        b. Right-click the HVR hub and select **Import Catalogs...**



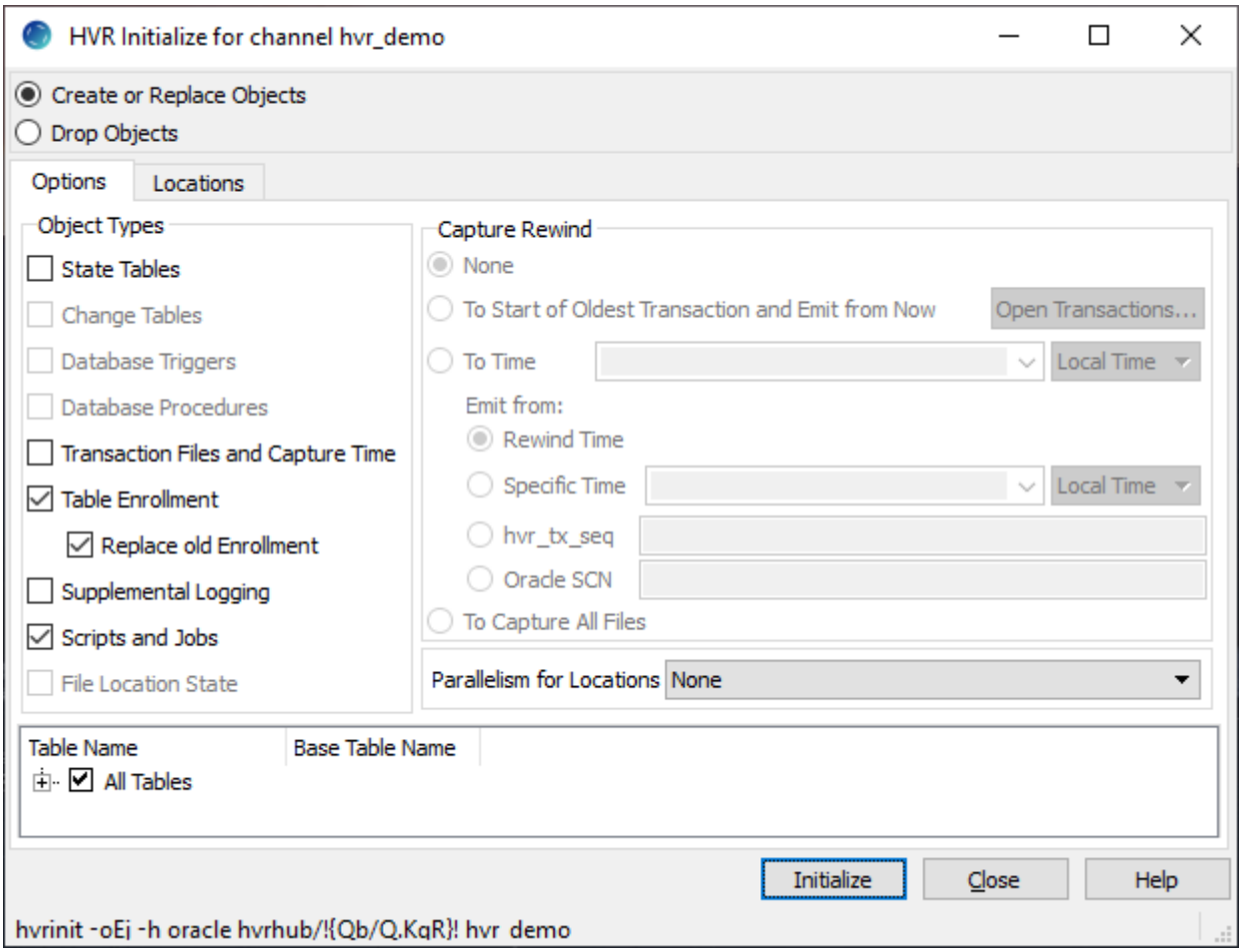
c. Select the catalog file (**.xml**) and click **Open**.
d. Click **OK** to confirm.

In CLI,

a. Execute the following command to import HVR catalog:

```
hvrcatalogimport hubdb catalogfile
```

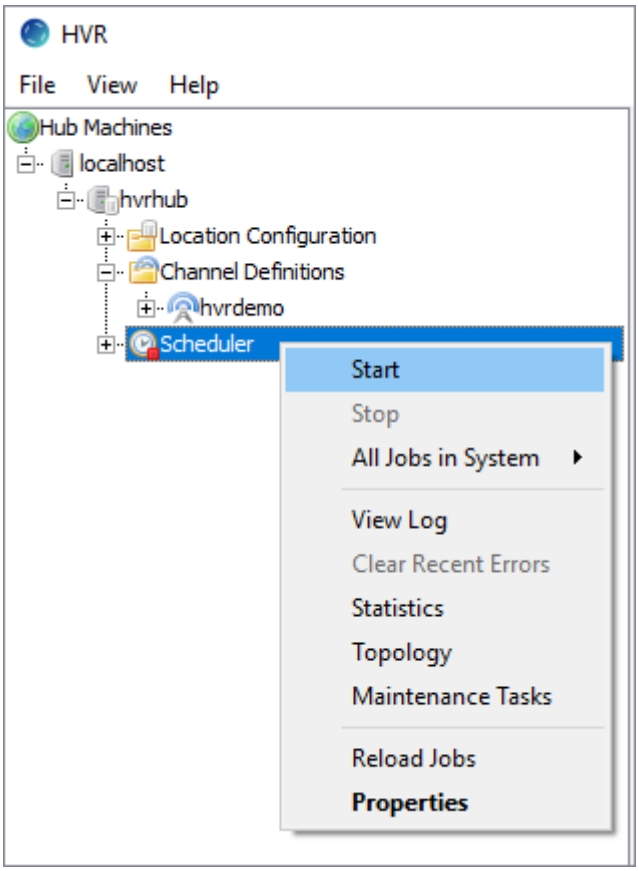
5. Execute **HVR Initialize** with options **Table Enrollment**, **Replace old Enrollment**, and **Scripts and Jobs** selected.



This can also be executed in the command line as:

```
hvrinit -oEj -h oracle hvrhub/hvrhub hvr_demo
```

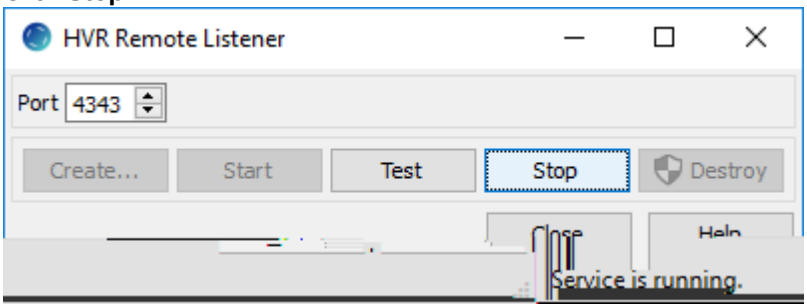
6. Start Replication.



Downgrading HVR Remote Agent (Capture and Integrate)

To downgrade (roll-back the upgrade) HVR remote agent, perform the following steps:

1. Stop the **HVR Remotelistener**, if running.
    - In Windows,
        a. Click **File** **HVR Remote Listener**.
        b. Click **Stop**.

        This can also be executed in the command line as:

        ```
        hvrremotelistener -ah 4343
        ```

    - In Unix/Linux,

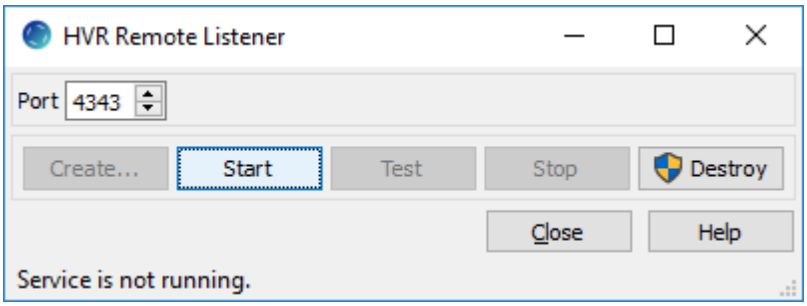
        Execute the following command to stop the **hvrremotelistener**:

        ```
        $ hvrremotelistener -k 4343
        ```

2. Install the previous version of HVR.

3. Start the **HVR Remotelistener**.
   - In Windows,

     a. Click **File** ▸ **HVR Remote Listener**.
     b. Click **Start**.

   - This can also be executed in the command line as:

     ```
     hvrremotelistener -as 4343
     ```

   - In Unix/Linux,

     Execute the following command to start the **hvrremotelistener**:

     ```
     $ hvrremotelistener -d 4343
     ```

Migrating HVR Hub to New Server

Contents

This section describes the procedural steps for an HVR administrator to migrate a current HVR Hub to a new server. It is assumed that moving the HVR Hub will not impact any **HVR Agent** installation or connection to any location.

The steps apply to migrating an HVR Hub from/to Windows or Linux/UNIX platforms. This procedure is for migration and not intended for HVR upgrade. Instructions for upgrading an HVR instance in place can be found on the Upgrading HVR page.

- This section does not address generating or using new SSL certificate information.
- This section does not address any Agent Plugin actions that reference a custom file path or any other environment variable that may reference a custom file path.

Scenario

At some point, an HVR administrator may need to upgrade the server, on which the HVR Hub is installed. The reason may be that the old hardware is no longer supported or the need to move to a server with an upgraded or different operating system or faster storage disks, and for a variety of other reasons. In this case, the administrator has to stop the replications services on the HVR Hub to move (aka migrate) all the replication objects (locations, channel definitions, jobs, schedules, and maintenance tasks) to a new server. The result is HVR replication services run on a new HVR Hub server without losing any transactional data during migration with minimal downtime.

Prerequisites

- Ensure you read the compatibility section to ensure that HVR is compatible with a new server you selected to be the HVR Hub.
- Ensure you download the HVR installation distribution from the HVR product download site for your specific operating system in advance of performing any steps to reduce the outage time during migration.
- Ensure you have already installed the database that will serve as the new HVR repository and created a schema /database/user that has the same name as the previous HVR repository prior to performing migration.

> ⚠ This section does not cover steps that involve renaming your repository schema/database/user during migration. Changing the owner of the HVR Catalogs impacts path statements in the router folders requiring additional steps not covered in this section.

- If installing the HVR Hub on a Windows server, ensure you have installed Perl.
- Review the requirements section for specific operating system (Windows or UNIX/Linux) for the new Hub.
- If you are moving the HVR Hub to Azure, ensure you read the section on Installing HVR on Azure using HVR Image.

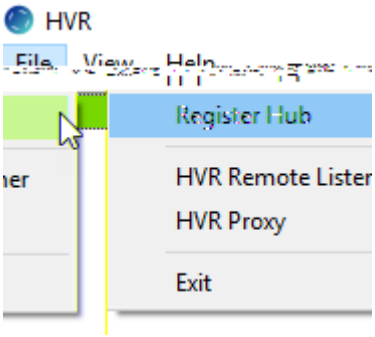
Prepare New HVR Hub

1. Install HVR on a new HVR Hub server.
2. Copy the HVR license file from the old server to the new server.
   - License files are located in the **HVR_HOME/lib** directory with a **.lic** file extension.
   - Some license files are generated with a unique HVR hub fingerprint matching your older server. If a license file contains a fingerprint, it will not be usable on the new server. To determine if your license file includes a fingerprint, view the contents of the file to see if it includes a line with # Hub fingerprint. Here is an example of a license file that includes a fingerprint:

```
# HVR license issued YYYY-MMM-DD
#
# Hub platform:      *
# Expiry date:       PERMANENT
# HVR features:      repl|cmp|refr|sapx
# Capture locations: <max and class>
# Integrate locations: <max and class>
# Hub fingerprint:   <HVR Hub Fingerprint>
```

> ⚠ While the default file name is **hvr.lic**, many sites have multiple license keys, so ensure you copy all files with the **.lic** file extension and transfer them to the new server to the **HVR_HOME/lib** directory.

   If your license file is bound to a unique server fingerprint, then you will need to obtain a new HVR license key. For more information on how to do this, refer to the 'HVR License File' section of the Installing and Upgrading HVR page.
3. Start the HVR Remote Listener on the new server.
4. Launch the **HVR GUI**. When you launch **HVR GUI** for the first time, the **Register Hub** window is displayed automatically. **Register Hub** can also be accessed from menu **File Register Hub**.

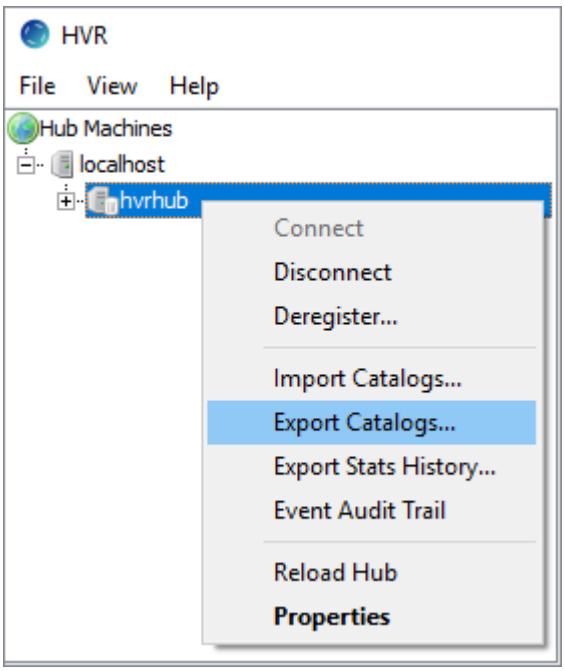


5. Enter the details to connect to your HVR Hub server, including:
   - HVR Remote Listener server node, port, and login information
   - Repository Class
   - Database Connection

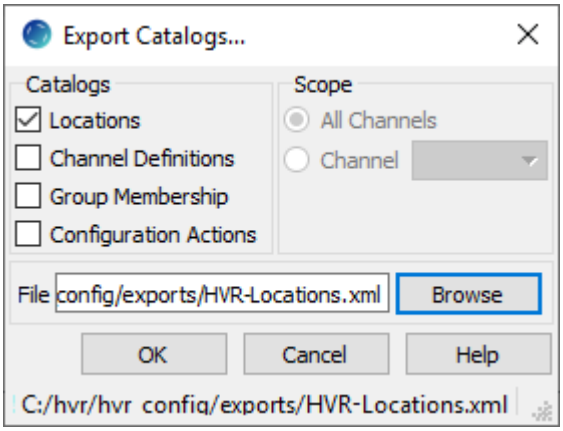
> ⚠ If the old hub connects to a remote database, you don't need to export/import catalogs after registering the new hub. In this case, firewalls relative to the new hub server must be opened to initiate connectivity.

Migrate HVR Location Definitions from Old HVR Hub to New HVR Hub

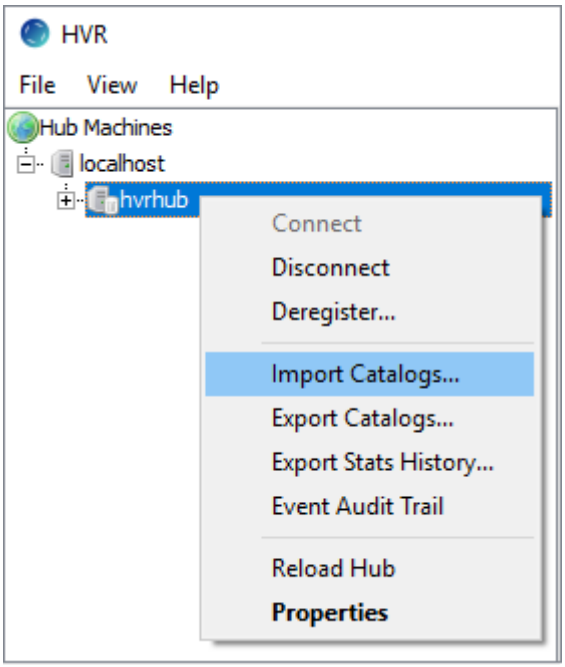
1. Export the location definitions: right-click the old HVR Hub and select **Export Catalogs**.



2. In the **Catalog** dialog, clear all the items except for **Locations**.
3. Use the **Browse** button to choose a folder and name the file to store the output. Click **OK**.



4. Import the location definitions: right-click the new HVR Hub and select **Import Catalogs**.



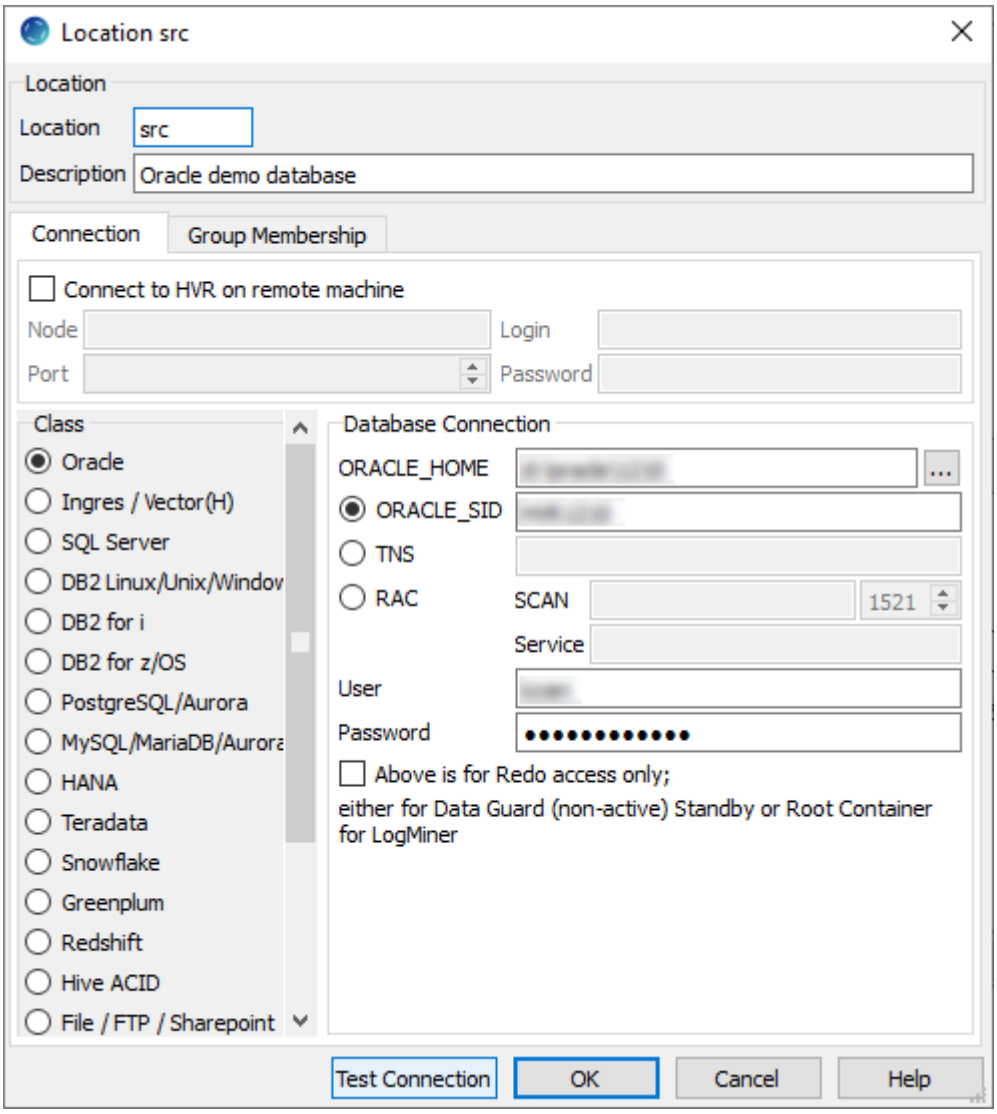
5. Select the XML file, to which you saved the exported location definitions and click **Open**.

Test Connection for All Locations in New HVR Hub

1. Right-click each location to view their **Properties**.

> ⚠ If any of the locations was previously defined on the old HVR Hub as a local connection, it may need to be updated to include the properties of the remote connection.

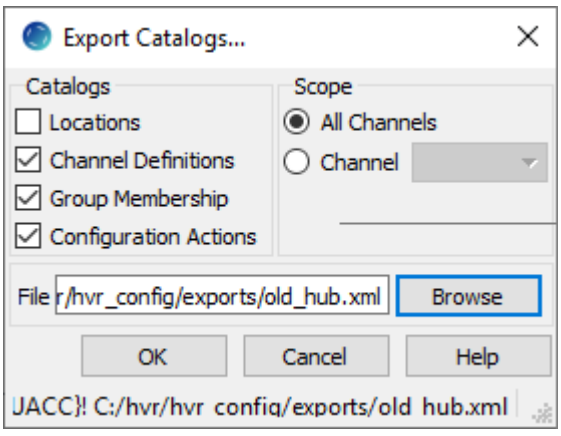
2. Click **Test Connection** and adjust the location properties if needed.



3. Repeat this step for all connections until all connection tests are successful.

Migrate Remaining Catalog Definitions from Old HVR Hub to New HVR Hub

1. Export the location definitions: right-click the old HVR Hub and select **Export Catalogs**.
2. In the **Catalog** dialog, clear the **Locations** item and leave the **Channel Definitions**, **Group Membership**, and **Configuration Actions** items for **All Channels** selected.
3. Use the **Browse** button to choose a folder and name the file to store the output. Click **OK**.



4. Import the catalog definitions: right-click the new HVR Hub and select **Import Catalogs**.

5. Select the XML file, to which you saved the exported catalog definitions and click **Open**.
6. Verify that your entire HVR Catalog has been imported to the new HVR Hub server.

> ⚠ The **hvr_stats** and **hvr_event** catalogs are not part of the catalog being exported/imported.

Shut Down Old Hub

There are two methods to consider when shutting down the old HVR Hub to transfer all of the stateful information to the new HVR Hub server.

Method 1: Stop the entire **HVR Scheduler**, which subsequently stops all replication jobs under the **Scheduler**. The advantage of the first method is that all jobs complete in one step, so you minimize the time to move files to the new HVR Hub server.

Method 2: First stop all the capture jobs and wait until all the integrate jobs have completed. The advantage of the second method is that you don't have to archive as many unprocessed HVR transaction files, which can sometimes contain hundreds of files.

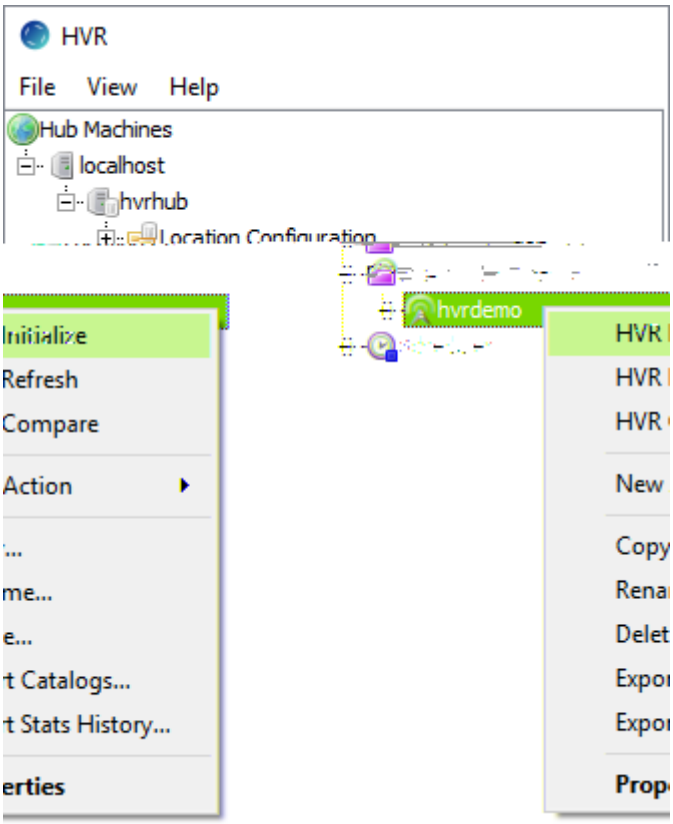
1. Stop the entire **HVR Scheduler,** thereby stopping all the jobs defined under the **HVR Scheduler**.
2. Alternatively, stop all the capture jobs first, wait for all the integrate jobs to drain all pending changes, then stop all the integrate jobs, and after that stop the entire **HVR Scheduler**.
3. Close all the **HVR GUI** dialog windows associated with the old HVR Hub server.

Migrate Stateful Information from Old HVR Hub to New HVR Hub Server

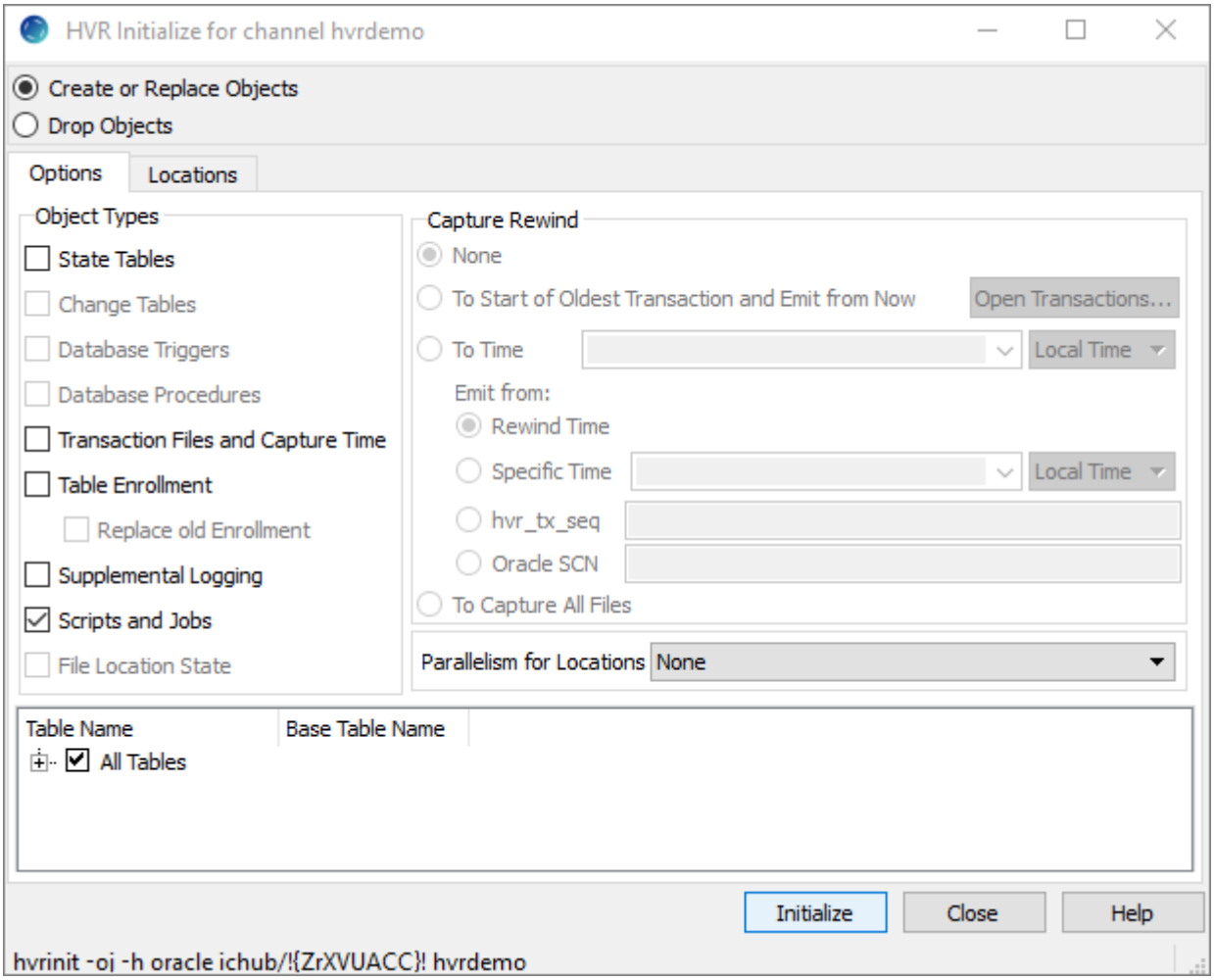
1. Archive the entire **$HVR_CONFIG** directory and its sub-directories from the old HVR Hub server.
2. Transfer the archived file from the old HVR Hub server to the new HVR Hub server.
3. Unzip the entire **$HVR_CONFIG** to the parent */hvr* directory on the new HVR Hub server.

Deploy and Start New HVR Hub

1. Launch the **HVR GUI** and connect to the new HVR Hub repository. For all channels, you must run **HVR Initialize** to deploy local runtime objects, including scripts and jobs.
2. Right-click the channel definition and select **HVR Initialize**.

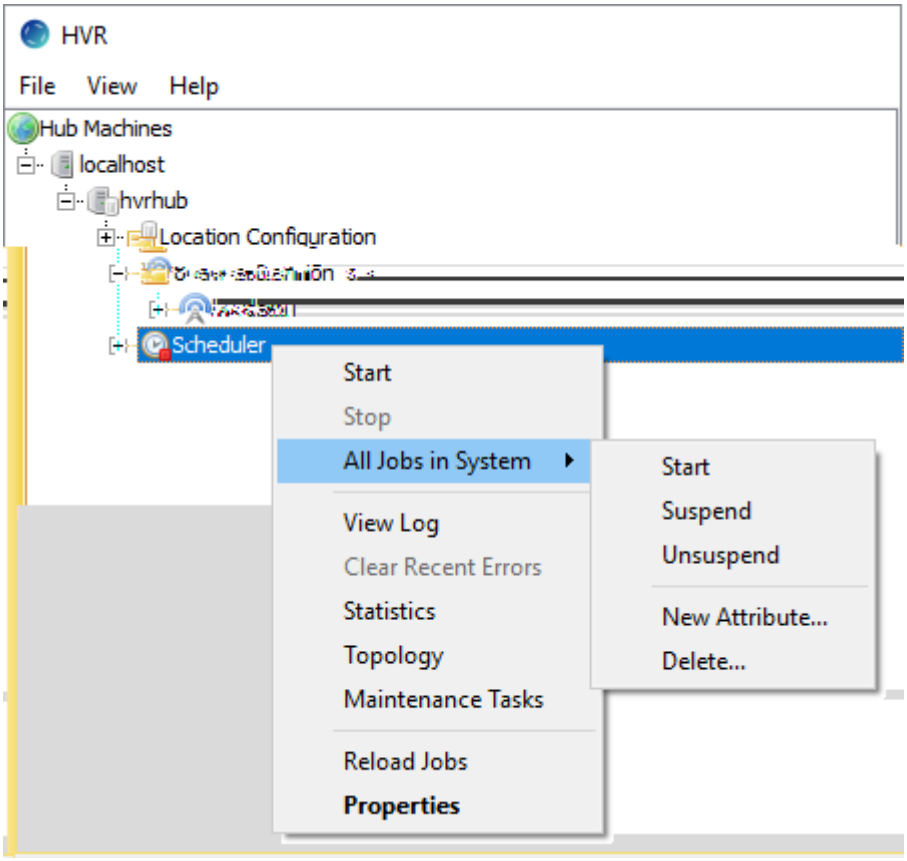


3. In the **Object Types** box, select **Scripts and Jobs**, leaving all other objects unselected, and click **Initialize**.
4. Repeat these steps for all channels.



5. Start the **HVR Scheduler**.

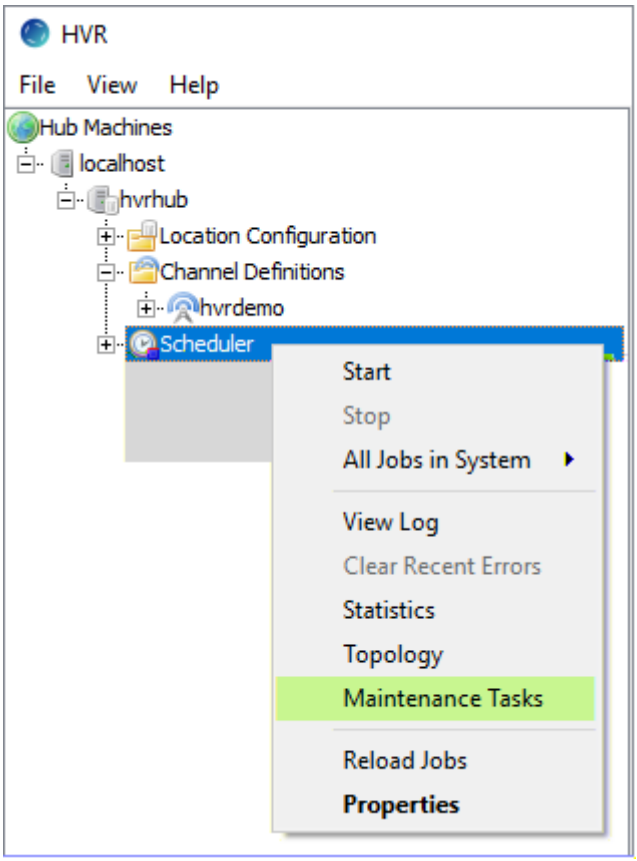
6. Right-click the **HVR Scheduler** and select **All Jobs in System Start**. This will start all the replication jobs under the **HVR Scheduler**.



Run Maintenance Tasks

You can configure maintenance tasks to manage and maintain the replication environment once the system goes live. HVR will send alerts as needed by email if errors are detected.

1. Right-click the **HVR Scheduler** and select **Maintenance Tasks**.
2. In the **Maintenance Tasks** dialog, schedule and run maintenance tasks as needed.



Upgrading from HVR Version 4 to 5

Upgrading installations can be a large undertaking, especially when it involves downtime or lots of machines. For this reason different HVR versions are typically fully compatible with each other; it is not necessary to upgrade all machines in a channel at once. Also HVR 4 and HVR 5 can connect to each other, for details see Changes to HVR Version 5 When Upgrading from HVR Version 4.

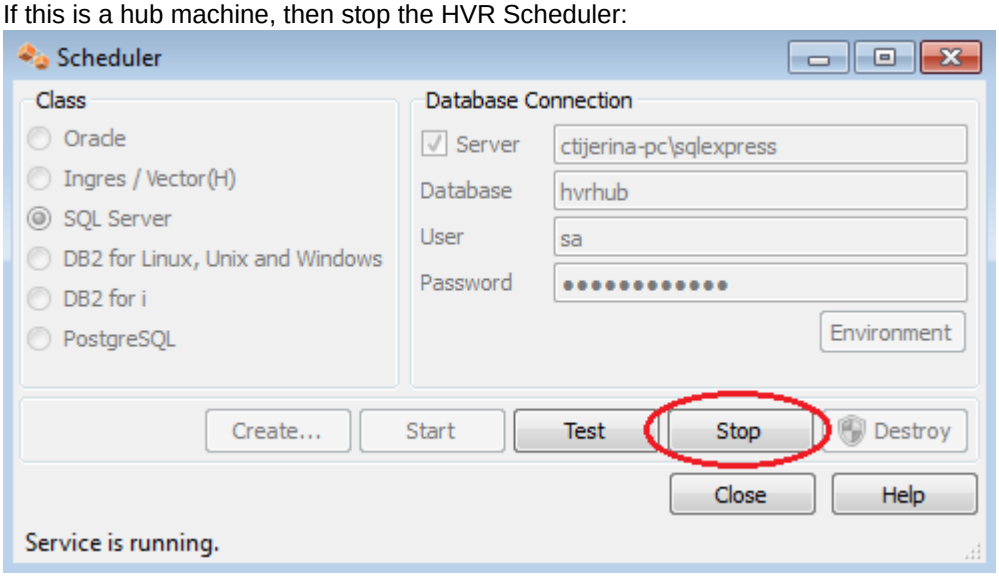
In many environments all machines will just be upgraded at the same time.

It is also possible to only upgrade certain machines (e.g. only the hub machine, GUI, remote source or target). If this is done, it should be understood that each HVR release fixes some bugs and or contains new features. Each fix or feature is only effective if the correct machine is upgraded. For example, if a new HVR release fixes an integrate bug, then that release must be installed on the machine(s) which do integrate. If only the GUI and/or hub machine is upgraded, then there will be no benefit. Read the HVR Release Notes (in **$HVR_HOME/hvr.rel**) for a description of which features and fixes have been added, and which machine must be upgraded for each feature and fix to be effective. New features should not be used until all machines that are specified for that feature are upgraded, otherwise errors can occur.

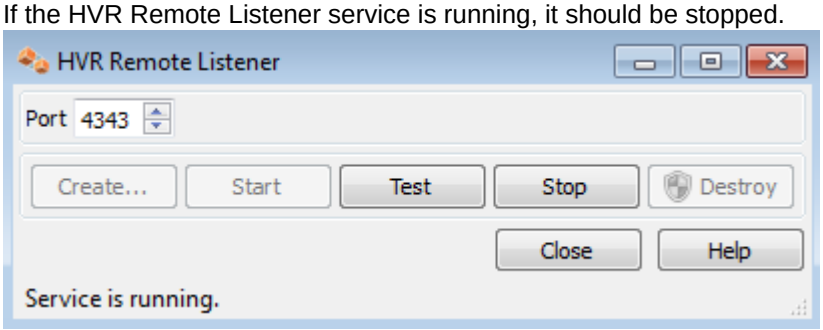
1. If a target location is upgraded from HVR version 4 to HVR 5 the replication needs to be flushed.
   Suspend the capture jobs, wait until the integrate jobs have finished integrating all the changes (so no transaction files are left in the router directory) and then suspend them too.

   ```
   $ hvrsuspend hubdb chn-cap
   $ hvrstart -w hubdb chn-integ
   $ hvrsuspend hubdb chn-integ
   ```

2. If this is a hub machine, then stop the HVR Scheduler:



3. If the HVR Remote Listener service is running, it should be stopped.



4. Stop all HVR GUI processes.
5. On AIX, uninstall the old HVR runtime executables using command (only for Unix & Linux):

   Unix & Linux

```
$ rm $HVR_HOME/lib/*
```

Alternatively, remove all cached shared libraries from the kernel by performing command **slibclean** as **root**. This step is not needed for other flavors of Unix or Linux.

6. Read and uncompress the distribution file.

Unix & Linux

In Unix and Linux this is done by:

```
$ cd $HVR_HOME
$ umask 0
$ gzip -dc /tmp/hvr-5.0.3-linux_glibc2.5-x64-64bit.tar.gz | tar xf -
```

On Microsoft Windows this is performed using the installation wizard:

```
C:\> d:\hvr-5.0.3-windows-x64-64bit-setup.exe
```

7. If HVR must perform log–based capture from Ingres, it needs a trusted executable to read from the DBMS logging system. Perform the following steps while logged in as **ingres**:

Ingres

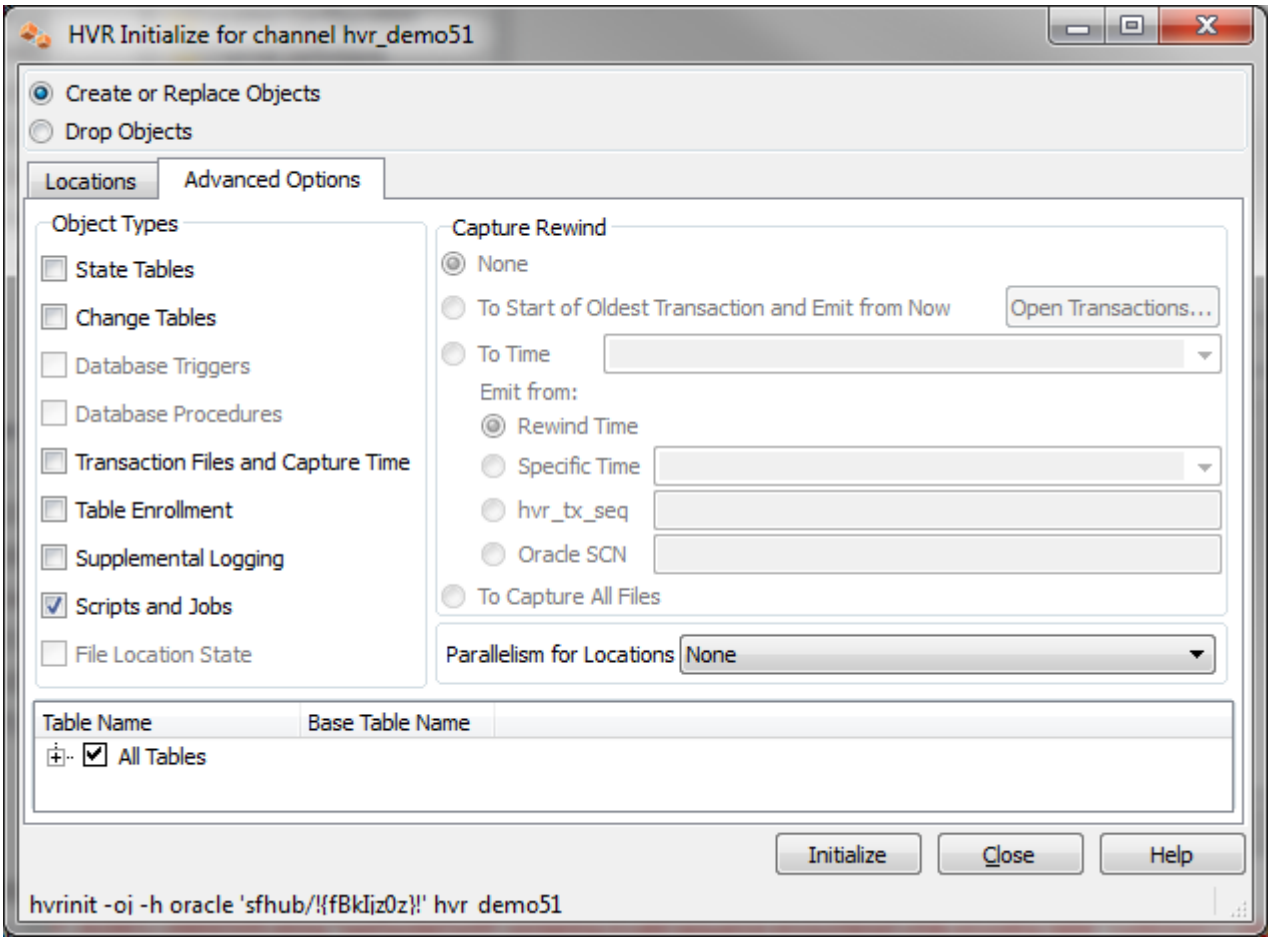
```
$ cd /usr/hvr/hvr_home
$ cp bin/hvr sbin/hvr_ingres
$ chmod 4755 sbin/hvr_ingres
```

Additionally, on Linux the trusted executable needs to be patched using

```
$ /usr/hvr/hvr_home/lib/patchelf --set-rpath /usr/hvr/hvr_home/lib --force-rpath /usr
/hvr/hvr_home/sbin/hvr_ingres
```

These steps are not needed if capture will be trigger–based or if capture will be from another machine.
If HVR and **ingres** share the same Unix group, then the permissions can be tightened from 4755 to 4750.

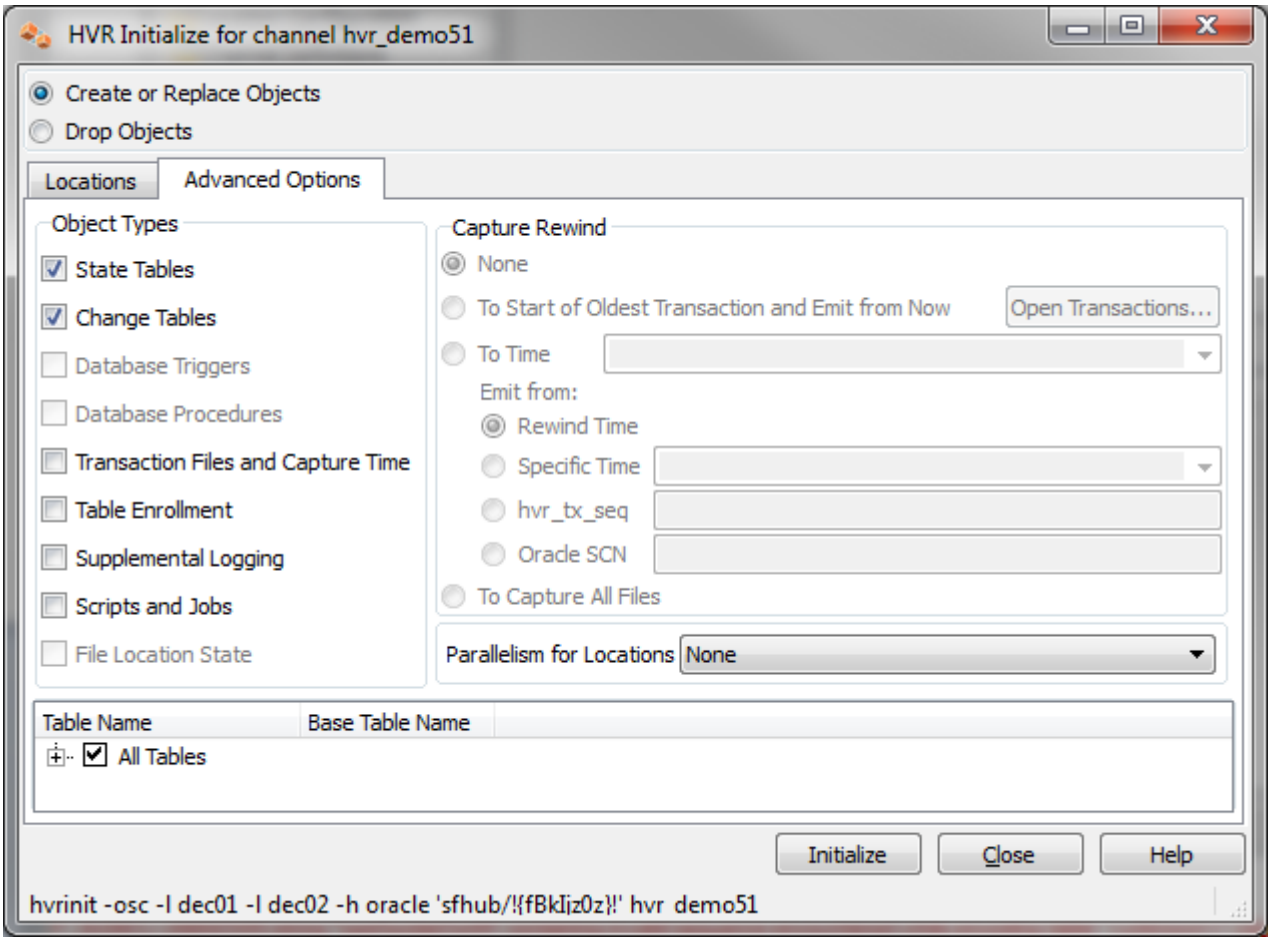
8. If the hub machine is being upgraded, then the HVR version 4 actions need to be upgraded to HVR 5 actions. This can be done automatically by connecting with a new version 5 HVRGUI to this hub installation; A pop-up will appear to change the HVR 4 definitions to HVR 5.
9. If the hub machine is being upgraded, the data type information in the HVR catalogs need to be reloaded. For each channel use the HVR GUI's **Table Explore** to improve this information: Click **Replace** for all tables that show differences. This step is optional, but if it is omitted and a subsequent HVR Refresh is instructed to recreate mismatched tables it could create target tables with incorrect data types.
10. If the hub machine is being upgraded, it is necessary regenerate the job scripts for all channels.



This can be done on the command line as follows:

```
$ hvrinit -oj <hubdb>/<password> <channel name>
```

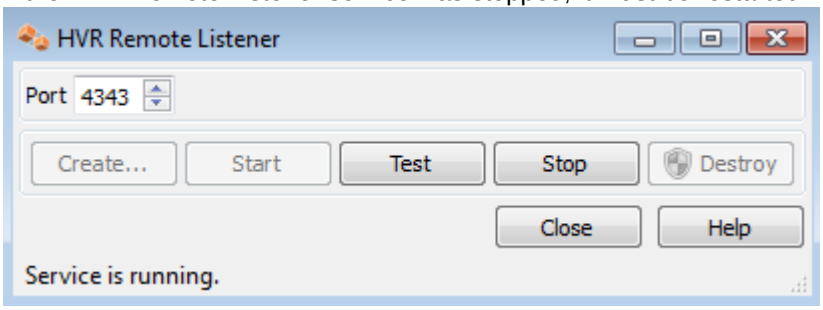
11. If a target machine was upgraded from HVR version 4 to HVR 5, the state tables need to be regenerated. If Integrate /Burst is used, the burst tables need to be recreated as well.



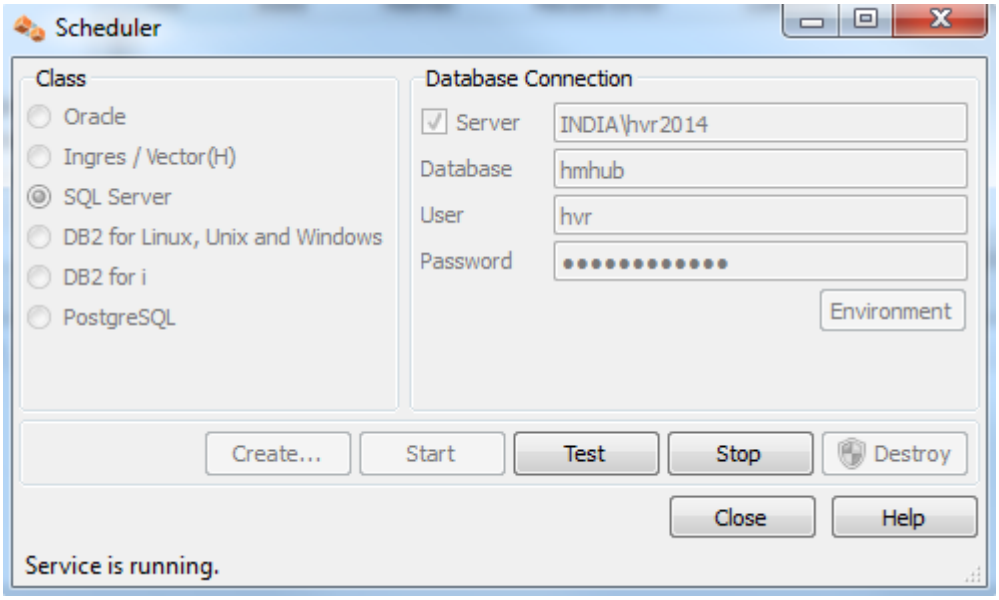
This can be done on the command line as follows (for each target location):

```
$ hvrinit -ocs -l <target location> <hubdb>/<password> <channel name>
```

12. If the HVR Remote Listener service was stopped, it must be restarted.



13. If this is a hub machine, restart the HVR Scheduler.



14. If the replication jobs were stopped, restart the replication

```
$ hvrstart hubdb chn-cap
$ hvrstart hubdb chn-integ
```

Changes to HVR Version 5 When Upgrading from HVR Version 4

Contents

HVR version 5 contains various changes to HVR version 4. This includes changes to HVR commands and actions. There are also limitations to HVR's network compatibility; the ability of HVR on one machine to talk to HVR on a different machine. When upgrading from HVR version 4 to HVR 5 additional steps are necessary. Fore more information, see Upgrading from HVR Version 4 to 5.

If a new (HVR version 5) GUI detects old (HVR version 4) actions in a new (version 5) hub machine, it will suggest that these actions be renamed. A popup assistant will do this renaming.

Changed Commands and Renamed Dialogs

The following HVR commands and dialogs are renamed between HVR version 4 and HVR version 5.

- Command **hvrload** (dialog **HVR Load**) is renamed to **Hvrinit** (dialog **HVR Initialize**).
- Command **hvrtrigger** (right-click **Trigger**) is renamed to **Hvrstart** (right-click **Start**)
- Dialog **Table Select** is renamed to **Table Explore**.
- **HVR Refresh** checkbox **Unicode Datatypes** (**Hvrrefresh** option **-cu**) has moved to **TableProperties /CreateUnicodeDatatypes**.

Changed Actions and Action Parameters

Various actions and parameters are renamed and/or redesigned. The major changes are;

- Actions **DbCapture**, **FileCapture** and **SalesforceCapture** are united into a single action called **Capture**.
- Actions **DbIntegrate**, **FileIntegrate** and **SalesforceIntegrate** are united into a single action called **Integrate**.
- Action **Transform** is redesigned; some functionality has moved to new action **FileFormat** (see parameters **/CaptureConverter** and **/IntegrateConverter**) while other functionality moved to **ColumnProperties /SoftDelete**.
- Action **Agent** is renamed to **AgentPlugin** because HVR's remote executable is also called 'agent'.

The following is a complete list of these changes:

Old (HVR 4) Actions and Parameters	New (HVR 5) Actions and Parameters
Action **DbCapture**	Action **Capture**. Remove parameter /**LogBased** (new default). Add /**TriggerBased** added if /**Log Based** is NOT defined. Other parameters are copied.
Actions **FileCapture** and **SalesforceCapture**	Action **Capture**
Actions **DbIntegrate FileIntegrate** and **SalesforceIntegrate**	Action **Integrate**

Parameters /**SupplementalLogsPK** and /**SupplementalLogsNoPK** of **DbCapture**	Parameter /**SupplementalLogging** of **Capture**. • If /**SupplementalLogsPK=CDC_TAB** and /**SupplementalLogsNoPK=CDC_TAB** then set /**SupplementalLogging=CDCTAB**. • If /**SupplementalLogsPK=ARTICLE** and /**SupplementalLogsNoPK=CDC_TAB** then set /**SupplementalLogging=ARTICLE_OR_CDCTAB**. • If /**SupplementalLogsPK** is not set and /**SupplementalLogsNoPK=CDC_TAB** then set /**SupplementalLogging=ARTICLE_OR_CDCTAB**. Other combinations are no longer supported.
Parameter /**NoTranLogTruncate** of **DbCapture**	Parameter /**LogTruncate=NATIVE_DBMS_AGENT** of **Capture**.
Parameter /**Journal** of **DbIntegrate**, **FileIntegrate** or **SalesforceIntegrate**	Parameter /**JournalRouterFiles** of **Integrate**
Parameters /**BulkAPI** and/or /**SerialMode** of *SalesforceIntegrate*	Parameters /**BulkAPI** and/or /**SerialMode** of **LocationProperties**
Parameters /**Resilient**, /**ResilientInsert**, /**ResilientUpdate**, /**ResilientDelete** and /**ResilientWarning** of **DbIntegrate**	**Integrate /Resilient=WARNING** if /**ResilientWarning** was defined, **Integrate /Resilient=SILENT_DELETES** if /**ResilientDeletes** was defined, otherwise /**Resilient=SILENT**.
Parameter /**DbProc** and /**DbProcDuringRefresh** of **DbIntegrate**	**Integrate /DbProc**. Parameter /**Context=|refr** is added if /**DbProcDuringRefresh** was not defined in HVR Version 4. ⚠ In HVR Version 5 refresh will by default use **/DbProc** if action **Integrate /DbProc** is set for specific tables unless refresh is run with context refr.
Parameter /**HvrOpField** of **FileIntegrate**.	Removed. Use **ColumnProperties /Name=hvr_op_val /Extra /IntegrateExpression={hvr_op}**
Parameter /**ConvertNewlinesTo** of **FileIntegrate**	**FileFormat /ConvertNewlinesTo**
Parameter /**DistributionKey** of **ColumnProperties**	This still exists, by the distribution key can now be set directly in **Table Explore** dialog instead. This is recommended.
Parameter /**RefreshWithGroup** of **Restrict**	Removed. Use **Restrict /Context** instead

Transform /Rows2Xml	Either remove (XML is default for integrating tables into a file location) or explicitly add action **FileFormat /Xml**. Options in /**ArgumentValue** should be replaced with parameters as follows; • **-n** (retired) • **-q** -> /**Compact**
Transform /Rows2Csv or /**Builtin=Xml2Csv** or /**Builtin=Csv2Xml**	**FileFormat /Csv** Options in /**ArgumentValue** should be replaced with parameters as follows; • **-c** -> /**Encoding** • **-h** -> /**Headerline** • **-f** -> /**FieldSeparator** • **-l** -> /**LineSeparator** • **-q** -> /**QuoteCharacter** • **-e** -> /**EscapeCharacter** • **-z** -> /**FileTerminator** • **-s** quote style has been retired. This feature is now activated by setting /**QuoteCharacter** and/or /**EscapeCharacter**. Specific options for /**Builtin=Xml2Csv**; • **-F** (retired) Specific options for /**Builtin=Csv2Xml**; • **-n** Use **Environment /Name=HVR_CSV2XML_EMPTY_IS_NULL /Value=1** • **-t** Use **Capture /Pattern** containing **{hvr_tbl_name}** or set **Environment /Name=HVR_TBL_NAME /Value=***tbl_name*
Transform /ArgumentValue.	If /**Command** is checked then use **CommandArgument** instead of **/ArgumentValue**
Transform /Command on file capture location to reformat input.	If /**Command** is **hvrcsv2xml.pl** then convert to **FileFormat /Csv** (see above). Otherwise **FileFormat /CaptureConvertor**
Transform /Command on file integrate location to reformat output.	If /**Command** is **hvrxml2csv.pl** then convert to **FileFormat /Csv** (see above). Otherwise **FileFormat /IntegrateConvertor**
Transform /Builtin=SoftDelete	**ColumnProperties /Extra /Name={colname} /SoftDelete /Datatype={datatype}**
Transform /Builtin=File2Column	**FileFormat /CaptureConvertor=hvrfile2column.pl**
Transform /Builtin=SapAugment	**Transform /SapAugment**
Transform /Builtin=SapXForm	**Transform /SapXForm** with /**SapTableListFile** and /**SapMetadataDirectory**
Transform /Builtin=Tokenize	**Transform /Command=hvrtokenize.pl**
Transform /XsltStyleSheet	XSLT transform feature is no longer supported.

Action **Agent**	Action renamed to **AgentPlugin**. All parameters are the same
Parameter /**SslCertificate** of **LocationProperties**	Parameter /**SslRemoteCertificate** of **LocationProperties**
Parameters /**CaptureOnceOnTrigger** and /**IntegrateOnceOnTrigger** of **Scheduling**	Parameters /**CaptureOnceOnStart** and /**IntegrateOnceOnStart** of **Scheduling**

Configuring HVR

This section provides information on configuring HVR after installation.

Auto-Starting HVR Scheduler after Unix or Linux Boot

Contents

This section describes about how to enable automatic restart of **HVR Scheduler** when a Unix or Linux machine is rebooted.

Based on the daemon type available in the server where HVR is installed, one of the following configuration methods can be used for enabling automatic restart of HVR processes.

Configuring Systemd

On a Unix/Linux server with **systemd**, create a service to enable **HVR Scheduler** to auto-start after a system reboot or service failure.

The following steps should be performed as user **root** to configure **systemd**:

Create a unit file **hvrscheduler.service** in **/etc/systemd/system** directory. The unit file should contain the following:

```
[Unit]

Description=HVR Scheduler Service

# Database service is only needed if hub database is on the same system as hvrscheduler

After=network.target hvr.socket database.service


[Service]
```

1. The **HVR Scheduler** service should start only after the Hub's database service (*database*.**service**) has started. Otherwise, after reboot the **HVR Scheduler** will fail immediately while trying to connect to the hub database.
2. The DB connection string syntax (in **ExecStart**) differs for each database. For more information about DB connection string syntax in HVR commands, see Calling HVR on the Command Line (*channel* is not required in the DB connection string (in **ExecStart**) to start **HVR Scheduler**).

1. To enable and start the service, execute the following commands:

```
systemctl enable hvrscheduler

systemctl start hvrscheduler
```

2. To verify whether the service is active, execute the following command:

```
systemctl status hvrscheduler
```

Sample output:

```
hvrscheduler.service - HVR Scheduler

 Loaded: loaded (/etc/systemd/system/hvrscheduler.service; disabled; vendor preset:
disabled)

 Active: active (running) since Mon 2020-02-17 10:03:18 EST; 14min ago

 Process: 7587 ExecStart=/home/hvruser/hvr/hvr_home/bin/hvrscheduler -horacle myhubdb
/mypwd mychannel

 Main PID: 7588 (hvrscheduler)

 CGroup: /system.slice/hvrscheduler.service

           7588 hvrscheduler -horacle myhubdb/mypwd mychannel
           7589 hvrscheduler -swork -horacle myhubdb/mypwd mychannel
           7591 hvr hvrstats -h mysql -horacle myhubdb/mypwd mychannel
           7592 hvr ora_mysql-cap-ora
           7593 hvr -shvrstats-1
           7594 hvr -shvrstats-1
```

Configuring Init

For a Unix/Linux server without **systemd**, **xinetd**, or **inetd**, use HVR's script file **hvr_boot** (available in **hvr_home/lib**) and a user-defined configuration file **hvrtab** to enable **HVR Scheduler** to auto-start after a system reboot.

The script file **hvr_boot** allows you to start and stop HVR processes (both **HVR Scheduler** and **HVR Remote Listener**) defined in the configuration file **hvrtab**. The script file **hvr_boot** should only be executed by the **root** user.

The following steps should be performed as user **root** to configure **init**:

1. Create the configuration file **hvrtab** in the **/etc** directory. Each line of this configuration file should contain four or more parameters separated by a space in the following format:

   username port_or_hub hvr_home hvr_config [*options*]

   - *username*: Indicates the Unix/Linux username under which HVR runs.
   - *port_or_hub*: Indicates a TCP/IP port number (for **HVR Remote Listener**) or the username and password of the hub database (for **HVR Scheduler**). The DB connection string syntax for the hub database differs for each database. For DB connection string syntax, see Calling HVR on the Command Line.
   - *hvr_home*: Indicates the path for **$HVR_HOME**.
   - *hvr_config*: Indicates the path for **$HVR_CONFIG**.
   - [options]: Indicates an optional parameter to pass other options to the HVR process. For more information about the options, see **hvrremotelistener** and **hvrscheduler**.

   Sample configuration file:

   ```
   # This hvrtab file starts a Remote Listener and two HVR schedulers (one for an Oracle hub and one for an Ingres hub).

   # The Oracle password is encrypted. For more information, see documentation for command hvrcrypt.

   root 4343 /home/hvruser/hvr/hvr_home /home/hvruser/hvr/hvr_config

   mylinuxuser orahubdb/!{DszmZY}! /home/hvruser/hvr/hvr_home /home/hvruser/hvr/hvr_config -EHVR_TMP=/home/hvruser/hvr/hvr_tmp -EORACLE_HOME=/opt/oracle -EORACLE_SID=prod

   mylinuxuser inghubdb /home/hvruser/hvr/hvr_home /home/hvruser/hvr/hvr_config -EHVR_TMP=/home/hvruser/hvr/hvr_tmp -EII_SYSTEM=/opt/ingres -EHVR_PUBLIC_PORT=50001
   ```

   Lines starting with a hash (#) are treated as comments.

2. Copy the script file **hvr_boot** (available in **hvr_home/lib**) to the **init.d** directory and create symlinks to the **rc.d** directory.

   > For an Oracle RAC, the script file **hvr_boot** can be enrolled in the cluster with command **crs_profile**.

   > The **HVR Scheduler** service should start only after the Hub's database service has started. Otherwise, after reboot the **HVR Scheduler** will fail immediately while trying to connect to the hub database.
   >
   > For non-Solaris machines, this can achieved using the start and stop sequence number in the boot filename. The start sequence of **hvr_boot** must be bigger than the start sequence of the DBMS service, and the stop sequence must be smaller than the stop sequence of the DBMS.
   >
   > For Solaris SMF, this can achieved by editing the file **hvr_boot.xml** and replacing the string **svc:/milestone/multi–user–server** with the name of the DBMS service (e.g. **svc:/application/database/oracle**).

The following example uses start sequence **97** and stop sequence **03** (except HP–UX which uses **997** and **003** because it requires three digits).

- On AIX, to start and stop HVR for run level 2:

```
$ cp hvr_boot /etc/rc.d/init.d
$ ln -s /etc/rc.d/init.d/hvr_boot /etc/rc.d/rc2.d/S97hvr_boot
$ ln -s /etc/rc.d/init.d/hvr_boot /etc/rc.d/rc2.d/K03hvr_boot
```

- On HP–UX, to start and stop HVR for run level 3:

```
$ cp hvr_boot /sbin/init.d
$ ln -s /sbin/init.d/hvr_boot /sbin/rc3.d/S997hvr_boot
$ ln -s /sbin/init.d/hvr_boot /sbin/rc3.d/K003hvr_boot
```

- On Linux, to start HVR for run levels 3 and 5 and stop for all run levels:

```
$ cp hvr_boot /etc/init.d
$ ln -s /etc/init.d/hvr_boot /etc/rc.d/rc3.d/S97hvr_boot
$ ln -s /etc/init.d/hvr_boot /etc/rc.d/rc5.d/S97hvr_boot
$ ln -s /etc/init.d/hvr_boot /etc/rc.d/rc0.d/K03hvr_boot
$ ln -s /etc/init.d/hvr_boot /etc/rc.d/rc1.d/K03hvr_boot
$ ln -s /etc/init.d/hvr_boot /etc/rc.d/rc2.d/K03hvr_boot
$ ln -s /etc/init.d/hvr_boot /etc/rc.d/rc3.d/K03hvr_boot
$ ln -s /etc/init.d/hvr_boot /etc/rc.d/rc4.d/K03hvr_boot
$ ln -s /etc/init.d/hvr_boot /etc/rc.d/rc5.d/K03hvr_boot
$ ln -s /etc/init.d/hvr_boot /etc/rc.d/rc6.d/K03hvr_boot
```

- On Solaris 8 or 9, to start and stop HVR for run level 2 (which implies level 3):

```
$ cp hvr_boot /etc/init.d
$ ln -s /etc/init.d/hvr_boot /etc/rc2.d/S97hvr_boot
$ ln -s /etc/init.d/hvr_boot /etc/rc2.d/K03hvr_boot
```

- On Solaris 10 and higher;

> For newer Solaris versions, the script file **hvr_boot** must be registered in the Solaris's System Management Facility (SMF). Also, the file **hvr_boot.xml** (available in **hvr_home/lib**) should be copied to the **init.d** directory.

```
$ cp /opt/hvr/hvr_home/lib/hvr_boot /lib/svc/method
$ cp /opt/hvr/hvr_home/lib/hvr_boot.xml /var/svc/manifest/application

$ svccfg
svc> import /var/svc/manifest/application/hvr_boot.xml
svc> quit

$ svcadm enable svc:/application/hvr_boot
$ svcs -a|grep hvr     # To check if the service is running
16:00:29 svc:/application/hvr_boot:default
```

Auto-Starting HVR after Windows Boot

Contents

This section describes the steps to enable the automatic restart of HVR processes (**HVR Remote Listener** and **HVR Scheduler**) when a Windows machine is rebooted. This section also describes about automatically restarting **HVR Scheduler** service in case of a failure.

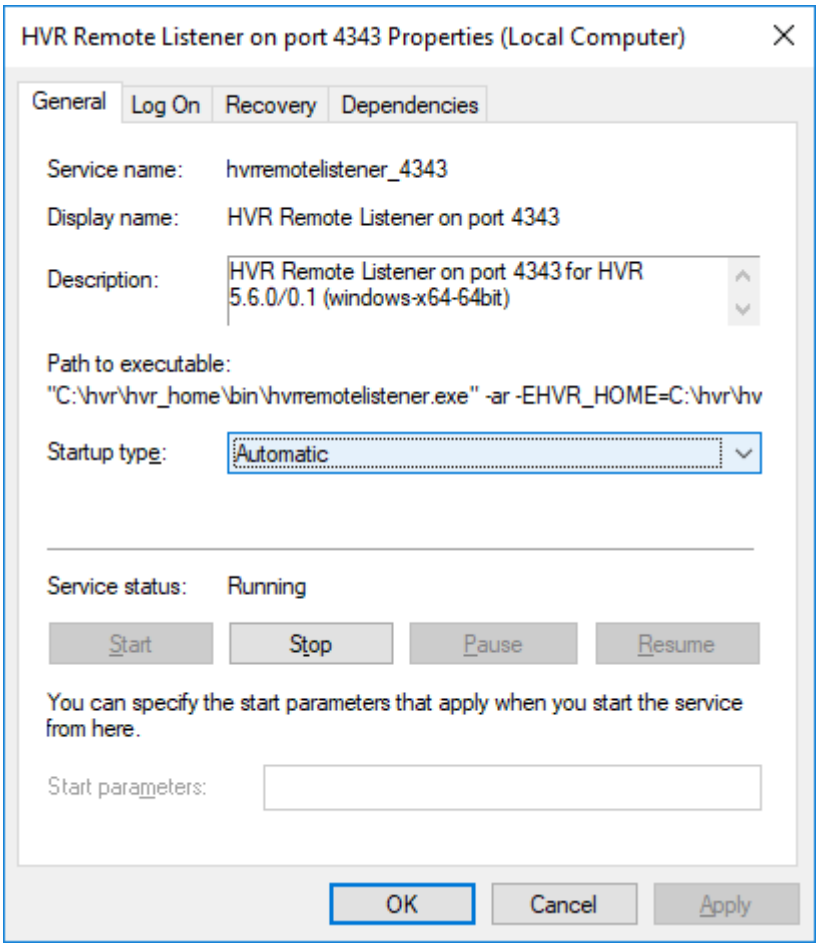
By default, the HVR processes start automatically after Windows boot. However, if the HVR processes do not start automatically after a Windows boot, the following procedure can be performed:

- Set the **HVR Remote Listener** service to start automatically on the server/machine, which has HVR Agent (for Capture/Integrate) installed.
- Set the **HVR Scheduler** service to start automatically on the server/machine, which has HVR Hub installed.
- In case you have both HVR Hub and HVR Agent (for Capture/Integrate) on same server/machine, set the **HVR Scheduler** and HVR Remote Listener services to start automatically.

Set HVR Remote Listener Service to Start Automatically

To set the **HVR Remote Listener** service to start automatically:

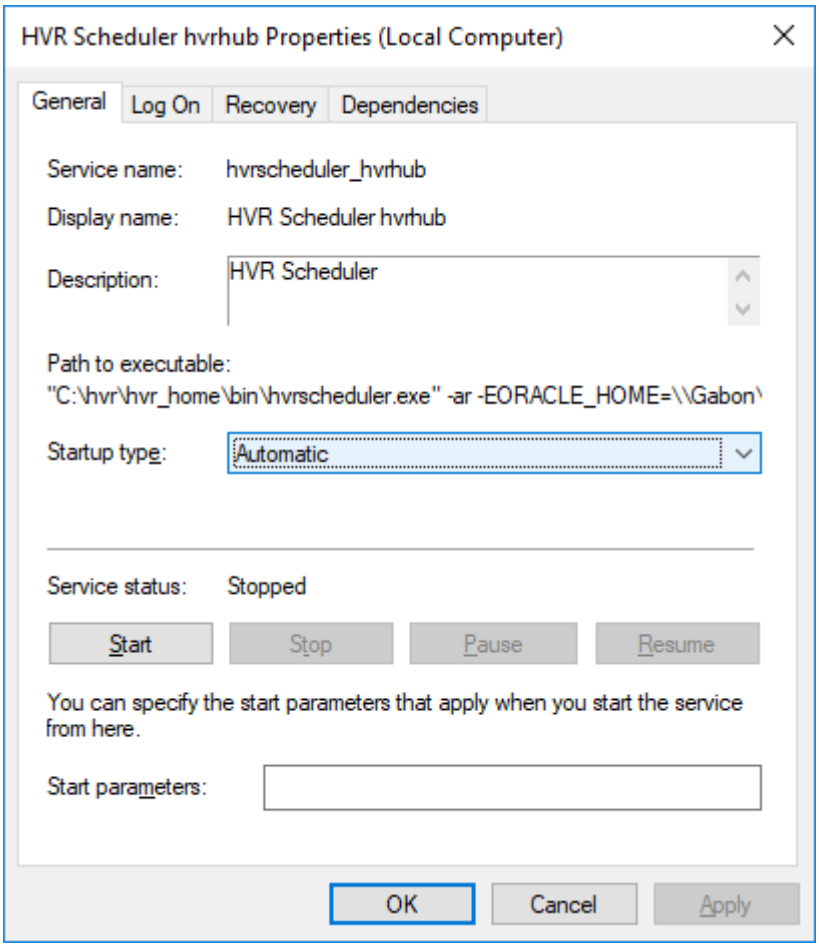
1. Access the Windows services and find the **HVR Remote Listener** service in the list of services.
2. Right-click the **HVR Remote Listener** service and select **Properties** from the context menu.
3. Under the **General** tab of the **HVR Remote Listener Properties** dialog, set the **Startup type** to **Automatic**.
4. Click **OK**.



Set HVR Scheduler Service to Start Automatically

To set the **HVR Scheduler** service to start automatically:

1. Access the Windows services and find the **HVR Scheduler** service in the list of services.
2. Right-click the **HVR Scheduler** service and select **Properties** from the context menu.
3. Under the **General** tab of the **HVR Scheduler Properties** dialog, set the **Startup type** to **Automatic**.
4. Click **OK**.

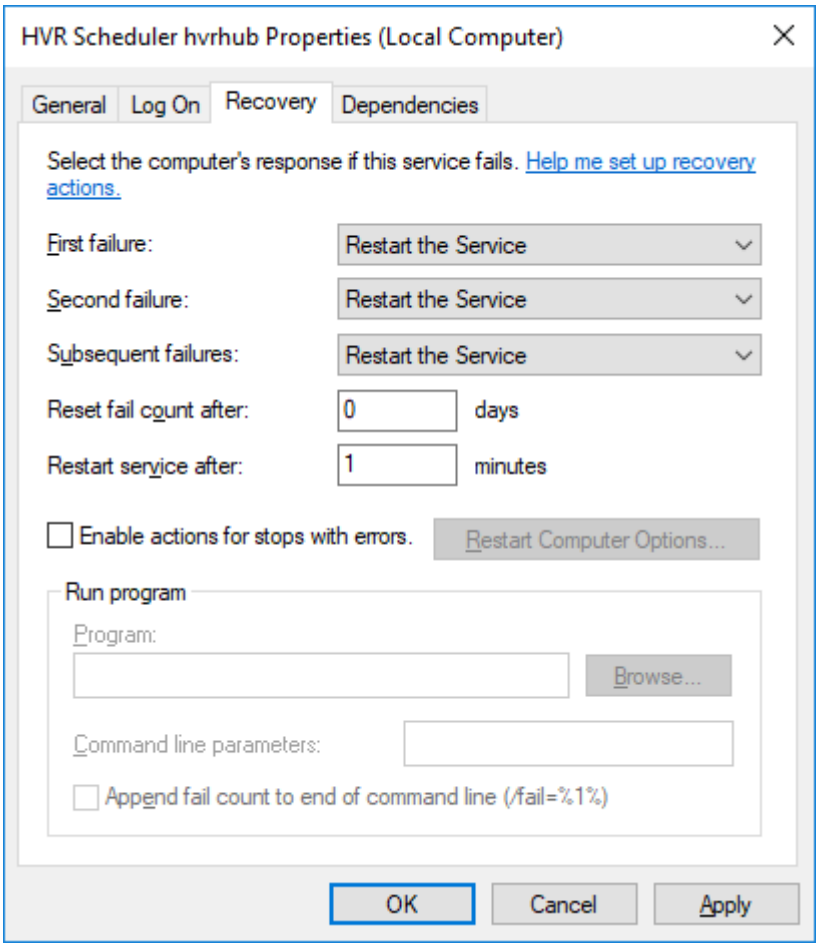


Set HVR Scheduler Service to Restart after Failing

In addition to auto-start on Windows, it is recommended to configure a recovery option in case there is a failure/recovery of the process.

To set the **HVR Scheduler** service to restart automatically after a failure:

1. Access the Windows services and find the **HVR Scheduler** service in the list of services.
2. Right-click the **HVR Scheduler** service and select **Properties** from the context menu.
3. Under the **Recovery** tab of the **HVR Scheduler Properties** dialog, set **Restart the Service** for the first and subsequent failures as shown below.
4. Click **OK**.



Set HVR Scheduler Service Dependent on DBMS Service

The **HVR Scheduler** service should be dependent on the DBMS service. Otherwise, after reboot the **HVR Scheduler** will fail immediately when it tries to connect to the hub database. A service dependency should be created between the **HVR Scheduler** and the DBMS (using the Windows command **sc config**).

For example, suppose the **HVR Scheduler** service is called **hvrscheduler_hvrhub** and the DBMS service is called **OracleServiceOHS**, use the following command:

```
sc config hvrscheduler_myhub depend=OracleServiceOHS
```

To list the Windows services (filtered by service name), use the following command:

```
sc queryex | find "hvr"
```

Configuring Remote Installation of HVR on Unix or Linux

Contents

This section describes the configuration required for a remote installation of HVR (also known as HVR remote agent) on Unix/Linux. These configuration steps are required either when:

- connecting from an HVR Hub to an HVR remote agent on the Source (capture) or Target (integrate) server,
- connecting from a PC (using **HVR GUI**) to an HVR Hub installed on a Unix/Linux server.

For connecting to a HVR remote agent installed on a Unix/Linux server, configure the system process (daemon) available on the remote Unix/Linux server. Also, an HVR listener port must be configured on the server running the HVR remote agent. Pick an arbitrary TCP/IP port number between **1024** and **65535** which is not already in use. We suggest using **4343** as the HVR listener port number and the following examples throughout this section will reference this port number.

An alternative method for connecting to a remote installation of HVR is by using the command **hvrremotelistener**. For more information, see section Using Hvrremotelistener below.

Using Unix/Linux Daemon

Depending on the daemon type available, one of the following configuration methods can be used to invoke HVR installed on a remote server:

- Configuring systemd
- Configuring xinetd
- Configuring inetd

If the above mentioned daemons are not available then use the alternate configuration method described in section Configuring Init below.

> The values used for **HVR_HOME**, **HVR_CONFIG** and **HVR_TMP** are for the current machine.

Configuring systemd

The following steps should be performed as user **root** to configure **systemd**:

1. Create the **systemd** unit files **hvr.socket** and **hvr@.service** in **/etc/systemd/system** directory.

    a. **hvr.socket** should contain the following:

```
[Unit]
Description=HVR service socket

[Socket]
ListenStream=4343
Accept=true
TriggerLimitIntervalSec=1s
TriggerLimitBurst=10000
MaxConnectionsPerSource=100
MaxConnections=500
KeepAlive=true

[Install]
WantedBy=sockets.target
```

- **TriggerLimitIntervalSec** is supported since systemd version 230.
- **TriggerLimitBurst** is supported since systemd version 230.
- **MaxConnectionsPerSource** is supported since systemd version 232.

b. **hvr@.service** should contain the following:

```
[Unit]
Description=HVR service

[Service]
Environment="HVR_HOME=/home/hvruser/hvr/hvr_home"
Environment="HVR_CONFIG=/home/hvruser/hvr/hvr_config"
Environment="HVR_TMP=/home/hvruser/hvr/hvr_tmp"
User=root
ExecStart=/home/hvruser/hvr/hvr_home/bin/hvr -r
StandardInput=socket
KillMode=process

[Install]
WantedBy=multi-user.target
```

Option **–r** tells **hvr** to run as a remote child process. For more options (like encryption, PAM) that can be supplied as the server program arguments (**ExecStart**), see command **Hvr**.

2. To enable and start the service, execute the following commands:

```
systemctl enable hvr.socket

systemctl start hvr.socket
```

3. To verify whether the service is active, execute the following command:

```
systemctl status hvr.socket
```

Sample output:

```
hvr.socket - HVR service socket
 Loaded: loaded (/etc/systemd/system/hvr.socket; enabled; vendor preset: enabled)
 Active: active (listening) since Mon 2019-07-08 17:54:44 CEST; 5s ago
 Listen: [::]:4343 (Stream)
 Accepted: 0; Connected: 0
```

Configuring xinetd

The following steps should be performed to configure **xinetd**:

1. Create **/etc/xinetd.d/hvr** file with the following content in it:

```
service hvr
{
 socket_type  = stream
 wait         = no
 user         = root
 server       = /home/hvruser/hvr/hvr_home/bin/hvr
 server_args  = -r
 env         += HVR_HOME=/home/hvruser/hvr/hvr_home
 env         += HVR_CONFIG=/home/hvruser/hvr/hvr_config
 env         += HVR_TMP=/home/hvruser/hvr/hvr_tmp
 disable      = no
 cps          = 10000 30
 per_source   = 100
 instances    = 500
}
```

> Option **–r** tells **hvr** to run as a remote child process. For more options (like encryption, PAM) that can be supplied as the server program arguments (**server_args**), see command **Hvr**.

2. The name of the **xinetd** service for HVR (created in the previous step) and the TCP/IP port number for HVR listener should be added to **/etc/services**:

```
hvr 4343/tcp    #Port for the remote installation of HVR
```

3. Reload or restart the **xinetd** service to apply the changes. For information about restarting the **xinetd** service, r efer to the operating system documentation.

Configuring inetd

The following steps should be performed to configure **inetd**:

1. For generic Unix, the following line should be added to **/etc/inetd.conf**:

```
hvr stream tcp nowait root /home/hvruser/hvr/hvr_home/bin/hvr hvr -r -EHVR_HOME=/home
/hvruser/hvr/hvr_home    -EHVR_CONFIG=/home/hvruser/hvr/hvr_config    -EHVR_TMP=/home
/hvruser/hvr/hvr_tmp
```

> Option **–r** tells **hvr** to run as a remote child process and **-E** defines the environment variables. For more options (like encryption, PAM) that can be supplied as the server program arguments, see command **Hvr**.

2. For Solaris version 10 and higher, the file **/etc/inetd.conf** must be imported into System Management Facility (SMF) using the command **inetconv**.
3. The name of the **inetd** service for HVR (created in the previous step) and the TCP/IP port number for HVR listener should be added to **/etc/services**:

```
hvr 4343/tcp    #Port for remote installation of HVR
```

4. Reload or restart the **inetd** service to apply the changes. For information about restarting the **inetd** service, refer to the operating system documentation.

Configuring Init

This method requires HVR's script file **hvr_boot** (available in **hvr_home/lib**) and a user-defined configuration file **hvrtab** .

> The script file **hvr_boot** allows you to start and stop HVR processes (both **HVR Scheduler** and **HVR Remote Listener**) defined in the configuration file **hvrtab**. The script file **hvr_boot** should only be executed by the **root** user.

The following steps should be performed as user **root** to configure **init**:

1. Create the configuration file **hvrtab** in the **/etc** directory. Each line of this configuration file should contain four or more parameters separated by a space in the following format:

   username port_or_hub hvr_home hvr_config [*options*]

   - *username*: Indicates the Unix/Linux username under which HVR runs.
   - *port_or_hub*: Indicates a TCP/IP port number (for **HVR Remote Listener**) or the username and password of the hub database (for **HVR Scheduler**). The DB connection string syntax for the hub database differs for each database. For DB connection string syntax, see Calling HVR on the Command Line.
   - *hvr_home*: Indicates the path for **$HVR_HOME**.
   - *hvr_config*: Indicates the path for **$HVR_CONFIG**.
   - [options]: Indicates an optional parameter to pass other options to the HVR process. For more information about the options, see **hvrremotelistener** and **hvrscheduler**.

   Sample configuration file:

```
# This hvrtab file starts a Remote Listener and two HVR schedulers (one for an
Oracle hub and one for an Ingres hub).

# The Oracle password is encrypted. For more information, see documentation for
command hvrcrypt.

root 4343 /home/hvruser/hvr/hvr_home /home/hvruser/hvr/hvr_config

mylinuxuser   orahubdb/!{DszmZY}!   /home/hvruser/hvr/hvr_home   /home/hvruser/hvr
/hvr_config   -EHVR_TMP=/home/hvruser/hvr/hvr_tmp   -EORACLE_HOME=/opt/oracle   -
EORACLE_SID=prod

mylinuxuser  inghubdb  /home/hvruser/hvr/hvr_home  /home/hvruser/hvr/hvr_config  -
EHVR_TMP=/home/hvruser/hvr/hvr_tmp -EII_SYSTEM=/opt/ingres -EHVR_PUBLIC_PORT=50001
```

Lines starting with a hash (#) are treated as comments.

2. Copy the script file **hvr_boot** (available in **hvr_home/lib**) to the **init.d** directory and create symlinks to the **rc.d** directory.

> For an Oracle RAC, the script file **hvr_boot** can be enrolled in the cluster with command **crs_profile**.

The following example uses start sequence **97** and stop sequence **03** (except HP–UX which uses **997** and **003** because it requires three digits).

- On AIX, to start and stop HVR for run level 2:

```
$ cp hvr_boot /etc/rc.d/init.d
$ ln -s /etc/rc.d/init.d/hvr_boot /etc/rc.d/rc2.d/S97hvr_boot
$ ln -s /etc/rc.d/init.d/hvr_boot /etc/rc.d/rc2.d/K03hvr_boot
```

- On HP–UX, to start and stop HVR for run level 3:

```
$ cp hvr_boot /sbin/init.d
$ ln -s /sbin/init.d/hvr_boot /sbin/rc3.d/S997hvr_boot
$ ln -s /sbin/init.d/hvr_boot /sbin/rc3.d/K003hvr_boot
```

- On Linux, to start HVR for run levels 3 and 5 and stop for all run levels:

```
$ cp hvr_boot /etc/init.d
$ ln -s /etc/init.d/hvr_boot /etc/rc.d/rc3.d/S97hvr_boot
$ ln -s /etc/init.d/hvr_boot /etc/rc.d/rc5.d/S97hvr_boot
$ ln -s /etc/init.d/hvr_boot /etc/rc.d/rc0.d/K03hvr_boot
$ ln -s /etc/init.d/hvr_boot /etc/rc.d/rc1.d/K03hvr_boot
$ ln -s /etc/init.d/hvr_boot /etc/rc.d/rc2.d/K03hvr_boot
$ ln -s /etc/init.d/hvr_boot /etc/rc.d/rc3.d/K03hvr_boot
$ ln -s /etc/init.d/hvr_boot /etc/rc.d/rc4.d/K03hvr_boot
$ ln -s /etc/init.d/hvr_boot /etc/rc.d/rc5.d/K03hvr_boot
$ ln -s /etc/init.d/hvr_boot /etc/rc.d/rc6.d/K03hvr_boot
```

- On Solaris 8 or 9, to start and stop HVR for run level 2 (which implies level 3):

```
$ cp hvr_boot /etc/init.d
$ ln -s /etc/init.d/hvr_boot /etc/rc2.d/S97hvr_boot
$ ln -s /etc/init.d/hvr_boot /etc/rc2.d/K03hvr_boot
```

- On Solaris 10 and higher;

> For newer Solaris versions, the script file **hvr_boot** must be registered in the Solaris's System Management Facility (SMF). Also, the file **hvr_boot.xml** (available in **hvr_home/lib**) should be copied to the **init.d** directory.

```
$ cp /opt/hvr/hvr_home/lib/hvr_boot /lib/svc/method
$ cp /opt/hvr/hvr_home/lib/hvr_boot.xml /var/svc/manifest/application

$ svccfg
svc> import /var/svc/manifest/application/hvr_boot.xml
svc> quit

$ svcadm enable svc:/application/hvr_boot
$ svcs -a|grep hvr     # To check if the service is running
16:00:29 svc:/application/hvr_boot:default
```

Security Notes

- On systems where restricted security is configured, it may be necessary to add the following line to file**/etc/hosts.allow** :

```
hvr: ALL
```

- It may be necessary to disable Security Enhanced Linux (SELinux). To disable SELinux, the following line should be available/added into the file **/etc/selinux.conf** and then reboot the server.

```
SELINUX=disable
```

> To see the status of SELinux, use the Linux command **sestatus**.

Using Hvremotelistener

In this method, for connecting to a remote installation of HVR, the command **hvrremotelistener** should be executed on the remote server. This method is preferred when:

- the Linux **xinetd** package is not installed (this is the case for RHEL5) or
- the **root** privilege is unavailable or
- the password authentication cannot be configured

The following command uses options **–d** (run as daemon) and **-N** (skip password authentication) to listen on port **4343**.

```
hvrremotelistener -d -N 4343
```

The following command uses option **–i** (interactive) and **-N** (skip password authentication) to listen on port **4343**. Note that, in this method exiting the shell will terminate the remote listener.

```
hvrremotelistener -i -N 4343
```

> Disabling password authentication (option **-N**) is a security hole, but may be useful as a temporary or troubleshooting measure.

Testing Connection to a Remote Installation of HVR

To test the connection, execute the command **hvrtestlistener** on the server from which you are connecting to a remote installation of HVR:

```
hvrtestlistener node port
```

Example:

```
hvrtestlistener myintegratenode 4343
```

If authorization (username and password) is required to connect to the remote server then use the command **hvrtestlistener** with option **-L**. For example,

```
hvrtestlistener -L myusername/mypassword myintegratenode 4343
```

Sample Output:

```
hvrtestlistener: HVR 5.6.0/0 (windows-x64-64bit)
hvrtestlistener: Connection with authorization to myintegradenode:4343 successful.
hvrtestlistener: Finished. (elapsed=0.22s)
```

See Also

- Auto-Starting HVR Scheduler after Unix or Linux Boot

Configuring Remote Installation of HVR on Windows

Contents

This section describes the configuration required for a remote installation of HVR (also known as HVR remote agent) on Windows. These configuration steps are required either when:

- connecting from an HVR Hub to an HVR remote agent on the Source (capture) or Target (integrate) server,
- connecting from a PC (using **HVR GUI**) to an HVR Hub installed on a Windows server.

For connecting to a HVR remote agent installed on a Windows server, create an **HVR Remote Listener** service on the remote Windows server. Also, an HVR listener port must be configured on the server running the HVR remote agent. Pick an arbitrary TCP/IP port number between **1024** and **65535** which is not already in use. We suggest using **4343** as the HVR listener port number and the following examples throughout this section will reference this port number.

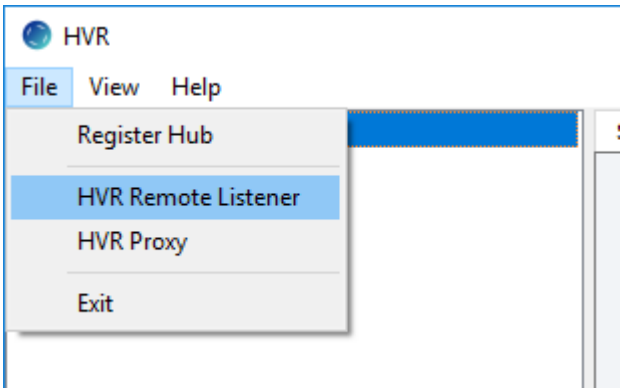
Creating and Starting HVR Remote Listener Service

In Windows, the HVR Remote Listener service can be created and started (or stopped) from HVR's GUI or Command Line Interface (CLI):

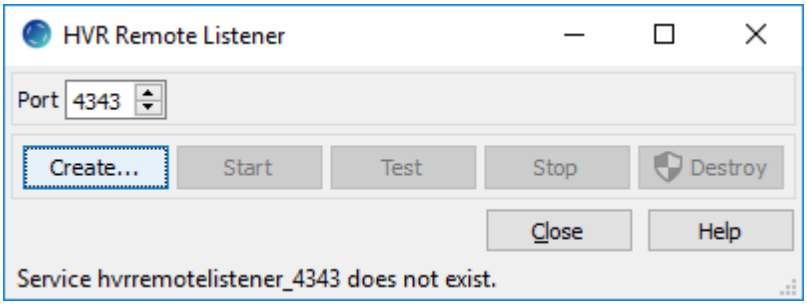
HVR GUI

To create and start the **HVR Remote Listener** service, the following steps should be performed on the remote Windows server where HVR is installed:

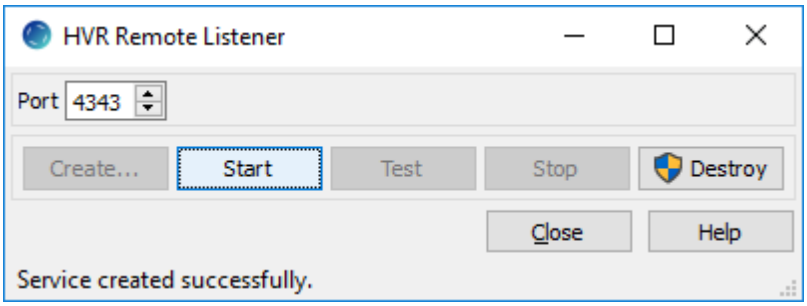
1. Launch **HVR GUI**.
2. Click **File HVR Remote Listener**.



3. Click **Create...** to create the **HVR Remote Listener** service.



4. Click **Start** to start the **HVR Remote Listener** service.



In CLI

To create and start the **HVR Remote Listener** service, execute the following command on the remote Windows server where HVR is installed:

```
hvrremotelistener -acs 4343
```

Testing Connection to a Remote Installation of HVR

To test the connection, execute the command **hvrtestlistener** on the server from which you are connecting to a remote installation of HVR:

```
hvrtestlistener node port
```

Example:

```
hvrtestlistener myintegratenode 4343
```

If authorization (username and password) is required to connect to the remote server then use the command **hvrtestlistener** with option **-L**. For example,

```
hvrtestlistener -L myusername/mypassword myintegratenode 4343
```

Sample Output:

```
hvrtestlistener: HVR 5.6.0/0 (windows-x64-64bit)
hvrtestlistener: Connection with authorization to myintegradenode:4343 successful.
hvrtestlistener: Finished. (elapsed=0.22s)
```

See Also

- Auto-Starting HVR after Windows Boot

Authentication and Access Control

Contents

Authentication

HVR supports the following modes to check the username and password when authenticating a connection to a remote HVR machine:

1. **Operating System (default)**
   HVR asks the operating system to verify the username and password. This is the default authentication system of HVR when the others (below) are not configured.

2. **No Password Authentication**
   Authentication can also be disabled when connecting to a remote HVR hub/location. If option **-N** is supplied to HVR (see options of **Hvr** and **Hvrremotelistener**) then all valid operating system usernames with any password is accepted. This mode can be used for testing. It can also be configured with an **access_conf.xml** file to authenticate the identity of incoming connections using SSL. For more information, see option **-a** of **Hvr**.

3. **Pluggable Authentication Module (PAM)**
   PAM is a service provided by Linux & Unix as an alternative to regular username/password authentication, e.g. checking the **/etc/passwd** file. For more information see option **–p** *pamsrv* in **Hvr** and **Hvrremotelistener**.

4. **LDAP Authentication**
   HVR authenticates incoming username/password by invoking its **Hvrvalidpwldap** plugin.

5. **Private Password File Authentication**
   HVR authenticates incoming username/password by invoking its **Hvrvalidpwfile** plugin.

6. **Custom hvrvalidpw Authentication Plugin**
   HVR also allows you to supply your own authentication plugin. The custom plugin file should be named as **Hvrvalidpw** and saved in **HVR_HOME/lib** directory. It should also obey (simple) call conventions.

> For some of the above authentication modes (e.g. PAM or LDAP), HVR should only use the username /password for authentication, but should not change from the current operating system user to that login. This is configured using option **-A** in **Hvr** and **Hvrremotelistener**. In this case the daemon should be configured to start the HVR child process as the correct operating system user (instead of **root**).

Access Control

Since v5.3.1/22

HVR allows different access levels for the authenticated users based on their username, their LDAP groups (if configured), and the hub database name.

The following access levels are supported by HVR:

- **ReadOnly** : User can view catalog information and job status.
- **ReadExec** : User can view catalog information, job status and also execute **HVR Compare**, **HVR Scheduler**.
- **ReadWrite** : User can view catalog information, job status and modify catalogs. They also can execute **HVR Compare**, **HVR Initialize**, **HVR Refresh**, **HVR Scheduler**, etc.

- **Refresh** : User can execute **HVR Refresh**.

To enable access control, the file **access_rights.conf** must be created and saved in **HVR_HOME/lib** directory. The file **access_rights.conf** can be used together with LDAP or Private Password File Authentication to limit permissions of HVR GUIs connecting remotely to HVR hub. An example configuration file **access_rights.conf_example** is available in **HVR_HOME/lib** directory.

Following is the syntax for defining the parameters in configuration file:

user **:** *hub* **:** *access_level*

or

@group **:** *hub* **:** *access_level*

- *user* can be a specific username or to match any user.
- *groups* can be used only if **Hvrvalidpwldap** is configured to fetch the LDAP user groups.
- *hub* can be a hub database name, or it can be to match any hub database.
- *access_level* can be **ReadOnly** or **ReadExec** or **ReadWrite**

> If a *user* is not assigned any access level then the connection is rejected even if the password is correct.

Example:

```
User1:Hub1:ReadExec                    # User1 has ReadExec access to Hub1.

User2:Hub1:ReadExec,Refresh               # User2 has ReadExec access to Hub1 and also
execute HVR Refresh.

User2:Hub2:ReadWrite                   # User2 has ReadWrite access to Hub2.

*:Hub2:ReadWrite                       # All users have ReadWrite access to Hub2.

@General-Users:*:ReadOnly               # All members of General-Users group has ReadOnly
access to all hubs.

@Privilege-Users:*:ReadOnly,Refresh    # All members of Privilege-Users group has ReadOnly
 access to all hubs and also execute HVR Refresh.

@Admin-Users:*:ReadWrite                # All members of Admin-Users group has ReadWrite
access to all hubs.
```

Encrypted Network Connection

An HVR connection to a remote HVR location can be configured so that communication over the network is encrypted. When encryption is activated for an HVR location, every byte sent over the network (in either direction) is encrypted with the TLS protocol.

⚠ Since HVR 5.3.1/21, HVR uses TLS version 1.3 and prior to HVR 5.3.1/21, HVR used TLS version 1.0.

For network encryption, HVR uses OpenSSL, which was developed by the OpenSSL Project.

If necessary, the HVR hub and each remote location in an HVR channel can be given their own private key/public certificate pairs, so that both the hub and the locations can verify each other's identity.

To allow the hub to verify the identity of each remote location:

1. Supply the location's public certificate and private key to the HVR child process on a remote machine.
2. On the hub, use parameter **LocationProperties /SslRemoteCertificate** to point to a copy of the location's public certificate.

To allow the remote location to verify the identity of the hub:

1. On the hub, supply the hub's public certificate and private key using parameter **LocationProperties /SslLocalCertificateKeyPair**.
2. On the remote location, point to a copy of the hub's public certificate in the HVR's access file access_conf.xml. For more information, refer to section Hvrproxy.

The RSA public certificate/private key pair is used to authenticate and start a session in HVR. The public certificate is embedded in an X509 certificate, and the private key is encrypted using an internal password with an AES256 algorithm. By default, the keys used for this asymmetric negotiation are 2048 bits long, although longer key lengths can be specified when generating a public certificate and private key pair. For more information, see command **hvrsslgen**.

The HVR hub guards against third parties impersonating the remote HVR location (e.g. by spoofing) by comparing the SHA256 checksums of the certificate used to create the secure connection and its own copy of the certificate.

Public certificates are self-signed. HVR checks that the hub on the remote machines' copies of this certificate are identical, so signing by a root authority is not required.

For the steps to set up an encrypted remote connection, refer to section Configuring Encrypted Network Connection.

Configuring Encrypted Network Connection

Contents

This section describes the steps required to set up a secure HVR network connection between an HVR Hub and a remote location using:

- the default private key and public certificate files which are delivered with HVR in **$HVR_HOME/lib/cert** (Standard Configuration Steps)
- a newly generated private key and public certificate (Advanced Configuration Steps)

> Only the public certificate should be stored on a hub machine, whereas on a remote location, both the public certificate and the private key should be present.

Standard Configuration Steps

The steps in this section are to set up a secure HVR connection using the default private key and public certificate provided by HVR. The example here assumes that a channel and remote locations are already created and configured.

> In case the connection to a remote HVR location is not configured, see the appropriate section Configuring Remote Installation of HVR on Unix or Linux or Configuring Remote Installation of HVR on Windows.

1. Set up HVR on the remote HVR location to expect an encrypted connection.

   Steps for Unix/Linux

   a. Depending on the daemon type available, one of the following configuration methods can be used:

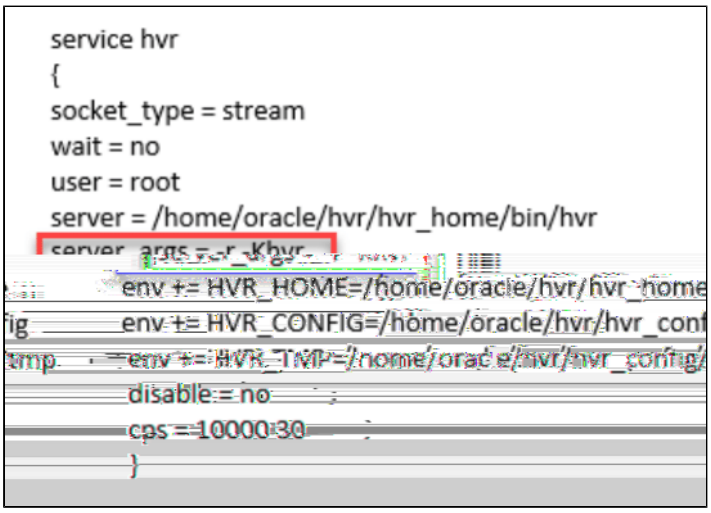
   > HVR on the remote machine consists of an HVR executable file with option **–r** which is configured to listen on a specific port number. Option **–K***pair* should be added to specify the public certificate and private key respectively. The default *basename* for the public certificate and private key pair is **hvr**. For more information about options **-r** and **-K**, refer to the respective sections of the **Hvr** command page.

   - For **systemd**, add option **-Khvr** to the following line into file in **/etc/systemd/system/hvr@.service**:

     ```
     ExecStart=/home/hvruser/hvr/hvr_home/bin/hvr -r -Khvr
     ```

   - For **xinetd**, add option **-Khvr** to the following line in file **/etc/xinetd.d/hvr**:

     ```
     server_args = -r -Khvr
     ```



- For **inetd**, add option **-Khvr** to the following line in file **/etc/inetd.conf**:

```
hvr stream tcp nowait root /usr/hvr/hvr_home/bin/hvr hvr -r -Khvr -
EHVR_HOME=/usr/hvr/hvr_home -EHVR_CONFIG=/usr/hvr/hvr_config -EHVR_TMP=
/tmp
```

b. Start **HVR Remote Listener** using the following command:

```
hvrremotelistener -N -d -Khvr 4343
```

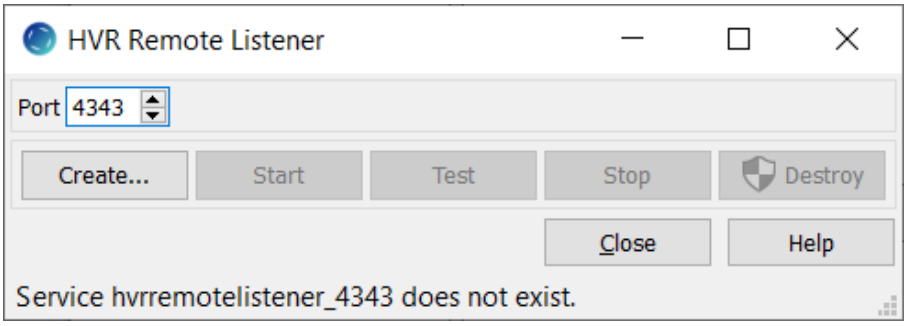
Steps for Windows

a. Create and run **HVR Remote Listener** service that requires SSL encryption for incoming connections.

i. In the HVR GUI on the remote machine, click **File** in the main menu and select **HVR Remote Listener**.

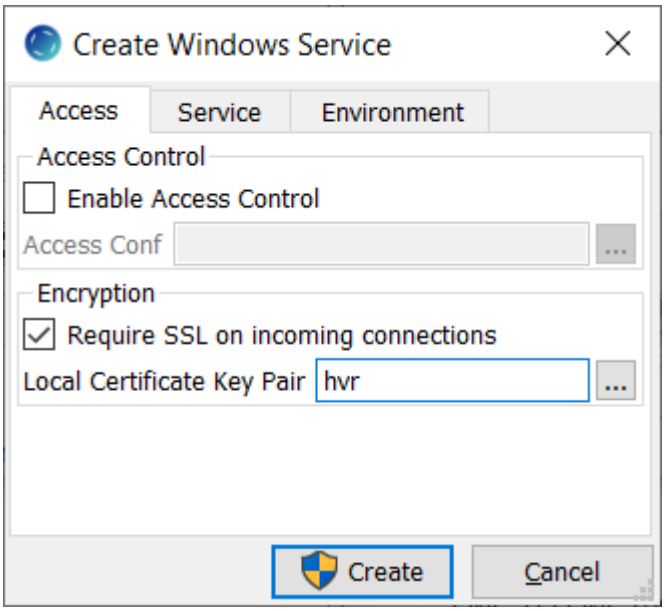
> If the **HVR Remote Listener** is already running, click **Stop** and then **Destroy** so that a new one with SSL encryption can be created.

ii. In the **HVR Remote Listener** dialog, click **Create**.

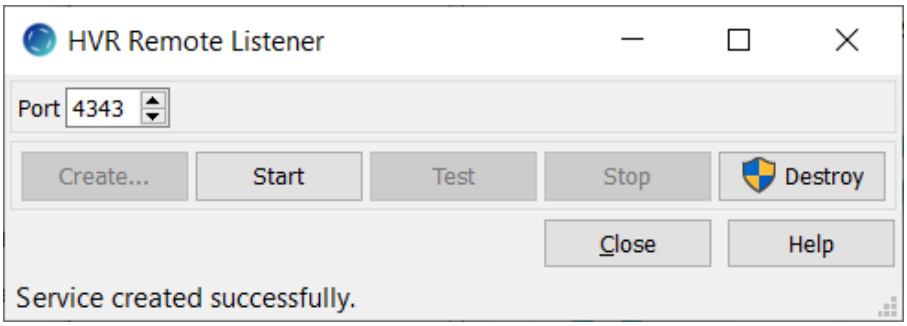


iii. In the **Create Windows Service** dialog, select **Require SSL on incoming connections** and specify the *basename* of the **Local Certificate Key Pair**. The default *basename* for the public certificate and private key pair is **hvr**.

iv. Click **Create**.



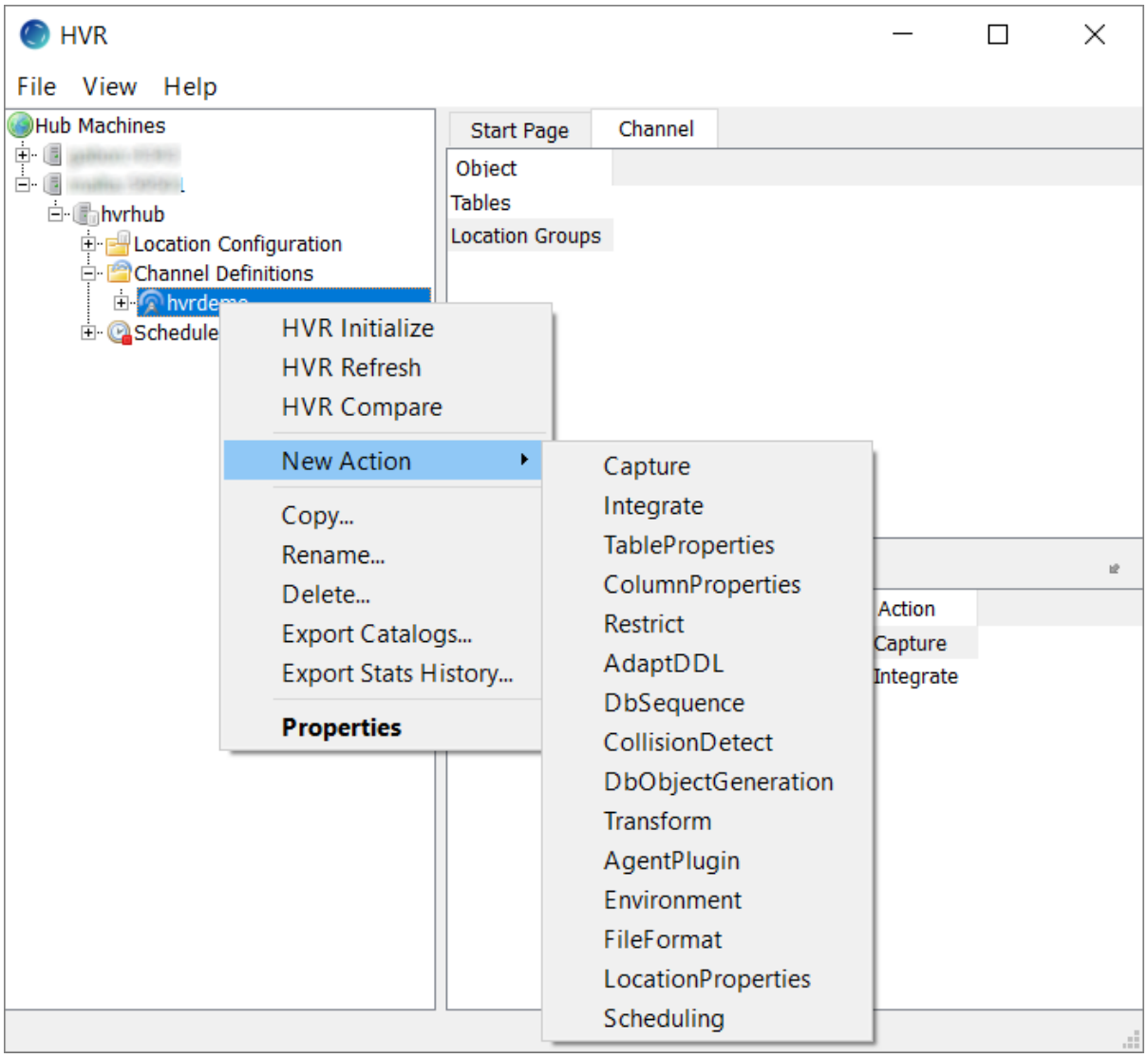
v. Click **Start** to start the service.



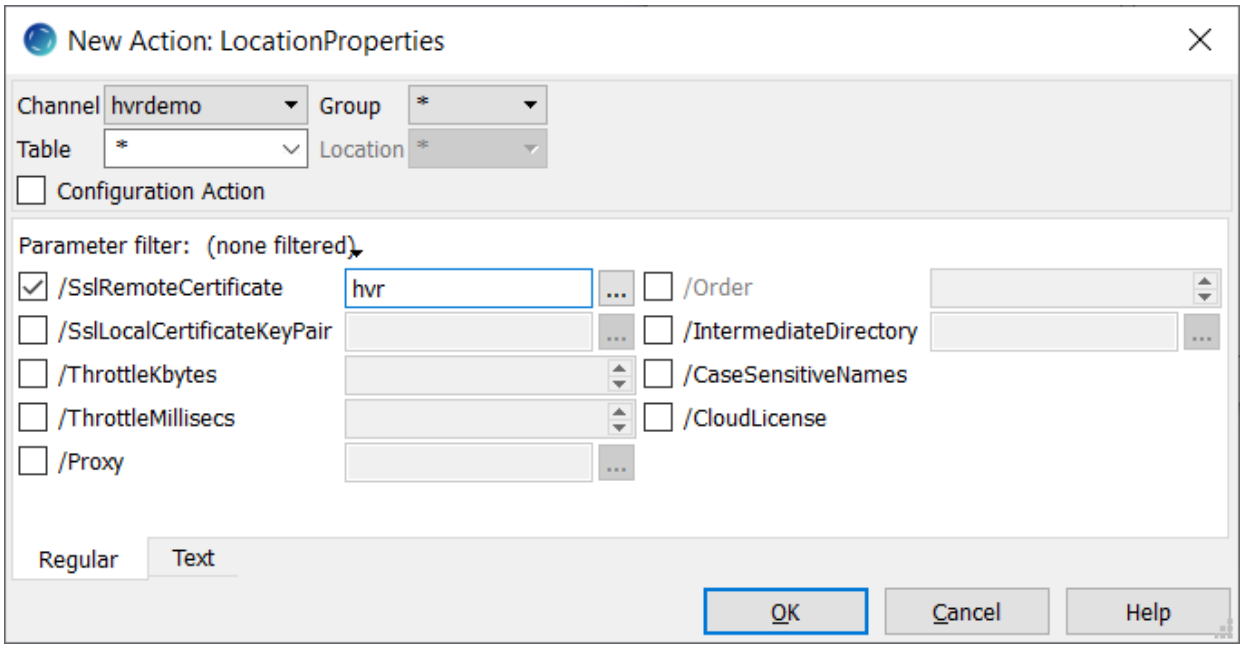
> In the Windows Services, you will see a service called HVR Remote Listener on port 4343 created and running.

2. Define action **LocationProperties** **/SslRemoteCertificate** to register the public certificate file in the channel configuration.

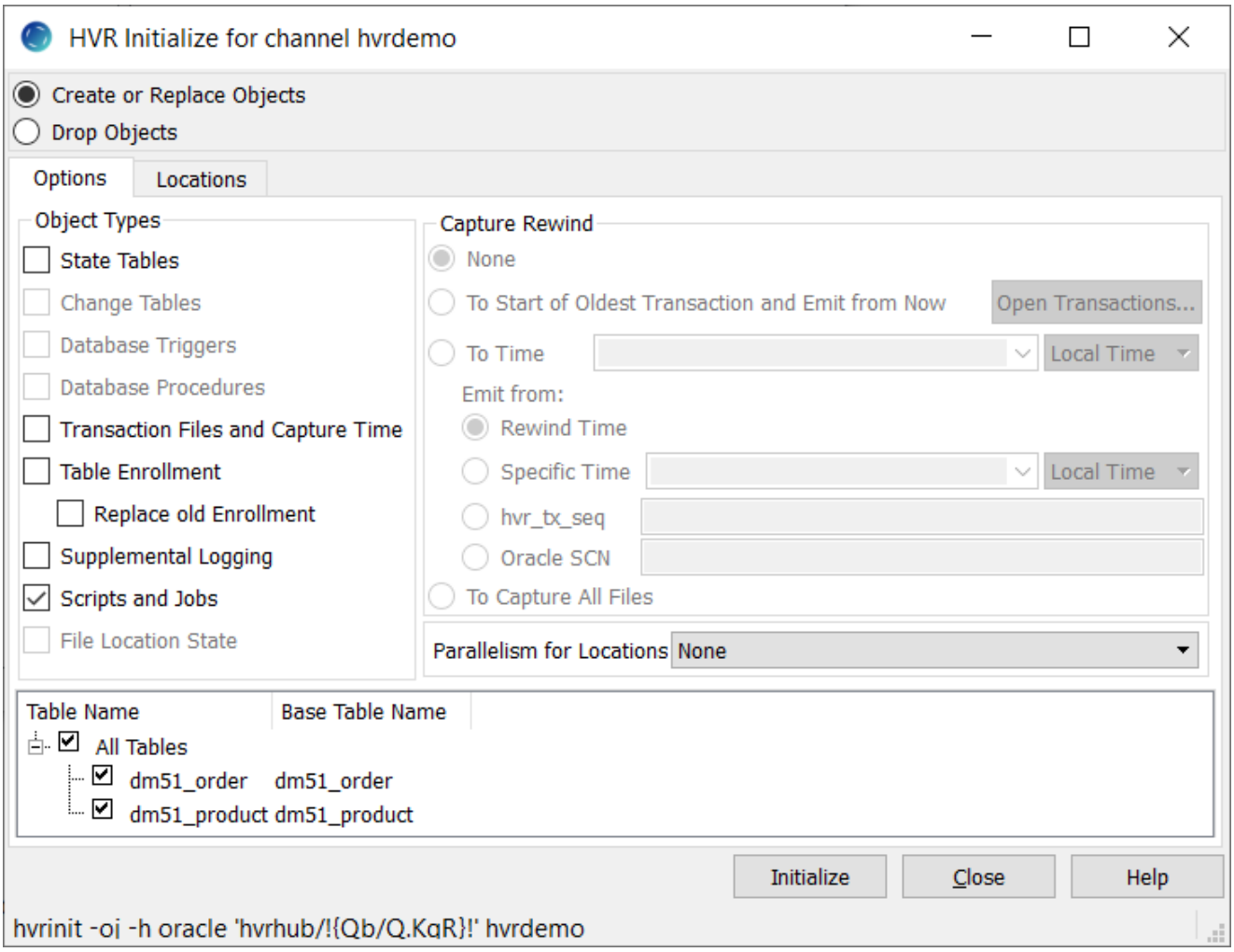
   a. Right-click the channel, navigate to **New Action** and click **LocationProperties**.



b. In the **New Action: LocationProperties** dialog, select **SSLRemoteCertificate** and specify the *basename* of the **Local Certificate Key Pair**. The default *basename* for the public certificate and private key pair is **hvr**.



3. Execute **HVR Initialize** for this action to take effect. Right-click the channel and select **HVR Initialize**. Ensure the **Scripts and Jobs** option is selected and click **Initialize**.



This (**hvrinit**) can also be executed on the command line as follows:

```
hvrinit -oj hubdb hvrdemo
```

Advanced Configuration Steps

The steps in this section are to set up a secure HVR connection using a newly generated private key and public certificate.

There are two methods how the encrypted connection can be configured in an HVR channel:

- Use different certificate key pair on source and target machines. Example: MyCertificate1 and MyCertificate2. For this method, copy the public certificate files on source (MyCertificate1) and target (MyCertificate2) to the hub machine. Define two separate actions **LocationProperty /SslRemoteCertificate** for source and target groups in the channel.
- Use same certificate key pair on source and target machines. Example: MyCertificate. For this method, the certificate key pair should be copied to the other remote location also. For example, if the certificate key pair was generated on the source machine, then it should be copied to the target machine also and the public certificate should be copied to the hub machine. Define one action **LocationProperty /SslRemoteCertificate** in the channel.

The example here assumes that a channel and remote source location are already created and configured.

In case the connection to a remote HVR location is not configured, see the appropriate section Configuring Remote Installation of HVR on Unix or Linux or Configuring Remote Installation of HVR on Windows.

1. On a remote location, generate a new private key and public certificate pair using command **Hvrsslgen**. For more information on the arguments and various options available for this command, refer to section **Hvrsslgen**.
2. Copy the public certificate file into the **$HVR_HOME/lib/cert** directory of the hub machine.
3. Set up HVR on the remote HVR location to expect an encrypted connection (using the newly generated public certificate and private key pair):

Steps for Unix/Linux

a. Depending on the daemon type available, one of the following configuration methods can be used:

HVR on the remote machine consists of an HVR executable file with option **–r** which is configured to listen on a specific port number. Option **–K***pair* should be added to specify the public certificate and private key respectively. In this example, the *basename* for the public certificate and private key pair is **MyCertificate**. For more information about options **-r** and **-K**, refer to the respective sections of the **Hvr** command page.

- For **systemd**, add option **-K** to the following line into file in **/etc/systemd/system/hvr@.service**:

```
ExecStart=/home/hvruser/hvr/hvr_home/bin/hvr  -r  StandardInput=socket
KillMode=process -KMyCertificate
```

- For **xinetd**, add option **-K** to the following line in file **/etc/xinetd.d/hvr**:

```
server_args = -r -KMyCertificate
```

- For **inetd**, edit file **/etc/inetd.conf**:

```
hvr stream tcp nowait root /usr/hvr/hvr_home/bin/hvr hvr -r -EHVR_HOME=
/usr/hvr/hvr_home  -EHVR_CONFIG=/usr/hvr/hvr_config  -EHVR_TMP=/tmp  -
KMyCertificate
```

b. Start **HVR Remote Listener** using the following command:

```
hvrremotelistener -N -d -KMyCertificate 4343
```

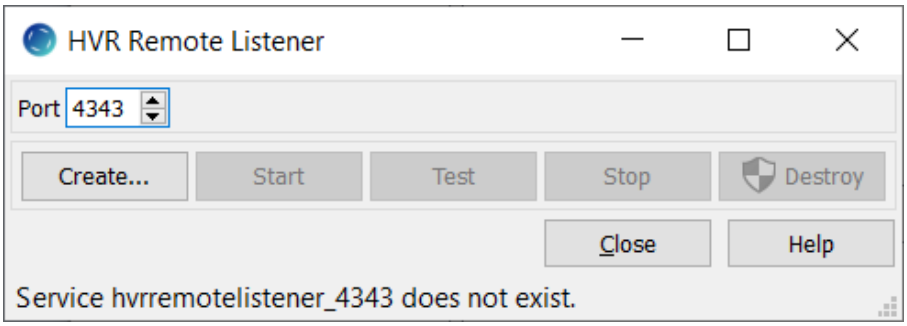
Steps for Windows

a. Create and run **HVR Remote Listener** service that requires SSL encryption for incoming connections.

i. In the HVR GUI on the remote machine, click **File** in the main menu and select **HVR Remote Listener**.

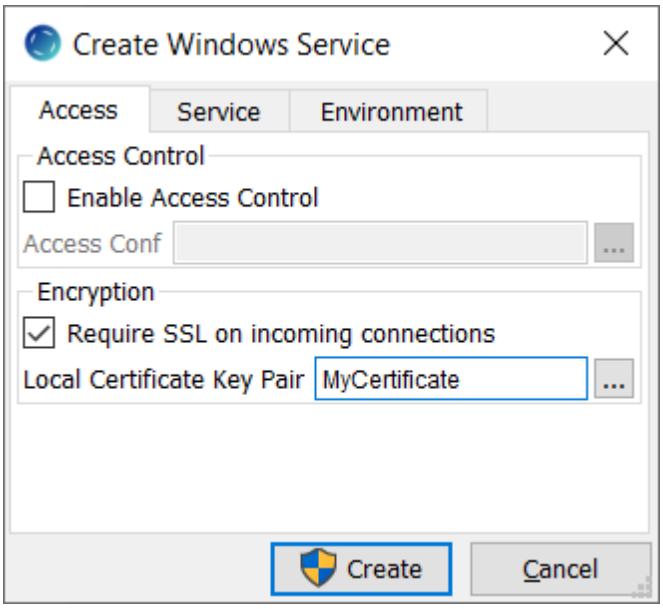
> If the **HVR Remote Listener** is already running, click **Stop** and then **Destroy** so that a new one with SSL encryption can be created.

ii. In the **HVR Remote Listener** dialog, click **Create**.

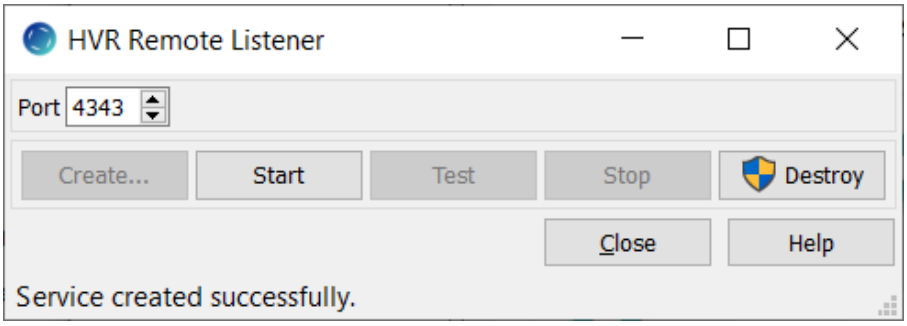


iii. In the **Create Windows Service** dialog, select **Require SSL on incoming connections** and specify the name for the **Local Certificate Key Pair**, e.g. 'MyCertificate'.

iv. Click **Create**.



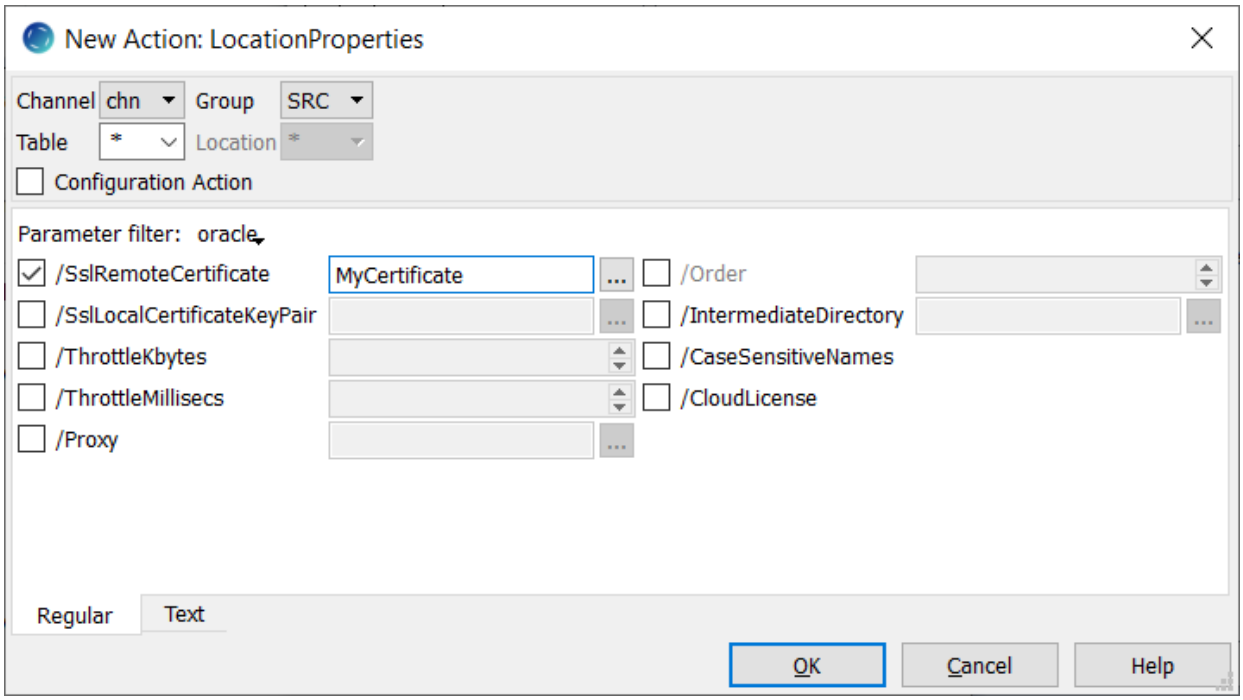
v. Click **Start** to start the service.



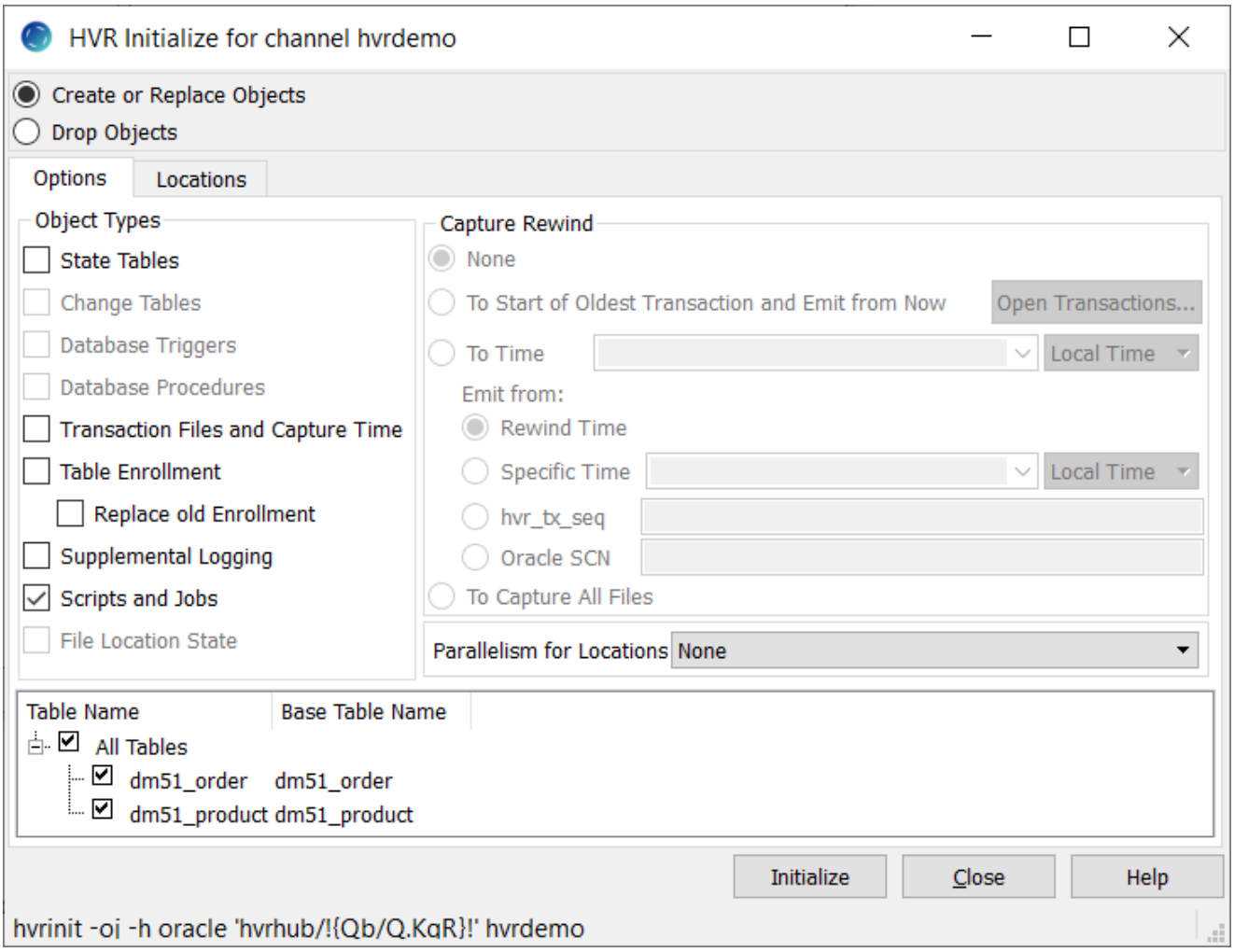
> In the Windows Services, you will see a service called HVR Remote Listener on port 4343 created and running.

4. Define action **LocationProperties /SslRemoteCertificate** to register the public certificate file in the channel configuration.

    a. Right-click the location group, navigate to **New Action** and click **LocationProperties**.
    b. In the **New Action: LocationProperties** dialog, select **SSLRemoteCertificate**.
    c. Browse and select the public certificate file (**MyCertificate.pub_cert**).



5. Execute **HVR Initialize** for this action to take effect. Right-click the channel and select **HVR Initialize**. Ensure the **Scripts and Jobs** option is selected and click **Initialize**.



This (**hvrinit**) can also be executed on the command line as follows:

```
hvrinit -oj hubdb mychn
```

Hub Wallet and Encryption

Since v5.6.5/5

Contents

The hub wallet is an advanced method for secure password encryption and storage. When you enable the hub wallet, all user passwords in HVR are encrypted using a modern AES256 encryption scheme and then stored in the hub database. The hub encryption key used for encryption is stored in the hub wallet. The hub wallet can be a software wallet file which is encrypted by a wallet password, or a KMS account (AWS). For more information, see Hub Wallet Types.

> Hub wallet is not a replacement for Encrypted Network Connection. Hub wallet and Network connection encryption should be used together for best security.

Benefits of using the hub wallet:

- Without the hub encryption key, anyone who has access to the hub database cannot decrypt the passwords stored in the hub database.
- HVR sends these encrypted passwords over the network to other HVR processes. If the connection is intercepted and the messages are accessed, it will not be possible to decrypt the sensitive information (like password) without the hub encryption key.

If the hub wallet is not configured and enabled, a less secure method is implemented. All user passwords in HVR are obfuscated (using a password obfuscation method) and stored in the hub database. If either of the following happens, unauthorized access to the hub database or the network connection is intercepted and the messages are read, the obfuscated passwords may be obtained and de-obfuscated.

For the steps to set up the hub wallet, see Configuring and Managing Hub Wallet.

> HVR commands such as **HVR GUI**, **HVR Remote Listener**, **HVR Scheduler**, or a job (such as capture, integrate running under scheduler) are separate HVR processes that take part in encryption.

Hub Wallet Configuration File

The hub wallet configuration file (**HVR_CONFIG/files/hvrwallet-***hubname***/hvrwallet-***tstamp***.conf**) is required for managing (creating, enabling, disabling, deleting) the hub wallet. It contains the configuration properties for the hub wallet like encryption type, wallet type, encryption key history, and information related to the KMS. The properties that can be saved/configured in the hub wallet configuration file differ based on hub wallet type. For more information about the properties that can be configured in the hub wallet configuration file, see section **Properties** in **hvrwalletconfig**. The hub wallet configuration file is a plain text file which must only be updated using the command **hvrwalletconfig**. If this

file is updated using a normal text editor, it could corrupt the wallet configuration file or lead to improper functioning of hub wallet. The timestamp in the hub wallet configuration file name is time when the hub wallet configuration is created. Each time the hub wallet configuration is updated, the hub wallet configuration file name gets a new timestamp.

> Prior to HVR 5.7.0/0, the hub wallet configuration file name and path was **HVR_CONFIG/files/hvrwallet-*hubname*-*timestamp*.conf**

How It Works

Whenever an HVR process running on the hub (called as hub process in this article) needs the hub encryption key (to encrypt/decrypt data), it looks into the hub wallet configuration file to determine the hub wallet type and other required information related to the hub wallet. The hub process then opens the hub wallet by supplying the wallet password and fetches the decrypted hub encryption key into this hub process's memory. A decrypted hub encryption key is never stored on the disk.

> Since the wallet password is stored in the process memory, whenever the **HVR Scheduler** is restarted, the wallet password is lost from its memory.

Methods to Supply Wallet Password

The wallet password must always be supplied for opening the hub wallet. Following are the different methods for supplying the wallet password:

- **Manual** (**default**): In this method, the user needs to supply the wallet password manually. The hub process waits for the wallet password until the user supplies it manually. The wallet password may be supplied:
  - In the HVR GUI, the hub process asks for the wallet password interactively. This is the GUI for the command **hvrwalletopen**.
  - Through the CLI, the hub process waits indefinitely for the user to supply the wallet password. The user needs to supply the wallet password using the command **hvrwalletopen**.
- **Auto-Open Password**: In this method, intervention from the user is not required; instead the wallet password is supplied by automatically fetching it from the wallet configuration file. The wallet password is stored obfuscated in the hub wallet configuration file. This method is good for a situation where user intervention is not desirable after a system restart, however, it is less secure to have the wallet password stored in the wallet configuration file.
  To enable or disable this method, see section Auto-Open Password in Configuring and Managing Hub Wallet.

  > If this method is used together with the software wallet, the backup of the files involved (wallet configuration file and software wallet file) should be taken at the same time. It is recommended to save /store the backups in separate locations to prevent the security threat involved in case one backup is compromised.

- **Auto-Open Plugin**: In this method, intervention from the user is not required; instead the wallet password is supplied by automatically executing a user defined plugin/script. The hub process can execute a user defined plugin/script to obtain the wallet password. Here, the wallet password may be stored obfuscated in the hub wallet configuration file or else it can be stored on a separate machine in the network. If the wallet password is stored on a separate machine, the user defined plugin/script can be used to fetch it from that machine.
  An example plugin:

```
#!/bin/sh
echo mywalletpassword | $HVR_HOME/bin/hvrwalletopen
```

To enable or disable this method, see section Auto-Open Plugin in Configuring and Managing Hub Wallet.

Classification of Data

Data/information in HVR is logically classified so that it can be used and protected more efficiently. Classification of data is based upon the sensitivity of the data and the impact it can have on the user/business should that data be accessed without authorization or be misused.

All data in HVR is classified into one of the following sensitivity levels, or categories:

- **Secret**: Unauthorized access/misuse of data in this category could cause a severe level of risk to the user /business. This category typically includes the passwords and private keys used for accessing/connecting to a database or technology.
- **Confidential**: Unauthorized access/misuse of data in this category could result in a moderate level of risk to the user/business. This category includes the user data like values in user-table that are part of replication, key-values exposed in error messages, and the files such as transaction (TX) files, diff files, intermediate files (direct file compare and online compare), **.coererr** file.
- **Official**: Unauthorized access/misuse of data in this category could result in a lesser risk (compared to Confidential) to the user/business. This includes the column names, timestamps and change metadata.
- **Public**: Data that is available in the public. This category includes the HVR product documentation.

Hub wallet supports encryption of **Secret** and **Confidential** data only. Also see **hvrwalletconfig** property **Encryption**.

Hub Wallet Types

HVR supports the following types of hub wallets:

1. **Software Wallet**: In this wallet type, the hub wallet is an encrypted (using PKCS #12 standard) and password protected file (**HVR_CONFIG/files/hvrwallet-***hubname***/hvrwallet-***timestamp***.p12**) that stores the hub encryption key. The password for the software wallet (file) should be supplied by the user while creating the software wallet.

   Prior to HVR 5.7.0/0, the software wallet file name and path was **HVR_CONFIG/files/hvrwallet-***hubname***-***timestamp***.p12**

   When the command to create the software wallet is executed, HVR creates a hub wallet configuration file and a software wallet file. It also generates a hub encryption key inside the software wallet file.

   Whenever the hub process needs the hub encryption key (to encrypt/decrypt data), it opens the software wallet by supplying the wallet password and fetches the decrypted hub encryption key from the software wallet file (.**p12**) and then stores it in the hub process's memory. Note that the hub encryption key is decrypted only after the wallet password is supplied.

   Whenever the hub encryption key is rotated or the software wallet password is changed, the hub encryption key is actually migrated to a new software wallet.

2. **KMS Wallet**: In this wallet type, the hub wallet is a network service (AWS KMS) that encrypts the hub encryption key. The encrypted hub encryption key is stored in the hub wallet configuration file. The KMS wallet is protected either by the AWS KMS credentials or the AWS IAM role, which should be supplied by the user while creating the KMS wallet.

   When the command to create the KMS wallet is executed, HVR contacts KMS using the credentials supplied /configured. The KMS then generates the hub encryption key, encrypts it, and then sends it to HVR. HVR will save the hub encryption key in the wallet configuration file. Depending on the authentication method chosen for the KMS wallet, the **KMS Access Key Id** or **KMS IAM Role** should be supplied with the KMS wallet create command, this will be saved in the wallet configuration file.

   - If the authentication method is **KMS Access Key Id**, the wallet password is the secret access key of the AWS IAM user used for connecting HVR to KMS.
   - If the authentication method is **KMS IAM Role**, there is no separate wallet password required since the authentication is done based on the AWS IAM Role. This authentication mode is used when connecting HVR to AWS S3 by using AWS Identity and Access Management (IAM) Role. This option can be used only if the HVR remote agent or the HVR Hub is running inside the AWS network on an EC2 instance and the AWS IAM role specified here should be attached to this EC2 instance. When a role is used, HVR obtains a temporary Access Keys Pair from the EC2 machine. For more information about IAM Role, refer to IAM Roles in AWS documentation.

   Whenever the hub process needs the hub encryption key (to encrypt/decrypt data), it fetches the encrypted hub encryption key from the wallet configuration file and sends it to AWS KMS. The KMS then decrypts the hub encryption key and sends it back to HVR, which is then stored in the hub process's memory.

   For the KMS wallet which is based on the **KMS Access Key Id** authentication, the wallet password is changed whenever the KMS credential (secret access key of the IAM user) is updated.

Hub Encryption Key Rotation

The hub encryption key should periodically be rotated to meet the cryptographic best practices and industry standards. Rotating the hub encryption key is when you retire the existing key and replace it with a newly generated key. The hub encryption key rotation helps to keep data leaks bounded in case the hub encryption key is stolen/compromised.

When the command to rotate the hub encryption key is executed, the following process takes place:

1. Generate new hub encryption key.
2. Decrypt catalog table data with the old hub encryption key.
3. Encrypt catalog table data with the new hub encryption key.

> The hub encryption key rotation does not re-encrypt data outside of HVR catalogs such as Job scripts, Windows services, UNIX crontabs, and HVR GUI configuration. These non-catalog items (services) have the hub encryption key in their memory. When the hub encryption key is rotated, the process memory will still have the old key. To obtain the new key, the non-catalog items (services) need to be restarted by the user manually, and for services that store the key, the script or service must be recreated.

After the hub encryption key rotation, the old hub encryption key is deactivated, encrypted with the newest key, and retained. It is retained to decrypt the data that was encrypted using the old hub encryption key. During the rotation and a short time afterwards, the old hub encryption key still needs to be available for HVR. After the rotation, it might be needed for non-catalogs items such as Job scripts, Windows services, UNIX crontabs, and HVR GUI configuration.

The hub encryption key has a unique sequence number to maintain the history or keep track of all versions of the hub encryption key.

History

The old hub encryption keys are stored in the hub wallet configuration file, which is protected with the new hub encryption key. HVR keeps the history of hub encryption key in the hub wallet configuration file in a JSON format.

The old/history hub encryption keys retained in hub wallet configuration file can be purged/deleted (**hvrwalletconfig** option **-d** or **-S** or **-T**) to avoid compromise/leakage of these keys.

Hub Wallet Migration

Moving the hub encryption key from one wallet to another is called wallet migration.

Following are the two modes of wallet migration:

- **Wallet migration with the same hub encryption key**
  In this mode, during the wallet migration, the hub encryption key does not change; instead only the wallet storage is changed. The encrypted wallet storage is first decrypted and the hub encryption key is moved to the new wallet that is encrypted. During wallet migration, both the old wallet and new wallet must be available simultaneously while the encryption key is decrypted. After the wallet migration completes, the old wallet is discarded.

- **Wallet migration with a new hub encryption key**
  In this mode, during the wallet migration, the hub encryption key is replaced (rotated) with a new hub encryption key and the wallet storage is changed. The encrypted wallet storage is first decrypted and the hub encryption key is rotated and then it is moved to the new wallet that is encrypted. During wallet migration, both the old wallet and new wallet must be available simultaneously while the encryption key is decrypted. After the wallet migration completes, the old wallet is discarded.

See also, **hvrwalletconfig** (option **-m**), and section Migrating a Hub Wallet in Configuring and Managing Hub Wallet.

Migration Scenarios

The following scenarios/conditions lead to the migration of hub wallet.

In the software wallet, migration happens if a user:

- Switches to another wallet type.
- Changes the wallet password.
  - When the command to change the software wallet password is executed, HVR creates a new wallet file (**.p12**) with a new password and then moves the hub encryption key from the existing wallet file to the new file. This is the reason for using **hvrwalletconfig** with option **-m** while changing the software wallet password.
- Rotates the hub encryption key.
  - When the hub encryption key is rotated, the existing hub encryption key stored in the software wallet file (**.p12**) is retired and it is replaced with a newly generated hub encryption key in a new software wallet file.

In the KMS wallet, migration happens if a user:

- Switches to another wallet type.
- Changes KMS credentials and uses **hvrwalletconfig** with option **-m**.
  - If option **-m** is not used, the KMS credential change is considered as a configuration update. The updated configuration is saved in the wallet configuration file.
- Changes KMS Customer Master Key (CMK) ID and uses **hvrwalletconfig** with option **-m**.
  - If option **-m** is not used, the KMS CMK ID change is considered as a configuration update. The updated configuration is saved in the wallet configuration file.

During the migration, the old KMS account will be accessed by HVR for decryption.

See Also

- Configuring and Managing Hub Wallet
- hvrwalletconfig
- hvrwalletopen
- Encrypted Network Connections

Configuring and Managing Hub Wallet

Since v5.6.5/5

Contents

This section describes the steps to create, enable, disable, delete, rotate encryption key, migrate wallet, or configure the hub wallet.

> The argument *hubdb* used in the command examples specifies the connection to the hub database. For more information about supported hub databases and the syntax for using this argument, see Calling HVR on the Command Line.

Creating and Enabling Hub Wallet

To use the hub wallet and encryption feature of HVR, first the hub wallet should be created and then the encryption enabled by defining the encryption data category.

Creating and Enabling Software Wallet

Following are the steps to create and enable the software wallet:

1. Create software wallet. Set the wallet password and the wallet type. Following is the command (**hvrwalletconfig**) to set the wallet type as **SOFTWARE**.

   ```
   hvrwalletconfig -p hubdb Wallet_Type=SOFTWARE
   ```

   After executing this command, a prompt asking to set a password for the software wallet will be displayed.
2. Enable encryption. Following is the command that will instruct HVR to start encryption of data (this includes the existing data in the hub database). The category of data to be encrypted depends on the property **Encryption** defined in this command.

   ```
   hvrwalletconfig hubdb Encryption=category_of_data
   ```

Creating and Enabling KMS Wallet

Following are the steps to create and enable the KMS wallet:

1. Set the wallet password and the wallet type. Select your preferred credential method for the KMS wallet:

   - If the authentication method is **KMS Access Key Id**, following is the command (**hvrwalletconfig**) to set the wallet type as **KMS**. The KMS connection properties **Wallet_KMS_Region**, **Wallet_KMS_Customer_Master_Key_Id**, **Wallet_KMS_Access_Key_Id** should be defined in this command.

     ```
     hvrwalletconfig -p hubdb Wallet_Type=KMS Wallet_KMS_Region=eu-west-1
     Wallet_KMS_Customer_Master_Key_Id=1234abcd-12ab-1234590ab
     Wallet_KMS_Access_Key_Id=AKIAJDRSJY123QWERTY
     ```

     After executing this command, a prompt asking to set a password for the hub wallet will be displayed. The password for the KMS (secret key) should be supplied.
   - If the authentication method is **KMS IAM role**, following is the command (**hvrwalletconfig**) to set the wallet type as **KMS**. The KMS connection properties **Wallet_KMS_Region**, **Wallet_KMS_Customer_Master_Key_Id**, **Wallet_KMS_IAM_Role** should be defined in this command.

     ```
     hvrwalletconfig hubdb Wallet_Type=KMS Wallet_KMS_Region=eu-west-1
      Wallet_KMS_Customer_Master_Key_Id=1234abcd-12ab-1234590ab Wallet_KMS_IAM_Role=
     PRODROLE
     ```

     > For **KMS IAM role**, since the authentication is done based on the AWS IAM Role, **hvrwalletconfig** option **-p** is not required in the command. For more information, see description for the **KMS Wallet** in section Hub Wallet Types of Hub Wallet and Encryption.

2. Enable encryption. Following is the command (**hvrwalletconfig**) that will instruct HVR to start encryption of data (this includes the existing data in the hub database). The category of data that is encrypted depends on the property **Encryption** defined in this command.

   ```
   hvrwalletconfig hubdb Encryption=category_of_data
   ```

Auto-Open Password

This section describes the steps/commands to enable or disable the auto-open password method for supplying the wallet password. For more information the about auto-open password method, see section Methods to Supply Wallet Password in Hub Wallet and Encryption.

Enable Auto-Open Password

- To enable the auto-open password method, option **-P** of the **hvrwalletconfig** command is required. Following is the command to enable auto-open password method:

  ```
  hvrwalletconfig -p -P hubdb
  ```

After executing this command, a prompt asking to supply the wallet password is displayed.

Disable Auto-Open Password

- To disable the auto-open password method, the property **Wallet_Auto_Open_Password** should be left blank. Following is the command to disable auto-open password method:

```
hvrwalletconfig -p hubdb Wallet_Auto_Open_Password=
```

Auto-Open Plugin

This section describes the steps/commands to enable or disable the auto-open plugin method for supplying the wallet password. For more information about the auto-open plugin method, see section Methods to Supply Wallet Password in Hub Wallet and Encryption.

Enable Auto-Open Plugin

- To enable the auto-open plugin method, the property **Wallet_Auto_Open_Plugin** should point to the plugin file directory. Following is the command to enable auto-open plugin method:

```
hvrwalletconfig hubdb Wallet_Auto_Open_Plugin=/home/user/myplugin.sh
```

Disable Auto-Open Plugin

- To disable the auto-open plugin method, the property **Wallet_Auto_Open_Plugin** should be left blank. Following is the command to disable auto-open plugin method:

```
hvrwalletconfig hubdb Wallet_Auto_Open_Plugin=
```

Rotating Hub Encryption Key

This section describes the steps/commands to rotate the hub encryption key and also view and manage the hub encryption key history. For more information, see section Hub Encryption Key Rotation in Hub Wallet and Encryption.

- Following is the command to rotate the hub encryption key:

```
hvrwalletconfig -r hubdb
```

- Following is the command to view the hub encryption key history sequences and rotation timestamps:

```
hvrwalletconfig hubdb Encryption_Key_History
```

- Following is the command to view the sequence number of the current hub encryption key:

```
hvrwalletconfig hubdb Encyption_Key_Sequence
```

- Following is the command to view the entire wallet configuration, including history and other wallet settings:

```
hvrwalletconfig hubdb
```

Migrating Hub Wallet

This sections describes the steps/commands for migrating the wallet. For more information, see section Hub Wallet Migration in Hub Wallet and Encryption.

Migrating to Software Wallet

Following is the command to migrate to the software wallet.

```
hvrwalletconfig -p -m hubdb Wallet_Type=SOFTWARE
```

Migrating to KMS Wallet

Following are the commands to migrate to the KMS wallet:

- To migrate to a KMS wallet with authentication method as **KMS Access Key Id**,

```
hvrwalletconfig -p -m hubdb Wallet_Type=KMS Wallet_KMS_Region=eu-west-1
Wallet_KMS_Customer_Master_Key_Id=1234abcd-12ab-1234590ab Wallet_KMS_Access_Key_Id=
AKIAJDRSJY123QWERTY
```

- To migrate to a KMS wallet with authentication method as **KMS IAM role**,

```
hvrwalletconfig -m hubdb Wallet_Type=KMS Wallet_KMS_Region=eu-west-1
   Wallet_KMS_Customer_Master_Key_Id=1234abcd-12ab-1234590ab Wallet_KMS_IAM_Role=
PRODROLE
```

Changing Software Wallet Password

The password for the software wallet can be changed, if required.

- Following is the command to change the password for the software wallet:

```
hvrwalletconfig -p -m hubdb
```

> When the command to change the software wallet password is executed, HVR creates a new wallet file (**.p12**) with a new password and then moves the hub encryption key from the existing wallet file to the new file. This is the reason for using **hvrwalletconfig** with option **-m** while changing the software wallet password.

Disabling and Deleting Hub Wallet

The hub wallet can be deleted if it is not required anymore or if the wallet itself is not accessible, etc. The hub wallet password is required for disabling and deleting the hub wallet. The artifacts (hub encryption key sequence and key history) can be deleted or retained while deleting the hub wallet.

If the wallet password is forgotten, the hub wallet can be deleted by following the steps mentioned in force deletion without wallet password.

Following are the steps to delete the software wallet:

1. Stop **HVR Scheduler**.
2. Disable encryption. Following is the command (**hvrwalletconfig**) that will instruct HVR to stop encrypting all passwords and also to decrypt the existing passwords in the hub database and obfuscate them.

```
hvrwalletconfig -p hubdb Encryption=NONE
```

> Encryption cannot be disabled if the hub wallet is not accessible (due to a corrupted software wallet file or inaccessible KMS, etc) or if the wallet password is wrong/forgotten.

3. Delete the wallet. One of the following command (**hvrwalletconfig**) can be used to delete the hub wallet and artifacts.

- To delete only the hub wallet and retain the artifacts,

```
hvrwalletconfig -p -da hubdb
```

> In case the hub wallet is not accessible anymore (due to a corrupted software wallet file or inaccessible KMS), then use the following command to force delete the wallet and retain the artifacts:
>
> ```
> hvrwalletconfig -p -daf hubdb
> ```

- To delete the hub wallet and artifacts,

```
hvrwalletconfig -p -dA hubdb
```

> In case the hub wallet is not accessible anymore (due to a corrupted software wallet file or inaccessible KMS), then use the following command to force delete the wallet and the artifacts:
>
> ```
> hvrwalletconfig -p -dAf hubdb
> ```

Force Deletion Without Wallet Password

If the hub wallet password is forgotten, then use the following command to force delete the hub wallet:

- Since the wallet password is not supplied in this command, the artifacts cannot be retained.
- The encryption is not disabled before deleting the wallet forcefully, so the information that were encrypted by using this wallet will have to be manually fixed by the user (for example, entering the password again in the location connection screen).

1. Force delete the wallet and artifacts:

```
hvrwalletconfig -dAf hubdb
```

2. Set encryption to **NONE**. The following command is optional.

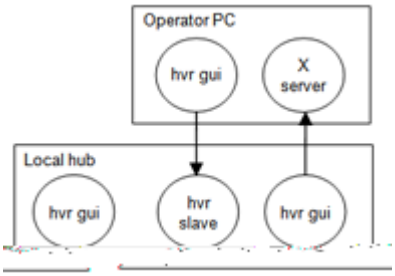
```
hvrwalletconfig hubdb Encryption=NONE
```

Setting encryption to **NONE** after deleting the wallet will not decrypt the passwords that were encrypted using the wallet. This command may be executed to avoid any problems while creating a new hub wallet in the future.

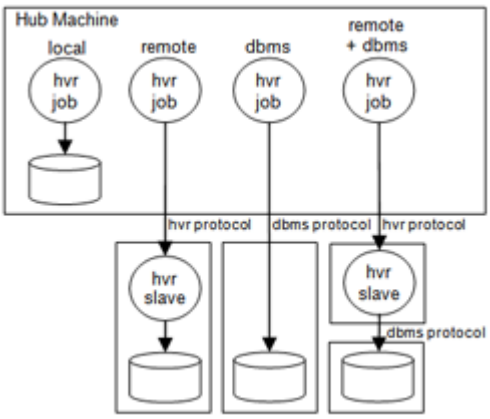
Network Protocols, Port Numbers and Firewalls

In order to configure the connections between the hub machine, remote HVR locations and an operator's PC, it is important to understand which TCP/IP ports HVR uses at runtime. The operator can control replication by logging into the hub machine and using command line commands or running the GUI from the hub. Alternatively, the HVR GUI can be used from the operator's PC, in which case several TCP/IP ports must be opened from the PC to the hub machine.

Protocol Options from GUI to Hub



Protocol Options from Hub to Replication Location



Connection	TCP/IP Port Number	Network Protocol (non-encrypted)	Network Protocol (encrypted)
HVR GUI connecting from PC to hub machine.	Arbitrary port, typically **4343**. On Unix and Linux the listener is the **inetd** daemon. On Windows the listener is the **HVR Remote Listener** service.	HVR internal protocol	
Replicating from hub machine to remote location using HVR remote connection.	Arbitrary port, typically **4343**. On Unix and Linux the listener is the **inetd** daemon. On Windows the listener is the **HVR Remote Listener** service.	HVR internal protocol	SSL encryption can be enabled using HVR action **LocationProperties /SslRemoteCertificate** and **/SslLocalCertificateKeyPair**.
Replicating from remote location using DBMS protocol.	DBMS dependent.	Oracle's TNS protocol, or Ingres /NET or SQL Server protocol	Database dependent

⚠ When connecting to Oracle RAC, HVR first connects to the SCAN listener port, after which it connects to the HVR agent installation. In a RAC setup, the HVR remote listener must be running on the same port (e.g. **4343**) on every node.

TCP Keepalives

HVR uses TCP keepalives. TCP keepalives control how quickly a socket will be disconnected if a network connection is broken.

By default, HVR enables TCP keepalives (**SO_KEEPALIVE** is true). TCP keepalives can be disabled by setting the environment variable **$HVR_TCP_KEEPALIVE** to value **0**.

On some platforms (Linux, Windows and AIX from version 5.3 and higher), the environment variable **$HVR_TCP_KEEPALIVE** can also be used to tune keepalive frequency. It can be set to a positive integer. The default is **10** (seconds). The first half of this time (e.g. first five seconds) is passive, so no keepalive packets are sent. The other half is active; HVR socket sends ten packets (e.g. 4 per second). If no response is received, then the socket connection is broken.

Regular Maintenance and Monitoring for HVR

Periodically it is necessary to restart the **HVR Scheduler**, purge old log files and check for errors. HVR must also be monitored in case a runtime error occurs.

Both maintenance and monitoring can be performed by script **Hvrmaint**. This is a standard Perl script which does nightly or weekly housekeeping of the **HVR Scheduler**. It can be scheduled on Unix or Linux using **crontab** or on Windows as a **Scheduled Task**.

HVR can also be watched for runtime errors by an enterprise monitoring system, such as TIVOLI or UNICENTER. Such monitoring systems complement **Hvrmaint** instead of overlapping it. Such systems can watch HVR in three ways:

- Check that the **Hvrscheduler** process is running in the operating system process table.
- Check that no errors occur in file **$HVR_CONFIG/log/***hubdb***/hvr.crit** or in file **$HVR_ITO_LOG** (see section **Hvrscheduler**).
- Check that file **$HVR_CONFIG/log/***hubdb***/hvr.out** is growing.

HVR High Availability

Contents

Introduction

The ability to replicate data between data stores has a number of dependencies. Source and target databases/data stores must be available and accessible, all of the infrastructure involved in the data replication must be available and allow connectivity. The software that makes data replication possible must work as designed. At any point in time, one or more of the components may fail.

This document describes how to configure HVR data replication for High Availability (HA). Fundamentally, the software is designed to resume replication at the point where replication was stopped. With an approach to HA data replication, downtime is either avoided or limited so that data keeps flowing between source(s) and target(s).

Backup, Disaster Recovery, High Availability

There are multiple strategies to get a setup that is no longer functioning back into a functioning state.

- A backup is a copy of your application data. If the data replication setup fails due to a component failure or corruption, it can be restored on the same or new equipment, and from there data replication can be recovered.
- Disaster Recovery (DR) is a completely separate setup that can take over in case of a major event - a disaster such as flooding or an earthquake - taking out many components at the time e.g. entire data centers, the electricity grid, or network connectivity for a large region.
- High Availability (HA) is a setup with no single point of failure. HA introduces redundancy into the setup to allow for components to step in if there is a failure.

What availability/recovery strategy or combination of strategies works best for your organization depends on a number of factors, including the complexity of the environment, the available budget to ensure availability, and the extent of replication downtime your organization can afford.

Recovery Time Objective

An important HA consideration is the impact of downtime on operations and with that the business cost to replication downtime. When replication is down, data is no longer flowing between source(s) and target(s). If the replication downtime is unrelated to the availability of source(s) and target(s) then latency will increase until replication is restored. However, data will still be available on the target, getting staler as time goes on, with current data available in the source. Facing high latency is similar to replication unavailability.

This paper discusses HA for data replication. The time period for which you can sustain the situation with data replication down determines your so-called Recovery Time Objective (RTO): how long can you afford for replication to be down. For some organizations, the RTO will be as low as minutes if not less, for example, if data replication is a core part of a broader application high availability strategy for a mission-critical system. Other organizations may have the

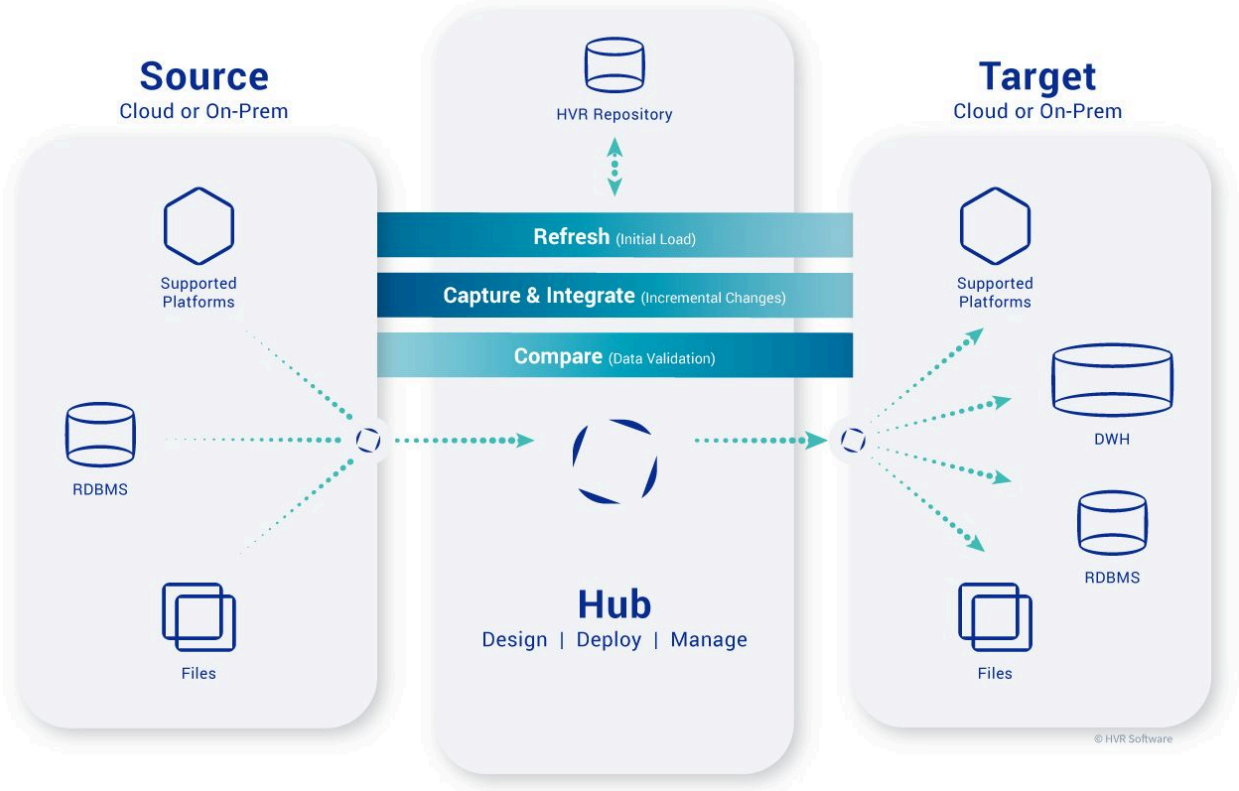
flexibility to redirect workloads allowing them to afford a replication downtime of multiple minutes if not hours. Some organizations may run non-mission-critical workloads on their replicated data sets and some occasional data replication downtime can be coped with relatively easily.

Weigh your RTO against the cost and complexity of implementing HA as you architect your data replication configuration.

HVR Architecture

HVR's real-time replication solution features a distributed approach to log-based Change Data Capture (CDC) and continuous integration. The HVR software consists of a single installation that can act as a so-called hub controlling data replication, or as an agent performing work as instructed by the hub. The hub and agents communicate over TCP/IP, optionally using encrypted messages. However, the architecture is flexible. Any installation of the software can act as an agent or as a hub, and technically the use of agents in a setup is optional.

The image below shows the distributed architecture, with, in this case, a single source agent and a single target agent. Real-world deployments often use a separate agent per source and a separate agent per target.



The distributed architecture provides several benefits:

- Offloading/distributing some of the most resource-intensive processing to avoid bottleneck processing changes centrally.
- Optimizing network communication with data compressed by the agent before sent over the wire.
- Improved security with unified authorization and the ability to encrypt data on the wire.

Agents

Agents are typically installed close to the source database(s) if not on the database server(s), and close to the destination systems. Agents store very limited state information (configuration dependent). Based on the state information stored on the hub, data replication can always resume at the point of stoppage if replication is stopped or fails . A single hub may orchestrate data replication between hundreds of agents.

Hub

The HVR hub is an installation of HVR software that is used to configure data replication flows. Metadata is stored in a repository database that can be any one of the relational databases that are supported as a source or Teradata. The hub also runs a scheduler to start jobs and make sure jobs are restarted if they fail. During operations, the scheduler requires a connection to the repository database. The repository database can either be local to the host running the hub, or remote using a remote database connection.

The main functions of the hub are:

- Maintain the replication state.
- Restart/resume replication in case of a failure.
- Route compressed transaction files arriving from the source(s) to the correct target(s).
- Be the access point for operator maintenance and monitoring.

Environments with a dedicated hub server will see the bulk of data replication processing taking place on the agent servers, with very little data replication load on the hub.

High Availability for HVR Agents

Data replication is recoverable based on the state information stored on the hub. High availability for an agent can be achieved by having a duplicate installation for the agent available. HVR identifies an agent in the so-called Location Configuration through a host name or IP-address where the HVR remote listener must be running to establish a connection. The remote listener is a light-weight process, similar to a database listener, that forks new processes to perform work.

If the agent runs on the source or target database server then the agent should be available if the server is available, and the remote listener is running. Make sure to configure the remote listener to start upon system startup through systemd, (x)inetd or hvr_boot on Linux/Unix or a service on Windows. See Installing HVR on Unix or Linux for more details.

Consider using a floating virtual IP-address/host name to identify the agent, or use a load balancer in front of the agent for automatic failover.

Cloud providers have services available that can help implement high availability for agents.

State Information Stored by HVR Agent

An HVR agent performing CDC on the source may store state information about long-running open transactions to prevent a long recovery time when the capture process restarts. By default, every five minutes the capture process writes a so-called checkpoint to disk, storing the in-memory state of the transactions open longer than 5 minutes. Upon restart, the most recent complete checkpoint becomes the starting point of CDC, followed by re-reading (backed up) transaction logs from the point of the checkpoint forward, ensuring no data changes are lost.

Long-running transactions may occur on Oracle databases, especially when using packaged applications such as SAP or Oracle eBusiness Suite. Long-running transactions are less likely on other databases. The default location to store the checkpoint is at the agent, but setting option **CheckPointStorage** to HUB for action **Capture** will move the checkpoints to the hub so that in case of a restart, the checkpoint stored on the hub can be the starting point. Note that storing the checkpoint on the hub will take more time than storing the checkpoint locally where the agent runs.

The integration agent stores state information:

- in the target, in the so-called state tables if the target is a database.
- implicitly using a directory **hvr_state** if the target is a file system (except for S3 as a file system, when the **hvrmanifestagent** should be used to publish files).

With the agent state information stored in the actual target location, data replication can leverage the current state irrespective of whether the same or a different instance of an agent is used. here is no need to consider integrate state when considering high availability for the integration agents.

High Availability for HVR Hub

For replication to function, the hub's scheduler must be running, which requires a connection to the repository database to retrieve and maintain the job state (e.g. **RUNNING**, **SUSPENDED**, etc.). Data replication definitions stored in the same repository are not required at runtime. However, data replication can be re-instantiated from the current definitions.

Any state information about the data replication beyond just the job state is stored on the file system in a directory identified by the environment setting **HVR_CONFIG**. To resume replication where it left off prior to a failure, files in this directory must be accessible.

High availability setup for the HVR hub requires the **Scheduler** to run, which requires:

- Repository database to be accessible (note that the repository database can be remote from the hub server or local to it).
- Access to up-to-date and consistent data in the **HVR_CONFIG** directory.

A common way to implement high availability for the HVR hub is to use a cluster with shared storage (see Installing HVR in a Cluster for installation instructions). In a clustered setup, the cluster manager is in charge of the HVR scheduler as a cluster resource, making sure that within the cluster only one hub scheduler runs at any point in time. File system location **HVR_CONFIG** must be on the attached storage, shared between the nodes in the cluster or switched over during the failover of the cluster (e.g. in a Windows cluster). If the repository database is local to the hub like an Oracle RAC Database or a SQL Server AlwaysOn cluster, then the connection to the database can always be established to the database on the local node, or using the cluster identifier. For a remote database (network) connectivity to the database must be available, and the database must have its own high availability setup that allows remote connectivity in case of failure.

Cloud environments provide services to support an HVR high availability configuration for the **HVR_CONFIG** file system like Elastic File System (EFS) on Amazon Web Services (AWS), Azure Files on Microsoft Azure, and Google Cloud Filestore on Google Cloud Services (GCS). The cloud providers also provide database services with built-in high availability capabilities and redundancies in the connectivity to allow for failures without impacting availability.

Recovering HVR Replication

If HVR replication fails and there is no high availability setup, or for whatever reason, the high availability configuration was unable to prevent an outage (e.g. in a perfect storm of failures or a disaster scenario, when DR could have provided business continuity but high availability cannot), then you must recover replication. How can you do this?

Restore and Recover from Backup

Make sure to always have a current backup of your HVR definitions by using the **Export Catalogs** function. In the worst case, with data replication otherwise unavailable, you can restore the environment from the export of the data definitions:

- Create a new repository database (with empty tables).
- Import the export file.
- Configure **Resilient** processing on **Integrate** (temporarily) to instruct HVR to merge changes into the target.

> ⚠ For a target that writes an audit trail of changes (i.e. configured using **TimeKey**), you will need to identify the most recently applied transaction and use this information to re-initialize data replication to avoid data overlap, or you must have downstream data consumption cope with possible data overlap.

- Initialize the data replication channels, using **Capture Rewind** (if possible, i.e. if transaction file backups are still available and accessible) to avoid loss of data.
- If backup transaction log files (or archived logs) are no longer available, then use **HVR Refresh** to re-sync the table definitions between a source and a target. Depending on the target and data volumes, a row-by-row Refresh, also referred to as repair, may be appropriate, and/or use action **Restrict** to define a suitable filter condition to re-sync the data.

Consider the backup retention strategy for your transaction logs if your recovery strategy for data replication includes recovery from a backup to allow for replication to recover by going through the backups of the transaction log files rather than having to re-sync from the source directly.

Disaster Recovery

To recover data replication from a disaster recovery environment is not unlike the recovery from a backup, except that the recovery time is likely lower, because the environment is ready and available. The disaster recovery environment may even be able to connect to the primary repository database for up-to-date data replication definitions and current job state information. However, the disaster recovery environment would not have access to an up-to-date **HVR_CONFIG** location so the data replication state would have to be recreated.

With this, the steps to implement a disaster recovery environment are:

- Configure Resilient processing on **Integrate** (temporarily) to instruct HVR to merge changes into the target.

> ⚠ For a target that writes an audit trail of changes (i.e. configured using **TimeKey**) you will need to identify the most recently applied transaction and use this information to re-initialize data replication to avoid data overlap, or you must have downstream data consumption cope with possible data overlap.

- **Initialize** the data replication channels, using **Capture Rewind** (if possible, i.e. if transaction file backups are still available and accessible) to avoid loss of data.
- If backup transaction log files (or archived logs) are no longer available, then use **HVR Refresh** to re-sync the table definitions between a source and a target.

Disaster Recovery - Using Heartbeat Table

To restore data replication from a backup or even in a disaster recovery environment is challenging because the data replication state, from **HVR_CONFIG** on the failed primary environment, is not available. With the loss of **HVR_CONFIG**, the following state information is lost:

- Capture status stored in the so-called **cap_state** file. Most importantly for recovery, this file includes the transaction log's position of the beginning of the oldest open transaction capture is tracking.
- Transaction files that have yet to be integrated into the target. Depending on the setup, it may be normal that there are hundreds of MBs waiting to be integrated into the target, and regularly copying these (consistently) to a disaster recovery location may be unrealistic.
- Table Enrollment information containing the mapping between table names and object IDs in the database. This information doesn't change frequently and it is often safe to recreate the Table Enrollment information based on current database object IDs. However, there is a potential for data loss if data object IDs did change, for example for a SQL Server source if primary indexes were rebuilt between the point in time where capture resumes and the current time.

The concept of the heartbeat table includes:

- A heartbeat table on every source database that HVR captures from. The table can be created in the application schema or in a separate schema that the HVR capture user has access to (for example, the default schema for the HVR capture user).
- The heartbeat table becomes part of every data replication channel that captures from this source and is integrated into every target.
- The heartbeat table is populated by a script, taking current capture state information from the capture state file and storing it in the database table. With the table being replicated, the data will make it into the target as part of data replication. Optionally, the heartbeat table also includes current Table Enrollment information.
- The script to populate the heartbeat table can be scheduled to run by the OS scheduler on the hub server, or may be invoked as an agent plugin as part of the capture cycle (with optionally some optimizations to limit the overhead to store state information as part of the capture cycle).

- Data replication recovery reads the state of the heartbeat table on the target, allowing it to recreate the capture state file to avoid data loss due to missing long-running transactions, and regenerate any transaction files that were not yet applied to the target will be recreated (since the heartbeat table becomes part of the regular data replication stream).

Using a heartbeat table requires privileges and database objects beyond the out-of-the-box HVR installation, as well as setup steps that go beyond regular running of HVR. Please contact your HVR representative should you need professional services help to implement these.

Conclusion

Data replication availability has a number of dependencies, including the source(s) and target(s) availability, open network communication, and software to function as designed. Individual components in such a complex setup may fail, resulting in replication downtime. However, downtime in replication doesn't necessarily mean no access to data. Data in the target(s) would be stale and become staler as time progresses, similar to data replication introducing latency.

This paper discusses strategies to achieve High Availability (HA) for HVR replication. To what extent you invest in an HA setup depends on your Recovery Time Objective (RTO) related to the direct cost or risk of replication downtime, but also your willingness to manage a more complex setup. Alternatives to high availability include restore and recovery from a backup, using a disaster recovery (DR) environment, and automating the disaster recovery through a custom heartbeat table.

Location Class Requirements

This section describes the pre-requisites, access privileges, and other configuration required for using HVR to capture or integrate changes into sources and targets (referred to as the location classes), listed below.

Requirements for Azure Blob FS

Since v5.5.5/4

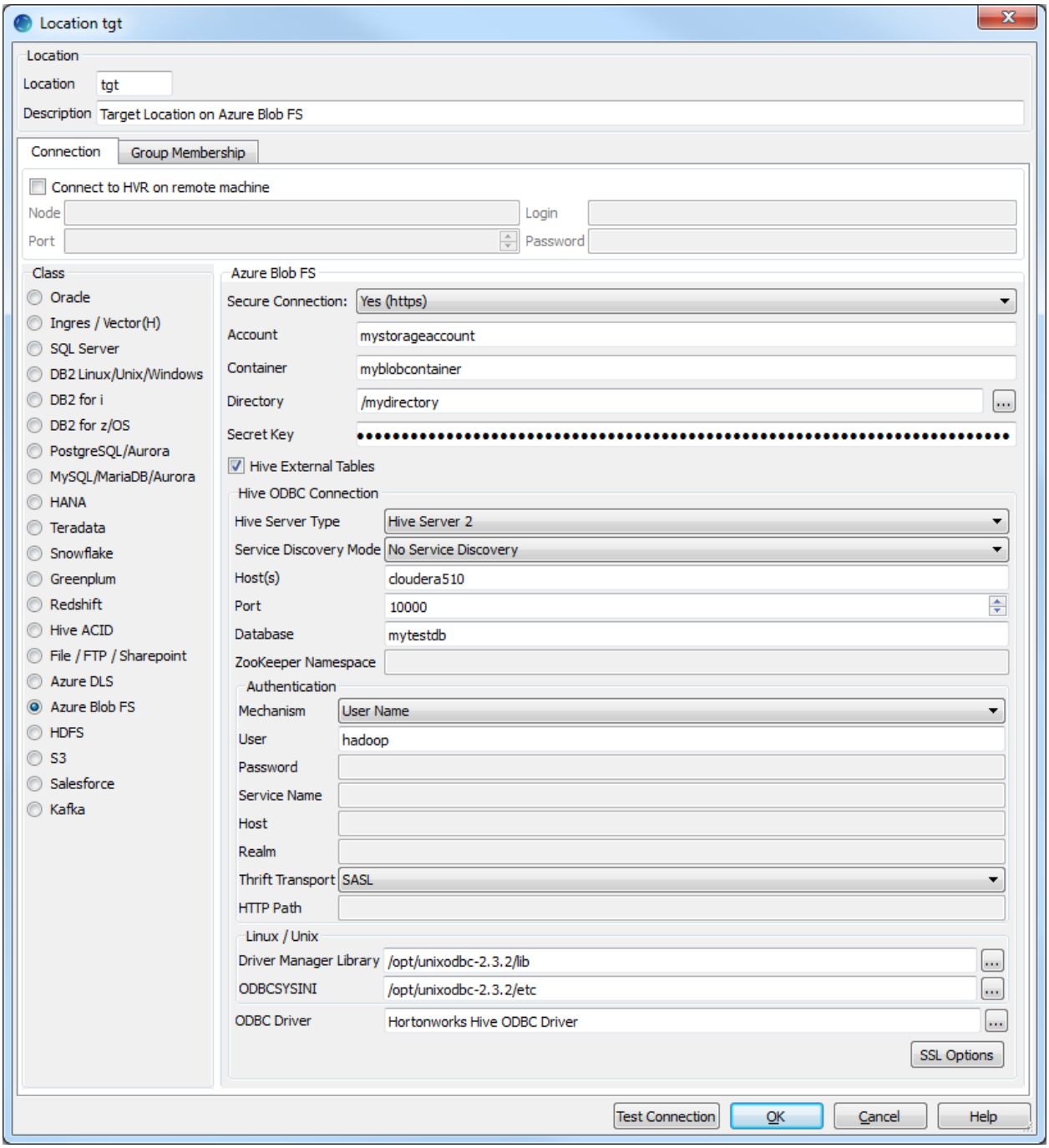
Contents

This section describes the requirements, access privileges, and other features of HVR when using Azure Blob FS for replication. For information about compatibility and support for Azure Blob FS with various HVR platforms, see Platform Compatibility Matrix.

For the capabilities supported by HVR, see Capabilities.

Location Connection

This section lists and describes the connection details required for creating Azure Blob FS location in HVR.



Field	Description
Azure Blob FS	
Secure connection	The type of security to be used for connecting to Azure Blob Server. Available options: • **Yes (https)** (default) : HVR will connect to Azure Blob Server using HTTPS. • **No (http)** : HVR will connect to Azure Blob Server using HTTP.
Account	The Azure Blob storage account. **Example:** mystorageaccount
Container	The name of the container available within storage **Account**. **Example:** myblobcontainer
Directory	The directory path in **Container** which is to be used for replication. **Example:** /folder
Secret key	The access key of the storage **Account**.

Hive External Tables	Enable/Disable Hive ODBC connection configuration for creating Hive external tables above Azure Blob FS.

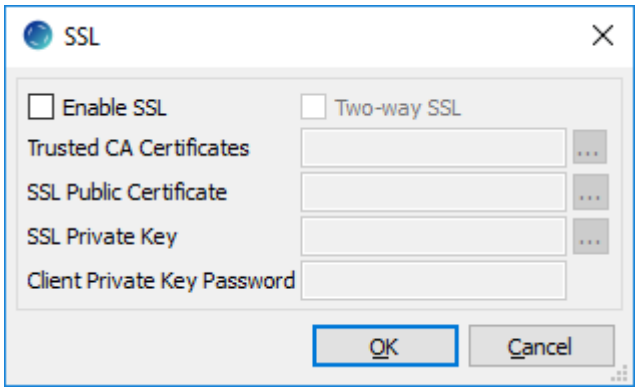
Hive ODBC Connection

HVR allows you to create 5702076 above Azure Blob FS files which are only used during compare. You can enable /disable the Hive configuration for Azure Blob FS in location creation screen using the field **Hive External Tables** . For more information about configuring Hive external tables, refer to Hadoop Azure Blob FS Support documentation.

Field	Description
Hive ODBC Connection	
Hive Server Type	The type of Hive server. Available options: • **Hive Server 1** (default): The driver connects to a Hive Server 1 instance. • **Hive Server 2**: The driver connects to a Hive Server 2 instance.
Service Discovery Mode	The mode for connecting to Hive. This field is enabled only if **Hive Server Type** is **Hive Server 2**. Available options: • **No Service Discovery** (default): The driver connects to Hive server without using the ZooKeeper service. • **ZooKeeper**: The driver discovers Hive Server 2 services using the ZooKeeper service.
Host(s)	The hostname or IP address of the Hive server. When Service Discovery Mode is ZooKeeper, specify the list of ZooKeeper servers in following format is
	• • • • •

Service Name	The Kerberos service principal name of the Hive server. This field is enabled only if **Mechanism** is **Kerberos**.
Host	The Fully Qualified Domain Name (FQDN) of the Hive Server 2 host. The value of **Host** can be set as **_HOST** to use the Hive server hostname as the domain name for Kerberos authentication. If **Service Discovery Mode** is disabled, then the driver uses the value specified in the Host connection attribute. If **Service Discovery Mode** is enabled, then the driver uses the **Hive Server 2** host name returned by ZooKeeper. This field is enabled only if **Mechanism** is **Kerberos**.
Realm	The realm of the Hive Server 2 host. It is not required to specify any value in this field if the realm of the Hive Server 2 host is defined as the default realm in Kerberos configuration. This field is enabled only if **Mechanism** is **Kerberos**.
Thrift Transport **Since** v5.5.0/2	The transport protocol to use in the Thrift layer. This field is enabled only if **Hive Server Type** is **Hive Server 2**. Available options: • **Binary** (This option is available only if **Mechanism** is **No Authentication** or **User Name and Password**.) • **SASL** (This option is available only if **Mechanism** is **User Name** or **User Name and Password** or **Kerberos**.) • **HTTP** (This option is not available if **Mechanism** is **User Name**.) For information about determining which Thrift transport protocols your Hive server supports, refer to HiveServer2 Overview and Setting Up HiveServer2 sections in Hive documentation.
HTTP Path **Since** v5.5.0/2	The partial URL corresponding to the Hive server. This field is enabled only if **Thrift Transport** is **HTTP**.
Linux / Unix	
Driver Manager Library	The directory path where the Unix ODBC Driver Manager Library is installed. **Example:** /opt/unixodbc-2.3.2/lib
ODBCSYSINI	The directory path where **odbc.ini** and **odbcinst.ini** files are located. **Example:** /opt/unixodbc-2.3.2/etc
ODBC Driver	The user defined (installed) ODBC driver to connect HVR to the Hive server.
SSL Options	Show **SSL Options**.

SSL Options



Field	Description
Enable SSL	Enable/disable (one way) SSL. If enabled, HVR authenticates the Hive server by validating the SSL certificate shared by the Hive server.
Two-way SSL	Enable/disable two way SSL. If enabled, both HVR and Hive server authenticate each other by validating each others SSL certificate. This field is enabled only if **Enable SSL** is selected.
Trusted CA Certificates	The directory path where the **.pem** file containing the server's public SSL certificate signed by a trusted CA is located. This field is enabled only if **Enable SSL** is selected.
SSL Public Certificate	The directory path where the **.pem** file containing the client's SSL public certificate is located. This field is enabled only if **Two-way SSL** is selected.
SSL Private Key	The directory path where the **.pem** file containing the client's SSL private key is located. This field is enabled only if **Two-way SSL** is selected.
Client Private Key Password	The password of the private key file that is specified in **SSL Private Key**. This field is enabled only if **Two-way SSL** is selected.

Hadoop Client

The Hadoop client must be installed on the machine from which HVR will access the Azure Blob FS. Internally, HVR uses C API libhdfs to connect, read and write data to the Azure Blob FS during capture, integrate (continuous), refresh (bulk) and compare (direct file compare).

Azure Blob FS locations can only be accessed through HVR running on Linux or Windows, and it is not required to run HVR installed on the Hadoop NameNode although it is possible to do so. For more information about installing Hadoop client, refer to Apache Hadoop Releases.

Hadoop Client Configuration

The following are required on the machine from which HVR connects to Azure Blob FS:

- Hadoop 2.6.x client libraries with Java 7 Runtime Environment or Hadoop 3.x client libraries with Java 8 Runtime Environment. For downloading Hadoop, refer to Apache Hadoop Releases.
- Set the environment variable **$JAVA_HOME** to the Java installation directory. Ensure that this is the directory that has a bin folder, e.g. if the Java bin directory is d:\java\bin, **$JAVA_HOME** should point to d:\java.
- Set the environment variable **$HADOOP_COMMON_HOME** or **$HADOOP_HOME** or **$HADOOP_PREFIX** to the Hadoop installation directory, or the **hadoop** command line client should be available in the path.
- One of the following configuration is recommended,
  - Set **$HADOOP_CLASSPATH=$HADOOP_HOME/share/hadoop/tools/lib/***
  - Create a symbolic link for **$HADOOP_HOME/share/hadoop/tools/lib/** in **$HADOOP_HOME/share/hadoop/common** or any other directory present in classpath.

> Since the binary distribution available in Hadoop website lacks Windows-specific executables, a warning about unable to locate **winutils.exe** is displayed. This warning can be ignored for using Hadoop library for client operations to connect to a HDFS server using HVR. However, the performance on integrate location would be poor due to this warning, so it is recommended to use a Windows-specific Hadoop distribution to avoid this warning. For more information about this warning, refer to Hadoop issue HADOOP-10051.

Verifying Hadoop Client Installation

To verify the Hadoop client installation,

1. The **HADOOP_HOME/bin** directory in Hadoop installation location should contain the hadoop executables in it.
2. Execute the following commands to verify Hadoop client installation:

```
$JAVA_HOME/bin/java -version
$HADOOP_HOME/bin/hadoop version
$HADOOP_HOME/bin/hadoop classpath
```

3. If the Hadoop client installation is verified successfully then execute the following command to check the connectivity between HVR and Azure Blob FS:

> To execute this command successfully and avoid the error "ls: Password fs.adl.oauth2.client.id not found", few properties needs to be defined in the file **core-site.xml** available in the hadoop configuration folder (for e.g., **<path>/hadoop-2.8.3/etc/hadoop**). The properties to be defined differs based on the **Mechanism** (authentication mode). For more information, refer to section 'Configuring Credentials' in Hadoop Azure Blob FS Support documentation.

```
$HADOOP_HOME/bin/hadoop fs -ls adl://<cluster>/
```

Verifying Hadoop Client Compatibility with Azure Blob FS

To verify the compatibility of Hadoop client with Azure Blob FS, check if the following JAR files are available in the Hadoop client installation location (**$HADOOP_HOME/share/hadoop/tools/lib**):

```
hadoop-azure-<version>.jar
azure-storage-<version>.jar
```

Authentication

HVR does not support client side encryption (customer managed keys) for Azure Blob FS. For more information about encryption of data in Azure Blob FS, search for "encryption" in Azure Blob storage documentation.

Client Configuration Files

Client configuration files are not required for HVR to perform replication, however, they can be useful for debugging. Client configuration files contain settings for different services like HDFS, and others. If the HVR integrate machine is

not part of the cluster, it is recommended to download the configuration files for the cluster so that the Hadoop client knows how to connect to HDFS.

The client configuration files for Cloudera Manager or Ambari for Hortonworks can be downloaded from the respective cluster manager's web interface. For more information about downloading client configuration files, search for "Client Configuration Files" in the respective documentation for Cloudera and Hortonworks.

Hive External Table

HVR allows you to create Hive external tables above Azure Blob FS files which are only used during compare. The Hive ODBC connection can be enabled for Azure Blob FS in the location creation screen by selecting the **Hive External Tables** field.

For more information about configuring Hive external tables for Azure Blob FS, refer to Hadoop Azure Support: Azure Blob Storage documentation.

ODBC Connection

HVR uses the ODBC connection to the Hadoop cluster that requires an ODBC driver (Amazon ODBC 1.1.1 or HortonWorks ODBC 2.1.2 and above) for Hive installed on the machine (or in the same network). The Amazon and HortonWorks ODBC drivers are similar and compatible with Hive 2.x. However, it is recommended to use the Amazon ODBC driver for Amazon Hive and the Hortonworks ODBC driver for HortonWorks Hive.

HVR uses the Amazon ODBC driver or HortonWorks ODBC driver to connect to Hive for creating Hive external tables to perform hvrcompare of files that reside on Azure Blob FS.

By default, HVR uses Amazon ODBC driver for connecting to Hadoop. To use the Hortonworks ODBC driver the following action definition is required:

For Linux,

Group	Table	Action
Azure Blob FS	*	Environment **/Name** = **HVR_ODBC_CONNECT_STRING_DRIVER /Value** = **Hortonworks Hive ODBC Driver 64-bit**

For Windows,

Group	Table	Action
Azure Blob FS	*	Environment **/Name** = **HVR_ODBC_CONNECT_STRING_DRIVER /Value** = **Hortonworks Hive ODBC Driver**

Channel Configuration

For the file formats (CSV, JSON, and AVRO) the following action definitions are required to handle certain limitations of the Hive deserialization implementation during Bulk or Row-wise **Compare**:

- For CSV,

Azure Blob FS	*	**FileFormat /NullRepresentation=\\N**
Azure Blob FS	*	**TableProperties /CharacterMapping="\x00>\\0;\n>\\n;\r>\\r;">\""**
Azure Blob FS	*	**TableProperties /MapBinary=BASE64**

- For JSON,

Azure Blob FS	*	**TableProperties /MapBinary=BASE64**
Azure Blob FS	*	**FileFormat /JsonMode=ROW_FRAGMENTS**

- For Avro,

Azure Blob FS	*	**FileFormat /AvroVersion=v1_8**

v1_8 is the default value for **FileFormat /AvroVersion**, so it is not mandatory to define this action.

Requirements for Azure Data Lake Store

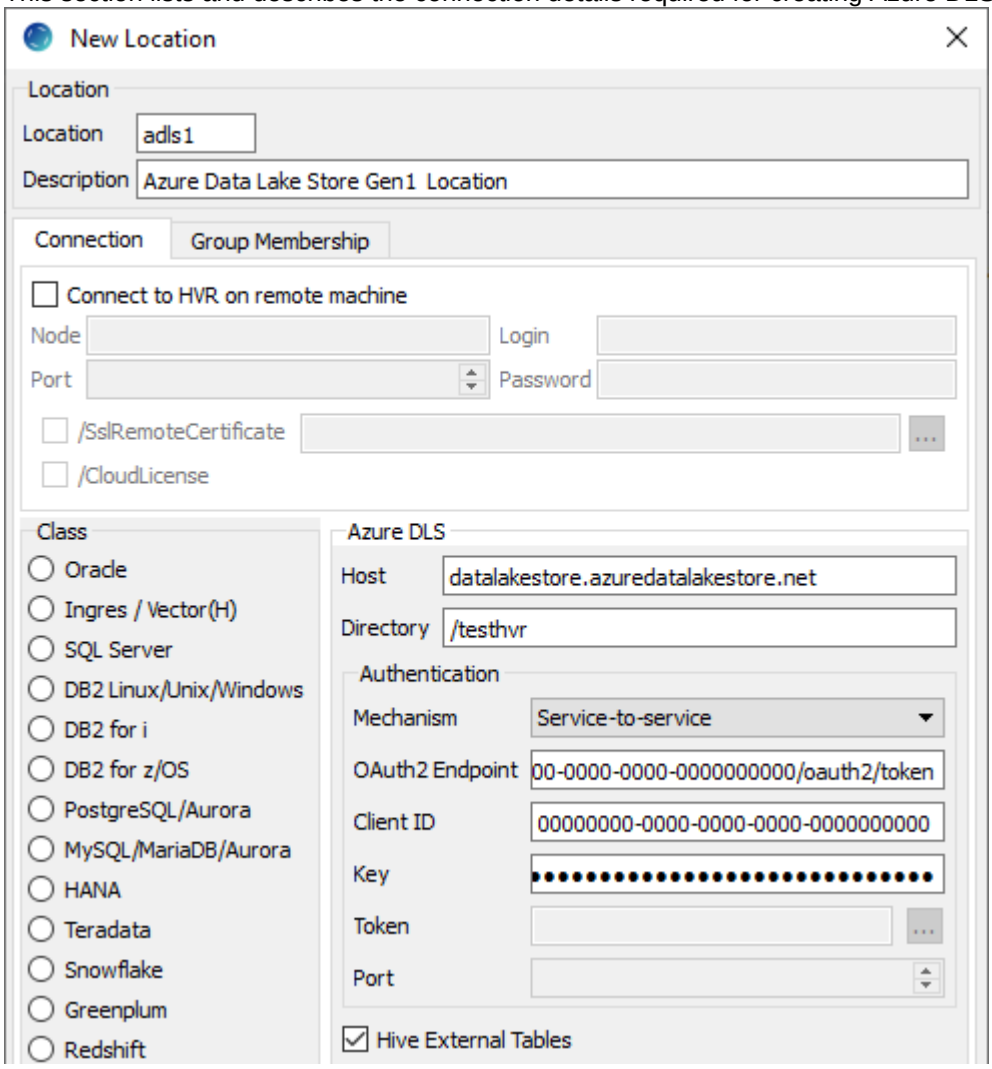
Contents

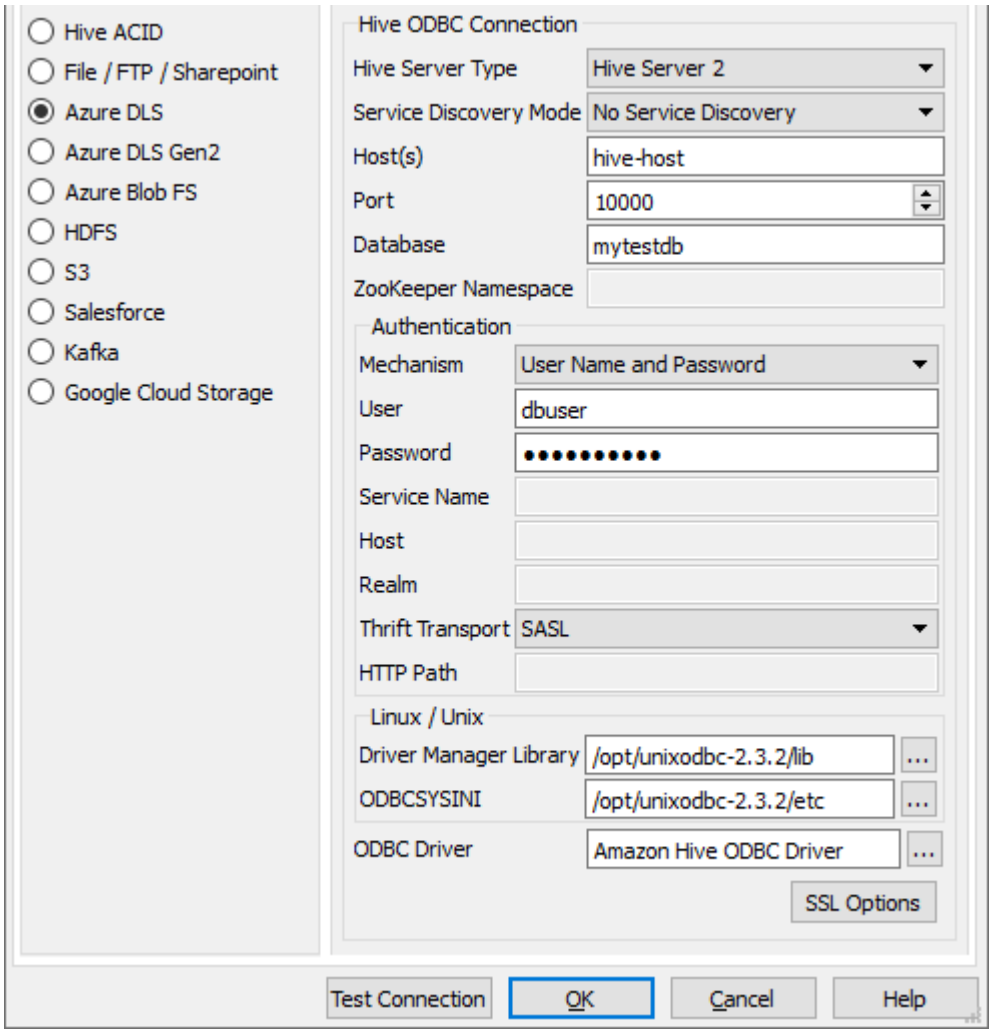
This section describes the requirements, access privileges, and other features of HVR when using Azure Data Lake Store (DLS) Gen1 for replication. For information about compatibility and support for Azure DLS Gen1 with HVR platforms, see Platform Compatibility Matrix.

For the capabilities supported by HVR, see Capabilities.

Location Connection

This section lists and describes the connection details required for creating Azure DLS location in HVR.





Field	Description
Azure DLS	
Host	The IP address or hostname of the Azure DLS server. **Example:** datalakestore.azuredatalakestore.net
Directory	The directory path in **Host** where the replicated changes are saved. **Example:** /testhvr
Authentication	
Mechanism	The authentication mode for connecting HVR to Azure DLS server. Available options: • **Service-to-service** • **Refresh Token** • **MSI** For more information about these authentication modes, see section Authentication.
OAuth2 Endpoint	The URL used for obtaining bearer token with credential token. This field is enabled only if the authentication **Mechanism** is **Service-to-service**. **Example:** https://login.microsoftonline.com/00000000-0000-0000-0000-0000000000/oauth2/token
Client ID	The Client ID (or Application ID) used to obtain access token with either credential or refresh token. This field is enabled only if the authentication **Mechanism** is either **Service-to-service** or **Refresh Token**. **Example:** 00000000-0000-0000-0000-0000000000

Key	The credential used for obtaining the initial and subsequent access tokens. This field is enabled only if the authentication **Mechanism** is **Service-to-service**.
Token	The directory path to the text file containing the refresh token. This field is enabled only if the authentication **Mechanism** is **Refresh Token**.
Port	The port number for the REST endpoint of the token service exposed to localhost by the identity extension in the Azure VM (default value: **50342**). This field is enabled only if the authentication **Mechanism** is **MSI**.
Hive External Tables	Enable/Disable Hive ODBC connection configuration for creating Hive external tables above Azure DLS.

Hive ODBC Connection

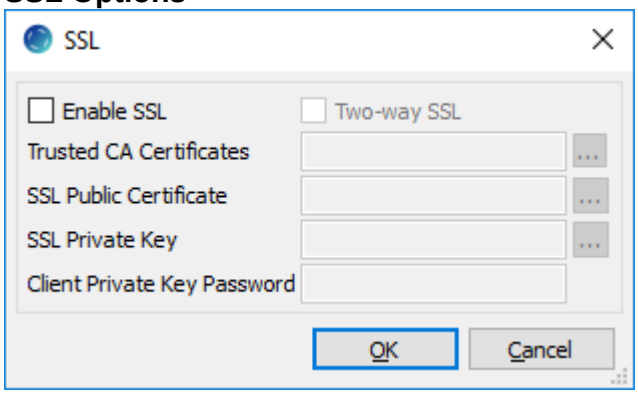
HVR allows you to create Hive External Tables above Azure DLS files which are only used during compare. You can enable/disable the Hive configuration for Azure DLS in location creation screen using the field **Hive External Tables**. For more information about configuring Hive external tables, refer to Hadoop Azure Data Lake Support documentation.

Field	Description
Hive ODBC Connection	
Hive Server Type	The type of Hive server. Available options: • **Hive Server 1** (default): The driver connects to a Hive Server 1 instance. • **Hive Server 2**: The driver connects to a Hive Server 2 instance.
Service Discovery Mode	The mode for connecting to Hive. This field is enabled only if **Hive Server Type** is **Hive Server 2**. Available options: • **No Service Discovery** (default): The driver connects to Hive server without using the ZooKeeper service. • **ZooKeeper**: The driver discovers Hive Server 2 services using the ZooKeeper service.
Host(s)	The hostname or IP address of the Hive server. When Service Discovery Mode is ZooKeeper, specify the list of ZooKeeper servers in following format **[ZK_Host1]:[ZK_Port1],[ZK_Host2]:[ZK_Port2]**, where **[ZK_Host]** is the IP address or hostname of the ZooKeeper server and **[ZK_Port]** is the TCP port that the ZooKeeper server uses to listen for client connections. **Example:** hive-host
Port	The TCP port that the Hive server uses to listen for client connections. This field is enabled only if **Service Discovery Mode** is **No Service Discovery**. **Example:** 10000
Database	The name of the database schema to use when a schema is not explicitly specified in a query. **Example:** mytestdb
ZooKeeper Namespace	The namespace on ZooKeeper under which Hive Server 2 nodes are added. This field is enabled only if **Service Discovery Mode** is **ZooKeeper**.
Authentication	

Mechanism	The authentication mode for connecting HVR to **Hive Server 2**. This field is enabled only if **Hive Server Type** is **Hive Server 2**. Available options: • **No Authentication** (default) • **User Name** • **User Name and Password** • **Kerberos** • **Windows Azure HDInsight Service** **Since** v5.5.0/2
User	The username to connect HVR to Hive server. This field is enabled only if **Mechanism** is **User Name** or **User Name and Password**. **Example:** dbuser
Password	The password of the **User** to connect HVR to Hive server. This field is enabled only if **Mechanism** is **User Name and Password**.
Service Name	The Kerberos service principal name of the Hive server. This field is enabled only if **Mechanism** is **Kerberos**.
Host	The Fully Qualified Domain Name (FQDN) of the Hive Server 2 host. The value of **Host** can be set as **_HOST** to use the Hive server hostname as the domain name for Kerberos authentication. If **Service Discovery Mode** is disabled, then the driver uses the value specified in the Host connection attribute. If **Service Discovery Mode** is enabled, then the driver uses the **Hive Server 2** host name returned by ZooKeeper. This field is enabled only if **Mechanism** is **Kerberos**.
Realm	The realm of the Hive Server 2 host. It is not required to specify any value in this field if the realm of the Hive Server 2 host is defined as the default realm in Kerberos configuration. This field is enabled only if **Mechanism** is **Kerberos**.
Thrift Transport **Since** v5.5.0/2	The transport protocol to use in the Thrift layer. This field is enabled only if **Hive Server Type** is **Hive Server 2**. Available options: • **Binary** (This option is available only if **Mechanism** is **No Authentication** or **User Name and Password**.) • **SASL** (This option is available only if **Mechanism** is **User Name** or **User Name and Password** or **Kerberos**.) • **HTTP** (This option is not available if **Mechanism** is **User Name**.) For information about determining which Thrift transport protocols your Hive server supports, refer to HiveServer2 Overview and Setting Up HiveServer2 sections in Hive documentation.
HTTP Path **Since** v5.5.0/2	The partial URL corresponding to the Hive server. This field is enabled only if **Thrift Transport** is **HTTP**.
Linux / Unix	
Driver Manager Library	The directory path where the Unix ODBC Driver Manager Library is installed. **Example:** /opt/unixodbc-2.3.2/lib
ODBCSYSINI	The directory path where **odbc.ini** and **odbcinst.ini** files are located. **Example:** /opt/unixodbc-2.3.2/etc

ODBC Driver	The user defined (installed) ODBC driver to connect HVR to the Hive server.
SSL Options	Show **SSL Options**.

SSL Options



Field	Description
Enable SSL	Enable/disable (one way) SSL. If enabled, HVR authenticates the Hive server by validating the SSL certificate shared by the Hive server.
Two-way SSL	Enable/disable two way SSL. If enabled, both HVR and Hive server authenticate each other by validating each others SSL certificate. This field is enabled only if **Enable SSL** is selected.
Trusted CA Certificates	The directory path where the **.pem** file containing the server's public SSL certificate signed by a trusted CA is located. This field is enabled only if **Enable SSL** is selected.
SSL Public Certificate	The directory path where the **.pem** file containing the client's SSL public certificate is located. This field is enabled only if **Two-way SSL** is selected.
SSL Private Key	The directory path where the **.pem** file containing the client's SSL private key is located. This field is enabled only if **Two-way SSL** is selected.
Client Private Key Password	The password of the private key file that is specified in **SSL Private Key**. This field is enabled only if **Two-way SSL** is selected.

Hadoop Client

The Hadoop client must be installed on the machine from which HVR accesses the Azure DLS. HVR uses C API libhdfs to connect, read and write data to the Azure Data Lake Store during capture, integrate (continuous), refresh (bulk) and compare (direct file compare).

Azure DLS locations can only be accessed through HVR running on Linux or Windows, and it is not required to run HVR installed on the Hadoop NameNode although it is possible to do so. For more information about installing Hadoop client, refer to Apache Hadoop Releases.

Hadoop Client Configuration

The following are required on the machine from which HVR connects to Azure DLS:

- Hadoop 2.6.x client libraries with Java 7 Runtime Environment or Hadoop 3.x client libraries with Java 8 Runtime Environment. For downloading Hadoop, refer to Apache Hadoop Releases.
- Set the environment variable **$JAVA_HOME** to the Java installation directory. Ensure that this is the directory that has a bin folder, e.g. if the Java bin directory is d:\java\bin, **$JAVA_HOME** should point to d:\java.

- Set the environment variable **$HADOOP_COMMON_HOME** or **$HADOOP_HOME** or **$HADOOP_PREFIX** to the Hadoop installation directory, or the **hadoop** command line client should be available in the path.
- One of the following configuration is recommended:
  - Set **$HADOOP_CLASSPATH=$HADOOP_HOME/share/hadoop/tools/lib/***
  - Create a symbolic link for **$HADOOP_HOME/share/hadoop/tools/lib** in **$HADOOP_HOME/share/hadoop/common** or any other directory present in classpath.

> Since the binary distribution available in Hadoop website lacks Windows-specific executables, a warning about unable to locate **winutils.exe** is displayed. This warning can be ignored for using Hadoop library for client operations to connect to a HDFS server using HVR. However, the performance on integrate location would be poor due to this warning, so it is recommended to use a Windows-specific Hadoop distribution to avoid this warning. For more information about this warning, refer to Hadoop issue [HADOOP-10051].

Verifying Hadoop Client Installation

To verify the Hadoop client installation:

1. The **HADOOP_HOME/bin** directory in Hadoop installation location should contain the Hadoop executables in it.
2. Execute the following commands to verify Hadoop client installation:

```
$JAVA_HOME/bin/java -version
$HADOOP_HOME/bin/hadoop version
$HADOOP_HOME/bin/hadoop classpath
```

3. If the Hadoop client installation is verified successfully then execute the following command to verify the connectivity between HVR and Azure DLS:

> To execute this command successfully and avoid the error "ls: Password fs.adl.oauth2.client.id not found", few properties needs to be defined in the file **core-site.xml** available in the hadoop configuration folder (for e.g., **<path>/hadoop-2.8.3/etc/hadoop**). The properties to be defined differs based on the **Mechanism** (authentication mode). For more information, refer to section Configuring Credentials and FileSystem in Hadoop Azure Data Lake Support documentation.

```
$HADOOP_HOME/bin/hadoop fs -ls adl://<cluster>/
```

Verifying Hadoop Client Compatibility with Azure DLS

To verify the compatibility of Hadoop client with Azure DLS, check if the following JAR files are available in the Hadoop client installation location (**$HADOOP_HOME/share/hadoop/tools/lib**):

```
hadoop-azure-<version>.jar
hadoop-azure-datalake-<version>.jar
azure-data-lake-store-sdk-<version>.jar
azure-storage-<version>.jar
```

Authentication

HVR supports the following three authentication modes for connecting to Azure DLS:

- **Service-to-service**
  This option is used if an application needs to directly authenticate itself with Data Lake Store. The connection parameters required in this authentication mode are OAuth2 Token Endpoint, Client ID (application ID), and Key (authentication key). For more information about the connection parameters, search for "Service-to-service authentication" in Data Lake Store Documentation.
- **Refresh Token**
  This option is used if a user's Azure credentials are used to authenticate with Data Lake Store. The connection parameters required in this authentication mode are Client ID (application ID), and Token (refresh token).
  The refresh token should be saved in a text file and the directory path to this text file should be mentioned in the **Token** field of location creation screen. For more information about the connection parameters and end-user authentication using REST API, search for "End-user authentication" in Data Lake Store Documentation.
- **MSI**
  This option is preferred when you have HVR running on a VM in Azure. Managed Service Identity (MSI) allows you to authenticate to services that support Azure Active Directory authentication. For this authentication mode to work, the VM should have access to Azure DLS and the MSI authentication should be enabled on the VM in Azure. The connection parameters required in this authentication mode is Port (MSI endpoint port), by default the port number is 50342. For more information about providing access to Azure DLS and enabling MSI on the VM, search for "Access Azure Data Lake Store" in Azure Active Directory Managed Service Identity Documentation.

HVR does not support client side encryption (customer managed keys) for Azure DLS. For more information about encryption of data in Azure DLS, search for "encryption" in Data Lake Store Documentation.

Client Configuration Files

Client configuration files are not required for HVR to perform replication, however, they can be useful for debugging. Client configuration files contain settings for different services like HDFS, and others. If the HVR integrate machine is not part of the cluster, it is recommended to download the configuration files for the cluster so that the Hadoop client knows how to connect to HDFS.

The client configuration files for Cloudera Manager or Ambari for Hortonworks can be downloaded from the respective cluster manager's web interface. For more information about downloading client configuration files, search for "Client Configuration Files" in the respective documentation for Cloudera and Hortonworks.

Hive External Table

HVR allows you to create Hive external tables above Azure DLS files which are only used during compare. The Hive ODBC connection can be enabled for Azure DLS in the location creation screen by selecting the **Hive External Tables** field. For more information about configuring Hive external tables for Azure DLS, refer to Hadoop Azure Data Lake Support documentation.

ODBC Connection

HVR uses ODBC connection to the Hadoop cluster that requires an ODBC driver (Amazon ODBC 1.1.1 or HortonWorks ODBC 2.1.2 and above) for Hive installed on the machine (or in the same network). The Amazon and HortonWorks ODBC drivers are similar and compatible with Hive 2.x. However, it is recommended to use the Amazon ODBC driver for Amazon Hive and the Hortonworks ODBC driver for HortonWorks Hive.

HVR uses the Amazon ODBC driver or HortonWorks ODBC driver to connect to Hive for creating Hive external tables to perform hvrcompare of files that reside on Azure DLS.

By default, HVR uses Amazon ODBC driver for connecting to Hadoop. To use the Hortonworks ODBC driver the following action definition is required:

For Linux:

Group	Table	Action
Azure DLS	*	**Environment /Name=HVR_ODBC_CONNECT_STRING_DRIVER /Value=Hortonworks Hive ODBC Driver 64-bit**

For Windows:

Group	Table	Action
Azure DLS	*	**Environment /Name=HVR_ODBC_CONNECT_STRING_DRIVER /Value=Hortonworks Hive ODBC Driver**

Channel Configuration

For the file formats (CSV, JSON, and AVRO) the following action definitions are required to handle certain limitations of the Hive deserialization implementation during Bulk or Row-wise **Compare**:

- For CSV

Group	Table	Action
Azure DLS	*	**FileFormat /NullRepresentation=\\N**
Azure DLS	*	**TableProperties /CharacterMapping="\x00>\\0;\n>\\n;\r>\\r;">\""**
Azure DLS	*	**TableProperties /MapBinary=BASE64**

- For JSON

Group	Table	Action
Azure DLS	*	**TableProperties /MapBinary=BASE64**
Azure DLS	*	**FileFormat /JsonMode=ROW_FRAGMENTS**

- For Avro

Group	Table	Action
Azure DLS	*	**FileFormat /AvroVersion=v1_8**

v1_8 is the default value for **FileFormat /AvroVersion**, so it is not mandatory to define this action.

Requirements for Azure Data Lake Storage Gen2

Since v5.6.5/2

Contents

This section describes the requirements, access privileges, and other features of HVR when using Azure Data Lake Storage (DLS) Gen2 for replication. For information about compatibility and support for Azure DLS Gen2 with HVR platforms, see Platform Compatibility Matrix.

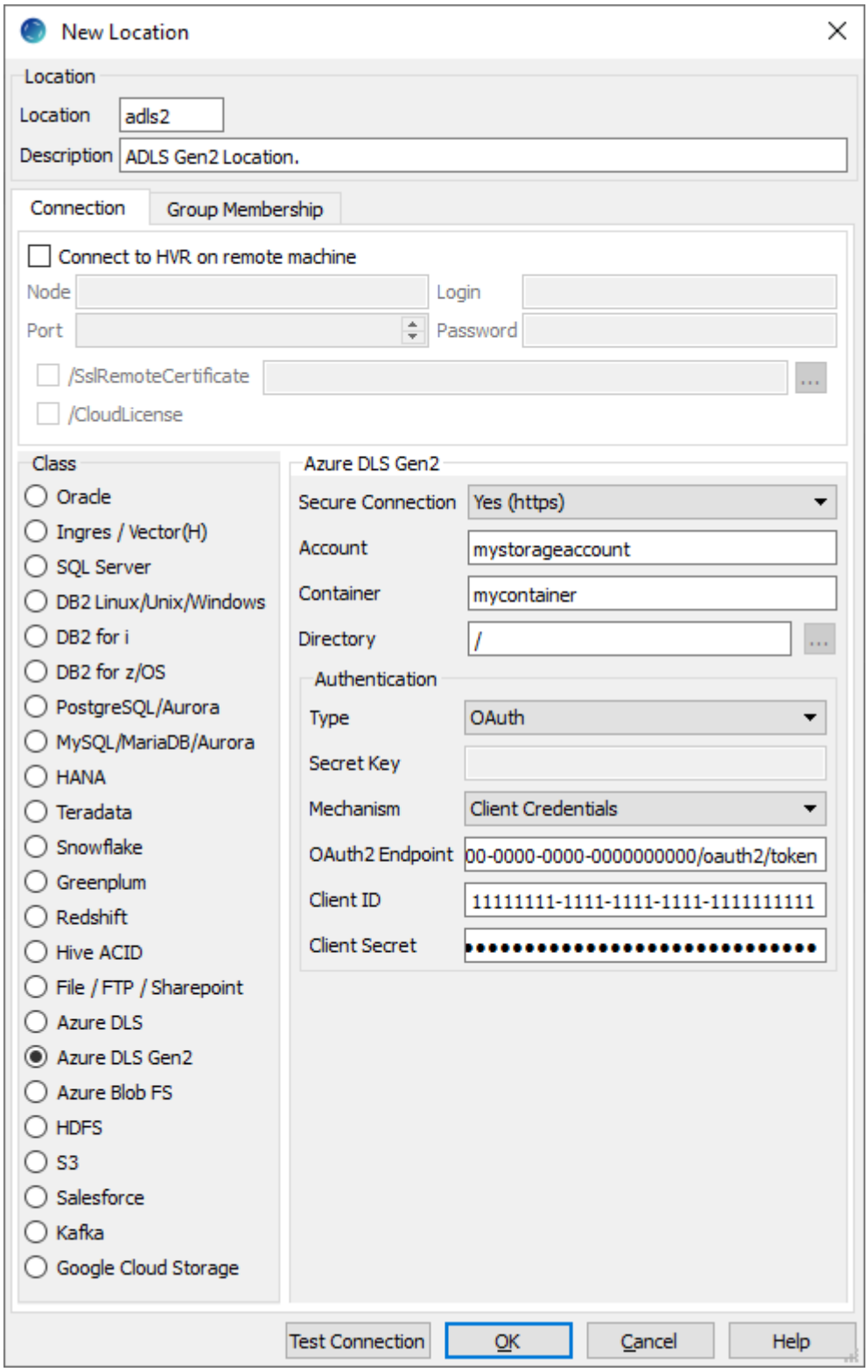
For the capabilities supported by HVR, see Capabilities.

For information about the supported data types and mapping of data types in source DBMS to the corresponding data types in target DBMS or file format, see Data Type Mapping.

To quickly set up replication using Azure DLS Gen2, see Quick Start for HVR - Azure DLS Gen2.

Location Connection

This section lists and describes the connection details required for creating Azure DLS Gen2 location in HVR.



Field	Description
Azure DLS Gen2	
Secure connection	The type of security to be used for connecting to Azure DLS Gen2. Available options: • **Yes (https)** (**default**): HVR will connect to Azure DLS Gen2 using HTTPS. • **No (http)**: HVR will connect to Azure DLS Gen2 using HTTP.
Account	The Azure DLS Gen2 storage account. **Example:** mystorageaccount

Container	The name of the container available within storage **Account**. **Example:** mycontainer
Directory	The directory path in **Container** to be used for replication. **Example:** /folder
Authentication	
Type	The type of authentication to be used for connecting to Azure DLS Gen2. Available options: • **Shared Key** (**default**): HVR will access Azure DLS Gen2 using **Shared Key** authentication. • **OAuth**: HVR will access Azure DLS Gen2 using **OAuth** authentication. For more information about these authentication types, see section Authentication.
Secret Key	The access key of the storage **Account**. This field is enabled only if authentication **Type** is **Shared Key**.
Mechanism	The authentication mode for connecting HVR to Azure DLS Gen2 server. This field is enabled only if authentication **Type** is **OAuth**. The available option is **Client Credentials**.
OAuth2 Endpoint	The URL used for obtaining bearer token with credential token. **Example:** https://login.microsoftonline.com/00000000-0000-0000-0000-0000000000/oauth2/token
Client ID	A client ID (or application ID) used to obtain Azure AD access token. **Example:** 00000000-0000-0000-0000-0000000000
Client Secret	A secret key used to validate the **Client ID**.

Hadoop Client

The Hadoop client must be installed on the machine from which HVR will access Azure DLS Gen2. HVR uses C API libhdfs to connect, read and write data to the Azure DLS Gen2 during capture, integrate (continuous), refresh (bulk) and compare (direct file compare).

Azure DLS Gen2 locations can only be accessed through HVR running on Linux or Windows. It is not required to run HVR installed on the Hadoop NameNode, although it is possible to do so. For more information about installing Hadoop client, refer to Apache Hadoop Releases.

⚠ On Linux, an extra warning is raised: "WARNING: HADOOP_PREFIX has been replaced by HADOOP_HOME. Using value of HADOOP_PREFIX".

To fix this behavior, comment out the following line in the **$HADOOP_PREFIX/libexec/hadoop-condig.sh** file:

```
hadoop_deprecate_envvar HADOOP_PREFIX HADOOP_HOME
```

Hadoop Client Configuration

The following is required on the machine from which HVR connects to Azure DLS Gen2:

- Hadoop client libraries version 3.2.0 and higher. For downloading Hadoop, refer to Apache Hadoop Download page.

- Java Runtime Environment version 8 and higher. For downloading Java, refer to Java Download page.
- Set the environment variable **$JAVA_HOME** to the Java installation directory. Ensure that this is the directory that has a bin folder, e.g. if the Java bin directory is d:\java\bin, **$JAVA_HOME** should point to d:\java.

> ⚠ If the environment variable **$HVR_JAVA_HOME** is configured, the value of this environment variable should point to the same path defined in **$JAVA_HOME**.

- Set the environment variable **$HADOOP_COMMON_HOME** or **$HADOOP_HOME** or **$HADOOP_PREFIX** to point to the Hadoop installation directory, or the **hadoop** command line client should be available in the path.
- One of the following configurations is recommended:
  - Set **$HADOOP_CLASSPATH=$HADOOP_HOME/share/hadoop/tools/lib/***
  - Create a symbolic link for **$HADOOP_HOME/share/hadoop/tools/lib** in **$HADOOP_HOME/share/hadoop/common** or any other directory present in the classpath.
- On Windows, **winutils.exe** along with **hadoop.dll** is required. These files can be downloaded from the GitHub and should be saved to **$HADOOP_HOME/bin** directory. This is required since the binary distribution of Hadoop lacks this executable.

Verifying Hadoop Client Installation

To verify the Hadoop client installation:

1. The **$HADOOP_HOME/bin** directory in the Hadoop installation location should contain the Hadoop executables in it.
2. Execute the following commands to verify the Hadoop client installation:

```
$JAVA_HOME/bin/java -version
$HADOOP_HOME/bin/hadoop version
$HADOOP_HOME/bin/hadoop classpath
```

3. If the Hadoop client installation is successfully verified, execute the following command to verify the connectivity between HVR and Azure DLS Gen2:

```
$HADOOP_HOME/bin/hadoop fs -ls abfs://<container>@<account>.dfs.core.windows.net
```

⚠ In case of any identification errors, certain properties need to be defined in the **core-site.xml** file available in the Hadoop configuration folder (for e.g., **<path>/hadoop-3.2.0/etc/hadoop**). For more information, refer to section Configuring ABFS in the Hadoop Azure Support: ABFS - Azure Data Lake Storage Gen2 documentation.

Click here for sample configuration when using Shared Key authentication

```
<property>
   <name>fs.azure.account.auth.type.storageaccountname.dfs.core.windows.net<
/name>
  <value>SharedKey</value>
  <description>Use Shared Key authentication</description>
</property>

<property>
  <name>fs.azure.account.key.storageaccountname.dfs.core.windows.net</name>
                     <value>JDlkIHxvySByZWFsbHkgdGabcdfeSSB3LDJgZ34pbm
/skdG8gcGD0IGEga2V5IGluIGhlcmSA</value>
  <description>The secret password.</description>
</property>
```

Click here for sample configuration when using OAuth authentication

```
<property>
  <name>fs.azure.account.auth.type</name>
  <value>OAuth</value>
  <description>Use OAuth authentication</description>
</property>


<property>
  <name>fs.azure.account.oauth.provider.type</name>
      <value>org.apache.hadoop.fs.azurebfs.oauth2.ClientCredsTokenProvider<
/value>
  <description>Use client credentials</description>
</property>


<property>
  <name>fs.azure.account.oauth2.client.endpoint</name>
  <value></value>
  <description>URL of OAuth endpoint</description>
</property>


<property>
  <name>fs.azure.account.oauth2.client.id</name>
  <value></value>
  <description>Client ID</description>
</property>


<property>
  <name>fs.azure.account.oauth2.client.secret</name>
  <value></value>
  <description>Secret</description>
</property>
```

Verifying Hadoop Client Compatibility with Azure DLS Gen2

To verify the compatibility of the Hadoop client with Azure DLS Gen2, check if the following JAR files are available in the Hadoop client installation directory (**$HADOOP_HOME/share/hadoop/tools/lib**):

```
wildfly-openssl-<version>.jar
hadoop-azure-<version>.jar
```

Authentication

HVR supports the following two authentication modes for connecting to Azure DLS Gen2:

- **Shared Key**
  When this option is selected, *hvruser* gains full access to all operations on all resources, including setting owner and changing Access Control List (ACL). The connection parameter required in this authentication mode is Secret Key - a shared access key that Azure generates for the storage account. For more information on how to manage access keys for Shared Key authorization, refer to Manage storage account access keys. Note that with this authentication mode, no identity is associated with a user and permission-based authorization cannot be implemented.
- **OAuth**
  This option is used to connect to Azure DLS Gen2 storage account directly with OAuth 2.0 using the service principal. The connection parameters required for this authentication mode are **OAuth2 Endpoint**, **Client ID**, and **Client Secret**. For more information, refer to Azure Data Lake Storage Gen2 documentation.

Encryption

HVR does not support client side encryption (customer managed keys) for Azure DLS Gen2. For more information about the encryption of data in Azure DLS Gen2 refer to Data Lake Storage Documentation.

Client Configuration Files for Hadoop

Client configuration files are not required for HVR to perform replication, however, they can be useful for debugging. Client configuration files contain settings for different services like HDFS, and others. If the HVR integrate machine is not part of the cluster, it is recommended to download the configuration files for the cluster so that the Hadoop client knows how to connect to HDFS.

The client configuration files for Cloudera Manager or Ambari for Hortonworks can be downloaded from the respective cluster manager's web interface. For more information about downloading the client configuration files, search for "Client Configuration Files" in the respective documentation for Cloudera and Hortonworks.

Requirements for Azure SQL Database

Contents

This section describes the requirements, access privileges, and other features of HVR when using Azure SQL Database for replication. Azure SQL Database is the Platform as a Service (PaaS) database of Microsoft's Azure Cloud Platform. It is a limited version of the Microsoft SQL Server. HVR supports Azure SQL Database through its regular SQL Server driver. For information about compatibility and supported versions of Azure SQL Database with HVR platforms, see Platform Compatibility Matrix.

For the capabilities supported by HVR on Azure SQL Database, see Capabilities for Azure SQL Database.

For information about the supported data types and mapping of data types in source DBMS to the corresponding data types in target DBMS or file format, see Data Type Mapping.

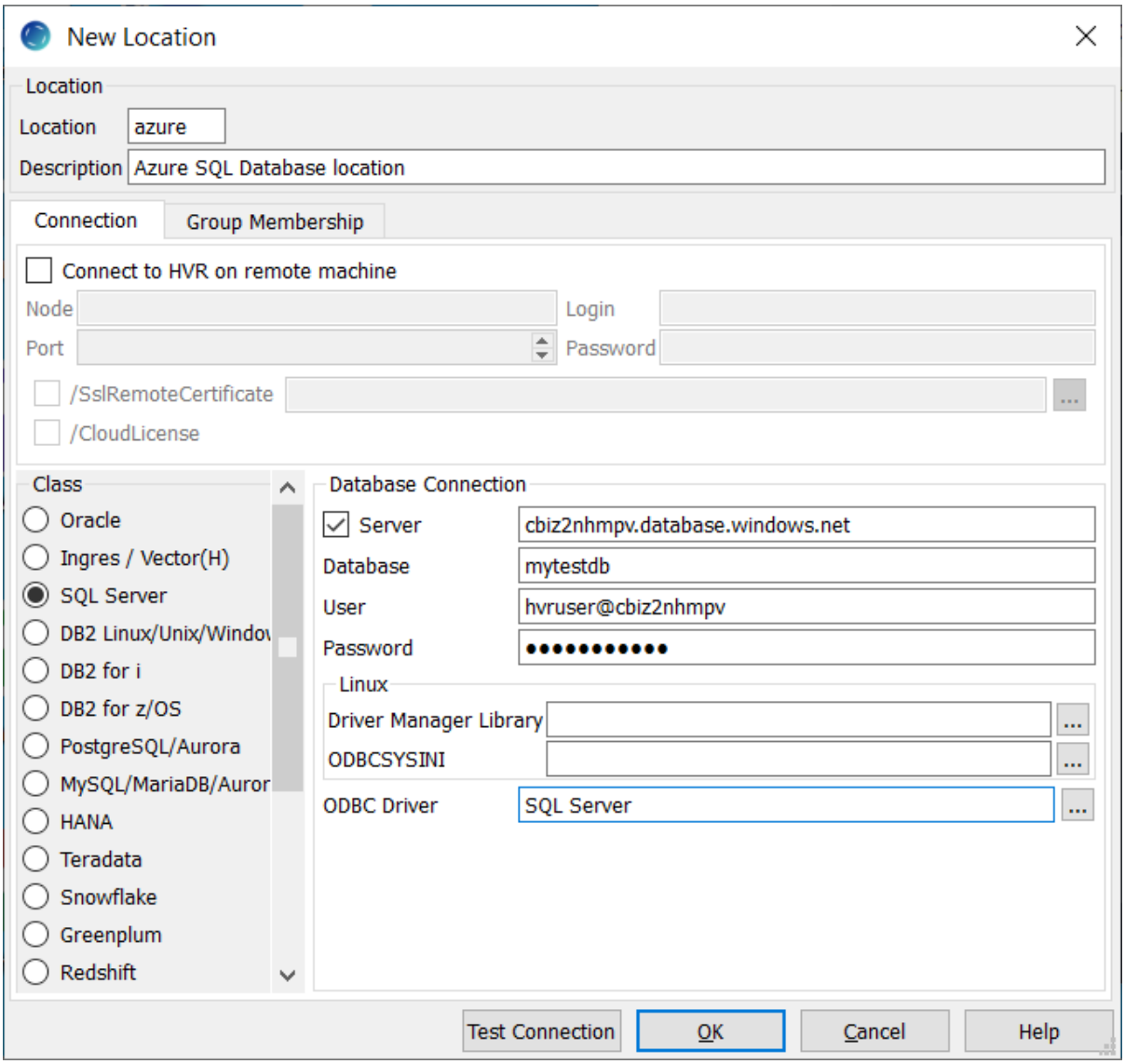
ODBC Connection

Microsoft SQL Server Native Client 11.0 ODBC driver must be installed on the machine from which HVR connects to Azure SQL Database. For more information about downloading and installing SQL Server Native Client, refer to Microsoft documentation.

HVR uses the SQL Server Native Client ODBC driver to connect, read and write data to Azure SQL Database during capture, integrate (continuous), and refresh (row-wise).

Location Connection

This section lists and describes the connection details required for creating Azure SQL Database location in HVR.

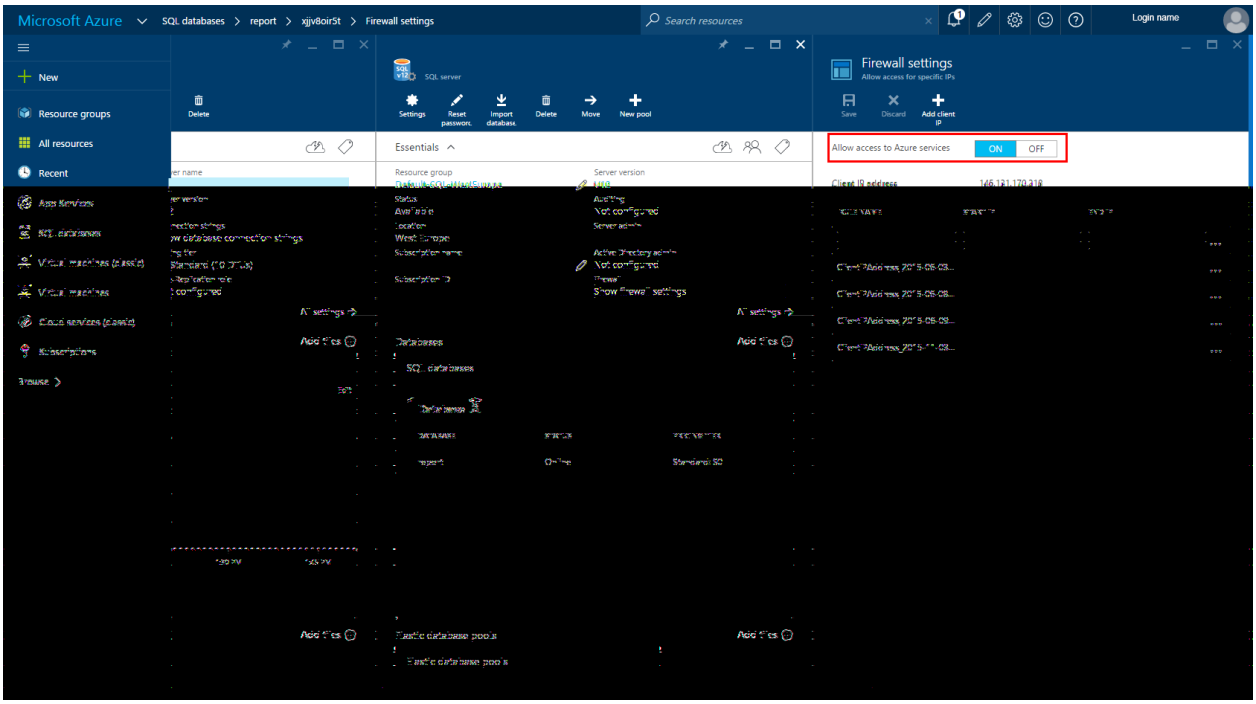


Field	Description
Database Connection	
Server	The fully qualified domain name (FQDN) name of the Azure SQL Database server. **Example:** cbiz2nhmpv.database.windows.net
Database	The name of the Azure SQL database. **Example:** mytestdb
User	The username to connect HVR to the Azure SQL **Database**. The username should be appended with the separator '@' and the host name of the **Server**. The format is **<user name>@<hostname>**. **Example:** hvruser@cbiz2nhmpv
Password	The password of the **User** to connect HVR to the Azure SQL **Database**.
ODBC Driver	The user defined (installed) ODBC driver to connect HVR to the Azure SQL Database.

Configuration Notes

The Azure SQL database server has a default firewall preventing incoming connections. This can be configured under **DAtabase server/Show firewall settings**. When connecting from an Azure VM (through an agent), enable **Allow access to Azure services**. When connecting directly from an on-premises hub, add its IP address to the

allowed range. An easy way to do this is to open the webportal from the machine, from which you connect to the database. Your IP address will be listed and by clicking **Add to the allowed IP addresses**, the IP address will be automatically added to the firewall.



Capture

- Log-based **Capture** is not supported from Azure SQL. Only trigger-based capture is supported.
- **Capture** parameter **/ToggleFrequency** must be specified because the Azure SQL database does not allow HVR's **hvrevent.dll** (no DLL libraries allowed). Keep in mind that if a high frequency is defined (e.g. cycle every 10 seconds) then many lines will be written to HVR's log files. Configure the command **Hvrmaint** to purge these files.

Integrate and Refresh

HVR uses SQL Server BCP interface for copying data into Azure SQL Database tables during bulk **Refresh** and loading data into burst tables during **Integrate** with **/Burst**.

When using **HVR Refresh** with option **Create absent tables** in Azure SQL database, enable the option "**With Key**" because Azure does not support tables without Clustered Indexes.

Requirements for Azure Synapse Analytics

Contents

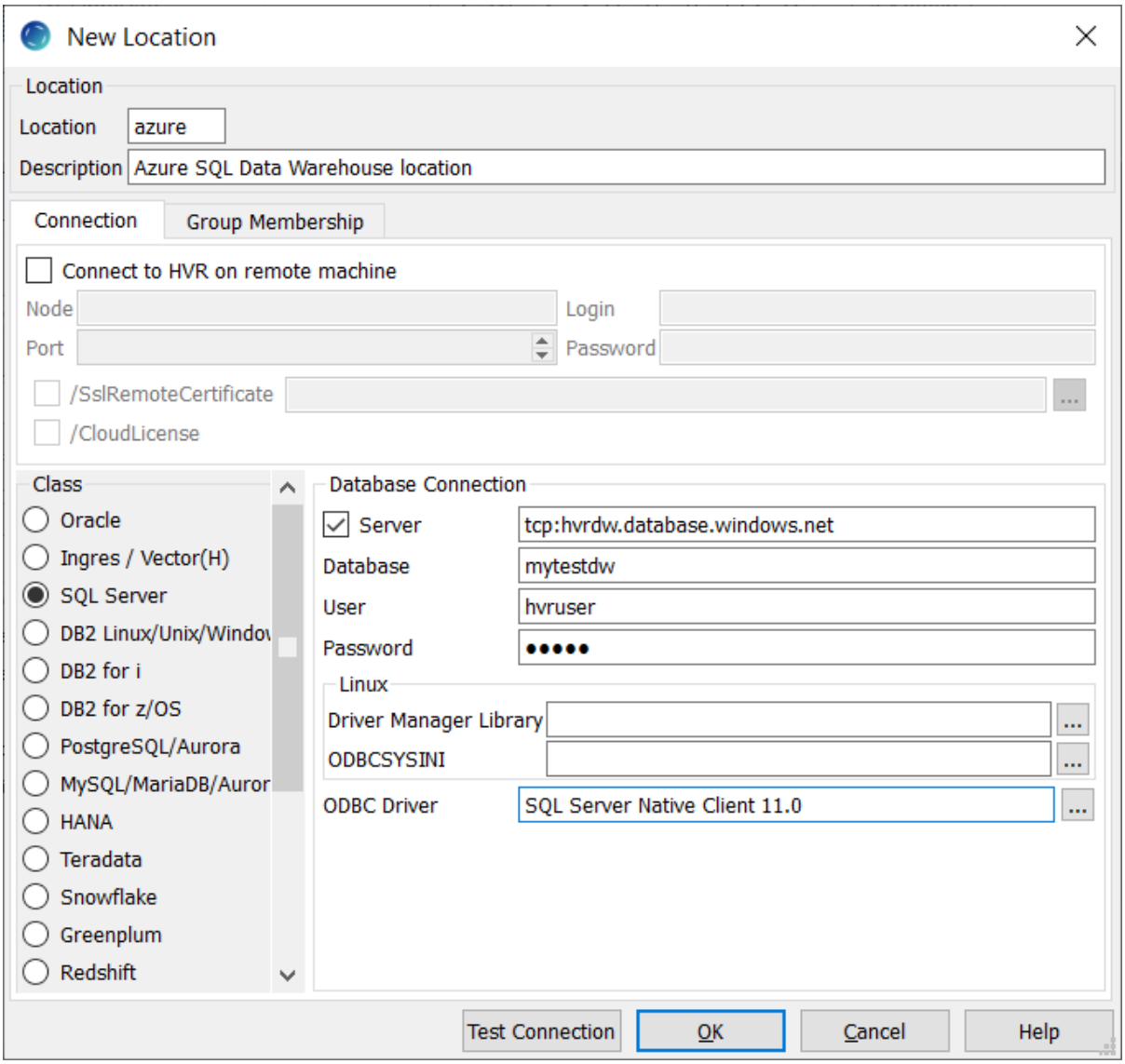
This section describes the requirements, access privileges, and other features of HVR when using Azure Synapse Analytics (formerly Azure SQL Data Warehouse) for replication. Azure Synapse Analytics is the Platform as a Service (PaaS) data warehouse and big data analytics of Microsoft's Azure Cloud Platform. HVR supports Azure Synapse Analytics through its regular SQL Server driver. For information about compatibility and supported versions of Azure Synapse Analytics with HVR platforms, see Platform Compatibility Matrix.

For information about the Capabilities supported by HVR on Azure Synapse Analytics, see Capabilities for Azure Synapse Analytics.

For information about the supported data types and mapping of data types in source DBMS to the corresponding data types in target DBMS or file format, see Data Type Mapping.

ODBC Connection

Microsoft SQL Server Native Client 11.0 ODBC driver must be installed on the machine from which HVR connects to Azure Synapse. For more information about downloading and installing SQL Server Native Client, refer to



Field	Description
Database Connection	
Server	The fully qualified domain name (FQDN) name of the Azure Synapse server. **Example:** tcp:hvrdw.database.windows.net
Database	The name of the Azure Synapse database. **Example:** mytestdw
User	The username to connect HVR to the Azure Synapse **Database**. **Example:** hvruser
Password	The password of the **User** to connect HVR to the Azure Synapse **Database**.
ODBC Driver	The user defined (installed) ODBC driver to connect HVR to the Azure Synapse.

Integrate and Refresh

HVR uses the following interfaces to write data into an SQL Server location:

- SQL Server ODBC driver, used to perform continuous **Integrate** and row-wise **Refresh**
- SQL Server BCP interface, used for copying data into database tables during bulk **Refresh** and loading data into burst tables during **Integrate** with **/Burst**.

Grants for Compare, Refresh and Integrate

The HVR **User** requires the following privileges:

```
grant create table to hvr_user
grant select, insert, update, delete on replicated tables to hvr_user
```

If the HVR **User** needs to bulk refresh or alter tables which are in another schema (using action **TableProperties /Schema=***myschema*) then the following grants are needed:

```
grant control on schema :: myschema to hvr_user
```

When HVR Refresh is used to create the target tables, the following is also needed:

```
grant create table on schema :: myschema to hvr_user
```

HVR's internal tables, like burst and state-tables, will be created in the user's default_schema. The default_schema can be changed using:

```
alter user hvr_user with default_schema = myschema
```

Requirements for DB2 for i

Contents

This section describes the requirements, access privileges, and other features of HVR when using 'DB2 for i' for replication. For information about compatibility and supported versions of DB2 for i with HVR platforms, see Platform Compatibility Matrix.

For the Capabilities supported by HVR on DB2 for i, see Capabilities for DB2 for i.

For information about the supported data types and mapping of data types in source DBMS to the corresponding data types in target DBMS or file format, see Data Type Mapping.

ODBC Connection

HVR is not installed on the DB2 for i system itself but is instead installed on a Linux or Windows machine, from which it uses ODBC to connect to the DB2 for i system. HVR uses ODBC connection to read and write data to DB2 for i location.

The following are required for HVR to establish an ODBC connection to the DB2 for i system:Data Type Ma2 13 c Targ

-
-

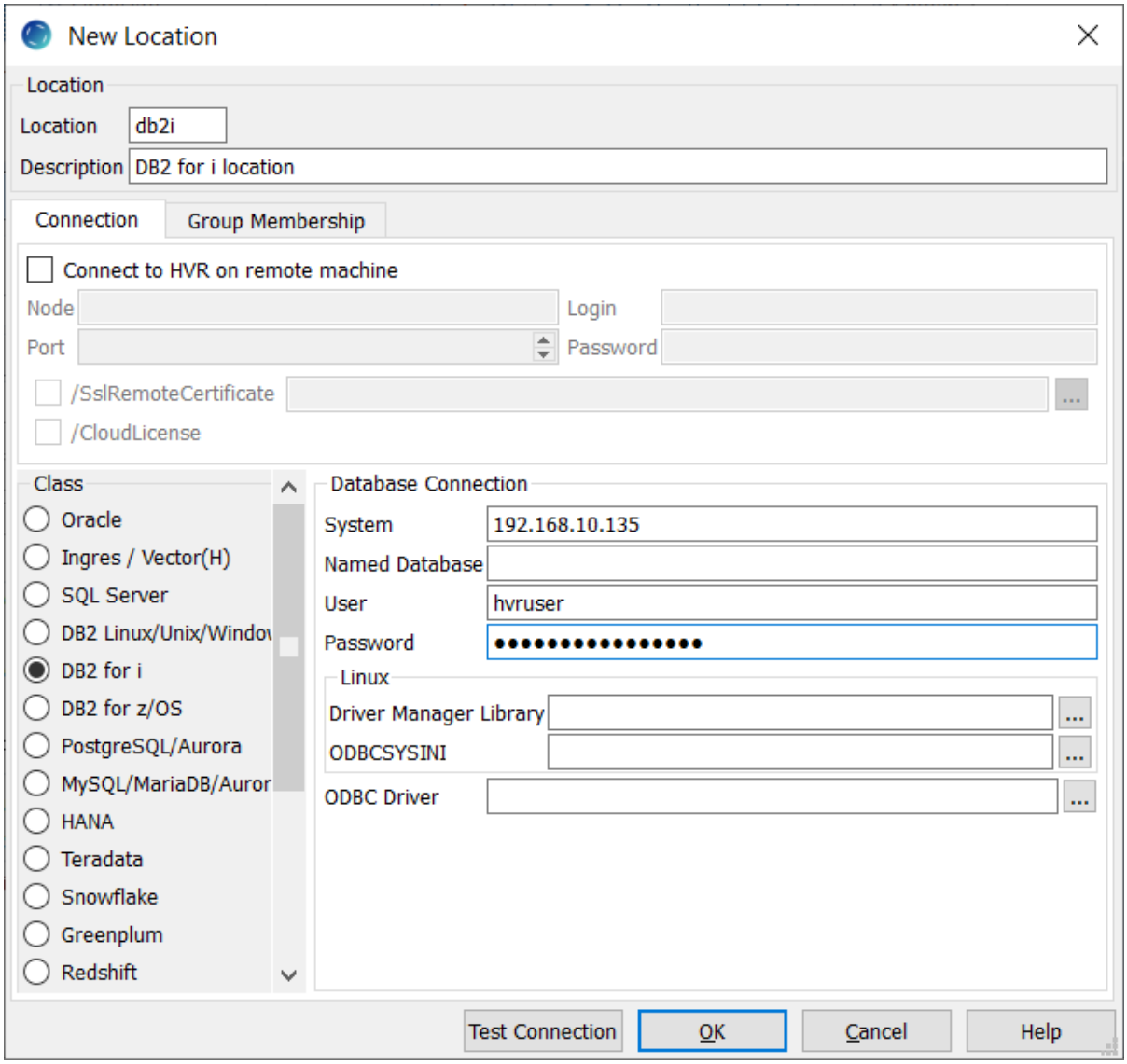
-

Sign-On Verification	as-signon	8476	9476
Database Access	as-database	8471	9471

⚠ The port numbers mentioned here are the default port numbers. To verify the default port numbers for the services names, use the command **wrksrvtble** on AS/400 console.

Location Connection

This section lists and describes the connection details required for creating DB2 for i location in HVR.



Field	Description
Database Connection	
System	The hostname or IP-address of the DB2 for i system. **Example:** 192.168.1.135

Named Database	The named database in DB2 for i. It could be on another (independent) auxiliary storage pool (IASP). The user profile's default setting will be used when no value is specified. Specifying ***SYSBAS** will connect a user to the SYSBAS database.
User	The username to connect HVR to the **Named Database** in DB2 for i. **Example:** hvruser
Password	The password of the **User** to connect HVR to the **Named Database** in DB2 for i.
Linux	
Driver Manager Library	The optional directory path where the ODBC Driver Manager Library is installed. For a default installation, the ODBC Driver Manager Library is available at **/usr/lib64** and does not need to be specified. When UnixODBC is installed in for example **/opt /unixodbc-2.3.1** this would be **/opt/unixodbc-2.3.1/lib**. **Example:** /opt/unixdbx-2.3.1/lib
ODBCSYSINI	The directory path where **odbc.ini** and **odbcinst.ini** files are located. For a default installation, these files are available at **/etc** and do not need to be specified. When UnixODBC is installed in for example **/opt/unixodbc-2.3.1** this would be **/opt /unixodbc-2.3.1/etc**. The **odbcinst.ini** file should contain information about the IBM i Access Client Solutions ODBC Driver under the heading **[IBM i Access ODBC Driver 64-bit]**. **Example:** /opt/unixdbx-2.3.1/etc
ODBC Driver	The user-defined (installed) ODBC driver to connect HVR to the DB2 for i system.

Hub

HVR allows you to create a hub database in DB2 for i. The hub database is a small database which HVR uses to control its replication activities. This database stores HVR catalog tables that hold all specifications of replication such as the names of the replicated databases, the list of replicated tables, and the replication direction.

Grants for Hub

To capture changes from a source database or to integrate changes into a target database, the following privileges are required:

- The **User** should have permission to create and drop HVR catalog tables

Grants

The **User** should have permissions to read the following system catalogs:

- **qsys2.systables**
- **qsys2.syscolumns**
- **qsys2.systypes**
- **qsys2.syscst**
- **qsys2.syscstcol**
- **qsys2.sysindexes**
- **qsys2.syskeys**
- **sysibm.sysdummy1**
- **sysibm.sqlstatistics**

According to IBM documentation, the tables and views in the catalogs are shipped with the **SELECT** privilege to **PUBLIC**. This privilege may be revoked and the **SELECT** privilege granted to individual users.

To grant the **SELECT** privilege on, for example, table columns in **qsys2** schema, use the following statement:

```
grant select on qsys2.syscolumns to hvruser;
```

Capture

HVR supports capturing changes from DB2 for i location. This section describes the configuration requirements for **capturing** changes from DB2 for i location. For the list of supported DB2 for i versions, from which HVR can capture changes, see Capture changes from location in Capabilities.

Table Types

HVR supports capture from the following table types in DB2 for i:

- Tables
- Physical files
- Source files

Log-Based Capture

HVR performs log-based capture from DB2 fo i location using the DISPLAY_JOURNAL table function.

- The **user** should have permission to select data from journal receivers. This can be achieved in two ways:

  1. Create a **user** profile (e.g. **hvruser**) and assign the special authority (***ALLOBJ**). For this, run the following command from AS/400 console :

     ```
     CRTUSRPRF USRPRF(HVRUSER) SPCAUT(*ALLOBJ)
     ```

  2. If ***ALLOBJ** authority cannot be granted to the **user** (or if the **user** does not have ***ALLOBJ** authority), then separate access rights should be given on each journal. For this, run the following commands from AS/400 console.

     a. Create a **user** profile (e.g. **hvruser**) :

     ```
     CRTUSRPRF USRPRF(HVRUSER)
     ```

     b. Grant the authority ***USE** on object (e.g. **HVR**) to **user** :

     ```
     GRTOBJAUT OBJ(HVR) OBJTYPE(*LIB) USER(HVRUSER) AUT(*USE)
     ```

     c. Grant the authority ***USE** and ***OBJEXIST** on journal (e.g. **HVR/QSQJRN**) to **user** :

     ```
     GRTOBJAUT OBJ(HVR/QSQJRN) OBJTYPE(*JRN) USER(HVRUSER) AUT(*USE)

     GRTOBJAUT OBJ(HVR/QSQJRN) OBJTYPE(*JRN) USER(HVRUSER) AUT(*OBJEXIST)
     ```

     d. Grant the authority ***USE** on all journal receiver (e.g. **HVR/*ALL**) to **user** :

     ```
     GRTOBJAUT OBJ(HVR/*ALL) OBJTYPE(*JRNRCV) USER(HVRUSER) AUT(*USE)
     ```

- Tables grouped in the same HVR channel should be using the same journal (**Capture /LogJournal**)
- All changes made to the replicated tables should be fully written to the journal receivers
- IBM i Table attribute *IMAGES* should be set to ***BOTH** or ***AFTER** (supported only if **ColumnProperties /CaptureFromRowId** and **/SurrogateKey** are defined, since HVR 5.7.0/0)
- To enable these settings for each replicated table the journaling needs to be stopped and started again with the new settings. Example, for table *TAB1_00001* in schema *HVR*:

```
ENDJRNPF FILE(HVR/TAB1_00001) JRN(HVR/QSQJRN)
STRJRNPF FILE(HVR/TAB1_00001) JRN(HVR/QSQJRN) IMAGES(*BOTH)
```

or

```
CHGJRNOBJ OBJ((HVR/*ALL *FILE)) ATR(*IMAGES) IMAGES(*BOTH)
```

- IBM i Journal attribute *MINENTDTA* should be set to ***NONE**
- IBM i Journal attribute *RCVSIZOPT* should contain either **MAXOPT3* or **MAXOPT2*. When using **MAXOPT2*, it is recommended to define action **Capture /LogJournalSysSeq**. Otherwise, if action **Capture /LogJournalSysSeq** is not defined and when the journal sequence numbers are reset, then **HVR Initialize** should be run with **Transaction Files and Capture Time** (option **-or**) to reset the capture start sequence, and if the target location is a database, also select option **State Tables** (option **-os**) to reset the target state tables. After executing **HVR Initialize**, optionally, run **HVR Refresh** to repair any changes from before the new capture start that were missed.

> ⚠ Action **Capture /LogJournalSysSeq** requires *FIXLENDTA* to contain **SYSSEQ*.

- The journal receivers should not be removed before HVR has been able to process the changes written in them.
- To enable these settings, run the following commands in the console. Example, for schema *HVR* running with **MAXOPT3*:

```
CHGJRN JRN(HVR/QSQJRN) JRNRCV(*GEN) MINENTDTA(*NONE) RCVSIZOPT(*MAXOPT3)
```

- For running with **MAXOPT2* and **Capture /LogJournalSysSeq**:

```
CHGJRN JRN(HVR/QSQJRN) JRNRCV(*GEN) MINENTDTA(*NONE) RCVSIZOPT(*MAXOPT2) FIXLENDTA
(*SYSSEQ)
```

- When Action **Capture /IgnoreSessionName** is used, the name of the user making a change should be logged. In that case, IBM i Journal attribute *FIXLENDTA* should contain **USR*. Example, for schema *HVR* running with **MAXOPT3*:

```
CHGJRN JRN(HVR/QSQJRN) JRNRCV(*GEN) MINENTDTA(*NONE) RCVSIZOPT(*MAXOPT3) FIXLENDTA
(*USR)
```

- For running with **MAXOPT2* and **Capture /LogJournalSysSeq**:

The **User** should have permission to use the *current schema* (default library) and to create and drop HVR state tables in it

```
grant createin, usage on schema current schema to hvruser
```

Alternatively, run the following command from AS/400 console:

```
  GRTOBJAUT OBJ(HVR) OBJTYPE(*CURLIB) USER(HVRUSER) AUT(*CHANGE)
```

Compare and Refresh Source

HVR supports compare and refresh (source location) into DB2 for i location. This section describes the configuration requirements for performing **HVR Compare** and **HVR Refresh** (source location) in DB2 for i location.

Grants for Compare and Refresh Source

The **User** should have permission to read replicated tables:

```
grant select on tbl to hvruser
```

Alternatively, run the following command from AS/400 console:

```
GRTOBJAUT OBJ(HVR/*ALL) OBJTYPE(*FILE) USER(HVRUSER) AUT(*USE)
```

Requirements for DB2 for Linux, UNIX and Windows

Contents

This section describes the requirements, access privileges, and other features of HVR when using DB2 for Linux, UNIX and Windows (LUW) for replication.

For the Capabilities supported by HVR on DB2 for Linux, UNIX and Windows, see Capabilities for DB2 for Linux, UNIX and Windows.

For information about the supported data types and mapping of data types in source DBMS to the corresponding data types in target DBMS or file format, see Data Type Mapping.

To quickly setup replication using DB2 for Linux, UNIX and Windows, see Quick Start for HVR - DB2 for LUW.

Supported Editions

HVR supports the following editions of DB2 for Linux, UNIX and Windows:

- Server Edition
- Advanced Enterprise Server Edition
- Express-C Edition

For information about compatibility and supported versions of DB2 for Linux, UNIX and Windows with HVR platforms, see Platform Compatibility Matrix.

Prerequisites

HVR requires DB2 client to be installed on the machine from which HVR connects to DB2. The DB2 client should have an instance to store the data required for the remote connection.

To setup the DB2 client, use the following commands to catalog the TCP/IP node and the remote database:

db2 catalog tcpip node *nodename* **remote** *nodename* **server** *portnumber*

db2 catalog database *databasename* **at node** *nodename*

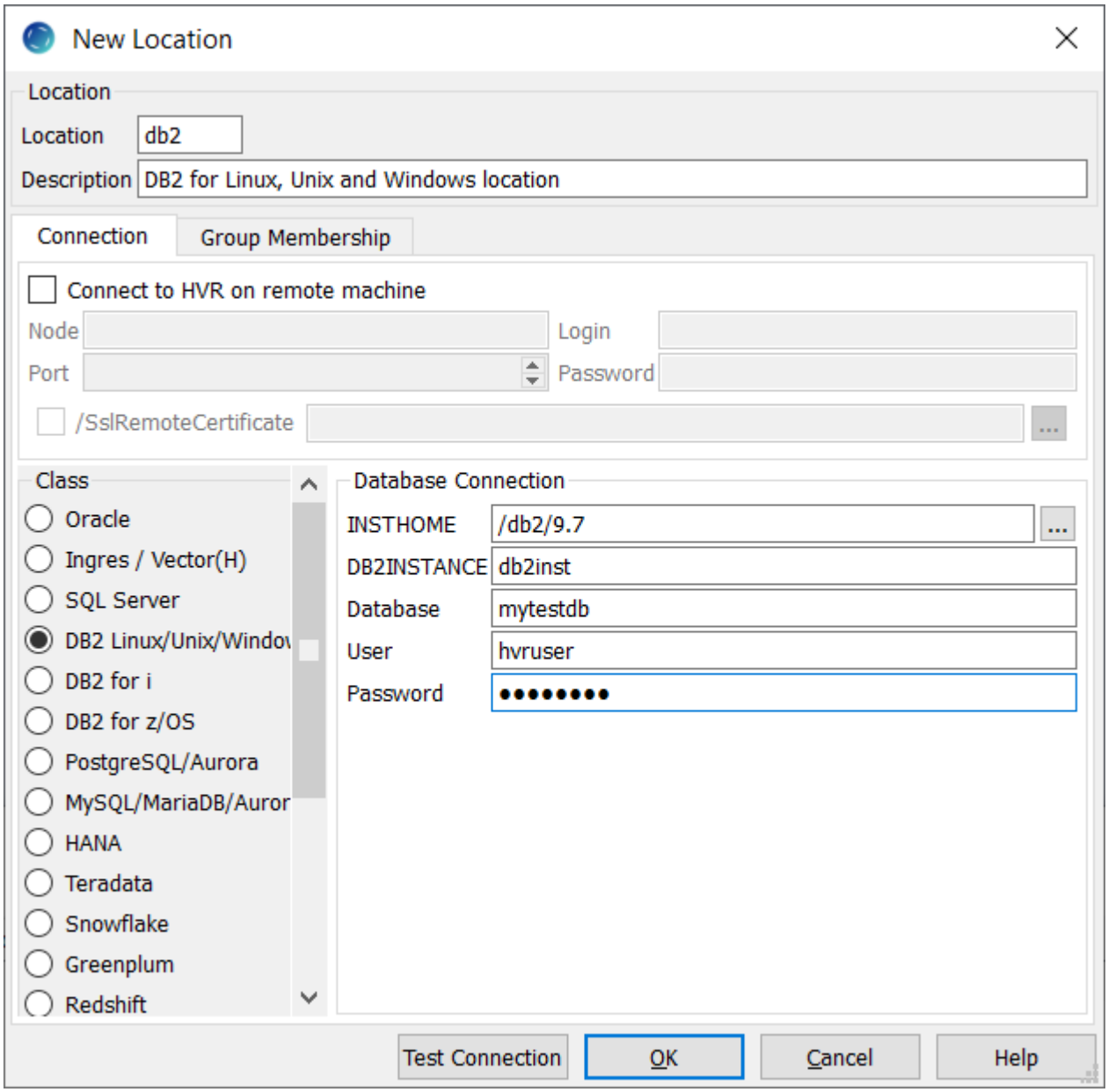
To test the connection with DB2 server, use the following command:

db2 connect to *databasename* **user** *username*

For more information about configuring DB2 client, refer to IBM Knowledge Center.

Location Connection

This section lists and describes the connection details required for creating DB2 for Linux, UNIX and Windows location in HVR. HVR uses SQL Call Level Interface to connect, read and write data to DB2 for Linux, UNIX and Windows location.



Field	Description
Database Connection	
INSTHOME	The directory path of the DB2 installation. **Example:** /db2/9.7
DB2INSTANCE	The name of the DB2 instance. **Example:** db2instl
Database	The name of the DB2 database. **Example:** mytestdb

User	The username to connect HVR to DB2 **Database**. **Example:** hvruser
Password	The password of the **User** to connect HVR to DB2 **Database**.

Hub

HVR allows you to create hub database in DB2 for Linux, UNIX and Windows. The hub database is a small database which HVR uses to control its replication activities. This database stores HVR catalog tables that hold all specifications of replication such as the names of the replicated databases, the list of replicated tables, and the replication direction.

Grants for Hub

To capture changes from source database or to integrate changes into target database, the following privileges are required:

- The **User** should have permission to **create** and **drop** HVR catalog tables.

```
grant createtab on database to user hvruser
```

Capture

HVR supports capturing changes from DB2 for Linux, UNIX and Windows. For the list of supported DB2 for Linux, UNIX and Windows versions, from which HVR can capture changes, see Capture changes from location in Capabilities.

Table Types

HVR supports capture from the following table types in DB2 for Linux, UNIX and Windows:

- Regular Tables
- Multidimensional Clustering (MDC) Tables
- Insert Time Clustering (ITC) Tables
- Uncompressed Tables
- Row Compressed Tables (both static and adaptive)
- Value Compressed Tables (both static and adaptive)

Log-based Capture

HVR uses the db2ReadLog API to read the DB2 transaction logs. For this the database user needs to have authorization **SYSADM** or **DBADM**.

Supplemental Logging

HVR supports supplemental logging for log-based capture from DB for Linux, UNIX and Windows.

Supplemental logging can be enabled while executing **HVR Initialize** by selecting option **Supplemental Logging** (option **-ol**).

- Alternatively, executing the following command on replicated tables has the same effect.

```
alter table tablename data capture changes include longvar columns
```

> To alter a table, the **User** should have one of the privileges (**alter** , **control** or **alterin**) or else the **User** should have **SYSADM** or **DBADM** authority.

While executing **HVR Initialize**, if supplemental logging is enabled, HVR also executes (only if required by DB2 to reorganize the tables for better performance) the following:

```
reorg table tablename
```

To enable "archive logging" in db2 , execute the following command:

```
db2 update db cfg for databasename using logarchmeth1 logretain
db2 update db cfg for databasename using logarchmeth2 off
```

> The user executing this command should be part of **SYSADM** , **SYSCTRL** or **SYSMAINT**. This does not have to be the HVR database user.

Integrate and Refresh Target

HVR supports integrating changes into DB2 for Linux, UNIX and Windows location. This section describes the configuration requirements for integrating changes (using **Integrate)** into DB2 for Linux, UNIX and Windows location. For the list of supported DB2 for Linux, UNIX and Windows versions, into which HVR can integrate changes, see Integrate changes into location in Capabilities.

- The **User** should have permission to read and change replicated tables.

```
grant select on tbl to user hvruser
grant insert on tbl to user hvruser
grant update on tbl to user hvruser
grant delete on tbl to user hvruser
```

- The **User** should have permission to load data.

```
grant load on database to user hvruser
```

- The **User** should have permission to **create** and **drop** HVR state tables.

```
grant createtab on database to user hvruser
```

Burst Integrate and Bulk Refresh

HVR uses db2Load API for copying data to a target during bulk refresh and loading data into burst tables during **Integrate** with **/Burst**.

Compare and Refresh Source

The **User** should have permission to read replicated tables

```
grant select on tbl to user hvruser
```

Requirements for DB2 for z/OS

Since v5.5.5/6

Contents

This section describes the requirements, access privileges, and other features of HVR when using 'DB2 for z/OS' for replication. For information about compatibility and supported versions of DB2 for z/OS with HVR platforms, see Platform Compatibility Matrix.

For the Capabilities supported by HVR on DB2 for z/OS, see Capabilities for DB2 for z/OS.

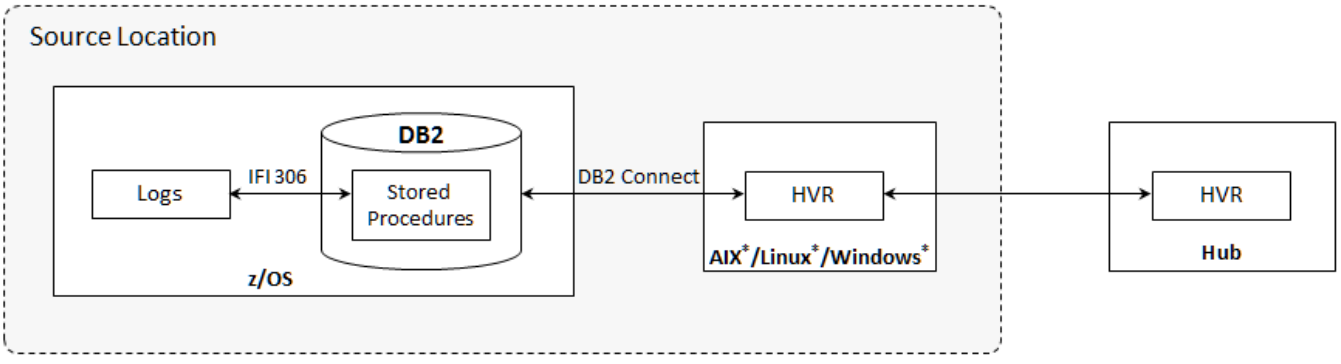
For information about the supported data types and mapping of data types in source DBMS to the corresponding data types in target DBMS or file format, see Data Type Mapping.

> HVR does not support the DB2 data sharing feature - Sysplex.

Introduction

To capture from DB2 for z/OS, HVR needs to be installed on a separate machine(either 64-bit Linux on Intel or 64-bit Windows on Intel or 64-bit AIX on PowerPC) from which HVR will access DB2 on z/OS machine. Additionally, the HVR stored procedures need to be installed on DB2 for z/OS machine for accessing DB2 log files. For steps to install the stored procedures on DB2 for z/OS machine, refer to section Installing HVR Capture Stored Procedures on DB2 for z/OS.

HVR requires 'DB2 Connect' for connecting to DB2 for z/OS.



Prerequisites for HVR Machine

HVR requires DB2 client or DB2 server or DB2 Connect to be installed on the machine from which HVR connects to DB2 on z/OS. The DB2 client should have an instance to store the data required for the remote connection.

To setup the DB2 client or DB2 server or DB2 Connect, use the following commands to catalog the TCP/IP node and the remote database:

```
db2 catalog tcpip node nodename remote hostname server portnumber
db2 catalog database databasename at node nodename
```

- *nodename* is the local nickname for the remote machine that contains the database you want to catalog.
- *hostname* is the name of the host/node where the target database resides.
- *databasename* is the name of the database you want to catalog.

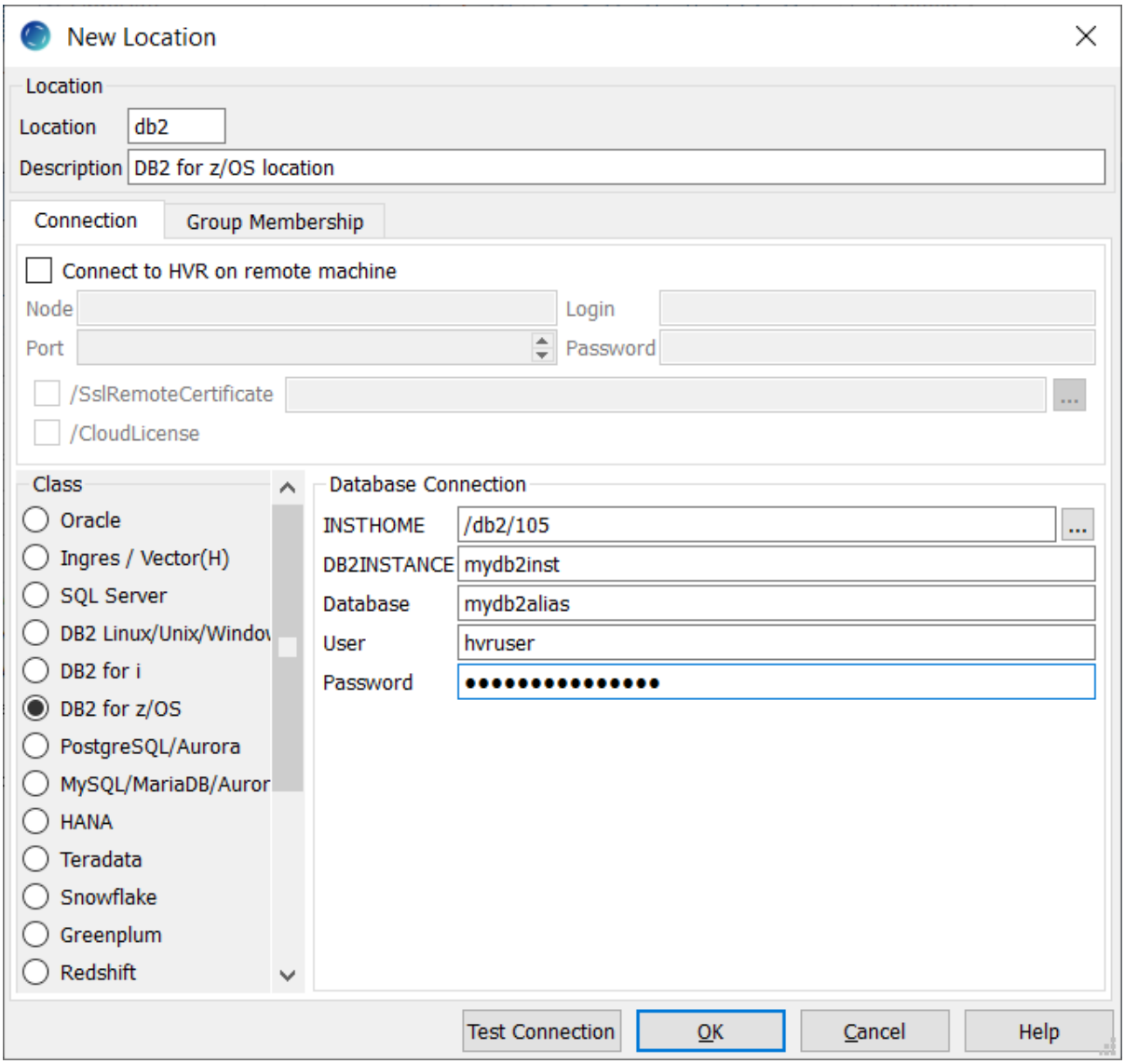
For more information about configuring DB2 client or DB2 server or DB2 Connect, refer to IBM documentation.

To test the connection with DB2 server on the z/OS machine, use the following command:

```
db2 connect to databasename user username
```

Location Connection

This section lists and describes the connection details required for creating DB2 for z/OS location in HVR. HVR connects, reads and writes data to DB2 for Linux, UNIX and Windows location using SQL Call Level Interface via db2Connect.



Field	Description
Database Connection	
INSTHOME	The directory path of the DB2 installation on HVR machine. **Example:** /db2/11.1
DB2INSTANCE	The name of the DB2 instance. **Example:** mydb2inst
Database	The database alias for DB2 for z/OS. **Example:** mydb2alias
User	The username to connect HVR to **Database**. **Example:** hvruser
Password	The password of the **User.**

Capture

HVR supports capturing changes from DB2 for z/OS. This section describes the configuration requirements for capturing changes from DB2 for z/OS location. For the list of supported DB2 for z/OS versions, from which HVR can capture changes, see Capture changes from location in Capabilities.

HVR uses IFI 306 via HVR stored procedures to capture data from DB2 for z/OS locatioin.

Table Types

HVR supports capture from the following table types in DB2 for z/OS:

- Regular Tables
- Compressed Tables
- Partitioned Tables

Grants for Capture

The following grants are required for capturing changes from DB2 for z/OS:

1. To create stored procedures, the **User** must be granted **createin** privilege on the schema.

```
grant createin on schema myschema to authid;
```

2. To read information from the transaction log, the **User** must be granted **monitor2** privilege.

```
grant monitor2 to hvruser;
```

3. To execute stored procedures created by the *authid* user, the **User** must be granted **execute on procedure** privilege for the stored procedures - **hvr.hvrcaplg** and **hvr.hvrcapnw**.

```
grant execute on procedure hvr.hvrcaplg to hvruser;
grant execute on procedure hvr.hvrcapnw to hvruser;
```

4. To fetch information about the DB2 for z/OS installation, the **User** must be granted **select** privilege for the following **SYSIBM** tables.

```
grant select on table sysibm.sysauxrels to hvruser;

grant select on table sysibm.syscolauth to hvruser;

grant select on table sysibm.syscolumns to hvruser;

grant select on table sysibm.sysdatabase to hvruser;

grant select on table sysibm.sysforeignkeys to hvruser;

grant select on table sysibm.sysindexes to hvruser;

grant select on table sysibm.syskeys to hvruser;

grant select on table sysibm.sysparms to hvruser;

grant select on table sysibm.sysrels to hvruser;

grant select on table sysibm.sysroutines to hvruser;

grant select on table sysibm.syssynonyms to hvruser;

grant select on table sysibm.systabauth to hvruser;

grant select on table sysibm.systablepart to hvruser;

grant select on table sysibm.systables to hvruser;
```

Supplemental Logging

Supplemental logging can be enabled by defining action **HVR Initialize /Supplemental Logging** or by using the command **hvrinit -ol**.

To enable supplemental logging, the **User** should be either owner of the replicated tables or have DBADM or SYSADM or SYSCTRL authority.

Alternatively, executing the following command on replicated tables has the same effect:

```
alter table tablename data capture changes;
```

Integrate and Refresh Target

HVR supports integrating changes into DB2 for z/OS location. This section describes the configuration requirements for integrating changes (using **Integrate** and **Refresh**) into DB2 for z/OS location. For the list of supported DB2 for z /OS versions, into which HVR can integrate changes, see Integrate changes into location in Capabilities.

Grants for Integrate and Refresh Target

The following grants are required for integrating changes into DB2 for z/OS:

1. To read and change the replicated tables, the **User** must be granted **select**, **insert**, **update**, and **delete** privileges.

   ```
   grant select, insert, update, delete on table myschema.mytable to hvruser;
   ```

2. To create and drop HVR state tables, the **User** must be granted **createtab** privilege.

   ```
   grant createtab on database mydatabase to hvruser;
   ```

3. To fetch information about the DB2 for z/OS installation, the **User** must be granted **select** privilege for the following **SYSIBM** tables:

```
grant select on table sysibm.sysauxrels to hvruser;
grant select on table sysibm.syscolauth to hvruser;
grant select on table sysibm.syscolumns to hvruser;
grant select on table sysibm.sysdatabase to hvruser;
grant select on table sysibm.sysforeignkeys to hvruser;
grant select on table sysibm.sysindexes to hvruser;
grant select on table sysibm.syskeys to hvruser;
grant select on table sysibm.sysparms to hvruser;
grant select on table sysibm.sysrels to hvruser;
grant select on table sysibm.sysroutines to hvruser;
grant select on table sysibm.syssynonyms to hvruser;
grant select on table sysibm.systabauth to hvruser;
grant select on table sysibm.systablepart to hvruser;
grant select on table sysibm.systables to hvruser;
```

Compare and Refresh Source

HVR supports compare and refresh (source location) in DB2 for z/OS location. This section describes the configuration requirements for performing **HVR Compare** and **HVR Refresh** (source location) in DB2 for z/OS location.

Grants for Compare and Refresh Source

The following grant is required for performing **HVR Compare** and **HVR Refresh** (source location) in DB2 for z/OS:

To read from the replicated tables, the **User** must be granted **select** privilege.

```
grant select on table myschema.mytable to hvruser;
```

Installing HVR Capture Stored Procedures on DB2 for z/OS

This section describes the process of installing the HVR capture stored procedures on DB2 for z/OS machine needed for accessing DB2 log files. During change data capture (CDC) replication, HVR calls the external stored procedures that will extract the information from DB2 log files using the IFI 306 interface.

To install the stored procedures on DB2 for z/OS machine, follow the steps listed below.

1. Allocate sequential data sets

The stored procedures are designed to store the compiled logic and are put in sequential data sets using z/OS command **XMIT**. The stored procedures need to be copied to the z/OS machine, from which capture is performed, and unpacked using z/OS command **RECEIVE**.

The sequential data sets should be allocated first using the following **JCL** script.

Note that the **HLQ** in the script (**IBMUSER.HVR**) has to be adapted to the required one on your system.

```
//HVRALLOC JOB,ZHERO,CLASS=A,MSGCLASS=X,NOTIFY=&SYSUID
//ALLOC EXEC PGM=IEFBR14
//HVRCNTL DD DSN=IBMUSER.HVR.CNTL.SEQ,
//DCB=(RECFM=FB,LRECL=80,BLKSIZE=3120,DSORG=PS),
//SPACE=(CYL,(5,2)),DISP=(,CATLG)
//HVRDBRM DD DSN=IBMUSER.HVR.DBRM.SEQ,
//DCB=(RECFM=FB,LRECL=80,BLKSIZE=3120,DSORG=PS),
//SPACE=(CYL,(5,2)),DISP=(,CATLG)
//HVRLOAD DD DSN=IBMUSER.HVR.LOADLIB.SEQ,
//DCB=(RECFM=FB,LRECL=80,BLKSIZE=3120,DSORG=PS),
//SPACE=(CYL,(5,2)),DISP=(,CATLG)
//HVRPROC DD DSN=IBMUSER.HVR.PROCLIB.SEQ,
//DCB=(RECFM=FB,LRECL=80,BLKSIZE=3120,DSORG=PS),
//SPACE=(CYL,(5,2)),DISP=(,CATLG)
/*
```

This should allocate the following data sets:

- **HLQ.CNTL.SEQ**
- **HLQ.DBRM.SEQ**
- **HLQ.LOADLIB.SEQ**
- **HLQ.PROCLIB.SEQ**

2. Transfer the sequential data set contents to the z/OS machine

To transfer the sequential data sets to the DB2 for z/OS machine, use the binary transfer.

For example, using ftp:

```
ftp <hostname>
ftp> bin
ftp> cd <HLQ>
ftp> put CNTL.SEQ
ftp> put DBRM.SEQ
ftp> put LOADLIB.SEQ
ftp> put PROCLIB.SEQ
```

This should populate the following data sets:

- **HLQ.CNTL.SEQ**
- **HLQ.DBRM.SEQ**
- **HLQ.LOADLIB.SEQ**
- **HLQ.PROCLIB.SEQ**

3. Receive the sequential data sets creating the actual data sets required by HVR

This can be done using the following **JCL** script.

Note that the **HLQ** in the script (**IBMUSER.HVR**) has to be adapted for your system.

```
//HVRRCV JOB ,ZHERO,CLASS=A,MSGCLASS=X,NOTIFY=&SYSUID
//IKJCMD EXEC PGM=IKJEFT01
//SYSPRINT DD SYSOUT=*
//SYSTSPRT DD SYSOUT=*
//SYSTSIN DD *
RECEIVE INDS ('IBMUSER.HVR.CNTL.SEQ')
DA('IBMUSER.HVR.CNTL')
RECEIVE INDS ('IBMUSER.HVR.DBRM.SEQ')
DA('IBMUSER.HVR.DBRM')
RECEIVE INDS ('IBMUSER.HVR.LOADLIB.SEQ')
DA('IBMUSER.HVR.LOADLIB')
RECEIVE INDS ('IBMUSER.HVR.PROCLIB.SEQ')
DA('IBMUSER.HVR.PROCLIB')
/*
```

This should create the following data sets:

- **HLQ.CNTL**
- **HLQ.DBRM**
- **HLQ.LOADLIB**
- **HLQ.PROCLIB**

4. APF authorize HLQ.LOADLIB

HLQ.LOADLIB should be added to the APF authorized library list.

5. Set Up Dedicated WLM Environment for HVR

For HVR to capture changes from DB2 on z/OS, the following are required on the z/OS machine:

- The loadlib (PDSE) authorized by the Authorized Program Facility (APF).
- HVR uses stored procedures that must run in a Workload Manager (WLM) environment. It is recommended to use a dedicated WLM environment for HVR (e.g. **HVRWLM**).
- The WLM environment should have an APF authorized address space.
- The recommended value for parameters while setting up the WLM environment for stored procedures are:
  - For an HVR dedicated WLM environment - **TIME**=**NOLIMIT**, **NUMTCB=1** (indicates one task per address space)
  - For a shared WLM environment - **TIME**=**NOLIMIT**, **NUMTCB=** a value between **10** and **40**.
- The Resource Recovery Services (RRS) should be active to run HVR's WLM managed stored procedures. The user ID that is associated with the WLM-established stored procedures address space must be authorized to run Recoverable Resource Manager Services Attachment Facility (RRSAF) and is associated with the **ssnm.RRSAF** profile.
- The Language Environment (LE) should be active for running C.
- To read the System Management Facility (SMF) log records, it is required to use the privileged **API IFI IFCID 306**.
- Trace for **IFCID 306** should be active.
- Permission to CONNECT to the DB2 subsystem from the HVR machine.

6. Adapt HLQ.CNTL(HVRCAP) (uploaded as part of .SEQ files)

This **JCL** script is used to create HVR's stored procedures. It needs to be adapted to your system:

- **IBMUSER.HVR** needs to be replaced with the actual **HLQ** used.
- **DBBG** needs to be replaced with the name of the actual DB2 installation.
- **HVRWLM** needs to be replaced with the name of the actual WLM going to be used.
- **HVRUSER** needs to be replaced with the name of z/OS user used by HVR to connect to DB2.

Requirements for FTP, SFTP, and SharePoint WebDAV

Contents

General

HVR supports different kinds of file locations; regular ones (a directory on a local file system), FTP file locations, SFTP file locations, and WebDAV file locations (this is the protocol used by HVR to connect to Microsoft SharePoint).

Generally, the behavior of HVR replication is the same for all of these kinds of file locations; capture is defined with action **Capture** and integration is defined with **Integrate**. All other file location parameters are supported and behave normally.

If HVR will be using a file protocol to connect to a file location (e.g. FTP, SFTP or WebDAV), it can either connect with this protocol directly from the hub machine, or it first connects to a remote machine with HVR's own remote protocol and then connect to the file location from that machine (using FTP, SFTP or WebDAV).

A small difference is the timing and latency of capture jobs. Normal file capture jobs check once a second for new files, whereas if a job is capture from a non local file location, then it only checks every 10 seconds. Also if **Capture** is defined without **/DeleteAfterCapture**, then the capture job may have to wait for up to a minute before capturing new files; this is because these jobs rely on comparing timestamps, but the file timestamps in the FTP protocol have a low granularity (minutes not seconds).

A proxy server to connect to FTP, SFTP or WebDAV can be configured with action **LocationProperties /Proxy**.

HVR uses the cURL library to communicate with the file systems.

FTPS, WebDAV: Server Authentication

FTP and WebDAV support certificates for authentication. These are held in the certificate directory **$HVR_HOME/lib/cert** (see section Files in Hvr). The following files are included or can be easily created:

- **ca-bundle.crt**: Used by HVR to authenticate SSL servers (FTPS, secure WebDAV, etc). It can be overridden by creating new file *host* **.pub_cert** in this same certificate directory. No authentication is done if neither file is found. Delete or move both files to disable FTPS authentication. This file can be copied from e.g. **/usr/share/ssl/certs /ca–bundle.crt** on Unix/Linux.
- *host* **.pub_cert**: Used to override **ca-bundle.crt** for server verification for *host* .
  FTP connections can be unencrypted or they can have three types of encryption; this is called FTPS, and should not be confused with SFTP. These FTPS encryption types are SSL/TLS implicit encryption (standard port: 990), SSL explicit encryption and TLS explicit encryption (standard port: 21).

Note that if the FTP/SFTP connection is made via a remote HVR machine, then the certificate directory on the remote HVR machine is used, not the one on the hub machine.

FTPS, WebDAV: Client Authentication

Normally, clients (HVR) are authenticated using a username and password. In addition to this, you can use client-side SSL certificates.
To set this up, you need to have a public/private key pair in PEM format. Place them in **$HVR_HOME/lib/cert** as: *client***.pub_cert** and *client***.priv_key**, and add the environment variables to a file location:

- **Environment /Name=HVR_SSL_CLIENT_CERT /Value=** *client* **.pub_cert**

- **Environment /Name=HVR_SSL_CLIENT_KEY /Value=** *client* **.priv_key**
  If you have a password/phrase on your key, set:
- **Environment /Name=HVR_SSL_CLIENT_KEY_PASSWD /Value=** *passwd*
  You can encrypt this password for HVR by running:

```
$ hvrcrypt client passwd
```

- If you have a PKCS#12 or PFX file, you can convert it to PEM format using openssl:

```
$ openssl pkcs12 -in cert.p12 -clcerts -nokeys -out client.pub_cert
$ openssl pkcs12 -in cert.p12 -nocerts -out client.priv_key
```

SFTP: Server Authentication

A file (named **$HVR_CONFIG/files/known_hosts**) is used internally during SFTP connections. It is updated the first time HVR connects to reach the SFTP machine. On Unix or Linux this file can be initialized by copying it from **$HOME/.ssh/known_hosts**.

Note that if the FTP/SFTP connection is made via a remote HVR machine, then the certificate directory on the remote HVR machine is used, not the one on the hub machine.

SFTP: Client Authentication

Normally, clients (HVR) are authenticated using a username and password. Instead of this, you can use client-side keys. To set this up, you need to have a public/private key pair. Place them in **$HVR_HOME/lib/cert**, and add the environment variables to a file location:

- **Environment /Name=HVR_SSH_PUBLIC_KEY /Value=** *client***.pub**
- **Environment /Name=HVR_SSH_PRIVATE_KEY /Value=** *client*
  If you have a password/phrase on your private key, you can set it in the password field of the location.

You can generate keys using:

```
$ ssh-keygen -f client
```

WebDAV: Versioning

HVR can replicate to and from a WebDAV location which has versioning enabled. By default, HVR's file integrate will delete the SharePoint file history, but the file history can be preserved if action **LocationProperties /StateDirectory** is used to configure a state directory (which is the then on the HVR machine, outside SharePoint). Defining a **/StateDirectory** outside SharePoint does not impact the 'atomicity' of file integrate, because this atomicity is already supplied by the WebDAV protocol.

Requirements for Google Cloud Storage

Since v5.6.5/2

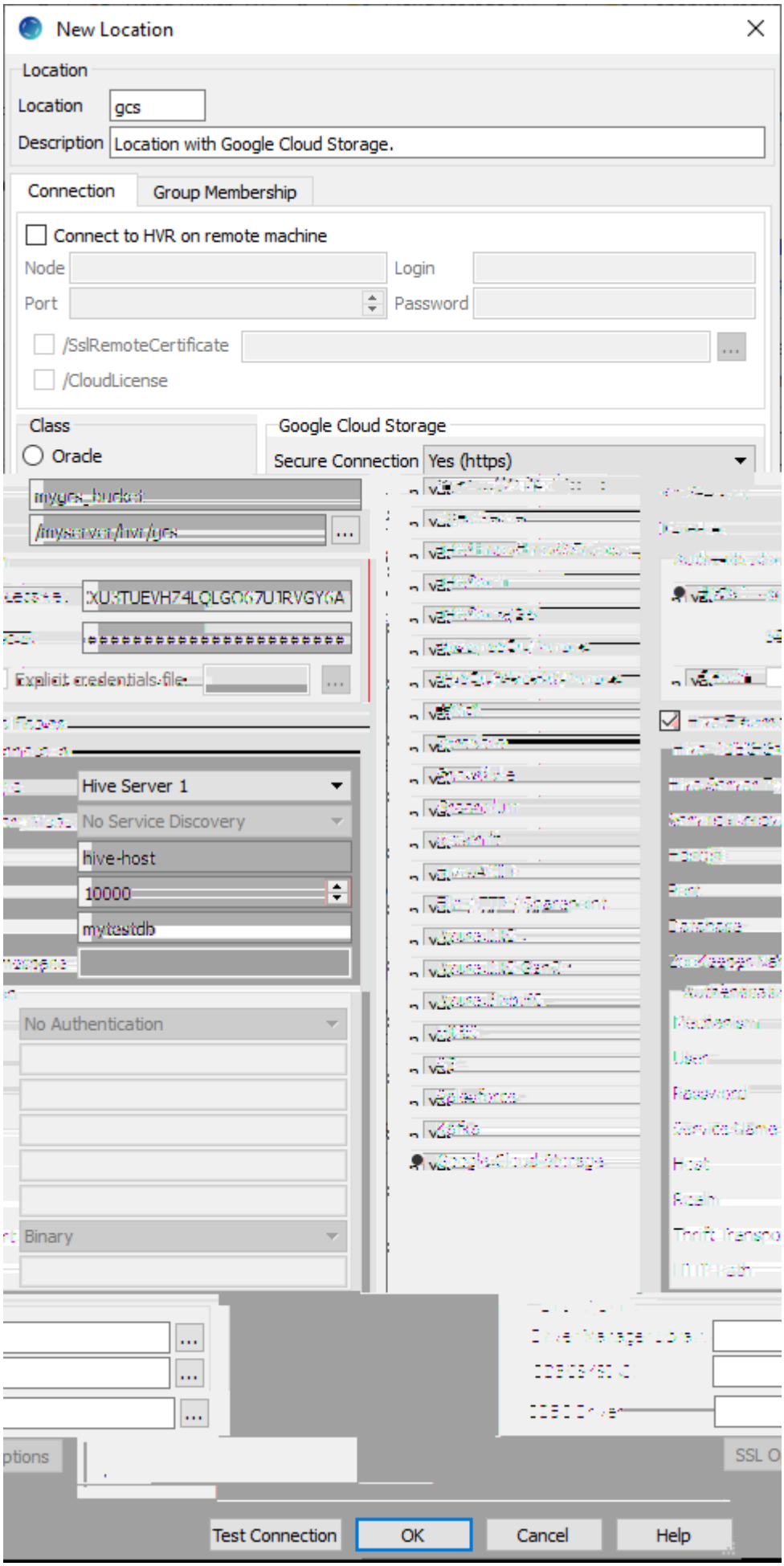
Contents

This section describes the requirements, access privileges, and other features of HVR when using Google Cloud Storage (GCS) for replication. For information about compatibility and support for Google Cloud Storage with HVR platforms, see Platform Compatibility Matrix.

For the capabilities supported by HVR, see Capabilities.

Location Connection

This section lists and describes the connection details required for creating Google Cloud Storage location in HVR. HVR uses GCS S3-compatible API (cURL library) to connect, read and write data to Google Cloud Storage during capture, integrate (continuous), refresh (bulk) and compare (direct file compare).



Field	Description
Google Cloud Storage	
Secure Connection	The type of security to be used for connecting HVR to Google Cloud Storage Server. Available options: • **Yes (https)** (default): HVR will connect to Google Cloud Storage Server using HTTPS. • **No (http)**: HVR will connect to Google Cloud Storage Server using HTTP.
GCS Bucket	The IP address or hostname of the Google Cloud Storage bucket. **Example:** mygcs_bucket
Directory	The directory path in **GCS Bucket** which is to be used for replication. **Example:** /myserver/hvr/gcs
Authentication	
HMAC	The HMAC authentication mode for connecting HVR to Google Cloud Storage by using the Hash-based Message Authentication Code (HMAC) keys (**Access key** and **Secret**). For more information, refer to HMAC Keys in Google Cloud Storage documentation.
Access Key	The HMAC access ID of the service account to connect HVR to the Google Cloud Storage. This field is enabled only when the authentication mode is **HMAC**. **Example:** GOOG2EIWQKJJO6C4R5WKCXU3TUEVHZ4LQLGO67UJRVGY6A
Secret	The HMAC secret of the service account to connect HVR to the Google Cloud Storage. This field is enabled only when the authentication mode is **HMAC**.
OAuth	The OAuth 2.0 protocol based authentication for connecting HVR to Google Cloud Storage by using the credentials fetched from the environment variable **GOOGLE_APPLICATION_CREDENTIALS**. For more information about configuring this environment variable, see Getting Started with Authentication in Google Cloud Storage documentation.
Explicit credentials file	The OAuth 2.0 protocol based authentication for connecting HVR to Google Cloud Storage by using the service account key file (JSON). This field is enabled only when the authentication mode is **OAuth**. For more information about creating service account key file, see Authenticating With a Service Account Key File in Google Cloud Storage documentation.
Hive External Tables	Enable/Disable Hive ODBC connection configuration for creating Hive external tables above Google Cloud Storage.

Hive ODBC Connection

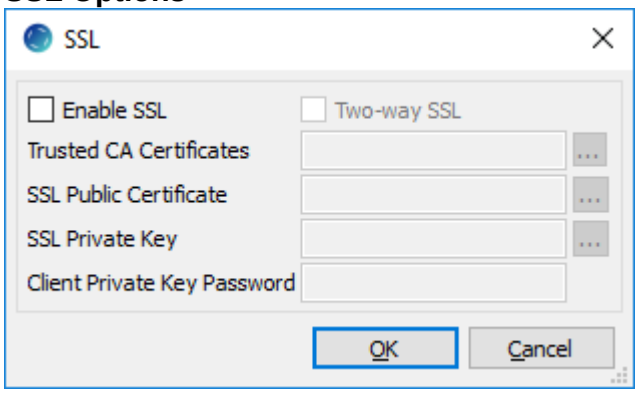
HVR uses the Hive ODBC driver to connect to Hive for creating Hive external tables above Google Cloud Storage to perform hvrcompare with files that reside on Google Cloud Storage. You can enable/disable the Hive configuration for Google Cloud Storage in the **New Location** screen using the **Hive External Tables** field.

Field	Description
Hive ODBC Connection	
Hive Server Type	The type of Hive server. Available options: • **Hive Server 1** (default): The driver connects to a Hive Server 1 instance. • **Hive Server 2**: The driver connects to a Hive Server 2 instance.

Service Discovery Mode	The mode for connecting to Hive. This field is enabled only if **Hive Server Type** is **Hive Server 2**. Available options: • **No Service Discovery** (default): The driver connects to Hive server without using the ZooKeeper service. • **ZooKeeper**: The driver discovers Hive Server 2 services using the ZooKeeper service.
Host(s)	The hostname or IP address of the Hive server. When Service Discovery Mode is ZooKeeper, specify the list of ZooKeeper servers in following format **[ZK_Host1]:[ZK_Port1],[ZK_Host2]:[ZK_Port2]**, where **[ZK_Host]** is the IP address or hostname of the ZooKeeper server and **[ZK_Port]** is the TCP port that the ZooKeeper server uses to listen for client connections. **Example:** hive-host
Port	The TCP port that the Hive server uses to listen for client connections. This field is enabled only if **Service Discovery Mode** is **No Service Discovery**. **Example:** 10000
Database	The name of the database schema to use when a schema is not explicitly specified in a query. **Example:** mytestdb
ZooKeeper Namespace	The namespace on ZooKeeper under which Hive Server 2 nodes are added. This field is enabled only if **Service Discovery Mode** is **ZooKeeper**.
Authentication	
Mechanism	The authentication mode for connecting HVR to **Hive Server 2**. This field is enabled only if **Hive Server Type** is **Hive Server 2**. Available options: • **No Authentication** (default) • **User Name** • **User Name and Password** • **Kerberos** • **Windows Azure HDInsight Service** **Since** v5.5.0/2
User	The username to connect HVR to Hive server. This field is enabled only if **Mechanism** is **User Name** or **User Name and Password**. **Example:** dbuser
Password	The password of the **User** to connect HVR to Hive server. This field is enabled only if **Mechanism** is **User Name and Password**.
Service Name	The Kerberos service principal name of the Hive server. This field is enabled only if **Mechanism** is **Kerberos**.
Host	The Fully Qualified Domain Name (FQDN) of the Hive Server 2 host. The value of **Host** can be set as **_HOST** to use the Hive server hostname as the domain name for Kerberos authentication. If **Service Discovery Mode** is disabled, then the driver uses the value specified in the Host connection attribute. If **Service Discovery Mode** is enabled, then the driver uses the **Hive Server 2** host name returned by ZooKeeper. This field is enabled only if **Mechanism** is **Kerberos**.
Realm	The realm of the Hive Server 2 host. It is not required to specify any value in this field if the realm of the Hive Server 2 host is defined as the default realm in Kerberos configuration. This field is enabled only if **Mechanism** is **Kerberos**.

Thrift Transport Since v5.5.0/2	The transport protocol to use in the Thrift layer. This field is enabled only if **Hive Server Type** is **Hive Server 2**. Available options: • **Binary** (This option is available only if **Mechanism** is **No Authentication** or **User Name and Password**.) • **SASL** (This option is available only if **Mechanism** is **User Name** or **User Name and Password** or **Kerberos**.) • **HTTP** (This option is not available if **Mechanism** is **User Name**.) ⓘ For information about determining which Thrift transport protocols your Hive server supports, refer to HiveServer2 Overview and Setting Up HiveServer2 sections in Hive documentation.
HTTP Path Since v5.5.0/2	The partial URL corresponding to the Hive server. This field is enabled only if **Thrift Transport** is **HTTP**.
Linux / Unix	
Driver Manager Library	The directory path where the Unix ODBC Driver Manager Library is installed. **Example:** /opt/unixodbc-2.3.2/lib
ODBCSYSINI	The directory path where **odbc.ini** and **odbcinst.ini** files are located. **Example:** /opt/unixodbc-2.3.2/etc
ODBC Driver	The user defined (installed) ODBC driver to connect HVR to the Hive server.
SSL Options	Show **SSL Options**.

SSL Options



Field	Description
Enable SSL	Enable/disable (one way) SSL. If enabled, HVR authenticates the Hive server by validating the SSL certificate shared by the Hive server.
Two-way SSL	Enable/disable two way SSL. If enabled, both HVR and Hive server authenticate each other by validating each others SSL certificate. This field is enabled only if **Enable SSL** is selected.
Trusted CA Certificates	The directory path where the **.pem** file containing the server's public SSL certificate signed by a trusted CA is located. This field is enabled only if **Enable SSL** is selected.

SSL Public Certificate	The directory path where the **.pem** file containing the client's SSL public certificate is located. This field is enabled only if **Two-way SSL** is selected.
SSL Private Key	The directory path where the **.pem** file containing the client's SSL private key is located. This field is enabled only if **Two-way SSL** is selected.
Client Private Key Password	The password of the private key file that is specified in **SSL Private Key**. This field is enabled only if **Two-way SSL** is selected.

Requirements for Greenplum

Contents

This section describes the requirements, access privileges, and other features of HVR when using Greenplum for replication. For information about compatibility and supported versions of Greenplum with HVR platforms, see Platform Compatibility Matrix.

For the Capabilities supported by HVR on Greenplum, see Capabilities for Greenplum.

For information about the supported data types and mapping of data types in source DBMS to the corresponding data types in target DBMS or file format, see Data Type Mapping.

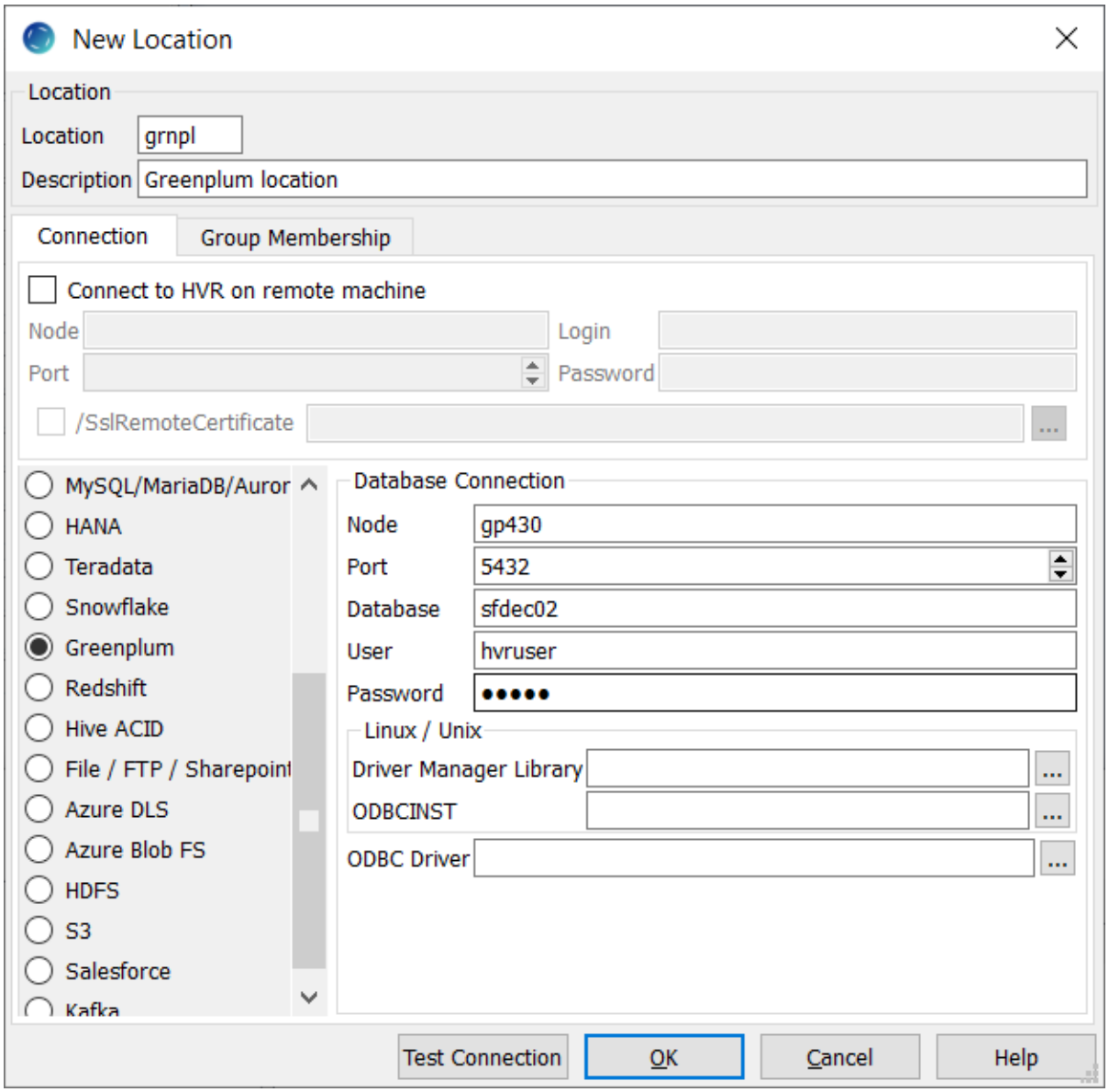
To quickly setup replication using Greenplum, see Quick Start for HVR - Greenplum.

ODBC Connection

HVR is not required on any of the nodes of the Greenplum cluster. It can be installed on a standalone machine, from which it connects to the Greenplum cluster. HVR requires the DataDirect Connect XE ODBC driver for Greenplum installed (on the machine from which HVR connects to a Greenplum server). HVR only supports the ODBC driver version from 7.1.3.99 to 7.1.6.

Location Connection

This section lists and describes the connection details required for creating Greenplum location in HVR.



Field	Description
Database Connection	
Node	The hostname or ip-address of the machine on which the Greenplum server is running. **Example:** gp430
Port	The port on which the Greenplum server is expecting connections. **Example:** 5432
Database	The name of the Greenplum database. **Example:** sfdec02
User	The username to connect HVR to the Greenplum **Database** . **Example:** hvruser
Password	The password of the **User** to connect HVR to the Greenplum **Database** .
Linux / Unix	
Driver Manager Library	The directory path where the Unix ODBC Driver Manager Library is installed. **Example:** /opt/Progress/DataDirect/Connect64_for_ODBC_71/lib
ODBCINST	The directory path where **odbcinst.ini** file is located. **Example:** /opt/Progress/DataDirect/Connect64_for_ODBC_71/odbcinst.ini

ODBC Driver	The user defined (installed) ODBC driver to connect HVR to the Greenplum server.

Integrate and Refresh Target

HVR supports integrating changes into a Greenplum location. This section describes the configuration requirements for integrating changes (using **Integrate** and **refresh**) into Greenplum location. For the list of supported Greenplum versions, into which HVR can integrate changes, see Integrate changes into location in Capabilities.

HVR uses DataDirect Connect XE ODBC driver to write data to Greenplum during continuous **Integrate** and row-wise **Refresh**. However, the preferred methods for writing data to Greenplum is **Integrate** with **/Burst** and Bulk **Refresh** using staging as they provide better performance.

Burst Integrate and Bulk Refresh

While **HVR Integrate** is running with parameter **/Burst** and Bulk **Refresh**, HVR can stream data into a target database straight over the network into a bulk loading interface specific for each DBMS (e.g. direct-path-load in Oracle), or else HVR puts data into a temporary directory ('staging file') before loading data into a target database.

For best performance, HVR performs **Integrate** with **/Burst** and Bulk **Refresh** into Greenplum using staging files and the Greenplum Parallel File Distribution (**gpfdist**) server. To use the **gpfdist** server for bulk loading operations, ensure that **gpfdist** is configured on the machine from which HVR will connect to Greenplum.

HVR implements **Integrate** with **/Burst** and Bulk **Refresh** (with file staging) into Greenplum as follows:

1. HVR first writes data into a temporary file in a staging directory on the machine where HVR connects to Greenplum. This directory does not have to be on the Greenplum database machine. The temporary file is written in the **.csv** format and is compressed.
2. HVR then uses Greenplum SQL '**copy**' command to pull the compressed data from **gpfdist://** or **gpfdists://** directory into a target table. This requires that a special Greenplum 'external table' exists for each target table that HVR loads data into. HVR will create these tables with names having the following patterns '**__x**' or '**__bx**'.

To perform **Integrate** with **/Burst** and Bulk **Refresh**, define action **LocationProperties** on a Greenplum location with the following parameters:

- **/StagingDirectoryHvr**: the location where HVR will create the temporary staging files. This should be the **-d** (directory) option of the **gpfdist** server command.
- **/StagingDirectoryDb**: the location from where Greenplum will access the temporary staging files. This should be set to **gpfdist:** *//<hostname>:<port>* where *hostname* is the name of the machine used to connect to Greenplum and *port* is the **-p** (http port) option of the **gpfdist** server command.

Example of a **gpfdist** command line in Linux & Unix:

```
/opt/gpfdist-4.3.0/gpfdist -p 33333 -d /home/hvr/tsuite_staging -l /home/hvr/staging
/gpfdist.log -m 10485760
```

On Windows, **gpfdist** is a service and the values can be retrieved from the "Path to Executable" in the properties dialog of the service.

Grants for Compare, Refresh and Integrate

The **User** should have the following privileges:

```
grant connect and create table on the database
grant select, insert, update, delete on replicated tables
```

If **User** needs to change tables which are in another schema (using action **TableProperties /Schema=***myschema*) then the following grants are needed:

```
grant usage on myschema to hvruser ;
```

When **HVR Refresh** is used to create the target tables, the following privilege is also needed:

```
grant create on myschema to hvruser ;
```

HVR's internal tables, like burst and state-tables, will be created in schema **public**.

Requirements for HANA

Contents

This section describes the requirements, access privileges, and other features of HVR when using SAP HANA for replication. For information about compatibility and supported versions of HANA with HVR platforms, see Platform Compatibility Matrix.

For the Capabilities supported by HVR on HANA, see Capabilities for HANA.

For information about the supported data types and mapping of data types in source DBMS to the corresponding data types in target DBMS or file format, see Data Type Mapping.

To quickly setup replication into HANA, see Quick Start for HVR - HANA.

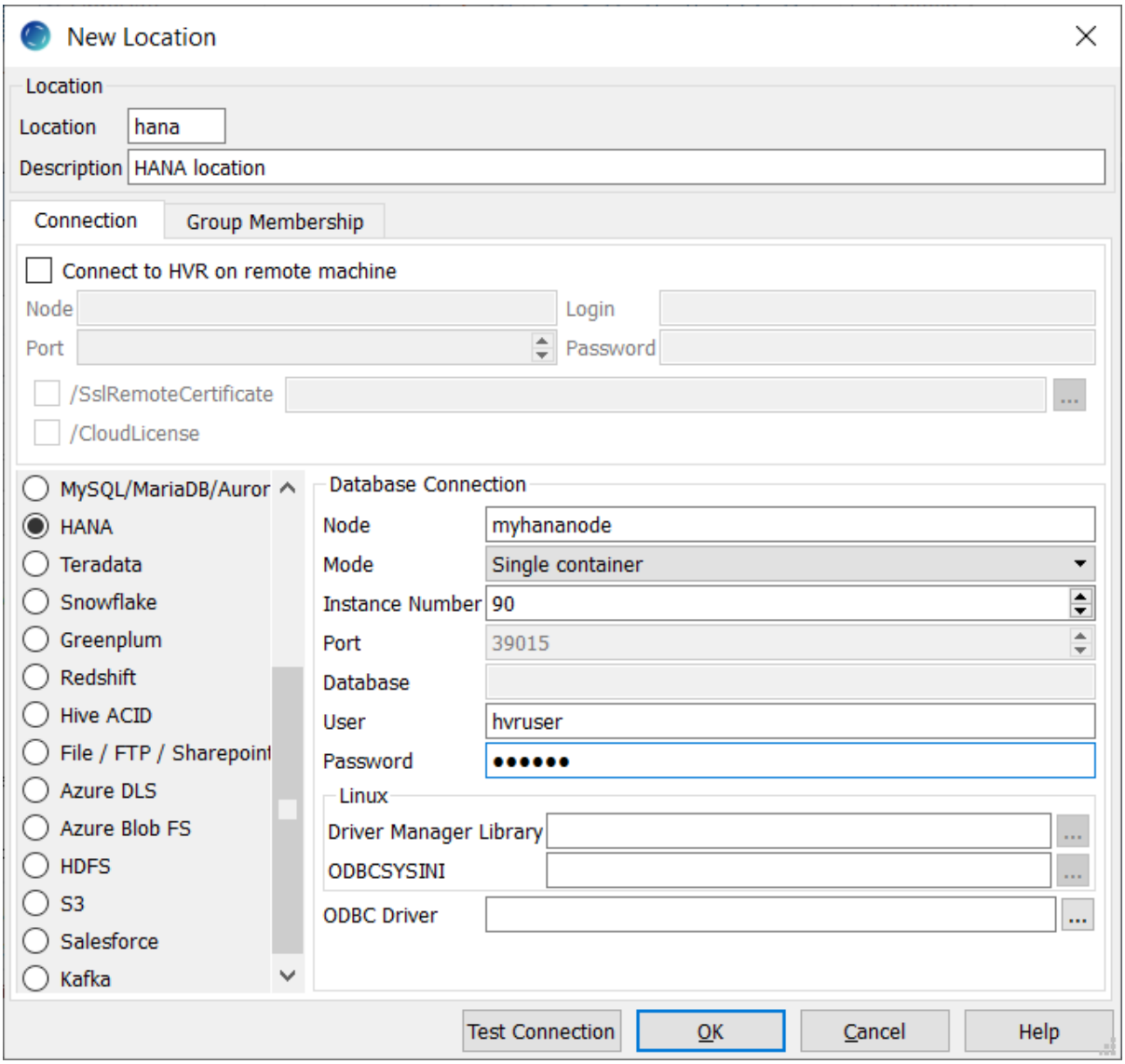
ODBC Connection

HVR requires that the HANA client (which contains the HANA ODBC driver) to be installed on the machine from which HVR connects to HANA. HVR uses HANA ODBC driver to connect, read and write data to HANA.

HVR does not support integrating changes captured from HANA into databases where the distribution key cannot be updated (e.g. Greenplum, Azure Synapse Analytics).

Location Connection

This section lists and describes the connection details required for creating HANA location in HVR.



Field	Description
Database Connection	
Node	The hostname or ip-address of the machine on which the HANA server is running. **Example:** myhananode
Mode	The mode for connecting HVR to HANA server. Available options: • **Single-container** • **Multiple containers - Tenant database** • **Multiple containers - System database** • **Manual port selection** - This option is used only if database **Port** needs to be specified manually.
Instance Number	The database instance number. **Example:** 90
Port	The port on which the HANA server is expecting connections. For more information about TCP/IP ports in HANA, refer to SAP Documentation **Example:** 39015
Database	The name of the specific database in a multiple-container environment. This field is enabled only if the **Mode** is either **Multiple containers - Tenant database** or **Manual port selection**.

User	The username to connect HVR to the HANA **Database**. **Example:** hvruser
Password	The password of the **User** to connect HVR to the HANA **Database**.
Linux	
Driver Manager Library	The directory path where the Unix ODBC Driver Manager Library is installed. For a default installation, the ODBC Driver Manager Library is available at **/usr/lib64** and does not need to specified. When UnixODBC is installed in for example **/opt /unixodbc-2.3.1** this would be **/opt/unixodbc-2.3.1/lib**
ODBCSYSINI	The directory path where **odbc.ini** and **odbcinst.ini** files are located. For a default installation, these files are available at **/etc** and does not need to be specified. When UnixODBC is installed in for example **/opt/unixodbc-2.3.1** this would be **/opt /unixodbc-2.3.1/etc**. The **odbcinst.ini** file should contain information about the HANA ODBC Driver under heading **[HDBODBC]** or **[HDBODBC32]**.
ODBC Driver	The user defined (installed) ODBC driver to connect HVR to the HANA database. If the user does not define the ODBC driver in the **ODBC Driver** field, then HVR will automatically load the correct driver for your current platform: **HDBODBC** (64-bit) or **H DBODBC32** (32bit).

Connecting to Remote HANA Location from Hub

HVR allows you to connect from a hub machine to a remote HANA database by using any of the following two methods:

1. Connect to an HVR installation running on the HANA database machine using HVR's protocol on a special TCP port number (e.g. 4343). This option must be used for log-based capture from HANA.
2. Use ODBC to connect directly to a HANA database remotely. In this case no additional software is required to be installed on the HANA database server itself. This option cannot be used for log-based capture from HANA database.

Capture

HVR only supports capture from HANA on Linux.

For the list of supported HANA versions, from which HVR can capture changes, see Capture changes from location in Capabilities.

Table Types

HVR supports capture from column-storage tables in HANA.

Grants for Capture

The following grants are required for capturing changes from HANA:

- To read from tables which are not owned by HVR User (using **TableProperties/Schema**), the **User** must be granted select privilege.

```
grant select on tbl to hvruser
```

- The **User** should also be granted select permission from some system table and views. In HANA, however, it is impossible to directly grant permissions on system objects to any user. Instead, special wrapper views should be created, and **User** should be granted read permission on this views. To do this:

1. Connect to the HANA database as user **SYSTEM**.
2. Create a schema called **_HVR**.

```
create schema _HVR;
```

3. Grant **SELECT** privilege on this schema to **User**.

```
grant select on schema _HVR to hvruser;
```

4. Execute the **hvrhanaviews.sql** script from the **$HVR_HOME/sql/hana** directory.
   This will create a set of views that HVR uses to read from system dictionaries on HANA.

Configuring Log Mode and Transaction Archive Retention Requirements

For HVR to capture changes from HANA, the automatic backup of transaction logs must be enabled in HANA. Normally HVR reads changes from the 'online' transaction log file, but if HVR is interrupted (say for 2 hours) then it must be able to read from transaction log backup files to capture the older changes. HVR is not interested in full backups; it only reads transaction log backup file.

To enable automatic log backup in HANA, the log mode must be set to **normal**. Once the log mode is changed from **overwrite** to **normal**, a full data backup must be created. For more information, search for Log Modes in SAP HANA Documentation.

The log mode can be changed using HANA Studio. For detailed steps, search for Change Log Modes in SAP HANA Documentation. Alternatively, you can execute the following SQL statement:

```
ALTER SYSTEM ALTER CONFIGURATION ('global.ini', 'SYSTEM') SET ('persistence',
'log_mode') = 'normal' WITH RECONFIGURE;
```

Transaction log (archive) retention: If a backup process has already moved these files to tape and deleted them, then HVR's capture will give an error and a refresh will have to be performed before replication can be restarted. The amount of 'retention' needed (in hours or days) depends on organization factors (how real-time must it be?) and practical issues (does a refresh take 1 hour or 24 hours?).

When performing log-shipping (**Capture /ArchiveLogOnly**), file names must not be changed in the process because begin-sequence and timestamp are encoded in the file name and capture uses them.

Archive Log Only Method

The **Archive Log Only** mode can be used for capturing changes from HANA. This allows the HVR process to reside on machine other than that on which HANA DBMS resides and read changes from backup transaction log files that may be sent to it by some file transfer mechanism.

HVR must be configured to find these files by defining action **Capture** with parameters **/ArchiveLogOnly**, **/ArchiveLogPath**, and **/ArchiveLogFormat** (optional).

The **Archive Log Only** method will generally expose higher latency than **non-Archive Log Only** method because changes can only be captured when the transaction log backup file is created. The **Archive Log Only** method enables high-performance log-based capture with minimal OS privileges, at the cost of higher capture latency.

OS Level Permissions or Requirements

To capture from HANA database, HVR should be installed on the HANA database server itself, and HVR remote listener should be configured to accept remote connections. The Operating System (OS) user the HVR is running under should have READ permission on the HANA database files. This can typically be achieved by adding this user to the **sapsys** user group.

Channel Setup Requirements

It is not possible to enable 'supplemental logging' on HANA. This means that the real key values are not generally available to HVR during capture. A workaround for this limitation is capturing the Row ID values and use them as a surrogate replication key.

The following two additional actions should be defined to the channel, prior to using **Table Explore** (to add tables to the channel), to instruct HVR to capture Row ID values and to use them as surrogate replication keys.

Location	Action	Annotation
Source	**ColumnProperties /Name=hvr_rowid /CaptureFromRowId**	This action should be defined for capture locations only.
*	**ColumnProperties /Name=hvr_rowid /SurrogateKey**	This action should be defined for both capture and integrate locations

Integrate and Refresh Target

HVR supports integrating changes into HANA location. This section describes the configuration requirements for integrating changes (using **Integrate** and **refresh**) into HANA location. For the list of supported HANA versions, into which HVR can integrate changes, see Integrate changes into location in Capabilities.

HVR uses HANA ODBC driver to write data to HANA during continuous **Integrate** and row-wise **Refresh**. For the method used during **Integrate** with **/Burts** and Bulk **Refresh,** see section 'Burst Integarte and Bulk Refresh' below.

Grants for Integrate and Refresh Target

- The **User** should have permission to read and change replicated tables

```
grant select, insert, update, delete on tbl to hvruser
```

- The **User** should have permission to create and alter tables in the target schema

```
grant create any, alter on schema schema to hvruser
```

- The **User** should have permission to create and drop HVR state tables

Burst Integrate and Bulk Refresh

While **HVR Integrate** is running with parameter **/Burst** and Bulk **Refresh**, HVR can stream data into a target database straight over the network into a bulk loading interface specific for each DBMS (e.g. direct-path-load in Oracle), or else HVR puts data into a temporary directory (‘staging file') before loading data into a target database.

For best performance, HVR performs **Integrate** with **/Burst** and Bulk **Refresh** on HANA location using staging files.

HVR implements **Integrate** with **/Burst** and Bulk **Refresh** (with file staging) into HANA as follows:

1. HVR first stages data into a local staging file (defined in **/StagingDirectoryHvr**)
2. HVR then uses SAP HANA SQL command '**import**' to ingest the data into SAP HANA target tables from the staging directory (defined in **/StagingDirectoryDb**).

To perform **Integrate** with parameter **/Burst** and Bulk **Refresh**, define action **LocationProperties** on HANA location with the following parameters:

- **/StagingDirectoryHvr**: the location where HVR will create the temporary staging files.
- **/StagingDirectoryDb**: the location from where HANA will access the temporary staging files. This staging-path should be configured in HANA for importing data,

```
alter system alter configuration ('indexserver.ini', 'system')
set ('import_export', 'csv_import_path_filter') = 'STAGING-PATH' with reconfigure
```

Grants for Burst Integrate and Bulk Refresh

The **User** should have permission to import data:

```
grant import to hvruser
```

Compare and Refresh Source

HVR supports **compare** and **refresh** in HANA location. This section describes the configuration requirements for performing **compare** and **refresh** in HANA (source) location.

Grants for Compare and Refresh Source

The **User** should have permission to read replicated tables

```
grant select on tbl to hvruser
```

Requirements for HDFS

Contents

This section describes the requirements, access privileges, and other features of HVR when using Hadoop Distributed File System (HDFS) for replication. HVR supports the WebHDFS API for reading and writing files from and to HDFS. For information about compatibility and supported versions of HDFS with HVR platforms, see Platform Compatibility Matrix.

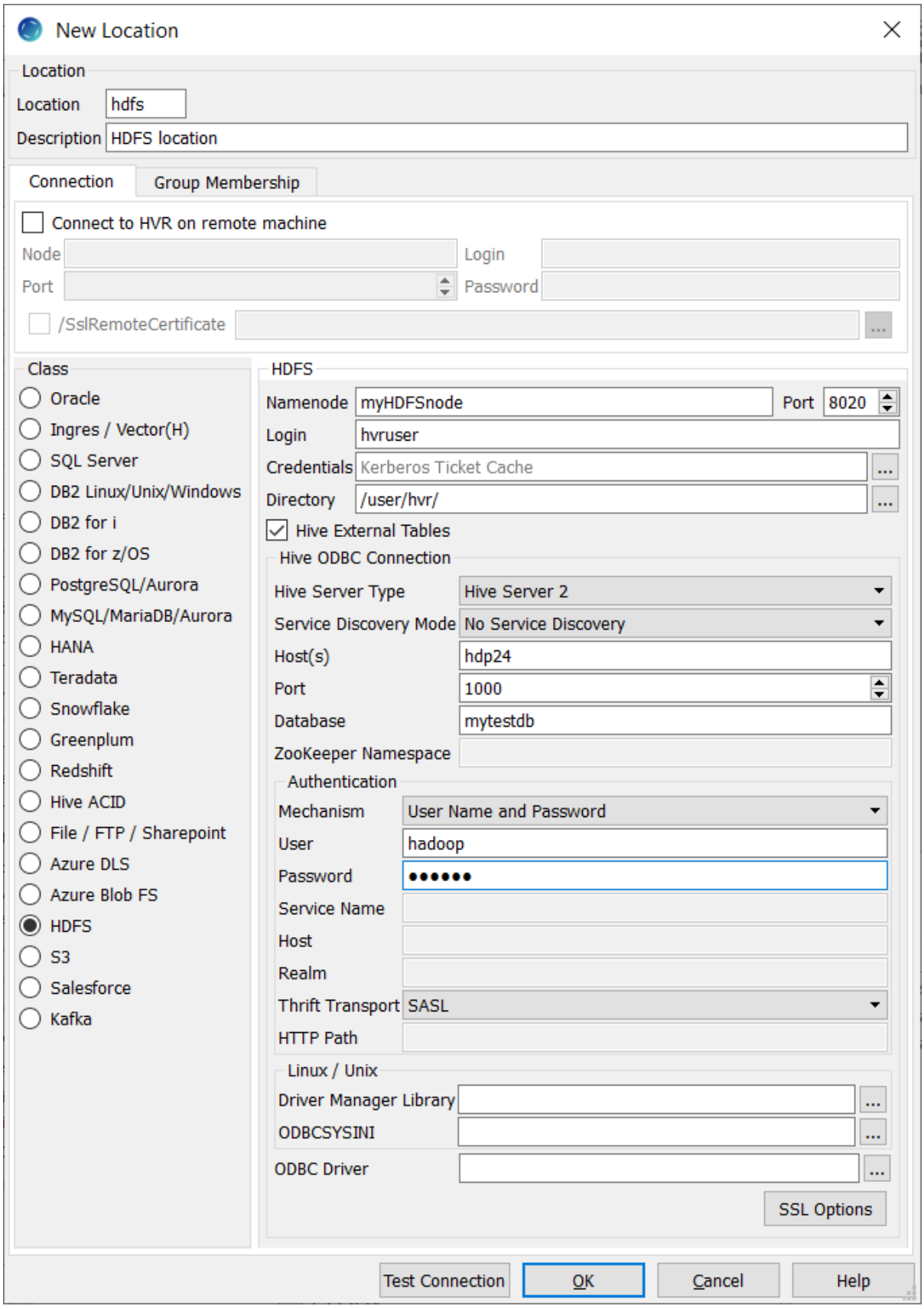
For the capabilities supported by HVR, see Capabilities.

To quickly setup replication into HDFS, see Quick Start for HVR - HDFS.

For requirements, access privileges, and other features of HVR when using MapR for replication, see Requirements for MapR.

Location Connection

This section lists and describes the connection details required for creating HDFS location in HVR.



Field	Description
Database Connection	
Namenode	The hostname of the HDFS NameNode. **Example:** myHDFSnode

Port	The port on which the HDFS server (**Namenode**) is expecting connections. **Example:** 8020
Login	The username to connect HVR to the HDFS **Namenode**. **Example:** hvruser
Credentials	The credential (Kerberos Ticket Cache file) for the username specified in **Login** to connect HVR to the HDFS **Namenode**. This field should be left blank to use a **keytab** file for authentication or if Kerberos is not used on the hadoop cluster. For more information about using Kerberos authentication, see HDFS Authentication and Kerberos.
Directory	The directory path in the HDFS **Namenode** to be used for replication. **Example:** /user/hvr/
Hive External Tables	Enable/Disable Hive ODBC connection configuration for creating Hive external tables above HDFS.

Hive ODBC Connection

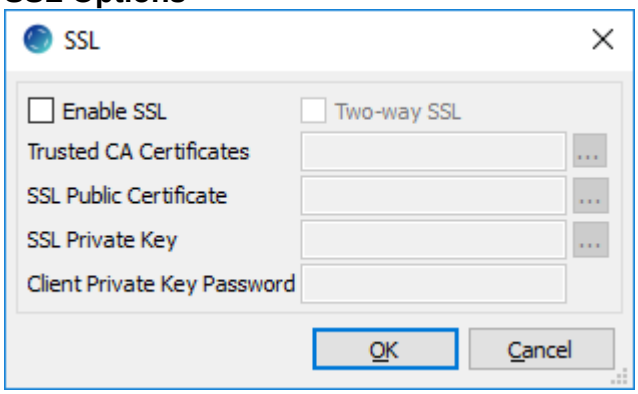
HVR uses the Hive ODBC driver to connect to Hive for creating Hive external tables above HDFS to perform hvrcompare with files that reside on the HDFS location. You can enable/disable the Hive configuration for HDFS in the **New Location** dialog using the **Hive External Tables** field.

Field	Description
Hive ODBC Connection	
Hive Server Type	The type of Hive server. Available options: • **Hive Server 1** (default): The driver connects to a Hive Server 1 instance. • **Hive Server 2**: The driver connects to a Hive Server 2 instance.
Service Discovery Mode	The mode for connecting to Hive. This field is enabled only if **Hive Server Type** is **Hive Server 2**. Available options: • **No Service Discovery** (default): The driver connects to Hive server without using the ZooKeeper service. • **ZooKeeper**: The driver discovers Hive Server 2 services using the ZooKeeper service.
Host(s)	The hostname or IP address of the Hive server. When Service Discovery Mode is ZooKeeper, specify the list of ZooKeeper servers in following format **[ZK_Host1]:[ZK_Port1],[ZK_Host2]:[ZK_Port2]**, where **[ZK_Host]** is the IP address or hostname of the ZooKeeper server and **[ZK_Port]** is the TCP port that the ZooKeeper server uses to listen for client connections. **Example:** hive-host
Port	The TCP port that the Hive server uses to listen for client connections. This field is enabled only if **Service Discovery Mode** is **No Service Discovery**. **Example:** 10000
Database	The name of the database schema to use when a schema is not explicitly specified in a query. **Example:** mytestdb
ZooKeeper Namespace	The namespace on ZooKeeper under which Hive Server 2 nodes are added. This field is enabled only if **Service Discovery Mode** is **ZooKeeper**.
Authentication	

Mechanism	The authentication mode for connecting HVR to **Hive Server 2**. This field is enabled only if **Hive Server Type** is **Hive Server 2**. Available options: • **No Authentication** (default) • **User Name** • **User Name and Password** • **Kerberos** • **Windows Azure HDInsight Service** **Since** v5.5.0/2
User	The username to connect HVR to Hive server. This field is enabled only if **Mechanism** is **User Name** or **User Name and Password**. **Example:** dbuser
Password	The password of the **User** to connect HVR to Hive server. This field is enabled only if **Mechanism** is **User Name and Password**.
Service Name	The Kerberos service principal name of the Hive server. This field is enabled only if **Mechanism** is **Kerberos**.
Host	The Fully Qualified Domain Name (FQDN) of the Hive Server 2 host. The value of **Host** can be set as **_HOST** to use the Hive server hostname as the domain name for Kerberos authentication. If **Service Discovery Mode** is disabled, then the driver uses the value specified in the Host connection attribute. If **Service Discovery Mode** is enabled, then the driver uses the **Hive Server 2** host name returned by ZooKeeper. This field is enabled only if **Mechanism** is **Kerberos**.
Realm	The realm of the Hive Server 2 host. It is not required to specify any value in this field if the realm of the Hive Server 2 host is defined as the default realm in Kerberos configuration. This field is enabled only if **Mechanism** is **Kerberos**.
Thrift Transport **Since** v5.5.0/2	The transport protocol to use in the Thrift layer. This field is enabled only if **Hive Server Type** is **Hive Server 2**. Available options: • **Binary** (This option is available only if **Mechanism** is **No Authentication** or **User Name and Password**.) • **SASL** (This option is available only if **Mechanism** is **User Name** or **User Name and Password** or **Kerberos**.) • **HTTP** (This option is not available if **Mechanism** is **User Name**.) For information about determining which Thrift transport protocols your Hive server supports, refer to HiveServer2 Overview and Setting Up HiveServer2 sections in Hive documentation.
HTTP Path **Since** v5.5.0/2	The partial URL corresponding to the Hive server. This field is enabled only if **Thrift Transport** is **HTTP**.
Linux / Unix	
Driver Manager Library	The directory path where the Unix ODBC Driver Manager Library is installed. **Example:** /opt/unixodbc-2.3.2/lib
ODBCSYSINI	The directory path where **odbc.ini** and **odbcinst.ini** files are located. **Example:** /opt/unixodbc-2.3.2/etc

ODBC Driver	The user defined (installed) ODBC driver to connect HVR to the Hive server.
SSL Options	Show **SSL Options**.

SSL Options



Field	Description
Enable SSL	Enable/disable (one way) SSL. If enabled, HVR authenticates the Hive server by validating the SSL certificate shared by the Hive server.
Two-way SSL	Enable/disable two way SSL. If enabled, both HVR and Hive server authenticate each other by validating each others SSL certificate. This field is enabled only if **Enable SSL** is selected.
Trusted CA Certificates	The directory path where the **.pem** file containing the server's public SSL certificate signed by a trusted CA is located. This field is enabled only if **Enable SSL** is selected.
SSL Public Certificate	The directory path where the **.pem** file containing the client's SSL public certificate is located. This field is enabled only if **Two-way SSL** is selected.
SSL Private Key	The directory path where the **.pem** file containing the client's SSL private key is located. This field is enabled only if **Two-way SSL** is selected.
Client Private Key Password	The password of the private key file that is specified in **SSL Private Key**. This field is enabled only if **Two-way SSL** is selected.

Hadoop Client

HDFS locations can only be accessed through HVR running on Linux or Windows, and it is not required to run HVR installed on the Hadoop **Namenode** although it is possible to do so. The Hadoop client should be present on the server from which HVR will access the HDFS. HVR uses HDFS compatible libhdfs API to connect, read and write data to HDFS during capture, integrate (continuous), refresh (bulk) and compare (direct file compare). For more information about installing Hadoop client, refer to Apache Hadoop Releases.

Hadoop Client Configuration

The following are required on the server from which HVR connects to HDFS:

- Install Hadoop 2.4.1 or later versions along with Java Runtime Environment:
  - Hadoop versions below 3.0 require JRE 7 or 8
  - Hadoop version 3.0 and higher requires only JRE 8

- Set the environment variable **$JAVA_HOME** to the Java installation directory. Ensure that this is the directory that has a bin folder, e.g. if the Java bin directory is d:\java\bin, **$JAVA_HOME** should point to d:\java.
- Set the environment variable **$HADOOP_COMMON_HOME** or **$HADOOP_HOME** or **$HADOOP_PREFIX** to the Hadoop installation directory, or the **hadoop** command line client should be available in the path.

> Since the binary distribution available in Hadoop website lacks Windows-specific executables, a warning about unable to locate **winutils.exe** is displayed. This warning can be ignored for using Hadoop library for client operations to connect to a HDFS server using HVR. However, the performance on integrate location would be poor due to this warning, so it is recommended to use a Windows-specific Hadoop distribution to avoid this warning. For more information about this warning, refer to Hadoop Wiki and Hadoop issue HADOOP-10051.

Verifying Hadoop Client Installation

To verify the Hadoop client installation,

1. The **HADOOP_HOME/bin** directory in Hadoop installation location should contain the hadoop executables in it.
2. Execute the following commands to verify Hadoop client installation:

```
$JAVA_HOME/bin/java -version
$HADOOP_HOME/bin/hadoop version
$HADOOP_HOME/bin/hadoop classpath
```

3. If the Hadoop client installation is verified successfully then execute the following command to verify the connectivity between HVR and HDFS:

```
$HADOOP_HOME/bin/hadoop fs -ls hdfs://cluster/
```

Client Configuration Files

Client configuration files are required if Kerberos authentication is used in the Hadoop cluster or else they can be useful for debugging. Client configuration files contain settings for different services like HDFS, and others. If the HVR integrate server is not part of the cluster, it is recommended to download the configuration files for the cluster so that the Hadoop client knows how to connect to HDFS.

The client configuration files for Cloudera Manager or Ambari for Hortonworks can be downloaded from the respective cluster manager's web interface. For more information about downloading client configuration files, search for "Client Configuration Files" in the respective documentation for Cloudera and Hortonworks.

Hive External Table

HVR allows you to create Hive external tables above HDFS files which are only used during compare. The Hive ODBC connection can be enabled for HDFS in the location creation screen by selecting the **Hive External Tables** field (see section Location Connection).

ODBC Connection

HVR uses ODBC connection to the Hadoop cluster for which it requires the ODBC driver (Amazon ODBC 1.1.1 or HortonWorks ODBC 2.1.2 and above) for Hive installed on the server (or in the same network).

The Amazon and HortonWorks ODBC drivers are similar and compatible to work with Hive 2.x release. However, it is recommended to use the Amazon ODBC driver for Amazon Hive and the Hortonworks ODBC driver for HortonWorks Hive.

By default, HVR uses Amazon ODBC driver for connecting to Hadoop. To use the Hortonworks ODBC driver the following action definition is required:

For Linux

Group	Table	Action
HDFS	*	Environment **/Name=HVR_ODBC_CONNECT_STRING_DRIVER /Value=Hortonworks Hive ODBC Driver 64-bit**

For Windows

Group	Table	Action
HDFS	*	Environment **/Name=HVR_ODBC_CONNECT_STRING_DRIVER /Value=Hortonworks Hive ODBC Driver**

Channel Configuration

For the file formats (CSV, JSON, and AVRO) the following action definitions are required to handle certain limitations of the Hive deserialization implementation during Bulk or Row-wise **Compare**:

- For CSV

Group	Table	Action
HDFS	*	**FileFormat /NullRepresentation=\\N**
HDFS	*	**TableProperties /CharacterMapping="\x00>\\0;\n>\\n;\r>\\r;">\""**
HDFS	*	**TableProperties /MapBinary=BASE64**

- For JSON

Group	Table	Action
HDFS	*	**TableProperties /MapBinary=BASE64**
HDFS	*	**FileFormat /JsonMode=ROW_FRAGMENTS**

- For Avro

Group	Table	Action
HDFS	*	**FileFormat /AvroVersion=v1_8**

v1_8 is the default value for **FileFormat /AvroVersion**, so it is not mandatory to define this action.

HDFS Authentication and Kerberos

Contents

HVR supports connecting to both insecure and Kerberos-secured HDFS clusters. Information on setting up Hadoop for HVR can be found in Requirements for HDFS.

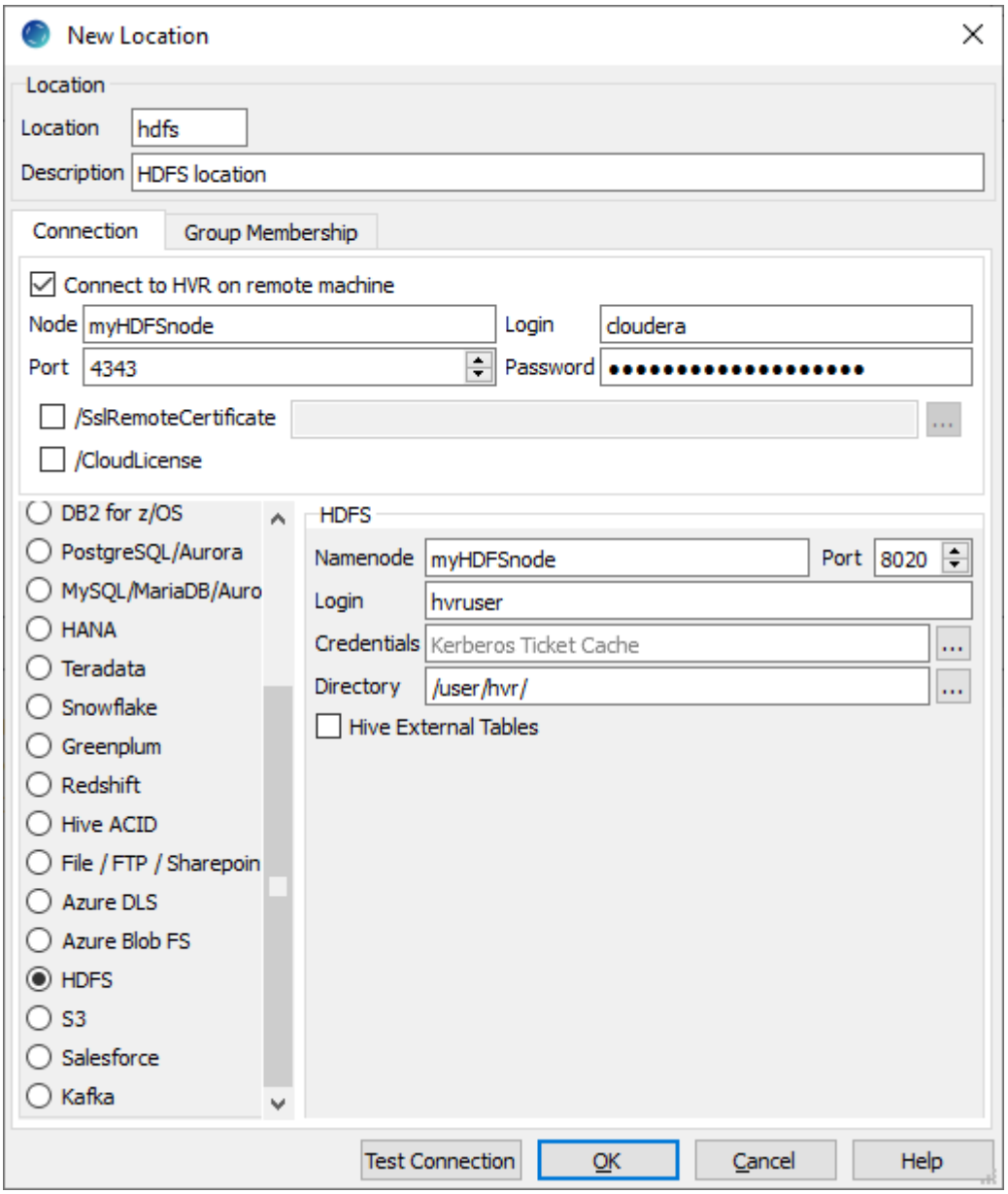
Insecure HDFS clusters are protected at network level by restricting which hosts can establish a connection. Any software that is able to establish a connection can claim to act as any user on HDFS system.

Secure HDFS clusters are protected by Kerberos authentication. Both HDFS servers (Hadoop **NameNode**, Hadoop **DataNode**) and HDFS clients (HVR) authenticate themselves against a central Kerberos server which grants them a ticket. Client and server exchange their tickets, and both verify each other's identity. HVR must have access to cluster configuration files (in **$HVR_HADOOP_HOME**) in order to verify **NameNode**'s and **DataNode**'s Kerberos identities.

Insecure Clusters

Insecure clusters require only a HDFS username. Enter this user in **Login** field of the location dialog. This username will be used in HDFS file permissions, such as any files created by HVR in HDFS will be owned by this user.

If the **Login** field is empty, HVR's operating system username will be used (If HVR is running on a remote location, remote operating system username will be used).



Kerberized Clusters

Accessing the kerberized clusters require authentication against a Kerberos server which can be achieved in the following two ways :

- Accessing Kerberized Clusters with Ticket Cache File
- Accessing Kerberized Clusters with Keytab File

To access the kerberized clusters, HVR requires the following:

- If HVR is installed on a separate server (outside the Hadoop cluster),

  a. Kerberos should be installed and configured on the server where HVR is installed. The configuration should be same as that of the hadoop cluster's configuration.
  b. Configure the Hadoop client. For more information, see section Hadoop Client in Requirements for HDFS.
  c. Using the cluster manager's web interface, Hadoop client configuration files should be downloaded from the Hadoop cluster to the server where HVR is installed.

  For more information, click here...

Client Configuration Files

Client configuration files are required if Kerberos authentication is used in the Hadoop cluster or else they can be useful for debugging. Client configuration files contain settings for different services like HDFS, and others. If the HVR integrate server is not part of the cluster, it is recommended to download the configuration files for the cluster so that the Hadoop client knows how to connect to HDFS.

The client configuration files for Cloudera Manager or Ambari for Hortonworks can be downloaded from the respective cluster manager's web interface. For more information about downloading client configuration files, search for "Client Configuration Files" in the respective documentation for Cloudera and Hortonworks.

- If HVR is installed on a Hadoop edge node, the steps mentioned above are not required for HVR to connect to the kerberized cluster.
- If the Kerberos server issues tickets with strong encryption keys then install Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files onto the JRE installation that HVR uses. JCE can be downloaded from Oracle website.

To verify the connectivity between HVR and HDFS, execute the following command on the server where HVR is installed :

```
$HADOOP_HOME/bin/hadoop fs -ls hdfs://cluster/
```

Accessing Kerberized Clusters with Ticket Cache File

A short-term ticket is issued by authenticating against Kerberos server with **kinit** command. This ticket usually expires in a timeframe of a few hours or a few days depending on the Kerberos configuration. An interactive operating system shell connection will renew Kerberos ticket as required, but this would not automatically happen on a non-interactive HVR setup.

- Command **kinit** can be used to obtain or renew a Kerberos ticket-granting ticket. For more information about this command, refer to MIT Kerberos Documentation.
- Command **klist** lists the contents of the default Ticket Cache file, also showing the default filename. For more information about this command, refer to MIT Kerberos Documentation.

By default, HVR is configured for the path of the Kerberos Ticket Cache file, and assumes tickets will be renewed by the user as needed. HVR will pick up any changes made to the Ticket Cache file automatically. It is user's responsibility to set up periodic renewal jobs to update the file before Kerberos tickets expire.

The Ticket Cache file must be located on the HVR remote location if HVR is running on a remote machine. The file must have correct file system permissions for HVR process to read.

To use a Ticket Cache file with HVR, enter the Kerberos principal's user part to **Login** field and full path of the Ticket Cache file to **Credentials** field in the location dialog.

- **Alternative configuration 1**
  Leave **Credentials** field empty, and configure your channel with:
  Environment /Name=HVR_HDFS_KRB_TICKETCACHE /Value=*ticketcache.file*
- **Alternative configuration 2**
  Leave **Login** and **Credentials** field empty, and configure your channel with:
  Environment /Name=HVR_HDFS_KRB_PRINCIPAL /Value=*full_principal_name (e.g. username@REALM)*
  Environment /Name=HVR_HDFS_KRB_TICKETCACHE /Value=*ticketcache.file*

Accessing Kerberized Clusters with Keytab File

Non-interactive daemons can also authenticate to the Kerberos server using a **Keytab** file. A **keytab** file holds a list of entries, consisting of Kerberos principal name, key version, encryption type, and an encryption key. Encryption key is generated with the password provided at creation time. It is crucial that an entry is added to this file for each possible encryption type your Kerberos server might ask for. It is not possible for a Kerberos client to respond to an encryption type not found in the file.

The **keytab** files can be copied across computers, they are not bound to the host they were created on. The **keytab** file is only used to acquire a real ticket from the Kerberos server when needed. If the file is compromised, it can be revoked from the Kerberos server by changing password or key version.

Keytab files are created using the **ktutil** command. Depending on your system Kerberos package, usage will vary. For more information about this command, refer to MIT Kerberos Documentation.

The **keytab** file must be located on the HVR remote location if HVR is running on a remote machine. The file must have correct file system permissions for HVR process to read.

To use a **keytab** file with HVR, leave **Login** and **Credentials** fields of the location dialog blank and configure your channel with:

> **Environment /Name=HVR_HDFS_KRB_PRINCIPAL /Value=***full_principal_name (e.g. username@REALM)*
>
> **Environment /Name=HVR_HDFS_KRB_KEYTAB /Value=***keytab.file*

Accessing Kerberized Clusters with HDFS Impersonation

In most cases, your HDFS cluster will be configured to map the username part of your Kerberos principal as the HDFS username (*"username"* in principal *"username@KERBEROS.REALM"*). If you need to use a different HDFS username than your Kerberos principal, and your cluster is configured to allow this setup, then you can configure HVR in the following way.

- To use Ticket Cache with HDFS impersonation, set your HDFS impersonate username in the **Login** entry of the Location dialog, leave **Credentials** entry blank, and define **$HVR_HDFS_KRB_PRINCIPAL** and **$HVR_HDFS_KRB_TICKETCACHE** environment actions as described in the **Accessing Kerberized Clusters with Ticket Cache File** section above.
- To use **keytab** with HDFS impersonation, set your HDFS impersonate username in the **Login** entry of the Location dialog, leave **Credentials** entry blank, and define **$HVR_HDFS_KRB_PRINCIPAL** and **$HVR_HDFS_KRB_KEYTAB** environment actions as described in the **Accessing Kerberized Clusters with Keytab File** section above.

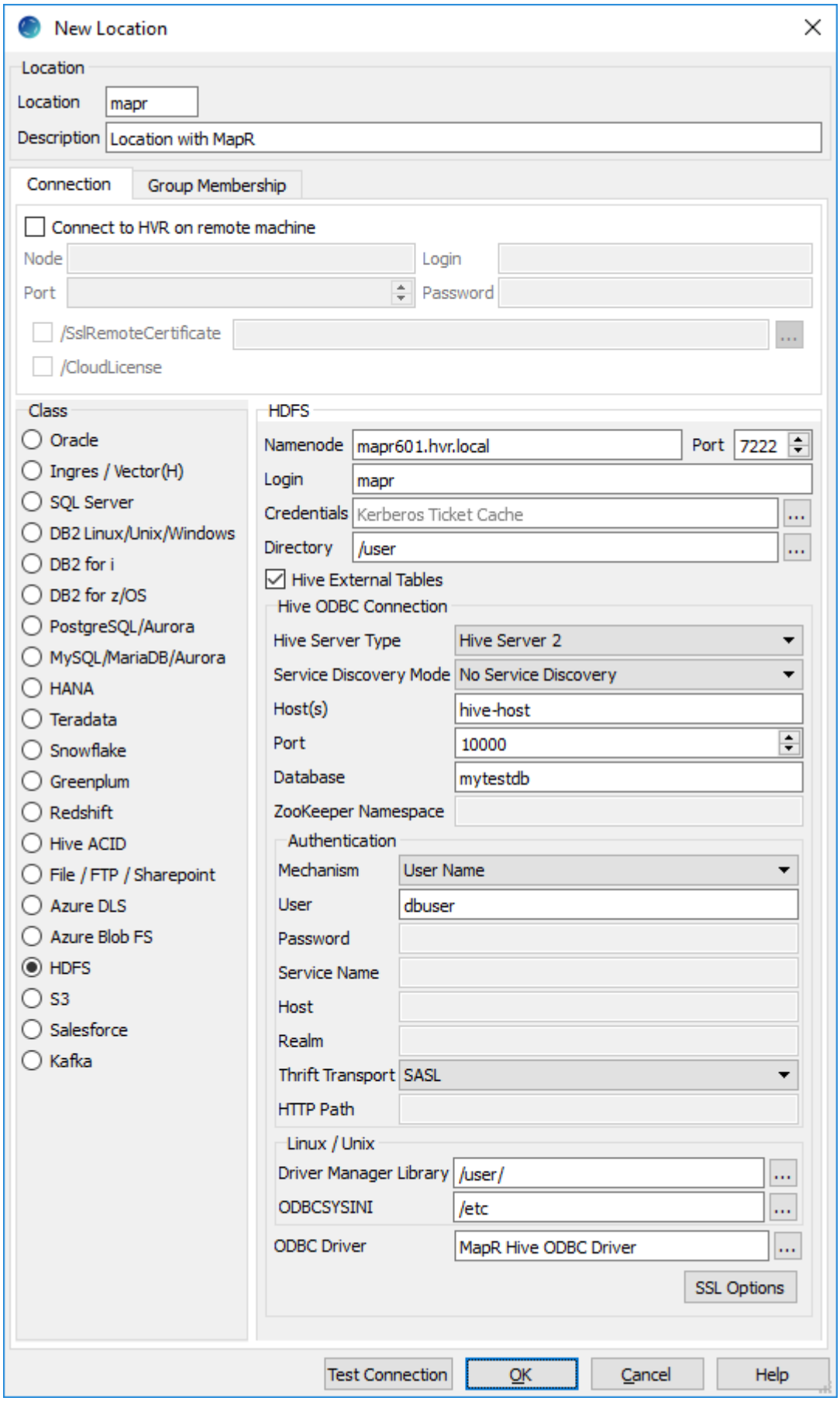
Requirements for MapR

Contents

This section describes the requirements, access privileges, and other features of HVR when using MapR for replication. HVR supports the WebHDFS API for reading and writing files from and to MapR.

Location Connection

This section lists and describes the connection details required for creating MapR location in HVR.



Field	Description
Database Connection	

Namenode	The hostname of the MapR Container Location Database (CLDB). **Example:** mapr601.hvr.local
Port	The port on which the MapR CLDB (**Namenode**) is expecting connections. The default port is **7222**. **Example:** 7222
Login	The username to connect HVR to the MapR CLDB (**Namenode**). The username can be either the MapR user or if impersonation is used then the username is the *remotelistener OS user*. **Example:** hvruser
Credentials	The credential (Kerberos Ticket Cache file) for the **Login** to connect HVR to the MapR CLDB (**Namenode**). This field should be left blank to use a **keytab** file for authentication or if Kerberos is not used on the MapR cluster. For more details, see HDFS Authentication and Kerberos.
Directory	The directory path in the MapR CLDB (**Namenode**) to be used for replication. **Example:** /user
Hive External Tables	Enable/Disable Hive ODBC connection configuration for creating Hive external tables above HDFS.

Hive ODBC Connection

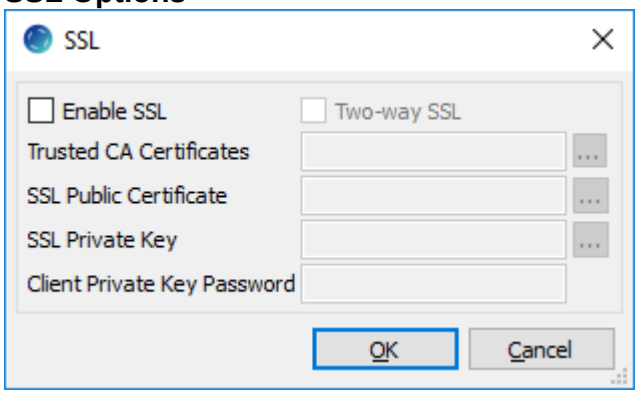
HVR allows you to create Hive External Tables above HDFS which are only used during compare. You can enable /disable the Hive configuration for HDFS in location creation screen using the **Hive External Tables** field.

Field	Description
Hive ODBC Connection	
Hive Server Type	The type of Hive server. Available options: • **Hive Server 1** (default): The driver connects to a Hive Server 1 instance. • **Hive Server 2**: The driver connects to a Hive Server 2 instance.
Service Discovery Mode	The mode for connecting to Hive. This field is enabled only if **Hive Server Type** is **Hive Server 2**. Available options: • **No Service Discovery** (default): The driver connects to Hive server without using the ZooKeeper service. • **ZooKeeper**: The driver discovers Hive Server 2 services using the ZooKeeper service.
Host(s)	The hostname or IP address of the Hive server. When Service Discovery Mode is ZooKeeper, specify the list of ZooKeeper servers in following format **[ZK_Host1]:[ZK_Port1],[ZK_Host2]:[ZK_Port2]**, where **[ZK_Host]** is the IP address or hostname of the ZooKeeper server and **[ZK_Port]** is the TCP port that the ZooKeeper server uses to listen for client connections. **Example:** hive-host
Port	The TCP port that the Hive server uses to listen for client connections. This field is enabled only if **Service Discovery Mode** is **No Service Discovery**. **Example:** 10000
Database	The name of the database schema to use when a schema is not explicitly specified in a query. **Example:** mytestdb
ZooKeeper Namespace	The namespace on ZooKeeper under which Hive Server 2 nodes are added. This field is enabled only if **Service Discovery Mode** is **ZooKeeper**.

Authentication	
Mechanism	The authentication mode for connecting HVR to **Hive Server 2**. This field is enabled only if **Hive Server Type** is **Hive Server 2**. Available options: • **No Authentication** (default) • **User Name** • **User Name and Password** • **Kerberos** • **Windows Azure HDInsight Service** **Since** v5.5.0/2
User	The username to connect HVR to Hive server. This field is enabled only if **Mechanism** is **User Name** or **User Name and Password**. **Example:** dbuser
Password	The password of the **User** to connect HVR to Hive server. This field is enabled only if **Mechanism** is **User Name and Password**.
Service Name	The Kerberos service principal name of the Hive server. This field is enabled only if **Mechanism** is **Kerberos**.
Host	The Fully Qualified Domain Name (FQDN) of the Hive Server 2 host. The value of **Host** can be set as **_HOST** to use the Hive server hostname as the domain name for Kerberos authentication. If **Service Discovery Mode** is disabled, then the driver uses the value specified in the Host connection attribute. If **Service Discovery Mode** is enabled, then the driver uses the **Hive Server 2** host name returned by ZooKeeper. This field is enabled only if **Mechanism** is **Kerberos**.
Realm	The realm of the Hive Server 2 host. It is not required to specify any value in this field if the realm of the Hive Server 2 host is defined as the default realm in Kerberos configuration. This field is enabled only if **Mechanism** is **Kerberos**.
Thrift Transport **Since** v5.5.0/2	The transport protocol to use in the Thrift layer. This field is enabled only if **Hive Server Type** is **Hive Server 2**. Available options: • **Binary** (This option is available only if **Mechanism** is **No Authentication** or **User Name and Password**.) • **SASL** (This option is available only if **Mechanism** is **User Name** or **User Name and Password** or **Kerberos**.) • **HTTP** (This option is not available if **Mechanism** is **User Name**.) For information about determining which Thrift transport protocols your Hive server supports, refer to HiveServer2 Overview and Setting Up HiveServer2 sections in Hive documentation.
HTTP Path **Since** v5.5.0/2	The partial URL corresponding to the Hive server. This field is enabled only if **Thrift Transport** is **HTTP**.
Linux / Unix	
Driver Manager Library	The directory path where the Unix ODBC Driver Manager Library is installed. **Example:** /opt/unixodbc-2.3.2/lib

ODBCSYSINI	The directory path where **odbc.ini** and **odbcinst.ini** files are located. **Example:** /opt/unixodbc-2.3.2/etc
ODBC Driver	The user defined (installed) ODBC driver to connect HVR to the Hive server.
SSL Options	Show **SSL Options**.

SSL Options



Field	Description
Enable SSL	Enable/disable (one way) SSL. If enabled, HVR authenticates the Hive server by validating the SSL certificate shared by the Hive server.
Two-way SSL	Enable/disable two way SSL. If enabled, both HVR and Hive server authenticate each other by validating each others SSL certificate. This field is enabled only if **Enable SSL** is selected.
Trusted CA Certificates	The directory path where the **.pem** file containing the server's public SSL certificate signed by a trusted CA is located. This field is enabled only if **Enable SSL** is selected.
SSL Public Certificate	The directory path where the **.pem** file containing the client's SSL public certificate is located. This field is enabled only if **Two-way SSL** is selected.
SSL Private Key	The directory path where the **.pem** file containing the client's SSL private key is located. This field is enabled only if **Two-way SSL** is selected.
Client Private Key Password	The password of the private key file that is specified in **SSL Private Key**. This field is enabled only if **Two-way SSL** is selected.

MapR Client

MapR locations can only be accessed through HVR running on Linux or Windows, and it is not required to run HVR installed on the **Namenode** although it is possible to do so. The MapR client should be present on the server from which HVR will access the MapR (**Namenode**). For more information about installing MapR client, refer to MapR Documentation.

MapR Client Configuration

The following are required on the server from which HVR connects to MapR:

- Install MapR Client
- Install Java Development Kit (JDK), version 1.7 or later
- Install Java Runtime Environment (JRE), version 7 or later
- Set the environment variable **$JAVA_HOME** to the Java installation directory. Ensure that this is the directory that has a bin folder, e.g. if the Java bin directory is d:\java\bin, **$JAVA_HOME** should point to d:\java.

- Set the environment variable **$MAPR_HOME** to the MapR installation directory, or the **hadoop** command line client should be available in the path.

> Since the binary distribution available in Hadoop website lacks Windows-specific executables, a warning about unable to locate **winutils.exe** is displayed. This warning can be ignored for using Hadoop library for client operations to connect to a HDFS server using HVR. However, the performance on integrate location would be poor due to this warning, so it is recommended to use a Windows-specific Hadoop distribution to avoid this warning. For more information about this warning, refer to Hadoop issue HADOOP-10051.

Verifying MapR Client Installation

To verify the MapR client installation,

1. The **MAPR_HOME/bin** directory in MapR installation location should contain the MapR executables in it.
2. Execute the following commands to verify MapR client installation:

```
$JAVA_HOME/bin/java -version
$MAPR_HOME/bin/hadoop version
$MAPR_HOME/bin/hadoop classpath
```

3. If the MapR client installation is verified successfully then execute the following command to verify the connectivity between HVR and MapR:

```
$MAPR_HOME/bin/hadoop fs -ls hdfs://cluster/
```

Client Configuration Files

Client configuration files are not required for HVR to perform replication, however, they can be useful for debugging. Client configuration files contain settings for different services like HDFS, and others. If the HVR integrate server is not part of the cluster, it is recommended to download the configuration files for the cluster so that the MapR client knows how to connect to HDFS.

Hive External Table

HVR allows you to create Hive external tables above HDFS files which are only used during compare. The Hive ODBC connection can be enabled for MapR in the location creation screen by selecting the **Hive External Tables** field (see section Location Connection).

ODBC Connection

HVR uses ODBC connection to the MapR cluster for which it requires the MapR ODBC driver for Hive installed on the server (or in the same network). For more information about using ODBC to connect to HiveServer 2, refer to MapR Documentation.

Channel Configuration

For the file formats (CSV and AVRO) the following action definitions are required to handle certain limitations of the Hive deserialization implementation during Bulk or Row-wise **Compare**:

- For CSV,

Group	Table	Action
HDFS	*	**FileFormat /NullRepresentation=\\N**
HDFS	*	**TableProperties /CharacterMapping="\x00>\\0;\n>\\n;\r>\\r;">\""**
HDFS	*	**TableProperties /MapBinary=BASE64**

- For Avro,

Group	Table	Action
HDFS	*	**FileFormat /AvroVersion=v1_8**

v1_8 is the default value for **FileFormat /AvroVersion**, so it is not mandatory to define this action.

The JSON file format is not supported in MapR.

Connecting to MapR

HVR can connect to MapR with or without using the MapR user impersonation. The configuration/setup requirements differ on each scenarios mentioned below.

- Without MapR User Impersonation
- With MapR User Impersonation

Without MapR User Impersonation

- When hub connects to MapR server using HVR remotelistener installed on the MapR server.
  1. Log in as MapR user and start the remotelistener on the MapR server.
- When hub connects directly to the MapR server or if the integrate server (this is separate from MapR server) connects to the MapR server.
  1. Create a MapR user on the hub/integrate server. Login as **root** user and execute the following commands.

```
groupadd -g2000 mapr

useradd -gmapr -m -u2000 mapr

passwd mapr
```

  2. Login as MapR user and start the remotelistener on hub/integrate server.

With MapR User Impersonation

For more information about MapR user impersonation, refer to MapR Documentation.

- When the hub connects to the MapR server using the remotelistener on the MapR server, the MapR user impersonation is not required.
- When the hub connects directly to the MapR server or if the integrate server (this is separate from MapR server) connects to the MapR server.

1. Login as **root** user on hub/integrate server and modify the **core-site.xml** file available in **/opt/mapr /hadoop/hadoop-2.7.0/etc/hadoop/** directory as shown below:

```
<property>

        <name>hadoop.proxyuser.mapr.hosts</name>

        <value>*</value>

</property>

<property>

        <name>hadoop.proxyuser.mapr.groups</name>

        <value>*</value>

</property>

<property>

    <name>fs.mapr.server.resolve.user</name>

    <value>true</value>

</property>
```

   For more information, refer to MapR Documentation.
2. Create a file in **/opt/mapr/conf/proxy/** that has name of the mapr superuser or any other user. This file can also be copied as shown below,

```
sudo      cp      /opt/mapr/conf/proxy/mapr      /opt/mapr/conf/proxy/
hvrremotelistener_os_user
```

3. Export MapR impersonation,

```
export MAPR_IMPERSONATION_ENABLED=true
```

4. Start hvrremotelistener as *hvrremotelistener_os_user*
5. On the MapR server, execute the following using the **config** command,

```
maprcli config save -values {cldb.security.resolve.user:1};
```

   For more information, refer to MapR Documentation.

   a. To verify this update, execute:

```
maprcli config load  -keys cldb.security.resolve.user
```

6. Restart the MapR CLDB.

Requirements for Hive ACID

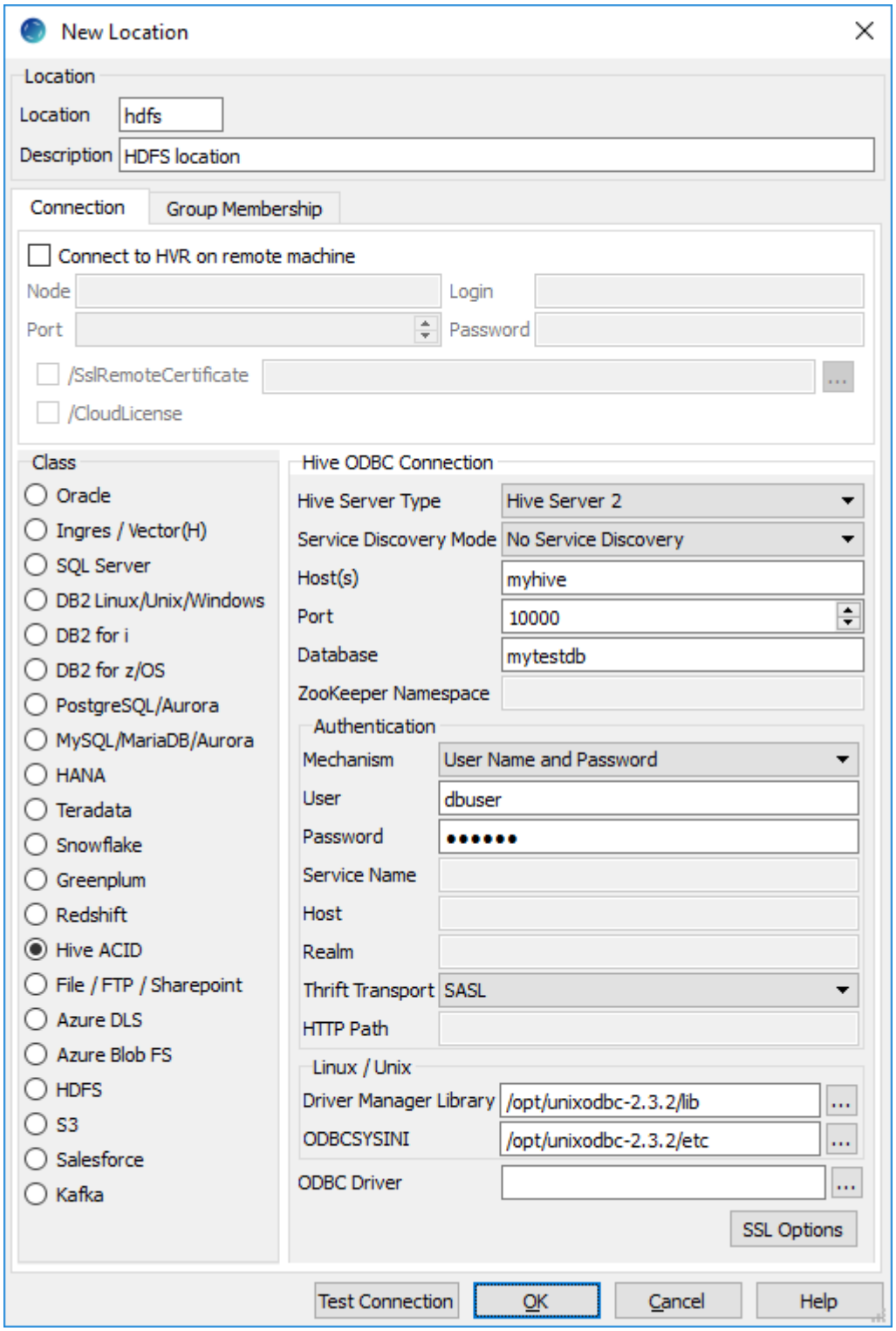
Contents

This section describes the requirements, access privileges, and other features of HVR when using Hive ACID (Atomicity, Consistency, Isolation, Durability) for replication. For information about compatibility and supported versions of Hive ACID with HVR platforms, see Platform Compatibility Matrix.

For the Capabilities supported by HVR on Hive ACID, see Capabilities for Hive ACID.

For information about the supported data types and mapping of data types in source DBMS to the corresponding data types in target DBMS or file format, see Data Type Mapping.

ODBC Connection

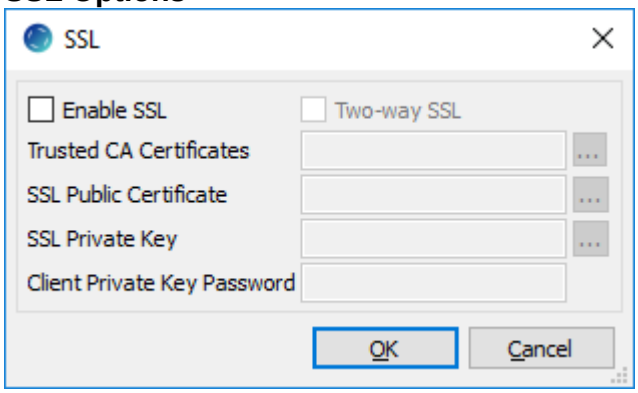


Field	Description
Hive ODBC Connection	
Hive Server Type	The type of Hive server. Available options: • **Hive Server 1** (default): The driver connects to a Hive Server 1 instance. • **Hive Server 2**: The driver connects to a Hive Server 2 instance.

Service Discovery Mode	The mode for connecting to Hive. This field is enabled only if **Hive Server Type** is **Hive Server 2**. Available options: • **No Service Discovery** (default): The driver connects to Hive server without using the ZooKeeper service. • **ZooKeeper**: The driver discovers Hive Server 2 services using the ZooKeeper service.
Host(s)	The hostname or IP address of the Hive server. When Service Discovery Mode is ZooKeeper, specify the list of ZooKeeper servers in following format **[ZK_Host1]:[ZK_Port1],[ZK_Host2]:[ZK_Port2]**, where **[ZK_Host]** is the IP address or hostname of the ZooKeeper server and **[ZK_Port]** is the TCP port that the ZooKeeper server uses to listen for client connections. **Example:** hive-host
Port	The TCP port that the Hive server uses to listen for client connections. This field is enabled only if **Service Discovery Mode** is **No Service Discovery**. **Example:** 10000
Database	The name of the database schema to use when a schema is not explicitly specified in a query. **Example:** mytestdb
ZooKeeper Namespace	The namespace on ZooKeeper under which Hive Server 2 nodes are added. This field is enabled only if **Service Discovery Mode** is **ZooKeeper**.
Authentication	
Mechanism	The authentication mode for connecting HVR to **Hive Server 2**. This field is enabled only if **Hive Server Type** is **Hive Server 2**. Available options: • **No Authentication** (default) • **User Name** • **User Name and Password** • **Kerberos** • **Windows Azure HDInsight Service** **Since** v5.5.0/2
User	The username to connect HVR to Hive server. This field is enabled only if **Mechanism** is **User Name** or **User Name and Password**. **Example:** dbuser
Password	The password of the **User** to connect HVR to Hive server. This field is enabled only if **Mechanism** is **User Name and Password**.
Service Name	The Kerberos service principal name of the Hive server. This field is enabled only if **Mechanism** is **Kerberos**.
Host	The Fully Qualified Domain Name (FQDN) of the Hive Server 2 host. The value of **Host** can be set as **_HOST** to use the Hive server hostname as the domain name for Kerberos authentication. If **Service Discovery Mode** is disabled, then the driver uses the value specified in the Host connection attribute. If **Service Discovery Mode** is enabled, then the driver uses the **Hive Server 2** host name returned by ZooKeeper. This field is enabled only if **Mechanism** is **Kerberos**.
Realm	The realm of the Hive Server 2 host. It is not required to specify any value in this field if the realm of the Hive Server 2 host is defined as the default realm in Kerberos configuration. This field is enabled only if **Mechanism** is **Kerberos**.

Thrift Transport Since v5.5.0/2	The transport protocol to use in the Thrift layer. This field is enabled only if **Hive Server Type** is **Hive Server 2**. Available options: • **Binary** (This option is available only if **Mechanism** is **No Authentication** or **User Name and Password**.) • **SASL** (This option is available only if **Mechanism** is **User Name** or **User Name and Password** or **Kerberos**.) • **HTTP** (This option is not available if **Mechanism** is **User Name**.) For information about determining which Thrift transport protocols your Hive server supports, refer to HiveServer2 Overview and Setting Up HiveServer2 sections in Hive documentation.
HTTP Path Since v5.5.0/2	The partial URL corresponding to the Hive server. This field is enabled only if **Thrift Transport** is **HTTP**.
Linux / Unix	
Driver Manager Library	The directory path where the Unix ODBC Driver Manager Library is installed. **Example:** /opt/unixodbc-2.3.2/lib
ODBCSYSINI	The directory path where **odbc.ini** and **odbcinst.ini** files are located. **Example:** /opt/unixodbc-2.3.2/etc
ODBC Driver	The user defined (installed) ODBC driver to connect HVR to the Hive server.
SSL Options	Show **SSL Options**.

SSL Options



Field	Description
Enable SSL	Enable/disable (one way) SSL. If enabled, HVR authenticates the Hive server by validating the SSL certificate shared by the Hive server.
Two-way SSL	Enable/disable two way SSL. If enabled, both HVR and Hive server authenticate each other by validating each others SSL certificate. This field is enabled only if **Enable SSL** is selected.
Trusted CA Certificates	The directory path where the **.pem** file containing the server's public SSL certificate signed by a trusted CA is located. This field is enabled only if **Enable SSL** is selected.
SSL Public Certificate	The directory path where the **.pem** file containing the client's SSL public certificate is located. This field is enabled only if **Two-way SSL** is selected.

SSL Private Key	The directory path where the **.pem** file containing the client's SSL private key is located. This field is enabled only if **Two-way SSL** is selected.
Client Private Key Password	The password of the private key file that is specified in **SSL Private Key**. This field is enabled only if **Two-way SSL** is selected.

Hive ACID on Amazon Elastic MapReduce (EMR)

To enable Hive ACID on Amazon EMR,

1. Add the following configuration details to the **hive-site.xml** file available in **/etc/hive/conf** on Amazon EMR:

```
<!-- Hive ACID support -->
<property>
   <name>hive.compactor.initiator.on</name>
   <value>true</value>
</property>
<property>
   <name>hive.compactor.worker.threads</name>
   <value>10</value>
</property>
<property>
   <name>hive.support.concurrency</name>
   <value>true</value>
</property>
<property>
   <name>hive.txn.manager</name>
   <value>org.apache.hadoop.hive.ql.lockmgr.DbTxnManager</value>
</property>
<property>
   <name>name>hive.enforce.bucketing</name>
   <value>true</value>
</property>
<property>
   <name>hive.exec.dynamic.partition.mode</name>
   <value>nostrict</value>
</property>
<!-- Hive ACID support end -->
```

2. Save the modified **hive-site.xml** file.
3. Restart Hive on Amazon EMR.
   For more information on restarting a service in Amazon EMR, refer to How do I restart a service in Amazon EMR? in AWS documentation.

Integrate and Refresh Target

HVR supports integrating changes into Hive ACID location. This section describes the configuration requirements for integrating changes (using **Integrate** and **refresh**) into Hive ACID location. For the list of supported Hive ACID versions, into which HVR can integrate changes, see Integrate changes into location in Capabilities.

Burst Integrate and Bulk Refresh

While is T Bn.83(T Bn.83(T Bi.83(T Bn.83(T Bi.83(i)-9.78(s)-9.w.83(T Bi.83(T Bt 92.66 Tm [(W)-4.26(

For best performance, HVR performs **Integrate** with **/Burst** and Bulk **Refresh** on Hive ACID location using staging files. HVR implements **Integrate** with **/Burst** and Bulk **Refresh** (with file staging) into Hive ACID as follows:

1. HVR first creates Hive external tables using Amazon/HortonWorks Hive ODBC driver
2. HVR then stages data into:
   - S3 using AWS S3 REST interface (cURL library) or
   - HDFS/Azure Blob FS/Azure Data Lake Storage using HDFS-compatible libhdfs API
3. HVR uses Hive SQL commands '**merge**' (**Integrate** with **/Burst**) or '**insert into**' (Bulk **Refresh**) against the Hive external tables linked to S3/HDFS/Azure Blob FS/Azure Data Lake Storage to ingest data into ACID Hive managed tables.

The following is required to perform **Integrate** with parameter **/Burst** and Bulk **Refresh** into Hive ACID:

1. HVR requires an AWS S3 or HDFS/Azure Blob FS/Azure Data Lake Storage location to store temporary data to be loaded into Hive ACID.

   If AWS S3 is used to store temporary data then HVR requires the AWS user with 'AmazonS3FullAccess' policy to access this location. For more information, refer to the following AWS documentation:
   - Amazon S3 and Tools for Windows PowerShell
   - Managing Access Keys for IAM Users
   - Creating a Role to Delegate Permissions to an AWS Service
2. Define action **LocationProperties** on Hive ACID location with the following parameters:
   - **/StagingDirectoryHvr**: the location where HVR will create the temporary staging files. The format for AWS S3 is **s3://***S3 Bucket***/***Directory* and for HDFS is **hdfs://***NameNode***:***Port***/***Directory*
   - **/StagingDirectoryDb**: the location from where Hive ACID will access the temporary staging files. If **/StagingDirectoryHvr** is an AWS S3 location then the value for **/StagingDirectoryDb** should be same as **/StagingDirectoryHvr**.
   - **/StagingDirectoryCredentials**: the AWS security credentials. The supported formats are **'aws_access_key_id="***key***";aws_secret_access_key="***secret_key***"'** or **'role="***AWS_role***"'**. How to get your AWS credential or Instance Profile Role can be found on the AWS documentation web page.
3. Since HVR uses CSV file format for staging, the following action definitions are required to handle certain limitations of the Hive deserialization implementation:

Hive ACID	*	**TableProperties /CharacterMapping="\x00>\\0;\n>\\n;\r>\\r;">\""**
Hive ACID	*	**TableProperties /MapBinary=BASE64**

Requirements for Ingres and Vector

Contents

This section describes the requirements, access privileges, and other features of HVR when using Ingres or Vector for replication. For information about compatibility and supported versions of Ingres or Vector with HVR platforms, see Platform Compatibility Matrix.

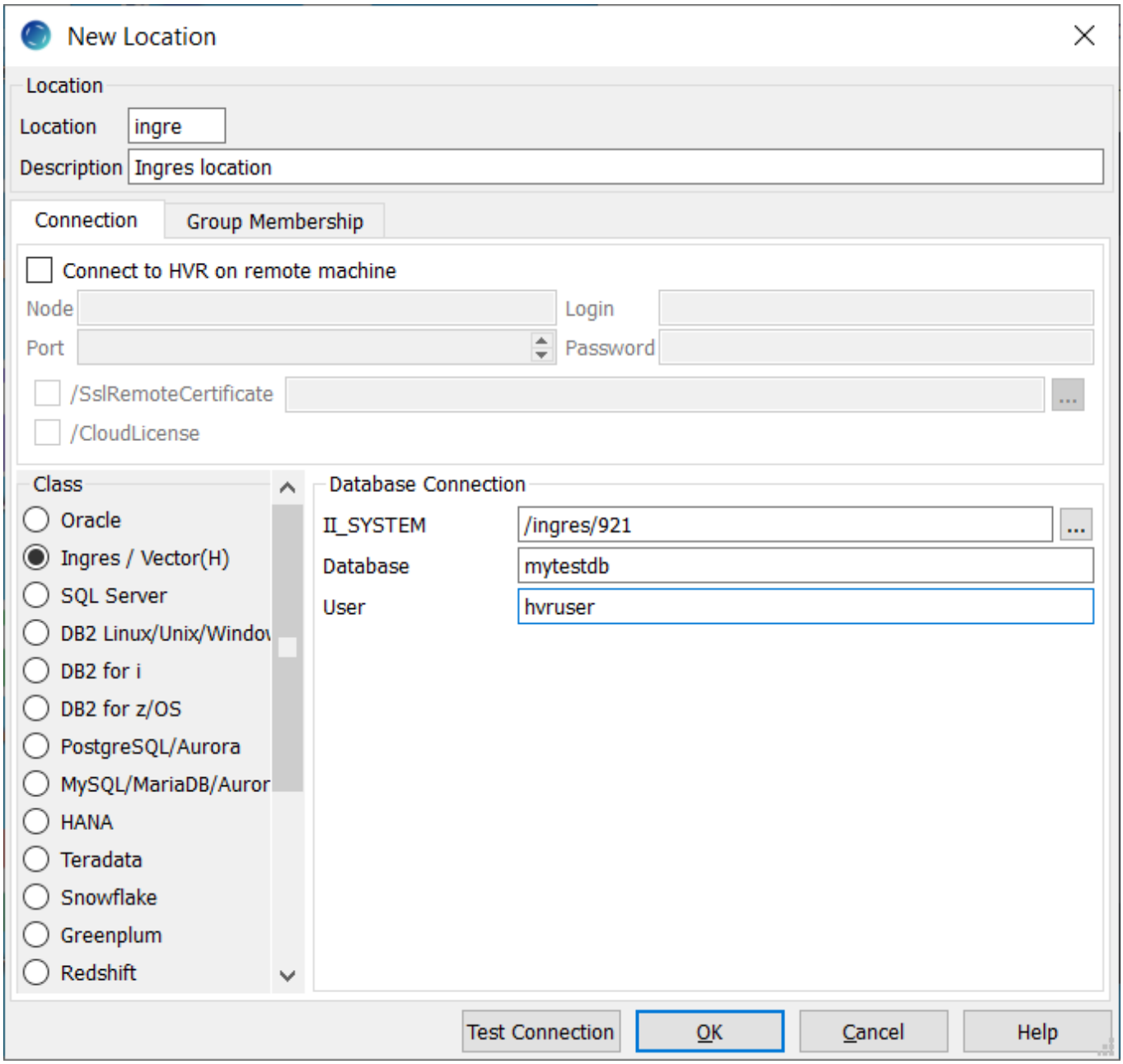
For the Capabilities supported by HVR on Ingres and Vector, see Capabilities for Ingres and Capabilities for Vector respectively.

For information about the supported data types and mapping of data types in source DBMS to the corresponding data types in target DBMS or file format, see Data Type Mapping.

To quickly setup replication using Ingres, see Quick Start for HVR - Ingres.

Location Connection

This section lists and describes the connection details required for creating an Ingres/Vector location in HVR. HVR uses Ingres OpenAPI interface to connect to an Ingres/Vector location.



Field	Description
Database Connection	
II_SYSTEM	The directory path where the Ingres/Vector database is installed. **Example:** /ingres/921
Database	The name of the Ingres/Vector database. **Example:** mytestdb
User	The username to connect HVR to Ingres/Vector **Database**. **Example:** hvruser

Access Privileges

For an Ingres or Vector hub database or database location, each account used by HVR must have permission to use Ingres.

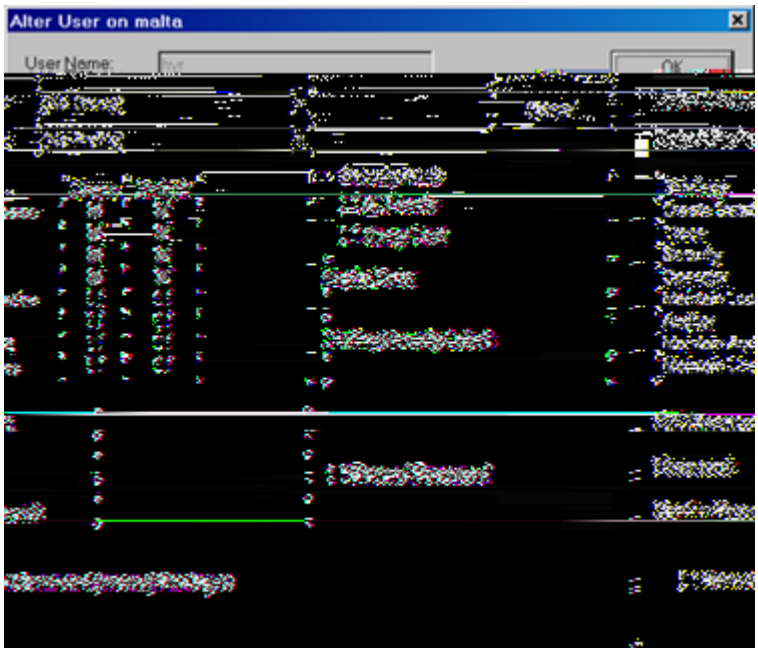
The HVR user should be the owner of the hub database.

Typically, HVR connects to database locations as the owner of that database. This means that either HVR is already running as the owner of the database, or it is running as a user with Ingres **Security Privilege**. HVR can also connect to a database location as a user who is not the database's owner, although the row-wise refresh into such a database is not supported if database rules are defined on the target tables.

Accessdb permission screen:

```
ACCESSDB - User Information

             User Name:  hvr
      Profile for User: _____________________________________
         Default Group: _____________________________________
           Expire Date: _____________________________________

Permissions:          Create Database: y              Operator: y
                  Security Administrator: y       Set Trace Flags: y
                      Maintain Locations: n        Maintain Users: n

 Databases Owned                  Authorized Databases
 hubdb

 Save(F10)   Help(Help)   End(PF3)   Password(F14)   Privileges(F17)   >
```

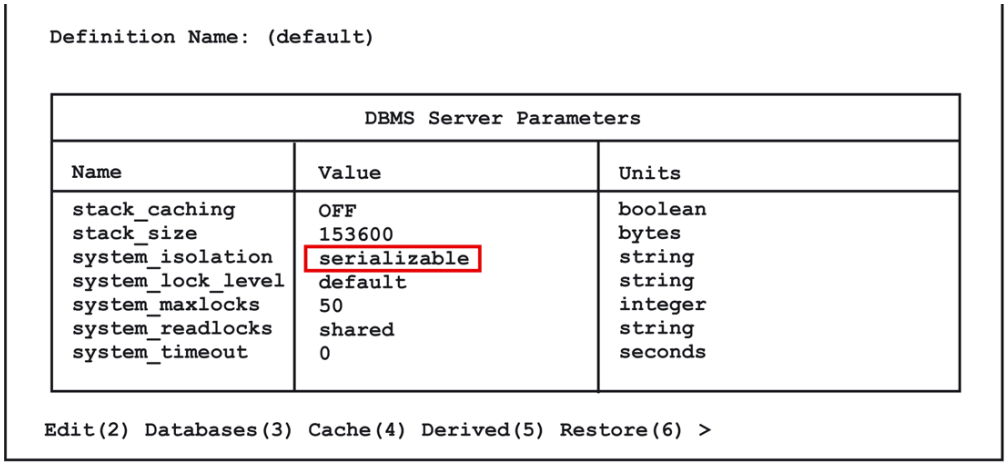
DBA permission screen:



CBF screen:

For trigger based capture from Ingres databases, the isolation level (parameter **system_isolation**) must be set to **serializable**. Other parameters (e.g. **system_readlocks**) can be anything.

```
CBF  Configure DBMS Server Definition

             Host: guam
       II_SYSTEM: /distr/ingres/303
  II_INSTALLATION: LL
```



Creating Trusted Executable

In UNIX & Linux, to perform log-based capture from Ingres a trusted executable must be created so that HVR can read from the internal DBMS logging files.

Execute the following commands while logged in as the DBMS owner (**ingres**):

```
$ cd /usr/hvr/hvr_home
$ cp bin/hvr sbin/hvr_ingres
$ chmod 4755 sbin/hvr_ingres
```

⚠ It is not required to create a trusted executable when either of the following are true:

- capture is trigger-based
- capture will be from another machine
- HVR is running as the DBMS owner (**ingres**)

Additionally, on Linux, the trusted executable should be patched using the following command:

```
$ /usr/hvr/hvr_home/lib/patchelf --set-rpath /usr/hvr/hvr_home/lib --force-rpath /usr
/hvr/hvr_home/sbin/hvr_ingres
```

If HVR and **ingres** share the same Unix group, then the permissions can be tightened from 4755 to 4750. Permissions on directories **$HVR_HOME** and **$HVR_CONFIG** may need to be loosened so that user **Ingres** can access them;

```
$ chmod g+rX $HVR_HOME
$ chmod -R g+rwX $HVR_CONFIG
```

Capture

HVR supports capturing changes from an Ingres/Vector location. HVR uses Ingres OpenAPI interface to capture changes from the Ingres/Vector location. This section describes the configuration requirements for **capturing** changes from Ingres location. For the list of supported Ingres versions, from which HVR can capture changes, see Capture changes from location in Capabilities.

Table Types

HVR supports capture from the following table types in Ingres:

- Regular tables (HEAP, HASH, ISAM, BTREE)
- Partitioned tables
- Compressed tables

Trigger-Based Capture

If trigger–based capture is defined for an Ingres database, HVR uses SQL DDL statement **modify to truncated** to empty the capture tables. The locks taken by this statement conflicts with locks taken by an on–line checkpoint. This can lead to HVR jobs hanging or deadlocking. These problems can be solved by creating file **$HVR_CONFIG/files/***dbname***.avoidddl** just before checkpointing database *dbname* and deleting it afterwards. HVR will check for this file, and will avoid DDL when it sees it.

In Unix, do this as follows:

```
$ touch $HVR_CONFIG/files/mydb.avoidddl
$ sleep 5
$ ckpdb mydb
$ rm $HVR_CONFIG/files/mydb.avoidddl
```

Log-Based Capture

If log–based capture is defined for an Ingres database (action **Capture**) then HVR may need to go back to reading the Ingres journal files. But each site has an existing backup/recovery regime that periodically deletes these Ingres checkpoint and journal files. Command **Hvrlogrelease** can make cloned copies of these files so that HVR's capture is not affected when these files are purged by the site's backup/recovery regime. When the capture job no longer needs these cloned files, then **Hvrlogrelease** will delete them again.

Integrate and Refresh

HVR supports integrating changes into Ingres and Vector locations. HVR uses Ingres OpenAPI interface to write data to Ingres/Vector during **Integrate** and **Refresh**. For **Integrate** with **/Burst** and Bulk **Refresh**, HVR uses Ingres SQL "**copy table ... () from program**" command. This section describes the configuration requirements for integrating changes (using **Integrate**) into Ingres and Vector locations. For the list of supported Ingres and Vector versions, into which HVR can integrate changes, see Integrate changes into location in Capabilities.

For HVR to integrate changes into an Ingres installation on a remote machine, special database roles (**hvr_integrate** , **hvr_refresh** and **hvr_scheduler**) must be created in that Ingres installation. Execute the following script to create these roles:

In UNIX & Linux,

```
$ sql iidbdb < $HVR_HOME/sql/ingres/hvrrolecreate.sql
```

In Windows,

```
C:\>sql iidbdb < %HVR_HOME%\sql\ingres\hvrrolecreate.sql
```

Requirements for Kafka

Contents

This section describes the requirements, access privileges, and other features of HVR when using Kafka for replication. For information about compatibility and supported versions of Kafka with HVR platforms, see Platform Compatibility Matrix.

For the Capabilities supported by HVR on Kafka, see Capabilities for Kafka.

To quickly setup replication into Kafka, see Quick Start for HVR - Kafka.

Installation Dependencies

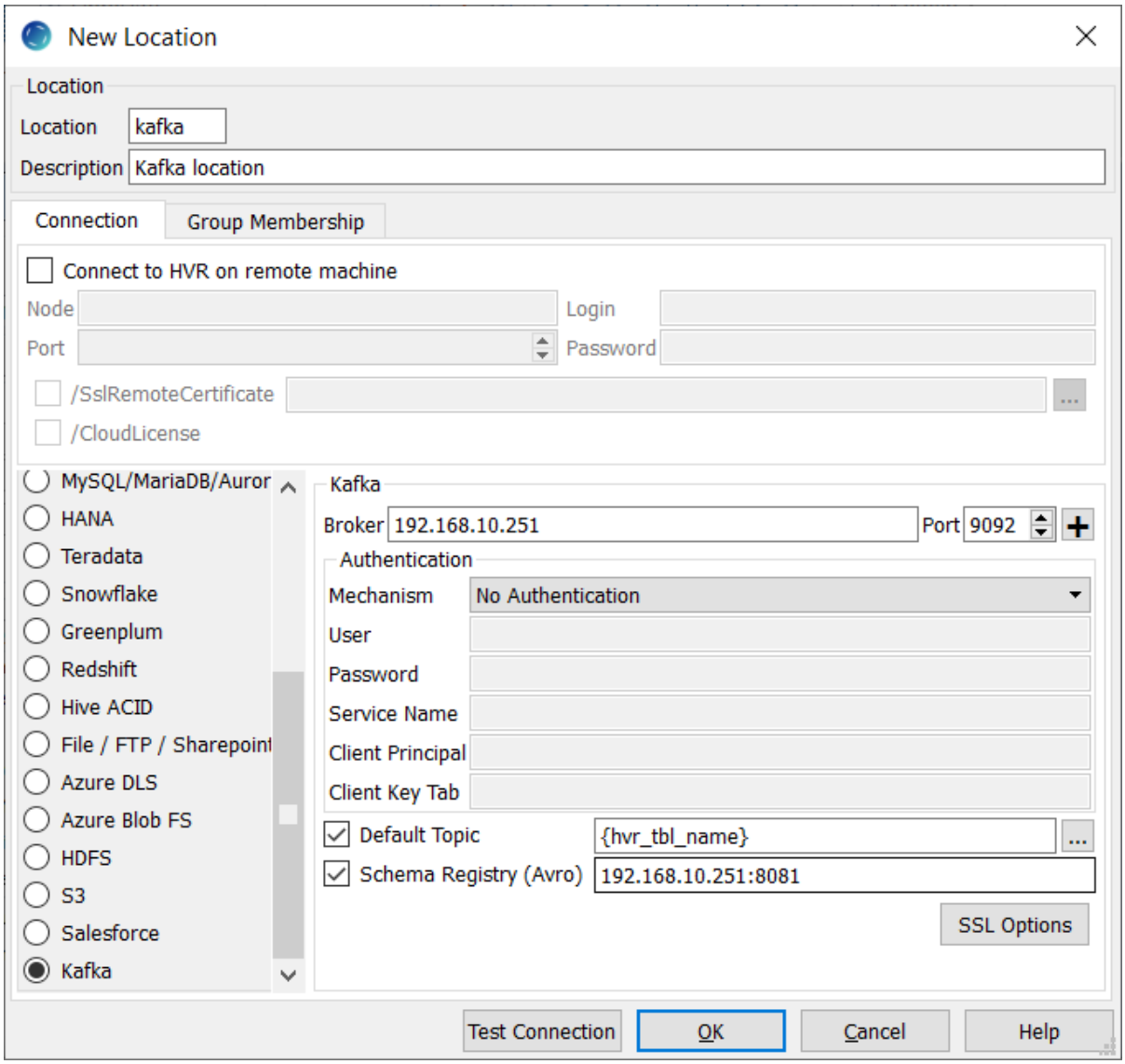
On Linux, to use either of the Kafka authentication **Mechanism** - **User Name and Password** or **Kerberos** (see Location Connection below), HVR requires the library **libsasl2.so.2** to be installed. This library is part of Cyrus SASL and can be installed as follows:

```
$ yum install cyrus-sasl    # On Red Hat Linux, CentOS
$ zypper install cyrus-sasl    # On SUSE Linux
```

There are no special requirements for installing Kafka on Windows.

Location Connection

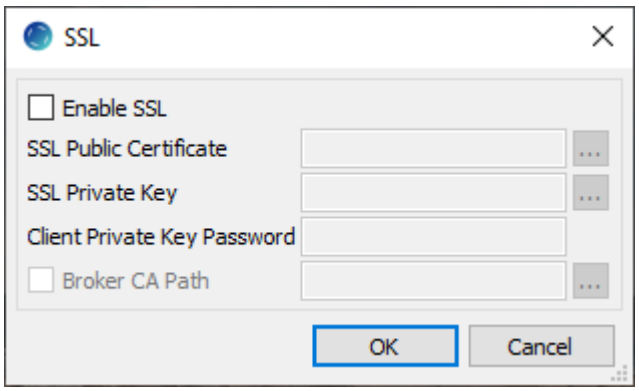
This section lists and describes the connection details required for creating Kafka location in HVR. HVR uses librdkafka (C library which talks Kafka's protocol) to connect to Kafka.



Field	Description
Kafka	
Broker	The hostname or IP address of the Kafka broker server. **Example:** 192.168.10.251
Port	The TCP port that the Kafka server uses to listen for client connections. The default port is 9092. **Example:** 9092 ⚠ HVR supports connecting to more than one Kafka broker servers. Click to add more Kafka brokers.
Authentication	

Mechanism	The authentication mode for connecting HVR to Kafka server (**Broker**). Available options: • **No Authentication** (default) • **User Name and Password** • **Kerberos** On Linux, to use **User Name and Password** or **Kerberos**, HVR requires the library **libsasl2.so.2** to be installed. For more information, see Installation Dependencies.
User	The username to connect HVR to Kafka server. This field is enabled only if **Mechanism** is **User Name and Password**.
Password	The password of the **User** to connect HVR to Kafka server. This field is enabled only if **Mechanism** is **User Name and Password**.
Service Name	The Kerberos Service Principal Name (SPN) of the Kafka server. This field is enabled only if **Mechanism** is **Kerberos**.
Client Principal	The full Kerberos principal of the client connecting to the Kafka server. This is required only on Linux. This field is enabled only if **Mechanism** is **Kerberos**.
Client Key Tab	The directory path where the Kerberos keytab file containing key for **Client Principal** is located. This field is enabled only if **Mechanism** is **Kerberos**.
Default Topic	The Kafka topic to which the messages are written. **Example:** {hvr_tbl_name}_avro
Schema Registry (Avro)	The http:// or https:// URL of the schema registry to use Confluent compatible messages in Avro format. For more information, see Kafka Message Format. **Example:** http(s)://192.168.10.251:8081
SSL Options	Show **SSL Options**.

SSL Options



Field	Description
Enable SSL	Enable/disable (one way) SSL. If enabled, HVR authenticates the Kafka server by validating the SSL certificate shared by the Kafka server.
SSL Public Certificate	The directory path where the **.pem** file containing the client's SSL public certificate is located.
SSL Private Key	The directory path where the **.pem** file containing the client's SSL private key is located.

Client Private Key Password	The password of the private key file that is specified in **SSL Private Key**.
Broker CA Path	The directory path where the file containing the Kafka broker's self-signed CA certificate is located.

Integrate and Refresh Target

HVR allows you to **Integrate** or **Refresh** changes into Kafka as a target location. HVR uses librdkafka to send data packages into Kafka message bus during **Integrate** (continuous) and **Refresh** (bulk).

This section describes the configuration requirements for integrating changes (using **Integrate** and **HVR Refresh**) into Kafka location. For the list of supported Kafka versions, into which HVR can integrate changes, see Integrate changes into location in Capabilities.

Kafka Message Format

HVR's Kafka location sends messages in JSON format by default, unless the location option **Schema Registry (Avro)** is used, in which case each message uses compact AVRO-based format. Note that this is not a true AVRO because each message would not be a valid AVRO file (e.g. no file header). Rather, each message is a 'micro AVRO', containing fragments of data encoded using AVRO data type serialization format. Both JSON (using mode **SCHEMA_PAYLOAD**, see **FileFormat** **/JsonMode**) and the 'micro AVRO' format conform to Confluent's 'Kafka Connect' message format standard and can be used with any implementation of Kafka sink connectors. When Kafka location is configured with option **Schema Registry (Avro)** (see section Location Connection above), action **FileFormat** with parameters **/Json**, **/Xml**, **/Csv**, **/AvroCompression** or **/Parquet** cannot be used.

> ⚠ If you want to use the Cloudera **Schema Registry**, you must use it in the Confluent compatible mode. This can be achieved by indicating the URL in the following format: **http://***FQDN***:***PORT***/api/v1/confluent**, where *FQDN:PORT* is the address of the Cloudera Schema Registry specified in the **Schema Registry (Avro)** field when configuring the location (see section **Location Connection** above).

Action **FileFormat** parameters **/Xml**, **/Csv**, **/Avro** or **/Parquet** can be used to send messages in other formats. If parameter **/Avro** is chosen without enabling location option **Schema Registry (Avro)** then each message would be a valid AVRO file (including a header with the schema and column information), rather than Kafka Connect's more compact AVRO-based format.

The Kafka messages should also contain special columns containing the operation type (delete, insert and update) and the sequence number. For achieving this, define action **ColumnProperties** for the Kafka location as mentioned below:

Group	Table	Action
KAFKA	*	**ColumnProperties /Name=op_val /Extra /Datatype=integer /IntegrateExpression="{hvr_op}"**
KAFKA	*	**ColumnProperties /Name=integ_seq /Extra /Datatype=varchar /Length=36 /IntegrateExpression="{hvr_integ_seq}" /TimeKey**

Metadata for Messages

To process HVR's messages, a Kafka consumer will often need metadata (table and column names, data types etc) about the replicated table. If the location is defined with option **Schema Registry (Avro)** then it can read this from that registry. For JSON format with the default mode (**SCHEMA_PAYLOAD**), each message contains this information. Another way to include metadata to each message is to add actions **ColumnProperties** **/Extra** **/IntegrateExpression** to add values like **{hvr_tbl_name}** and **{hvr_op}**.

Kafka Message Bundling and Size

By default, each Kafka message contains just one row, regardless of the format chosen. Multiple rows can be bundled into a single message using **Integrate /MessageBundling** with either of the following bundling modes:

- **CHANGE:** update message contains both 'before' and 'after' rows, inserts and deletes just contain one row
- **TRANSACTION:** message contains all rows associated with a captured transaction
- **THRESHOLD:** message is filled with rows until it reaches limit. Bundled messages simply consist of the contents of several single-row messages concatenated together.

For more information on bundling modes, refer to section /MessageBundling on the Integrate page.

Although bundling of multiple rows can be combined with the Kafka Connect compatible formats (JSON with default mode **SCHEMA_PAYLOAD**), the resulting (longer) messages no longer conform to Confluent's 'Kafka Connect' standard.

For bundling modes **TRANSACTION** and **THRESHOLD**, the number of rows in each message is affected by action **Integrate /MessageBundlingThreshold** (default is 800,000). For those bundling modes, rows continue to be bundled into the same message until after this threshold is exceeded. After that happens, the message is sent and new rows are bundled into the next message.

Parameter /**MessageBundlingThreshold** has no effect on the bundling modes **ROW** or **CHANGE**.

By default, the minimum size of a Kafka message sent by HVR is 4096 bytes; the maximum size of a Kafka message is 1,000,000 bytes; HVR will not send a message exceeding this size and will instead give a fatal error; if **Integrate /MessageCompress** parameter is used, this error will be raised by a Kafka broker. You can change the maximum Kafka message size that HVR will send by defining **$HVR_KAFKA_MSG_MAX_BYTES**, but ensure not to exceed the maximum message size configured in Kafka broker (settings **message.max.bytes**). If the message size exceeds this limit then the message will be lost.

HVR_KAFKA_MSG_MAX_BYTES works in two ways:

- checks the size of a particular message and raises an HVR error if the size is exceeded even before transmitting it to a Kafka broker.
- checks the maximum size of compressed messages inside the Kafka transport protocol.

If the message is too big to be sent because it contains multiple rows, then less bundling (e.g. **/MessageBundling=ROW**) or using a lower **MessageBundlingThreshold** can help reducing the number of rows in each message. Otherwise, the number of bytes used for each row must be lowered; either with a more compact message format or even by actually truncating a column value (by adding action **ColumnProperties /TrimDatatype** to the capture location).

Syncing Kafka, Interruption of Message Sending, and Consuming Messages with Idempotence

An HVR integrate job performs a sync of messages sent into Kafka at the end of each integrate cycle, instead of after each individual message. This means if the job is interrupted while it is sending messages, and when it is restarted, the sending of multiple rows from the interrupted cycle may be repeated. Programs consuming Kafka messages must be able to cope with this repetition; this is called being 'idempotent'. One technique to be idempotent is to track an increasing sequence in each message and use detect which messages have already been processed. A column with such an increasing sequence can be defined using action **ColumnProperties /Name=integ_key /Extra /Datatype=varchar /Length=32 /IntegrateExpression="{hvr_integ_seq}"**. If HVR resends a message, its contents will be identical each time, including this sequence number.

Kafka Message Keys and Partitioning

Kafka messages can contain a 'key' which Kafka uses to put messages into partitions, so consuming can be parallelized. HVR typically puts a key into each message which contains a hash computed from values in the 'distribution key' column of each row. This key is present only if the messages are in JSON or AVRO format. It is not present when the message bundling (**/MessageBundling**) mode is **TRANSACTION** or **THRESHOLD**.

Known Issue

When using Kafka broker version 0.9.0.0 or 0.9.0.1, an existing bug (KAFKA-3547) in Kafka causes timeout error in HVR.

The workaround to resolve this issue is to define action **Environment** for the Kafka location as mentioned below:

Group	Table	Action
KAFKA	*	**Environment /Name=HVR_KAFKA_BROKER_VERSION /Value=0.9.0.1**

Note that if the Kafka broker version used is 0.9.0.0 then **/Value=0.9.0.0**

Requirements for MySQL and MariaDB

Since v5.2.3/15

Contents

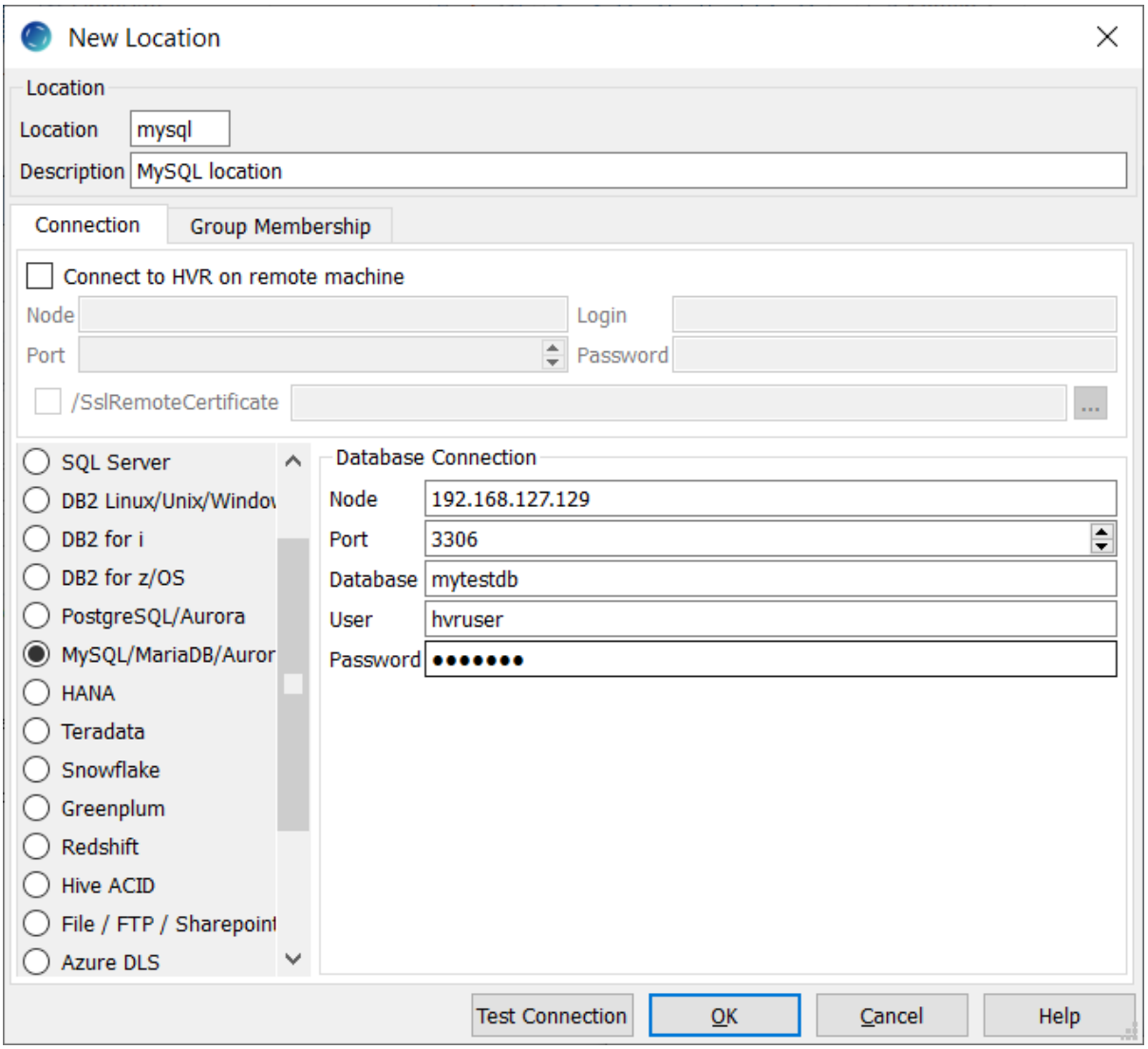
This section describes the requirements, access privileges, and other features of HVR when using MySQL/MariaDB /Aurora MySQL for replication. For information about compatibility and supported versions of MySQL/MariaDB with HVR platforms, see Platform Compatibility Matrix.

For the Capabilities supported by HVR on MySQL, MariaDB, and Aurora MySQL, see Capabilities for MySQL, Capabilities for MariaDB, and Capabilities for Aurora MySQL respectively.

For information about the supported data types and mapping of data types in source DBMS to the corresponding data types in target DBMS or file format, see Data Type Mapping.

Location Connection

This section lists and describes the connection details required for creating MySQL/MariaDB/Aurora MySQL location in HVR. HVR uses MariaDB's native Connector/C interface to connect, read, and write data to MySQL/MariaDB /Aurora MySQL. HVR connects to the MySQL/MariaDB/Aurora MySQL server using the TCP protocol.



Field	Description
Node	The hostname or IP-address of the machine on which the MySQL/MariaDB server is running. **Example:** 192.168.127.129
Port	The TCP port on which the MySQL/MariaDB/Aurora MySQL server is expecting connections. **Example:** 3306
Database	The name of the MySQL/MariaDB/Aurora MySQL database. **Example:** mytestdb
User	The username to connect HVR to MySQL/MariaDB/Aurora MySQL **Database**. **Example:** hvruser
Password	The password of the **User** to connect HVR to MySQL/MariaDB/Aurora MySQL **Database**.

HUB

HVR allows you to create hub database in MySQL/MariaDB/Aurora MySQL. The hub database is a small database which HVR uses to control its replication activities. This database stores HVR catalog tables that hold all specifications of replication such as the names of the replicated databases, the list of replicated tables, and the replication direction.

Grants for Hub

To capture changes from source database or to integrate changes into target database the HVR hub database **User** (e.g., *hvruser*) requires the following grants:

- Permission to create and drop HVR catalog tables.

Capture

Since v5.3.1/13

HVR supports **capturing** changes from MySQL/MariaDB (includes regular MySQL, MariaDB, Amazon RDS for MySQL, and Aurora MySQL) location. HVR uses MariaDB's native Connector/C interface to capture data from MySQL/MariaDB. For the list of supported MySQL, MariaDB or Aurora MySQL versions from which HVR can capture changes, see Capture changes from location in Capabilities.

There are two log read methods supported for capturing changes from MySQL/MariaDB: **SQL** and **DIRECT**. In terms of capture speed and database resource consumption, there is no much difference between using **SQL** or **DIRECT** method for capturing from a MySQL/MariaDB location.

By default, HVR captures changes from MySQL/MariaDB using the **SQL** log read method (**Capture /LogReadMethod**=**SQL**).

The **DIRECT** method requires:

- an HVR agent to be installed on the MySQL/MariaDB source database server
- the Operating System (**OS**) **user** the HVR is running under must have grants to read from binary log files.

⚠ From HVR 5.3.1/13 to HVR 5.3.1/20, capturing changes from MySQL using the **DIRECT** connection method is not available. Because of this behavior, the option **Capture /LogReadMethod** is not available for these versions of MySQL.

Grants for Capture

To capture changes from MySQL the **User** requires the following grants:

```
grant replication client on *.* to 'hvruser'@'%' ;
grant replication slave on *.* to 'hvruser'@'%' ;
grant select on *.* to 'hvruser'@'%' ;
```

Binary Logging

In MySQL, the transaction updates are recorded in the binary logs. For HVR to capture changes, the binary logging should be configured in MySQL database. MySQL allows you to define system variables (parameters) at server level (global) and at session level. The configuration for binary logging should be strictly defined as mentioned in this section. Defining parameters not mentioned in this section can lead to HVR not capturing changes.

For more information about binary logging, search for "binary logging" in MySQL Documentation.

⚠ If binary logging is not enabled in MySQL, a similar error is displayed in HVR: "hvrinit: F_JD0AC8: The 'SHOW MASTER STATUS' command returned no results. Please check that the binary logging is enabled

Binary Logging for Regular MySQL

The following parameters should be defined in MySQL configuration file **my.cnf** (Unix) or **my.ini** (Windows):

- **log_bin=ON** - to enable binary logging in MySQL.
- **binlog_format=ROW** - to set the binary logging format.
- **binlog_row_image=full** or **binlog_row_image=noblob** - to determine how row images are written to the binary log.

Binary Logging for Amazon RDS for MySQL and Aurora MySQL

This section provides information required for configuring binary logging in Amazon RDS for MySQL and Aurora MySQL database.

1. To enable binary logging, perform the steps mentioned in Amazon documentation - How do I enable binary logging for Amazon Aurora for MySQL?.
   While performing the steps to enable binary logging, the following parameters should be defined:
   - **binlog_format=ROW** - to set the binary logging format.
   - **binlog_checksum=CRC32** or **binlog_checksum=NONE** - to enable or disable writing a checksum for each event in the binary log.
2. For Aurora MySQL, the cluster should be restarted after enabling the binary logging. The replication will begin only after restarting the cluster.

3. **Backup retention period in Amazon RDS fo48(a)-o484yDQL.**

```
call mysql.rds_set_configuration('binlog retention hours', 24);
```

To display the current setting, use the **mysql.rds_show_configuration** stored procedure:

```
call mysql.rds_show_configuration;
```

Integrate and Refresh Target

HVR supports integrating changes into MySQL/MariaDB (includes regular MySQL, MariaDB, Amazon RDS for MySQL, and Aurora MySQL) location. This section describes the configuration requirements for integrating changes (using **Integrate** and **refresh**) into MySQL/MariaDB location. For the list of supported MySQL or Aurora MySQL versions, into which HVR can integrate changes, see Integrate changes into location in Capabilities.

HVR uses MariaDB's native Connector/C interface to write data into MySQL/MariaDB during continuous **Integrate** and row-wise **Refresh**. For the methods used during **Integrate** with **/Burts** and Bulk **Refresh,** see section 'Burst Integarte and Bulk Refresh' below.

Grants for Integrate and Refresh Target

To integrate changes into MySQL/MariaDB location, **User** requires the following grants:

- Permission to read and change replicated tables.

```
grant select, insert, update, delete on tbl to hvruser
```

- Permission to create and drop HVR state tables.

Prerequisites for Bulk Load

The two options available to use bulk load during **Refresh** or **Integrate** in MySQL/MariaDB are:

1. Direct loading by the MySQL/MariaDB server. The following conditions should be satisfied to use this option:
   - The **User** should have **FILE** permission.
   - The system variable (of MySQL/MariaDB server) **secure_file_priv** should be set to "" (blank).
2. Initial loading by the MySQL/MariaDB client followed by MySQL/MariaDB server. The following condition should be satisfied to use this option:
   - The system variable (of MySQL/MariaDB client and server) **local_infile** should be enabled.

Burst Integrate and Bulk Refresh

While **HVR Integrate** is running with parameter **/Burst** and Bulk **Refresh**, HVR can stream data into a target database straight over the network into a bulk loading interface specific for each DBMS (e.g. direct-path-load in Oracle), or else HVR puts data into a temporary directory (‘staging file') before loading data into a target database.

For best performance, HVR performs **Integrate** with **/Burst** and Bulk **Refresh** into a MySQL/MariaDB location using staging files. HVR implements **Integrate** with **/Burst** and Bulk **Refresh** (with file staging) into MySQL/MariaDB as follows:

Server File Staging - Direct Loading

1. HVR first stages data to a **server** local staging file (file write)
2. HVR then uses MySQL command '**load data**' to load the data into MySQL/MariaDB target tables

Client File Staging - Initial Loading

1. HVR first stages data to a **client** local staging file (file write)
2. HVR then uses MySQL command '**load data local**' to ingest the data into MySQL/MariaDB target tables

To perform **Integrate** with parameter **/Burst** and Bulk **Refresh**, define action **LocationProperties** on MySQL /MariaDB location with the following parameters:

- **/StagingDirectoryHvr**: a directory local to the MySQL/MariaDB server which can be written to by the HVR user from the machine that HVR uses to connect to the DBMS.
- **/StagingDirectoryDb**: the location from where MySQL/MariaDB will access the temporary staging files.

For MySQL on-premise, you can either define both parameters (**/StagingDirectoryHvr** and **/StagingDirectoryDb**) or define only one parameter (**/StagingDirectoryHvr**).

For MySQL on cloud, you should define only one parameter (**/StagingDirectoryHvr**).

⚠ In MySQL bi-directional channel, Bulk **Refresh** may result in looping truncates on either side of the bi-directional channel. For a workaround, refer to section MySQL Bi-directional Replication.

Compare and Refresh Source

HVR supports **compare** and **refresh** in MySQL/MariaDB location. This section describes the configuration requirements for performing **compare** and **refresh** in MySQL/MariaDB (source) location.

Grants for Compare and Refresh Source

To perform **HVR Compare** or **HVR Refresh**(in Source Database), the **User** requires the following grant to read the replicated tables:

```
grant select on tbl to hvruser
```

Requirements for Oracle

Contents

This section describes the requirements, access privileges, and other features of HVR when using Oracle for replication.

For the Capabilities supported by HVR on Oracle, see Capabilities for Oracle.

For information about the supported data types and mapping of data types in source DBMS to the corresponding data types in target DBMS or file format, see Data Type Mapping.

To quickly set up replication using Oracle, see Quick Start for HVR - Oracle.

Supported Editions

HVR supports the following Oracle editions:

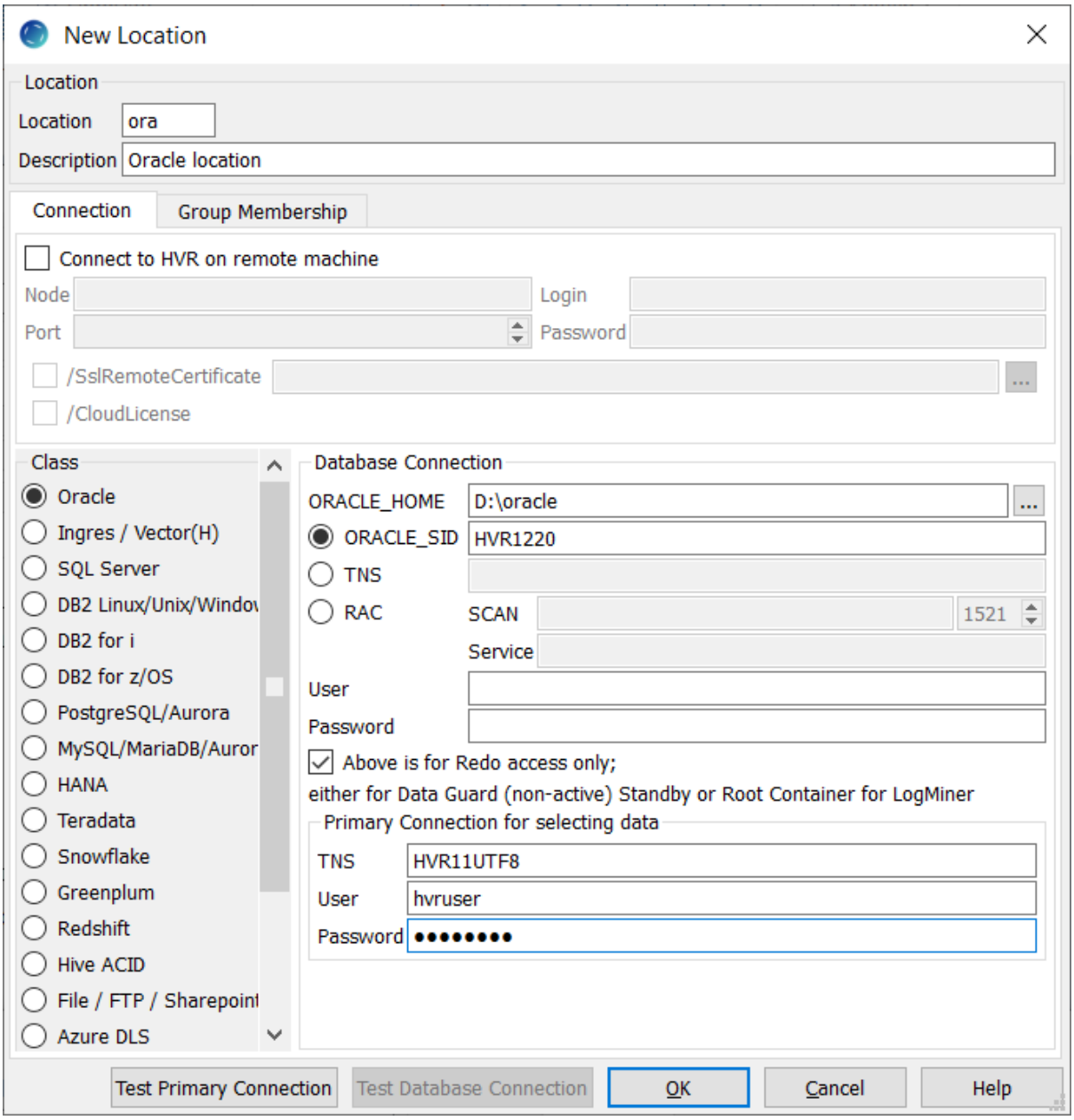
- Enterprise edition
- Standard edition (since HVR 5.6.0/0)

> HVR log-based capture is not supported on Oracle 18c Express Edition because the supplemental logging settings cannot be modified in this edition.

For information about compatibility and supported versions of Oracle with HVR platforms, see Platform Compatibility Matrix.

Location Connection

This section lists and describes the connection details required for creating the Oracle location in HVR. HVR uses Oracle's native OCI interface to connect to Oracle.



Field	Description
Database Connection	
Oracle_Home	The directory path where Oracle is installed. **Example:** D:\oracle
Oracle_SID	The unique name identifier of the instance/database. **Example:** HVR1900
TNS	The connection string for connecting to an Oracle database. The format for the connection string is **host:port/service_name**. Alternatively, the connection details can be added into the clients **tnsnames.ora** file and specify that net service name in this field. **Example:** myserver:1522/HVR1900

RAC	Parameters for connecting to RAC using SCAN configuration. • **SCAN**: Single Client Access Name (SCAN) DNS entry which can be resolved to IP address. • **Service**: The instance **service_name**. **Example:** HVR1900 Ensure that EZCONNECT is specified by the NAMES.DIRECTORY_PATH parameter in the **sqlnet.ora** file. **Example:** NAMES.DIRECTORY_PATH= (ezconnect, tnsnames)
User	The username to connect HVR to the Oracle database. **Example:** redohvruser
Password	The password of the **User** to connect HVR to the Oracle database.
Above is for Redo access only; either for Data Guard (non-Active) Standby or Root Container for LogMiner	Show/hide **Primary Connection for selecting data**. When this field is selected, the above fields **Oracle_SID** or **TNS** or **RAC** and **User** are used for connecting to the Data Guard Standby Database or Root Container for LogMiner.
Primary Connection for selecting data	
TNS	The connection string for connecting to the primary Oracle database. The format for the connection string is **host:port/service_name**. **Example:** myserver:1522/HVR1220
User	The username to connect HVR to the primary Oracle database. **Example:** hvruser
Password	The password of the **User** to connect HVR to the primary Oracle database.

Hub

HVR allows you to create a hub database (schema) in Oracle. The hub database is a repository that HVR uses to control its replication activities. This repository stores HVR catalog tables that hold all specifications of replication such as the names of the replicated databases, the list of replicated tables, and the replication direction.

Grants for Hub Schema

The hub database (schema) can be located inside an Oracle instance on the hub machine, or it can be located on another machine and connected using a TNS connection. The following grants are required for hub database user (e. g. *hvruser*):

```
grant create session to hvruser;
grant create table to hvruser;
grant create trigger to hvruser;
grant create procedure to hvruser;
```

Capture

HVR allows you to Capture changes from an Oracle database. This section describes the configuration requirements for **capturing** changes from the Oracle location. For the list of supported Oracle versions, from which HVR can capture changes, see Capture changes from location in Capabilities.

By default, HVR performs log-based capture.

Table Types

HVR supports capture from the following table types in Oracle:

- Ordinary (heap-organized) tables
- Partitioned tables
- Index-organized tables

Grants for Log-Based Capture

HVR does log-based capture if action **Capture** is defined. HVR can either connect to the database as the owner of the replicated tables, or it can connect as a special user (e.g. *hvruser*).

1. The database **User** must be granted the **create session** privilege.

   ```
   grant create session to hvruser;
   ```

2. To improve the performance of **HVR Initialize** for channels with a large number of tables (more than 150), HVR creates a temporary table (**hvr_sys_table**) within a schema.
   For HVR to automatically create this temporary table the **User** must be granted **create table** privilege.

   > If you do not wish to provide **create table** privilege to the **User**, the alternative method is to manually create this temporary table using the SQL statement:
   >
   > ```
   > create global temporary table hvr_sys_table (table_name varchar(128),
   > table_owner varchar(128));
   > ```
   >
   > This temporary table is not used when capturing from a Data Guard standby database.

3. To replicate tables that are owned by other schemas (using action **TableProperties /Schema**) the **User** must be also granted **select any table** privilege, or the user should be given individual table-level **select** privileges.
4. The **User** must be granted **select** privilege for the following data dictionary objects:

```
grant select on sys.v_$archive_dest to hvruser;

grant select on sys.v_$archived_log to hvruser;

grant select on sys.v_$database to hvruser;

grant select on sys.v_$database_incarnation to hvruser;

/* Required for identifying the redo files located on DirectNFS */

grant select on sys.v_$dnfs_files to hvruser;

/* Required for decryption */

grant select on sys.v_$encryption_wallet to hvruser;

grant select on sys.v_$log to hvruser;

grant select on sys.v_$logfile to hvruser;

/* Required for Oracle SQL/LogMiner mode. */

grant select on sys.v_$logmnr_contents to hvruser;

grant select on sys.v_$nls_parameters to hvruser;

grant select on sys.v_$parameter to hvruser;

grant select on sys.v_$pdbs to hvruser;

/* Required for reading the value of 'filesystemio_options' parameter which in
turn is used for reading the redo logs */

grant select on sys.v_$system_parameter to hvruser;

grant select on sys.all_cons_columns to hvruser;

grant select on sys.all_constraints to hvruser;

grant select on sys.all_ind_columns to hvruser;

grant select on sys.all_indexes to hvruser;

grant select on sys.all_lobs to hvruser;

grant select on sys.all_log_groups to hvruser;

grant select on sys.all_objects to hvruser;

grant select on sys.all_tab_cols to hvruser;

grant select on sys.all_tables to hvruser;

grant select on sys.all_triggers to hvruser;

grant select on sys.all_encrypted_columns to hvruser;

grant select on sys.dba_tablespaces to hvruser;

grant select on sys.obj$ to hvruser;

/* Required for Oracle 11g(11.2) fast-true feature. Since HVR version 5.6.5/3 and
5.6.0/9 */

grant select on sys.ecol$ to hvruser;
```

```
grant select on sys.user$ to hvruser;

grant select on sys.col$ to hvruser;

grant select on sys.enc$ to hvruser;

grant select on sys.tabpart$ to hvruser;

grant select on sys.tabsubpart$ to hvruser;

/* Following three grants are required for Refresh (option -qrw) and action
AdaptDDL */

grant select on sys.v_$locked_object to hvruser;

grant select on sys.v_$transaction to hvruser;

grant select on sys.dba_objects to hvruser;
```

Alternatively, the **User** can be granted the **select any dictionary** privilege to read the data dictionaries in Oracle's **SYS** schema.

```
grant select any dictionary to hvruser;
```

5. To capture **create table** statements and add supplemental logging to the newly created table(s), the **User** must be granted the following privilege:

```
grant alter any table to hvruser;
```

6. To use action **DbSequence**, the **User** must be granted the following privileges:

```
grant select any sequence to hvruser;

grant select on sys.seq$ to hvruser;
```

An alternative to all of the above grants is to provide the **sysdba** privilege to the Oracle **User** (e.g. *hvruser*):

- On Unix and Linux, add the user name used by HVR to the line in **/etc/group** for the Oracle sysadmin group.
- On Windows, right-click **My Computer** and select **Manage** ▸ **Local Users and Groups** ▸ **Groups** ▸ **ora_dba** ▸ **Add to Group** ▸ **Add**.

Related Topics	Extra Grants for Supplemental Logging, Extra Grants For Accessing Redo Files Over TNS, Extra Grants for LogMiner, Extra Grants for Amazon RDS for Oracle

Supplemental Logging

HVR needs the Oracle supplemental logging feature enabled on replicate tables that it replicates. Otherwise, when an **update** is done, Oracle will only log the columns which are changed. But HVR also needs other data (e.g. the key columns) so that it can generate a full **update** statement on the target database. The Oracle supplemental logging can be set at the database level and on specific tables. In certain cases, this requirement can be dropped. This requires **ColumnProperties** **/CaptureFromRowId** to be used and is explained below.

The very first time that **HVR Initialize** runs, it will check if the database allows any supplemental logging at all. If it is not, then **HVR Initialize** will attempt statement **alter database add supplemental log data** (see Extra Grants for Supplemental Logging to execute this statement). Note that this statement will hang if other users are changing tables. This is called 'minimal supplemental logging'; it does not actually cause extra logging; that only happens once supplemental logging is also enabled on a specific table. To see the status of supplemental logging, perform query **select log_group_type from all_log_groups where table_name='mytab'** .

The current state of supplemental logging can be checked with query **select supplemental_log_data_min, supplemental_log_data_pk, supplemental_log_data_all from v$database**. This query should return at least [**'YES'**, **'NO'**, **'NO'**].

Supplemental logging can be easily disabled (**alter database drop supplemental log data**).

HVR Initialize will normally only enable supplemental logging for the key columns of each replicated table, using statement **alter table** *tab1* **add supplemental log data (primary key) columns**. But in some cases, **HVR Initialize** will instead perform **alter table** *tab1* **add supplemental log data (all) columns**. This will happen if the key defined in the replication channel differs from the Oracle table's primary key, or if one of the following actions is defined on the table:

- On the capture location:
  - **ColumnProperties** **/CaptureExpression**
  - **Restrict** with **/CaptureCondition** , **/HorizColumn**
  - **TableProperties** **/DuplicateRows**
- On the target location:
  - **FileFormat** **/BeforeUpdateColumns**
- On any location:
  - **CollisionDetect**
  - **ColumnProperties** with **/IntegrateExpression** , **/Key** or **/TimeKey**
  - **Integrate** with **/DbProc** or **/Resilient**
  - **Integrate** **/RenameExpression**
  - **Restrict** with **/AddressTo** or **/IntegrateCondition**

Supplemental Log Data Subset Database Replication

HVR does not support the 'Supplemental Log Data Subset Database Replication' option on Oracle 19c and higher versions. This feature must be disabled for your Oracle database when using HVR for replication.

To verify whether the database is enabled for subset database replication ('YES' or 'NO'), run the following command:

```
select supplemental_log_data_sr from v$database;
```

To disable this option, run the following command:

```
alter database drop supplemental log data subset database replication;
```

Capturing from Oracle ROWID

If none of the above requirements force HVR to enable supplemental logging on all columns, the requirement for supplemental logging on key columns can be removed, if the channel is configured with **ColumnProperties**

/CaptureFromRowId and **ColumnProperties /SurrogateKey**. When these actions are defined, HVR will consider the Oracle **rowid** column as part of the table and will use it as the key column during replication, and integrate it into the target database.

The following two additional actions should be defined to the channel to instruct HVR to capture **rowid** values and to use them as surrogate replication keys. Note that these actions should be added before adding tables to the channel.

Source	**ColumnProperties /Name=hvr_rowid /CaptureFromRowId**	This action should be defined for capture locations only.
*	**ColumnProperties /Name=hvr_rowid /SurrogateKey**	This action should be defined for both capture and integrate locations.

Extra Grants for Supplemental Logging

The **User** must have the privileges mentioned in sections Grants for Log-Based Capture and the following grants for using supplemental logging:

1. To execute **alter database add supplemental log data** the **User** must have the **sysdba** privilege. Otherwise, HVR will write an error message which requests that a different user (who does have this privilege) execute this statement.
2. If HVR needs to replicate tables that are owned by other schemas, then optionally the HVR user can also be granted **alter any table** privilege, so that **HVR Initialize** can enable supplemental logging on each of the replicated tables. If this privilege is not granted then **HVR Initialize** will not be able to execute the **alter table…add supplemental log data** statements itself; instead, it will write all the statements that it needs to execute into a file and then write an error message which requests that a different user (who does have this privilege) execute these **alter table** statements.

Accessing Redo and Archive

The Oracle instance must have archiving enabled. If archiving is not enabled, HVR will lose changes if it falls behind the redo logs or it is suspended for a time.

> HVR supports capturing changes made by Oracle's direct load `insert` feature (e.g. using **insert** statements with 'append hints' (**insert /*+ append */ into**)). For HVR to capture these changes:
>
> a. a table/tablespace must not be in the **NOLOGGING** mode, because in this mode, data is inserted without redo logging.
>
> b. the archive log mode must be enabled.

Archiving can be enabled by running the following statement as **sysdba** against a mounted but unopened database: **alter database archivelog**. The current state of archiving can be checked with query **select log_mode from v$database**.

The current archive destination can be checked with the query **select destination, status from v$archive_dest**. By default, this will return values **USE_DB_RECOVERY_FILE_DEST, VALID**, which means that HVR will read changes from within the flashback recovery area. Alternatively, an archive destination can be defined with the following statement: **alter system set log_archive_dest_1='location=/disk1/arch'** and then restart the instance.

Often Oracle's **RMAN** will be configured to delete archive files after a certain time. But if they are deleted too quickly then HVR may fail if it falls behind or it is suspended for a time. This can be resolved either by (a) re-configuring **RMAN** so that archive files are guaranteed to be available for a specific longer period (e.g. 2 days), or by configuring **Hvrlogrelease** . Note that if HVR is restarted it will need to go back to the start oldest transaction that was still open, so if the system has long running transactions then archive files will need to be kept for longer.

Managing Archive/Redo Log files

If log-based capture is defined for an Oracle database (action **Capture**) then HVR may need to go back to reading the Oracle archive/redo files. But each site has an existing backup/recovery regime (normal RMAN) that periodically deletes these Oracle archive files. There are two ways to ensure that these archive files are available for HVR:

- Configure RMAN so that the archive files are always available for sufficient time for the HVR capture job(s). The 'sufficient time' depends on the replication environment; how long could replication be interrupted for, and after what period of time would the system be reset using an HVR Refresh.
- Install command **Hvrlogrelease** on the source machine to make cloned copies of the archive files so that HVR's capture is not affected when these files are purged by the site's backup/recovery regime. When the capture job no longer needs these cloned files, then **Hvrlogrelease** will delete them again.

Capturing from Oracle Data Guard Physical Standby

HVR can perform log-based capture from a Data Guard (DG) physical standby database, as well as from Active Data Guard (ADG) physical database using direct capture.

Grants for Capturing from Data Guard Physical Standby Databases

- The HVR user must have a **sysdba** privilege when capturing from a non-Active Data Guard physical standby database.
- To capture from an Active Data Guard physical standby database, the privileges described in Grants for Log-Based Capture apply.

HVR does not support SQL-based capture from a Data Guard physical standby database.

Active Data Guard

To capture from an Active Data Guard physical standby database, the following steps are required:

- Configure the standby database as the HVR location (see below) and define action **Capture** for the location.
- Configure archiving on the standby database.
- Set the necessary log-based capture grants
- Configure supplemental logging on the primary database.

HVR can also capture from an Oracle database that was previously a Data Guard physical database target.

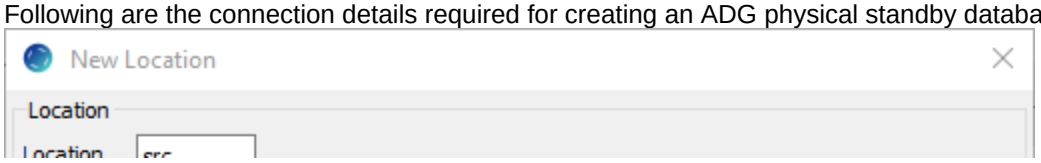
If HVR was capturing changes from one primary Oracle database and a lossless role transition occurs (so that a different Data Guard physical standby database becomes the new primary one), HVR can continue capturing from the new primary, including capturing any changes which occurred before the transition.

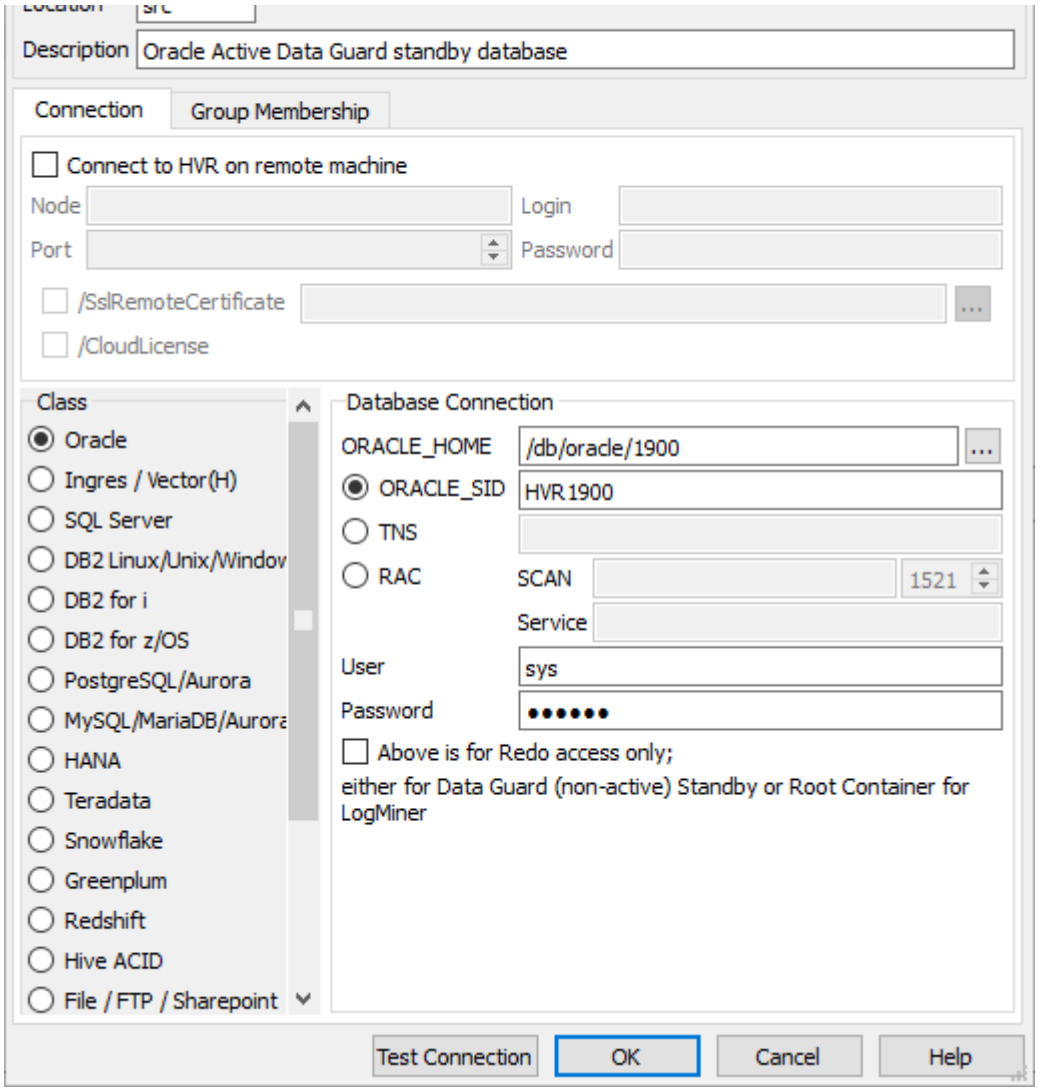
This process is automatic providing that the HVR location is connecting to Oracle in a way which 'follows the primary'. When the capture job connects to the database again, it will continue to capture from its last position (which is saved on the hub machine).

> Operator intervention is required if a failover took place requiring an OPEN RESETLOGS operation to open the primary database. To start reading transaction logs after OPEN RESETLOGS, perform **HVR Initialize** with option **Table Enrollment**.

Location Connection for Active Data Guard

Following are the connection details required for creating an ADG physical standby database location in HVR:





Field	Description
Database Connection	
ORACLE_HOME	The directory path where the standby database is installed.
ORACLE_SID	The unique name identifier of the standby database.
User	The username to connect HVR to the standby database.
Password	The password of the **User** to connect HVR to the standby database.

Non-Active Data Guard

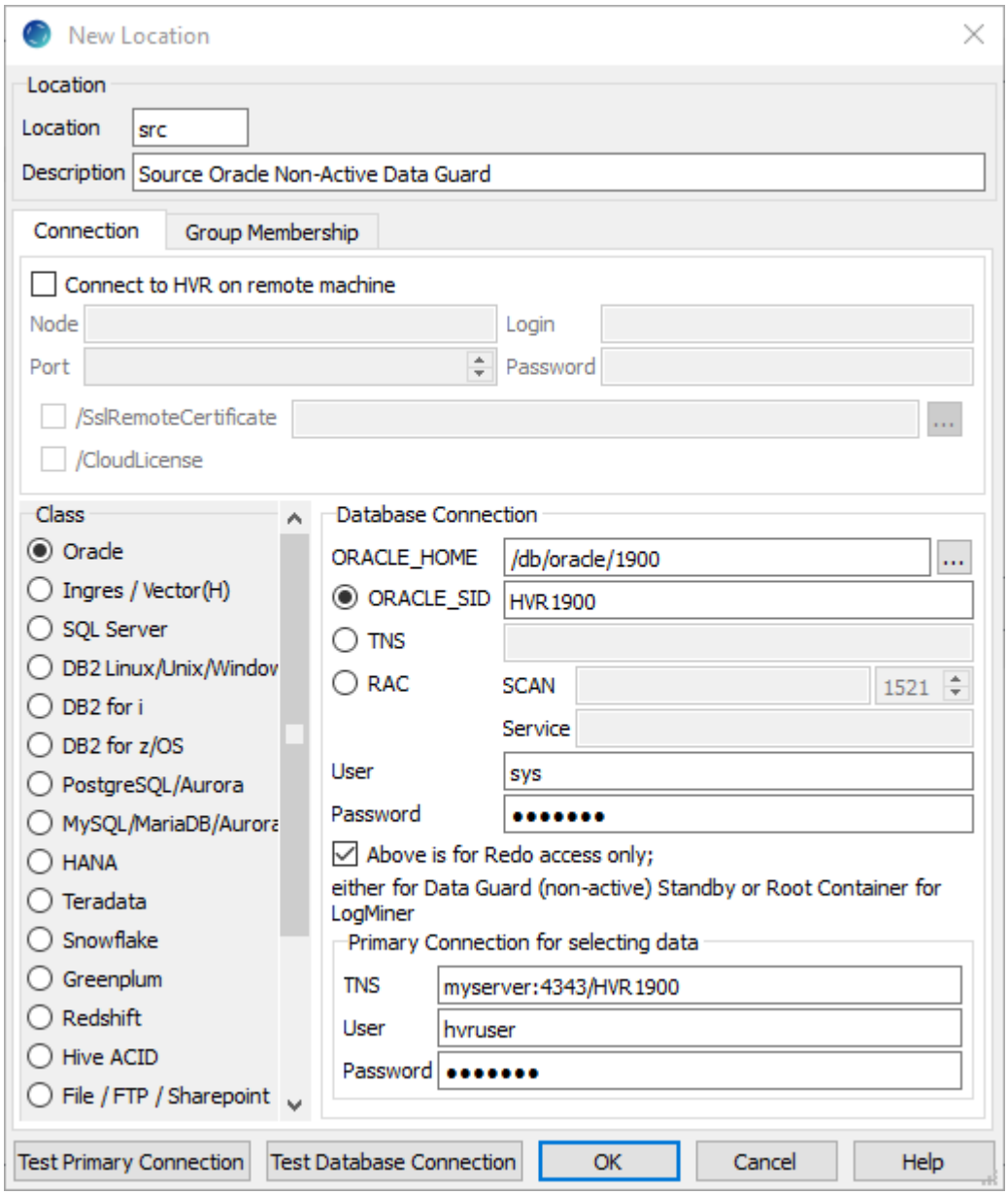
To capture from a non-Active Data Guard physical standby database, the same steps as for Active Data Guard should be done. But the HVR source location must be configured with two connections, one to the physical standby database for access to the redo and one to the primary database to access data (for example, to run HVR Refresh).

In the location creation screen, the lower part ('Primary Connection for selecting data') describes the connection to the primary database. HVR needs a connection to the primary database to do all standard operations (like **HVR Initialize**, **HVR Refresh** and **HVR Compare**). It is assumed that the primary database is reachable from the host of the standby database through a regular TNS connection.

The upper part ('Database Connection') describes the connection to a standby database. HVR needs a connection to the standby database to query the system views about the current state of redo files. With a non-Active Data Guard physical standby database that is in a mounted state as the source, this user must have a **sysdba** privilege.

Location Connection for non-Active Data Guard Physical Standby Database

Following are the connection details required for creating an Oracle Data Guard physical standby database location in HVR:



Field	Description
Database Connection	
ORACLE_HOME	The directory path where a standby database is installed.
ORACLE_SID	The unique name identifier of the standby database.
User	The username to connect HVR to the standby database.
Password	The password of the **User** to connect HVR to the standby database.
Above is for Redo access only; either for Data Guard (non-Active) Standby or Root Container for LogMiner	**Show Primary Connection for selecting data.**
Primary Connection for selecting data	
TNS	

	The connection string for connecting to the primary database. The format for the connection string is **host:port/service_name**.
User	The username to connect HVR to the primary database.
Password	The password of the **User** to connect HVR to the primary database.

Log Read Method - Direct (Default)

By default, HVR captures changes using the **DIRECT** log read method (**Capture /LogReadMethod** = **DIRECT**). In this method, HVR reads transaction log records directly from the DBMS log file using the file I/O. This method is very fast in capturing changes from the Oracle database. The **DIRECT** log read method requires that the HVR agent is installed on the source database machine.

Extra Grants For Accessing Redo Files Over TNS

For certain Oracle versions (11.2.0.2 and 11.2.0.3), HVR reads the redo and archive files c38(t)-4.c:72(c)-4.5(e)-.72(

1.
2.
   -
   -
3.
   -
   -
   -

```
$ setfacl -m u:hvr:rx $ORACLE_HOME/oradata
$ setfacl -m u:hvr:rx $ORACLE_HOME/oradata/SID
$ setfacl -m u:hvr:rx $ORACLE_HOME/oradata/SID/*
$ setfacl -m d:u::rwx,d:g::rx,d:o:,d:m:rwx $ORACLE_HOME/ora_arch
$ setfacl -m u:hvr:rx,d:u:hvr:rx $ORACLE_HOME/ora_arch
$ setfacl -m u:hvr:rx,d:u:hvr:rx $ORACLE_HOME/ora_arch/*
$ setfacl -m u:hvr:rx $ORACLE_HOME/ora_arch/*/*
```

> Sometimes a Unix file system must be mounted in **/etc/fstab** with option **acl** otherwise ACLs are not allowed. On Linux, the user **root** can use command **mount –o remount,acl** to change this dynamically.

Accessing Oracle ASM

HVR supports log-based capture from Oracle databases whose redo and archive files are located on the ASM storage. HVR uses dbms_file package to access ASM files for direct capture.

To configure this, define environment variable **$HVR_ASM_CONNECT** to a username/password pair such as **sys /sys**. The user needs sufficient privileges on the ASM instance; **sysdba** for Oracle version 10 and **sysasm** for Oracle 11+. If the ASM is only reachable through a TNS connection, you can use *username/password@TNS* as the value of **$HVR_ASM_CONNECT**. If HVR is not able to get the correct value for the **$ORACLE_HOME** of the ASM instance (e.g. by looking into **/etc/oratab**), then that path should be defined with environment variable **$HVR_ASM_HOME**. These variables should be configured using environment actions on the Oracle location.

The password can be encrypted using the **Hvrcrypt** command. For example:

Unix & Linux

```
$ export HVR_ASM_CONNECT="myuser/`hvrcrypt myuser mypass`"
```

Archive Log Only

HVR allows you to capture data from archived redo files in the directory defined using action **Capture /ArchiveLogPath**. It does not read anything from online redo files or the 'primary' archive destination.

This allows the HVR process to reside on a different machine than the Oracle DBMS and read changes from files that are sent to it by some remote file copy mechanism (e.g. **FTP**). The capture job still needs an SQL connection to the database for accessing dictionary tables, but this can be a regular connection.

Replication in this mode can have longer delays in comparison with the 'online' one. To control the delays, it is possible to force Oracle to issue an archive once per predefined period of time.

On RAC systems, delays are defined by the slowest or the less busy node. This is because archives from all threads have to be merged by SCNs in order to generate replicated data flow.

Capturing Compressed Archive Log Files

Since v5.6.5/4

HVR supports capturing data from compressed archive log files that are moved from a 'primary' archive log directory to a custom directory. HVR automatically detects the compressed files, decompresses them, and reads data from them. This feature is activated when action **Capture /ArchiveLogPath** is set to the custom directory.

HVR supports only **gzip** compressed files.

Archive Log Format

If the names of the compressed archive log files differ from the original names of the archive log files, then action **Capture /ArchiveLogFormat** should be defined to set the relevant naming format. The format variables, such as **%d** , **%r**, **%s**, **%t**, **%z**, supported for Oracle are defined in **ArchiveLogFormat** section on the **Capture** page.

Example 1:

Suppose an archive log file is named '**o1_mf_1_41744_234324343.arc**' according to a certain Oracle archive log format pattern '**o1_mf_<thread>_<sequence>_<some_number>.arc**'. The archive file is copied to some custom directory and compressed to '**o1_mf_1_41744_234324343.arc.gz**' with the **.gz** extension added to its name. In such a case, action **Capture /ArchiveLogFormat** should be defined with the following pattern '**o1_mf_%t_%s_%z.arc.gz**'.

Example 2:

Suppose the compressed archive log file is named **CQ1arch1_142657_1019376160.dbf.Z** with the **.Z** extension added to its name. In such a case, action **Capture /ArchiveLogFormat** should be defined with the following pattern '**CQ1arch%t_%s_%r.dbf.Z'**.

If action **Capture /ArchiveLogFormat** is not defined, then by default, HVR will query the database for Oracle's initialization parameter - **LOG_ARCHIVE_FORMAT**. The following are used by HVR if action **Capture /ArchiveLogFormat** is not defined,

- For Oracle ASM system, the default name pattern used is '**thread_%t_seq_%s.%d.%d**'.
- Non-ASM system,
  - if Fast-Recovery-Area (FRA) is used, then the default name pattern used is '**o1_mf_%t_%s_%z_.arc**'
  - if FRA is not used, then HVR uses the following SQL query:

```
SELECT value

FROM v$parameter

WHERE name = 'log_archive_format'
```

HVR picks up the first valid archive destination and then finds the format as described above.

To determine whether the destination uses FRA, HVR uses the following query:

```
SELECT destination

FROM v$archive_dest

WHERE dest_name='LOG_ARCHIVE_DEST_[n]';
```

For example, for destination 1, the query is as follows:

```
SELECT destination

FROM v$archive_dest

WHERE dest_name='LOG_ARCHIVE_DEST_1';
```

If the query returns **USE_DB_RECOVERY_FILE_DEST**, it indicates the destination uses FRA.

Capturing Encrypted (TDE) Tables

HVR supports capturing tables that are encrypted using Oracle Transparent Data Encryption (TDE). Capturing tables located in encrypted tablespace and tables with encrypted columns are supported for Oracle version 11 and higher.

HVR supports software and hardware (HSM) wallets. If the wallet is not configured as auto-login (Oracle internal file **cwallet.sso**), using command **Hvrlivewallet** set the password for the wallet on HVR Live Wallet port.

Software wallets can be located in ASM or in a local file system. If the wallet is located in a local file system then HVR either needs permission to read the wallet file or an HVR trusted executable should be created in **$HVR_HOME /sbin** with **chmod +4750** . If the wallet located in a local file system is configured as auto-login, then HVR or the trusted executable must be run as the user who created the wallet (usually the **oracle** user).

In Oracle 12, for replicating encrypted columns, *hvruser* should have explicit **select** privileges on **sys.user$** and **sys. enc$** tables.

```
grant select on sys.user$ to hvruser;
grant select on sys.enc$ to hvruser;
```

Further channel configuration changes are not required; HVR automatically detects encryption and opens the wallet when it is encountered.

HVR does not support capturing encrypted (TDE) tables on the HP-UX platform.

Log Read Method - SQL (LogMiner)

HVR captures changes using the **SQL** log read method (**Capture /LogReadMethod** = **SQL**). In this method, HVR uses dbms_logmnr package to read transaction log records using a special SQL function. This method reads change data over an SQL connection and does not require the HVR agent to be installed on the source database machine. However, the **SQL** log read method is slower than the **DIRECT** log read method and exposes additional load on the source database.

The **SQL** log read method enables capture from LogMiner.

Limitations of SQL Log Read Method

- Only Oracle version 11.2.0.3 and above are supported for capturing changes from LogMiner.
- Updates that only change LOB columns are not supported.
- Capture from XML Data Type columns is not supported.
- Index Organized Tables (IOT) with an overflow segment is not supported.
- Capturing DDL (using **AdaptDDL**) changes such as **add table as...** , **drop table...** and **alter table...**, including partition operations are not supported.
- Capture of truncate statements is not supported.

Extra Grants for LogMiner

The **User** must have the privileges mentioned in section Grants for Log-Based Capture and the following grants for using LogMiner:

1. **execute on dbms_logmnr**
2. **select any transaction**
3. **execute_catalog_role**
4. For Oracle 12.1 and later, the **User** must be granted the **logmining** system privilege.

Related Topics	Extra Grants for Amazon RDS for Oracle

Amazon RDS for Oracle

Since v5.3.1/9

HVR supports log-based capture and integrate into Amazon RDS for Oracle database. This section provides the information required for replicating changes in Amazon RDS for Oracle.

The following logging modes must be enabled for the Amazon RDS DB instances. You can use the Amazon RDS procedure mentioned below to enable/disable the logging modes.

- **Force Logging** - Oracle logs all changes to the database except changes in temporary tablespaces and temporary segments (NOLOGGING clauses are ignored).

```
exec rdsadmin.rdsadmin_util.force_logging(p_enable => ''true'');
```

- **Supplemental Logging** - To ensure that LogMiner and products that use LogMiner have sufficient information to support chained rows and storage arrangements such as cluster tables.

```
exec rdsadmin.rdsadmin_util.alter_supplemental_logging('''ADD''');
```

- **Switch Online Log Files** - To prevent the following error in HVR: Log scanning error F_JZ1533. The scanner could not locate the RBA file sequence *number* in the thread.

```
exec rdsadmin.rdsadmin_util.switch_logfile;
```

- **Retaining Archive Redo Logs** - To retain archived redo logs on your Amazon RDS DB instance, database backups must be enabled by setting the archivelog retention hours to greater than 0 (zero) hours. Enabling database backup can be done while creating the instance or after by going to **Instances > Modify > Backup** and set the number of days.

  The following example retains 24 hours of the redo log.

```
begin
rdsadmin.rdsadmin_util.set_configuration (name => 'archivelog retention hours',
value => '24');
end;
/
```

  Set the backup retention period for your Amazon RDS DB instance to one day or longer. Setting the backup retention period ensures that the database is running in ARCHIVELOG mode.

  For example, enable automated backups by setting the backup retention period to three days:

```
aws rds modify-db-instance \

--db-instance-identifier mydbinstance \

--backup-retention-period 3 \

--apply-immediately
```

> In Amazon RDS for Oracle, disabling automated backups may lead to replication issues in HVR.

> For better performance, it is recommended to install HVR on Amazon Elastic Cloud 2 (EC2) instance in the same region of the RDS instance. For more information about installing the HVR image on AWS, see Installing HVR on AWS using HVR Image.

Extra Grants for Amazon RDS for Oracle

The **User** must have the privileges mentioned in sections Grants for Log-Based Capture, Extra Grants for LogMiner and the following grants for using Amazon RDS for Oracle:

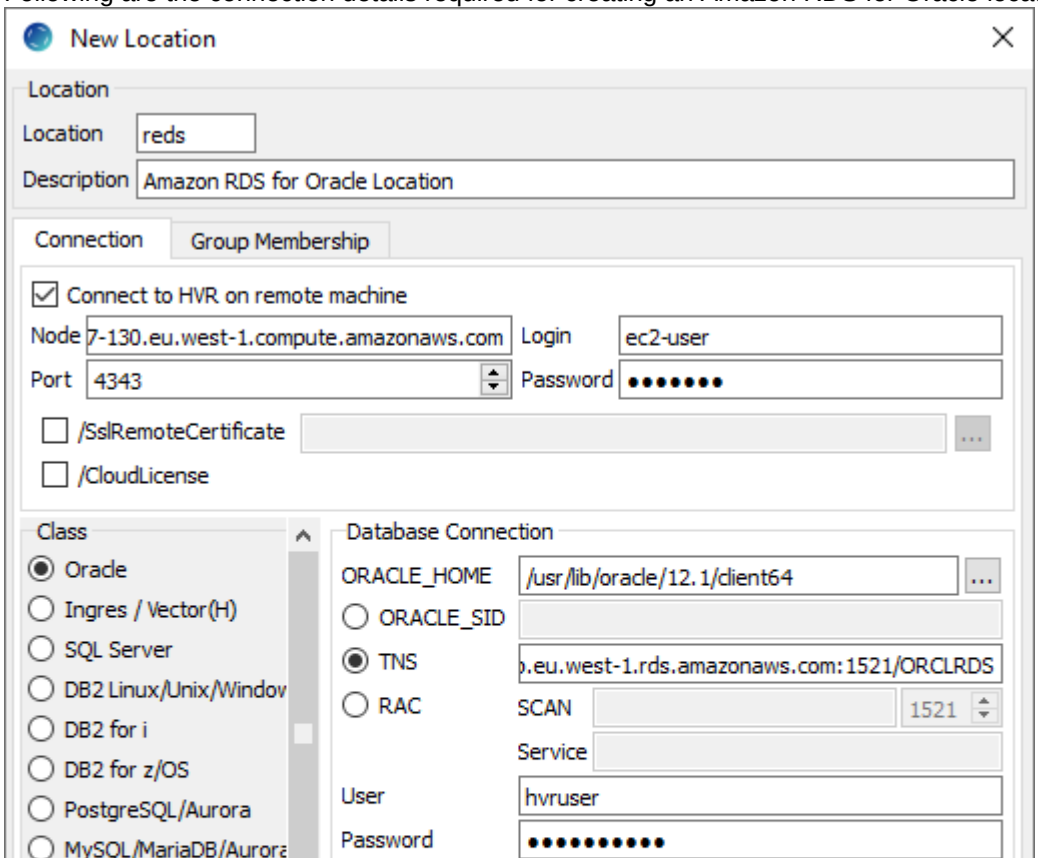
1. The **User** must be granted the **select any transaction** privilege.
2. For Amazon RDS for Oracle 12 and later, the **User** must be granted the **logmining** system privilege.
3. The **User** must be granted the LogMiner specific grants by using the Amazon RDS procedure:

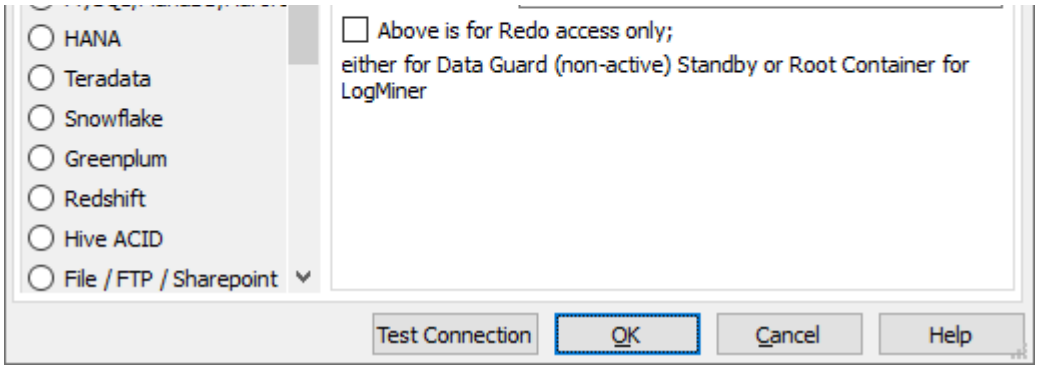
```
begin
rdsadmin.rdsadmin_util.grant_sys_object(
 p_obj_name     => 'DBMS_LOGMNR',
 p_grantee      => 'HVRUSER',
 p_privilege    => 'EXECUTE',
 p_grant_option =>  true);
end;
/
```

For more information about Amazon RDS for Oracle-specific DBA tasks, refer to Common DBA Tasks for Oracle DB Instances in AWS Documentation.

Location Connection - Amazon RDS for Oracle

Following are the connection details required for creating an Amazon RDS for Oracle location in HVR:





Field	Description
Connect to HVR on remote machine	
Node	The Public DNS of the EC2 instance.
Port	The port for the **HVR Remote Listener**.
Login	The operating system username for the EC2 instance of the **HVR Remote Listener**.
Password	The password for the operating system user account. This password is not validated if the **HVR Remote Listener** is started without password validation (option **-N**).
SslRemoteCertificate	The SSL certificate created on EC2 instance.
Database Connection	
ORACLE_HOME	The ORACLE HOME path of the EC2 instance. **Example:** /usr/lib/oracle/12.1/client64
TNS	The connection string for connecting to the RDS database. The format for the connection string is **AWS Endpoint:Port/DB Name**. Alternatively, the connection details can be added into the client's **tnsnames.ora** file and specify that net service name in this field.
User	The username to connect HVR to the Amazon RDS for Oracle database. Example: hvruser
Password	The password of the **User** to connect HVR to the Amazon RDS for Oracle database.

Capturing from Amazon RDS for Oracle

HVR uses LogMiner to capture from Amazon RDS for Oracle. DDL changes are not captured since LogMiner is used for capture. To Capture from Amazon RDS for Oracle, the following action definitions are required:

Group	Table	Action
AMAZONRDSORACLE	*	**Capture /LogReadMethod** = **SQL**
AMAZONRDSORACLE	*	**Environment /Name** = **TZ /Value** = **UTC**

Capturing from Oracle RAC

When capturing from Oracle Real Application Clusters (RAC), HVR will typically connect with its own protocol to an **HVR Remote Listener** installed in the RAC nodes. **HVR Remote Listener** should be configured to run inside all cluster nodes using the same Port, User/Login, and Password for all the Nodes. The hub then connects to one of the remote nodes by first interacting with the Oracle RAC 'SCAN' address.

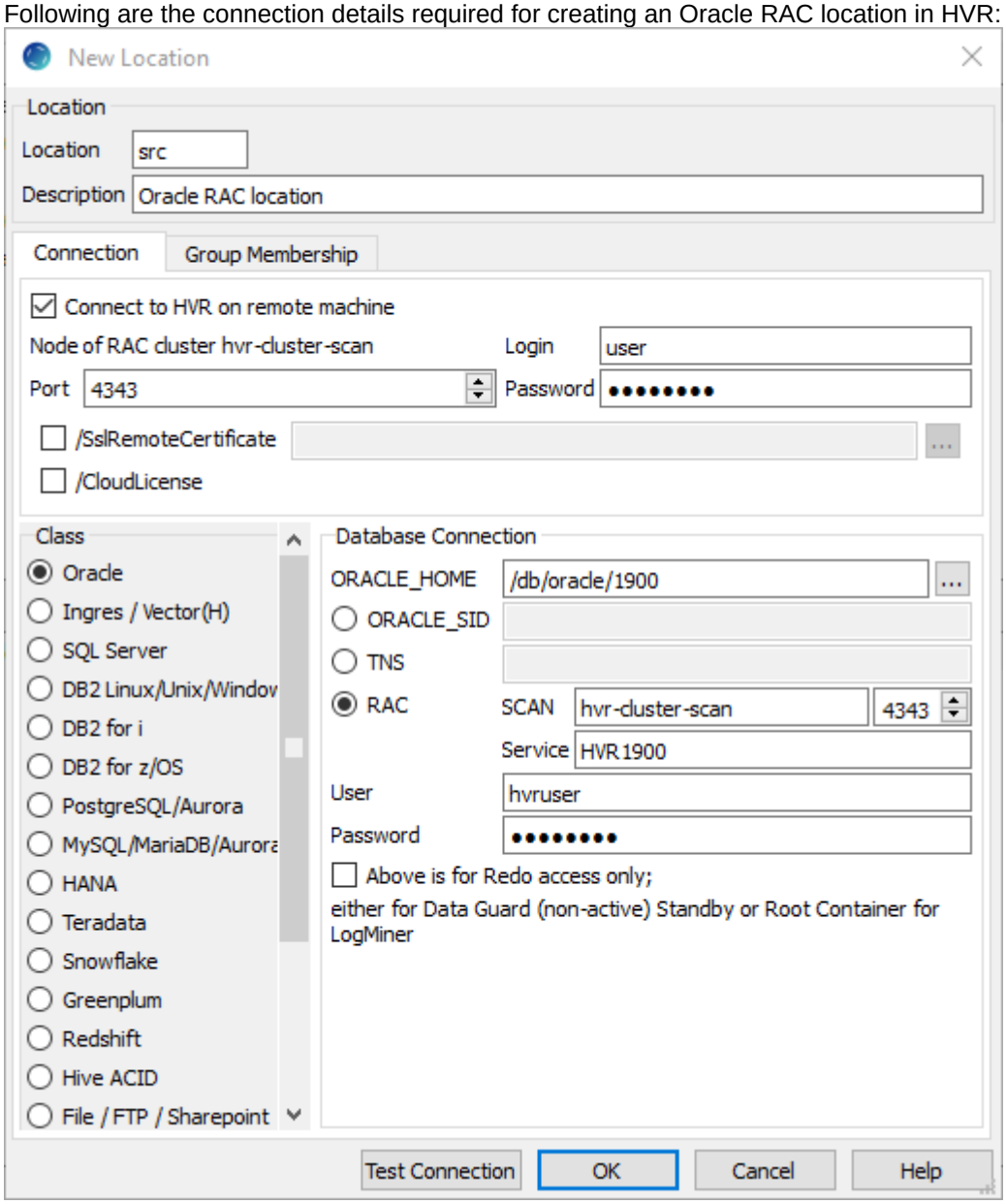
The HVR channel only needs one location for the RAC and there is only one capture job at runtime. This capture job connects to just one node and keeps reading changes from the shared redo archives for all nodes.

Directory **$HVR_HOME** and **$HVR_CONFIG** should exist on both machines but does not normally need to be shared. If **$HVR_TMP** is defined, then it should not be shared.

> Prior to HVR 5.5.0/8, capture from Oracle RAC was only supported using the **DIRECT** mode (**Capture /LogReadMethod** = **DIRECT**). However, since HVR 5.5.0/8, capture from Oracle RAC is also supported using the **SQL** mode (**Capture /LogReadMethod** = **SQL**).

Location Connection for Oracle RAC

Following are the connection details required for creating an Oracle RAC location in HVR:



Field	Description
Connect to HVR on remote machine	
Port	The TCP/IP port number of the **HVR Remote Listener** on the RAC nodes.

Login	The operating system username for the RAC nodes where the **HVR Remote Listener** is running.
Password	The password for the operating system user account.
Database Connection	
ORACLE_HOME	The directory path where an Oracle RAC database is installed.
SCAN	The Single Client Access Name (SCAN) DNS entry which can be resolved to IP address. **Example: hvr-cluster-scan**
Service	The Oracle service name. **Example: HVR1900**
User	The username to connect HVR to the Oracle RAC database.
Password	The password of the **User** to connect HVR to the Oracle RAC database.

Capturing from Oracle Pluggable Database

HVR supports capturing from Oracle's pluggable database (PDB). A pluggable database (PDB) is a portable collection of schemas, schema objects, and non-schema objects that appears to an Oracle Net client as a separate database. One or more PDBs together function as a root multitenant container database (CDB). CDB is where the PDB is managed. For more information about PDB and CDB, refer to Oracle's documentation.

To enable HVR capturing from a PDB using log read method **SQL**, two underlying connections must be set up for the capture location: one to a PDB and other one to a CDB, to which the PDB is plugged. For the PDB, the connection string should point to the database service of the PDB. The connection method for PDB is always TNS.

Grants for Capturing from Pluggable Database

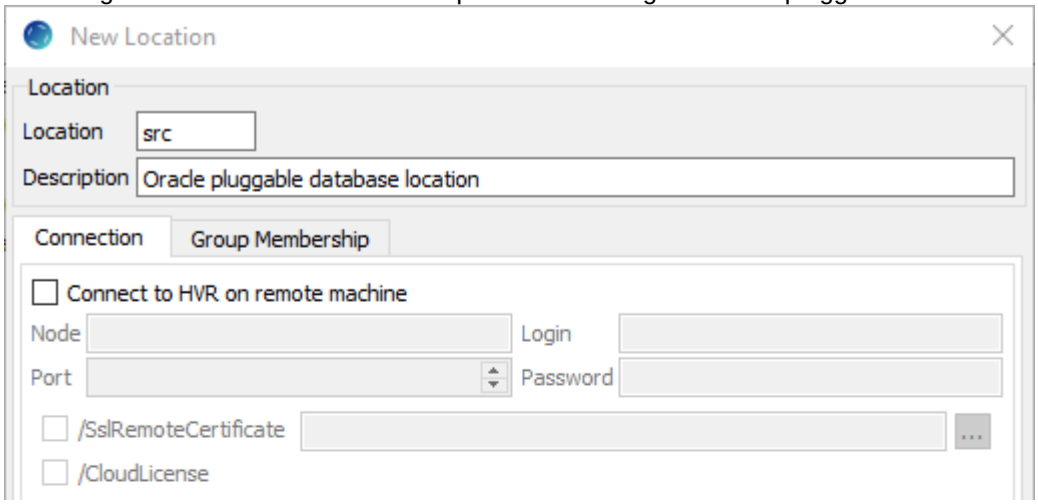
The container database **User** must be a CDB common user with the default prefix **c##** (for example, **c##hvruser**).

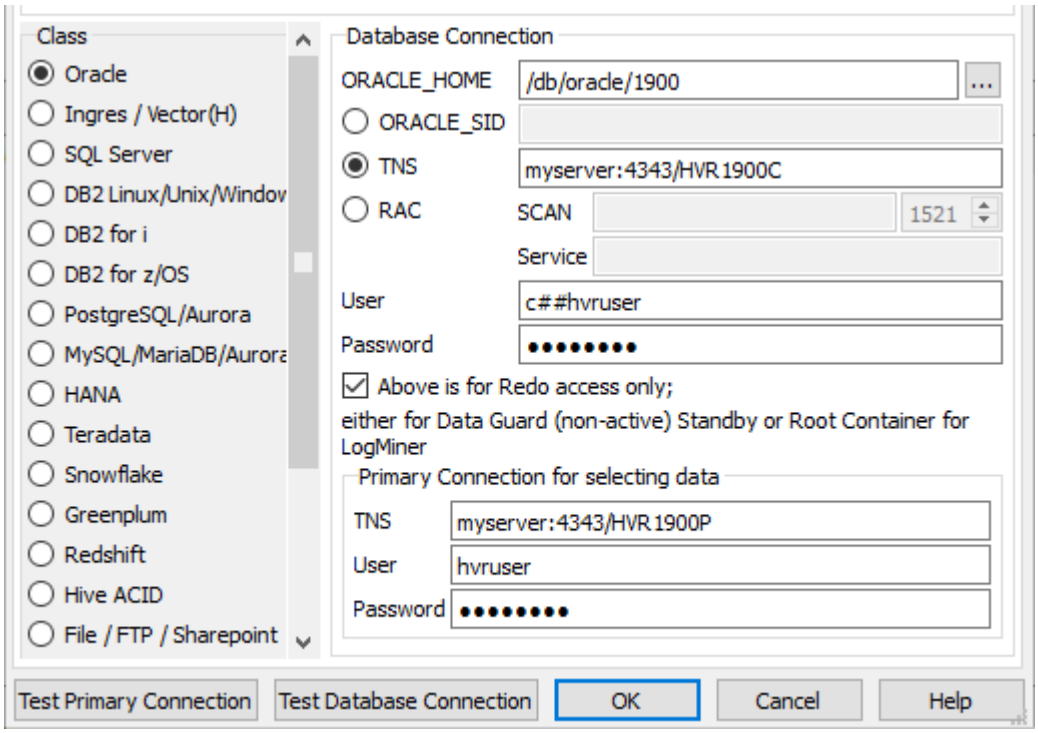
The privileges required for the CDB common user (**User**) and the PDB user are the same as the log-based capture grants. For granting privileges to the PDB user, the following command should be first executed to switch to that container (PDB user):

```
ALTER SESSION SET CONTAINER=pdbuser;
```

Location Connection for Pluggable Database

Following are the connection details required for creating an Oracle pluggable database location in HVR:





Field	Description
Database Connection	
ORACLE_HOME	The directory path where a CDB is installed.
TNS	The connection string required for connecting to the CDB (where the PDB is located). The format for the connection string is **host:port/service_name**.
User	The username to connect HVR to the CDB. **Example:** c##hvruser
Password	The password of the **User** to connect HVR to the CDB.
Above is for Redo access only; either for Data Guard (non-Active) Standby or Root Container for LogMiner	Show/Hide **Primary Connection for selecting data**.
Primary Connection for selecting data	
TNS	The connection string to a PDB. The format for the connection string is **host:port/service_name**.
User	The username to connect HVR to the PDB. **Example:** hvruser
Password	The password of the **User** to connect HVR to the PDB.

Trigger-Based Capture

HVR allows you to perform trigger-based capture when action **Capture** is defined with parameter **/TriggerBased**. HVR can either connect to the database as the owner of the replicated tables, or it can connect as a special user (e. g. **hvr**).

Grants for Trigger-Based Capture

1. The database **User** must have the following privileges:
   - **create session**
   - **create table**
   - **create trigger**
   - **create sequence**
2. To replicate tables which are owned by other schemas (using action **TableProperties /Schema**) the **User** must be granted the following privileges :
   - **select any table**
   - **execute any procedure**
   - **create any trigger**
3. To read the data dictionaries in Oracle's **SYS** schema the **User** must be granted **select any dictionary** privilege.
   An alternative to this specific grant is to provide the **sysdba** privilege to **User**.
4. Trigger-based capture will use package **dbms_alert** , unless action **Capture /TriggerBased** is defined with parameter **/ToggleFrequency** or action **Scheduling** is defined with parameters **/CaptureStartTimes** or **/CaptureOnceOnStart** . This grant can only be given by a user with the **sysdba** privilege.

Upgrading Oracle Database on Source Location

When upgrading your Oracle source database to a next release version, e.g. from 11g to 12c, the compatible mode can still be set to 11g.

The best practice when upgrading an Oracle source database to ensure no data is lost would be as follows:

1. Stop the application making changes to the database.
2. Ensure all the changes made by the application are captured: anticipate the latency to be at zero. For more information on monitoring the replication latency, refer to the **Statistics** page.
3. Stop all capture and integrate jobs under the **HVR Scheduler**.
4. Upgrade the database.
5. Run **HVR Initialize** with the following options selected: **Transaction Files and Capture Time**, **Table Enrollment**, and **Scripts and Jobs**.
6. Restart all the jobs under the **HVR Scheduler**.
7. Start the application.

Integrate and Refresh Target

HVR allows you to **Integrate** or **HVR Refresh** changes into Oracle database in the target location. This section describes the configuration requirements for integrating changes (using **Integrate** and **Refresh**) into Oracle location. For the list of supported Oracle versions into which HVR can integrate changes, see Integrate changes into location in Capabilities.

HVR uses the following interfaces to write data to Oracle during **Integrate** and **Refresh**:

- Oracle native OCI interface, used for continuous **Integrate** and row-wise **Refresh**.
- Oracle OCI direct-path-load interface, used for **Integrate** with **/Burst** and Bulk **Refresh.**

Grants for Integrate and Refresh

1. To **Integrate** changes into a database, or to load data into a database using **HVR Refresh**, the **User** must be granted the following privileges:
   - **create session**
   - **create table**
2. The **User** must be allocated a quota on the default tablespace. For this, the **User** must be granted **alter user ... quota ... on ...** or **grant unlimited tablespace to ...** privilege.

3. To change tables which are owned by other schemas (using action **TableProperties /Schema**) the **User** must be granted the following privileges:
    - **select any table**
    - **insert any table**
    - **update any table**
    - **delete any table**
4. To perform **bulk refresh** (option **-gb**) of tables which are owned by other schemas, the **User** must be granted the following privileges :
    - **alter any table**
    - **lock any table**
    - **drop any table** (needed for **truncate** statements)
5. To disable/re-enable triggers in target schema the **User** must be granted **alter any trigger** and **create any trigger** privilege.
6. If **HVR Refresh** will be used to create target tables then the **User** must be granted the following privileges:
    - **create any table**
    - **create any index**
    - **drop any index**
    - **alter any table**
    - **drop any table**
7. If action **Integrate /DbProc** is defined, then the **User** must be granted **create procedure** privilege.
8. If action **DbSequence /Schema** is defined then the **User** must be granted the following privileges:
    - **create any sequence**
    - **drop any sequence**

Compare and Refresh Source

HVR allows you to perform **HVR Compare** and **HVR Refresh** for Oracle database in the source location.

Grants for Compare or Refresh (Source Database)

1. To perform **HVR Compare** and **HVR Refresh**, the **User** must be granted the **create session** privilege.
2. If **HVR Compare** or **HVR Refresh** needs to read from tables which are owned by other schemas (using action **TableProperties /Schema**) the **User** must be granted **select any table** privilege.
3. If the **Select Moment** feature (option **-M** in **HVR Compare** and **HVR Refresh**) is used then the **User** must be granted the following privileges:
    - **flashback any table**
    - **select any transaction**

Requirements for PostgreSQL

Contents

This section describes the requirements, access privileges, and other features of HVR when using PostgreSQL /Aurora PostgreSQL for replication. For information about compatibility and supported versions of PostgreSQL with HVR platforms, see Platform Compatibility Matrix.

For the Capabilities supported by HVR on PostgreSQL, and Aurora PostgreSQL, see Capabilities for PostgreSQL and Capabilities for Aurora PostgreSQL respectively.

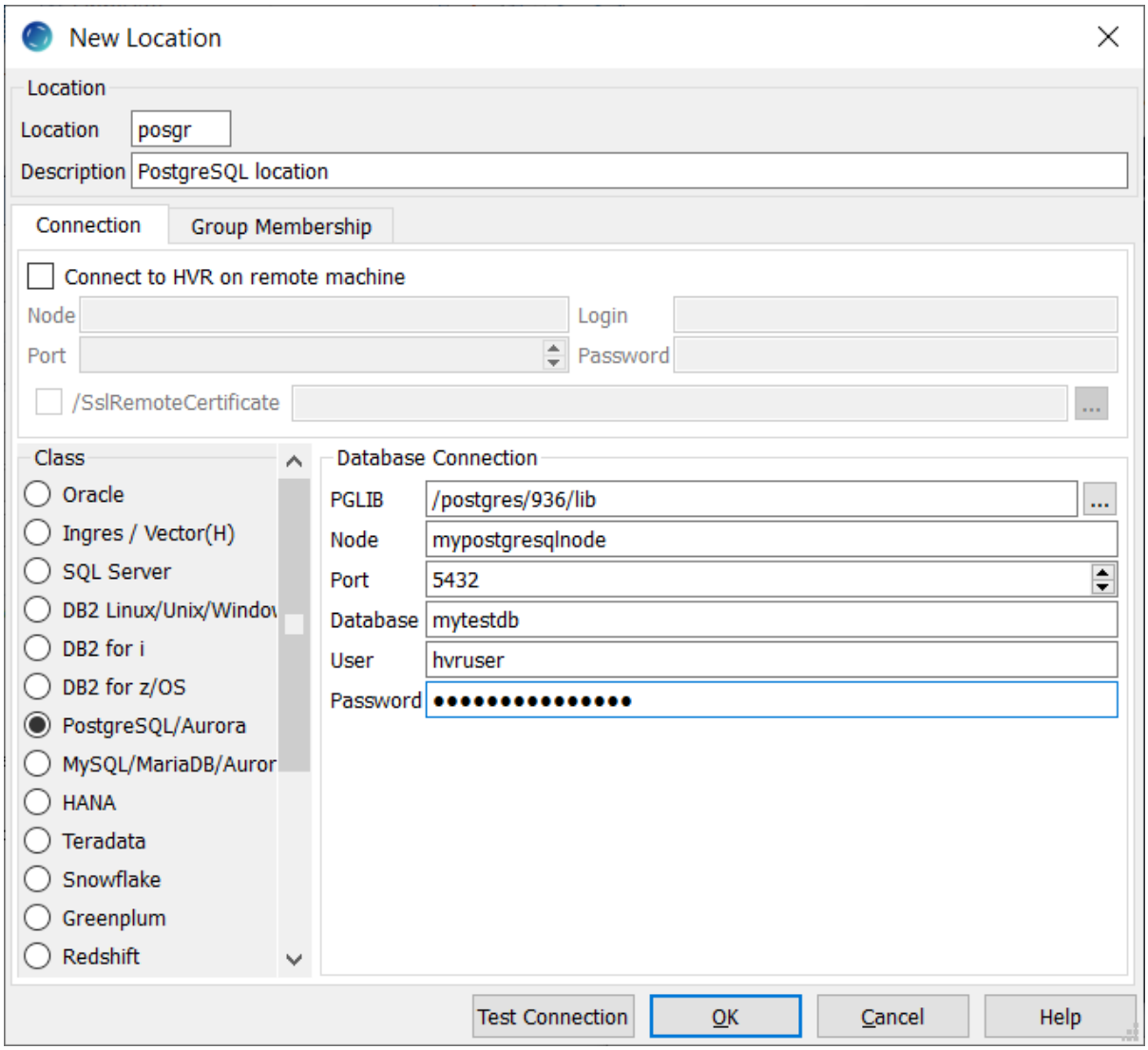
For information about the supported data types and mapping of data types in source DBMS to the corresponding data types in target DBMS or file format, see Data Type Mapping.

Connection

HVR requires that the PostgreSQL native LIBPQ (i.e. **libpq.so.5** and its dependencies) version 9, 10, or 11 is installed on the machine from which HVR will connect to PostgreSQL. Either of the client versions mentioned here can be used to connect PostgreSQL 8, 9, 10 or 11 server versions.

Location Connection

This section lists and describes the connection details required for creating PostgreSQL/Aurora PostgreSQL location in HVR.



Field	Description
PGLIB	The optional directory path of the PostgreSQL client. **Example:** /postgres/935/lib
Node	The hostname or IP-address of the machine on which the PostgreSQL server is running. **Example:** mypostgresnode
Port	The port on which the PostgreSQL server is expecting connections. **Example:** 5432
Database	The name of the PostgreSQL database. **Example:** mytestdb
User	The username to connect HVR to PostgreSQL **Database**. **Example:** hvruser
Password	The password of the **User** to connect HVR to PostgreSQL **Database**.

Hub

HVR allows you to create a hub database in PostgreSQL/Aurora PostgreSQL. The hub database is a small database that HVR uses to control its replication activities. This database stores HVR catalog tables that hold all specifications of replication such as the names of the replicated databases, the list of replicated tables, and the replication direction.

Grants for Hub

To capture changes from a source database or to integrate changes into a target database, the **User** should have a permission to create and drop HVR catalog tables.

Capture

HVR supports capturing changes from PostgreSQL (includes regular PostgreSQL, Amazon RDS for PostgreSQL , and Aurora PostgreSQL) location. HVR uses PostgreSQL native LIBPQ for capturing changes from PostgreSQL. For the list of supported PostgreSQL or Aurora PostgreSQL versions from which HVR can capture changes, see Capture changes from location in Capabilities.

⚠ Logical replication is only available with Aurora PostgreSQL version 2.2.0 (compatible with PostgreSQL 10.6) and later. For more information, refer to AWS Documentation.

Table Types

HVR supports capture from regular tables in PostgreSQL.

Grants for Log-Based Capture

In order to perform log-based capture the following statement must be executed for the replicated tables:

```
alter table tbl replica identity full
```

HVR Initialize with option **Supplemental Logging** will run these queries. This requires the **User** to be either **superuser** or the owner of the replicated tables. Alternatively, these statements can be performed by a DBA and **HVR Initialize** should be run without option **Supplemental Logging**.

Log Read Method - DIRECT

HVR captures changes using the **DIRECT** log read method (**Capture /LogReadMethod**=**DIRECT**). This capture method is supported to capture changes only from regular PostgreSQL. In this method, HVR reads transaction log records directly from the DBMS log file using the file I/O. This section describes the configuration requirements for capturing changes using **DIRECT** log read method:

1. The HVR agent must be installed on the PostgreSQL source database server.
2. PostgreSQL configuration file **postgresql.conf** should have the following settings:
   - **wal_level** = **logical**

     ```
     show wal_level;
     alter system set wal_level=logical; -- server restart needed
     ```

   - **archive_mode = on**

     ```
     show archive_mode;
     alter system set archive_mode = on; -- server restart needed
     ```

   - **archive_command**
     The value of **archive_command** depends on the location of the archive directory, the operating

system and the way archiving is done in a PostgreSQL installation. For example:
In Unix & Linux

```
 show archive_command;
  alter system set archive_command = 'test ! -f /var/lib/pgsql/9.5/data
/archive/%f && cp %p /var/lib/pgsql/9.5/data/archive/%f';     -- server
restart needed
```

In Windows

```
 show archive_command;
  alter system set archive_command = 'copy "%p" "c:\\Program
Files\\PostgreSQL\\9.5\\data\\archive\\%f"';   -- server restart needed
```

3. HVR action **Capture /XLogDirectory** should be defined. Parameter **/XLogDirectory** should contain the directory path to the PostgreSQL transaction log file directory. The operating system user as which HVR is running when connecting to PostgreSQL should have read permission to the files in this directory either directly, by running HVR as the DBMS owner (**postgres**) or via a trusted executable **$HVR_HOME/sbin/hvr_postgres**.

4. HVR action **Environment /Name /Value** should be defined. Parameter **/Name** should be set to **HVR_LOG_RELEASE_DIR**. Parameter **/Value** should contain the directory path to the directory where the PostgreSQL transaction log files are archived (for example: /distr/postgres/935/archive). The operating system user as which HVR is running when connecting to PostgreSQL should have read permission to the files in this directory either directly, by running HVR as the DBMS owner (**postgres**), or via a trusted executable **$HVR_HOME/sbin/hvr_postgres**.

   a. To create a **hvr_postgres** executable, execute the following commands while logged in as the DBMS owner (**postgres**):

   ```
   $ cd /usr/hvr/hvr_home
   $ cp bin/hvr sbin/hvr_postgres
   $ chmod 4755 sbin/hvr_postgres
   ```

   b. When user **postgres** does not have permission to write to the HVR installation directories, the following commands can be executed as user **root**:

   ```
   $ cd /usr/hvr/hvr_home
   $ cp /usr/hvr/hvr_home/bin/hvr /usr/hvr/hvr_home/sbin/hvr_postgres
   $ chown postgres:postgres /usr/hvr/hvr_home/sbin/hvr_postgres
   $ chmod 4755 /usr/hvr/hvr_home/sbin/hvr_postgres
   ```

   c. Additionally, on Linux the trusted executable needs to be patched using:

   ```
   $ /usr/hvr/hvr_home/lib/patchelf --set-rpath /usr/hvr/hvr_home/lib --force-
   rpath /usr/hvr/hvr_home/sbin/hvr_postgres
   ```

Log Read Method - SQL

HVR captures changes using the **SQL** log read method (**Capture /LogReadMethod**=**SQL**). This capture method supports capturing changes from regular PostgreSQL, Amazon RDS for PostgreSQL, and Aurora PostgreSQL. In this method, HVR reads transaction log records using a special SQL function.

Replication Slots

Capture/LogReadMethod = **SQL** uses PostgreSQL **replication slots**. The names for these slots have to be unique for an entire PostgreSQL cluster.

HVR uses the following naming convention for these replication slots:

hvr_*hub-name_channel-name_location-name*

For example: **hvr_hubdb_mychn_src**

This should allow multi capture in most situations. This includes multiple HVR capture jobs and also coexistence with other replication products.

PostgreSQL will not remove transaction log files for which changes exist that have not been processed by a replication slot. For this reason, replication slots have to be removed when a channel is no longer needed. This can be done manually or by running **hvrinit -d** (**Drop Object** option in GUI).

To retrieve existing replication slots, execute:

```
select slot_name from pg_replication_slots;
```

To manually remove a specific replication slot:

```
select pg_drop_replication_slot('slot_name');
```

For example:

```
    select pg_drop_replication_slot('hvr_hubdb_mychn_src');
```

Capture from Regular PostgreSQL

This section describes the configuration requirements for capturing changes from regular on-premises PostgreSQL using **SQL** log read method:

1. PostgreSQL configuration file **postgresql.conf** should have the following settings:
   - **wal_level** = **logical**

     ```
     show wal_level;
     alter system set wal_level = logical;   -- server restart needed
     ```

   - **max_replication_slots** = *number_of_slots*

     ```
     show max_replication_slots;

     alter system set max_replication_slots = number_of_slots; -- server restart
     needed
     ```

     number_of_slots should be set to at least the number of channels multiplied by the number of capture locations in this PostgreSQL installation.

2. The **User** should either have replication permission or be superuser:

```
alter user hvruser replication;
```

3. The replication plug-in **test_decoding** should be installed and **User** should have permission to use it. This plug-in is typically installed in **$PG_DATA/lib**. To test whether the plug-in is installed and **User** has the required permissions to execute the following SQL commands:

```
select pg_create_logical_replication_slot('hvr', 'test_decoding');
select pg_drop_replication_slot('hvr');
```

⚠ When capturing using **SQL** log read method:

- PostgreSQL versions before 9.4.12 should be avoided due to a PostgreSQL bug (detected in 9.4.6) which affects this log read method.
- Capture rewind (**hvrinit -i**) is not supported
- **SQL** log read method is the only option when capturing from Aurora PostgreSQL machines because the HVR agent cannot be installed on them.

Capture from Amazon RDS for PostgreSQL and Aurora PostgreSQL

HVR supports capturing changes from PostgreSQL at Amazon RDS for PostgreSQL and Aurora PostgreSQL using the log read method **SQL** (**Capture /LogReadMethod**=**SQL**).

To get the required settings and permissions the Parameter Group assigned to the Instance should have **rds.logical_replication=1**. Changing this needs to be followed by a restart of PostgreSQL.

Limitations

Only **insert**, **update** and **delete** changes are captured, **truncate** is not captured.

Integrate and Refresh Target

HVR supports integrating changes into PostgreSQL (includes regular PostgreSQL, Amazon RDS for PostgreSQL, and Aurora PostgreSQL) location. This section describes the configuration requirements for integrating changes (using **Integrate** and **refresh**) into PostgreSQL location. For the list of supported PostgreSQL or Aurora PostgreSQL versions, into which HVR can integrate changes, see Integrate changes into location in Capabilities.

HVR uses the following interfaces to write data to PostgreSQL during **Integrate** and **Refresh**:

- PostgreSQL native LIBPQ, used for continuous **Integrate** and row-wise **Refresh**.
- PostgreSQL "**copy from stdin**" command via native LIBPQ, used for **Integrate** with **/Burst** and Bulk **Refresh**.

Grants for Integrate and Refresh

- The **User** should have permission to read and change replicated tables.

```
grant select, insert, update, delete on tbl to hvruser
```

- The **User** should have permission to create and drop HVR state tables.

Compare and Refresh Source

- The **User** should have permission to read replicated tables.

```
grant select on tbl to hvruser
```

Requirements for Redshift

Contents

This section describes the requirements, access privileges, and other features of HVR when using Amazon Redshift for replication. For information about compatibility and supported versions of Redshift with HVR platforms, see Platform Compatibility Matrix.

For the Capabilities supported by HVR on Redshift, see Capabilities for Redshift.

For information about the supported data types and mapping of data types in source DBMS to the corresponding data types in target DBMS or file format, see Data Type Mapping.

To quickly setup replication using Redshift, see Quick Start for HVR - Redshift.

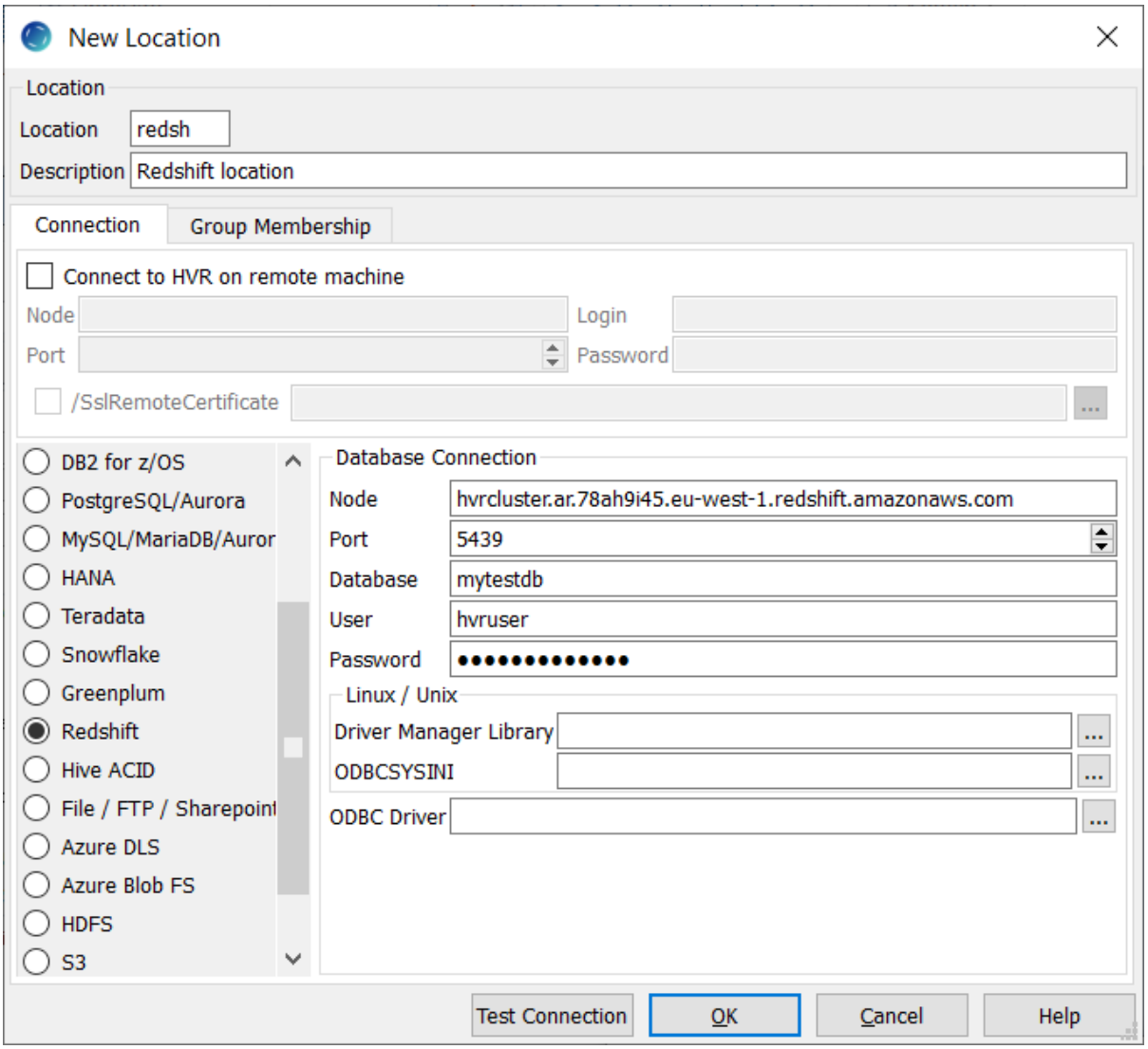
ODBC Connection

HVR uses ODBC connection to the Amazon Redshift clusters. The Amazon Redshift ODBC driver version 1.2.6.1006-1 must be installed on the machine from which HVR connects to the Amazon Redshift clusters. For more information about downloading and installing Amazon Redshift ODBC driver, refer to AWS documentation.

On Linux, HVR additionally requires unixODBC 2.3.0 or later.

Location Connection

This section lists and describes the connection details required for creating Redshift location in HVR.



Field	Description
Database Connection	
Node	The hostname or IP-address of the machine on which the Redshift server is running. **Example:** hvrcluster.ar.78ah9i45.eu-west-1.redshift.amazonaws.com
Port	The port on which the Redshift server is expecting connections. **Example:** 5439
Database	The name of the Redshift database. **Example:** mytestdb
User	The username to connect HVR to Redshift **Database**. **Example:** hvruser
Password	The password of the **User** to connect HVR to Redshift **Database**.
Linux / Unix	
Driver Manager Library	The optional directory path where the ODBC Driver Manager Library is installed. For a default installation, the ODBC Driver Manager Library is available at **/usr/lib64** and does not need to specified. When UnixODBC is installed in for example **/opt /unixodbc-2.3.1** this would be **/opt/unixodbc-2.3.1/lib**.

ODBCSYSINI	The directory path where **odbc.ini** and **odbcinst.ini** files are located. For a default installation, these files are available at **/etc** and does not need to be specified. When UnixODBC is installed in for example **/opt/unixodbc-2.3.1** this would be **/opt /unixodbc-2.3.1/etc**. The **odbcinst.ini** file should contain information about the Amazon Redshift ODBC Driver under the heading **[Amazon Redshift (x64)]**.
ODBC Driver	The user defined (installed) ODBC driver to connect HVR to the Amazon Redshift clusters.

Integrate and Refresh Target

HVR supports integrating changes into Redshift location. This section describes the configuration requirements for integrating changes (using **Integrate** and **refresh**) into Redshift location. For the list of supported Redshift versions, into which HVR can integrate changes, see Integrate changes into location in Capabilities.

HVR uses the Amazon Redshift ODBC driver to write data to Redshift during continuous **Integrate** and row-wise **Refresh**. However, the preferred methods for writing data to Redshift are **Integrate** with **/Burst** and Bulk **Refresh** using staging as they provide better performance.

Burst Integrate and Bulk Refresh

While **HVR Integrate** is running with with parameter **/Burst** and Bulk **Refresh**, HVR can stream data into a target database straight over the network into a bulk loading interface specific for each DBMS (e.g. direct-path-load in Oracle), or else HVR puts data into a temporary directory (‘staging file') before loading data into a target database.

For best performance, HVR performs **Integrate** with **/Burst** and Bulk **Refresh** into Redshift using staging files. HVR implements **Integrate** with **/Burst** and Bulk **Refresh** (with file staging) into Redshift as follows:

1. HVR first connects to S3 using the curl library and writes data into a temporary Amazon S3 staging file (s). This S3 temporary file is written in a CSV format.
2. HVR then uses Redshift SQL '**copy from s3://**' command to load data from S3 temp files and ingest them into Redshift tables.

HVR requires the following to perform **Integrate** with parameter **/Burst** and Bulk **Refresh** on Redshift:

1. An AWS S3 bucket to store the temporary data to be loaded into Redshift and an AWS user with 'AmazonS3FullAccess' policy to access this S3 bucket. For more information, refer to the following AWS documentation:
   - Amazon S3 and Tools for Windows PowerShell
   - Managing Access Keys for IAM Users
   - Creating a Role to Delegate Permissions to an AWS Service
2. Define action **LocationProperties** on the Redshift location with the following parameters:
   - **/StagingDirectoryHvr**: the location where HVR will create the temporary staging files (e.g. **s3://my_bucket_name/**).
   - **/StagingDirectoryDb**: the location from where Redshift will access the temporary staging files. This should be the S3 location that is used for **/StagingDirectoryHvr**.
   - **/StagingDirectoryCredentials**: the AWS security credentials. The supported formats are **'aws_access_key_id="***key***";aws_secret_access_key="***secret_key***"'** or **'role="***AWS_role***"'**. How to get your AWS credential or Instance Profile Role can be found on the AWS documentation web page.
3. If the S3 bucket used for the staging directory do not reside in the same region as the Redshift server, the region of the S3 bucket must be explicitly specified (this is required for using the Redshift "**copy from**" feature). For more information, search for "COPY from Amazon S3 - Amazon Redshift" in AWS documentation . To specify the S3 bucket region, define the following action on the Redshift location:

Redshift	*	**Environment /Name**=**HVR_S3_REGION /Value**=*s3_bucket_region*

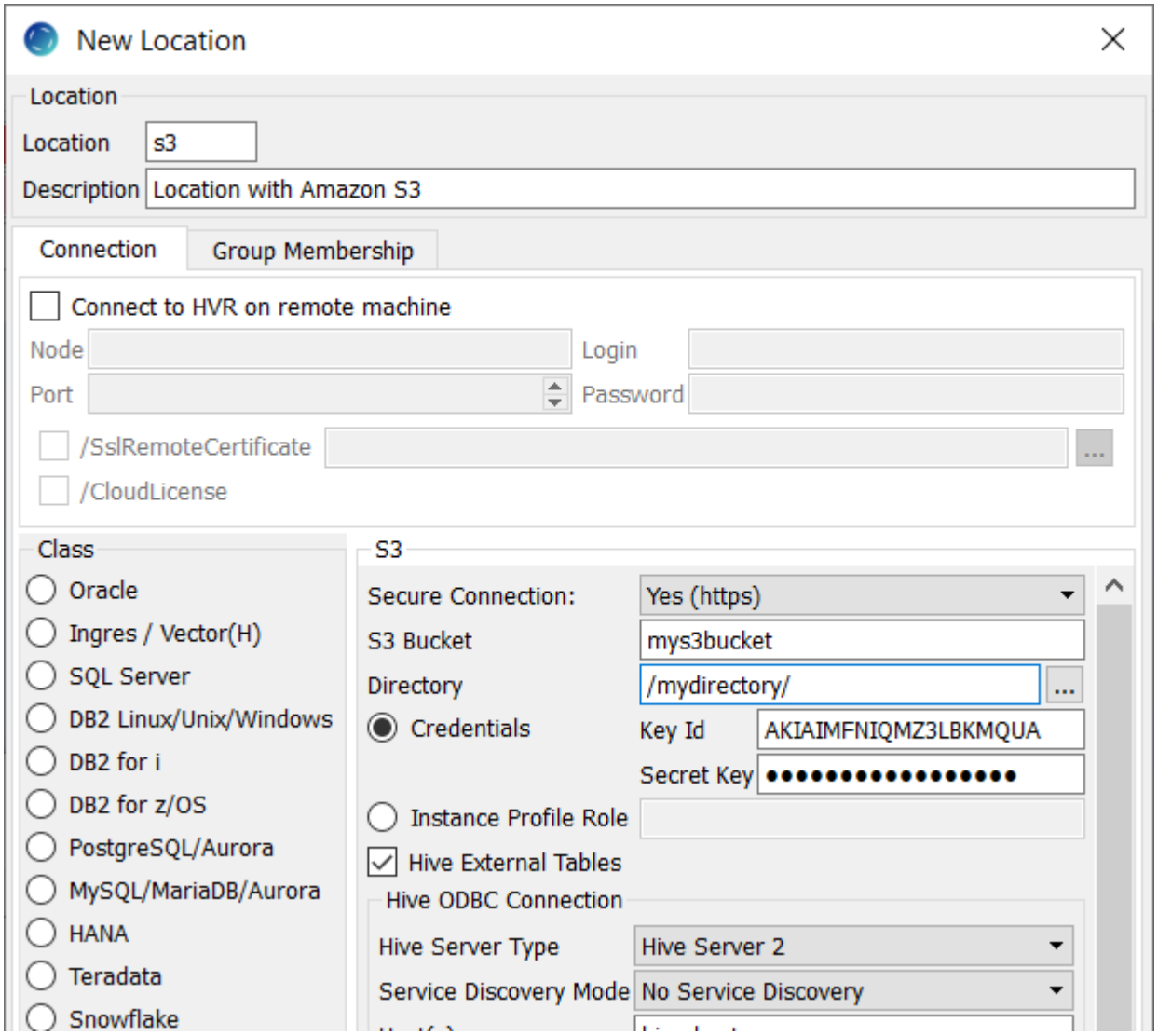
Requirements for S3

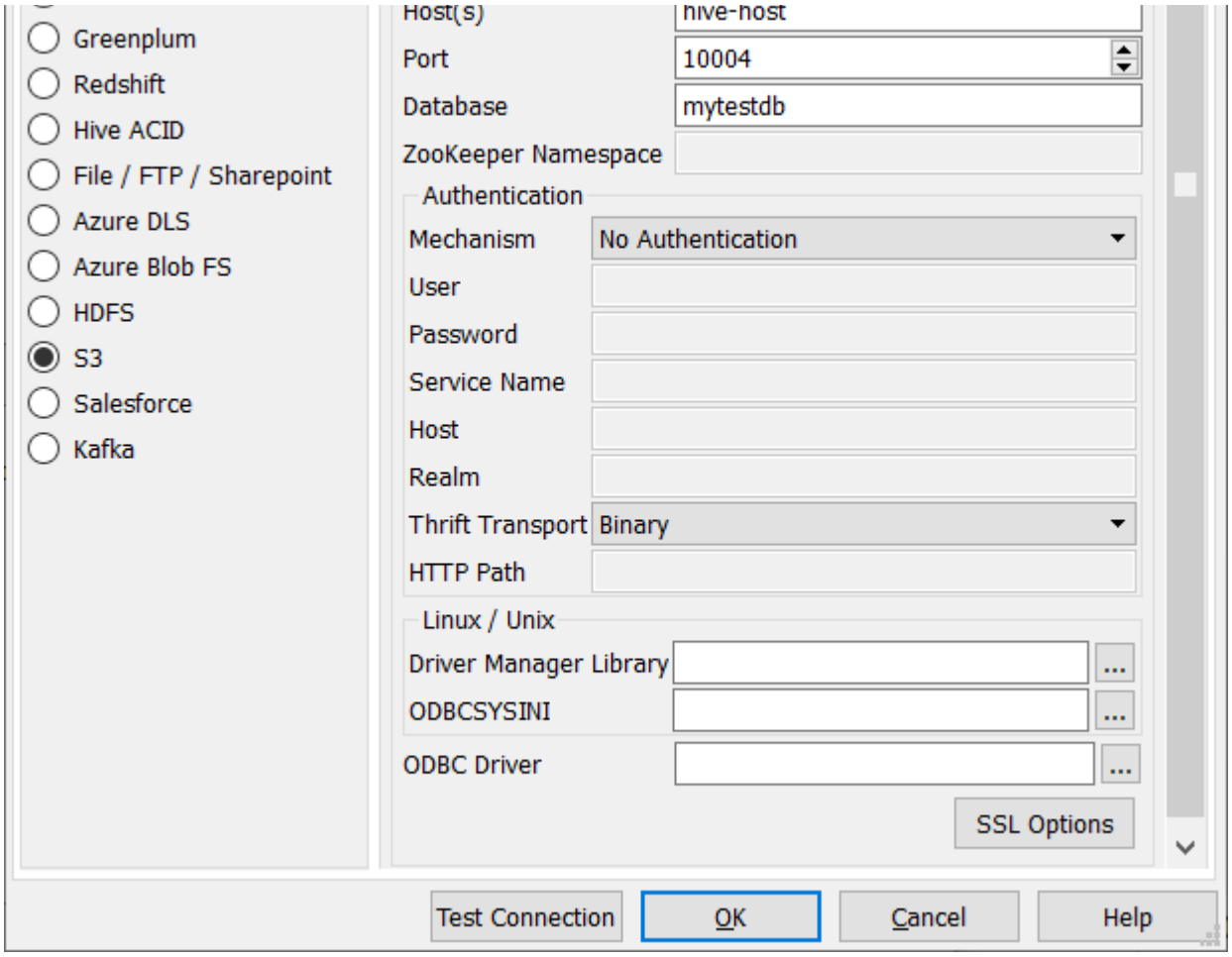
Contents

This section describes the requirements, access privileges, and other features of HVR when using Amazon S3 (Simple Storage Service) for replication. For information about compatibility and support for S3 with HVR platforms, see Platform Compatibility Matrix.

Location Connection

This section lists and describes the connection details required for creating an S3 location in HVR. HVR uses the S3 REST interface (cURL library) to connect, read and write data to S3 during capture, integrate (continuous), refresh (bulk) and compare (direct file compare).





Field	Description
S3	
Secure Connection	The type of security to be used for connecting to S3 Server. Available options: • **Yes (https)** (**default**): HVR will connect to S3 Server using HTTPS. • **No (http)**: HVR will connect to S3 Server using HTTP.
S3 Bucket	The IP address or hostname of the S3 bucket. **Example:** rs-bulk-load
Directory	The directory path in **S3 Bucket** which is to be used for replication. **Example:** /myserver/hvr/s3
Credentials	The authentication mode for connecting HVR to S3 by using IAM User Access Keys (**Key ID** and **Secret Key**). For more information about Access Keys, refer to Access Keys (Access Key ID and Secret Access Key) in section 'Understanding and Getting Your Security Credentials' of AWS documentation.
Key ID	The access key ID of IAM user to connect HVR to S3. This field is enabled only if **Credentials** is selected. **Example:** AKIAIMFNIQMZ2LBKMQUA
Secret Key	The secret access key of IAM user to connect HVR to S3. This field is enabled only if **Credentials** is selected.

Instance Profile Role	The AWS IAM role name. This authentication mode is used when connecting HVR to S3 by using AWS Identity and Access Management (IAM) Role. This option can be used only if the HVR remote agent or the HVR Hub is running inside the AWS network on an EC2 instance and the AWS IAM role specified here should be attached to this EC2 instance. When a role is used, HVR obtains temporary Access Keys Pair from the EC2 machine. For more information about IAM Role, refer to IAM Roles in AWS documentation. **Example:** Role1 or PRODROLE
Hive External Tables	Enable/Disable Hive ODBC connection configuration for creating Hive external tables above S3.

> If there is an HVR agent running on Amazon EC2 node, which is in the AWS network together with the S3 bucket, then the communication between the HUB and AWS network is done via HVR protocol, which is more efficient than direct S3 transfer. Another approach to avoid the described bottleneck is to configure the HUB on an EC2 node.

Hive ODBC Connection

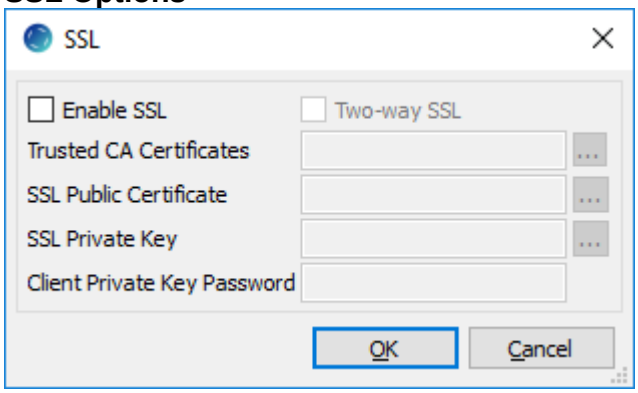
HVR uses the Hive ODBC driver to connect to Hive for creating Hive external tables above S3 to perform hvrcompare with files that reside on the S3 location. You can enable/disable the Hive configuration for S3 in the **New Location** dialog using the **Hive External Tables** field.

Field	Description
Hive ODBC Connection	
Hive Server Type	The type of Hive server. Available options: • **Hive Server 1** (default): The driver connects to a Hive Server 1 instance. • **Hive Server 2**: The driver connects to a Hive Server 2 instance.
Service Discovery Mode	The mode for connecting to Hive. This field is enabled only if **Hive Server Type** is **Hive Server 2**. Available options: • **No Service Discovery** (default): The driver connects to Hive server without using the ZooKeeper service. • **ZooKeeper**: The driver discovers Hive Server 2 services using the ZooKeeper service.
Host(s)	The hostname or IP address of the Hive server. When Service Discovery Mode is ZooKeeper, specify the list of ZooKeeper servers in following format **[ZK_Host1]:[ZK_Port1],[ZK_Host2]:[ZK_Port2]**, where **[ZK_Host]** is the IP address or hostname of the ZooKeeper server and **[ZK_Port]** is the TCP port that the ZooKeeper server uses to listen for client connections. **Example:** hive-host
Port	The TCP port that the Hive server uses to listen for client connections. This field is enabled only if **Service Discovery Mode** is **No Service Discovery**. **Example:** 10000
Database	The name of the database schema to use when a schema is not explicitly specified in a query. **Example:** mytestdb

ZooKeeper Namespace	The namespace on ZooKeeper under which Hive Server 2 nodes are added. This field is enabled only if **Service Discovery Mode** is **ZooKeeper**.
Authentication	
Mechanism	The authentication mode for connecting HVR to **Hive Server 2**. This field is enabled only if **Hive Server Type** is **Hive Server 2**. Available options: • **No Authentication** (default) • **User Name** • **User Name and Password** • **Kerberos** • **Windows Azure HDInsight Service** **Since** v5.5.0/2
User	The username to connect HVR to Hive server. This field is enabled only if **Mechanism** is **User Name** or **User Name and Password**. **Example:** dbuser
Password	The password of the **User** to connect HVR to Hive server. This field is enabled only if **Mechanism** is **User Name and Password**.
Service Name	The Kerberos service principal name of the Hive server. This field is enabled only if **Mechanism** is **Kerberos**.
Host	The Fully Qualified Domain Name (FQDN) of the Hive Server 2 host. The value of **Host** can be set as **_HOST** to use the Hive server hostname as the domain name for Kerberos authentication. If **Service Discovery Mode** is disabled, then the driver uses the value specified in the Host connection attribute. If **Service Discovery Mode** is enabled, then the driver uses the **Hive Server 2** host name returned by ZooKeeper. This field is enabled only if **Mechanism** is **Kerberos**.
Realm	The realm of the Hive Server 2 host. It is not required to specify any value in this field if the realm of the Hive Server 2 host is defined as the default realm in Kerberos configuration. This field is enabled only if **Mechanism** is **Kerberos**.
Thrift Transport **Since** v5.5.0/2	The transport protocol to use in the Thrift layer. This field is enabled only if **Hive Server Type** is **Hive Server 2**. Available options: • **Binary** (This option is available only if **Mechanism** is **No Authentication** or **User Name and Password**.) • **SASL** (This option is available only if **Mechanism** is **User Name** or **User Name and Password** or **Kerberos**.) • **HTTP** (This option is not available if **Mechanism** is **User Name**.) For information about determining which Thrift transport protocols your Hive server supports, refer to HiveServer2 Overview and Setting Up HiveServer2 sections in Hive documentation.
HTTP Path **Since** v5.5.0/2	The partial URL corresponding to the Hive server. This field is enabled only if **Thrift Transport** is **HTTP**.
Linux / Unix	

Driver Manager Library	The directory path where the Unix ODBC Driver Manager Library is installed. **Example:** /opt/unixodbc-2.3.2/lib
ODBCSYSINI	The directory path where **odbc.ini** and **odbcinst.ini** files are located. **Example:** /opt/unixodbc-2.3.2/etc
ODBC Driver	The user defined (installed) ODBC driver to connect HVR to the Hive server.
SSL Options	Show **SSL Options**.

SSL Options



Field	Description
Enable SSL	Enable/disable (one way) SSL. If enabled, HVR authenticates the Hive server by validating the SSL certificate shared by the Hive server.
Two-way SSL	Enable/disable two way SSL. If enabled, both HVR and Hive server authenticate each other by validating each others SSL certificate. This field is enabled only if **Enable SSL** is selected.
Trusted CA Certificates	The directory path where the **.pem** file containing the server's public SSL certificate signed by a trusted CA is located. This field is enabled only if **Enable SSL** is selected.
SSL Public Certificate	The directory path where the **.pem** file containing the client's SSL public certificate is located. This field is enabled only if **Two-way SSL** is selected.
SSL Private Key	The directory path where the **.pem** file containing the client's SSL private key is located. This field is enabled only if **Two-way SSL** is selected.
Client Private Key Password	The password of the private key file that is specified in **SSL Private Key**. This field is enabled only if **Two-way SSL** is selected.

Permissions

To run a capture or integration with Amazon S3 location, it is recommended that the AWS User has the AmazonS3FullAccess permission policy. AmazonS3ReadOnlyAccess policy is enough only for capture locations, which have a **LocationProperties /StateDirectory** defined. The minimal permission set for integrate location are:

- *s3:GetBucketLocation*
- *s3:ListBucket*
- *s3:ListBucketMultipartUploads*
- *s3:AbortMultipartUpload*
- *s3:GetObject*
- *s3:PutObject*
- *s3:DeleteObject*

Sample JSON file with a user role permission policy for S3 location

```
{
  "Statement": [
     {
          "Effect":"Allow",
          "Action":[
               "s3:PutObject",
               "s3:GetObject",
               "s3:GetObjectVersion",
               "s3:DeleteObject",
               "s3:DeleteObjectVersion"
               "s3:ListBucketMultipartUploads",
               "s3:AbortMultipartUpload"
           ],
          "Resource":"arn:aws:s3:::s3_bucket/directory_path/*"
     },
     {
          "Effect":"Allow",
          "Action":"s3:ListBucket",
          "Resource":"arn:aws:s3:::s3_bucket",
          "Condition":{
               "StringLike":{
                    "s3:prefix":[
                         "directory_path/*"
                     ]
                }
           }
       }
   ]
}
```

For more information on the Amazon S3 permissions policy, refer to the AWS S3 documentation.

S3 Encryption

HVR supports client or server-side encryption for uploading files into S3 locations. To enable the client or server-side encryption for S3, see action **LocationProperties /S3Encryption**.

AWS China

For enabling HVR to interact with AWS China cloud, define the **Environment** variable **HVR_AWS_CLOUD** with value **CHINA** on the hub and remote machine.

> S3 encryption with Key Management Service (KMS) is not supported in the AWS China cloud.

Hive External Table

HVR uses the Hive ODBC driver to connect to Hive for creating Hive external tables above S3 to perform hvrcompare with files that reside on S3. The Hive ODBC connection can be enabled for S3 in the location creation screen by selecting the **Hive External Tables** field (see section Location Connection).

ODBC Connection

HVR uses an ODBC connection to the Amazon EMR cluster for which it requires the ODBC driver (Amazon ODBC 1.1.1 or HortonWorks ODBC 2.1.2 and above) for Hive installed on the machine (or in the same network). On Linux, HVR additionally requires unixODBC 2.3.0 or later.

The Amazon and HortonWorks ODBC drivers are similar and compatible to work with Hive 2.x release. However, it is recommended to use the Amazon ODBC driver for Amazon Hive and the Hortonworks ODBC driver for HortonWorks Hive.

By default, HVR uses Amazon ODBC driver for connecting to Hadoop. Since HVR 5.3.1/25.1, use the **ODBC Driver** field available in the **New Location** screen to select the (user installed) Hortonworks ODBC driver. Prior to HVR 5.3.1 /25.1, to use the Hortonworks ODBC driver the following action definition is required:

For Linux

Group	Table	Action
S3	*	**Environment /Name=HVR_ODBC_CONNECT_STRING_DRIVER /Value=Hortonworks Hive ODBC Driver 64-bit**

For Windows

Group	Table	Action
S3	*	**Environment /Name=HVR_ODBC_CONNECT_STRING_DRIVER /Value=Hortonworks Hive ODBC Driver**

Amazon does not recommend changing the security policy of the EMR. This is the reason why it is required to create a tunnel between the machine where the ODBC driver is installed and the EMR cluster. On Linux, Unix and macOS you can create the tunnel with the following command:

```
ssh  -i  ~/mykeypair.pem  -N  -L  8157:ec2-###-##-##-###.compute-1.amazonaws.com:8088
hadoop@ec2-###-##-##-###.compute-1.amazonaws.com
```

Channel Configuration

For the file formats (CSV, JSON, and AVRO) the following action definitions are required to handle certain limitations of the Hive deserialization implementation during Bulk or Row-wise **Compare**:

- For CSV

Group	Table	Action
S3	*	**FileFormat /NullRepresentation=\\N**
S3	*	**TableProperties /CharacterMapping="\x00>\\0;\n>\\n;\r>\\r;">\""**
S3	*	**TableProperties /MapBinary=BASE64**

- For JSON

Group	Table	Action
S3	*	**TableProperties** /MapBinary=BASE64
S3	*	**FileFormat /JsonMode=ROW_FRAGMENTS**

- For Avro

Group	Table	Action
S3	*	**FileFormat /AvroVersion=v1_8**

v1_8 is the default value for **FileFormat /AvroVersion**, so it is not mandatory to define this action.

Requirements for Salesforce

Contents

This section describes the requirements, access privileges, and other features of HVR when using Salesforce for replication. For information about compatibility and supported versions of Salesforce with HVR platforms, see Platform Compatibility Matrix.

For the Capabilities supported by HVR on Salesforce, see Capabilities for Salesforce.

To quickly setup replication into Salesforce, see Quick Start for HVR - Salesforce.

Prerequisites

- HVR requires that the Salesforce Data Loader is installed to establish connection with Salesforce. The location of the Data Loader is supplied with the Salesforce location (e.g. **C:\Program Files\salesforce.com\Data Loader\Dataloader.jar**).

  > HVR uses Salesforce Data Loader to capture and integrate changes into Salesforce. So the capture from Salesforce is neither Log-based nor Trigger-based.

  > Since Salesforce Data Loader is supported only for Windows and MacOS, HVR supports Salesforce only on Windows.

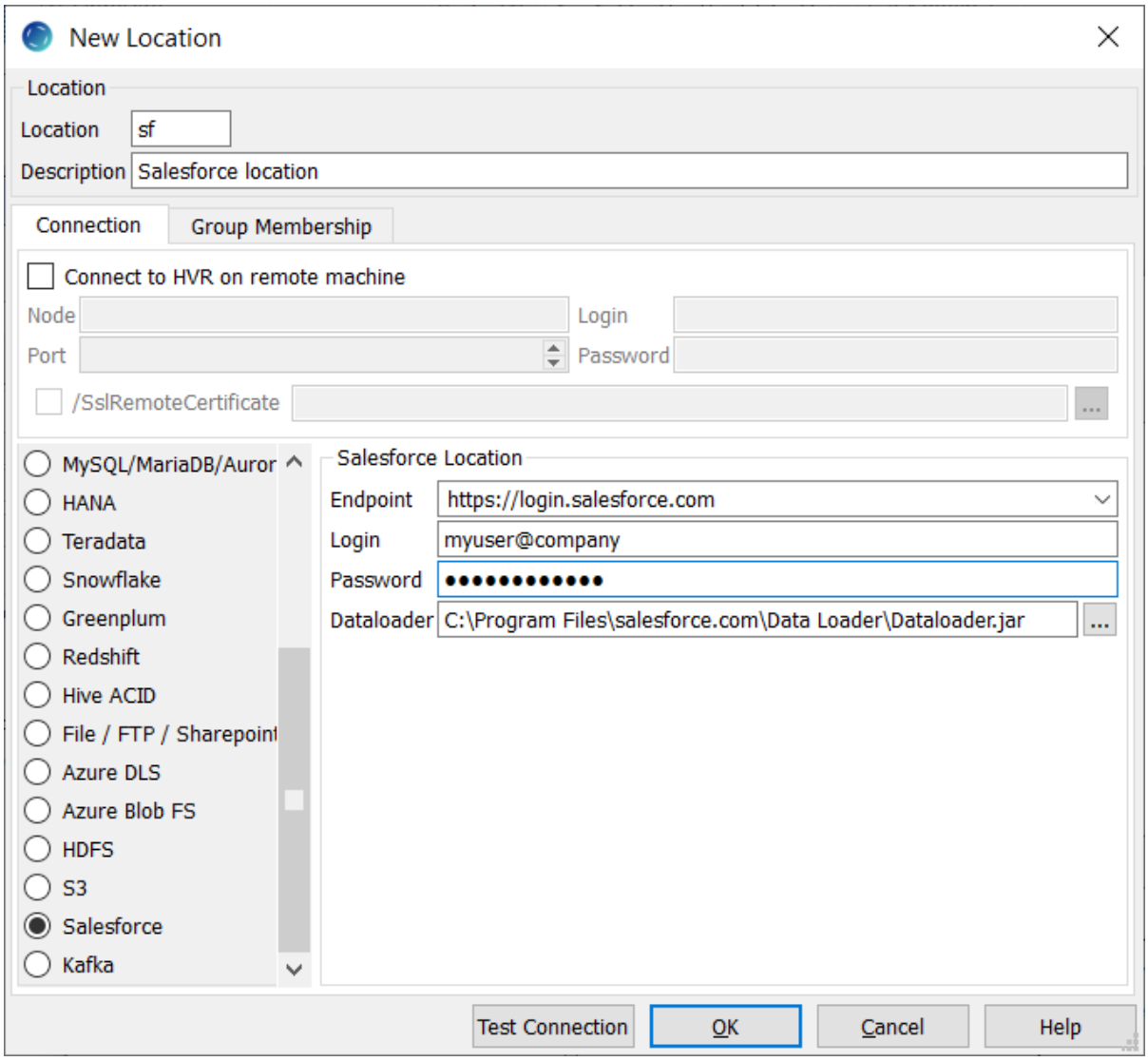
- Java2SE or Java2EE version 5 or higher must be installed. If Java is not in the system **PATH**, then the environment variable **$JAVA_HOME** must be set to the Java installation directory using action **Environment** . Ensure that this is the directory that has a bin folder, e.g. if the Java bin directory is d:\java\bin, **$JAVA_HOME** should point to d:\java.

  > Data Loader version 45 and above requires Zulu OpenJDK whereas prior versions required Java Runtime Environment (JRE).
  >
  > If JRE related errors are encountered, ensure that the latest version of Data Loader is installed.

- HVR can either connect to Salesforce directly from the hub machine, or it first connects to a remote machine with HVR's own remote protocol and then connects to Salesforce from that machine. A proxy server can be configured with action **LocationProperties /Proxy**.

Location Connection

This section lists and describes the connection details required for creating Salesforce location. HVR uses Salesforce Dataloader tool to connect to the Salesforce location.



Field	Description
Salesforce Location	
Endpoint	The URL to connect to Salesforce. **Example:** https://login.salesforce.com/
Login	The Salesforce login name to connect HVR to Salesforce. **Example:** myuser@company
Password	The password for **Login** to connect HVR to Salesforce.
Dataloader	The directory path where the **dataloader.jar** file is located. **Example:** C:\Program Files\salesforce.com\Data Loader\Dataloader.jar

Capture

HVR supports capture from Salesforce location. HVR uses Salesforce Dataloader tool to capture changes from a Salesforce location. This section describes the configuration requirements/notes for **capturing** changes from Salesforce location.

- Rows can be read from Salesforce locations using action **Capture** and integrated into any database location. They can also be sent to a file location where they are written as XML files.

- A capture job reads all rows from the Salesforce table instead of just detecting changes. This means that the capture job should not be scheduled to run continuously. Instead, it can be run manually or periodically with action **Scheduling /CaptureStartTimes**.
- A channel with **Capture** must have table information defined; it cannot be used with a 'blob' file channel. The Salesforce 'API names' for tables and columns are case–sensitive and must match the 'base names' in the HVR channel. This can be done by defining **TableProperties /BaseName** actions on each of the capture tables and **ColumnProperties /BaseName** actions on each column.
- A capture restriction can be defined on a Salesforce location in Salesforce Object Query Language (SOQL) using action **Restrict /CaptureCondition**.
- All rows captured from Salesforce are treated as inserts (**hvr_op=1**). Deletes cannot be captured.
- Salesforce locations can only be used for replication jobs; **HVR Refresh** and **HVR Compare** are not supported.

Integrate and Refresh

HVR uses Salesforce Dataloader tool to write data to a Salesforce location during **Integrate** (continuous) and **Refresh** (bulk).

Requirements for SapXForm

Contents

HVR's SapXForm feature allows capturing changes from SAP "cluster" and "pool" tables and replicate into target database as "unpacked" data. For example, HVR can capture the SAP cluster table (**RFBLG**) from an Oracle based SAP system and unpack the contents (**BSEG**, **BSEC**) of the cluster table into a Redshift database; the HVR pipeline does the 'unpacking' dynamically. An essential component of the SapXForm feature is the SapXForm engine executable.

For using SapXForm with HVR 4.7.x, see Configuring SapXForm with HVR 4.7.

SAP System

SAP database typically contains tables that fall into one of the following three categories:

- Transparent tables - ordinary database tables which can be replicated in a usual way.
- Pooled and Cluster tables - are special in that the data for several Pooled or Cluster tables are grouped and physically stored together in a single database table. HVR uses "SAP Transform Engine" to extract and replicate individual tables from SAP table pools and clusters.
- SAP Catalogs - contain metadata and do not usually need to be replicated. HVR and SAP Transform Engine themselves, however, need data from SAP Catalogs for the purpose of Pooled and Cluster tables processing.
  HVR SapXForm supports capturing changes from SAP system with either of the databases Oracle, DB2i, HANA and SQL Server.

To enable replication from SAP database using SapXForm, ensure that the SAP Dictionary tables (**DD02L**, **DD03L**, **DD16S**, **DDNTF**, **DDNTT**) exist in the source database. HVR uses the information available in SAP dictionary for unpacking data from SAP pool and cluster tables.

HVR unpacks all pool tables available in pack tables **ATAB** or **KAPOL**.

HVR also unpacks tables identified as cluster in the SAP dictionary. Examples of such tables are **BSEG** and **BSEC** which are packed inside **RFBLG**.

There are tables that SAP does not identify in its dictionary as "cluster tables" even though the tables contain clustered data. These are not supported. Examples include **PCL1**, **PCL2**, **MDTC** and **STXL**.

HVR License for SapXForm

The regular HVR license file does not enable the SapXForm feature, so an additional license is needed. Contact HVR Support to obtain the proper SapXForm license. Place the license file (e.g. **hvr.lic** and **hvrsapx.lic**) in **hvr_home/lib**.

HVR supports "accumulating" license files; this mean a hub machine could have several license files. For example, one license file (**hvr.lic**) enables all features (except SapXForm) of HVR to be used perpetually and another license file (**hvrsap.lic**) enables SapXForm feature.

> If a valid license to use SapXForm is not available in the **hvr_home/lib** directory then the option to query the **SAP Dictionaries** in **Table Explore** will not be displayed in the HVR GUI.

Requirements for SAP Transform Engine

SAP Transform Engine is special executable which is shipped as part of the HVR distribution, and is licensed separately. The SAP Transform Engine can be run on Windows and Linux platforms; it can be configured to execute either on the HVR hub machine or the integrate machines.

SAP Transform Engine on Windows

In Windows, the SAP Transform Engine requires .NET Framework version 4.5 or higher to be installed in the system.

> To verify the version of .NET Framework installed in the system, see Microsoft Documentation.

SAP Transform Engine on Linux

In Linux, the SAP Transform Engine requires Mono version 4.4 or higher to be installed in the system. Mono is an open source implementation of Microsoft's .NET Framework.

- For newer Linux versions (EPEL 6, EPEL 7, Debian 7, Debian 8), Mono can be downloaded and installed from Mono Downloads.
- For older Linux version (EPEL 5) or to use as an alternative method for EPEL 6 and EPEL 7, the installation for Mono is not available in the Mono Downloads.
  However, the Mono installation files for EPEL 5 can be obtained from a community managed repository - https://copr.fedorainfracloud.org/coprs/tpokorra/mono. This repository contains Mono-community built packages which install into **/opt** for CentOS (5,6,7). These are automatically compiled from original sources.
  1. Click **Packages** tab.
  2. Click **mono-opt** package ("mono installed in opt").
  3. Click on the latest successful **Mono 4.6.***x* build (for example **4.6.2-1**).
  4. In the **Results** pane, click **epel-5-x86_64**.
  5. Click **mono-opt-4.6.***x***.***x***.x86_64.rpm** to download the RPM file.

Verifying Mono Installation

The Mono installation directory contains a script file **env.sh**. This script is intended to be included before running Mono, and is used to configure host specific environment. To verify if the installed Mono is working, perform the following steps (assuming Mono is installed in **/opt/mono**) :

1. Start a new clean shell.
2. Execute the following command to include the mono environment setup file into current shell:

   ```
   source /opt/mono/env.sh
   ```

3. Execute the following command to view Mono:

   ```
   mono --version
   ```

4. If any error such as library error is displayed, it indicates incorrect installation/configuration of Mono. Verify the parameters defined for configuring the host specific environment in **/opt/mono/env.sh** and rectify them, if required.
5. Repeat this verification procedure in a new clean shell.

Configuring SapXForm with HVR 4

Contents

SAP database typically contains tables that fall into one of the following three categories:

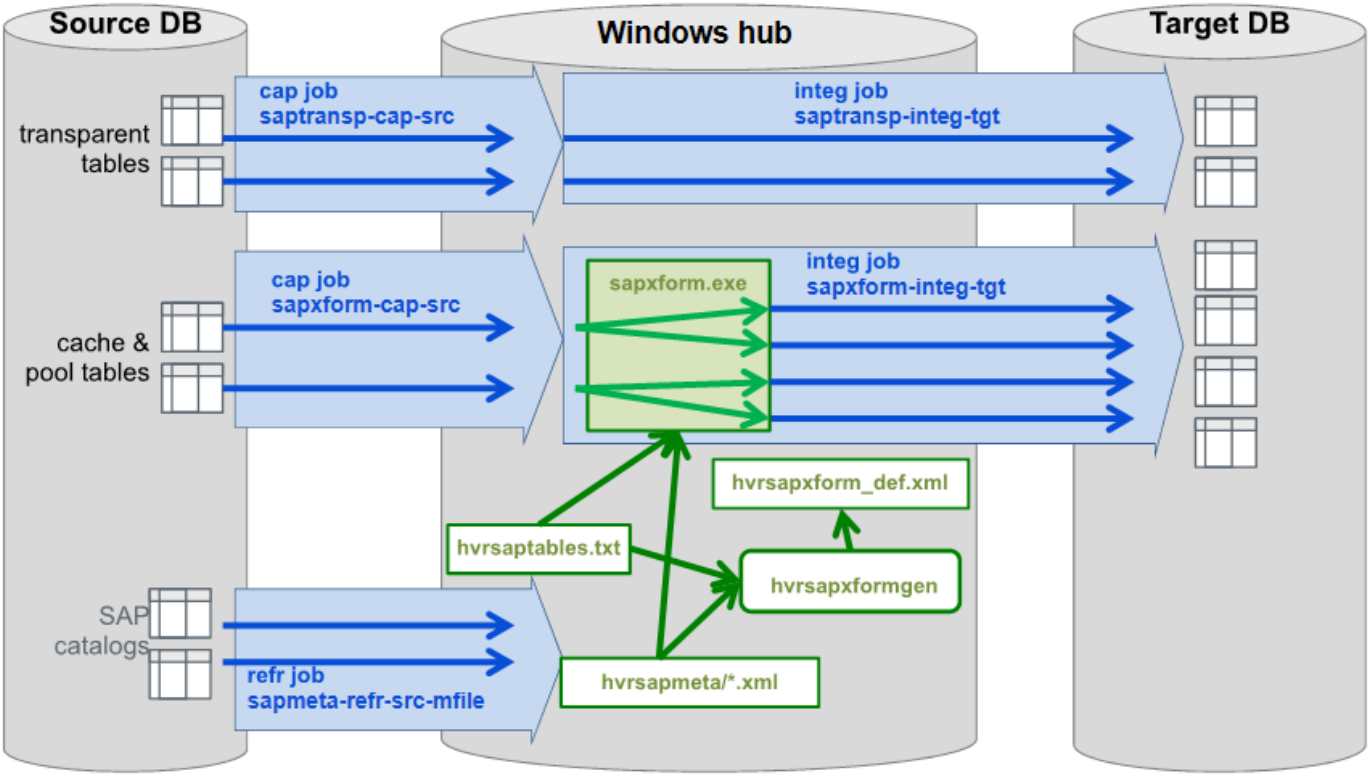
- Transparent Tables
  Transparent tables are just an ordinary database tables, and can be replicated in a usual way.
- Pooled and Cluster tables
  Pooled and Cluster tables, are special in that the data for several Pooled or Cluster tables is grouped and physically stored together in a single database table. With a help of a third-party "SAP Transform" Engine, HVR is capable to extract and replicate individual tables from SAP table pools and clusters.

  > SAP Transform Engine is a third-party product. It is not shipped as part of the HVR, and is licensed separately.

- SAP Catalogs
  SAP Catalogs only contain metadata and do not usually need to be replicated. HVR and SAP Transform Engine themselves, however, need data from SAP Catalogs for the purpose of Pooled and Cluster tables processing.

The following diagram shows general architecture of the HVR SAP Transform solution. In a typical SAP database replication scenario, three channels need to be setup:

- **saptransp**: a channel to replicate the Transparent tables.
- **sapmeta**: a channel to extract metadata out of SAP Catalog tables. This data is required to build and run *sapxform* channel.
- **sapxform**: a channel to extract (transform) and replicate Pooled and Cluster tables.



Prerequisites

In this document the reader is assumed to be familiar with the basic HVR concepts. We also assume that the hub database already exist, and source and destination database locations are already configured.

HVR hub should be installed on Windows. This requirement is due to SAP Transform Engine can only be run on a Windows platform, and HVR requires that the SAP Transform Engine is installed on a hub machine. The SAP Transform Engine also requires .NET Framework version 4.5 to be installed.

Before you begin, all SAP tables that need to be replicated should be identified. The Pooled and Cluster tables should be listed in a plain-text file, each table on a separate line. We will refer to this file further as **hvrsaptables.txt**. Empty lines are ignored, lines beginning with a hash mark (#) are considered comments and are also ignored.

An example of the **hvrsaptables.txt** follows:

```
#SAP tables needed for HVR's sapxform channel

BSEC   # Cluster table
BSEG   # Cluster table
BSET   # Cluster table
T009B  # Pool table
```

Transparent tables should be listed separately from the Pooled and Cluster tables.

Installing SAP Transform Engine

SAP Transform Engine is distributed separately from HVR. Once you have obtained a copy of the SAP Transform Engine, copy the executable file **sapxform.exe** to the **$HVR_HOME/lib/transform** directory.

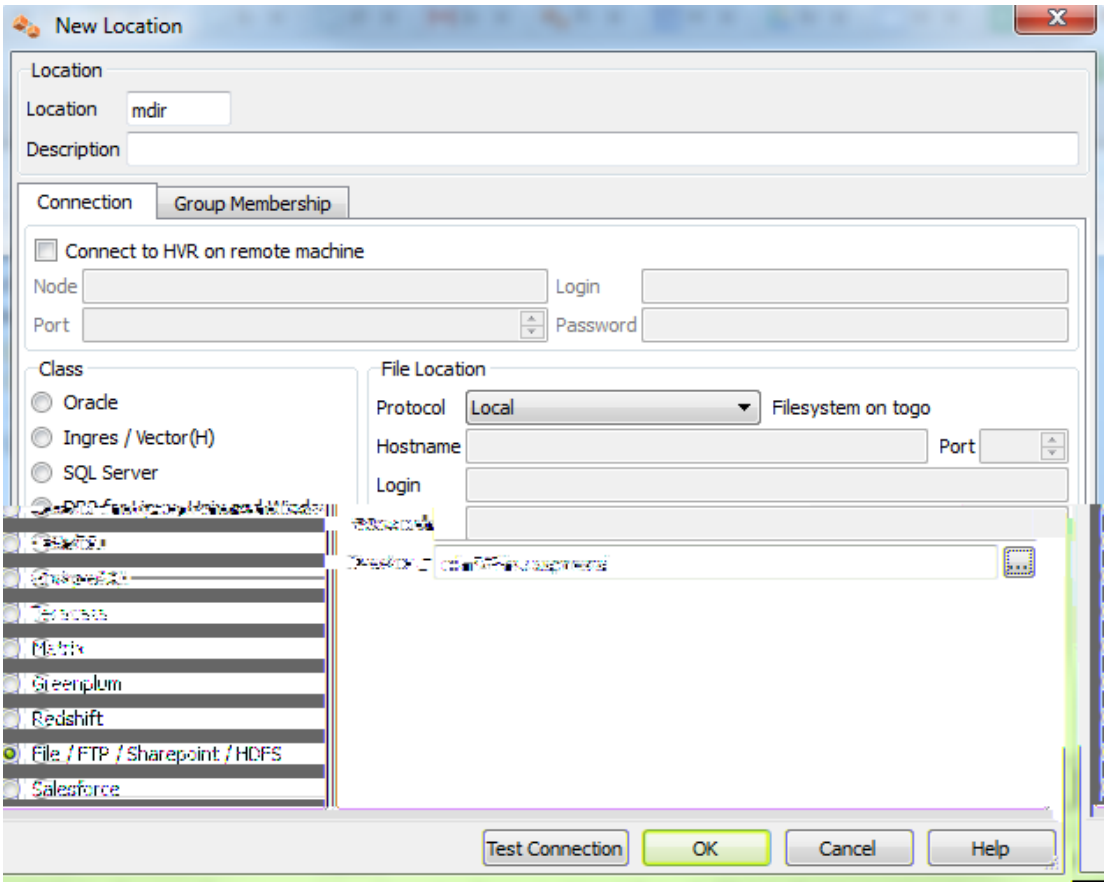
Setting up Transparent Channel : saptransp

A Transparent channel is an ordinary database to database one-way replication channel. We will not focus on creating the Transparent channel in details in this document.

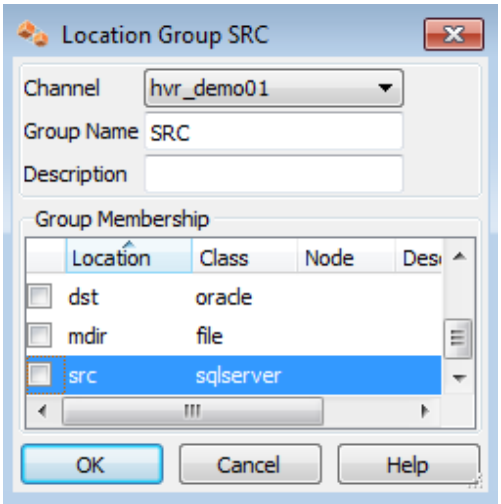
Setting up Meta-data Channel : sapmeta

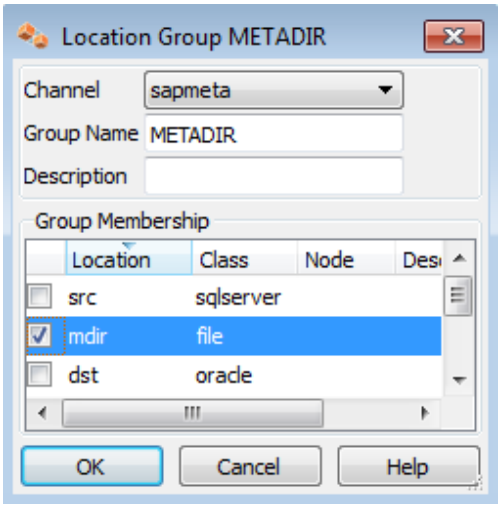
SAP Catalog data is required to build and run the SAP XForm channel. A special Meta channel is used to extract SAP Catalog data into a set of XML files. To set up the Meta channel the following actions are required.

First, a **File** Location should be create, pointing to a directory where SAP Catalog data will be stored.



Once the new location is added, a predefined Meta channel definition should be imported. To do this, right-click on the hub name, choose **Import Catalogs...**, navigate to **$HVR_HOME/demo/sapxform** and choose **sapmeta_def.xml** file. A new channel called **sapmeta** will be added to the hub. This channel contains two Location Groups. Add your Source database location to the **SRC** Location Group. Then add the File Location (created on the previous step) to the **METADIR** Location Group.





As a last step, execute **HVR Refresh** in this channel.

Setting up Transform Channel : sapxform

Setting up SAP XForm channel is a complex task which cannot be performed manually. Instead a special script **hvrsapxformgen** is provided by HVR to generate the SAP XForm channel. This script takes as parameters the file with a list of the Pooled and Cluster tables (**hvrsaptables.txt** below) and a path to the directory the SAP Catalog data is extracted to (**C:\HVR\hvrsapmeta** below). It should be invoked from the OS command line as follow:

```
c:\>    hvrsapxformgen    sapxform_hub/password    sapxform    hvrsaptables.txt    C:
\HVR\hvrsapmeta               # Oracle
 or
c:\>  hvrsapxformgen  -u  hvr_user/password  sapxform_hub  sapxform  hvrsaptables.txt  C:
\HVR\hvrsapmeta    # SQL Server
```

When this channel is generated, users of the **HVRGUI** must use **Reload** their hub database, After this, the channel can be setup normally using HVR Refresh and HVR Initialize.

Requirements for Snowflake

Since v5.2.3/16

Contents

This section describes the requirements, access privileges, and other features of HVR when using Snowflake for replication. For information about compatibility and supported versions of Snowflake with HVR platforms, see Platform Compatibility Matrix.

For the Capabilities supported by HVR on Snowflake, see Capabilities for Snowflake.

For information about the supported data types and mapping of data types in source DBMS to the corresponding data types in target DBMS or file format, see Data Type Mapping.

To quickly setup replication using Snowflake, see Quick Start for HVR - Snowflake.

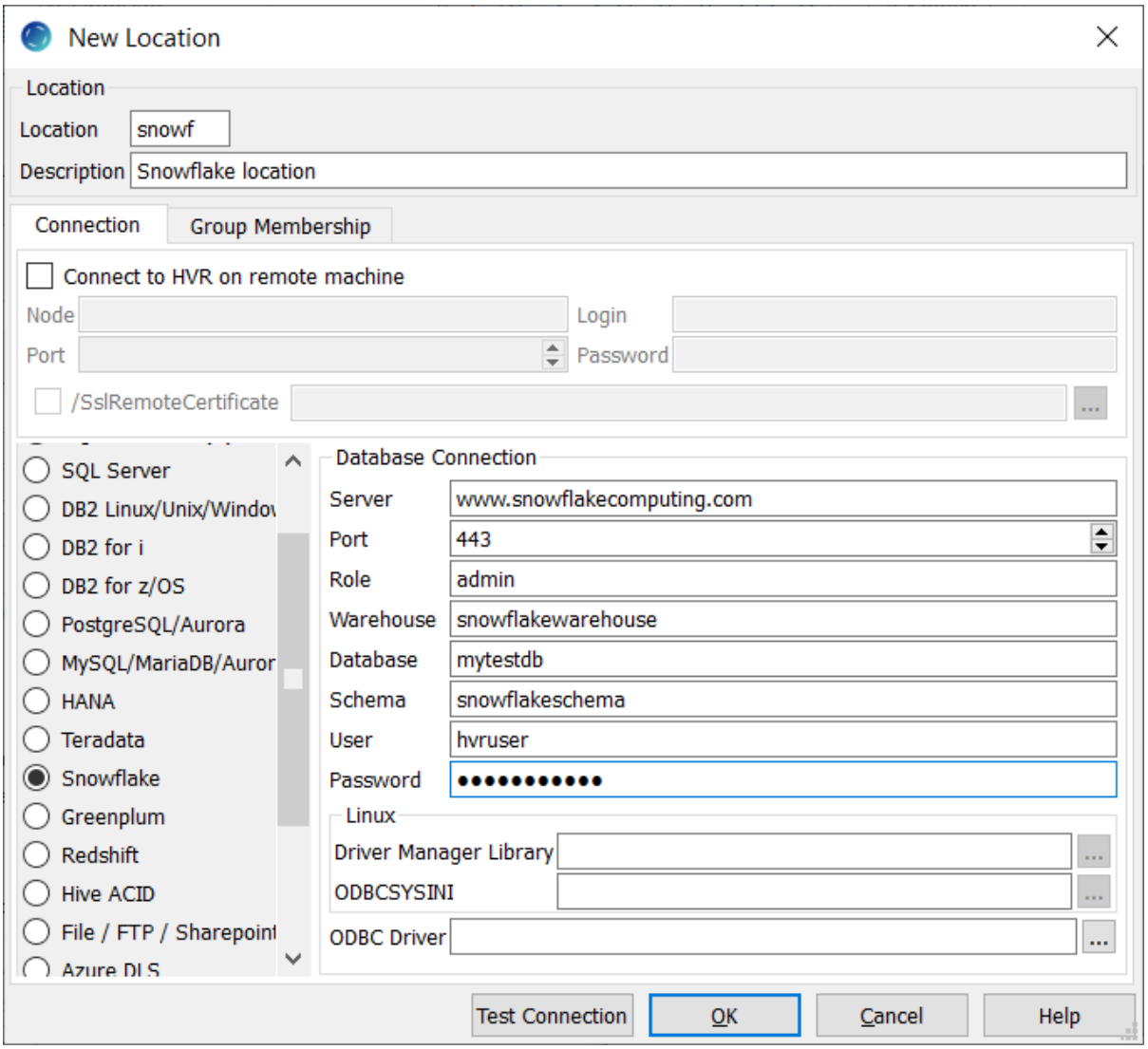
ODBC Connection

HVR requires that the Snowflake ODBC driver is installed on the machine from which HVR connects to Snowflake. For more information on downloading and installing Snowflake ODBC driver, see Snowflake Documentation.

⚠ After installing the Snowflake ODBC driver, configure the **LogLevel** configuration parameter as specified in ODBC Configuration and Connection Parameters of the Snowflake Documentation.

Location Connection

This section lists and describes the connection details required for creating Snowflake location in HVR.



Field	Description
Database Connection	
Server	The hostname or ip-address of the machine on which the Snowflake server is running. **Example:** www.snowflakecomputing.com
Port	The port on which the Snowflake server is expecting connections. **Example:** 443
Role	The name of the Snowflake role to use. **Example:** admin
Warehouse	The name of the Snowflake warehouse to use. **Example:** snowflakewarehouse
Database	The name of the Snowflake database. **Example:** mytestdb
Schema	The name of the default Snowflake schema to use. **Example:** snowflakeschema
User	The username to connect HVR to the Snowflake **Database**. **Example:** hvruser
Password	The password of the **User** to connect HVR to the Snowflake **Database**.
Linux / Unix	

Driver Manager Library	The optional directory path where the ODBC Driver Manager Library is installed. For a default installation, the ODBC Driver Manager Library is available at **/usr/lib64** and does not need to specified. When UnixODBC is installed in for example **/opt /unixodbc-2.3.1** this would be **/opt/unixodbc-2.3.1/lib**.
ODBCSYSINI	The directory path where **odbc.ini** and **odbcinst.ini** files are located. For a default installation, these files are available at **/etc** and does not need to be specified. When UnixODBC is installed in for example **/opt/unixodbc-2.3.1** this would be **/opt /unixodbc-2.3.1/etc**. The **odbcinst.ini** file should contain information about the Snowflake ODBC Driver under the heading **[SnowflakeDSIIDriver]**.
ODBC Driver	The user defined (installed) ODBC driver to connect HVR to the Snowflake server.

Integrate and Refresh Target

HVR supports integrating changes into Snowflake location. This section describes the configuration requirements for integrating changes (using **Integrate** and **refresh**) into Snowflake location. For the list of supported Snowflake versions, into which HVR can integrate changes, see Integrate changes into location in Capabilities.

HVR uses the Snowflake ODBC driver to write data to Snowflake during continuous **Integrate** and row-wise **Refresh** . However, the preferred methods for writing data to Snowflake are **Integrate** with **/Burst** and Bulk **Refresh** using staging as they provide better performance (see section 'Burst Integrate and Bulk Refresh' below).

⚠ When performing the **refresh** operation using slicing (option **-S**), a refresh job is created per each slice for refreshing only rows contained in the slice. These refresh jobs must not be run in parallel but should be scheduled one after another to avoid a risk of corruption on a Snowflake target location.

Grants for Integrate and Refresh Target

- The **User** should have permission to read and change replicated tables.

```
grant all privileges on future tables in database database to role hvrrole;
grant select, insert, update, delete, truncate on future tables in database
database to role hvrrole;
```

- The **User** should have permission to create and drop HVR state tables.
- The **User** should have permission to create and drop tables when HVR Refresh will be used to create target tables.

```
grant usage, modify, create table on schema schema in database database to role
hvrrole
```

Burst Integrate and Bulk Refresh

While **Integrate** is running with parameter **/Burst** and Bulk **Refresh**, HVR can stream data into a target database straight over the network into a bulk loading interface specific for each DBMS (e.g. direct-path-load in Oracle), or else HVR puts data into a temporary directory (‘staging file') before loading data into a target database. For more information about staging files on Snowflake, see Snowflake Documentation.

For best performance, HVR performs **Integrate** with **/Burst** and Bulk **Refresh** into Snowflake using staging files. HVR implements **Integrate** with **/Burst** and Bulk **Refresh** (with file staging) into Snowflake as follows:

1. HVR first stages data into the configured staging platform:

    - Snowflake Internal Staging using Snowflake ODBC driver (**default**) **Since** v5.6.5/12
    - AWS or Google Cloud Storage using cURL library
    - Azure Blob FS using HDFS-compatible libhdfs API
2. HVR then uses Snowflake SQL command '**copy into**' to ingest data from the staging directories into the Snowflake target tables

HVR supports the following cloud platforms for staging files:

- Snowflake Internal Staging
- Snowflake on AWS
- Snowflake on Azure
- Snowflake on Google Cloud Storage

Snowflake Internal Staging

Since v5.6.5/12

By default, HVR stages data on the Snowflake internal staging before loading it into Snowflake while performing **Integrate** with **Burst** and **Bulk Refresh**. To use the Snowflake internal staging, it is not required to define action **LocationProperties** on the corresponding Integrate location.

Snowflake on AWS

HVR can be configured to stage the data on AWS S3 before loading it into Snowflake. For staging the data on AWS S3 and perform **Integrate** with **Burst** and **Bulk Refresh**, the following are required:

1. An AWS S3 location (bucket) - to store temporary data to be loaded into Snowflake. For more information about creating and configuring an S3 bucket, refer to AWS Documentation.
2. An AWS user with 'AmazonS3FullAccess' policy - to access this location. For more information, refer to the following AWS documentation:
    - Amazon S3 and Tools for Windows PowerShell
    - Managing Access Keys for IAM Users
    - Creating a Role to Delegate Permissions to an AWS Service
3. Define action **LocationProperties** on the Snowflake location with the following parameters:
    - **/StagingDirectoryHvr**: the location where HVR will create the temporary staging files (ex. **s3://my_bucket_name/**).
    - **/StagingDirectoryDb**: the location from where Snowflake will access the temporary staging files. If **/StagingDirectoryHvr** is an Amazon S3 location then the value for **/StagingDirectoryDb** should be same as **/StagingDirectoryHvr**.
    - **/StagingDirectoryCredentials**: the AWS security credentials. The supported formats are **'aws_access_key_id="***key***";aws_secret_access_key="***secret_key***"'** or **'role="***AWS_role***"'**. How to get your AWS credential or Instance Profile Role can be found on the AWS documentation webpage.
4. If the S3 bucket used for the staging directory does not reside in the default **us-east-1** region, the region of the S3 bucket (e.g **eu-west-2** or **ap-south-1**) must be explicitly specified. To set the S3 bucket region, define the following action on the Snowflake location:

Snowflake	*	**Environment /Name**=**HVR_S3_BOOTSTRAP_REGION /Value**=*s3_bucket_region*

Snowflake on Azure

Since v5.5.5/4

HVR can be configured to stage the data on Azure BLOB storage before loading it into Snowflake. For staging the data on Azure BLOB storage and perform **Integrate** with **Burst** and **Bulk Refresh**, the following are required:

1. An Azure BLOB storage location - to store temporary data to be loaded into Snowflake
2. An Azure user (storage account) - to access this location. For more information, refer to the Azure Blob storage documentation.
3. Define action **LocationProperties** on the Snowflake location with the following parameters:

- **/StagingDirectoryHvr**: the location where HVR will create the temporary staging files (e.g. **wasbs://myblobcontainer**).
- **/StagingDirectoryDb**: the location from where Snowflake will access the temporary staging files. If **/StagingDirectoryHvr** is an Azure location, this parameter should have the same value.
- **/StagingDirectoryCredentials**: the Azure security credentials. The supported format is **"azure_account=***azure_account***;azure_secret_access_key=***secret_key***"**.

4. Hadoop client should be present on the machine from which HVR will access the Azure Blob FS. Internally, HVR uses the WebHDFS REST API to connect to the Azure Blob FS. Azure Blob FS locations can only be accessed through HVR running on Linux or Windows, and it is not required to run HVR installed on the Hadoop NameNode although it is possible to do so. For more information about installing Hadoop client, refer to Apache Hadoop Releases.

Hadoop Client Configuration

The following are required on the machine from which HVR connects to Azure Blob FS:

- Hadoop 2.6.x client libraries with Java 7 Runtime Environment or Hadoop 3.x client libraries with Java 8 Runtime Environment. For downloading Hadoop, refer to Apache Hadoop Releases.
- Set the environment variable **$JAVA_HOME** to the Java installation directory. Ensure that this is the directory that has a bin folder, e.g. if the Java bin directory is d:\java\bin, **$JAVA_HOME** should point to d:\java.
- Set the environment variable **$HADOOP_COMMON_HOME** or **$HADOOP_HOME** or **$HADOOP_PREFIX** to the Hadoop installation directory, or the **hadoop** command line client should be available in the path.
- One of the following configuration is recommended,
  - Set **$HADOOP_CLASSPATH=$HADOOP_HOME/share/hadoop/tools/lib/***
  - Create a symbolic link for **$HADOOP_HOME/share/hadoop/tools/lib** in **$HADOOP_HOME /share/hadoop/common** or any other directory present in classpath.

> ⚠ Since the binary distribution available in Hadoop website lacks Windows-specific executables, a warning about unable to locate **winutils.exe** is displayed. This warning can be ignored for using Hadoop library for client operations to connect to a HDFS server using HVR. However, the performance on integrate location would be poor due to this warning, so it is recommended to use a Windows-specific Hadoop distribution to avoid this warning. For more information about this warning, refer to Hadoop Wiki and Hadoop issue HADOOP-10051.

Verifying Hadoop Client Installation

To verify the Hadoop client installation,

a. The **HADOOP_HOME/bin** directory in Hadoop installation location should contain the hadoop executables in it.
b. Execute the following commands to verify Hadoop client installation:

```
$JAVA_HOME/bin/java -version
$HADOOP_HOME/bin/hadoop version
$HADOOP_HOME/bin/hadoop classpath
```

c. If the Hadoop client installation is verified successfully then execute the following command to check the connectivity between HVR and Azure Blob FS:

> ⚠ To execute this command successfully and avoid the error "ls: Password fs.adl.oauth2.client.id not found", few properties needs to be defined in the file **core-site.xml** available in the hadoop configuration folder (for e.g., **<path>/hadoop-2.8.3/etc/hadoop**). The properties to be defined differs based on the **Mechanism** (authentication mode). For more information, refer to section 'Configuring Credentials' in Hadoop Azure Blob FS Support documentation.

```
$HADOOP_HOME/bin/hadoop fs -ls wasbs://containername@accountname.blob.core.
windows.net/
```

Verifying Hadoop Client Compatibility with Azure Blob FS

To verify the compatibility of Hadoop client with Azure Blob FS, check if the following JAR files are available in the Hadoop client installation location (**$HADOOP_HOME/share/hadoop/tools/lib**):

```
hadoop-azure-<version>.jar
azure-storage-<version>.jar
```

Snowflake on Google Cloud Storage

Since v5.6.5/7

HVR can be configured to stage the data on Google Cloud Storage before loading it into Snowflake. For staging the data on Google Cloud Storage and perform **Integrate** with **Burst** and **Bulk Refresh**, the following are required:

1. A Google Cloud Storage location - to store temporary data to be loaded into Snowflake
2. A Google Cloud user (storage account) - to access this location.
3. Configure the storage integrations to allow Snowflake to read and write data into a Google Cloud Storage bucket. For more information, see Configuring an Integration for Google Cloud Storage in Snowflake documentation.
4. Define action **LocationProperties** on the Snowflake location with the following parameters:
   - **/StagingDirectoryHvr**: the location where HVR will create the temporary staging files (e.g. **gs://mygooglecloudstorage_bucketname**).
   - **/StagingDirectoryDb**: the location from where Snowflake will access the temporary staging files. If **/StagingDirectoryHvr** is a Google cloud storage location, this parameter should have the same value.
   - **/StagingDirectoryCredentials**: Google cloud storage credentials. The supported format is **"gs_access_key_id=***key***;gs_secret_access_key=***secret_key***;gs_storage_integration=** *integration_name for google cloud storage***"**.

Compare and Refresh Source

- The **User** should have permission to read replicated tables.

```
grant select on tbl to hvruser
```

Requirements for SQL Server

Contents

This section describes the requirements, access privileges, and other features of HVR when using SQL Server for replication.

For the Capabilities supported by HVR on SQL Server, see Capabilities for SQL Server.

For information about the supported data types and mapping of data types in source DBMS to the corresponding data types in target DBMS or file format, see Data Type Mapping.

To quickly setup replication using SQL Server, see Quick Start for HVR - SQL Server.

Supported Editions

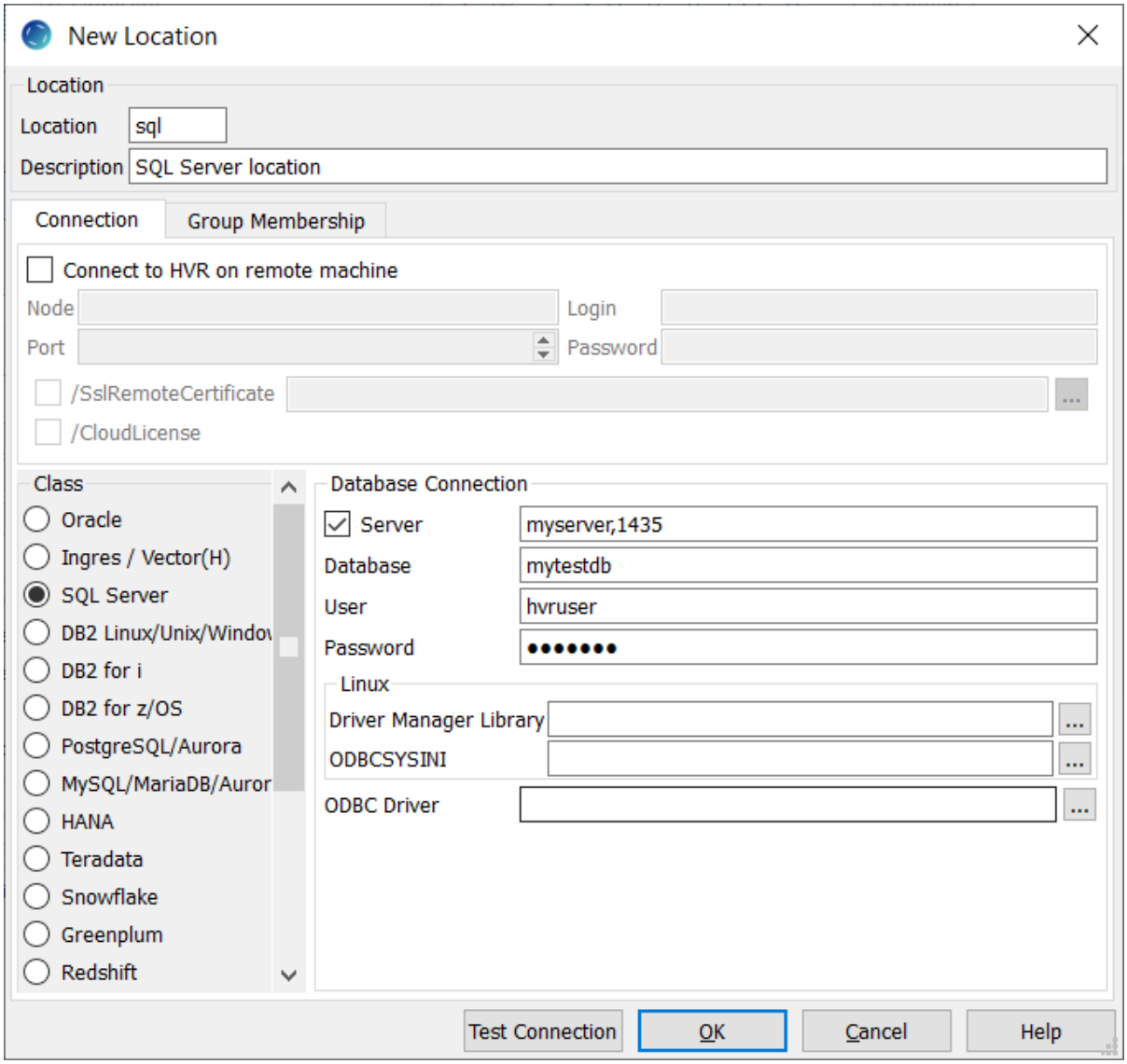
HVR supports the following SQL Server editions:

- SQL Server Developer Edition
- SQL Server Enterprise Edition
- SQL Server Standard Edition

For information about compatibility and supported versions of SQL Server with HVR platforms, see Platform Compatibility Matrix.

Location Connection

This section lists and describes the connection details required for creating SQL Server location in HVR.



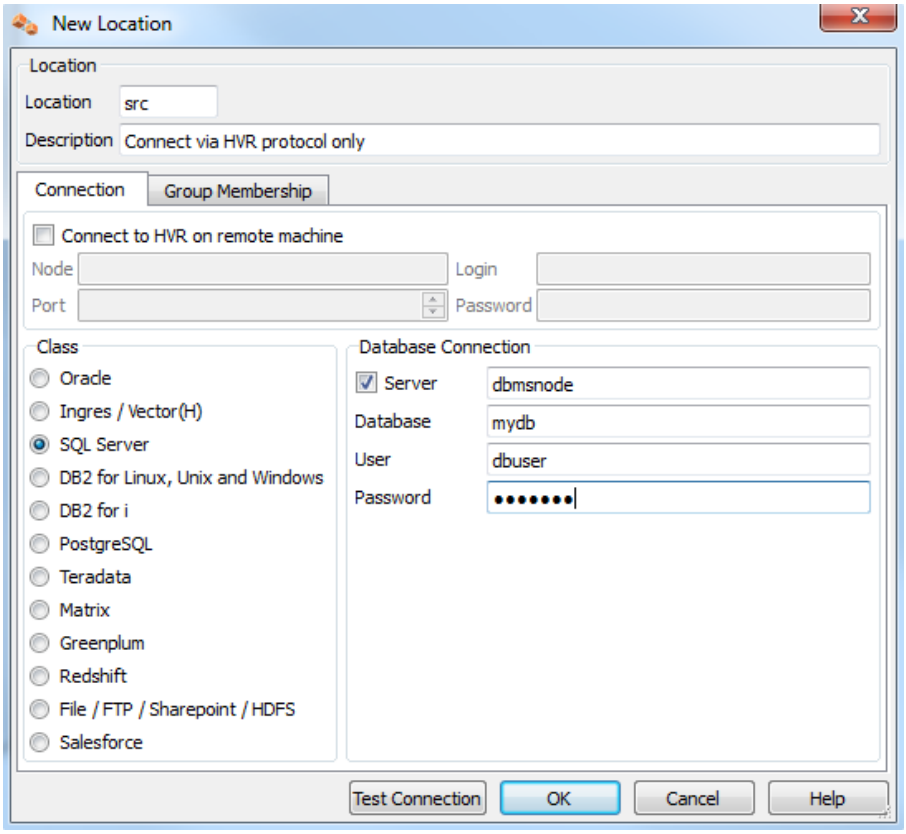
Field	Description
Server	The name of the machine on which SQL Server is running and the Port number or SQL Server instance name. The following formats are supported for this field: • server name : Specify only server name and HVR will automatically use the default port to connect to the server on which SQL Server is running. **Example:**myserver • *server name*,*port number* : Specify server name and port number separated by a comma (,) to connect to the server on which SQL Server is running. This format is required when using custom port for connection. **Example:** myserver,1435 • *server name**server instance name* : Specify server name and server instance name separated by a backslash (\) to connect to the server on which SQL Server is running. This format is not supported on Linux. Also, it is not supported when **Integrate /NoTriggerFiring** is to be defined for this location. **Example:** myserver\HVR5048 For more details on the connection methods, see Connecting HVR Hub to a Remote SQL Server Database.
Database	The name of the SQL Server database which is to be used for replication. **Example:** mytestdb

User	The username to connect HVR to SQL Server **Database**. This user should be defined with SQL Server Authentication or Windows Authentication. **Example:** hvruser If Windows Authentication is used for connecting to SQL Server **Database**, the **User** and **Password** field should be left blank (empty).
Password	The password of the **User** to connect HVR to SQL Server **Database**.
Linux	
Driver Manager Library	The directory path where the Unix ODBC Driver Manager Library is installed. For a default installation, the ODBC Driver Manager Library is available at **/usr/lib64** and does not need to specified. When UnixODBC is installed in for example **/opt /unixodbc-2.3.1** this would be **/opt/unixodbc-2.3.1/lib**
ODBCSYSINI	The directory path where **odbc.ini** and **odbcinst.ini** files are located. For a default installation, these files are available at **/etc** and does not need to be specified. When UnixODBC is installed in for example **/opt/unixodbc-2.3.1** this would be **/opt /unixodbc-2.3.1/etc**.
ODBC Driver	The user defined (installed) ODBC driver to connect HVR to the SQL Server **Database** . It is recommended to leave this field empty, HVR will automatically load the correct driver for your current platform. Otherwise, select one of the available **SQL Server Native Client** options.

Connecting HVR Hub to a Remote SQL Server Database

For connecting HVR hub machine to a remote SQL Server database, the following three methods are available:

All of the following connection options can be used for both **Capture** and **Integrate**. Specifically: HVR's log-based capture can get changes from a database without HVR's executables being physically installed on the source machine.

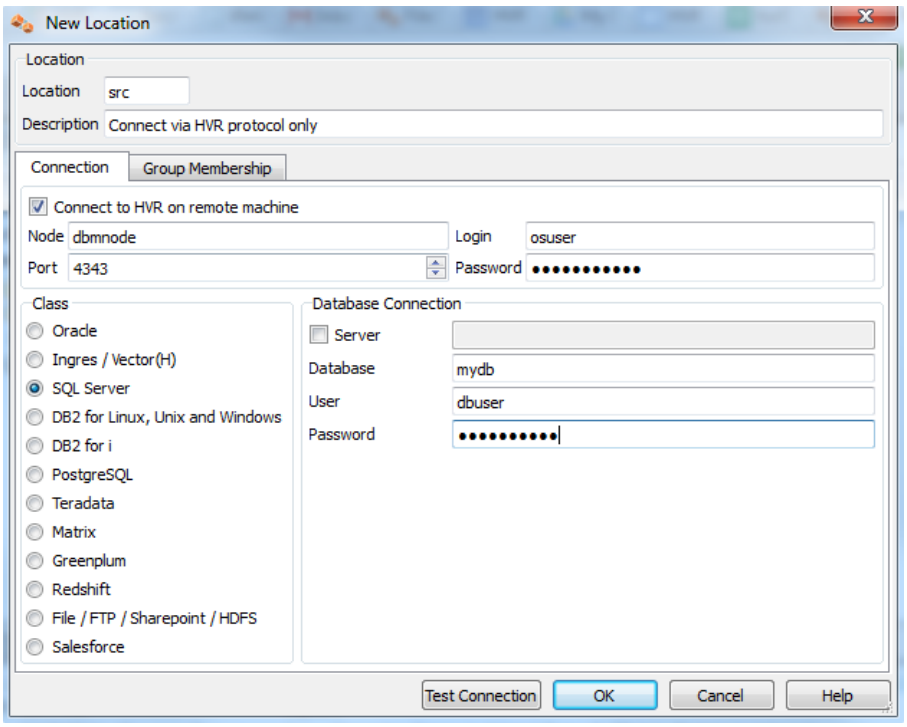


Method 1

Connect to a SQL Server database using the SQL Server protocol (equivalent to TNS).

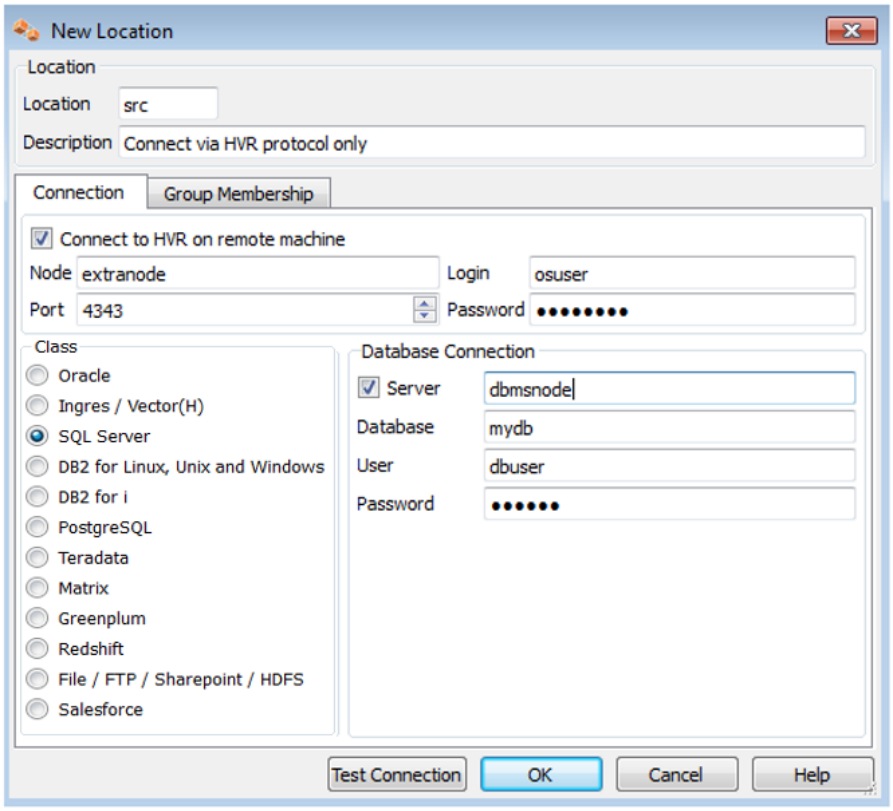
To use this method, Microsoft SQL Server Native Client should be installed on the machine from which HVR will connect to SQL Server database.

SQL Server Native Client can be downloaded from this Microsoft download page and the instructions for installation is available in Microsoft documentation - Installing SQL Server Native Client.



Method 2

Connect to a HVR installation running on the machine containing the SQL Server database using HVR's protocol on a special TCP/IP port number, e.g. 4343. On Windows this port is serviced by a Windows service called HVR Remote Listener. This option gives the best performance, but is the most intrusive.



Method 3

Connect first to a HVR installation on an extra machine using HVR's protocol (a sort of proxy) and then connect from there to the SQL Server database machine using the SQL Server protocol (equivalent to TNS). This option is useful when connecting from a Unix/Linux hub to avoid an (intrusive) installation of HVR's software on the machine containing the SQL Server database.

SQL Server on Linux

HVR supports **Capture** and **Integrate** for SQL Server running on Linux.

- HVR requires the Microsoft ODBC Driver (version 17.5 or higher) for SQL Server.
  1. Download and install the latest Microsoft ODBC Driver for SQL Server on Linux. For more information, refer to Installing the Microsoft ODBC Driver for SQL Server on Linux and macOS in Microsoft documentation.
  2. Create a symbolic link (symlink) for the ODBC driver. Following is an example for Microsoft ODBC Driver for SQL Server libmsodbcsql-17.5.so.1.1,

```
 ln -s  /opt/microsoft/msodbcsql17/lib64/libmsodbcsql-17.5.so.1.1  $HVR_HOME
/lib/libmsodbcsql-17.so
```

- The **User** (username used for connecting HVR to SQL Server) should have read access to the **.mdf** and **.ldf** files. For this, the **User** should typically be added to the Operating System user group **mssql**.
- After installing the Microsoft ODBC Driver for SQL Server, it is recommended to verify the dynamic dependencies. For example,

```
ldd $HVR_HOME/lib/hvr_ms17.so
```

Hub

HVR allows you to create hub database in SQL Server. The hub database is a small database present on the hub machine which HVR uses to control its replication activities. This database stores HVR catalog tables that hold all specifications of replication such as the names of the replicated databases, the list of replicated tables, and the replication direction.

Grants for Hub Database

To **Capture** changes from source database or to **Integrate** changes into target database, the HVR hub database(**User**) requires the privileges mentioned in this section.

It is recommended to create a new database (schema) for HVR hub. If an existing database is to be used for HVR Hub, then the HVR's catalog tables can be separated by creating a new database schema and associating it with HVR's user as follows:

```
create schema hvrschema
grant create table to hvruser
grant create procedure to hvruser
grant select, insert, delete, update on schema::hvrschema to hvruser
grant control on schema::hvrschema to hvruser
alter user hvruser with default_schema=hvrschema
```

Capture

HVR allows you to **Capture** changes from an SQL Server database. HVR uses SQL Server ODBC driver to capture changes from an SQL Server location. This section describes the configuration requirements for **capturing** changes from SQL Server location. For the list of supported SQL Server versions, from which HVR can capture changes, see Capture changes from location in Capabilities.

Table Types

HVR supports capture from the following table types in SQL Server:

- clustered (row/page compressed and uncompressed)
- heap (row/page compressed and uncompressed)

> HVR does not support capture from memory optimized tables and trigger-based capture from temporal tables.

Capture Methods

The following two methods are supported for capturing (**Capture**) changes from SQL Server:

- **LogReadMethod:** capture changes from transaction log files.
  - **DIRECT:** capture changes directly from SQL Server's logging files.
  - **SQL:** capture changes over an SQL connection.
- **Archive Log Only:** capture changes from transaction log backup files.

DIRECT Log Read Method

By default, HVR uses the **DIRECT** method to capture changes from the SQL Server's current and backup transaction log files, as well as compressed backup transaction log files. It is not required to define action **Capture /LogReadMethod**=**DIRECT**.

The benefits of the **DIRECT** method are:

- the **DIRECT** method is faster and less resource-intensive when capturing changes from database locations, especially for highly loaded databases. To ensure uninterrupted, low-latency CDC, capture must be running faster than the database is writing the logs. The **DIRECT** method and pipelined execution ensures optimum efficiency to keep up with the database log writers. As a result, when capture runs continuously, it will be capturing from the tail end of the log where the log writer(s) are writing.
- the **DIRECT** method supports capture from SQL Server Always On AG secondary database.

The **DIRECT** method requires the following:

- The HVR agent must be installed on the SQL Server source database server.
- The database account must have **SysAdmin** (SA) privileges.
- On Windows, the HVR **User** must have Windows Administrator privileges.
- On Linux, the HVR **User** must have read access to the **.ldf** files.

SQL Log Read Method

In the case of the **SQL** method, HVR captures changes over an SQL connection. This method uses stored database function calls to retrieve incremental log fragments. The benefits of the **SQL** method include:

- the ability to run with minimal OS and database privileges
- the ability to capture from a local/remote database without using an HVR agent

> Note that remote capture is less efficient than capture using an HVR remote agent on the database server. The **SQL** method will impose more overhead on the transactional database than the **DIRECT** method.

The **SQL** method is slower than the **DIRECT** method, exposes additional load on a source database and, for certain column types, HVR may receive partial/incomplete values and may require HVR to perform additional steps to retrieve the full value from the source database, this is called augmenting (**/AugmentIncomplete**).

To capture changes using the **SQL** method, define action **Capture /LogReadMethod**=**SQL**.

Archive Log Only Method

The **Archive Log Only** method may be activated using options **/ArchiveLogPath**, **/ArchiveLogFormat**, and **/ArchiveLogOnly** in the **Capture** action. The **Archive Log Only** method will generally expose higher latency than non-**Archive Log Only** method because changes can only be captured when the transaction log backup file is created. The **Archive Log Only method** enables high performance log-based Change Data Capture (CDC) with minimal OS and minimal database privileges, at the cost of higher capture latency.

Grants for Log-Based Capture

HVR's log-based capture using the **SQL** log read method supports three permission models: **SysAdmin**, **DbOwner**, and **Minimal**. However, the **DIRECT** log read method supports only the **SysAdmin** model.

The **DbOwner** and **Minimal** models are only available for SQL Server 2012 and above. For the older versions of SQL Server, the **SysAdmin** model should be used.

- **SysAdmin**
  The **User** should be granted a **sysadmin** role. There is no need for operators to perform special SQL statements manually. For this permission model, perform only installation steps 1 and 2 below; the others are unnecessary.

  This permission model is required when using **Capture /LogReadMethod=DIRECT**, except when using the **DIRECT** log read method combined with **Capture /ArchiveLogOnly**.

- **DbOwner**
  The **User** should be granted a **db_owner** role for the source database, but not a **sysadmin** role. An operator must perform several special SQL statements manually only when setting up a new database for capture. For this permission model, perform only installation steps 1-4 and 7 below; numbers 5-6 are unnecessary.
- **Minimal**
  The **User** does not require **sysadmin** or **db_owner** roles at runtime. But whenever an **HVR Initialize** command is run (for example to add a new table to a channel) then a user with a **db_owner** privilege must perform SQL statements manually. For this permission model, perform all the installation steps below: numbers 1-7.

Installation steps depend on the setting of action **Capture /SupplementalLogging**. It can use the elements of SQL Server's Change Data Capture feature or the elements of SQL Server's own replication components (called 'articles'), or both. Articles can only be created on tables with a primary key. CDC tables are used by default on the SQL Server Enterprise and Developer editions. Articles are always used for the SQL Server Standard edition (prior to SQL Server 2016 Service Pack 1) to implement supplemental logging.

Installation Steps

1. For log-based capture from the SQL Server Enterprise Edition or Developer Edition, if articles are used (see above), HVR requires that the SQL Server Replication Components option is installed. This step is needed once when HVR is installed for an SQL Server instance.
2. If articles are used (see above), a user with a **sysadmin** privilege must create a distribution database for the SQL Server instance, unless one already exists. To do this, a user with a **sysadmin** privilege should run SQL Server wizard **Configure Distribution Wizard**, which can be run by clicking **Replication > Configure Distribution...** Any database name can be supplied (click **Next > Next > Next**). This step is needed once when HVR is installed for an SQL Server instance.

   If Always On AG is installed and articles are used, then only one distribution database should be configured. Either this can be set up inside the first node and the other nodes get a distributor that points to it. Or the distribution database can be located outside the Always On AG cluster and each node gets a distributor that points to it there.
3. For this step and subsequent steps, the HVR binaries must already be installed. For log read method **SQL**, a user with a **sysadmin** privilege must create a special 'wrapper' SQL procedure called **sp_hvr_dblog** so that the HVR can call the SQL Server's read-only function **fn_dump_dblog**. This must be done inside the SQL Server database's special database **msdb**, not the actual capture database. The SQL query to create these procedures is available in the file called **hvrcapsysadmin.sql** in directory **%HVR_HOME%\sql\sqlserver**. The HVR user must then be allowed to execute this procedure. For this, the HVR User (e.g. **hvruser**) must be added to the special **msdb** database and the following grants must be provided:

```
use msdb

create user hvruser for login hvruser

grant execute on sp_hvr_dblog to hvruser

grant execute on sp_hvr_dbcc to hvruser      -- only for HVR versions upto 5.3.1/4

grant execute on sp_hvr_dbtable to hvruser   -- only for HVR versions since 5.3.1
/5
```

This step is needed once when HVR is installed for an SQL Server instance. But if Always On AG is installed, then this step is needed on each Always On AG node.

4. A **sysadmin** user must grant the HVR user login a special read-only privilege in the **master** database.

```
use master

grant view server state to hvruser
```

This step is needed once when HVR is installed for an SQL Server instance. But if Always On AG is installed, then this step is needed on each Always On AG node.

5. If Articles are used or the log read method is **SQL**, then a user with a **db_owner** (or **sysadmin**) privilege must create 'wrapper' SQL procedures in each capture database so that HVR can call the SQL Server's read-only procedures **sp_helppublication**, **sp_helparticle** and **fn_dblog**. The SQL query to create these three read-only procedures is available in the file called **hvrcapdbowner.sql** in directory **%HVR_HOME%\sql\sqlserver**. The **User** must then be allowed to execute these procedures.

   The following grants must be provided inside each capture database:

```
use capdb

grant execute on sp_hvr_check_publication to hvruser

grant execute on sp_hvr_check_article to hvruser

grant execute on sp_hvr_dblog to hvruser

grant execute on sp_hvr_repldone to hvruser

grant execute on sp_hvr_repltrans to hvruser
```

   This step is needed once when each new source database is being set up.

6. A user with **db_owner** (or **sysadmin**) privilege must grant the HVR user a read-only privilege.

   This step is needed once when each new source database is being set up.

```
use capdb

alter role db_datareader add member hvruser
```

7. When the **HVR Initialize** command is performed, it may need to perform SQL statements that would require **sysadmin** or **db_owner** privilege. One example is that it may need to create an Article on a replicated table to track its changes. In that case, **HVR Initialize** will write a script containing necessary SQL statements, and then show a popup asking for this file to be executed. The file will be written in directory **%HVR_CONFIG%\files** on the capture machine; its exact filename and the necessary permission level is shown in the error

message. The first time **HVR Initialize** gives this message, a user with a **sysadmin** privilege must perform these SQL statements. Subsequently, these SQL statements can be performed by a user that just has a **db_owner** privilege.

Capturing from SQL Server Always On Availability Groups

HVR allows you to capture from SQL Server Always On Availability Groups (AG) - a technology which provides High-Availability (HA) and Disaster-Recovery (DR) solution in SQL Server.

When using **DIRECT** method, HVR can be configured to capture from either the primary or secondary node (active or passive).

However, when using **SQL** method, HVR can be configured to capture only from the primary node.

Configuring Failover for Connections to SQL Server Always On AG

If HVR is connecting using its own protocol to a database inside an SQL Server Always On AG, the following is required:

1. Create an HVR Remote Listener on each node
2. Inside Failover Cluster Manager configure a Role with a static IP address that controls the HVR Remote Listener service on all nodes. Run **Configure Role Wizard, Next > Generic Service > HVR Remote Listener > Name > Next > Next > Next > Finish**. To configure a static IP address, click on **Resources > Name> IP address>** then change it to an available static address.
3. Configure a Group Listener inside the Availability Group. Start Management Studio on the primary node, click **AlwaysOn High Availability > Availability Groups > Name > Availability Group Listeners > Add Listener** ….

   For Always On AG inside Azure an extra step is required:
4. Configure an Internal or External Azure load balancer for the HVR Remote Listener using Powershell. Then attach each Azure node to this load balancer. When this is done, fill in the IP address of the Azure load balancer into the **Node** field in the **Connect to HVR on remote machine** section of the HVR location dialog.

HVR can now connect to Node as configured in step 1 or step 4 and Server as configured in Step 3.

Configuring Backup Mode and Transaction Archive Retention

HVR log-based capture requires that the source database is in **Full recovery** model and a full database backup has been done since this was enabled. Normally HVR reads changes from the 'online' transaction log file, but if HVR is interrupted (say for 2 hours) then it must be able to read from transaction log backup files to capture the older changes. HVR is not interested in full or incremental backups; it only reads transaction log backup files.

- HVR supports only native single-media-family backup(s).
- HVR does not support backup(s) on 'virtual devices'.
- For **DIRECT** method, the backup(s) must be accessible to HVR on the file system.
- For **SQL** method, the backup(s) must be accessible to SQL Server on the file system.
- If **SQL** method is defined with **Capture /ArchiveLogPath**, the backup(s) must be accessible to both SQL Server and HVR using the same path (UNC in case of network backup).

Transaction log (archive) retention: If a backup process has already moved these files to tape and deleted them, then HVR capture will give an error and an HVR Refresh will be needed before replication can be restarted. The amount of 'retention' needed (in hours or days) depends on organization factors (how real-time must it be?) and practical issues (does a refresh take 1 hour or 24 hours?).

HVR normally locates the transaction log backup files by querying the backup history tables in the **msdb** database. But if Always On AG is configured then this information source is not available on all nodes. So when HVR is used

with Always On AG, the transaction log backups must be made on a directory which is both accessible from all Always On AG nodes and also from the machine where the HVR capture process is running (if this is different) via the same path name. HVR should be configured to find these files by defining action **Capture** with two additional parameters **/ArchiveLogPath** and **/ArchiveLogFormat**. The parameter **/ArchiveLogPath** should point to a file system directory which HVR will use for searching directly for the transaction log backup files, instead of querying **msdb**. The parameter **/ArchiveLogFormat** should specify a pattern for matching files in that directory. The pattern can contain these special characters:

Pattern	Description
*	Wildcard to match zero or more characters.
%d	Database name. This is not case sensitive.
%Y	Year (up to 4 digit decimal integer).
%M	Month (up to 2 digit decimal integer)
%D	Day (up to 2 digit decimal integer)
%h	Hours (up to 2 digit decimal integer)
%m	Minutes (up to 2 digit decimal integer)
%s	Seconds (up to 2 digit decimal integer)
%n	File sequence number (up to 64 bit decimal integer)
%%	Matches %

All other characters must match exactly. HVR uses the %Y, %M, %D, %h, %m, %s and %n values to order files.

Dropping the Source Database

Depending on the setting of action **Capture /SupplementalLogging** HVR will use some of SQL Server's own 'replication components' or it will use SQL Server's Change Data Capture (CDC) feature.

Based on this, HVR may enable the source database for publication, which will mean that attempts to drop the database will give an SQL Server error.

Alternatively HVR may enable Change Data Capture (CDC) on source databases, which will also mean attempts to drop the database may also give an SQL Server error because of the running CDC capture and cleanup jobs.

When command **HVR Initialize** is used with **Drop Objects** (option **-d**) then it will disable the 'publish' replication option if there are no other systems capturing from that database. It will also disable the CDC for the database, if there are no other CDC table instances exist in that database. The database can then be dropped.

To drop the database immediately (without running the **HVR Initialize** first) the **sysadmin** must perform the following SQL statement:

```
exec sp_replicationdboption 'capdb', 'publish', 'false'
```

```
use [capdb]
exec sp_cdc_disable_db
```

Grants for Trigger-Based Capture

HVR allows you to perform trigger-based capture (**Capture /TriggerBased**) from SQL Server. To enable triggerbased capture for SQL Server:

- HVR's user should be made a database owner (**db_owner** role).
- The extended stored procedure **hvrevent** should normally be installed on the capture machine. This is not needed if parameters **Capture /TriggerBased** is not defined or **/ToggleFrequency** or **Scheduling /CaptureStartTimes** or **/CaptureOnceOnStart** are defined. This step must be performed by a user that is a member of the system administrator role. For more information, see Installing HVR on Windows.

Limitations

- HVR does not support log-based capture from Amazon RDS for SQL Server.

Integrate and Refresh Target

HVR allows you to **Integrate** changes into SQL Server database. This section describes the configuration requirements for integrating changes (using **Integrate** and **Refresh**) into SQL Server location. For the list of supported SQL Server versions, into which HVR can integrate changes, see Integrate changes into location in Capabilities.

HVR uses the following interfaces to write data into an SQL Server location:

- SQL Server ODBC driver, used to perform continuous **Integrate** and row-wise **Refresh**
- SQL Server BCP interface, used for copying data into database tables during bulk **Refresh** and loading data into burst tables during **Integrate** with **/Burst**

Grants for HVR on Target Database

This section provides information about the user privileges required for replicating changes into SQL Server database using **HVR Refresh**.

When replicating changes into a target SQL Server database, HVR supports the following two permission models: **DbOwner**, and **Minimal**.

- **DbOwner**
  In this permission model, the HVR **User** must be made a database owner (**db_owner** role).Normally, the database objects which HVR sometimes creates will be part of the dbo schema as the replicated tables. Alternatively, these HVR database objects can be put in a special database schema so that they are not visible to other users. The following SQL is needed:

```
create schema hvrschema

grant control on schema::hvrschema to hvruser

alter user hvruser with default_schema=hvrschema
```

- **Minimal**
  In this permission model, the **User** does not need to be a database owner. This model cannot use parameter **/Schema** to change tables with a different owner. The following SQL is needed so that HVR can create its own tables:

```
grant create table to hvruser

create schema hvrschema

grant control on schema::hvrschema to hvruser

alter user hvruser with default_schema=hvrschema
```

If action **Integrate** **/DbProc** is defined, then **create procedure** privilege is also needed.

Compare and Refresh Source

To perform **HVR Compare** or **HVR Refresh** (in Source Database), the HVR **User** requires the privileges mentioned in this section.

When HVR is reading rows from a database (no capture) it supports two permission models: **DbOwner**, and **Minimal**.

- **DbOwner**
  In this permission model, the HVR **User** must be made owner of the source database (**db_owner** role).

- **Minimal**
  In this permission model, the HVR **User** does not need to be a database owner.

If the HVR **User** needs to select from tables in another schema (for example if action **TableProperties** **/Schema** is defined), then **select** privilege should be granted.

```
grant select to hvruser -- Let HVR read all tables

grant select on schema::dbo to hvruser -- Let HVR only read DBO tables
```

Managing SQL Server Log File Truncation

SQL Server

Contents

Overview

SQL Server log file need to be truncated periodically to prevent its excessive growth. HVR provides following options for that:

- have HVR capture job with automatic log truncation turned on. It suits most simple cases, easily configurable and provides good performance;
- employ Hvrlogrelease command. This may be required for more complex scenarios and requires more configuration;
- utilize native SQL Server agent to read and truncate log file.

Capture Job Log Truncation

This is default option for HVR capture job. Here log file truncation is performed automatically by HVR capture job when it runs. It's fast and powerful and yet simple to use as it has zero configuration. Yet it has disadvantage to not fit in following scenarios:

- coexistence, when several brands of replication tools capture simultaneously from the same database;
- multi-capture, when several HVR capture jobs capture simultaneously from the same database;
- long period of HVR capture job inactivity had already happened or planned, so that regular automatic log file truncation was or is impossible.
  In such cases it's better to apply **Hvrlogrelease** command as described below.

Native SQL Server Agent

HVR capture can work along with native SQL Server agent. But this option should be taken with care as it's sensitive to configuration errors. So it's suggested to consult with HVR support team to decide whether it needed and how HVR capture job need to be configured.

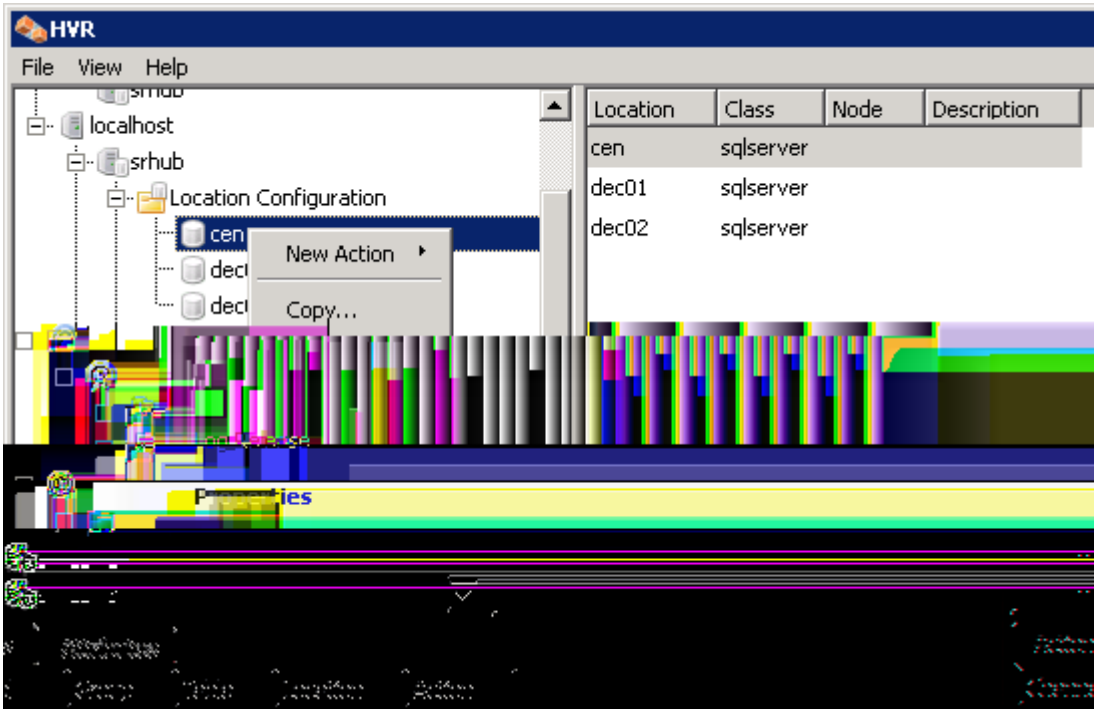
Hvrlogrelease

Hvrlogrelease command allows to deal with complex scenarios where HVR capture job is not enough such as multi-capture and long HVR capture job inactivity. This approach employs having specific HVR task to run **Hvrlogrelease** command to truncate log file. Such task can be run either manually or by schedule once configured.

Hvrlogrelease Configuration

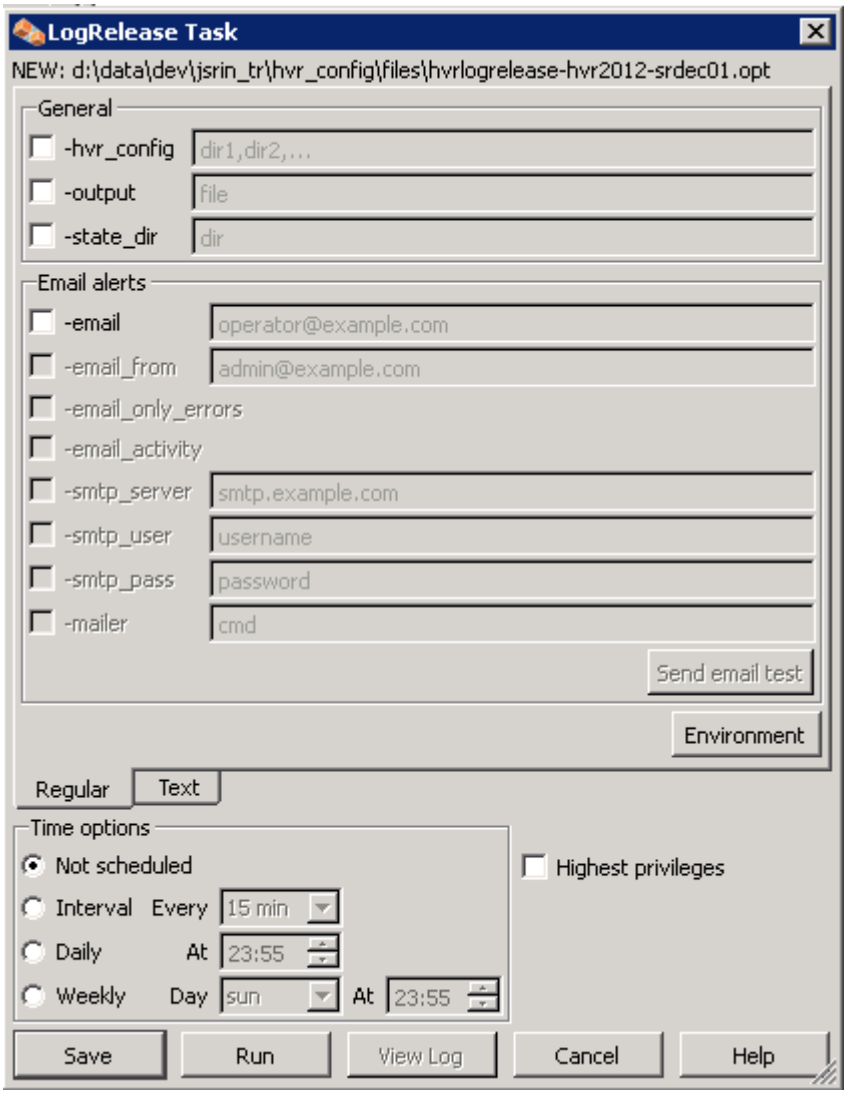
Hvrlogrelease task can be configured from **HVRGUI** or in the command line.

1. Right-click on SQL Server location and select **Log Release**.



Log Release menu item is enabled only for SQL Server location class.

2. **Hvrlogrelease** task options can be configured in the top part of the dialog box.



3. For HVR hub on Windows, a user with chosen privileges level can schedule the task to run at specific time intervals (by configuring the **Time options**). This option is not available in Linux.
4. Click **Save** to store the configuration to *optfile*.
5. It's also possible to run the task immediately by clicking the **Run** button.

Requirements for Teradata

Contents

This section describes the requirements, access privileges, and other features of HVR when using Teradata for replication. For information about compatibility and supported versions of Teradata with HVR platforms, see Platform Compatibility Matrix.

For the Capabilities supported by HVR on Teradata, see Capabilities for Teradata.

For information about the supported data types and mapping of data types in source DBMS to the corresponding data types in target DBMS or file format, see Data Type Mapping.

To quickly setup replication using Teradata, see Quick Start for HVR - Teradata.

ODBC Connection

HVR uses ODBC connection to the Teradata server for which it requires the Teradata ODBC driver installed on the machine from which it connects to the Teradata server. HVR only supports Teradata ODBC driver version 15.00 or 16.10. HVR also requires Teradata Parallel Transporter(TPT) packages to use HVR Refresh in bulk mode.

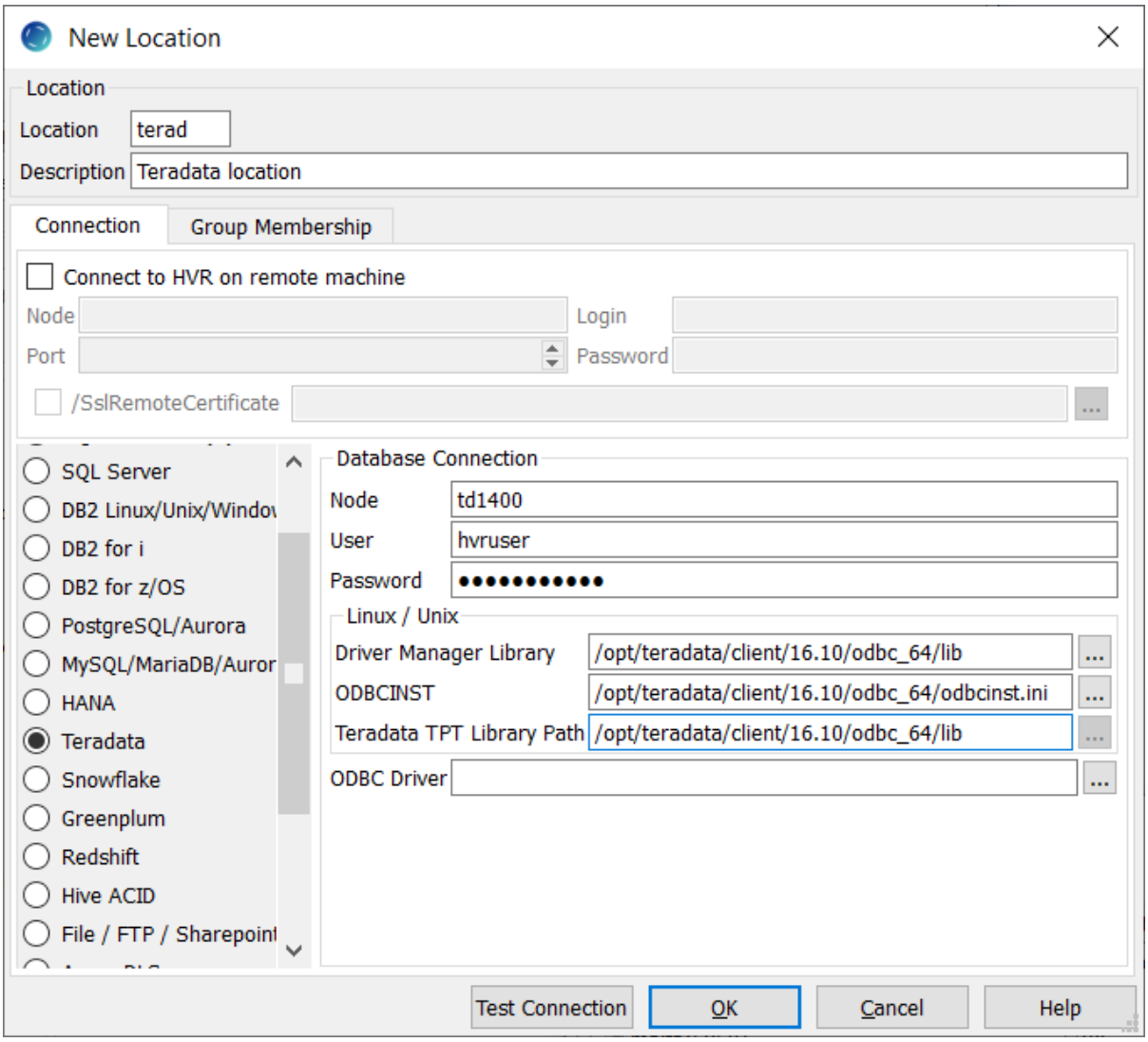
Teradata ODBC driver and TPT packages can be installed using Teradata Tools and Utilities (TTU) package. TTU 16.10 is available for Linux and Windows on Teradata download page, and TTU 15.00 is available for download from Teradata Support Portal (requires user authentication).

The following action definitions are required for TTU to find the correct message files:

Group	Table	Action	Annotation
Teradata	*	**Environment /Name**=*NLSPATH*	
Teradata	*	**Environment /Value="/opt/teradata/client/***teradata_version***/odbc_64/msg/%N:/opt/teradata/client/***teradata_version***/tbuild/msg/%"**	For Teradata 15.00
Teradata	*	**Environment /Value="/opt/teradata/client/***teradata_version***/odbc_64/msg/%N:/opt/teradata/client/***teradata_version***/msg/%N"**	For Teradata 16.10 and higher

Location Connection

This section lists and describes the connection details required for creating Teradata location in HVR.



Field	Description
Database Connection	
Node	The hostname or ip-address of the machine on which the Teradata server is running. **Example:** td1400
User	The username to connect HVR to the Teradata **Node** . **Example:** hvruser
Password	The password of the **User** to connect HVR to the Teradata **Node** .
Linux / Unix	
Driver Manager Library	The directory path where the Unix ODBC Driver Manager Library is installed. **Example:** /opt/teradata/client/16.10/odbc_64/lib
ODBCINST	The directory path where the **odbcinst.ini** file is located. **Example:** /opt/teradata/client/16.10/odbc_64/odbcinst.ini
Teradata TPT Library Path	The directory path where the Teradata TPT Library is installed. **Example:** /opt/teradata/client/16.10/odbc_64/lib
ODBC Driver	The user defined (installed) ODBC driver to connect HVR to the Teradata server.

- TPT Load, used for copying data to Teradata tables during bulk **Refresh**
- TPT Stream/Upload, used load data into burst table during **Integrate** with **/Burst**.

Grants for Integrate and Compare

By default, the HVR **User** has all required permissions to Integrate changes into default user database. Following are the grants required for integrating changes and perform compare in Teradata:

1. To change/replicate into target tables which are not owned by HVR **User** (using **TableProperties** **/Schema**), the **User** must be granted **select**, **insert**, **update**, and **delete** privileges.

```
grant select, insert, update, delete on targetdb to hvruser;
```

2. To create target tables using **Hvrrefresh**, the **User** must be granted **create table** and **drop any table** privileges.

```
grant create table on targetdb to hvruser;

grant drop any table on targetdb to hvruser;
```

3. To read from tables which are not owned by HVR **User** (using **TableProperties** **/Schema**) during **Hvrcompare** or **Hvrrefresh**, the **User** must be granted **select** privilege.

```
grant select on targetdb to hvruser;
```

4. To perform bulk load, the **User** must be granted **create macro** privilege.

```
grant create macro on targetdb to hvruser;
```

Actions

Actions in HVR allows you to define the behavior of replication. Every action has a collection of parameters that provide a finer control of the behavior. To initiate data replication, at least two actions **Capture** and **Integrate** must be defined on source and target locations accordingly.

This section describes all HVR actions and their parameters.

Action Reference

AdaptDDL AgentPlugin Capture CollisionDetect ColumnProperties DbObjectGeneration DbSequence Environment FileFormat Integrate LocationProperties Restrict Scheduling TableProperties Transform

Action	Parameter	Argument	Description
AdaptDDL	**/AddTablePattern**	*patt*	Add new tables to channel if they match.
	/IgnoreTablePattern	*patt*	Ignore new tables which match pattern.
	/CaptureSchema	*db_schema*	Database schema for matching tables.
	/IntegrateSchema	*db_schema*	Generate /Schema for target location(s).
	/RefreshOptions	*refr_opts*	Configure options for adapt's refresh of target.
	/OnDropTable	*pol*	Behavior when source table dropped. Default: from channel only.
	/KeepExistingStructure		Preserve old columns in target, and do not reduce data types sizes.
	/KeepOldRows		Preserve old rows in target during recreate.
AgentPlugin	**/Command**	*path*	Call OS command during replication jobs.
	/DbProc	*dbproc*	Call database procedure *dbproc* during replication jobs.
	/UserArgument	*str*	Pass argument *str* to each agent execution.
	/ExecOnHub		Execute agent on hub instead of location's machine.
	/Order	*int*	Specify order of agent execution.
	/Path	*dir*	Search directory *dir* for agent.
	/Context	*context*	Action only applies if Refresh/Compare *context* matches.
Capture	**/IgnoreSessionName**	*sess_name*	Capture changes directly from DBMS logging system.
	/Coalesce		Coalesce consecutive changes on the same row into a single change.
	/NoBeforeUpdate		Only capture the new values for updated rows.
	/NoTruncate		Do not capture truncate table statements.
	/SupplementalLogging	*action*	Mechanism used to enable supplemental logging for SQL Server tables.
	/LogReadMethod	*method*	Method of reading SQL Server's transaction log.
	/LogTruncate	*action*	Specify who advances transaction log truncation point.
	/AugmentIncomplete	*col_type*	Capture job must select for column values. Can be **NONE** , **LOB** or **ALL** .
	/ArchiveLogPath	*dir*	Read archives from an alternative directory.
	/ArchiveLogFormat	*format*	Format of base filename of archive files in directory **/ArchiveLogPath** .
	/ArchiveLogOnly		Capture data from archives only. Do not read from online redos.
	/XLogDirectory	*dir*	Directory containing current PostgreSQL xlog files.
	/LogJournal	*schema.journal*	Specify DB2-for-i journal.
	/LogJournalSysSeq		Capture from journal using ***SYSSEQ** .
	/CheckpointFrequency	*secs*	Checkpointing frequency in seconds for long running transactions, so the capture job can recover quickly when it restarts.
	/CheckpointStorage	*STOR*	Storage location of capture checkpoint files for quick capture recovery.
	/CheckpointRetention	*period*	Retains capture checkpoint files up to the specified *period* .

	/TriggerBased		Capture changes through generated DMBS triggers.
	/QuickToggle		Avoid shared lock on toggle table.
	/ToggleFrequency	*secs*	Sleep between toggles instead of waiting for database alert (in seconds).
	/KeyOnlyCaptureTable		Only keep keys in capture table; outer join others later.
	/IgnoreCondition	*sql_expr*	Ignore changes that satisfy expression.
	/IgnoreUpdateCondition	*sql_expr*	Ignore update changes that satisfy expression.
	/HashBuckets	*int*	Hash structure to improve parallelism of captured tables.
	/HashKey	*col_list*	Hash capture table on specific key columns.
	/DeleteAfterCapture		Delete file after capture, instead of capturing recently changed files.
	/Pattern	*pattern*	Only capture files whose names match *pattern* .
	/IgnorePattern	*pattern*	Ignore files whose names match *pattern* .
	/IgnoreUnterminated	*pattern*	Ignore files whose last line does not match *pattern* .
	/IgnoreSizeChanges		Changes in file size during capture is not considered an error.
	/AccessDelay	*secs*	Delay read for *secs* seconds to ensure writing is complete.
	/UseDirectoryTime		Check timestamp of parent dir, as Windows move doesn't change mod-time.
CollisionDetect	**/TreatCollisionAsError**		Do not resolve collisions automatically.
	/TimestampColumn	*col_name*	Exploit timestamp column *col_name* for collision detection.
	/AutoHistoryPurge		Delete history table row when no longer needed for collision detection.
	/DetectDuringRefresh	*colname*	During row–wise refresh, discard updates if target timestamp is newer.
	/Context	*context*	Action only applies if Refresh/Compare *context* matches.
ColumnProperties	**/Name**	*col_name*	Name of column in **hvr_column** catalog.
	/DatatypeMatch	*data_type*	Data type used for matching instead of **/Name** .
	/BaseName	*col_name*	Database column name differs from **hvr_column** catalog.
	/Extra		Column exists in base table but not in **hvr_column** catalog.
	/Absent		Column does not exist in base table.
	/CaptureExpression	*sql_expr*	SQL expression for column value when capturing or reading.
	/CaptureExpressionType		Type of mechanism used by HVR capture, refresh and compare job to evaluate value in parameter **/CaptureExpression** .
	/IntegrateExpression	*sql_expr*	SQL expression for column value when integrating.
	/ExpressionScope	*expr_scope*	Operation scope for expressions, e.g. **INSERT** , **DELETE** or **UPDATE_AFTER** .
	/CaptureFromRowId		Capture values from table's DBMS row-id.
	/TrimDatatype	*int*	Reduce width of data type when selecting or capturing changes.
	/Key		Add column to table's replication key.
	/SurrogateKey		Use column instead of the regular key during replication.
	/DistributionKey		Distribution key column.
	/PartitionKeyOrder	*int*	Define the column as a partition key and set partitioning order for the column.
	/SoftDelete		Convert deletes to update of this column to 1. Value 0 means not deleted.

	/TimeKey		Convert all changes to inserts, using this column for time dimension.
	/IgnoreDuringCompare		Ignore values in column during compare and refresh.
	/Datatype	*data_type*	Data type in database if it differs from **hvr_column** catalog.
	/Length	*int*	String length in db if it differs from length in catalog.
	/Precision	*int*	Precision in db if it differs from precision in catalog.
	/Scale	*int*	Integer scale in db if it differs from scale in catalog.
	/Nullable		Nullability in db if it differs from nullability in catalog.
	/Identity		Column has SQL Server identity attribute.
	/Context	*ctx*	Ignore action unless refresh/compare context *ctx* is enabled.
DbObjectGeneration	**/NoCaptureInsertTrigger**		Inhibit generation of capture insert trigger.
	/NoCaptureUpdateTrigger		Inhibit generation of capture update trigger.
	/NoCaptureDeleteTrigger		Inhibit generation of capture delete trigger.
	/NoCaptureDbProc		Inhibit generation of capture database procedures.
	/NoCaptureTable		Inhibit generation of capture tables.
	/NoIntegrateDbProc		Inhibit generation of integrate database procedures.
	/IncludeSqlFile	*file*	Search directory for include SQL file.
	/IncludeSqlDirectory	*dir*	Search directory for include SQL file.
	/CaptureTableCreateClause	*sql_expr*	Clause for trigger-based capture table creation statement.
	/StateTableCreateClause	*sql_expr*	Clause for state table creation statement.
	/BurstTableCreateClause	*sql_expr*	Clause for integrate burst table creation statement.
	/FailTableCreateClause	*sql_expr*	Clause for fail table creation statement.
	/HistoryTableCreateClause	*sql_expr*	Clause for history table creation statement.
	/RefreshTableCreateClause	*sql_expr*	Clause for base table creation statement during refresh.
	/RefreshTableGrant		Executes a grant statement on the base table created during **HVR Refresh** .
DbSequence	**/CaptureOnly**		Only capture db sequences, do not integrate them.
	/IntegrateOnly		Only integrate db sequences, do not capture them.
	/Name	*seq_name*	Name of database sequence in HVR catalogs.
	/Schema	*db_schema*	Schema which owns db sequence.
	/BaseName	*seq_name*	Name of sequence in db if it differs from name in HVR.
Environment	**/Name**	*name*	Name of environment variable.
	/Value	*value*	Value of environment variable.
FileFormat	**/Xml**		Transform rows form/into xml-files.
	/Csv		Transforms rows from/into csv files.
	/Avro		Transforms rows into Apache AVRO format. Integrate only.

	/JSON		Transforms rows into JSON format. The content of the file depends on the value for parameter /JsonMode. This parameter only has an effect on the integrate location.
	/Parquet		Read and write files as Parquet format.
	/Compact		Write compact XML tags like <r> & <c> instead of <row> & <column>.
	/Compress	*algorithm*	Compress/uncompress while writing/reading. *algorithm* is **GZIP** or **LZ4** .
	/Encoding	*encoding*	Encoding of file.
	/HeaderLine		First line of file contains column names.
	/FieldSeparator	*str_esc*	Field separator. Defaults to comma (,). Examples: , \\x1f or \\t
	/LineSeparator	*str_esc*	Line separator. Defaults to newline (\\n). Examples: ;\\n or \r\\n
	/QuoteCharacter	*str_esc*	Character to quote a field with, if the fields contains separators. Defaults to quote (\").
	/EscapeCharacter	*str_esc*	Character to escape the quote character with. Defaults to quote (\").
	/FileTerminator	*str_esc*	File termination at end-of-file. Example: EOF or \xff
	/NullRepresentation	*esc_str*	String representation for columns with NULL value.
	/AvroCompression	*codec*	Avro compression codec. Value should be **Deflate** .
	/AvroVersion	*version*	Version of Apache AVRO format. Possible values are **v1_6** , **v1_7** and **v1_8** (the default).
	/JsonMode	*mode*	Style used to write row into JSON format.
	/PageSize		Parquet page size in bytes.
	/RowGroupThreshold		Maximum row group size in bytes for Parquet.
	/ParquetVersion	*version*	Category of data types to represent complex data into Parquet format.
	/ConvertNewlinesTo	*style*	Write files with UNIX or DOS style newlines.
	/CaptureConverter	*path*	Run files through converter before reading.
	/CaptureConverterArguments	*userarg*	Arguments to the capture converter.
	/IntegrateConverter	*path*	Run files through converter after writing.
	/IntegrateConverterArguments	*userarg*	Arguments to the integrate converter program.
	/Context	*context*	Action only applies if Refresh/Compare context matches.
Integrate	**/Burst**		Resort changes, load into staging table and apply with set-wise SQL.
	/BurstCommitFrequency	*freq*	Frequency of commits. Values **STATEMENT** , **TABLE** or **CYCLE** .
	/Coalesce		Coalesce consecutive changes on the same row into a single change.
	/ReorderRows	*mode*	Control order in which changes are written to files. Values **NONE** , **BATCH_BY_TABLE** , **ORDER_BY_TABLE** or **SORT_COALESCE** .
	/Resilient	*mode*	Resilient integrate for inserts, updates and deletes. Values **WARNING** or **SILENT** .
	/OnErrorSaveFailed		Write failed row to fail table.
	/DbProc		Apply changes by calling integrate database procedures.
	/TxBundleSize	*int*	Bundle small transactions for improved performance.
	/TxSplitLimit	*int*	Split very large transactions to limit resource usage.
	/NoTriggerFiring		Enable/Disable triggering of database rules.

	/SessionName	*sess_name*	Integrate changes with special session name *sess_name* .
	/Topic	*expression*	Name of the Kafka topic. You can use strings/text or expressions as Kafka topic name.
	/MessageBundling	*mode*	Number of messages written into single Kafka message. Kafka message contains one row by default.
	/MessageBundlingThreshold	*int*	The threshold for bundling rows in a Kafka message. The default value is 800,000 bytes.
	/MessageKey	*expression*	Expression to generate user defined key in a Kafka message.
	/RenameExpression	*expression*	Expression to name new files, containing brace substitutions.
	/ComparePattern	*patt*	Perform direct file compare.
	/ErrorOnOverwrite		Error if a new file has same name as an existing file.
	/MaxFileSize	*size*	Limit each XML file to *size* bytes.
	/Verbose		Report name of each file integrated.
	/TableName	*apitab*	API name of table to upload attachments into.
	/KeyName	*apikey*	API name of attachment table's key column.
	/CycleByteLimit	*int*	Max amount of routed data (compressed) to process per integrate cycle.
	/JournalRouterFiles		Move processed router files to journal directory on hub.
	/Delay	*N*	Delay integration of changes for *N* seconds.
	/Context	*ctx*	Action only applies if Refresh/Compare context matches.
LocationProperties	**/SslRemoteCertificate**	*file*	Enable SSL encryption to remote location; verify location with certificate.
	/SslLocalCertificateKeyPair	*path*	Enable SSL encryption to remote location; identify with certificate/key.
	/ThrottleKbytes	*kbytes*	Restrain net bandwidth into packets of *kbytes* bytes.
	/ThrottleMillisecs	*msecs*	Restrain net bandwidth by *msecs* second(s) wait between packets.
	/Proxy	*proxy*	Proxy server URL for FTP, SFTP, WebDAV or Salesforce locations.
	/Order	*N*	Specify order of hub->loc proxy chain.
	/StateDirectory	*path*	Directory for file location state files. Defaults to <top>/_hvr_state.
	/IntermediateDirectory	*dir*	Directory for storing 'intermediate files' that are generated during compare.
	/CaseSensitiveNames		DBMS table and columns names are treated case sensitive by HVR.
	/StagingDirectoryHvr	*URL*	Directory for bulk load staging files.
	/StagingDirectoryDb	*URL*	Location for the bulk load staging files visible from the Database.
	/StagingDirectoryCredentials	*credentials*	Credentials to be used for S3 authentication during RedShift bulk load.
	/S3Encryption	*keyinfo*	Key information to be used for S3 client side encryption.
	/BucketsCount		Number of buckets to be specified while creating a table in Hive ACID.
	/BulkAPI		Use Salesforce Bulk API (instead of the SOAP interface).
	/SerialMode		Force serial mode instead of parallel processing for Bulk API.
	/CloudLicense		Location runs on cloud node with on-demand licensing, for example in Amazon or Azure Marketplace.
Restrict	**/CaptureCondition**	*sql_expr*	Restrict during capture.
	/IntegrateCondition	*sql_expr*	Restrict during integration.

	/RefreshCondition	*sql_expr*	Restrict during refresh and compare.
	/CompareCondition	*sql_expr*	Restrict during compare.
	/HorizColumn	*col_name*	Horizontal partition table based on value in *col_name* .
	/HorizLookupTable	*tbl_name*	Join partition column with horizontal lookup table.
	/DynamicHorizLookup		Changes to lookup table also trigger replication.
	/AddressTo	*addr*	Only send changes to locations specified by address.
	/AddressSubscribe	*addr*	Get copy of any changes sent to matching address.
	/SelectDistinct		Filter duplicate records during refresh/compare.
	/Context	*ctx*	Action only applies if Refresh/Compare context matches.
Scheduling	**/CaptureStartTimes**	*times*	Trigger capture job at specific times, rather than continuous cycling.
	/CaptureOnceOnStart		Capture job runs for one cycle after trigger.
	/IntegrateStartAfterCapture		Trigger integrate job only after capture job routes new data.
	/IntegrateStartTimes	*times*	Trigger integrate job at specific times, rather than continuous cycling.
	/IntegrateOnceOnStart		Integrate job runs for one cycle after trigger.
	/RefreshStartTimes	*times*	Trigger refresh job at specific times.
	/CompareStartTimes	*crono*	Trigger compare job at specific times.
	/StatsHistory	*size*	Size of history maintained by **hvrstats** job, before it purges its own rows.
TableProperties	**/BaseName**	*tbl_name*	Name of table in database differs from name in catalogs.
	/Absent		Exclude table (which is available in the channel) from being replicated /integrated into target.
	/DuplicateRows		Table has duplicate rows and no unique key.
	/Schema	*schema*	Database schema which owns table.
	/IgnoreCoerceError		Coerce illegal/big/small values to empty/max/min.
	/CoerceErrorPolicy		Defines a policy to handle type coercion error.
	/CoerceErrorType		Defines which types of coercion errors are affected by **/CoerceErrorPolicy** .
	/TrimWhiteSpace		Remove trailing whitespace from **varchar** .
	/TrimTime	*policy*	Trim time when converting from Oracle and SqlServer date.
	/MapEmptyStringToSpace		Convert between empty varchar and Oracle varchar space.
	/MapEmptyDateToConstant	*date*	Convert between constant **date** (dd/mm/yyyy) and Ingres empty date.
	/CreateUnicodeDatatypes		On table creation use Unicode data types, e.g. map **varchar** to **nvarchar** .
	/DistributionKeyLimit	*int*	Maximum number of columns in the implicit distribution key.
	/DistributionKeyAvoidPattern	*patt*	Avoid putting given columns in the implicit distribution key.
	/CharacterMapping	*rules*	Specify the replacement rules for unsupported characters.
	/MapBinary	*policy*	Specify how binary data is represented on the target side.
	/MissingRepresentationString	*str*	Inserts value *str* into the string data type column(s) if value is missing /empty in the respective column(s) during integration.

	/MissingRepresentationNumeric	*str*	Inserts value *str* into the numeric data type column(s) if value is missing /empty in the respective column(s) during integration.
	/MissingRepresentationDate	*str*	Inserts value *str* into the date data type column(s) if value is missing /empty in the respective column(s) during integration.
Transform	**/Command**	*path*	Path to script or executable performing custom transformation.
	/CommandArguments	*userarg*	Value(s) of parameter(s) for transform (space separated).
	/SapAugment		Capture job selecting for de-clustering of multi-row SAP cluster tables.
	/SapXForm		Invoke SAP transformation for SAP pool and cluster tables.
	/UnpackTables		Transform will map *_**pack** tables into ***_*_unpack*** tables.
	/ExecOnHub		Execute transform on hub instead of location's machine.
	/Parallel	*n*	Distribute rows to multiple transformation processes.
	/Context	*context*	Action only applies if Refresh/Compare *context* matches.

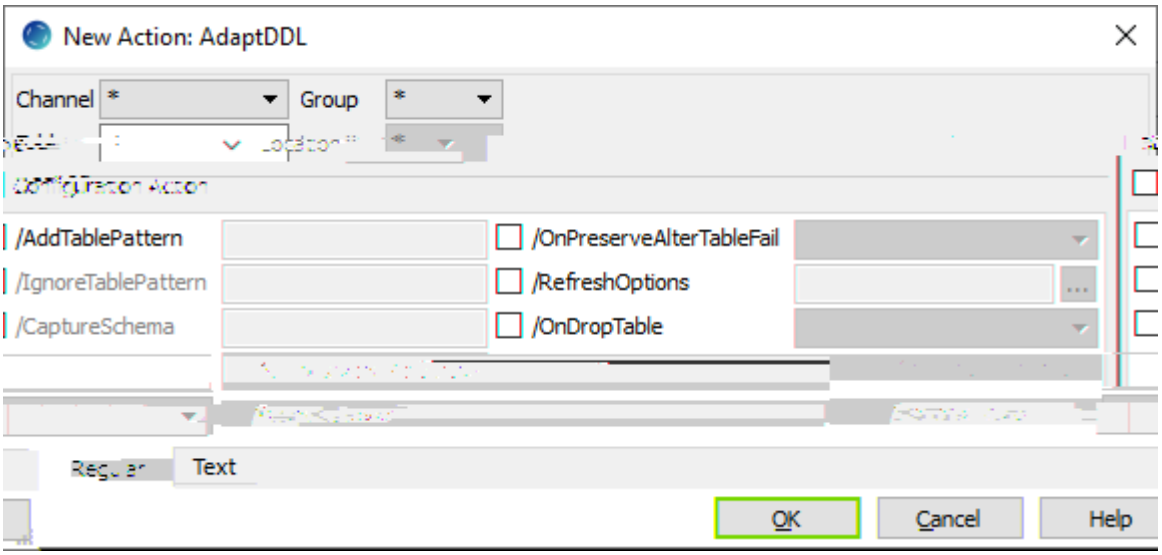
AdaptDDL

Contents

Description

Normally HVR only handles database DML statements (such as **insert**, **update** and **delete**). Action **AdaptDDL** causes HVR to also react to DDL statements such as **create table**, **drop table**, **alter table** ... **add column** or **drop column**.

This action should normally be defined on both the the capture location and the integrate location. When on the capture database, the capture job will react to DDL changes to tables already in the channel by changing the column information in the HVR catalogs. If parameter **/AddTablePattern** is defined it will also add new tables to the channel. If the action is also defined on the integrate database then the capture job will then apply these DDL changes to the integrate



Parameter	Argument	Description
/AddTablePattern	*patt*	Add new tables to channel if the new table name matches *patt*. If this parameter is not defined then new tables are never be added to the channel. Patterns can include wildcards (***** or **o?_line_***) or ranges (**ord_[a-f]**). For a list patterns, either use a pattern containing a **|** symbol (example, **tmp*|temp***) or defining multiple **AdaptDDL /AddTablePattern** actions. This action should be defined on **Table *** (all tables) and on typically on both capture and integrate locations. If **/CaptureSchema** is not defined then this table must be in the location's 'current' schema. A table will not be replicated twice, even if it matches multiple **AdaptDDL /AddTablePattern** actions. This parameter is only effective when defined on a capture location. This parameter is not supported for certain databases. For the list of supported databases, see Log-based capture of DDL statements using action AdaptDDL in Capabilities.
/IgnoreTablePattern	*patt*	Ignore a new table despite it matching a pattern defined by **/AddTablePattern**. The styles of pattern matching is the same as the pattern of **/AddTablePattern**. This parameter only affects tables matched by the **/AddTablePattern** parameter on the same **AdaptDDL** action, not those matched by other **/AddTablePattern** parameters. For example a channel defined with these actions: • **AdaptDDL /AddTablePattern=t* /IgnoreTablePattern=tmp*|temp*** • **AdaptDDL /AddTablePattern=*x** This channel will automatically add to tables **tab_1** and **tmp_x** but not table **tmp_y**. This parameter is only effective when defined on a capture location. This parameter is not supported for certain databases. For the list of supported databases, see Log-based capture of DDL statements using action AdaptDDL in Capabilities.
/CaptureSchema	*schema*	This parameter controls which schema's new tables are matched by **/AddTablePattern**. Value *schema* is not a pattern (no '*****' wildcards) but it is case-insensitive. If this parameter is not defined then the only new table that are matched are those in the location's 'current' or 'default' schema. When a new table is added using this parameter then the HVR capture job will also generate **TableProperties/Schema** action(s), unless the schema is the capture location's current schema. This parameter is only effective when defined on a capture location. This parameter is not supported for certain databases. For the list of supported databases, see Log-based capture of DDL statements using action AdaptDDL in Capabilities.
/IntegrateSchema	*schema*	This parameter allows a new table which is matched in a schema on a capture database defined with **/CaptureSchema** to be sent to a schema different from the default schema on an integrate database. One or more mappings to be defined. So when a channel has action **AdaptDDL /AddTablePattern="*" /CaptureSchema=aa1 /IntegrateSchema=bb1** and action **AdaptDDL /AddTablePattern="*" /CaptureSchema=aa2 /IntegrateSchema=bb2** then table **aa1.tab** would be created in the integrate database as **bb1.tab** whereas table **aa2.tab** would be created in the target database as **bb2.tab**. Each table would be added to the channel with two **TableProperties /Schema** actions; one on the capture location and one on the integrate location. This parameter is only effective when defined on a capture location, even though it actually causes actions to be generated on the integrate location group(s). This parameter is not supported for certain databases. For the list of supported databases, see Log-based capture of DDL statements using action AdaptDDL in Capabilities.

/OnEnrollBreak **Since** v5.6.5/13	*pol*	This parameter applies policy *pol* to control the behavior of capture job (whether to execute **HVR Refresh** because of a DDL change) for an existing table if there is a break in the enroll information (like data type changes, partition changes etc.). This parameter is only effective if defined on the target location where the **HVR Refresh** would load data into. This parameter does not control the behavior of a capture job for a new table being added to the channel. Available values for *pol* are: • **REFRESH** (**default**): Refresh the table. • **FAIL_INTEG_JOB**: Send a breakpoint control to all involved integrate jobs. Once all changes up to the DDL sequence are integrated, the control will cause the integrate job to fail with an error. The problem must be solved manually and the control must be removed manually. • **WARNING**: Issue a warning and then continue without the **HVR Refresh**. If necessary, perform only the **alter table.**
/OnPreserveAlterTableFail **Since** v5.6.5/13	*pol*	This parameter applies policy *pol* to control the behavior of capture job for an existing table to handle any failure while performing **alter table** on the target table. This parameter is only effective if defined on the target location where the **alter table** is being performed. Available values for *pol* are: • **CREATE_AS_SELECT** (**default**): Move existing table to a temporary table and create new table with new layout as selected from old table. • **FAIL_INTEG_JOB**: Send a breakpoint control to all involved integrate jobs. Once all changes up to the DDL sequence are integrated, the control will cause the integrate job to fail with an error. The problem must be solved manually and the control must be removed manually. • **WARNING**: Issue a warning and then continue replication without retrying to perform the **alter table**.
/RefreshOptions	*refropts*	Configure which **HVR Refresh** options the capture job should used to create or alter the target table(s) and (when necessary) re-populate the data. Value *refropts* is a list of option letters, separated by spaces. Possible options are: **-f**: Fire database triggers/rules while applying SQL changes for with row-wise refresh. The default behavior is that database trigger/rule firing is disabled during refresh. For Oracle and SQL Server, this is avoided by disabling and re-enabling the triggers. Requires **-gr** (row-wise refresh). **-g** *x*: Granularity of refresh. Parameter *x* can be either: • **b** (**default**) : Bulk refresh using bulk data load. • **r** : Row–wise refresh. **-m** *mask*: Mask (ignore) some differences between the tables that are being compared. Parameter *mask* should be composed of one of these letters: • **d** : Mask out delete differences. • **u** : Mask out update differences. • **i** : Mask out insert differences. Letters can be combined, for example **-mid** means mask out inserts and deletes. If a difference is masked out, then the refresh will not rectify it. Requires **-gr** (row-wise refresh). **-p** *N*: Perform refresh into multiple locations in parallel using *N* sub–processes. Not effect if only one integrate location. **-v**: Verbose. This causes row–wise refresh to display each difference detected. Differences are presented as SQL statements. Requires **-gr** (row-wise refresh). All refreshes implied by **AdaptDDL** use context **adaptddl** (like **hvrrefresh -Cadaptddl**) so data truncated and selected can be controlled using action **Restrict** with **/Context**=**adaptddl**. This parameter is only effective when defined on a integrate location.
/OnDropTable	*pol*	Policy *pol* controls behavior if a **drop table** is done to a replicated table. Available values for *pol* are: • **KEEP**: Table will remain in channel. Capture job will write a warning message in log. The next **hvrinit** will give error ('table not found') when it attempts to regenerate enroll information for this channel. • **DROP_FROM_CHANNEL_ONLY** (**default**): Table (and its actions) are deleted from catalogs only, but the table is left in any target databases. • **DROP_FROM_CHANNEL_AND_TARGET**: Table (and all its actions) are deleted from catalogs and the target table is dropped from the target databases. Defining the parameter on the capture location controls whether the table is dropped from the channel catalogs, whereas defining it on the integrate location controls whether the target table is dropped. Note that, if this is the last table in the channel then HVR will not drop it from the catalog, instead the capture job will fail because an HVR channel must always contains at least one table.
/KeepExistingStructure		Preserve old columns in target, and do not reduce data types sizes. This means if an **alter table** statement was done on the capture table to drop a column or make it smaller (e.g. **varchar(12)** to **varchar(5)**) this will not be propagated to the target table. This can used to protect historical data, which could have been purge of the capture database was not replicated (using **Capture /IgnoreSessionNames**) or if the integrate table contains a row for each capture change (**Column Properties /TimeKey**). This parameter is only effective when defined on a integrate location.
/KeepOldRows **Since** v5.6.5/12		Preserve old/existing rows (**hvrrefresh -cp**) in target table if the table is dropped and recreated with a new layout during **HVR Refresh**.

Behavior for Specific DDL Statements and Capture DBMSs

DDL SQL Statement	Behavior without AdaptDDL	Behavior if AdaptDDL is defined	Notes for specific capture DBMS
create table	Capture job ignores DDL. Operator must manually perform 'Adapt steps' (including **Table Explore** and **HVR Refresh**) to add table to channel.	If new table is not in channel but the capture location has action **AdaptDDL** with a matching **/AddTablePattern** then the table is added to the channel and supplemental logging is enabled (if necessary). If integrate database(s) also have action **AdaptDDL** then the capture job will do an HVR refresh which will also create the table in the target database(s).This refresh should be quick because the new table should be empty or at least very small. If the table already existed in the integrate database it will be recreated or an **alter table** used to make its columns match. If **/AddTablePattern** is not defined or the table name does not match then this DDL statement is ignored.	
drop table	If a table was in the channel then capture job will write a warning message in log. The next **hvrinit** will give error ('table not found') when it attempts to regenerate enroll information for this channel.	If the table in is the channel then the behavior depends on value of **AdaptDDL** parameter **/OnDropTable**. Possible values are: • **KEEP**: Table will remain in channel. capture job will write a warning message in log. The next **hvrinit** will give error ('table not found') when it attempts to regenerate enroll information for this channel. • **DROP_FROM_CHANNEL_ONLY** (**default**): Table (and its actions) are deleted from catalogs only, but the table is left in any target database(s). • **DROP_FROM_CHANNEL_AND_TARGET**: Table (and all its actions) are deleted from catalogs and the target table is dropped from the target database(s). Defining the parameter on the capture location controls whether the table is dropped from the channel catalogs, whereas defining it on the integrate location controls whether the target table is dropped. Note that if this is the last table in the channel then HVR will not drop it from the catalog. Instead the capture job will fail, because an HVR channel must always contains at least one table. If the value is **KEEP** or **DROP_FROM_CHANNEL_ONLY** and the table is created again in the capture database, then the old table in the integrate database will be reused; it will be recreated or an **alter table** done to make its columns match.	For SQL Server, this is not allowed if **Capture /SupplementalLog=ARTICLE_OR_CDCTAB** if the table has a primary key because when HVR is capturing a table, the **drop table** statement gives "Cannot drop table … because it is being used for replication" [error 3724].
create table, followed quickly by **drop table**	Both DDL statements are ignored.	If the **drop table** is already complete by the time the capture job encounters the first **create table** in the DBMS logging then the capture job will ignore both DDL statements. If the **drop table** occurs after the capture job has finished processing the **create table** statement then each DDL statement will processed individually (see lines above). But if the **drop table** occurs while the capture job is still processing the **create table** statement then its refresh may fail with a 'table not found' error. But the capture job will then retry and succeed, because the **drop table** is already complete (see above).	
drop table, followed quickly by **create table**	Capture job will write a warning message when it sees the **drop table** and when it sees **create table** it will update its internal enroll information so that it can still parse new values.	If the **create table** is already complete by the time the capture job encounters the first **drop table** in the DBMS logging then the capture job will refresh the table again, because there may be updates to the newly recreated table which HVR cannot process because supplemental logging had no been created yet. It will then update its internal enroll information so that it can still parse new values. If the **create table** is has not happened by the time the capture job encounters the first **drop table** then these statements will be processed individually.	

alter table ... add column – without a specified default value clause	New column will be ignored; it won't be added to target and its value won't be replicate or refreshed. But replication of other columns continues normally. Subsequent **hvrinit** or **hvrrefresh** commands will also work normally.	Capture job will add the column to the channel catalogs. If an integrate database(s) has action **AdaptDDL** then the capture job's behavior will do an **alter table** to add the column to the table in the target database(s). For some DBMSs the capture job will then refresh the data into the integrate location(s). Then replication will resume.	For Oracle and SQL Server HVR will not refresh the data and just continue replication.
alter table ... add column – with a specified default value clause	Same as regular **alter table ... add column** above.	Same as regular **alter table ... add column** above, except the target table will just get an **alter table ... add column** with a default value defined by HVR. This means existing columns will have the HVR default value instead of the default value from the original **alter table ... add column** statement. Newly replicated values will get the correct value from the source.	Same as for regular **alter table ... add column**.
alter table ... drop column	Capture job will only update its internal enroll information so that it can still parse new values. If this was a key column or it was not nullable and had no default then then integrate errors will start to occur.	Capture job will drop the column from the channel catalogs. If an integrate database(s) has action **AdaptDDL** then the capture job will use **alter table** to drop the column to the table in the target database (s), unless **/KeepExistingStructure** is defined. In this case the columns is kept in the target. For some DBMSs the capture job will then refresh the data into the integrate location(s). Then replication will resume.	For Oracle and SQL Server HVR will not refresh the data and just continue replication. Note: Both **alter table ... add column** and **alter table ... drop column** usually resume replication without a refresh. However, there are some exceptions to this rule. If a column was dropped and then added again, HVR needs to refresh the data to assure that all data is replicated correctly. Additionally, for SQL Server locations which have **Capture /LogReadMethod** set to **SQL**, dropping a column will cause a refresh to prevent potential issues with the ongoing capture.

alter table … modify column – to make column 'bigger', e.g., **varchar(5)** to **varchar(12)**.	Capture job will only update its internal enroll information so that it can still parse new values. But when a new large value is captured it will either cause an error in the integrate job, or (if **TableProperties /IgnoreCoerceError** is defined) it will be truncated.	Capture job will change the column's information from the channel catalogs. If an integrate database(s) has action **AdaptDDL** then the capture job's will do an **alter table** to change the target column's width. No refresh will be done to the target table. Then replication will resume.	
alter table … modify column – to make column 'smaller', e.g., **varchar(12)** to **varchar(5)**.	Capture job will only update its internal enroll information so that it can still parse new values. No errors.	Capture job will change the column's information from the channel catalogs. If an integrate database(s) has action **AdaptDDL** then the capture job's will do an **alter table** to change the target column's width, unless **/KeepExistingStructure** is defined. The capture job will then refresh the target table. Then replication will resume.	
alter table ... modify column – to change 'data type', e.g., **number** to **varchar(5)**.	Capture job will only update its internal enroll information so that it can still parse new values. But when a new value is captured the integrate job may give an error if it cannot convert the new value into the target's old data type.	Capture job will change the column's information in the channel catalogs. If an integrate database(s) has action **AdaptDDL** then the capture job will either do an **alter table** to drop the column to the table in the target database(s)., or if **alter table** in the target DBMS cannot change data types then the table will be dropped and recreated. The capture job will then refresh the target table. Then replication will resume.	
alter table ... modify column – to change 'encryption', e.g., enable encryption or change encryption algorithm.	Capture job will warn that the channel definition should be upgraded and and a refresh should be done. It will also give an error because it cannot handle the encrypted columns correctly.	Capture job will change the column's encryption information in its internal enroll information. It will then refresh the target table. Then replication will resume. The capture job will not replicate the encryption setting change to the target table.	This is supported only on Oracle 11 and higher. For more information on HVR's support of Oracle's encryption feature (TDE) see the TDE section in the Requirements for Oracle.
alter table ... rename column	Capture job will only update its internal enroll information so that it can still parse new values. If this was a key column or it was not nullable and had no default then then integrate errors will start to occur.	Capture job will change the table's information in the channel catalogs. If an integrate database(s) has action **AdaptDDL** then the capture job will either do an **alter table** to rename the column to the table in the target database(s)., or if **alter table** in the target DBMS cannot rename columns then the table will be dropped and recreated. The capture job will then refresh the target table. Then replication will resume.	SQl Server does not support **alter table ... rename column** but it uses the build in function **sp_rename** .

truncate table	HVR captures this as a special DML statement (**hvr_op=5**), unless **Capture /NoTruncate** is defined. This changes is applied as **truncate table** by the integrate job, unless **Restrict /RefreshCondition** is defined there.	HVR captures this as a special DML statement (**hvr_op=5**), unless **Capture /NoTruncate** is defined. This changes is applied as **truncate table** by the integrate job, unless **Restrict /RefreshCondition** is defined there.	
alter index ... on..rebuild – in online mode, e.g., with **(online=on)** SQL Server only	Capture job will only update its internal enroll information so that it can still parse new values.	Capture job will only update its internal enroll information so that it can still parse new values.	
alter table ... add constraint ... primary key **create unique index** **create index ... local (partition ...)** **drop index**	Ignored. But if a uniqueness constraint is relaxed on the capture database (for example if the primary key gets an extra column) then a uniqueness constraint violation error could occur during integration	HVR only maintains a single key (the "replication key") in the HVR channel catalogs and on the target tables. If there are multiple uniqueness constraints on the the capture table (e.g. a primary key and several unique indexes) then HVR uses a hierarchy rule to decide which is its replication key (e.g. a primary key would 'win'). When the capture job encountered this DDL statement then it will re-inspect the capture table and see if its 'replication key' has now changed. If it has then the capture job will change the channel catalogs to either add, remove or change this 'replication index'. If integrate database(s) also have action AdaptDDL then the capture job will change the 'replication index' on the target table in the target database(s). The index 'name' and other attributes (such as 'fill factor') are ignored, as are other 'secondary' indexes on the capture table. No refresh is needed.	
alter table ... add foreign key	Ignored. Replication continues correctly.	Ignored. Replication continues correctly. DDL not replicated /propagated to target database(s).	
alter table ... rename to ... – Rename table	Capture job will write a warning message in log. The next **hvrinit** will give error ('table not found') when it attempts to regenerate enroll information for this channel.	This is treated like a **drop table** and a **create table**. So the old name is deleted from the catalogs and added to the target depending on parameter **/OnDropTable** (see above). If the new table name matches **/AddTablePattern** then it is added to the channel. If integrate database(s) also have action **AdaptDDL** then the capture job will do an HVR refresh which will also create the new table name in the target database(s).	
alter table … truncate partition	Ignored. The deletes implied by this DDL statement will not be replicated.	If an integrate database(s) has action **AdaptDDL** then the capture job will refresh the target table. Then replication will resume.	
alter table... merge partition	Ignored. Replication continues correctly.	Ignored. Replication continues correctly. DDL not replicated /propagated to target database(s).	
alter table ... split partition	Ignored. Replication continues correctly.	Ignored. Replication continues correctly. DDL not replicated /propagated to target database(s).	
alter table ... exchange partition	Ignored. The changes implied by this DDL statement will not be replicated.	If an integrate database(s) has action **AdaptDDL** then the capture job will refresh the target table. Then replication will resume.	

alter table ... move tablespace	Ignored. Replication continues correctly.	Ignored. Replication continues correctly. DDL not replicated /propagated to target database(s).	
alter tablespace ...	Ignored. Replication continues correctly.	Ignored. Replication continues correctly. DDL not replicated /propagated to target database(s).	
create sequence	Changes captured and integrated if action **DbSequence** is defined. See that action for limitations.	Changes captured and integrated if action **DbSequence** is defined. See that action for limitations.	
drop sequence	Ignored.	Ignored.	
create/drop view create/drop synonym	Ignored. Replication continues correctly.	Ignored. Replication continues correctly. DDL not replicated /propagated to target database(s).	
create/drop trigger	Ignored. Replication continues correctly.	Ignored. Replication continues correctly. DDL not replicated /propagated to target database(s).	
create/drop procedure create/alter /drop function create/alter /drop user create/alter /drop role create/drop directory	Ignored. Replication continues correctly.	Ignored. Replication continues correctly. DDL not replicated /propagated to target database(s).	
dbms_redefintion – to change tables storage (partitioning, compression, tablespace, LOB storage etc..) but not information stored in HVR catalogs (column names, data types or key)	Capture job will only update its internal enroll information so that it can still parse new values.	HVR recognizes Oracle **dbms_redefintion** because it sees that the create time is same but the table id has changed. HVR assumes that no other zero other DDL (alter table) subsequently. in which case no refresh needed. Enroll information will be updated and capture will continue.	This is supported only on Oracle.
dbms_redefintion – to change tables storage (partitioning, compression, tablespace, LOB storage etc..) but not info stored in HVR catalogs (column names, data types or key), followed by an alter table to change other column information.	Capture job will only update its internal enroll information, and will treat the subsequent DDL statement individually.	HVR recognizes Oracle **dbms_redefintion** because it sees that the create time is same but the table id has changed. HVR assumes (incorrectly) that no other zero other DDL (alter table) subsequently. so it neglects to do a refresh.	This is supported only on Oracle.

dbms_redefintion – which changes information in the HVR catalogs (the column names, data types or primary key)	See row above showing behavior for specific **alter table** type.	See row above showing behavior for specific **alter table** type.	This is supported only on Oracle.

Use of Capture Rewind with AdaptDDL

Problems can occur when **Capture Rewind** is used to go back to a time before a DDL statement changed a replicated table.

Background: The capture job parses its tables changes (called 'DML') using 'enroll information' which is created by **HVR Initialize**. This has an **Advanced Option** called **Table Enrollment** (option **-oe**) can be used to either (a) not regenerate this enroll information or to (b) only regenerate this enroll information. When the capture job encounters a DDL statement it will re-inspect the table and save the table's new structure as a 'revision' to its original enrollment information. This will help it process subsequent DML statements from the logging.

But if **Capture Rewind** is used with **HVR Initialize** then the 'original' enrollment information created by that command may be newer than the DML changes that the capture job must parse. If a DDL statement (such as **alter table ... drop column**) was performed between the 'rewind' point where the capture job must start parsing and the moment when **HVR Initialize** generated the enrollment information, the capture job may fail when fail if it encounters a DML record using the old table structure. Such errors will no longer happen after the capture job encounters the actual DDL statement or after it passes the moment that **HVR Initialize** was run.

If the channel already existed then one tactic to avoid such capture errors is to not regenerate existing enroll information when using **HVR Initialize** for **Capture Rewind**. But this could cause a different error, if a DDL statement happened after the 'old' capture job stopped running and before the new rewind point.

Restrictions

- Capturing DDL changes using **AdaptDDL** is not supported for certain databases. For the list of supported databases, see Log-based capture of DDL statements using action AdaptDDL in Capabilities.
- **AdaptDDL** cannot be used together with **Transform /SapXForm**.
- For Oracle, **AdaptDDL** is not supported if the log read method is **SQL** (**Capture /LogReadMethod**=**SQL**).
- For SQL Server, **AdaptDDL** is supported if the log read method is **SQL** (**Capture /LogReadMethod**=**SQL**) since HVR 5.2.3/0.

See Also

- Manually Adapting a Channel for DDL Statements

AgentPlugin

Contents

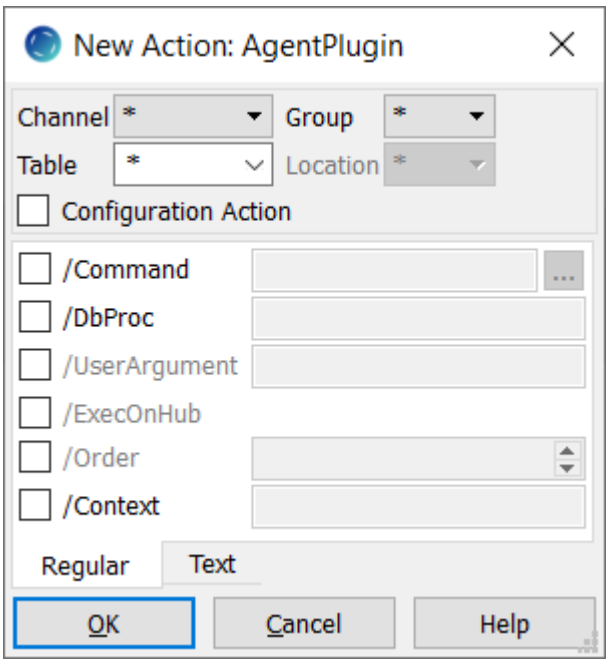
Description

An agent plugin is a block of user–supplied logic which is executed by HVR during replication. An agent plugin can be an operating system command or a database procedure. Each time HVR executes an agent plugin it passes parameters to indicate what stage the job has reached (e.g. start of capture, end of integration etc.). If action **AgentPlugin** is defined on a specific table, then it affects the entire job including data from other tables for that location.

> Since HVR 5.6.5/13, HVR will only execute binaries and scripts available inside **$HVR_HOME/lib/agent** or **$HVR_HOME/lib/transform**. So, it is recommended to save custom scripts/agent plugins in these directories.
>
> HVR can also execute binaries and scripts available inside other directories if they are whitelisted. Directories can be whitelisted by defining the property **Allowed_Plugin_Paths** in file **$HVR_HOME/lib/hvraccess.conf**. For reference, the sample configuration file **hvraccess.conf_example** can be found in the same directory.

Parameters

This section describes the parameters available for action **AgentPlugin**.



Parameter	Argument	Description

/Command	*path*	Name of the agent plugin command. This can be a script or an executable. Scripts can be shell scripts on Unix and batch scripts on Windows or can be files beginning with a 'magic line' (shebang) containing the interpreter for the script e.g. **#! perl**. Argument *path* can be an absolute or a relative pathname. If a relative pathname is supplied the agents should be located in **$HVR_HOME/lib/agent** or in a directory specified with parameter **/Path**. This field is disabled when **/DbProc** is selected.
/DbProc	*dbproc*	Call database procedure *dbproc* during replication jobs. The database procedures are called in a new transaction; changes that do not commit themselves will be committed after agent plugin invocation by the HVR job. **/DbProc** cannot be used with parameters **/Command**, **/ExecOnHub**, and *path*. This field is disabled when **/Command** is selected.
/UserArgument	*userarg*	Pass extra argument *userarg* to each agent plugin execution.
/ExecOnHub		Execute agent plugin on hub machine instead of location's machine. This field is disabled when **/DbProc** is selected.
/Order	*int*	Order of executing the agent plugin.
/Path	*dir*	Search directory *dir* for agent plugin. This field is disabled when **/DbProc** is selected. This parameter is not available since HVR version 5.6.5/13.
/Context	*context*	Action only applies if **Refresh**/**Compare** *context* matches

Agent Plugins in HVR Distribution

By default, HVR distribution contains the following agent plugins:

- Agent Plugin for BigQuery
- Agent Plugin for Cassandra
- Agent Plugin for Manifest Files
- Agent Plugin for MongoDB

Agent Plugin Arguments

If an agent plugin is defined, it is called several times at different points of the replication job. On execution, the first argument that is passed indicates the position in the job, for example **cap_begin** for when the agent plugin is called before capture.

Argument *mode* is either **cap_begin**, **cap_end**, **integ_begin**, **integ_end**, **refr_read_begin**, **refr_read_end**, **refr_write_begin** or **refr_write_end** depending on the position in the replication job where the agent plugin was called. Agent plugins are not called during **HVR Compare**.

Modes **cap_end** and **integ_end** are passed information about whether data was actually replicated.

Command agent plugins can use **$HVR_TBL_NAMES** or **$HVR_FILE_NAMES** and database procedure agent plugins can use parameter **hvr_changed_tables**. An exception if an integrate job is interrupted; the next time it runs it does not know anymore which tables were changed so it will set these variables to an empty string or **-1**.

Agent plugins specified with **/Command** are called as follows:

```
agent mode chn_name loc_name userarg
```

Database procedure agents (specified in **/DbProc**) are called as follows:

- In Ingres,

```
execute procedure agent (hvr_agent_mode='mode', hvr_chn_name='chn', hvr_loc_name='
locname', hvr_agent_arg='userarg', hvr_changed_tables=N);
```

- In Oracle,

```
agent (hvr_agent_mode$=>'mode', hvr_chn_name$=>'chn', hvr_loc_name$=>'locname',
hvr_agent_arg$=>'userarg', hvr_changed_tables$=N);
```

- In SQL Server,

```
execute agent @hvr_agent_mode='mode', @hvr_chn_name='chn', @hvr_loc_name='locname',
@hvr_agent_arg='userarg', @hvr_changed_tables=N;
```

> The parameter **hvr_changed_tables** specifies the number (*N*) of tables that were changed.

Agent Plugin Interpreter

If the agent plugin is a script, HVR will consider its shebang line to execute it with an interpreter. It is recommended that only the interpreter program name is specified here (for example, **#!perl** or **#!python**). It is not required to specify the absolute path in the shebang line. HVR will automatically determine the path for the specified interpreter using the environment variable **PATH**.

Agent Plugin Environment

An agent plugin inherits the environment of its parent process. On the hub, the parent of the agent plugin's parent process is the **HVR Scheduler**. On a remote Unix machine it is the **inetd** daemon. On a remote Windows machine it is the HVR Remote Listener service. Differences with the environment of the parent process are as follows:

- Environment variable **$HVR_TBL_NAMES** is set to a colon–separated list of tables for which the job is replicating (for example **HVR_TBL_NAMES=tbl1:tbl2:tbl3**). Also variable **$HVR_BASE_NAMES** is set to a colon–separated list of table 'base names', which are prefixed by a schema name if **/Schema** is defined (for example **HVR_BASE_NAMES=base1:sch2.base2:base3**). For modes **cap_end** and **integ_end** these variables are restricted to only the tables actually processed. Environment variables **$HVR_TBL_KEYS** and **$HVR_TBL_KEYS_BASE** are colon–separated lists of keys for each table specified in **$HVR_TBL_NAMES** (e.g. **k1,k2:k:k3,k4**). The column list is specified in **$HVR_COL_NAMES** and **$HVR_COL_NAMES_BASE**.
- Environment variable **$HVR_CONTEXTS** is defined with a comma–separated list of contexts defined with HVR Refresh or Compare (option **–C***ctx*).
- Environment variables **$HVR_VAR**_*XXX* are defined for each context variable supplied to HVR Refresh or Compare (option **–V***xxx*=*val*).
- For database locations, environment variable **$HVR_LOC_DB_NAME**, **$HVR_LOC_DB_USER** (unless no value is necessary).

- For Oracle locations, the environment variables **$HVR_LOC_DB_USER**, **$ORACLE_HOME** and **$ORACLE_SID** are set and **$ORACLE_HOME/bin** is added to the path.
- For Ingres locations the environment variable **$II_SYSTEM** is set and **$II_SYSTEM/ingres/bin** is added to the path.
- For SQL Server locations, the environment variables **$HVR_LOC_DB_SERVER**, **$HVR_LOC_DB_NAME**, **$HVR_LOC_DB_USER** and **$HVR_LOC_DB_PWD** are set (unless no value is necessary).
- For file locations variables **$HVR_FILE_LOC** and **$HVR_LOC_STATEDIR** are set to the file location's top and state directory respectively. For modes **cap_end** and **integ_end** variable **$HVR_FILE_NAMES** is set to a colon–separated list of replicated files, unless this information is not available because of recovery. For mode **integ_end**, the following environment variables are also set: **$HVR_FILE_NROWS** containing colon-separated list of number of rows per file for each file specified in **$HVR_FILE_NAMES** (for example **HVR_FILE_NROWS=1005:1053:1033**); **$HVR_TBL_NROWS** containing colon-separated list of number of rows per table for each table specified in **$HVR_TBL_NAMES**; **$HVR_TBL_CAP_TSTAMP** containing colon-separated list of first row's capture timestamp for each table specified in **$HVR_TBL_NAMES**; **$HVR_TBL_OPS** containing colon-separated list of comma-separated *hvr_op=count* pairs per table for each table specified in **$HVR_TBL_NAMES** (for example **HVR_TBL_OPS=1=50,2=52:1=75,2=26:1=256**). If the number of files or tables replicated are extremely large then these values are abbreviated and suffixed with "**...**". If the values are abbreviated, refer to **$HVR_LONG_ENVIRONMENT** for the actual values.
- Environment variables with too long values for operating system are abbreviated and suffixed with "**...**". If the values are abbreviated, HVR creates a temporary file containing original values of these environment variables. The format for this temporary file is a JSON map consisting of key value pairs and the absolute path of this file is set in **$HVR_LONG_ENVIRONMENT**.
- Any variable defined by action **Environment** is also set in the agent plugin's environment.
- The current working directory for local file locations (not FTP, SFTP, SharePoint/WebDAV, HDFS or S3) is the top directory of the file location. For other locations (e.g. database locations) it is **$HVR_TMP**, or **$HVR_CONFIG/tmp** if this is not defined.
- **stdin** is closed and **stdout** and **stderr** are redirected (via network pipes) to the job's logfiles.

If a command agent plugin encounters a problem it should write an error message and return with exit code 1, which will cause the replication job to fail. If the agent does not want to do anything for a mode or does not recognize the mode (new modes may be added in future HVR versions) then the agent should return exit code 2, without writing an error message.

Examples

This section lists few examples of agent plugin scripts:

- Example 1: An agent plugin script (in Perl), which prints "hello world"
- Example 2: An agent plugin script (in Perl), which prints out arguments and environment at the end of every integrate cycle
- Example 3: An agent plugin script (in Python), which utilizes $HVR_LONG_ENVIRONMENT to print environment variables at the end of every integrate cycle
- Example 4: A database procedure agent plugin that populates table order_line after a refresh.

Example 1: An agent plugin script (in Perl), which prints "hello world"

```
#!perl
# Exit codes: 0=success, 1=error, 2=ignore_mode
if($ARGV[0] eq "cap_begin") \{
    print "Hello World\n";
    exit 0;
\}
else \{
    exit 2;
\}
```

Example 2: An agent plugin script (in Perl), which prints out arguments and environment at the end of every integrate cycle

```
#!perl
require 5;
if ($ARGV[0] eq "integ_end")
\{
   print "DEMO INTEGRATE END AGENT (";
   foreach $arg (@ARGV) \{
       print "$arg ";
   \}
   print ")\n";
   # print current working directory
   use Cwd;
   printf("cwd=%s\n", cwd());
   # print (part of the) environment
   printf("HVR_FILE_NAMES=$ENV\{HVR_FILE_NAMES\}\n");
   printf("HVR_FILE_LOC=$ENV\{HVR_FILE_LOC\}\n");
   printf("HVR_LOC_STATEDIR=$ENV\{HVR_LOC_STATEDIR\}\n");
   printf("HVR_TBL_NAMES=$ENV\{HVR_TBL_NAMES\}\n");
   printf("HVR_BASE_NAMES=$ENV\{HVR_BASE_NAMES\}\n");
   printf("HVR_TBL_KEYS=$ENV\{HVR_TBL_KEYS\}\n");
   printf("HVR_TBL_KEYS_BASE=$ENV\{HVR_TBL_KEYS_BASE\}\n");
   printf("HVR_COL_NAMES=$ENV\{HVR_COL_NAMES\}\n");
   printf("HVR_COL_NAMES_BASE=$ENV\{HVR_COL_NAMES_BASE\}\n");
   printf("PATH=$ENV\{PATH\}\n");
   exit 0; # Success
\}
else
\{
   exit 2; # Ignore mode
\}
```

Example 3: An agent plugin script (in Python), which utilizes $HVR_LONG_ENVIRONMENT to print environment variables at the end of every integrate cycle

```
#!python

import os
import sys
import json

if __name__ == "__main__":
    if sys.argv[1] == 'integ_end':
        if 'HVR_LONG_ENVIRONMENT' in os.environ:
            with open(os.environ['HVR_LONG_ENVIRONMENT'], 'r') as f:
                long_environment= json.loads(f.read())
        else:
            long_environment= \{\}  # empty dict

        # print (part of the) environment
        if 'HVR_FILE_NAMES' in long_environment:
            print 'HVR_FILE_NAMES=\{0\}'.format(long_environment['HVR_FILE_NAMES'])
        elif 'HVR_FILE_NAMES' in os.environ:
            print 'HVR_FILE_NAMES=\{0\}'.format(os.environ['HVR_FILE_NAMES'])
        else:
            print 'HVR_FILE_NAMES=<not set>'
        if 'HVR_TBL_NAMES' in long_environment:
            print 'HVR_TBL_NAMES=\{0\}'.format(long_environment['HVR_TBL_NAMES'])
        elif 'HVR_TBL_NAMES' in os.environ:
            print 'HVR_TBL_NAMES=\{0\}'.format(os.environ['HVR_TBL_NAMES'])
        else:
            print 'HVR_TBL_NAMES=<not set>'

        if 'HVR_BASE_NAMES' in long_environment:
            print 'HVR_BASE_NAMES=\{0\}'.format(long_environment['HVR_BASE_NAMES'])
        elif 'HVR_BASE_NAMES' in os.environ:
            print 'HVR_BASE_NAMES=\{0\}'.format(os.environ['HVR_BASE_NAMES'])
        else:
            print 'HVR_BASE_NAMES=<not set>'

        sys.exit(0)  # Success
    else:
        sys.exit(2)  # Ignore mode
```

Example 4: A database procedure agent plugin that populates table order_line after a refresh.

```
create procedure [dbo].[refr_agent] (
   @hvr_agent_mode varchar(20),
   @hvr_chn_name varchar(20),
   @hvr_loc_name varchar(20),
   @hvr_agent_arg varchar(20),
   @hvr_changed_tables numeric
) as
begin
   if @hvr_agent_mode = 'refr_write_begin'
   begin
       begin try
           delete from order_line
       end try
       begin catch
          -- ignore errors; nothing to delete
       end catch
   end
   else if @hvr_agent_mode = 'refr_write_end'
   begin
       insert into order_line
           SELECT i.item_id,
             i.item_no,
             i.description,
             i.attribute,
             i.item_type,
             p.id,
             p.date_set,
             p.price
             FROM items i, price p
   end
end
```

Agent Plugin for BigQuery

Contents

Name

hvrbigqueryagent.py

Synopsis

hvrbigqueryagent.py *mode chn loc* [*–options*]

Description

The agent plugin **Hvrbigqueryagent** enables HVR to replicate data into BigQuery database. This agent plugin should be defined in the HVR channel using action **AgentPlugin**. The behaviour of this agent plugin depends on the *–options* supplied in **/UserArgument** field of **AgentPlugin** screen.

This agent plugin supports replication of data in Avro format only.

For better performance it is recommended to install HVR remote listener on VM (virtual machine) located in Google Cloud and use the HVR transfer protocol with compression when using BigQuery for replication.

Options

This section describes the parameters that can be used with **Hvrbigqueryagent**:

Parameter	Description
-r	Truncates existing data from target and then recreates table and insert new rows. If this option is not defined, appends data into table.
-s *col_name*	Soft deletes the column *col_name*.

Environment Variables

The **Environment** variables listed in this section should be defined when using this agent plugin:

Environment Variable Name	Description
$HVR_GBQ_CREDFILE	The directory path for the credential file. The default directory path in - • Linux: **$HOME/.config/gcloud/application_default_credentials.json** • Windows: **Users/**<*user name*>**/.config/gcloud/application_default_credentials.json**

$HVR_GBQ_PROJECTID	The Project ID in BigQuery. This is the Project ID of the dataset being used for replication.
$HVR_GBQ_DATASETID	The Dataset ID in BigQuery. This dataset should belong to the Project ID defined in $HVR_GBQ_PROJECTID.

Installing Python Environment and BigQuery Client

Google BigQuery client is required for uploading data into BigQuery from local source and convert it into tables.

To enable data upload into BigQuery using HVR, perform the following on HVR **Integrate** machine:

1. Install Python 2.7.x +/3.x. Skip this step if the mentioned python version is already installed in the machine.
2. Install the following python client modules:

```
pip install google_cloud_bigquery
pip install enum34
```

> ⚠ Installing enum34 is not required for python versions 3.4 and above.

3. Pass authorization process with Google:

```
gcloud auth application-default login
```

4. Copy configuration file (location is differ on different platforms, see below) into integration side.

   Linux, **$HOME/.config/gcloud/application_default_credentials.json**

   Windows, **Users/***user_name***/.config/gcloud/application_default_credentials.json**

BigQuery Date and Timestamp Limitations

1. BigQuery maps all Avro date and Avro timestamp-millis/micros data types into one common TIMESTAMP type
2. BigQuery has the following limitations:
   a. the minimum date value is 0001-01-01 00:00:00.000000
   b. the maximum date value is 9999-12-31 23:59:59.999999
3. BigQuery only recognizes dates of the Gregorian calendar, even if the date is less than 1582-10-15. Note that this calendar is named the Proleptic Gregorian calendar.
4. By default, the Avro file contains dates calculated according to the rules of the Julian calendar if the date is less than 1582-10-04; otherwise, the Gregorian calendar is used. Such difference leads to incorrect dates translation while uploading Avro files into BigQuery. To resolve this issue, set action **Column Properties** with specific **/DataTypeMatch** parameters (the following example is for an Oracle source):

Group	Table	Action
FILE	*	FileFormat /Avro
FILE	*	ColumnProperties /DatatypeMatch=date /Datatype="avro date gregorian"
FILE	*	ColumnProperties /DatatypeMatch=timestamp (oracle)[prec < 4] /Datatype="avro timestamp millis gregorian"

FILE	*	ColumnProperties /DatatypeMatch=timestamp (oracle)[prec > 3] /Datatype="avro timestamp micros gregorian"

5. Some dates in the Julian calendar are not present in the Gregorian calendar and vice versa.

   Example:

Julian	Proleptic Gregorian
1500-02-29	Not present, replaced by 1500-03-01
Not present	Range: 1582-10-(05-14)

   If the source has such dates, it could lead to data inconsistency. To resolve this issue, dates must be converted to strings.

Use Case

Use Case 1: BigQuery tables with timekey column (No burst table idiom).

Group	Table	Action
FILE	*	Integrate /ReorderRows=SORT_COALESCE /RenameExpression="{hvr_integ_tstamp}-{hvr_tbl_name}.avro"
FILE	*	FileFormat /Avro
FILE	*	ColumnProperties /Name=hvr_op_val /Extra /IntegrateExpression={hvr_op} /Datatype=integer
FILE	*	ColumnProperties /Name=hvr_integ_tstamp /Extra /IntegrateExpression={hvr_integ_tstamp} /Datatype=datetime
FILE	*	ColumnProperties /Name=hvr_integ_key /Extra /IntegrateExpression={hvr_integ_seq} /Datatype=varchar /Length=36 /Key /TimeKey
FILE	*	AgentPlugIn /Command=hvrbigqueryagent.py /Context=!preserve
FILE	*	AgentPlugIn /Command=hvrbigqueryagent.py /UserArgument="-r" /Context=preserve
FILE	*	Environment /Name=HVR_GBQ_CREDFILE /Value=<path>
FILE	*	Environment /Name=HVR_GBQ_DATASETID /Value=<dataset_id>
FILE	*	Environment /Name=HVR_GBQ_PROJECTID /Value=<proejct_id>

In this use case, during the execution of mode **refr_write_end**,

- If option **-r** is not defined, then HVR appends new data into table.
- If option **-r** is defined, then HVR re-creates table and insert new rows.

Use Case 2: BigQuery tables with soft delete column (using burst table).

Group	Table	Action
FILE	*	Integrate /ReorderRows=SORT_COALESCE /RenameExpression="{hvr_integ_tstamp}-{hvr_tbl_name}.avro"
FILE	*	FileFormat /Avro
FILE	*	ColumnProperties /Name=hvr_is_deleted /Extra /SoftDelete /Datatype=integer

FILE	*	ColumnProperties /Name=hvr_integ_tstamp /Extra /IntegrateExpression={hvr_integ_tstamp} /Datatype=datetime
FILE	*	AgentPlugIn /Command=hvrbigqueryagent.py /UserArgument="-s hvr_is_deleted" /Context=!preserve
FILE	*	AgentPlugIn /Command=hvrbigqueryagent.py /UserArgument="-r -s hvr_is_deleted" /Context=preserve
FILE	*	Environment /Name=HVR_GBQ_CREDFILE /Value=<path>
FILE	*	Environment /Name=HVR_GBQ_DATASETID /Value=<dataset_id>
FILE	*	Environment /Name=HVR_GBQ_PROJECTID /Value=<proejct_id>

In this use case, during the execution of mode **refr_write_end**, burst table is not used. Data is uploaded directly into base table,

- If option **-r** is not defined, then HVR appends new data into table.
- If option **-r** is defined, then HVR recreates table and insert new rows.

During the execution of mode **integ_end**, HVR

- Updates all rows in base table if rows with corresponding keys are present in temporal burst table.
- Inserts all rows into base table from burst table if they are missed in base table.
- Drops burst table on BigQuery.

Agent Plugin for Cassandra

Contents

Name

hvrcassagent.py

Synopsis

hvrcassagent.py *mode chn loc* [*userargs*]

Description

The agent plugin **Hvrcassagent** enables HVR to replicate data into Cassandra database. This agent plugin should be defined in the HVR channel using action **AgentPlugin**. The behaviour of this agent plugin depends on the *–options* supplied in **/UserArgument** field of **AgentPlugin** screen.

This agent plugin supports only Cassandra data type **text**.

Options

This section describes the parameters that can be used with **Hvrcassagent**:

Parameter	Description
-p	Preserves existing row(s) in target during refresh and appends data into table. Not applicable if table structure has been changed. If this option is not defined, truncates existing data from target, then recreates table and insert new rows.
-s	Converts DELETE in source location as UPDATE in target location. To indicate a delete in source, the extra column **hvr_is_deleted** available only in target is updated as "1". For more information, see **ColumnProperties /SoftDelete**.
-t *timecol*	Converts all changes (INSERT, UPDATE, DELETE) in source location as INSERT in target location. For more information, see **ColumnProperties /TimeKey**.

⚠ The column name **hvr_is_deleted** is hardcoded into this plugin, so it is not allowed to change this name.

Environment Variables

The **Environment** variables listed in this section should be defined when using this agent plugin:

Environment Variable Name	Description
$HVR_CASSANDRA_PORT	The port number of the Cassandra server. If this environment variable is not defined, then the default port number **9042** is used.
$HVR_CASSANDRA_HOST	The IP address or hostname of the Cassandra server. It is mandatory to define this environment variable.
$HVR_CASSANDRA_KEYSPACE	The name of Cassandra keyspace. It is mandatory to define this environment variable.
$HVR_CASSANDRA_USER	The username to connect HVR to Cassandra database. The default value is blank (blank password - leave field empty to connect). This environment variable is used only if Cassandra requires authorization.
$HVR_CASSANDRA_PWD	The password of the **$HVR_CASSANDRA_USER** to connect HVR to Cassandra database.

Installing Python Environment

To enable data upload into Cassandra using HVR, perform the following on HVR **Integrate** machine:

1. Install Python 2.7.x +/3.x. Skip this step if the mentioned python version is already installed in the machine.
2. Install the following python client modules:

```
pip install cassandra-driver
pip install six
pip install scales
pip install enum
```

Use Case

Use Case 1: Cassandra tables with plain insert/update/delete.

Group	Table	Action
CASS	*	Integrate /Burst
CASS	*	FileFormat /Csv /QuoteCharacter="
CASS	*	AgentPlugIn /Command=hvrcassagent.py /Context=!preserve_during_refr
CASS	*	AgentPlugIn /Command=hvrcassagent.py /UserArgument="-p" /Context=preserve_during_refr
CASS	*	Environment /Name=HVR_CASSANDRA_HOST /Value=<valid host list comma separated>
CASS	*	Environment /Name=HVR_CASSANDRA_KEYSPACE /Value=<valid keyspace>

In this use case, during the execution of mode **refr_write_begin**,

- If option **-p** is not defined, then HVR drops and recreates each Cassandra table.
- If option **-p** is defined, then HVR appends data into the Cassandra table. If the table does not exist in target, then creates table.

During the execution of mode **refr_write_end** and **integ_end**,

- HVR loads data from CSV file into Cassandra table.

Use Case 2: Cassandra tables with soft delete column.

Group	Table	Action
CASS	*	Integrate /Burst
CASS	*	FileFormat /Csv /QuoteCharacter=''
CASS	*	ColumnProperties /Name=hvr_is_deleted /Extra /SoftDelete
CASS	*	AgentPlugIn /Command=hvrcassagent.py /UserArgument="-s" /Context=!preserve_during_refr
CASS	*	AgentPlugIn /Command=hvrcassagent.py /UserArgument="-p -s" /Context=preserve_during_refr
CASS	*	Environment /Name=HVR_CASSANDRA_HOST /Value=<valid host list comma separated>
CASS	*	Environment /Name=HVR_CASSANDRA_KEYSPACE /Value=<valid keyspace>

In this use case, during the execution of mode **refr_write_begin**,

- If option **-p** is not defined, then HVR drops and recreates each Cassandra table with an extra column **hvr_is_deleted**.
- Else do create-if-not-exists instead.

During the execution of mode **refr_write_end** and **integ_end**,

- HVR loads data from CSV file into Cassandra table.

Use Case 3: Cassandra tables with timekey column.

Group	Table	Action
CASS	*	Integrate /Burst
CASS	*	FileFormat /Csv /QuoteCharacter=''
CASS	*	ColumnProperties /Name=hvr_op_val /Extra /IntegrateExpression={hvr_op} /Datatype=int
CASS	*	ColumnProperties /Name=hvr_integ_key /Extra /IntegrateExpression={hvr_integ_seq} /TimeKey /Key /Datatype=varchar /Length=36
CASS	*	AgentPlugIn /Command=hvrcassagent.py /UserArgument="-t" /Context=!preserve_during_refr
CASS	*	AgentPlugIn /Command=hvrcassagent.py /UserArgument="-t -p" /Context=preserve_during_refr
CASS	*	Environment /Name=HVR_CASSANDRA_HOST /Value=<valid host list comma separated>
CASS	*	Environment /Name=HVR_CASSANDRA_KEYSPACE /Value=<valid keyspace>

In this use case, during the execution of mode **refr_write_begin**,

- If option **-p** is not defined, then HVR drops and recreates each Cassandra table with two extra columns **hvr_op_val**, **hvr_integ_key**.
- Else do create-if-not-exists instead.

During the execution of mode **refr_write_end** and **integ_end**,

- HVR loads data from CSV file into Cassandra table.

Agent Plugin for Manifest Files

Contents

Name

hvrmanifestagent.py

Synopsis

hvrmanifestagent.py *mode chn loc* [*userargs*]

Description

The agent plugin **hvrmanifestagent** writes manifest file for every integrate cycle. A manifest file contains the summary of files or tables that have been changed during an integrate cycle so this information can be used for further downstream processing. This agent plugin should be defined in the HVR channel using action **AgentPlugin**. The behaviour of this agent plugin depends on the agent mode and options supplied in parameter **/UserArgument**. The possible values for **/UserArgument** field in **AgentPlugin** screen are described in section Options.

Agent Modes

Hvrmanifestagent supports only **integ_end** and **refr_write_end** mode. This agent plugin should be executed using action **AgentPlugin** during either Integrate or Refresh.

Parameter	Description
integ_end	Write manifest file implied by option **-m** *mani_fexpr*. Existing manifest files are not deleted by this. Value in manifest file for **initial_load** is **false**
refr_write_end	Write manifest file implied by option **-m** *mani_fexpr*. Existing manifest files are not deleted by this. Value in manifest file for **initial_load** is **true**

Options

This section describes the parameters that can be used with **Hvrmanifestagent**:

Parameter	Description
–i*integ_fexpr*	Integrate file rename expression. This is optional if there is only one table in an integrate cycle. If multiple tables are in a cycle this option is mandatory. It is used to correlate integrated files with corresponding table manifest files. Must be same as **Integrate /RenameExpression** parameter. Sub-directories are allowed. Example: **{hvr_tbl_name}_{hvr_integ_tstamp}.csv**

-m*mani_fexpr*	Manifest file rename expression. This option is mandatory. Sub-directories are allowed. Example: **manifest-{hvr_tbl_name}-{hvr_integ_tstamp}.json**. It is recommended that table name is followed by a character that is not present in table name, such as: **-m {hvr_tbl_name}-{hvr_integ_tstamp}.json** or **-m {hvr_tbl_name}/{hvr_integ_tstamp}.json** or **-m manifests/{hvr_tbl_name}/{hvr_integ_tstamp}.json**
-s*statedir*	Use *statedir* for state files and manifest files instead of **$HVR_LOC_STATEDIR**. This option is mandatory when **$HVR_LOC_STATEDIR** points to a non-native file system (e.g. S3).
-v=*val*	Set **JSON** path *a.b.c* to string value *val* inside new manifest files. This option can be specified multiple times. Example: **-v cap_locs.cen.dbname=mydb**

Example Actions

Group	Table	Action
SRC	*	Capture
TGT	*	Integrate /RenameExpression="{hvr_tbl_name}/{hvr_integ_tstamp}.xml"
TGT	*	AgentPlugin /Command="hvrmanifestagent.py" /UserArgument="-m {hvr_integ_tstamp}-{hvr_tbl_name}.m -s /hvr/hvr_config/files/work/manifests -i {hvr_tbl_name}/{hvr_integ_tstamp}.xml "

Example Manifest File

When using the above example, the manifest files are located in **/hvr/hvr_config/files/work/manifests** and are formatted as *{hvr_integ_tstamp}-{hvr_tbl_name}.m*. E.g. when source tables **aggr_order** and **aggr_product** are integrated in a cycle that ended at August 31st, 10:47:32, the manifest file names are **20170831104732-aggr_order.m** and **20170831104732-aggr_product.m**.

Example manifest file for table aggr_product

```
{
 "cap_rewind": "2017-08-31T08:36:12Z",
 "channel": "db2file",
 "cycle_begin": "2017-08-31T08:47:31Z",
 "cycle_end": "2017-08-31T08:47:32Z",
 "initial_load": false,
 "integ_files": [
     "aggr_product/20170831084731367.xml",
     "aggr_product/20170831084731369.xml",
     "aggr_product/20170831084731370.xml",
     "aggr_product/20170831084731372.xml",
     "aggr_product/20170831084731374.xml",
     "aggr_product/20170831084731376.xml"
 ],

 "integ_files_properties": {
     "aggr_product/20170831084731367.xml": {
         "hvr_tx_seq_min": "0000403227260001",
         "num_rows": 4
 },

 "aggr_product/20170831084731369.xml": {
         "hvr_tx_seq_min": "0000403227480001",
         "num_rows": 72
 },

 "aggr_product/20170831084731370.xml": {
         "hvr_tx_seq_min": "00004032280B0001",
         "num_rows": 60
 },

 "aggr_product/20170831084731372.xml": {
         "hvr_tx_seq_min": "0000403228B70001",
         "num_rows": 60
 },

 "aggr_product/20170831084731374.xml": {
         "hvr_tx_seq_min": "0000403229570001",
         "num_rows": 56
 },

 "aggr_product/20170831084731376.xml": {
           "hvr_tx_seq_min": "0000403229F50001",
           "num_rows": 56
 },

 "integ_loc": {
           "dir": "s3s://rs-bulk-load/",
           "name": "s3",
           "state_dir": "s3s://rs-bulk-load//_hvr_state"
 },

 "next": null,
 "prev": "20170831104732-aggr_order.m",
 "tables": {
    "aggr_product": {
         "basename": "aggr_product",
         "cap_tstamp": "2017-08-31T08:45:31Z",
         "num_rows": 308
       }
   }
}
```

Agent Plugin for MongoDB

Contents

Name

hvrmongodbagent.py

Synopsis

hvrmongodbagent.py *mode chn loc* [*userargs*]

Description

The agent plugin **Hvrmongodbagent** enables HVR to replicate data into MongoDB. This agent plugin should be defined in the HVR channel using action **AgentPlugin**. The behavior of this agent plugin depends on the *–options* supplied in **/UserArgument** field of **AgentPlugin** screen.

This agent plugin supports replication of data in JSON format only and it is mandatory to define action **1=FileFormat /JsonMode=ROW_FRAGMENTS**.

Options

This section describes the parameters that can be used with **Hvrmongodbagent**:

Parameter	Description
-r	Truncates existing data from target and then recreates table and insert new rows. If this option is not defined, appends data into table.
-s *col_name*	Soft deletes the column *col_name*.

Environment Variables

The **Environment** variables listed in this section should be defined when using this agent plugin:

Environment Variable Name	Description
$HVR_MONGODB_DATABASE	The name of the database on MongoDB server.
$HVR_MONGODB_HOST	The IP address or hostname of the MongoDB server.
$HVR_MONGODB_PORT	The port number of the MongoDB server. If this environment variable is not defined, then the default port number **27017** is used.

$MONGODB_COLLECTION	Support for the special substitutions - **hvr_tbl_name**, **hvr_base_name** and **hvr_schema**. **Example:** Source database contains a table **TEST1**. In HVR catalog this table has following names: **TEST1** and **TEST1_BASE**. Destination schema **REMOTE_USER** (defined using **Environment** variable **$HVR_SCHEMA**). So, if **$HVR_MONGODB_COLLECTION** is defined as **{hvr_schema}.{hvr_base_name}_{hvr_tbl_name}_tag**, it will be encoded as **REMOTE_USER.TEST1_BASE_TEST1_tag**.

Installing Python Environment and MongoDB Client

MongoDB client is required for uploading data into MongoDB from local source and convert it into MongoDB collections. To enable data upload into MongoDB using HVR, perform the following on HVR **Integrate** machine:

1. Install Python 2.7.x +/3.x. Skip this step if the mentioned python version is already installed in the machine.
2. Install the following python client modules:

```
pip install pymongo (version > 3.0)
pip install enum
```

Use Case

- **Use Case 1:** MongoDB collections with timekey column.

Group	Table	Action
FILE	*	Integrate /ReorderRows=SORT_COALESCE /RenameExpression="{hvr_integ_tstamp}-{hvr_tbl_name}.json"
FILE	*	FileFormat /Json /JsonMode=ROW_FRAGEMENTS
FILE	*	ColumnProperties /Name=hvr_op_val /Extra /IntegrateExpression={hvr_op} /Datatype=integer
FILE	*	ColumnProperties /Name=hvr_integ_seq /Extra /IntegrateExpression={hvr_integ_seq} /Datatype=varchar /Length=24 /Key /TimeKey
FILE	*	AgentPlugIn /Command=hvrmongodbagent.py /Context=!preserve
FILE	*	AgentPlugIn /Command=hvrmongodbagent.py /UserArgument="-r" /Context=preserve
FILE	*	Environment /Name=HVR_MONGODB_HOST /Value=<host>
FILE	*	Environment /Name=HVR_MONGODB_PORT /Value=<port>
FILE	*	Environment /Name=HVR_MONGODB_COLLECTION /Value={hvr_tbl_name}
FILE	*	Environment /Name=HVR_MONGODB_DATABASE /Value=<database>

In this use case, during the execution of mode **refr_write_end**,

- If option **-r** is not defined, then HVR appends new row into MongoDB Collection.
- If option **-r** is defined, then HVR re-creates MongoDB Collection and inserts new rows.

Tables are mapped to MongoDB collection. Each collection contains documents and each document is mapped to one row from file.

- **Use Case 2:** MongoDB collections with timekey column and static collection name.

Group	Table	Action
FILE	*	Integrate /ReorderRows=SORT_COALESCE /RenameExpression="{hvr_integ_tstamp}-{hvr_tbl_name}.json"
FILE	*	FileFormat /Json /JsonMode=ROW_FRAGEMENTS
FILE	*	ColumnProperties /Name=hvr_op_val /Extra /IntegrateExpression={hvr_op} /Datatype=integer
FILE	*	ColumnProperties /Name=hvr_integ_seq /Extra /IntegrateExpression={hvr_integ_seq} /Datatype=varchar /Length=24 /Key /TimeKey
FILE	*	ColumnProperties /Name=table_name /Extra /IntegrateExpression={hvr_tbl_name} /Datatype=varchar /Length=1000
FILE	*	AgentPlugIn /Command=hvrmongodbagent.py /Context=!preserve
FILE	*	AgentPlugIn /Command=hvrmongodbagent.py /UserArgument="-r" /Context=preserve
FILE	*	Environment /Name=HVR_MONGODB_HOST /Value=<host>
FILE	*	Environment /Name=HVR_MONGODB_PORT /Value=<port>
FILE	*	Environment /Name=HVR_MONGODB_COLLECTION /Value=collection_name
FILE	*	Environment /Name=HVR_MONGODB_DATABASE /Value=<database>

- **Use Case 3:** MongoDB collection with softdelete column and dynamic collection name.

Group	Table	Action
FILE	*	Integrate /ReorderRows=SORT_COALESCE /RenameExpression="{hvr_integ_tstamp}-{hvr_tbl_name}.json"
FILE	*	FileFormat /Json /JsonMode=ROW_FRAGEMENTS
FILE	*	ColumnProperties /Name=hvr_is_deleted /Extra /SoftDelete /Datatype=integer
FILE	*	ColumnProperties /Name=hvr_integ_tstamp /Extra /IntegrateExpression={hvr_integ_tstamp} /Datatype=timestamp
FILE	*	AgentPlugIn /Command=hvrmongodbagent.py /UserArgument="-s hvr_is_deleted" /Context=!preserve
FILE	*	AgentPlugIn /Command=hvrmongodbagent.py /UserArgument="-r -s hvr_is_deleted" /Context=preserve
FILE	*	Environment /Name=HVR_MONGODB_HOST /Value=<host>
FILE	*	Environment /Name=HVR_MONGODB_PORT /Value=<port>
FILE	*	Environment /Name=HVR_MONGODB_COLLECTION /Value={hvr_tbl_name}
FILE	*	Environment /Name=HVR_MONGODB_DATABASE /Value=<database>

In this use case, during the execution of mode **refr_write_end**,

- If option **-r** is not defined, then HVR appends new row into MongoDB Collection.
- If option **-r** is defined, then HVR re-creates MongoDB Collection and inserts new rows.

_id is a special name for the unique document identifier. The extra column **_id** is built based on key columns in table.
All values are converted to string like **{"c1": 100, "c2": "string", "c3": value, "hvr_is_deleted": 1}** where **c1** and **c2** are key columns. So **_id** will look like **{"_id": "100string"}**.

- **Use Case 4:** MongoDB collection with softdelete column and static collection name.

  In case of using static collection names for all tables in channel, a new synthetic key column should be added.

Group	Table	Action
FILE	*	Integrate /ReorderRows=SORT_COALESCE /RenameExpression="{hvr_integ_tstamp}-{hvr_tbl_name}.json"
FILE	*	FileFormat /Json /JsonMode=ROW_FRAGEMENTS
FILE	*	ColumnProperties /Name=hvr_is_deleted /Extra /SoftDelete /Datatype=integer
FILE	*	ColumnProperties /Name=hvr_integ_tstamp /Extra /IntegrateExpression={hvr_integ_tstamp} /Datatype=timestamp
FILE	*	ColumnProperties /Name=table_name /Extra /IntegrateExpression={hvr_tbl_name} /Key /Datatype=varchar /Length=1000
FILE	*	AgentPlugIn /Command=hvrmongodbagent.py /UserArgument="-s hvr_is_deleted" /Context=!preserve
FILE	*	AgentPlugIn /Command=hvrmongodbagent.py /UserArgument="-r -s hvr_is_deleted" /Context=preserve
FILE	*	Environment /Name=HVR_MONGODB_HOST /Value=<host>
FILE	*	Environment /Name=HVR_MONGODB_PORT /Value=<port>
FILE	*	Environment /Name=HVR_MONGODB_COLLECTION /Value=collection_name
FILE	*	Environment /Name=HVR_MONGODB_DATABASE /Value=<database>

Capture

Contents

Description

Action **Capture** instructs HVR to capture changes from a location. Various parameters are available to modify the functionality and performance of capture.

For a database location, HVR gives you an option to capture changes using the log-based method (**/LogReadMethod)** or trigger-based method (**/TriggerBased**). HVR recommends using the log-based data capture because it has less impact on database resources as it reads data directly from its logs, without affecting transactions, manages large volumes of data and supports more data operations, such as truncates, as well as DDL capture. In contrast, the trigger-based data capture creates triggers on tables that require change data capture, so firing the triggers and storing row changes in a shadow table slow down transactions and introduces overhead.

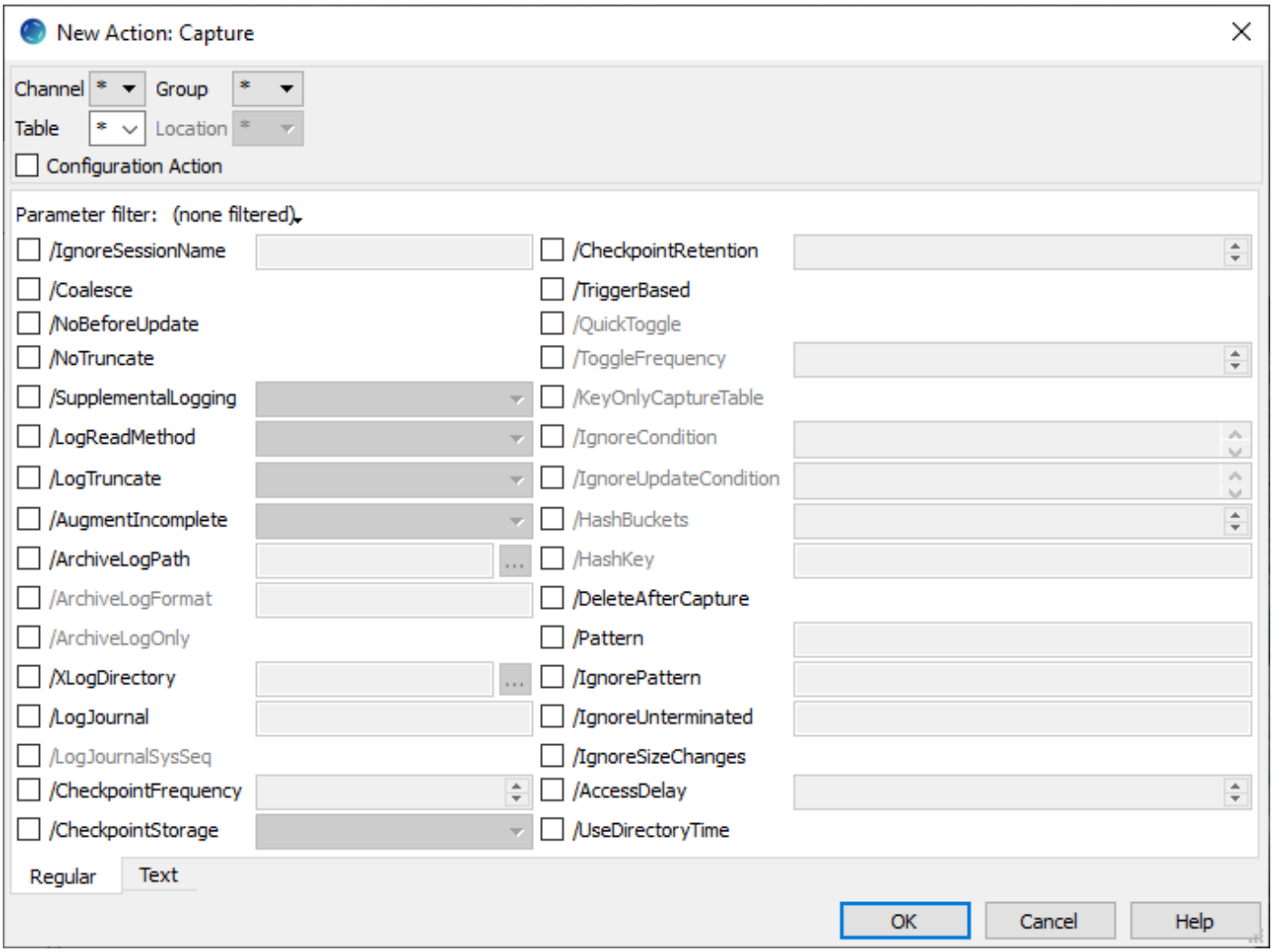
When defined on a file location this action instructs HVR to capture files from a file location's directory. Changes from a file location can be replicated both to a database location and to a file location if the channel contains table information. In this case any files captured are parsed (see action **FileFormat**).

If **Capture** is defined on a file location without table information then each file captured is treated as a 'blob' and is replicated to the integrate file locations without HVR recognizing its format. If such a 'blob' file channel is defined with only actions **Capture** and **Integrate** (no parameters) then all files in the capture location's directory (including files in sub-directories) are replicated to the integrate location's directory. The original files are not touched or deleted, and in the target directory the original file names and sub-directories are preserved. New and changed files are replicated, but empty sub-directories and file deletions are not replicated.

Bidirectional replication (replication in both directions with changes happening in both file locations) is not currently supported for file locations. File deletion is not currently captured by HVR.
If **Capture** is defined on a file location without parameter **/DeleteAfterCapture** and action **LocationProperties /StateDirectory** is used to define a state directory outside of the file location's top directory, then HVR's file capture becomes read only; write permissions are not needed.

Parameters

This section describes the parameters available for action **Capture**. By default, only the supported parameters for the selected location class are displayed in the **Capture** window.



Parameter	Argument	Description
/IgnoreSessionName	*sess_name*	This action instructs the capture job to ignore changes performed by the specified session name. Multiple ignore session names can be defined for a job, either by defining **/IgnoreSessionName** multiple times or by specifying a comma separated list of names as its value. Normally HVR's capture avoids recapturing changes made during HVR integration by ignoring any changes made by sessions named **hvr_integrate**. This prevents looping during bidirectional replication but means that different channels ignore each other's changes. The session name actually used by integration can be changed using **Integrate /SessionName**. For more information, see Managing Recapturing Using Session Names. If this parameter is defined for any table with log based capture, then it affects all tables captured from that location. For an example of using action **Capture** with parameter **/IgnoreSessionName**, refer to section Examples below.
/Coalesce		Causes coalescing of multiple operations on the same row into a single operation. For example, an **INSERT** and an **UPDATE** can be replaced by a single **INSERT**; five **UPDATE**s can be replaced by one **UPDATE**, or an **INSERT** and a **DELETE** of a row can be filtered out altogether. The disadvantage of not replicating these intermediate values is that some consistency constraints may be violated on the target database. This parameter should not be used together with **Transform /SapXForm**.

/NoBeforeUpdate		Do not capture 'before row' for an update. By default when an update happens HVR will capture both the 'before' and 'after' version of the row. This lets integration only update columns which have been changed and also allows collision detection to check the target row has not been changed unexpectedly. Defining this parameter can improve performance, because less data is transported. But that means that integrate will update all columns (normally HVR will only update the columns that were actually changed by the update statements and will leave the other columns unchanged). If this parameter is defined for any table with log based capture, then it affects all tables captured from that location.
/NoTruncate		Do not capture SQL truncate table statements such as **TRUNCATE** in Oracle and **modify** *mytbl* **to truncated** in Ingres. If this parameter is not defined, then these operations are replicated using **hvr_op** value **5**. For DB2 for z/OS, this parameter affects only **TRUNCATE IMMEDIATE**. HVR will always capture **TRUNCATE** if used without **IMMEDIATE** option (this will be replicated using **hvr_op** value **0**). This parameter is not supported for Microsoft SQL Server.
/SupplementalLogging **SQL Server**	*method*	Specify what action should be performed to enable supplemental logging for tables. Supplemental logging should be enabled to make log-based capture of updates possible. Valid values for *method* are: • **CDCTAB** (**default** when source SQL Server supports CDC tables - SQL Server 2008 or higher (Enterprise edition) and SQL Server 2016 SP1 or higher (Enterprise and Standard editions)): Enable supplemental logging of updates by creating a Change Data Capture (CDC) instance for the source table. If users attempt some DDL such as **TRUNCATE TABLE** when a CDC instance exists for the table they will give an error message. If **/LogTruncate=NATIVE_DBMS_AGENT** is defined creating a CDC instance may causes I/O overhead (SQL Server jobs copy each change to a CDC table, which no-one uses). This method is needed to capture updates to tables without a primary key. It is not available for SQL Server Standard and Express editions or for SQL Server 2005. This means that HVR on such databases can only capture changes to tables with primary keys. • **ARTICLE_OR_CDCTAB** (**default** when source SQL Server does not support CDC tables): Enable supplemental logging of updates by creating an SQL Server transactional replication article if the source table has Primary Key, or by creating a CDC table instance if the source table does not have Primary Key. If users attempt some DDL such as **DROP TABLE** or **TRUNCATE TABLE** when an article exists for the table they will give an error message. • **EXISTING**: Check that an article or a CDC table instance already exists for the source table. No new articles or CDC table instances are created. If an article or a CDC table instance does not exists the **Hvrinit** will give an error message. • **EXISTING_OR_CDCTAB**: Check if an article or a CDC table instance already exists for the source table. Enable supplemental logging of updates by creating a Change Data Capture (CDC) instance for the source table if no article or a CDC table instance exists.

/LogReadMethod	*method*	Select method of reading changes from the DBMS log file. This parameter is supported only for certain location classes. For the list of supported location class, see Log-based capture with **/LogReadMethod** parameter in Capabilities. Valid values for *method* are: • **DIRECT** (**default**): Read transaction log records directly from the DBMS log file using file I/O. This method is generally faster and more efficient than the SQL mode. The **DIRECT** log read method requires that HVR agent is installed on the source database machine. This method is supported only for certain location classes. For the list of supported location classes, see Direct access to logs on a file system in Capabilities. • **SQL**: Query transaction log records using a special SQL function. The advantage of this method is that it reads change data over an SQL connection and does not require an HVR agent to be installed on the source database machine. The disadvantages of the **SQL** method is that it is slower than the **DIRECT** method and exposes additional load on the source database. This method is supported only for certain location classes. For the list of supported location classes, see Access to logs using SQL interface in Capabilities. For MySQL, the **default** method of reading changes from the DBMS log file is **SQL**. For Oracle, the **SQL** method enables capture from LogMiner. For PostgreSQL, prior to HVR version 5.5, the **SQL** method does not support bidirectional replication because changes will be re-captured and replicated back. For SQL Server, the **DIRECT** method requires Windows Administrator privileges and reduced permission models are not supported. The **SQL** m ethod supports reduced permission models but it may require incomplete row augmenting.

/LogTruncate **SQL Server**	*method*	Specify who advances SQL Server transaction log truncation point (truncates the log). Valid values for *method* are; • **CAP_JOB** (**default**): is used to indicate that the capture job regularly calls **sp_repldone** to unconditionally release the hold of the truncation point for replication. When this option is selected and **HVR Initialize** is run, depending on the value of **/SupplementalLogging**, HVR will also drop/disable the SQL Server Agent jobs that are created when CDC tables are created through the CDC stored procedures, and the Agent jobs related to data distribution. As a result, the additional impact of auxiliary database objects to enable supplemental logging is minimized. For example, the CDC tables are not populated (and the space allocation increased) because the Agent job to do this is dropped. Multiple capture jobs can be set up on a single database with option **CAP_JOB** selected. However, note that if no capture job is running with the CDC tables and/or Articles in place, the TLog will grow because the truncation point for replication is not released. Do not set this option if there is another data replication solution or the database uses CDC tables. • **CAP_JOB_RETAIN**: This value must be used when capturing from a SQL Server database with the recovery mode set to Simple Recovery. The capture job moves the truncation point of the transacti on log forward by calling the **sp_repldone** stored procedure at the end of each sub-cycle. Only part of the transaction log that has already been processed (captured) is marked for truncation (this is different from the **CAP_JOB** mode, where all records in the transaction log are marked for truncation, including those that have not been captured yet). This value is not compatible with multi-capture and does not allow for coexistence with a third party replication solution. This setting will also result in SQL Server's Agent jobs being dropped/disabled, so the TLog will grow when the capture job is not running and CDC tables and/or Articles are still in place. Do not set this option if another data replication solution is in place or CDC tables are used in the database. • **LOGRELEASE_TASK**: should be set if a separate job/task is created to release the truncation point for replication. For example, schedule a separate SQL Server Agent job to unconditionally call **sp_repldone** at an interval. HVR also provides its own **Log Release** capability that can be accessed in the GUI through the pop-up menu on the SQL Server location. Choosing the option **LOGRELEASE_TASK** will also result in SQL Server's Agent jobs being dropped /disabled. However, as long as the scheduled log release task runs, the truncation point for replication is released, even if the capture job (s) is(are) not running. This option should only be used in conjunction with another replication or CDC solution if the log release task that is scheduled is aware of the requirements of the other solution. • **NATIVE_DBMS_AGENT**: should be used if native replication and/or CDC tables are used on the database. With this option, HVR will not drop/disable the native SQL Server Agent jobs that are created when CDC tables and/or Articles are created. HVR will also not interfere with the release of the truncation point for REPLICATION. If **/SupplementalLogging=ARTICLE_OR_CDCTAB** is defined this may causes I/O overhead (SQL Server jobs copy each change to a CDC table, which no-one uses).

/AugmentIncomplete	*col_type*	During **capture**, HVR may receive partial/incomplete values for certain column types. Partial/incomplete values are the values that HVR cannot **capture** entirely due to technical limitations in the database interface. This parameter instructs HVR to perform additional steps to retrieve the full value from the source database, this is called augmenting. This parameter also augments the missing values for key updates. ⚠ Defining this parameter can adversely affect the **capture** performance. If this parameter is not defined, and when a partial/incomplete value is received, the **capture** will fail with an error. Valid values for *col_type* are: • **NONE** (**default**): No extra augmenting is done. • **LOB**: Capture will augment partial/incomplete values for all columns of a table, if that table contains at least one lob column. For key-updates, missing values are augmented too. • **ALL**: Capture will augment partial/incomplete values for all columns of any table. For key-updates, missing values are augmented too. In certain situations, the default behavior changes and setting **/AugmentIncomplete** can only override the behavior with a 'stronger' value. • For DB2 for Linux Unix and Windows, **LOB** should be selected to capture columns with **xml** data type. • For DB2 for z/OS, the **default** *col_type* is **LOB** and can only be changed to **ALL**. • For SQL Server, capture when **/LogReadMethod** is set to **SQL** and tables that contain non-key columns, the **default** *col_type* is **ALL** and can not be changed. • For Oracle, capture when **/LogReadMethod** is set to **SQL** the **default** *col_type* is **LOB** and can only be changed to **ALL**.
/ArchiveLogPath	*dir*	Instruct HVR to search for the transaction log archives in the given directory. For Oracle, HVR will search for the log archives in the directory *dir* in addition to the 'primary' Oracle archive directory. If **/ArchiveLogOnly** parameter is enabled then HVR will search for the log archives in the directory *dir* only. Any process could be copying log archive files to this directory; the Oracle archiver (if another **LOG_ARCHIVE_DEST**_*N* is defined), RMAN, **Hvrlogrelease** or a simple shell script. Whoever sets up copying of these files must also arrange that they are purged periodically, otherwise the directory will fill up. For SQL Server, HVR normally locates the transaction log backup files by querying the backup history table in the msdb database. Specifying this parameter tells HVR to search for the log backup files in the *dir* folder instead. When this parameter is defined, the **/ArchiveLogFormat** parameter must also be defined. For HANA, HVR will search for the log backups in the directory *dir* instead of the default log backup location for the source database.

/ArchiveLogFormat	*format*	Describes the filename format (template) of the transaction log archive files stored in the directory specified by the **/ArchiveLogPath** parameter. The list of supported format variables and the default *format* string are database-specific. For Oracle, this parameter accepts the following format variables: • **%d** - match numbers (zero or more decimal digits). Numbers matched using this variable are ignored by HVR. • **%r** or **%R** - resetlogs ID • **%s** or **%S** - log sequence number • **%t** or **%T** - thread number • **%z** or **%Z** - match alphanumeric characters. Characters matched using this variable are ignored by HVR. • Wildcard character ***** is not supported. For Oracle, when this parameter is not defined, then by **default** HVR will query the database for Oracle's initialization parameter - **LOG_ARCHIVE_FORMAT**. For SQL Server, this parameter accepts the following format variables: • **%d** - database name • **%Y** - year (up to 4 digit decimal integer) • **%M** - month (up to 2 digit decimal integer) • **%D** - day (up to 2 digit decimal integer) • **%h** - hours (up to 2 digit decimal integer) • **%m** - minutes (up to 2 digit decimal integer) • **%s** - seconds (up to 2 digit decimal integer) • **%n** - file sequence number (up to 64 bit decimal integer) • **%%** - matches % • ***** - wildcard, matches zero or more characters HVR uses the **%Y**, **%M**, **%D**, **%h**, **%m**, **%s** and **%n** values to sort and processes the log backup files in the correct (chronological) order. The combinations of the **%Y**, **%M**, **%D** and **%h**, **%m**, **%s** values are expected to form valid date and time values, however no validation is performed. Any value that is missing from the *format* string is considered to be 0. When sorting the files comparison is done in the following order: **%Y**, **%M**, **%D**, **%h**, **%m**, **%s**, **%n**. For SQL Server, this parameter has no **default** and must be specified if **/ArchiveLogPath** parameter is defined. For HANA, this parameter accepts the following format variables: • **%v** - log volume ID • **%p** - log partition ID • **%s** - start sequence number • **%e** - end sequence number • **%t** - start timestamp (in milliseconds since UNIX epoch) • **%%** - matches % • ***** - wildcard, matches zero or more characters The **%s**, **%e** and **%t** format variables are mandatory. For HANA, this parameter is optional, the **default** *format* value is **log_backup_%v_%p_%s_%e.%t**.

/ArchiveLogOnly		Capture data from archived redo files in directory defined by **/ArchiveLogPath** only and do not read anything from online redo files or the 'primary' archive destination. This allows the HVR process to reside on a different machine than the Oracle DBMS or SQL Server and read changes from files that are sent to it by some remote file copy mechanism (e.g. **FTP**). The capture job still needs an SQL connection to the database for accessing dictionary tables, but this can be a regular connection. Replication in this mode can have longer delays in comparison with the 'online' mode. For Oracle, to control the delays it is possible to force Oracle to issue an archive once per predefined period of time. For Oracle RAC systems, delays are defined by the slowest or the least busy node. This is because archives from all threads have to be merged by SCNs in order to generate replicated data flow. For SQL Server, it is possible to control this delay by **HVR_MSSQL_ZREAD_ARCHIVE_SCAN_INTERVAL** environment variable.
/XLogDirectory	*dir*	Directory containing current PostgreSQL **xlog** files.
/LogJournal **Db2 for i**	*schema.journal*	Capture from specified DB2 for i journal. Both the schema (library) of the journal and the journal name should be specified (separated by a dot). This parameter is mandatory for DB2 for i. All tables in a channel should use the same journal. Use different channels for tables associated with different journals. If this parameter is defined for any table, then it affects all tables captured from that location.
/LogJournalSysSeq **Db2 for i**		Capture from journal using ***SYSSEQ**. This parameter requires **/LogJournal**.

/CheckpointFrequency **Since** v5.2.3/15	*secs*	Checkpointing frequency in seconds for long running transactions, so the capture job can recover quickly when it restarts. Value *secs* is the interval (in seconds) at which the capture job creates checkpoints. The **default** frequency (when **/CheckpointFrequency** is not defined) is **3 00** seconds (5 minutes). Value **0** means no checkpoints are written. Without checkpoints, capture jobs must rewind back to the start of the oldest open transaction, which can take a long time and may require access to many old DBMS log files (e.g. archive files). The checkpoints are written into directory **$HVR_CONFIG/capckp/***hub /chn*. If a transaction continues to make changes for a long period then successive checkpoints will not rewrite its same changes each time; instead the checkpoint will only write new changes for that transaction; for older changes it will reuse files written by earlier checkpoints. Only long-running transactions are saved in the checkpoint. For example if the checkpoint frequency is each 5 minutes but users always do an SQL commit within 4 minutes then checkpoints will never be written. If however some users keep transactions open for 10 minutes, then those transactions will be saved but shorter-lived ones in the same period will not. The frequency with which capture checkpoints are written is relative to the capture jobs own clock, but it decides whether a transaction has been running long enough to be checkpointed by comparing the timestamps in its DBMS logging records. As consequence, the maximum (worst-case) time that an interrupted capture job would need to recover (rewind back over all its open transactions) is its checkpoint frequency (default 5 minutes) plus the amount of time it takes to reread the amount of changes that the DBMS can write in that period of time. When a capture job is recovering it will only use checkpoints which were written before the 'capture cycle' was completed. This means that very frequent capture checkpointing (say every 10 seconds) is wasteful and will not speed up capture job recovery time. This parameter is supported only for certain location classes. For the list of supported location classes, see Log-based capture checkpointing in Capabilities.
/CheckpointStorage **Since** v5.2.3/15	*STOR*	Storage location of capture checkpoint files for quick capture recovery. Available options for *STOR* are: • **LOCATION** (**default**): Save checkpoints in a directory on capture location. • **HUB**: Save checkpoints in a directory on hub machine. Writing checkpoints on the hub is more expensive because extra data must be sent across the network. Checkpoints should be stored on the hub machine when capturing changes from an Oracle RAC, because the directory on the remote location where capture job would otherwise write checkpoints (**$HVR_CONFIG/capckp/**) may not be shared inside the RAC cluster, so it may not be available when the capture job restarts. Checkpoints are saved in directory **$HVR_CONFIG/capckp/** (this can be either on capture machine or hub). If capture job is restarted but it cannot find the most recent checkpoint files (perhaps the contents of that directory have been lost during a failover) then it will write a warning and then rewind back to the start of the oldest open transaction.

/CheckpointRetention **Since** v5.5.5/6	*period*	Retains capture checkpoint files up to the specified *period* (in seconds). The retained checkpoint files are saved in **$HVR_CONFIG/capckpretain/** *hub**channel**location* (this can be either on capture machine or hub) based on the location defined in **/CheckpointStorage**.
/TriggerBased		Capture changes through DBMS triggers generated by HVR, instead of using log-based capture. This parameter is supported only for certain location class. For the list of supported location class, see Trigger-based capture in Capabilities.
/QuickToggle		Allows end user transactions to avoid lock on toggle table. The toggle table is changed by HVR during trigger based capture. Normally all changes from user transactions before a toggle is put into one set of capture tables and changes from after a toggle are put in the other set. This ensures that transactions are not split. If an end user transaction is running when HVR changes the toggle then HVR must wait, and if other end user transactions start then they must wait behind HVR. Parameter **/QuickToggle** allows these other transactions to avoid waiting, but the consequence is that their changes can be split across both sets of capture tables. During integration these changes will be applied in separate transactions; in between these transactions the target database is not consistent. If this parameter is defined for any table, then it affects all tables captured from that location. This parameter requires **/TriggerBased**. For Ingres, variable **ING_SET** must be defined to force **readlock**=**nolock** on the quick toggle table. **Example**: `$ ingsetenv ING_SET 'set lockmode on hvr_qtogmychn where readlock=nolock'`
/ToggleFrequency	*secs*	Instruct HVR's trigger based capture jobs to wait for a fixed interval *secs* (in seconds) before toggling and reselecting capture tables. If this parameter is defined for any table then it affects all tables captured from that location. If this parameter is not selected, the trigger based capture job dynamically waits for a capture trigger to raise a database alert. Raising and waiting for database alerts is an unnecessary overhead if the capture database is very busy. This parameter requires **/TriggerBased**.
/KeyOnlyCaptureTable		Improve performance for capture triggers by only writing the key columns into the capture table. The non key columns are extracted using an outer join from the capture table to the replicated table. Internally HVR uses the same outer join technique to capture changes to long columns (e.g. **long varchar**). This is necessary because DBMS rules/triggers do not support long data types. The disadvantage of this technique is that 'transient' column values can sometimes be replicated, for example if a delete happens just after the toggle has changed, then the outer join could produce a NULL for a column which never had that value. This parameter requires **/TriggerBased**.

/IgnoreCondition	*sql_expr*	Ignore (do not capture) any changes that satisfy expression *sql_expr* (e.g. **Prod_id < 100**). This logic is added to the HVR capture rules/triggers and procedures. This parameter differs from the **Restrict /CaptureCondition** as follows: • The SQL expression is simpler, i.e. it cannot contain subselects. • The sense of the SQL expression is reversed (changes are only replicated if the expression is false). • No 'restrict update conversion'. Restrict update conversion means if an update changes a row which did not satisfy the condition into a row that does satisfy the condition then the update is converted to an insert. This parameter requires **/TriggerBased**.
/IgnoreUpdateCondition	*sql_expr*	Ignore (do not capture) any update changes that satisfy expression *sql_expr*. This logic is added to the HVR capture rules/triggers and procedures. This parameter requires **/TriggerBased**.
/HashBuckets **Ingres**	*int*	Identify the number *int* of hash buckets, with which the capture table is created. This implies that Ingres capture tables have a hash structure. This reduces the chance of locking contention between parallel user sessions writing to the same capture table. It also makes the capture table larger and I/O into it sparser, so it should only be used when such locking contention could occur. Row level locking (default for Oracle and SQL Server and configurable for Ingres) removes this locking contention too without the cost of extra I/O. This parameter requires **/TriggerBased**.
/HashKey **Ingres**	*col_list*	Identify the list of columns *col_list*, the values of which are used to calculate the hash key value. The **default** hash key is the replication key for this table. The key specified does not have to be unique; in some cases concurrency is improved by choosing a non-unique key for hashing. This parameter requires **/TriggerBased**.
/DeleteAfterCapture **File/FTP/Sharepoint**		Delete file after capture, instead of capturing recently changed files. If this parameter is defined, then the channel moves files from the location. Without it, the channel copies files if they are new or modified.
/Pattern **File/FTP/Sharepoint**	*pattern*	Only capture files whose names match pattern. The **default** pattern is **'**/*'** which means search all sub-directories and match all files. Possible patterns are:

- **'*.c'** – Wildcard, for files ending with **.c**. A single asterisk matches all or part of a file name or sub-directory name.
- **'**/*txt'** – Recursive Sub-directory Wildcard, to walk through the directory tree, matching files ending with **txt**. A double asterisk matches zero, one or more sub-directories but never matches a file name or part of a sub-directory name.
- **'*.lis'** Files ending with **.lis** or **.xml**
- **'a?b[d0 9]'** Files with first letter **a**, third letter **b** and fourth letter **d** or a digit. Note that [**a f**] matches characters, which are alphabetically between **a** and **f**. Ranges can be used to escape too; [*****] matches ***** only and [**[**] matches character **[** only.
- '***.csv|*.xml|*.pdf**' Multiple patterns may be specified. In this case, all csv files, all xml files, all pdf files will be captured.
- **{***hvr_tbl_name***}** is only used when data is replicated from structured files to a database with multiple tables. If there are multiple tables in your channel, the capture job needs to determine to which table a file should be replicated and will use the file name for this. In this case, the **Capture/Pattern** must be defined. The **Capture /Pattern** is not required for channels with only 1 table in them.

  Example: In your channel, you have a file **audit.csv** that needs to be replicated to a table called **file_a**, in which column names are the same as in csv file. To do this, the following actions should be defined on the source group **Capture /Pattern={file_a}.csv** and **FileFormat/Csv /HeaderLine.**
- **{***hvr_address***}** When a file is matched with this pattern, it is only replicated to integrate locations specified by the matching part of the file name. Locations can be specified as follows:
  - An integrate location name, such as **tgt1**.
  - A location group name containing integrate locations, such as **TGTGRP**.
  - An alias for an integrate location, defined with **Restrict /AddressSubscribe**, for example, **22** or **Alias7**.
  - A list of the above, separated by a semicolon, colon or comma, such as **src**, **tgt1**.
- **{***name***}** is only used for replicating files between directories ("blob file" or "flat file"). An example of the **{***name***}** pattern is **{abc}.txt**. The value inside the braces is an identifier. Although the 'named pattern' works the same as a wildcard (*****), but it also associates the captured file with a property named **{abc}**. This property can be used in the 'named substitution' (see **Integrate /RenameExpression**).

  Example 1: suppose a channel has capture pattern **{office}.txt** and rename expression **xx_{office}.data**. If file **paris.txt** is matched, then property **{office}** is assigned string value **paris**. This means it is renamed to **xx_paris.data**.

  Example 2: suppose the **77-99.pdf** file on source needs to be renamed to **new-77-suff-99.pdf2** on target. In this case, the **Capture /Pattern** is **{a}-{b}.pdf**, and the **Integrate /RenameExpression** must be defined as **new-{a}-suff-{b}-pdf2**.

On Unix and Linux, file name matching is case sensitive (e.g. ***.lis** does not match file **FOO.LIS**), but on Windows and SharePoint it is case-insensitive. For FTP and SFTP the case sensitivity depends on the OS on which HVR is running, not the OS of the FTP/SFTP server.

/IgnorePattern File/FTP/Sharepoint	*pattern*	Ignore files whose names match *pattern*. For example, to ignore all files underneath sub-directory **qqq** specify ignore pattern **qqq/**/***. The rules and valid forms for **/IgnorePattern** are the same as for **/Pattern**, except that 'named patterns' are not allowed.
/IgnoreUnterminated File/FTP/Sharepoint	*pattern*	Ignore files whose last line does not match *pattern*. This ensures that incomplete files are not captured. This pattern matching is supported for UTF 8 files but not for UTF 16 file encoding.
/IgnoreSizeChanges File/FTP/Sharepoint		Changes in file size during capture is not considered an error when capturing from a file location.
/AccessDelay File/FTP/Sharepoint	*secs*	Delay reading file for *secs* seconds to ensure that writing is complete. HVR will ignore this file until its last create or modify timestamp is more than *secs* seconds old.
/UseDirectoryTime File/FTP/Sharepoint		When checking the timestamp of a file, check the modify timestamp of the parent directory (and its parent directories), as well as the file's own modify timestamp. This can be necessary on Windows when **/DeleteAfterCapture** is not defined to detect if a new file has been added by someone moving it into the file location's directory; on Windows file systems moving a file does not change its timestamp. It can also be necessary on Unix/Windows if a sub-directory containing files is moved into the file location directory. The disadvantage of this parameter is that when one file is moved into a directory, then all of the files in that directory will be captured again. This parameter cannot be defined with **/DeleteAfterCapture** (it is not necessary).

Writing Files while HVR is Capturing Files

It is often better to avoid having HVR capture from files while they are still be written. One reason is to prevent HVR replicating an incomplete version of the file to the integrate machine. Another problem is that if **/DeleteAfterCapture** is defined, then HVR will attempt to delete the file before it is even finished.

Capture of incomplete files can be avoided by defining **/AccessDelay** or **/IgnoreUnterminated**.

Another technique is to first write the data into a filename that HVR capture will not match (outside the file location directory or into a file matched with **/IgnorePattern**) and then move it when it is ready to a filename that HVR will match. On Windows this last technique only works if **/DeleteAfterCapture** is defined, because the file modify timestamp (that HVR capture would otherwise rely on) is not changed by a file move operation.

A group of files can be revealed to HVR capture together by first writing them in sub-directory and then moving the whole sub-directory into the file location's top directory together.

- If column **hvr_op** is not defined, then it default to **1** (insert). Value **0** means delete, and value **2** means update.
- Binary values can be given with the **format** attribute (see example above).
- If the **name** attribute is not supplied for the **<column>** tag, then HVR assumes that the order of the **<column>** tags inside the **<row>** matches the order in the HVR catalogs (column **col_sequence** of table **hvr_column**).

Examples

This section includes an example of using the **/IgnoreSessionName** parameter.

Using /IgnoreSessionName

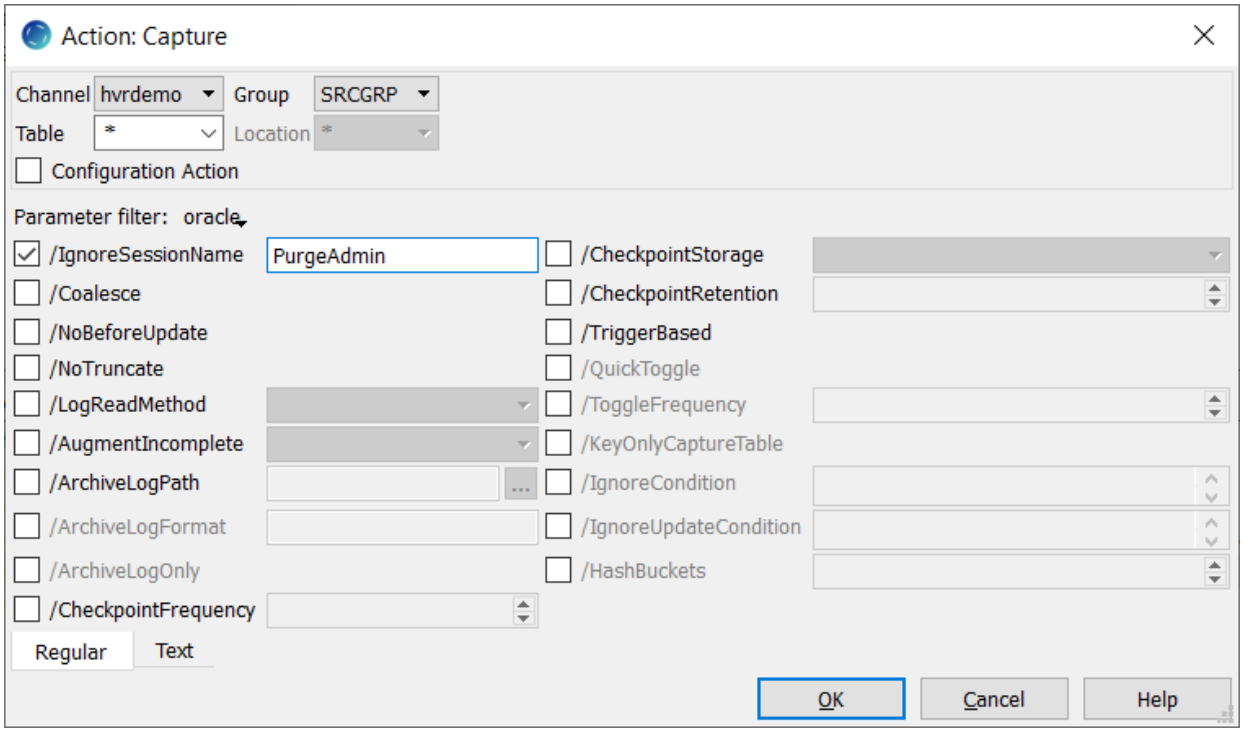
HVR allows to run a purge process on an Oracle source location without stopping active replication. Purging is deleting o bsolete data from a database. To ensure that the deleted data does not replicate to a target location, the purge process must be started by a database user (e.g. **PurgeAdmin**) other than the user (e.g. **hvruser**) under which the replication process is running, and HVR must be configured to ignore the session name of the **PurgeAdmin**.

The steps for implementing this scenario are as follows:

1. In a source database, create a new user **PurgeAdmin** that will run a purge script against this database.
2. Grant the applicable permissions to user **PurgeAdmin**, e.g. a privilege to delete rows in another schema:

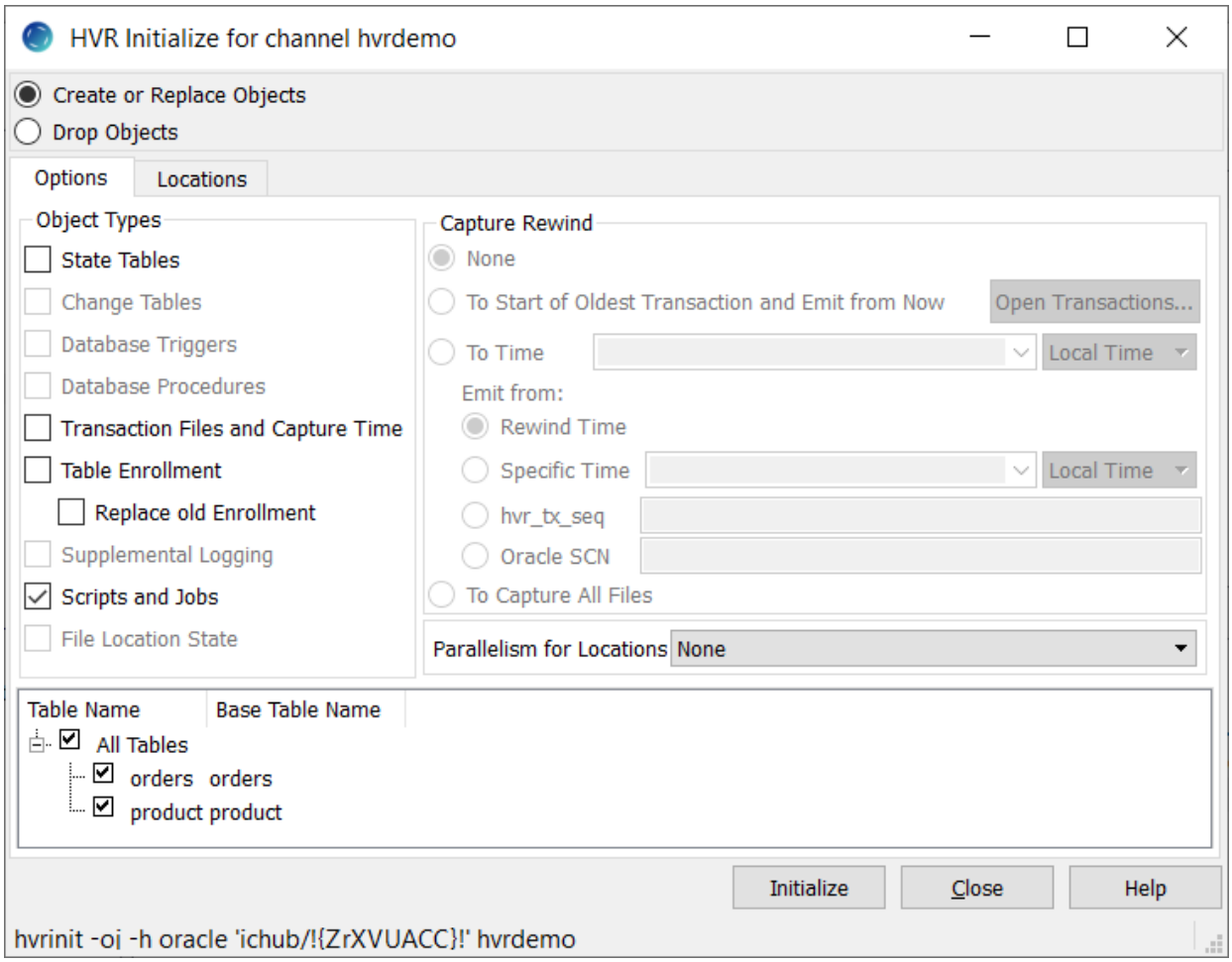
```
grant delete any table to PurgeAdmin;
```

3. In the HVR GUI, update action **Capture** defined on the existing channel by adding parameter **/IgnoreSessionName**:
   a. Under the **Actions** pane, double-click a row with action **Capture**.
   b. In the **Action: Capture** dialog, select parameter **/IgnoreSessionName** and specify the user name ' PurgeAdmin'.
   c. Click **OK**.



4. Re-Initialize the capture job:

   a. In the navigation tree pane, right-click channel (e.g. **chn**) and click **HVR Initialize**.
   b. In the **Options** pane, select **Script and Jobs** and click **Initialize**. Running **HVR Initialize** with option **Scripts and Jobs** (option **-oj**) will suspend and restart the affected jobs automatically.



CollisionDetect

Contents

Description

Action **CollisionDetect** allows HVR to detect collisions during data replication. Collisions can happen during bi-directional replication. For example, if the same row is changed to different values in databases A and B so quickly that there is no time to replicate the changes to the other database, then there is a risk that the change to A will be applied to B while the change to B is on its way to A. Collisions can occur in cases other than bi-directional replication. For example, if changes are made first to A and then to B, then the change to B can reach database C before the change from A reaches C. Undetected collisions can lead to inconsistencies between the replicated databases.

The default behavior for **CollisionDetect** is automatic resolution using a simple rule: the most recent change is kept and the older changes are discarded. The timestamps used have a one-second granularity; if changes occur in the same second, then one arbitrary location (the one whose name is sorted first) will 'win'. Parameters are available to change this automatic resolution rule and to tune performance.

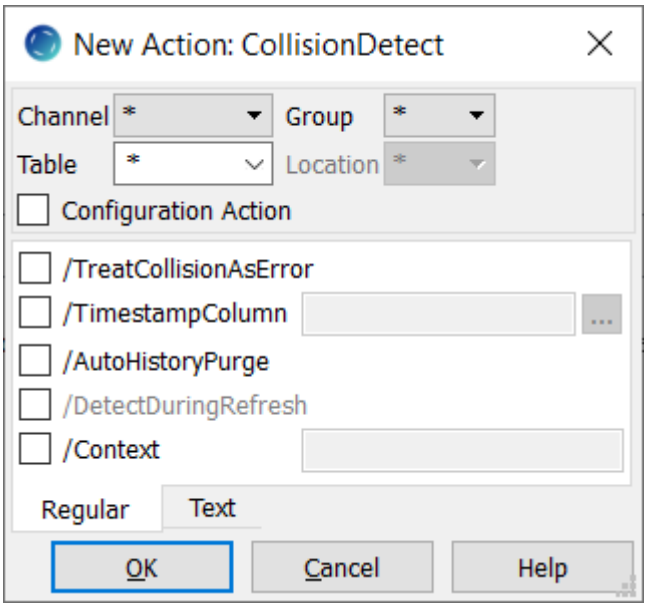
Collision detection requires that replicated tables have a reliable last-updated timestamp column indicating when the data was last updated. Such a column must be manually added to each table involved in the replication and defined in parameter **/TimestampColumn**.

For Oracle and Ingres databases, if parameter **/TimestampColumn** is not defined, HVR maintains extra timestamp information for each tuple in a special history table (named *tbl*__**h**). This table is created and maintained in both capture and integrate locations for each replicated table. The old rows in this history table must be periodically purged using timestamp information from the 'integrate receive timestamp' table (see also section Integrate Receive Timestamp Table). In this case, parameter **/AutoHistoryPurge** must be defined.

⚠ Action **CollisionDetect** is supported only for certain location classes depending on the parameter defined with the action. For the list of supported location classes, see the corresponding sections for **CollisionDetect** in **Capabilities**.

Parameters

This section describes the parameters available for action **CollisionDetect**.



Parameter	Argument	Description
/TreatCollisionAsError		Treat a collision as an error instead of performing automatic resolution using the 'first wins' rule. If **Integrate** **/OnErrorSaveFailed** is defined then the collision will be written to the fail table and the integration of other changes will continue. If **/OnErrorSaveFailed** not defined then the integrate job will keep failing until the collision is cleared, either by deleting the row from the history table or by deleting the transaction file in the **HVR_CONFIG/router** directory.
/TimestampColumn	*col_name*	Exploit a timestamp column named *col_name* in the replicated table for collision detection. By relying on the contents of this column collision detection can avoid the overhead of updating the history table. Deletes must still be recorded. One disadvantage of this parameter is that collision handling relies on this column being filled accurately by the application. Another disadvantage is that if there is more than one database where changes can occur then if a change occurs in the same second, then the collision cannot be detected properly.
/AutoHistoryPurge		Delete rows from history table once the receive stamp table indicates that they are no longer necessary for collision detection. These rows can also be deleted using command **hvrhistorypurge**.
/DetectDuringRefresh	*colname*	During row-wise refresh, discard updates if the timestamp value in *colname* is newer in the target then the source. This parameter can be used with **Hvrrefresh** **–mui** to reconcile the difference between two databases without removing newer rows from either. This parameter must be used with parameter **/Context** (e.g. **/Context**=refr). **CollisionDetect** with parameter **/DetectDuringRefresh** can be used for any supported DBMS.
/Context		Action only applies if refresh or compare context matches.

Manually Purging History Tables

Command **hvrhistorypurge** will purge old data from collision handling history tables. This is done automatically after integration if **5702523** **/AutoHistoryPurge** is defined. The command can be called as follows:

```
$ hvrhistorypurge [-a] [-ffreq] [-hclass] [-ttbl]… [-uuser] hubdb chn loc
```

The first argument *hubdb* specifies the connection to the hub database. This can be an Oracle, Ingres, SQL Server, DB2 for LUW, DB2 for i, PostgreSQL, or Teradata database depending on its form. See further section Calling HVR on the Command Line. The following options are allowed:

Parameter	Description
–a	Purge all history, not just changes older than receive stamps.
–f*freq*	Commit frequency. By **default** a commit is done after every 100 deletes.
–t*tbl*	Only parse output for a specific table. Alternatively, a specific table can be omitted using form **–t|***tbl*. This option can be specified multiple times.

ColumnProperties

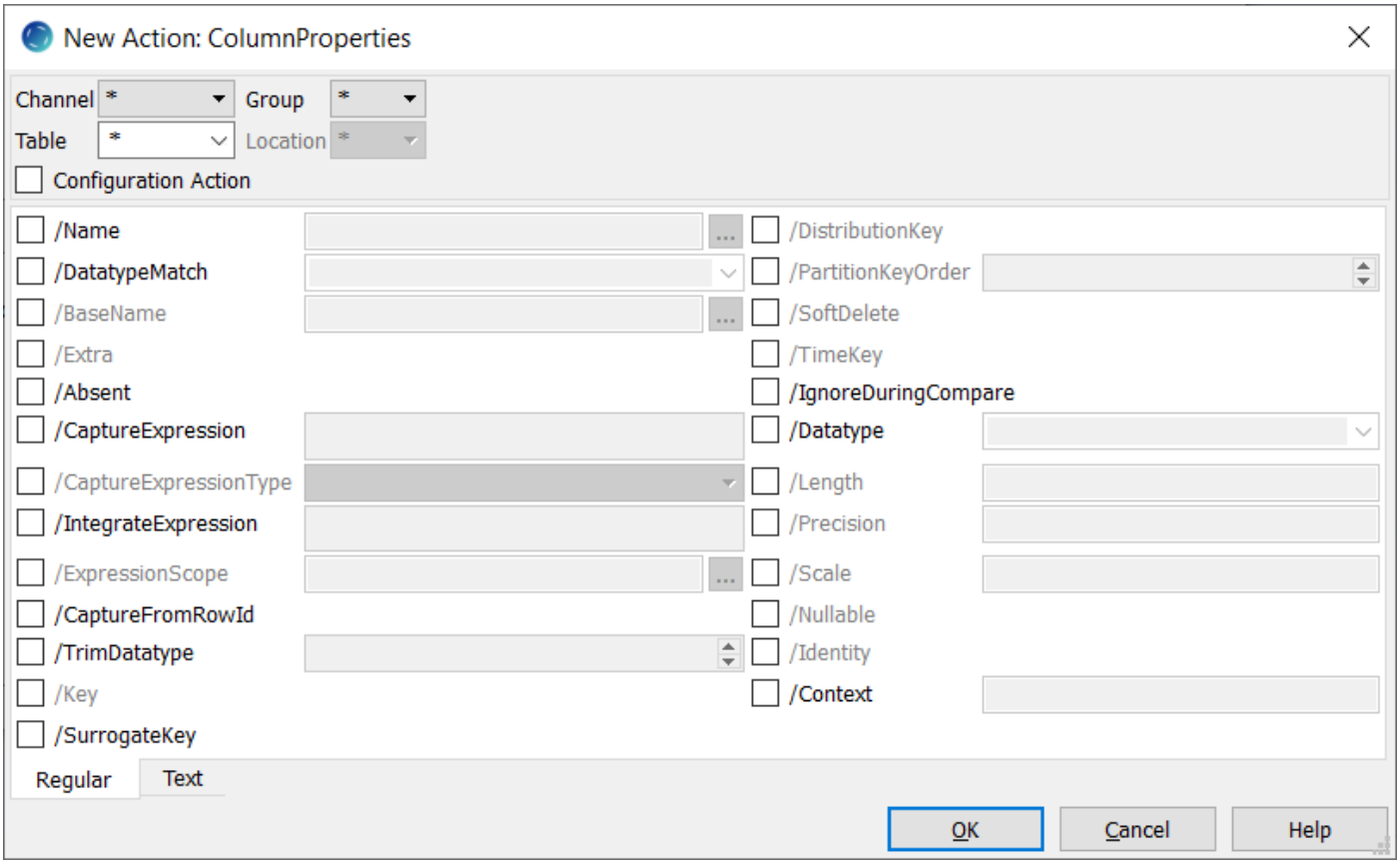
Contents

Description

Action **ColumnProperties** defines properties of a column. This column is matched either by using parameter **/Name** or **/DataType**. The action itself has no effect other than the effect of the other parameters used. This affects both replication (capture and integration) and HVR refresh and compare.

Parameters

This section describes the parameters available for action **ColumnProperties**.



Parameter	Argument	Description

/Name	*col_name*	Name of column in **hvr_column** catalog. ⚠ The *col_name* should not be same as the substitution defined in **/CaptureExpression** and **/IntegrateExpression**. HVR will not populate values for the column if *col_name* and substitution (*sql_expr*) are same. For example, when **/IntegrateExpression=hvr_op** is used then in that action definition **/Name=hvr_op** should not be used.
/DatatypeMatch	*datatypematch*	Data type used for matching a column, instead of **/Name**. **Since** v5.3.1/3 Value *datatypematch* can either be single data type name (such as **number**) or have form *datatype***[***condition***]**.*condition* has form *attribute operator value*. *attribute* can be **prec**, **scale**, **bytelen**, **charlen**, **encoding** or **null** *operator* can be **=**, **<>**, **!=**, **<**, **>**, **<=** or **>=** *value* is either an integer or a single quoted string. Multiple conditions can be supplied, which must be separated by &&. This parameter can be used to associate a **ColumnProperties** action with all columns which match the data type and the optional attribute conditions. Examples are: **/DatatypeMatch**=**"number"** **/DatatypeMatch**=**"number[prec>=19]"** **/DatatypeMatch**=**"varchar[bytelen>200]"** **/DatatypeMatch**=**"varchar[encoding='UTF-8' && null='true']"** **/DatatypeMatch**=**"number[prec=0 && scale=0]"** matches Oracle numbers without any explicit precision or scale.

/BaseName	*tbl_name*	This action defines the actual name of the column in the database location, as opposed to the column name that HVR has in the channel. This parameter is needed if the 'base name' of the column is different in the capture and integrate locations. In that case the column name in the HVR channel should have the same name as the 'base name' in the capture database and parameter **/BaseName** should be defined on the integrate side. An alternative is to define the **/BaseName** parameter on the capture database and have the name for the column in the HVR channel the same as the base name in the integrate database. The concept of the 'base name' in a location as opposed to the name in the HVR channel applies to both columns and tables, see **/BaseName** in **TableProperties**. Parameter **/BaseName** can also be defined for file locations (to change the name of the column in XML tag) or for Salesforce locations (to match the Salesforce API name). Parameter **/BaseName** cannot be used together with **/Extra** and **/Absent**.
/Extra		Column exists in database but not in **hvr_column** catalog. If a column has **/Extra** then its value is not captured and not read during refresh or compare. If the value is omitted then appropriate default value is used (null, zero, empty string, etc.). Parameter **/Extra** cannot be used together with **/BaseName** and **/Absent**. Parameters **/Extra** cannot be used on columns which are part of the replication key. Also, it cannot be defined in a given database on the same column, nor can either be combined on a column with parameter **/BaseName**.
/Absent		Column does not exist in database table. If no value is supplied with **/CaptureExpression** then an appropriate default value is used (null, zero, empty string, etc.). When replicating between two tables with a column that is in one table but is not in the other there are two options: either register the table in the HVR catalogs with all columns and add parameter **/Absent**; or register the table without the extra column and add parameter **/Extra**. The first option may be slightly faster because the column value is not sent over the network. Parameter **/Absent** cannot be used together with **/BaseName** and **/Extra**. Parameters **/Absent** cannot be used on columns which are part of the replication key. Also, it cannot be defined in a given database on the same column, nor can either be combined on a column with parameter **/BaseName**.
/CaptureExpression	*sql_expr*	SQL expression for column value when capturing changes or reading rows. This value may be a constant value or an SQL expression. This parameter can be used to 'map' values data values between a source and a target table. An alternative way to map values is to define an SQL expression on the target side using **/IntegrateExpression**. Possible SQL expressions include **null**, **5** or '**hello**'. For many databases (e.g. Oracle and SQL Server) a subselect can be supplied, for example **select descrip from lookup where id={id}**.

The following substitutions are allowed:

- **{***colname* [*spec*]**}** is replaced/substituted with the value of current table's column *colname*. If the target column has a character based data type or if **/Datatype**=*character data type* then the default format is **%[localtime] %Y-%m-%d %H:%M:%S**, but this can be overridden using the timestamp substitution format specifier *spec*. For more information, see Timestamp Substitution Format Specifier.
- **{hvr_cap_loc}** is replaced with the name of the source location where the change occurred.
- **{hvr_cap_tstamp** [*spec*]**}** is replaced with the moment (time) that the change occurred in source location. If the target column has a character based data type or if **/Datatype** =*character data type* then the default format is **%[localtime] %Y-%m-%d %H:%M:%S**, but this can be overridden using the timestamp substitution format specifier *spec*. For more information, see Timestamp Substitution Format Specifier.
- **{hvr_cap_user}** is replaced with the name of the user which made the change.
- **{{hvr_col_name}}** is replaced with the value of the current column.
- **{hvr_var**_*xxx***}** is replaced with value of 'context variable' *xxx* . The value of a context variable can be supplied using option **–V***xxx*=*val* in **hvrrefresh** or **hvrcompare**.
- **{hvr_slice_num}**: is replaced with the current slice number if slicing is defined with **Count** (option **-S** *num*) in **hvrrefresh** or **hvrcompare**. **Since** v5.6.5/0
- **{hvr_slice_total}**: is replaced with the total number of slices if slicing is defined with **Count** (option **-S** *num*) in **hvrrefresh** or **hvrcompare**. **Since** v5.6.5/0
- **{hvr_slice_value}**: is replaced with the current slice value if slicing is defined with **Series** (option **-S** *val1*[**;***val2*]...) in **hvrrefresh** or **hvrcompare**. **Since** v5.6.5/0

⚠ It is recommended to define **/Context** when using the substitutions **{hvr_var**_*xxx***}**, **{hvr_slice_num}**, **{hvr_slice_total}**, or **{hvr_slice_value}**, so that it can be easily disabled or enabled.

{hvr_slice_num}, **{hvr_slice_total}**, **{hvr_slice_value}** cannot be used if the one of the old slicing substitutions **{hvr_var_slice_condition}**, **{hvr_var_slice_num}**, **{hvr_var_slice_total}**, or **{hvr_var_slice_value}** is defined in the channel/table involved in the compare /refresh.

/CaptureExpressionType **Since** v5.3.1/21	*expr_type*	Type of mechanism used by HVR capture, refresh and compare job to evaluate value in parameter **/CaptureExpression**. Available options: • **SQL_PER_CYCLE** (**default** for database locations if the capture expression matches a pattern in file **hvr_home/lib/constsqlexpr.pat)**: The capture job only evaluates the expression once per replication cycle, so every row captured by that cycle will get the same value. It requires less database 'round-trips' than **SQL_PER_ROW** and **SQL_WHERE_ROW**. For refresh and compare jobs the expression is just included in main select statement, so no extra database round-trips are used and the database could assign each row a different value. This type is is not supported for file locations. • **SQL_PER_ROW** (**default** for database locations if the capture expression does not match a pattern in file **hvr_home/lib/constsqlexpr.pat**): The capture job evaluates the expression for each change captured. This means every row captured by that cycle could get a different value but requires more database 'round-trips' than **SQL_PER_CYCLE**. For refresh and compare jobs the expression is just included in main **select** statement, so no extra database round-trips are used and the database could assign each row a different value. This type is is not supported for file locations. • **SQL_WHERE_ROW**: The capture job evaluates the expression for each change captured.but with an extra **where** clause containing the key value for the table on which the change occurred. This allows that expression to include expressions like **{colx}** which reference other columns of that table. Each row captured could get a different value but requires more database 'round-trips' than **SQL_PER_CYCLE**. For refresh and compare jobs the expression is just included in main **select** statement (without the extra **where** clause), so no extra database round-trips are used and the database could assign each row a different value. This type is not supported for file locations. • **INLINE**: String-based replacement by HVR itself. This type is only supported for capturing changes from file location.
/IntegrateExpression	*sql_expr*	Expression for column value when integrating changes or loading data into a target table. HVR may evaluate itself or use it as an SQL expression. This parameter can be used to 'map' values between a source and a target table. An alternative way to map values is to define an SQL expression on the source side using /CaptureExpression. For many databases (e.g. Oracle and SQL Server) a subselect can be supplied, for example **select descrip from lookup where id={id}**. Possible expressions include **null**, **5** or '**hello**'. The following substitutions are allowed:

- **{***colname* [*spec*]**}** is replaced/substituted with the value of current table's column *colname*. If the target column has a character based data type or if **/Datatype**=*character data type* then the default format is **%[localtime] %Y-%m-%d %H:%M:%S**, but this can be overridden using the timestamp substitution format specifier *spec*. For more information, see Timestamp Substitution Format Specifier.
- **{hvr_cap_loc}** is replaced with the name of the source location where the change occurred.
- **{hvr_cap_user}** is replaced with the name of the user who made the change.
- **{hvr_cap_tstamp** [*spec*]**}** is replaced with the moment (time) that the change occurred in source location. If the target column has a character based data type or if **/Datatype**=*character data type* then the default format is **%[localtime] %Y-%m-%d %H:%M:%S**, but this can be overridden using the timestamp substitution format specifier *spec*. For more information, see Timestamp Substitution Format Specifier.
- **{hvr_chn_name}** is replaced with the name of the channel.
- **{{hvr_col_name}}** is replaced with the value of the current column.
- **{hvr_integ_key}** is replaced with a 16 byte string value (hex characters) which is unique and continuously increasing for all rows integrated into the target location. The value is calculated using a high precision timestamp of the moment that the row is integrated. This means that if changes from the same source database are captured by different channels and delivered to the same target location then the order of this sequence will not reflect the original change order. This contrasts with substitution **{hvr_integ_seq}** where the order of the value matches the order of the change captured. Another consequence of using a (high precision) integrate timestamp is that if the same changes are sent again to the same target location (for example after option 'capture rewind' of **hvrinit**, or if a Kafka location's integrate job is restarted due to interruption) then the 're-sent' change will be assigned a new value. This means the target databases cannot rely on this value to detect 're-sent' data. This substitution is recommended for **ColumnProperties /TimeKey** if the channel has multiple source locations.
- **{hvr_integ_seq}** is replaced with a 36 byte string value (hex characters) which is unique and continuously increasing for a specific source location. If the channel has multiple source locations then this substitution must be combined with **{hvr_cap_loc}** to give a unique value for the entire target location. The value is derived from source database's DBMS logging sequence, e.g. the Oracle System Change Number (SCN). This substitution is recommended for **ColumnProperties /TimeKey** if the channel has a single source location. This substitution only has a value during **Integrate** or during **Refresh** if a select moment (option **-M**) was specified.

- **{hvr_integ_tstamp** [*spec*]**}** is replaced with the moment (time) that the change was integrated into target location. If the target column has a character based data type or if **/Datatype**=*character data type* then the default format is **%Y-%m-%d %H:%M:%S[.SSS]**, but this can be overridden using the timestamp substitution format specifier spec. For more information, see Timestamp Substitution Format Specifier.
- **{{hvr_key_names** *sep***}}** is replaced with the values of table's key columns, concatenated together with separator *sep*.
- **{hvr_key_names** *sep***}** is replaced with the names of table's key columns, concatenated together with separator *sep*. **Since** v5.5.5/8
- **{hvr_op}** is replaced with the HVR operation type. Values are **0** (delete), **1** (insert), **2** (after update), **3** (before key update), **4** (before non–key update) or **5** (truncate table). See also Extra Columns for Capture, Fail and History Tables. Note that this substitution cannot be used with parameter **/ExpressionScope**.
- **{hvr_schema}** is replaced with the schema name of the table.
- **{hvr_tbl_name}** is replaced with the name of the current table.
- **{hvr_tbl_base_name}** is replaced with the base name of the current table
- **{hvr_tx_countdown}** is replaced with countdown of changes within transaction, for example if a transaction contains three changes the first change would have countdown value **3**, then **2**, then **1**. A value of **0** indicates that commit information is missing for that change. This substitution only has a value during **Integrate** or during **Refresh** if a select moment (option **-M**) was specified.
- **{hvr_tx_scn}** is replaced with the source location's SCN (Oracle). This substitution can only be used if the source location database is Oracle. This substitution can only be used for ordering if the channel has a single source location. This substitution only has a value during **Integrate** or during **Refresh** if a select moment (option **-M**) was specified.
- **{hvr_tx_seq}** is replaced with a hex representation of the sequence number of transaction. For capture from Oracle this value can be mapped back to the SCN of the transaction's commit statement. Value **[hvr_tx_seq, hvr_tx_countdown]** is increasing and uniquely identifies each change. **{hvr_tx_seq}** gets value only if **Select Moment** (option **-M**) is selected while performing HVR Refresh. This substitution only has a value during **Integrate** or during **Refresh** if a select moment (option **-M**) was specified.
- **{hvr_var**_*xxx***}** is replaced with value of 'context variable' *xxx*. The value of a context variable can be supplied using option **–V***xxx*=*val* to command **hvrrefresh** or **hvrcompare**.
- **{hvr_slice_num}**: is replaced with the current slice number if slicing is defined with **Count** (option **-S** *num*) in **hvrrefresh** or **hvrcompare**. **Since** v5.6.5/0
- **{hvr_slice_total}**: is replaced with the total number of slices if slicing is defined with **Count** (option **-S** *num*) in **hvrrefresh** or **hvrcompare**. **Since** v5.6.5/0

- **{hvr_slice_value}**: is replaced with the current slice value if slicing is defined with **Series** (option **-S** *val1*[**;***val2*]...) in **hvrrefresh** or **hvrcompare**. **Since** v5.6.5/0

It is recommended to define **/Context** when using the substitutions **{hvr_var_*xxx*}**, **{hvr_slice_num}**, **{hvr_slice_total}**, or **{hvr_slice_value}**, so that it can be easily di29hvr_slicm (,)67 0.3sily /F2 9.Tm Tm Tm 21.59 593.85 l 520.

/ExpressionScope	*expr_scope*	Scope for which operations (e.g. insert or delete) an integrate expression (parameter **/IntegrateExpression**) should be used. Value *expr_scope* should be a comma-separated list of the one of the following; **DELETE**, **INSERT**, **UPDATE_AFTER** or **TRUNCATE**. Values **DELETE** and **TRUNCATE** can be used only if parameter **/SoftDelete** or **/TimeKey** is defined. Currently this parameter can be used with integration when parameter **/Burst** is defined. It is ignored for database targets if **/Burst** is not defined and for file-targets (such as HDFS or S3). This burst restriction means that no scopes exist yet or for 'update before' operations (such as **UPDATE_BEFORE_KEY** and **UPDATE_BEFORE_NONKEY**). Only bulk refresh obeys this parameter (it always uses scope **INSERT**); row-wise refresh ignores the expression scope. This value of the affected **/IntegrateExpression** parameter can contain its regular substitutions except for **{hvr_op}** which cannot be used. **Example 1:** To add a column **opcode** to a target table (defined with **/SoftDelete**) containing values **'I'**, **'U'** and **'D'** (for insert, update and delete respectively), define these actions; • **ColumnProperties /Name=opcode /Extra /IntegrateExpression="'I'" /ExpressionScope=INSERT /Datatype=varchar /Length=1 /Nullable** • **ColumnProperties /Name=opcode /Extra /IntegrateExpression="'U'" /ExpressionScope=UPDATE /Datatype=varchar /Length=1 /Nullable** • **ColumnProperties /Name=opcode /Extra /IntegrateExpression="'D'" /ExpressionScope=DELETE /Datatype=varchar /Length=1 /Nullable** **Example 2:** To add a column **insdate** (only filled when a row is inserted) and column **upddate** (filled on update and [soft] delete), define these actions; • **ColumnProperties /Name=insdate /Extra /IntegrateExpression=sysdate /ExpressionScope=INSERT /Datatype=timestamp** • **ColumnProperties /Name=upddate /Extra /IntegrateExpression=sysdate /ExpressionScope=DELETE,UPDATE_AFTER /Datatype=timestamp** Note that HVR Refresh can create the target tables with the **/Extra** columns, but if the same column has multiple actions for different scopes then these must specify the same data type (parameters **/Datatype** and **/Length**).
/CaptureFromRowId		Capture values from table's DBMS row-id. Define on the capture location. This parameter is supported only for certain location classes. For the list of supported location class, see Log-based capture from hidden rowid column in Capabilities. ⚠ This parameter is not supported for Oracle's Index Organized Tables (IOT).

/TrimDatatype	*int*	Reduce width of data type when selecting or capturing changes. This parameter affects string data types (such as **varchar**, **nvachar** and **clob**) and binary data types (such as **raw** and **blob**). The value is a limit in bytes; if this value is exceeded then the column's value is truncated (from the right) and a warning is written. An example of usage is **ColumnProperties /DatatypeMatch**=**clob /TrimDatatype**=**10 /Datatype**=**varchar /Length**=**30** which will replicate all columns with data type **clob** into a target table as strings. Note that parameter **/Datatype** and **/Length** ensures that HVR Refresh will create target tables with the smaller data type. Its length is smaller because **/Length** parameter is used. This parameter is supported only for certain location classes. For the list of supported location class, see Reduce width of datatype when selecting or capturing changes in Capabilities.
/Key		Add column to table's replication key.
/SurrogateKey		Use column instead of the regular key during replication. Define on the capture and integrate locationsT BT /F1 9.75 Tf 1 0 0 1 242.89 620 thaie's f utabl(Oracle)ey.
nProperti Nara(=)Tj ET BT /F2 9.75 Tf 1 0 0 1 390.21 260.02 Tml c2ar **nProperti Nara(=)Tj ET BT /F2 9.75 Tf 1 0 0 1 390.21 208.99 Tml c3cl** **nProperti Nara(=)Tj ET BT /F2 9.75 Tf 1 0 0 1 390.21 132.37 Tml c3cl**		

/SoftDelete		Convert deletes to update of this column to 1. Value 0 means not deleted.
/TimeKey		Convert all changes (inserts, updates and deletes) into inserts, using this column for time dimension. Defining this parameter affects how all changes are delivered into the target table. This parameter is often used with **/IntegrateExpression**=**{hvr_integ_seq}**, which will populate a value.
/IgnoreDuringCompare		Ignore values in this column during compare and refresh. Also during integration this parameter means that this column is overwritten by every update statement, rather than only when the captured update changed this column. This parameter is ignored during row-wise compare/refresh if it is defined on a key column.
/Datatype	*data_type*	Data type in database if this differs from **hvr_column** catalog.
/Length	*attr_val*	String length in database if this differs from value defined in **hvr_column** catalog. When used together with **/Name** or **/DatatypeMatch**, keywords **bytelen** and **charlen** can be used and will be replaced by respective values of matched column. Additionally, basic arithmetic (+,-,*,/) can be used with **bytelen** and **charlen**, e.g., **/Length**=**"bytelen/3"** will be replaced with the byte length of the matched column divided by 3. Parameter **/Length** can only be used if **/Datatype** is defined.
/Precision	*attr_val*	Integer precision in database if this differs from value defined in **hvr_column** catalog. When used together with **/Name** or **/DatatypeMatch**, keywords **prec** can be used and will be replaced by respective values of matched column. Additionally, basic arithmetic (+,-,*,/) can be used with **prec**, e.g., **/Precision**=**"prec+5"** will be replaced with the precision of the matched column plus 5. Parameter **/Precision** can only be used if **/Datatype** is defined.
/Scale	*attr_val*	Integer scale in database if this differs from value defined in **hvr_column** catalog. When used together with **/Name** or **/DatatypeMatch**, keyword **scale** can be used and will be replaced by respective values of matched column. Additionally, basic arithmetic (+,-,*,/) can be used with **scale**, e.g., **/Scale**=**"scale*2"** will be replaced with the scale of the matched column times 2. Parameter **/Scale** can only be used if **/Datatype** is defined.
/Nullable		Nullability in database if this differs from value defined in **hvr_column** catalog. Parameter **/Nullable** can only be used if **/Datatype** is defined.
/Identity **SQL Server**		Column has SQL Server identity attribute. Only effective when using integrate database procedures (**Integrate /DbProc**). Parameter **/Identity** can only be used if **/Datatype** is defined.

/Context	*ctx*	Ignore action unless refresh/compare context *ctx* is enabled. The value should be the name of a context (a lowercase identifier). It can also have form **!***ctx*, which means that the action is effective unless context *ctx* is enabled. One or more contexts can be enabled for HVR Compare or Refresh (on the command line with option **–C***ctx*). Defining an action which is only effective when a context is enabled can have different uses. For example, if action **5702756 /IgnoreDuringCompare /Context**=**qqq** is defined, then normally all data will be compared, but if context **qqq** is enabled (**-Cqqq**), then the values in one column will be ignored.

Columns Which Are Not Enrolled In Channel

Normally all columns in the location's table (the 'base table') are enrolled in the channel definition. But if there are extra columns in the base table (either in the capture or the integrate database) which are not mentioned in the table's column information of the channel, then these can be handled in two ways:

- They can be included in the channel definition by adding action **ColumnProperties /Extra** to the specific location. In this case, the SQL statements used by HVR integrate jobs will supply values for these columns; they will either use the **/IntegrateExpression** or if that is not defined, then a default value will be added for these columns (**NULL** for nullable data types, or **0** for numeric data types, or **''** for strings).
- These columns can just not be enrolled in the channel definition. The SQL that HVR uses for making changes will then not mention these 'unenrolled' columns. This means that they should be nullable or have a default defined; otherwise, when HVR does an insert it will cause an error. These 'unenrolled' extra columns are supported during HVR integration and HVR compare and refresh, but are not supported for HVR capture. If an 'unenrolled' column exists in the base table with a default clause, then this default clause will normally be respected by HVR, but it will be ignored during bulk refresh on Ingres, or SQL Server unless the column is a 'computed' column.

Substituting Column Values Into Expressions

HVR has different actions that allow column values to be used in SQL expressions, either to map column names or to do SQL restrictions. Column values can be used in these expressions by enclosing the column name embraces, for example a restriction "**{price} > 1000**" means only rows where the value in price is higher than 1000.

But in the following example it could be unclear which column name should be used in the braces;

> Imagine you are replicating a source base table with three columns (**A**, **B**, **C**) to a target base table with just two columns named (**E**, **F**). These columns will be mapped together using HVR actions such as **ColumnProperties /CaptureExpression** or **/IntegrateExpression**. If these mapping expressions are defined on the target side, then the table would be enrolled in the HVR channel with the source columns (**A**, **B**, **C**). But if the mapping expressions are put on the source side then the table would be enrolled with the target columns (**D**, **E**). Theoretically mapping expressions could be put on both the source and target, in which case the columns enrolled in the channel could be different from both, e.g. (**F**, **G**, **H**), but this is unlikely.
>
> But when an expression is being defined for this table, should the source column names be used for the brace substitution (e.g. **{A}** or **{B}**)? Or should the target parameter be used (e.g. **{D}** or **{E}**)? The answer is that this depends on which parameter is being used and it depends on whether the SQL expression is being put on the source or the target side.

For parameters **/IntegrateExpression** and **/IntegrateCondition** the SQL expressions can only contain {} substitutions with the column names as they are enrolled in the channel definition (the "HVR Column names"), not the "base table's" column names (e.g. the list of column names in the target or source base table). So in the example above substitutions **{A}** **{B}** and **{C}** could be used if the table was enrolled with the columns of the source and with mappings on the target side, whereas substitutions **{E}** and **{F}** are available if the table was enrolled with the target columns and had mappings on the source.

But for **/CaptureExpression** **/CaptureCondition** and **/RefreshCondition** the opposite applies: these expressions must use the "base table's" column names, not the "HVR column names". So in the example these parameters could use **{A}** **{B}** and **{C}** as substitutions in expressions on the source side, but substitutions **{E}** and **{F}** in expressions on the target.

Timestamp Substitution Format Specifier

Timestamp substitution format specifiers allows explicit control of the format applied when substituting a timestamp value. These specifiers can be used with **{hvr_cap_tstamp**[*spec*]**}**, **{hvr_integ_tstamp**[*spec*]**}**, and **{***colname* [*spec*]**}** if *colname* has timestamp data type. The components that can be used in a timestamp format specifier *spec* are:

Component	Description	Example
%a	Abbreviate weekday according to current locale.	Wed
%b	Abbreviate month name according to current locale.	Jan
%d	Day of month as a decimal number (01–31).	07
%H	Hour as number using a 24–hour clock (00–23).	17
%j	Day of year as a decimal number (001–366).	008
%m	Month as a decimal number (01 to 12).	04
%M	Minute as a decimal number (00 to 59).	58
%s	Seconds since epoch (1970–01–01 00:00:00 UTC).	1099928130
%S	Second (range 00 to 61).	40
%T	Time in 24–hour notation (%H:%M:%S).	17:58:40
%U	Week of year as decimal number, with Sunday as first day of week (00 – 53).	30
%V **Linux**	The ISO 8601 week number, range 01 to 53, where week 1 is the first week that has at least 4 days in the new year.	15
%w	Weekday as decimal number (0 – 6; Sunday is 0).	6
%W	Week of year as decimal number, with Monday as first day of week (00 – 53)	25
%y	Year without century.	14
%Y	Year including the century.	2014
%[localtime] **Since** v5.5.5/xx	Perform timestamp substitution using machine local time (not UTC). This component should be at the start of the specifier (e.g. **{{hvr_cap_tstamp %[localtime]%H}}**).	
%[utc] **Since** v5.5.5/xx	Perform timestamp substitution using UTC (not local time). This component should be at the start of the specifier (e.g. **{{hvr_cap_tstamp %[utc]%T}}**).	

DbObjectGeneration

Contents

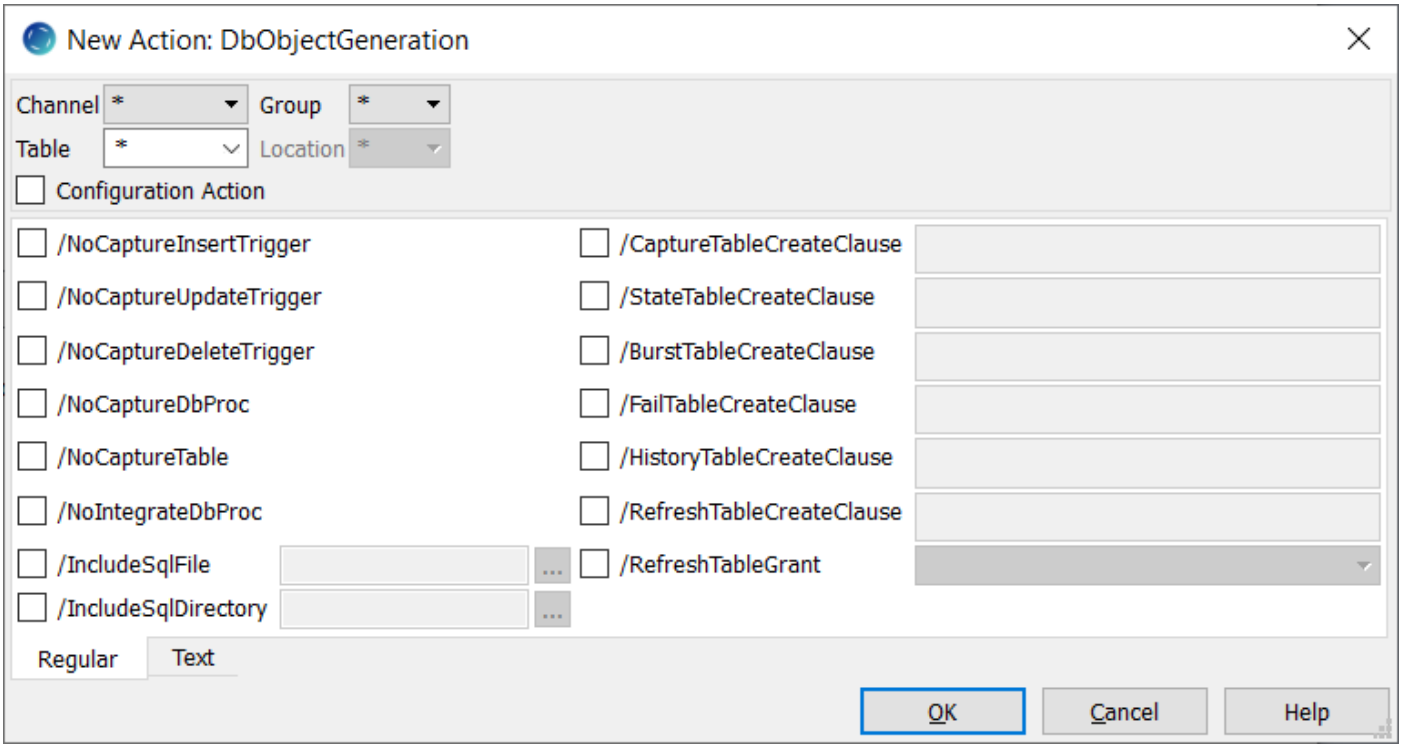
Description

Action **DbObjectGeneration** allows control over the database objects which are generated by HVR in the replicated databases. The action has no effect other than that of its parameters.

Parameters **/NoCaptureInsertTrigger, /NoCaptureUpdateTrigger, /NoCaptureDeleteTrigger, /NoCaptureDbProc, /NoCaptureTable** can either be used to inhibit capturing of changes for trigger–based capture or can be used with parameter **/IncludeSqlFile** to replace the procedures that HVR would normally generate with new procedures containing special logic.

Parameters

This section describes the parameters available for action **DbObjectGeneration**.



Parameter	Argument	Description
/NoCaptureInsertTrigger		Inhibit generation of capture insert trigger/rule.
/NoCaptureUpdateTrigger		Inhibit generation of capture update trigger/rule.
/NoCaptureDeleteTrigger		Inhibit generation of capture delete trigger/rule.

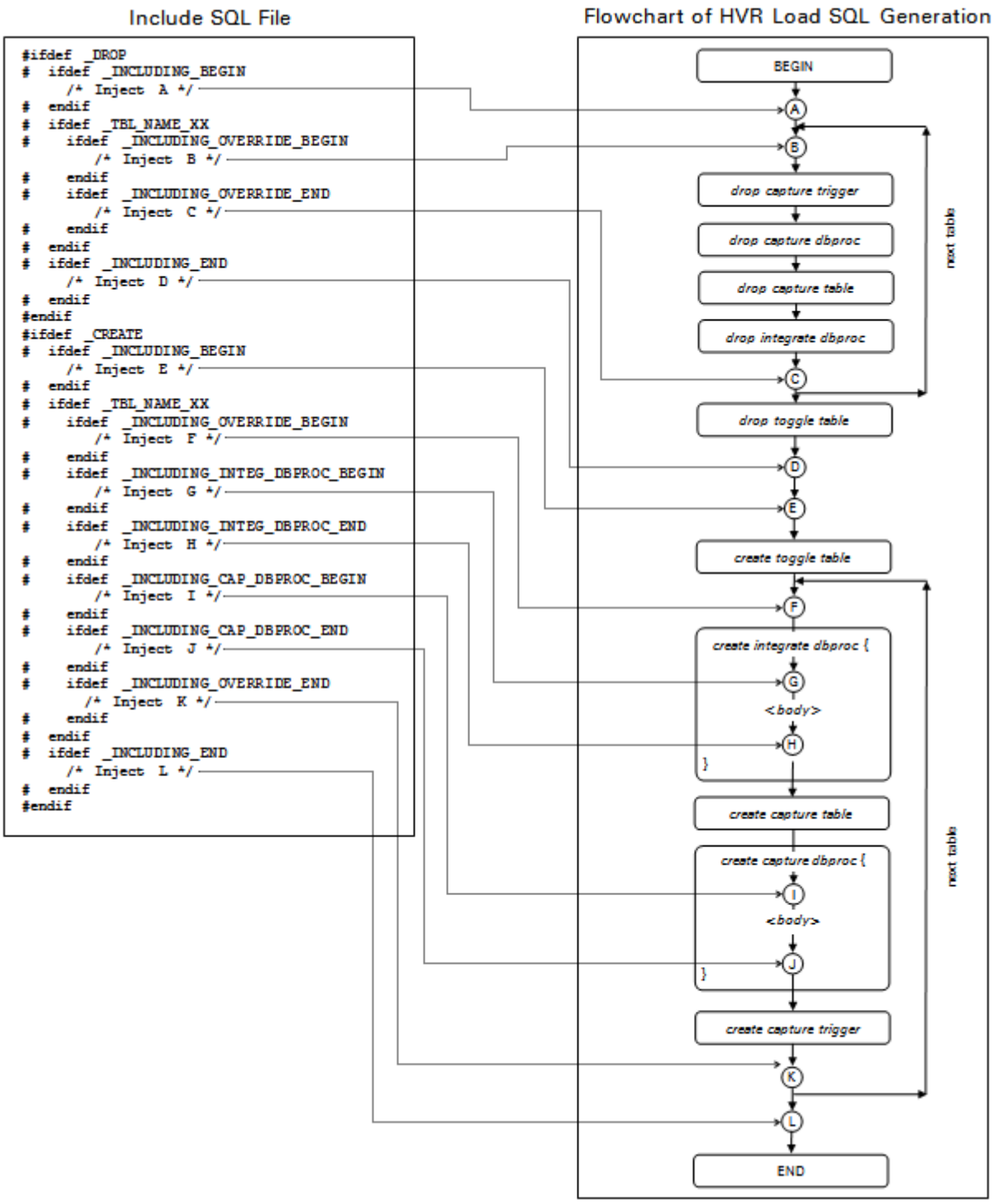
/NoCaptureDbProc		Inhibit generation of capture database procedures.
/NoCaptureTable		Inhibit generation of capture tables for trigger–based capture.
/NoIntegrateDbProc		Inhibit generation of integrate database procedures.
/IncludeSqlFile	*file*	Include file for customizing database objects. Argument *file* can be an absolute pathname or a relative path in a directory specified with **/IncludeSqlDirectory**. Option **–S** of **Hvrinit** can be used to generate the initial contents for this file. If this parameter is defined for any table, then it affects all objects generated for that location.
/IncludeSqlDirectory	*dir*	Search directory *dir* for include SQL file. If this parameter is defined for any table, then it affects all objects generated for that location.
/CaptureTableCreateClause	*sql_expr*	Clause for capture table creation statement.
/StateTableCreateClause	*sql_expr*	Clause for state table creation statement. If this parameter is defined for any table, then it affects all state tables generated for that location.
/BurstTableCreateClause	*sql_expr*	Clause for integrate burst table creation statement.
/FailTableCreateClaus	*sql_expr*	Clause for fail table creation statement. If this parameter is defined for any table, then it affects all tables integrated to that location.
/HistoryTableCreateClause	*sql_expr*	Clause for history table creation statement.
/RefreshTableCreateClause	*sql_expr*	Clause for base table creation statement during refresh. Allow all users to access HVR database objects.
/RefreshTableGrant		Executes a grant statement on the base table created during **HVR Refresh**. Available options: • **NONE** (**default**, if the table is created in default schema): Do not execute grant statement. • **SELECT_TO_PUBLIC**: executes **grant select on** *tablename* **to public** • **ALL_TO_PUBLIC** (**default**): executes **grant all on** *tablename* **to public**

Injecting SQL Include Files

Parameter **/IncludeSqlFile** can be used to inject special logic inside standard SQL which is generated by **Hvrinit**. The SQL that HVR would normally generate can be seen with **Hvrinit** option **–S**. Conditions (using **#ifdef** syntax lent from the C preprocessor), control where abouts this SQL is injected. There are twelve inject points (see diagram below). SQL code will be injected at a specific point depending on the **#ifdef** conditions specified for macros **_INCLUDING_***, **_CREATE**, **_DROP** and **_TABLE_NAME_***. If a file contains none of these conditions then its content will be injected in all twelve injections points.

These sections will not always be generated:

- Triggers are only generated for trigger–based capture locations (**/TriggerBased** defined)
- Integrate database procedures are only defined if **Integrate/DbProc** is defined.
- The **_CREATE** section is omitted if **Hvrinit** option **–d** is defined without **–c**.
- Sections for specific tables are omitted if **Hvrinit** option **–t** is specified for different tables.
- Database procedures are only generated if **Hvrinit** option **–op** is defined or no **–o** option is supplied.
- Database procedures and triggers are only generated if option **–ot** is defined or no **–o** option is supplied.



The following macros are defined by **Hvrinit** for the contents of the file specified by parameter **/IncludeSqlFile**. These can also be used with **#if** or **#ifdef** directives.

Macro	Description
_CREATE	Defined when **Hvrinit** is creating database objects.
_DB_CAPTURE	Defined if action **Capture** is defined on this location.
_DB_INTEGRATE	Defined if action **Integrate** is defined on this location.
_DBPROC_COL_NAMES	Contains the list of columns in the base table, separated by commas.

_DBPROC_COL_VALS	Contains the list of values in the base table, separated by commas.
_DBPROC_KEY_EQ	Contains **where** condition to join database procedure parameters to the key columns of the base table. For example, if the table has keys (**k1**, **k2**), then this macro will have value **k1**=**k1$** and **k2**=**k2$**.
_DROP	Defined when **Hvrinit** is dropping database objects.
_FLAG_OC	Defined when **Hvrinit** option **–oc** or no **–o** option is supplied.
_FLAG_OP	Defined when **Hvrinit** option **–op** or no **–o** option is supplied.
_FLAG_OS	Defined when **Hvrinit** option **–os** or no **–o** option is supplied.
_FLAG_OT	Defined when **Hvrinit** option **–ot** or no **–o** option is supplied.
_HVR_VER	HVR version number.
_HVR_OP_VAL	Defined when **_INCLUDING_INTEG_DBPROC_*** is defined with value 0, 1 or 2. It means the current database procedure is for delete, insert or update respectively.
_INCLUDING_BEGIN	Defined when **Hvrinit** is including the SQL file at the beginning of its SQL.
_INCLUDING_END	Defined when **Hvrinit** is including the SQL file at the end of its SQL.
_INCLUDING_CAP_DBPROC_BEGIN	Defined when **Hvrinit** is including the SQL file at the beginning of each capture database procedure.
_INCLUDING_CAP_DBPROC_DECLARE	Defined when **Hvrinit** is including the SQL file for the declare block of each capture database procedure.
_INCLUDING_CAP_DBPROC_END	Defined when **Hvrinit** is including the SQL file at the end of each capture database procedure.
_INCLUDING_INTEG_DBPROC_BEGIN	Defined when **Hvrinit** is including the SQL file at the beginning of each integrate database procedure.
_INCLUDING_INTEG_DBPROC_DECLARE	Defined when **Hvrinit** is including the SQL file for the declare block of each integrate database procedure.
_INCLUDING_INTEG_DBPROC_END	Defined when **Hvrinit** is including the SQL file at the end of each integrate database procedure.
_INCLUDING_OVERRIDE_BEGIN	Defined as **Hvrinit** is including the SQL file at a point where database objects can be dropped or created. Each SQL statement in this section must be preceded by macro **_SQL_BEGIN** and terminated with macro **_SQL_END**.
_INCLUDING_OVERRIDE_END	Defined as **Hvrinit** is including the SQL file at a point where database objects can be dropped or created. Each SQL statement in this section must be preceded by macro **_SQL_BEGIN** and terminated with macro **_SQL_END**.
_INGRES	Defined when the current location is an Ingres database.
_LOC_DBNAME	Database name.
_LOC_NAME	Name of current location.
_ORACLE	Defined when the current location is an Oracle database.
TBL_NAME_X	Indicates that a database procedure for table *x* is generated. This macro is only defined when **_INCLUDING_*_DBPROC_*** is defined.

_SQL_BEGIN	Macro marking the beginning of an SQL statement in a section for **_INCLUDING_OVERRIDE**.
_SQL_END	Macro marking the end of an SQL statement for an **_INCLUDING_OVERRIDE** section.
_SQLSERVER	Defined when the current location is an SQL Server database.

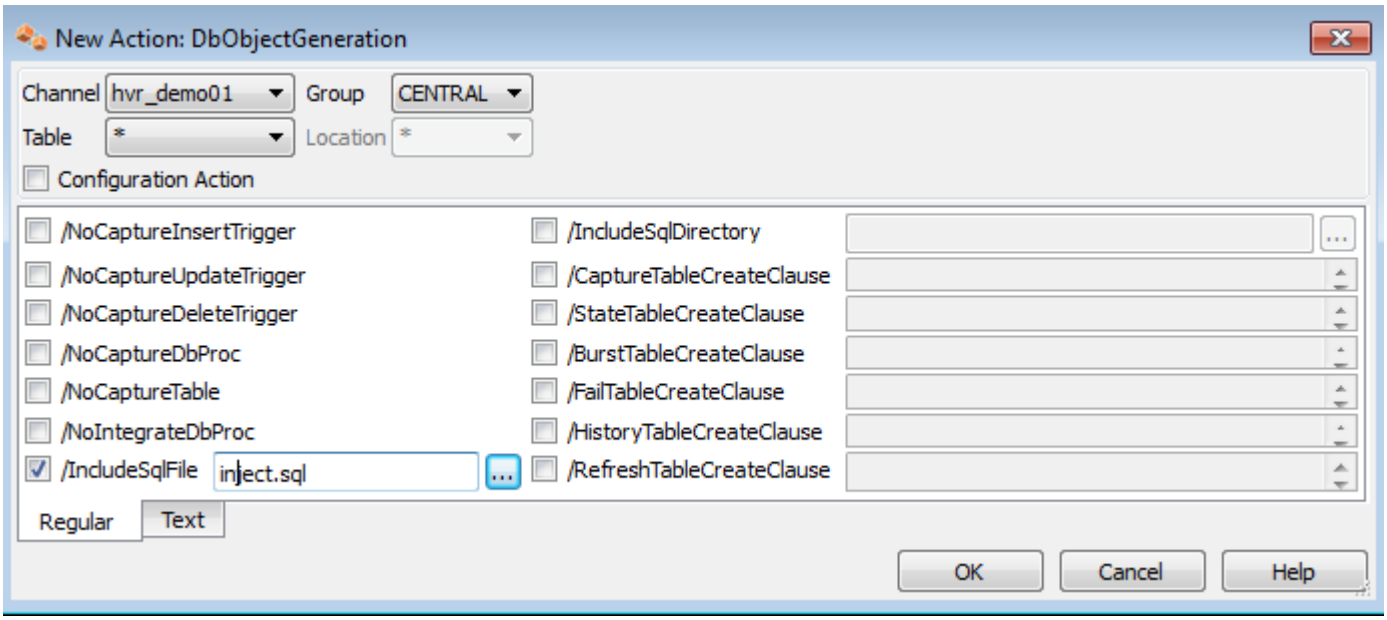
Examples

This section describes examples of using the following parameters of

Example 1

The following example uses action **DbObjectGeneration** to inject some special logic (contained in file **inject.sql**) into the integrate database procedure for table **mytable**. This logic either changes the value of column **status** or deletes the target row if the status has a certain value. Parameter **/DbProc** must also be added to action **Integrate** so that integrate database procedures are generated.

```
#if defined _INCLUDING_INTEG_DBPROC_BEGIN && \
   defined _TBL_NAME_MYTABLE && \
   _HVR_OP_VAL == 2
if :status = 'Status Two' then
       :status = 'Status Three';
elseif  :status = 'Status Four' then
        :status = 'Status Five';
elseif :status = 'Status Six' then
       delete from mytable where id = :id;
       return;
endif;
#endif
```



Example 2

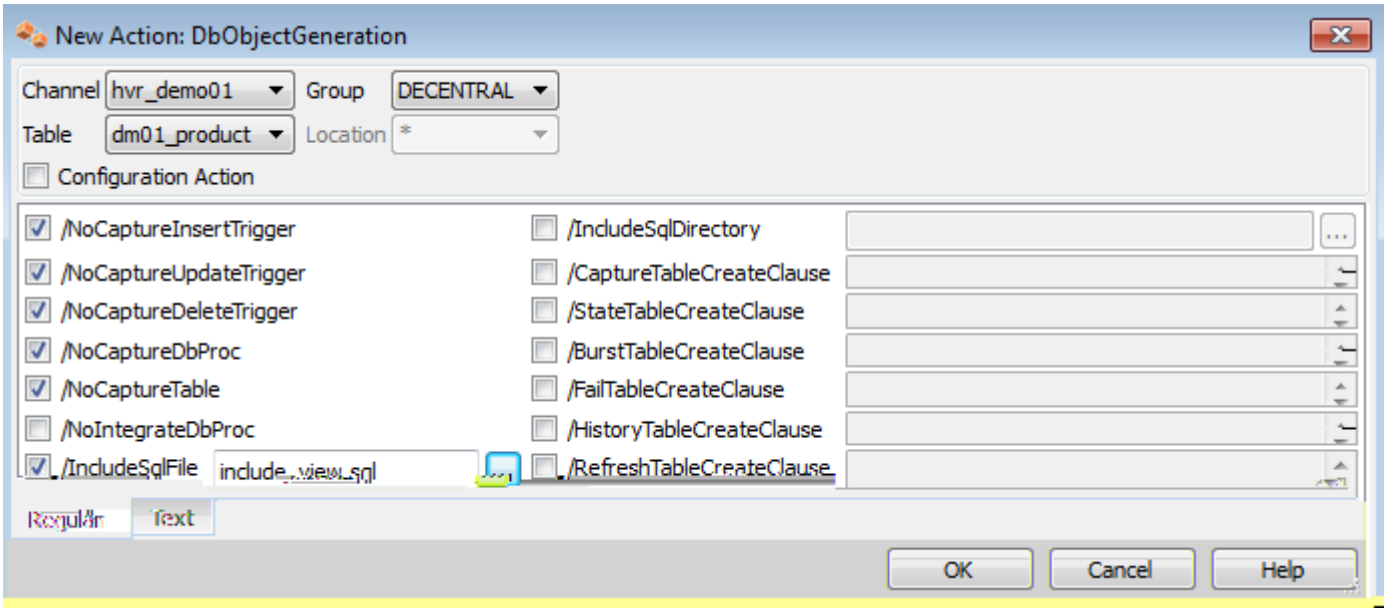
The following example replicates updates to column **balance** of table **account** as differences, instead of as absolute values. The channel should contain the following actions: **Capture** (not log–based), **Integrate /DbProc** (at least for this table) and **DbObjectGeneration /IncludeSqlFile=***thisfile*.

```
#ifdef _TBL_NAME_ACCOUNT
#  ifdef _INCLUDING_CAP_DBPROC_BEGIN
    /* HVR will inject this SQL at the top of capture dbproc account__c */
    /* Note: old value is in <balance>, new value is <balance_> */
    if hvr_op=2 then /* hvr_op=2 means update */
            balance_= balance_ - balance;
    endif;
#  endif
#  if defined _INCLUDING_INTEG_DBPROC_BEGIN && _HVR_OP_VAL == 2
    /* HVR will inject this SQL at the top of integ dbproc account__iu */
    select balance= balance + :balance
    from account
    where account_num = :account_num;
#  endif
#endif
```

Example 3

The following example is a channel that captures changes from SQL views, which are supplied by the end user in file **include_view.sql**. The channel defines the **Capture** for trigger–based capture, but then uses action **DbObjectGeneration** to disable automatic generation of all the trigger–based capture objects. Instead it uses / **IncludeSqlFile** to create a pair of capture views.

```
#if defined _FLAG_OC && defined _DB_CAPTURE
#  ifdef _DROP
#    ifdef _INCLUDING_BEGIN
_SQL_BEGIN_DROP
drop view dm01_order__c0
_SQL_END
_SQL_BEGIN_DROP
drop view dm01_order__c1
_SQL_END
#    endif
#  endif
#  ifdef _CREATE
#    ifdef _INCLUDING_END
_SQL_BEGIN
create view dm01_order__c0 as
   select  ' ' as hvr_tx_id,
           1 as hvr_op,
#      ifdef _ORACLE
           1 as hvr_seq,
           sysdate as hvr_cap_tstamp,
#      endif
#      ifdef _INGRES
           byte('', 8) as hvr_seq,
           date('now') as hvr_cap_tstamp,
#      endif
#      ifdef _SQLSERVER
           cast(1 as binary(8)) as hvr_seq,
           ' ' as hvr_cap_tstamp,
#      endif
           user as hvr_cap_user, dm01_order.prod_id, dm01_order.ord_id,
           dm01_order.cust_name, dm01_order.cust_addr, dm01_product.prod_price,
           dm01_product.prod_descrip
   from    dm01_order, dm01_product, hvr_toghvr_demo01
   where   dm01_order.prod_id    =     dm01_product.prod_id
   and     dm01_order.order_date >=    hvr_toghvr_demo01.cap_begin_prev
   and     dm01_order.order_date <     hvr_toghvr_demo01.cap_begin
_SQL_END
_SQL_BEGIN
create view dm01_order__c1 as select * from dm01_order__c0
_SQL_END
#    endif
#  endif
#endif
```



- If long data types are needed (such as Oracle **clob** or SQL Server **text**) then these should be excluded from the capture view but still registered in the HVR catalogs; the HVR capture job will then do a **-58.7( )-**
-

DbSequence

Contents

Description

Action **DbSequence** allows database sequences to be replicated.

If a single **DbSequence** action is defined without any parameters for the entire channel (i.e. location group '*') then operations on all database sequences in the capture location(s) which are owned by the current schema will be replicated to all integrate locations. This means that if a **nextval** is done on the capture database then after replication a **nextval** on the target database is guaranteed to return a higher number. Note that however that if database sequence 'caching' is enabled in the DBMS, then this **nextval** on the target database could display a 'jump'.

SQL statement **create sequence** is also replicated, but **drop sequence** is not replicated.

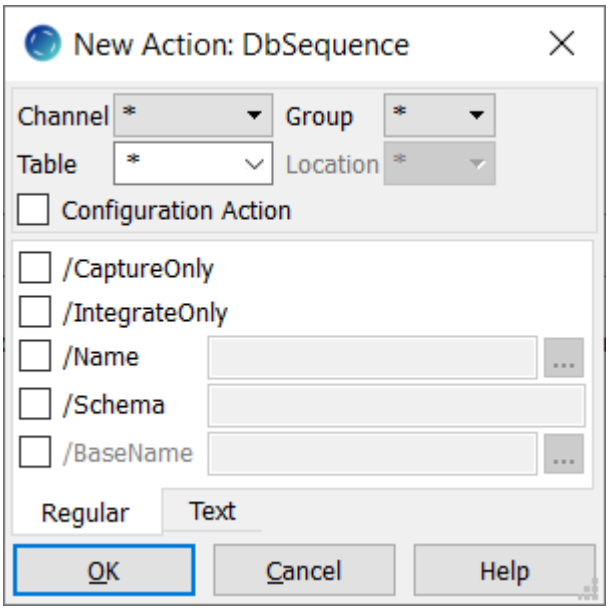
Commands **Hvrcompare** and **Hvrrefresh** also affect any database sequences matched by action **DbSequence**, unless option **–Q** is defined. Database sequences which are only in the 'write' database are ignored.

⚠

1. This action cannot be used with **Integrate /Burst**.
2. This action is only supported for certain DBMSs, for more information see Replicate database sequences (using action DbSequence) in Capabilities.
3. Capture of database sequence requires log–based capture (**Capture**).
4. For an Oracle RAC cluster, sequences should be defined with parameter **ORDER** (default is **NOORDER**), unless the next values are only generated on one node.

Parameters

This section describes the parameters available for action **DbSequence**.



Parameter	Argument	Description
/CaptureOnly		Only capture database sequences, do not integrate them.
/IntegrateOnly		Only integrate database sequences, do not capture them.
/Name	*seq_name*	Name of database sequence. Only capture or integrate database sequence named *seq_name*. By default, this action affects all sequences.
/Schema	*db_schema*	Schema which owns database sequence. By default, this action only affects sequences owned by the current user name.
/BaseName	*seq_name*	Name of sequence in database, if this differs from the name used in HVR. This allows a single channel to capture multiple database sequences that have the same sequence name but different owners.

⚠ Replication can be defined for a specific sequence (**/Name**) or for all sequences in a schema (**/Schema** without **/Name**) or for all sequences owned by the current user (neither **/Name** nor **/Schema**).
To capture all sequences from multiple schemas it is not allowed to just define multiple **DbSequence** actions with **/Schema** but not **/Name**. Instead either define lots of **DbSequence** actions with both **/Schema** and **/Name** or use multiple capture locations or channels, each with its own **DbSequence /Schema** action.

Bidirectional Replication

Bidirectional replication of sequences causes problems because the sequence change will 'boomerang back'. This means that after the integrate job has changed the sequence, the HVR capture job will detect this change and send it back to the capture location. These boomerangs make it impossible to run capture and integrate jobs simultaneously. But it is possible to do bidirectional replication for a failover system; i.e. when replication is normally only running from A to B, but after a failover the replication will switch to run from B to A. Immediately after the switchover a single boomerang will be sent from B to A, but afterwards the system will consistent and stable again.

If bidirectional replication is defined, then HVR Refresh of database sequences will also cause a single 'boomerang' to be captured by the target database's capture job.

Session names cannot be used to control bidirectional replication of database sequences in the way that they work for changes to tables. For more information, see Managing Recapturing Using Session Names.

Replication of Sequence Attributes

Database sequence 'attributes' (such as minimum, maximum, increment, randomness and cycling) are not replicated by HVR. When HVR has to create a sequence, it uses default attributes and only the value is set accurately. This means that if a database sequence has non–default attributes, then sequence must be manually created (outside of HVR) on the target database with the same attributes as on the capture database. But once these attributes are set correctly, then HVR will preserve these attributes while replicating the **nextval** operations.

Environment

Contents

Description

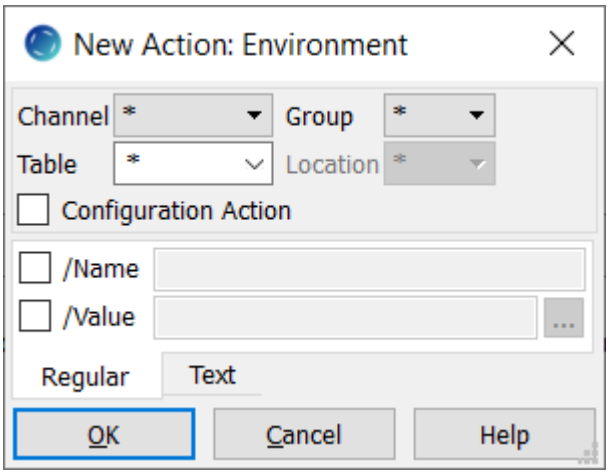
Action **Environment** sets an operating system environment variable for the HVR process which connects to the affected location. It also affects an agent called for this location.

Environment variables can also be in the environment of the HVR Scheduler, but these are only inherited by HVR processes that run locally on the hub machine; they are not exported to HVR child processes that are used for remote locations.

If this action is defined on a specific table, then it affects the entire job including data from other tables for that location.

Parameters

This section describes the parameters available for action **Environment**.



Parameter	Argument	Description
/Name	*env_var*	Name of the environment variable.
/Value	*path*	Value of the environment variable.

Examples

Variable	Description
HVR_COMPRESS_LEVEL	Controls amounts of replication for capture and integrate jobs. Value **s** disables compression, which will reduce CPU load. By default, compression is enabled. This variable must have the same value for all locations.

HVR_LOG_RELEASE_DIR **Ingres** **Oracle** **PostgreSQL**	Directory chosen by **Hvrlogrelease** for private copies of DBMS journal or archive files. By default, **Hvrlogrelease** writes these files into the DBMS tree (inside **$II_SYSTEM** or **$ORACLE_HOME**). In Oracle, if **Capture /ArchiveLogPath** is defined then the value set in this environment variable is ignored/overridden.
HVR_SORT_BYTE_LIMIT	Amount of memory to use before sorting large data volumes in temporary files. The default limit is **512Mb**.
HVR_SORT_COMPRESS	When set to value **1** the sorting of large amounts of data will be compressed on the fly to reduce disk room.
HVR_SORT_ROW_LIMIT	Number of rows to keep in memory before sorting large amounts of data using temporary files. The default limit is **10Mb**.

FileFormat

Contents

Description

Action **FileFormat** can be used on file locations (including HDFS and S3) and on Kafka locations.

For file location, it controls how HVR's read and write files. The default format for file locations is HVR's own XML format.

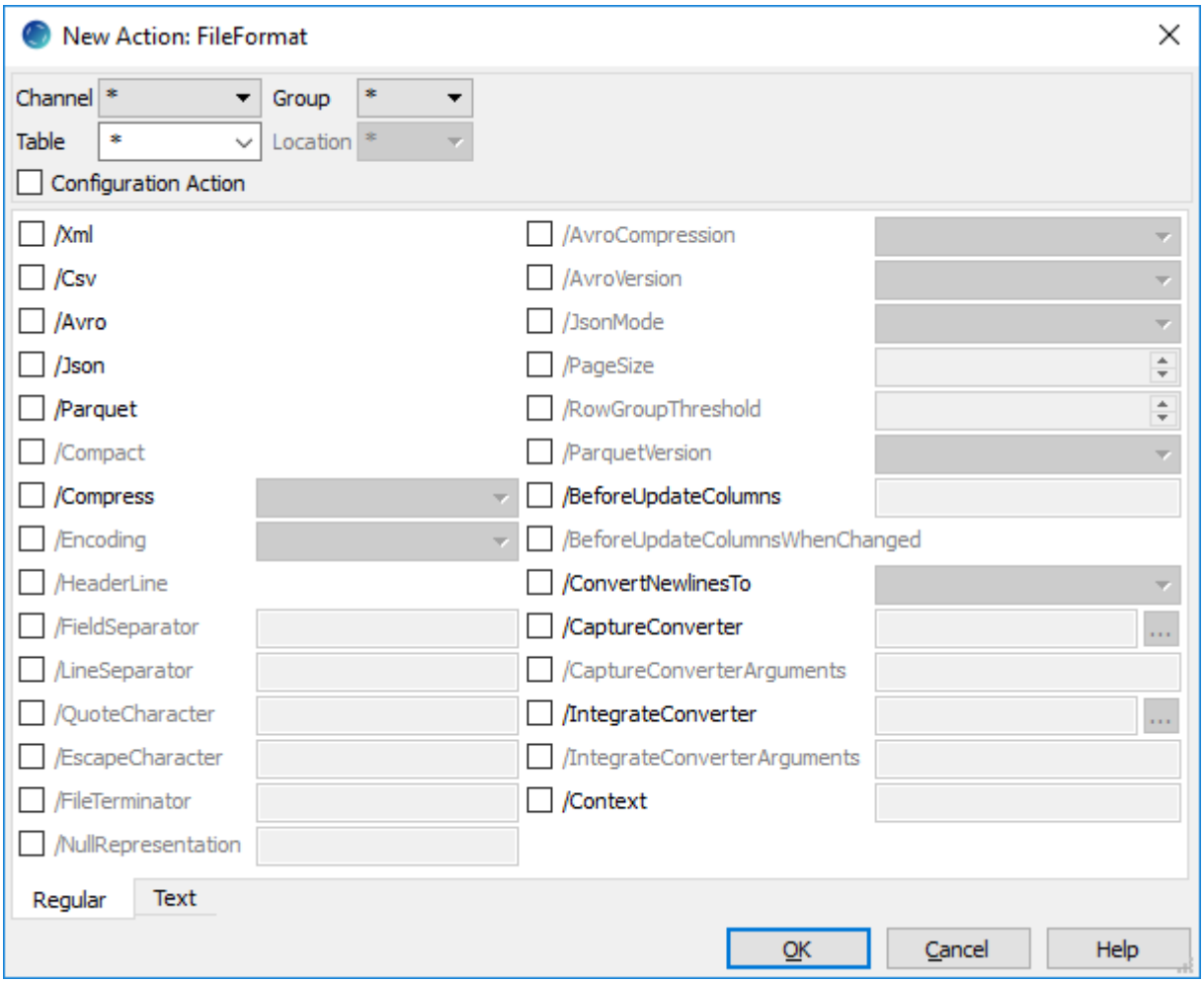
For Kafka, this action controls the format of each message. HVR's Kafka location sends messages in JSON format by default, unless the option **Schema Registry (Avro)** in Kafka location connection is used, in which case each message uses Kafka Connect's compact Avro-based format. Note that this is not a true Avro because each message would not be a valid Avro file (e.g. no file header). Rather, each message is a 'micro Avro', containing fragments of data encoded using Avro's data type serialization format. Both JSON (using mode **SCHEMA_PAYLOAD**, see parameter **/JsonMode** below) and the 'micro AVRO' format conform to Confluent's 'Kafka Connect' message format standard. The default Kafka message format can be overridden by parameter such as **/Xml**, **/Csv**, **/Avro**, **/Json** or **/Parquet**.

A custom format can be used using **/CaptureConverter** or **/IntegrateConverter**. Many parameters only have effect if the channel contains table information; for a 'blob file channel' the jobs do not need to understand the file format.

- If this action is defined on a specific table, then it affects all tables in the same location.
- Defining more than one file format (**Xml**, **Csv**, **Avro**, **Json** or **Parquet**) for the same file location using this action is not supported, i.e., defining different file formats for each table in the same location is not possible. For example, if one table has the file format defined as **/XML** then another table in the same location cannot have **/CSV** file format defined.

Parameters

This section describes the parameters available for action **FileFormat**.



Parameter	Argument	Description
/Xml		Read and write files as HVR's XML format. Default. This parameter is only for the channels with table information; not a 'blob file'.
/Csv		Read and write files as Comma Separated Values (CSV) format. This parameter is only for the channels with table information; not a'blob file'.
/Avro		Transforms the captured rows into Avro format during **Integrate**. An Avro file contains the schema defining data types in JSON and a compact binary representation of the data. See Apache Avro documentation for the detailed description of schema definition and data representation. Avro supports primitive and logical data types. The normal way to represent Avro file in human-readable format is converting it to JSON via Apache Avro tools. However, there is a drawback in representing decimal values using standard Avro-tools utilities. The **decimal** type in Avro is supported as a logical type and is defined in the Avro schema file as follows:

		<pre>{
		  "type": "bytes",
		  "logicalType": "decimal",
		  "precision": precision,
		  "scale": scale
		}</pre>
		• **Precision** is the total number of digits in the number.
		• **Scale** is the number of digits after the decimal point.
		
		The **decimal** logical type represents an arbitrary-precision signed decimal number of the form **unscaled × 10^{-scale}**. For example, value 1.01 with a precision of 3 and scale of 2, is represented as 101.
		
		Decimal values are encoded as a sequence of bytes in Avro. In their JSON representation, decimal values are displayed as an unreadable string instead of human-readable values.
		
		For example:
		
		A source table is defined as follows:
		
		<pre>create table Dec (c1 number(10,2) , c2 number(10,4)
		);</pre>
		wherein column c1 stores a decimal value with precision 10 and scale 2, and column c2 stores a decimal value with precision 10 and scale 4.
		
		If we insert values (1, 1) into **Dec** table and select them from the table, we expect to see (1, 1) as an output.
		
		But Avro format uses the specified scales and represents them in binary format as 100 (1.00) in column c1 and 10000 (1.0000) in column c2. According to the JSON specification, a binary array is encoded as a string.
		
		JSON will display these values as "d" (wherein "d" is 100 according to ASCII) and "'\x10" (wherein 10000 is 0x2710, and 0x27 is **'** according to the ASCII encoding).
		
		Formats like **/Parquet /ParquetVersion=v2** or **v3**, **/Json /JsonMode=SCHEMA_PAYLOAD** uses the same rules to encode decimal data types.
		
		When using Hive (Hive external table) to read Avro files, the **decimal** data type is displayed properly.

/Json		Transforms the captured rows into JSON format during **Integrate**. The content of the file depends on the value for parameter **/JsonMode**.
/Parquet		Transforms the captured rows into Parquet format during **Integrate**.
/Compact		Write compact XML tags like **<r>** & **<c>** instead of **<row>** and **<column>**. This parameter is enabled only if the parameter **/Xml** is selected.

/Compress	*algorithm*	HVR will compress files while writing them, and uncompress them while reading. Available options for *algorithm* are: • **GZIP** • **LZ4** The file suffix is ignored but when integrated, a suffix can be added to the files with action like **Integrate** **/RenameExpression="{hvr_cap_filename}.gz"**.
/Encoding	*encoding*	Encoding for reading or writing files. Available options for *encoding* are: • **US-ASCII** • **ISO-8859-1** • **ISO-8859-9** • **WINDOWS-1251** • **WINDOWS-1252** • **UTF-8** • **UTF-16LE** • **UTF-16BE**
/HeaderLine		First line of CSV file contains column names.
/FieldSeparator	*str_esc*	Field separator for CSV files. The **default** value for this parameter is comma (**,**). Note that only a single Unicode glyph is supported as a separator for this parameter. Examples: **, \x1f** or **\t**. This parameter is enabled only if the parameter **/Csv** is selected.
/LineSeparator	*str_esc*	Line separator for CSV files. The **default** value for this parameter is newline (**\n**). Examples: **;\n** or **\r\n** This parameter is enabled only if the parameter **/Csv** is selected.
/QuoteCharacter	*str_esc*	Character to quote a field with, if the fields contains separators. The **default** value for this parameter is double quotes (**"**). This parameter is enabled only if the parameter **/Csv** is selected.
/EscapeCharacter	*str_esc*	Character to escape the quote character with. The **default** value for this parameter is double quotes (**"**). This parameter is enabled only if the parameter **/Csv** is selected.
/FileTerminator	*str_esc*	File termination at end-of-file. Example: **EOF** or **\xff**. This parameter is enabled only if the parameter **/Csv** is selected.

/NullRepresentation	*str_esc*	String representation for column with NULL value. Note that Hive 'deserializers' recognize **\N** as NULL when reading a CSV file back as an SQL row, this can be configured by setting this parameter to **\\N**". Example: **\\N** This parameter is enabled only if the parameter **/Csv** is selected.
/AvroCompression	*codec*	Codec for Avro compression. Available option for *codec* is: • **Deflate**. This parameter is enabled only if the parameter **/Avro** is selected.
/AvroVersion	*version*	Version of Avro format. Available options for *version* are: • **v1_6**: supports only the following basic types: **Boolean**, **int** (32-bit size), **long** (64-bit size), **float**, **double**, **bytes**, and **string**. • **v1_7**: supports only the following basic types: **Boolean**, **int** (32-bit size), **long** (64-bit size), **float**, **double**, **bytes**, and **string**. • **v1_8** (**default**): supports the above mentioned basic types and the following logical types: **decimal**, **date**, **time** and **timestamp** (with micro and millisecond resolutions), and **duration**. This parameter is enabled only if the parameter **/Avro** is selected.
/JsonMode	*mode*	Style used to write row into JSON format. This parameter is enabled only if the parameter **/Json** is selected. Available options for *mode* are:

- **ROW_FRAGMENTS**: This format is compatible with Hive and BigQuery deserializers. Note that this option produces an illegal JSON file as soon as there is more than one row in the file.
  Example:

```
{ "c1":44, "c2":55 }
{ "c1":66, "c2":77 }
```

- **ROW_ARRAY**:
  Example:

```
[
{ "c1":44, "c2":55 },
{ "c1":66, "c2":77 }
]
```

- **TABLE_OBJECT** (**default** JSON mode for all location classes except for Kafka):
  Example:

```
{ "tab1" : [ { "c1":44, "c2":55 },
{ "c1":66, "c2":77 } ] }
```

- **TABLE_OBJECT_BSON**: This format is the same as **TABLE_OBJECT** , but in BSON format (binary). Note that a BSON file cannot be bigger than 2GB. This makes this format inapplicable for some tables (e.g. when LOB values are present).
- **TABLE_ARRAY**: This mode is useful if **/RenameExpression** does not contain a substitution which depends on the table name and when the location class is not Kafka.
  Example:

```
[
{ "tab1" : [{ "c1":44, "c2":55 },
{ "c1":66, "c2":77 }] },
{ "tab2" : [{ "c1":88, "c2":99 }] }
]
```

- **SCHEMA_PAYLOAD** (**default** JSON mode for location class Kafka): T his format is compatible with Apache Kafka Connect deserializers. Note that this option produces an illegal JSON file as soon as there is more than one row in the file.

```
{ "schema": {"type":"struct", "name":"tab1",
"fields": [{"name":"c1", "type":"int"}, {"name":"
c2", "type":"int"}]}, "payload": { "c1": 44, "c2":
55 }}
{ "schema": {"type":"struct", "name":"tab1",
"fields": [{"name":"c1", "type":"int"}, {"name":"
c2", "type":"int"}]}, "payload": { "c1": 66, "c2":
77 }}
```

/PageSize		Parquet page size in bytes. **Default** value is 1MB. This parameter is enabled only if the parameter **/Parquet** is selected.
/RowGroupThreshold		Maximum row group size in bytes for Parquet. This parameter is enabled only if the parameter **/Parquet** is selected.
/ParquetVersion **Since** v5.3.1/4	*version*	Category of data types to represent complex data into Parquet format. • **v1** : Supports only the basic data types - **boolean**, **int32**, **int64**, **int96**, **float**, **double**, **byte_array** to represent any data. The logical data types **decimal** and **date/time** types are not supported. However, **decimal** is encoded as **double**, and **date/time** types are encoded as **int96**. • **v2** (**default**): Supports all basic data types and one logical data type (**decimal**). The **date/time** types are encoded as **int96**. This is compatible with Hive, Impala, Spark, and Vertica. • **v3** : Supports basic data types and logical data types - **decimal**, **date**, **time_millis**, **time_micros**, **timestamp_millis**, **timestamp_micros**. For more information about parquet data types, refer to Parquet Documentation. This parameter is enabled only if the parameter **/Parquet** is selected.

/BeforeUpdateColumns
 	
Kafka
	prefix	By default, the update operation is captured as 2 rows: 'before' and 'after' versions of a row. During the update operation, this option merges these two rows into one and adds user-defined *prefix* to all the columns of the 'before' version.
		
		For example:
		
		`insert into t values (1, 1, 1)`
		`update t set c2=2 where c1=1`
		
		The output will be as follows:
		
		`{"c1": 1, "c2": 2, "c3": 1, "old&c1": 1, "old&c2": 1, "old&c3": 1}`
		
		where 'old&' is a user-defined *prefix*

/BeforeUpdateWhenChanged
 	
Kafka
		Adds the user-defined *prefix* (**/BeforeUpdateColumns**) only to columns, in which values were updated. This is supported only for JSON and XML formats.
		
		This option can be applied only when **/BeforeUpdateColumns** is selected.
		
		For example:
		
		`insert into t values (1, 1, 1)`
		`update t set c2 =2 where c1=1`
		
		The output will be as follows:
		
		`{"c1": 1, "c2": 2, "c3": 1, "old&c2": 1}`
		
		where 'old&' is a *prefix* defined for the **/BeforeUpdateColumns** option.

/ConvertNewlinesTo	*style*	Write files with **UNIX** or **DOS** style newlines.
/CaptureConverter	*path*	Run files through converter before reading. Value *path* can be a script or an executable. Scripts can be shell scripts on Unix and batch scripts on Windows or can be files beginning with a 'magic line' containing the interpreter for the script e.g. **#!perl**. A converter command should read from its **stdin** and write to **stdout**. Argument *path* can be an absolute or a relative pathname. If a relative pathname is supplied the command should be located in **$HVR_HOME/lib/transform**.
/CaptureConverterArguments	*userarg*	Arguments to the capture converter.
/IntegrateConverter	*path*	Run files through converter before writing. Value *path* can be a script or an executable. Scripts can be shell scripts on Unix and batch scripts on Windows or can be files beginning with a 'magic line' containing the interpreter for the script e.g. **#!perl**. A converter command should read from its **stdin** and write to **stdout**. Argument *path* can be an absolute or a relative pathname. If a relative pathname is supplied the command should be located in **$HVR_HOME/lib/transform**.
/IntegrateConverterArguments	*userarg*	Arguments to the integrate converter program.
/Context	*context*	Action only applies if **Refresh** or **Compare** context matches.

HVR's XML Format

The XML schema used by HVR can be found in **$HVR_HOME/lib/hvr.dtd**.

Simple Example

Following is a simple example of an XML file containing changes which were replicated from a database location.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<hvr version="1.0">
<table name="dm01_product">
<row>
<column name="prod_id">1</column>
<column name="prod_price">30</column>
<column name="prod_descrip">DVD</column>
</row>
<row>
<column name="prod_id">2</column>
<column name="prod_price">300</column>
<column name="prod_descrip" is_null="true"/>
</row>
</table>
</hvr>
```

Extended Example

Following is an extended example of HVR's XML.

- The following Oracle tables are defined;

  ```
  create table mytab (aa number not null, bb date, constraint mytab_pk primary key
  (aa));
  create table tabx (a number not null, b varchar2(10) not null, c blob, constraint
  tabx_pk primary key (a, b));
  ```

- Switch to a different user to create new table with same name.

  ```
  create table tabx (c1 number, c2 char(5), constraint tabx_pk primary key (c1));
  ```

- An HVR channel is then built, using **Capture**, **Integrate** and **ColumnProperties** **/Name=hvr_op_val /Extra /IntegrateExpression={hvr_op} /TimeKey** and then changes are applied to the source database using the following SQL statements;

  ```
  insert into tabx (a,b,c)         -- Note: column c contains binary/hex data
                                  v a l u e s         ( 1 ,         ' h e l l o ' ,
  '746f206265206f72206e6f7420746f2062652c007468617420697320746865');
  insert into tabx (a,b,c)
      values (2, '<world>', '7175657374696f6e');
  insert into mytab (aa, bb) values (33, sysdate);
  update tabx set c=null where a=1;
  commit;
  update mytab set aa=5555 where aa=33;  -- Note: key update
  delete from tabx;              -- Note: deletes two rows
  insert into user2.tabx (c1, c2)     -- Note: different tables share same name
      values (77, 'seven');
  commit;
  ```

The above SQL statements would be represented by the following XML output. Note that action **ColumnProperties /Name=hvr_op_val /Extra /IntegrateExpression={hvr_op} /TimeKey** causes an extra column to be shown named **hvr_op_val** which says the operation type (**0**=delete, **1**=insert, **2**=update, **3**=before key update, **4**=before key update). If this parameter were not defined that only insert and updates would be shown; other changes (e.g. deletes and 'before updates') would be from the XML output.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<hvr version="1.0">
 <table name="tabx">
  <row>
   <column name="hvr_op_val">1</column>
```

```
                                -- Note: Hvr_op=1 means insert -->

                                    <-- Note: Binary shown in hex       -->
                                r hash (#) is comment -->
                                74 2074 6f20  # to be or not to
                                7320 7468 65  # be,.that is the


                          val">1</column>
                          column>
                          orld></column>  <-- Note: Standard XML escapes used -->
                          ormat="hex">
                          6f6e     # question


                        <-- Note: Table tag switches current table -->

                  ="mytab">

                  name="hvr_op_val">1</column>
                  mn name="aa">33</column>
                  umn name="bb">2012-09-17 17:32:27</column> <-- Note: HVRs own date format -->
                  ow>
                table>

 <table name="tabx">
  <row>
   <column name="hvr_op_val">4</column> <-- Note: Hvr_op=4 means non-key update before -->
   <column name="a">1</column>
   <column name="b">hello</column>
  </row>
  <row>          <-- Note: No table tag because no table switch -->
   <column name="hvr_op_val">2</column> <-- Note: Hvr_op=2 means update-after -->
   <column name="a">1</column>
   <column name="b">hello</column>
   <column name="c" is_null="true"/>  <-- Note: Nulls shown in this way -->
  </row>
 </table>

 <table name="mytab">
  <row>
   <column name="hvr_op_val">3</column> <-- Note: Hvr_op=4 means key update-before -->
   <column name="aa">33</column>
  </row>
  <row>
   <column name="hvr_op_val">2</column>
   <column name="aa">5555</column>
  </row>
 </table>

 <table name="tabx">
  <row>
   <column name="hvr_op_val">0</column> <-- Note: Hvr_op=0 means delete -->
   <column name="a">1</column>
   <column name="b">hello</column>
   <column name="c" is_null="true"/>
  </row>
  <row>
   <column name="hvr_op_val">0</column> <-- Note: One SQL statement generated 2 rows -->
   <column name="a">2</column>
   <column name="b"><world></column>
   <column name="c" format="hex">
    7175 6573 7469 6f6e     # question
   </column>
  </row>
 </table>

 <table name="tabx1"> <-- Note: Name used here is channels name for table.   -->
<-- Note: This may differ from actual table 'base name' -->
  <row>
   <column name="hvr_op">1</column>
   <column name="c1">77</column>
   <column name="c2">seven</column>
  </row>
```

```
  </table>
 </hvr>   <-- Note: No more changes in replication cycle -->
```

Capture and Integrate Converters

Environment

A command specified with **/CaptureConverter** or **/IntegrateConverter** should read from its **stdin** and write the converted bytes to **stdout**. If the command encounters a problem, it should write an error to **stderr** and return with exit code **1**, which will cause the replication jobs to fail. The transform command is called with multiple arguments, which should be defined with **/CaptureConverterArguments** or **/IntegrateConverterArguments**.

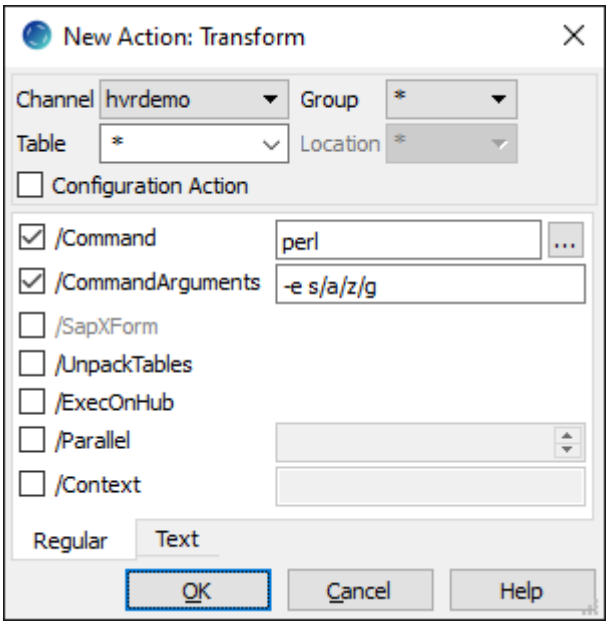
A converter command inherits the environment from its parent process. On the hub, the parent of the parent process is the HVR Scheduler. On a remote Unix machine, it is the **inetd** daemon. On a remote Windows machine it is the HVR Remote Listener service. Differences with the environment process are as follows:

- Environment variables **$HVR_CHN_NAME** and **$HVR_LOC_NAME** are set.
- Environment variable **$HVR_TRANSFORM_MODE** is set to either value **cap**, **integ**, **cmp**, **refr_read** or **refr_write** .
- Environment variable **$HVR_CONTEXTS** is defined with a comma–separated list of contexts defined with HVR Refresh or Compare (option **–C***ctx*).
- Environment variables **$HVR_VAR**_*XXX* are defined for each context variable supplied to HVR Refresh or Compare (option **–V***xxx*=*val*).
- For file locations variables **$HVR_FILE_LOC** and **$HVR_LOC_STATEDIR** are set to the file location's top and state directory respectively.
- For an integrate converter for 'blob' file channel without table information and for all capture converters, environment variables **$HVR_CAP_LOC**, **$HVR_CAP_TSTAMP**, **$HVR_CAP_FILENAME** and **$HVR_CAP_SUBDIRS** are set with details about the current file.
- Environment variable **$HVR_FILE_PROPERTIES** contains a colon–separated *name*=*value* list of other file properties. This includes values set by 'named patterns' (see **Capture /Pattern**).
  If a channel contains tables: Environment variable **$HVR_TBL_NAMES** is set to a colon–separated list of tables for which the job is replicating or refreshing (for example **HVR_TBL_NAMES**=**tbl1:tbl2:tbl3**). Also variable **$HVR_BASE_NAMES** is set to a colon–separated list of table 'base names', which are prefixed by a schema name if **/Schema** is defined (for example **HVR_BASE_NAMES**=**base1:sch2.base2:base3**). For modes **cap_end** and **integ_end** these variables are restricted to only the tables actually processed. Environment variables **$HVR_TBL_KEYS** and **$HVR_TBL_KEYS_BASE** are colon–separated lists of keys for each table specified in **$HVR_TBL_NAMES** (e.g. **k1**,**k2:k:k3**,**k4**). The column list is specified in **$HVR_COL_NAMES** and **$HVR_COL_NAMES_BASE**. Any variable defined by action **Environment** is also set in the converter's environment.
- The current working directory is **$HVR_TMP**, or **$HVR_CONFIG/tmp** if this is not defined.
- **stdin** is redirected to a socket (HVR writes the original file contents into this), whereas **stdout** and **stderr** are redirected to separate temporary files. HVR replaces the contents of the original file with the bytes that the converter writes to its **stdout**. Anything that the transform writes to its **stderr** is printed in the job's log file on the hub machine.

The output of a capture converter must conform the format implied by other parameters of this **FileFormat** action. Therefore if **/Csv** is not defined then the command should be XML.

Examples

A simple example is **FileFormat /IntegrateConverter=perl /IntegrateConverterArguments="-e s/a/z/g"**. This will replace all occurrences of letter **a** with **z**.



Directory **$HVR_HOME/lib/transform** contains other examples of command transforms written in Perl. Converter **hvrcsv2xml.pl** maps CSV files (Comma Separated Values) to HVR's XML.

Converter **hvrxml2csv.pl** maps HVR's XML back to CSV format. And **hvrfile2column.pl** maps the contents of a file into a HVR compatible XML file; the output is a single record/row.

Integrate

Contents

Description

Action **Integrate** instructs HVR to integrate changes into a database table or file location. Various parameters are available to tune the integration functionality and performance.

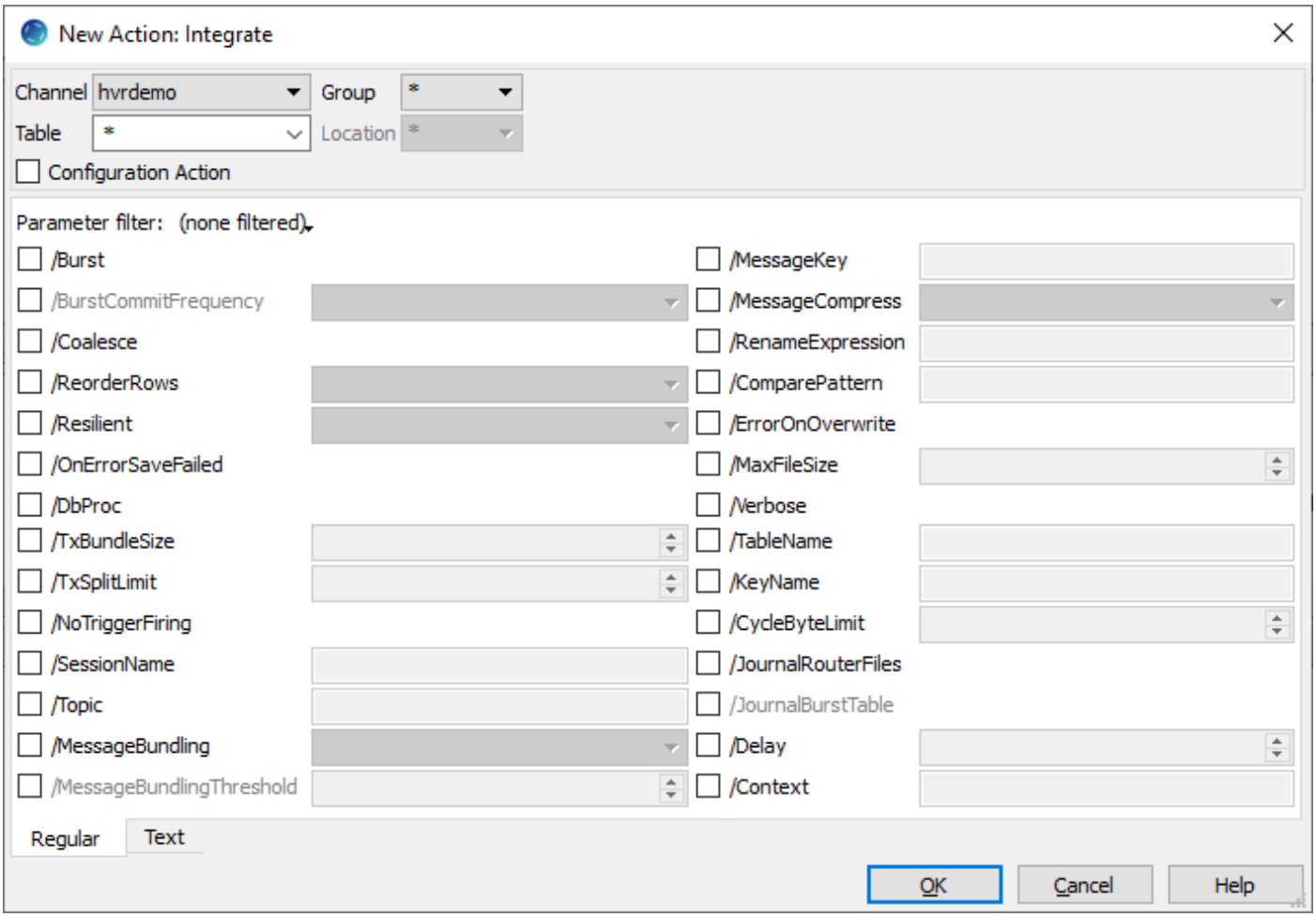
If integration is done on file location in channel with table information then any changes are integrated as records in either XML, CSV, AVRO or Parquet format. For details see action **FileFormat**

Alternatively, a channel can contain only file locations and no table information. In this case, each file captured is treated as a 'blob' and is replicated to the integrate file locations without HVR recognizing its format. If such a 'blob' file channel is defined with only actions **Capture** and **5702553** (no parameters) then all files in the capture location's directory (including files in subdirectories) are replicated to the integrate location's directory. The original files are not touched or deleted, and in the target directory, the original file names and subdirectories are preserved. New and changed files are replicated, but empty subdirectories and file deletions are not replicated.

If a channel is integrating changes into Salesforce, then the Salesforce 'API names' for tables and columns (case–sensitive) must match the 'base names' in the HVR channel. This can be done by defining **TableProperties** **/BaseName** actions on each of the tables and **ColumnProperties** **/BaseName** actions on each column.

Parameters

This section describes the parameters available for action **Integrate**. By default, only the supported parameters available for the selected location class are displayed in the **Integrate** window.



Parameter	Argument	Description

/Burst		Integrate changes into a target table using Burst algorithm. All changes for the cycle are first sorted and coalesced, so that only a single change remains for each row in the target table (see parameter **/Coalesce**). These changes are then bulk loaded into 'burst tables' named *tbl*_ _**b**. Finally, a single set wise SQL statement is done for each operation type (insert, update and delete). The end result is the same as normal integration (called continuous integration) but the order in which the changes are applied is completely different from the order in which they occurred on the capture machine. For example, all changes are done for one table before the next table. This is normally not visible to other users because the burst is done as a single transaction, unless parameter **/BurstCommitFrequency** is used. If database triggers are defined on the target tables, then they will be fired in the wrong order. This parameter cannot be used if the channel contains tables with foreign key constraints. If this parameter is defined for any table, then it affects all tables integrated to that location. During **/Burst** integrate, for some databases HVR 'streams' data into target databases straight over the network into a bulk loading interface specific for each DBMS (e.g. direct-path-load in Oracle) without storing the data on a disk. For other DBMSs, HVR puts data into a temporary directory ('staging file') before loading data into a target database. For more information about staging, see section "Burst Integrate and Bulk Refresh" in the respective location class requirements page. **/Burst** is required for online analytical processing (OLAP) databases, such as Greenpulm, Teradata, Snowflake, Redshift, for better performance during integration. This parameter cannot be used for file locations. A similar effect (reduce changes down to one per row) can be defined with parameter **/ReorderRows** =**SORT_COALESCE**. This parameter cannot be used with **DbSequence**.
/BurstCommitFrequency	*freq*	Frequency for committing burst set wise SQL statements. Available options for *freq* are: • **CYCLE** (**default**): All changes for the integrate job cycle are committed in a single transaction. If this parameter is defined for any table then it affects all tables integrated to that location. • **TABLE**: All changes for a table (the set wise **delete**, **update** and **insert** statements) are committed in a single transaction. • **STATEMENT**: A commit is done after each set wise SQL statement.
/Coalesce		Causes coalescing of multiple operations on the same row into a single operation. For example, an insert and an update can be replaced by a single insert; five updates can be replaced by one update, or an insert and a delete of a row can be filtered out altogether. The disadvantage of not replicating these intermediate values is that some consistency constraints may be violated on the target database. Parameter **/Burst** performs a sequence of operations including coalescing. Therefore this parameter should not be used with **/Burst**.

/ReorderRows	*mode*	Control order in which changes are written to files. If the target file-name depends on the table name (for example parameter **/RenameExpression** contains substitution **{hvr_tbl_name}**) and if the change-stream fluctuates between changes for different tables; then keeping the original order will cause HVR to create lots of small files (a few rows in a file for **tab1**, then a row for **tab2**, then another file for **tab1** again). This is because HVR does not reopen files after it has closed them. Reordering rows during integration will avoid these 'micro-files'. Available options for *mode* are: • **NONE** (**default**, if **/RenameExpression** does not contain a substitution which depends on the table name, which is common for XML format): Wr ite all rows in their original order, even if it means creating micro-files per table. • **BATCH_BY_TABLE** (**default** if **/RenameExpression** does contain a substitution which depends on the table name, as is common for CSV and AVRO format): Reorder interleaved rows of different tables in batches, with best effort. This mode both prevents micro-files and avoids a full-sort, so performance is good and resource usage is minimal. If there is lots of data then multiple (larger) files can be created for a single table. • **ORDER_BY_TABLE**: Reorder rows of different tables so there is only 1 file created per table. This value makes HVR buffer all changes during the integrate cycle before it writes them. • **SORT_COALESCE**: Sort and coalesce all changes for the cycle, so that only a single change remains for each row in the target file. This is equivalent to **/Burst** for database locations. A side effect of this value is that key updates (**hvr_op 3** then **2**) are replaced with a delete and insert (**hvr_op 0** then **1**) and a 'before' value for a regular update (**hvr_op 4**) is not written. A column can be added using **ColumnProperties** **/Extra** which contains the type of operation. This can use either **/IntegrateExpression**=**{hvr_op}** or multiple actions with **/ExpressionScope**=*scope* with **/IntegrateExpression**=*const*. Adding such a column does not change which operations (e.g. insert, update, delete etc..) are actually written to files. This depends instead on whether actions **/SoftDelete** or **/TimeKey** are defined. Inserts (**hvr_op** value **1**) and 'after updates' (**hvr_op 2**) are are always written. Deletes (**hvr_op 0**), 'before key updates' (**hvr_op 3**) and 'truncates' (**hvr_op 5**) are written if either **/SoftDelete** or **/TimeKey** is defined. A 'before non-key updates' (**hvr_op 4**) is only written if **/TimeKey** is defined.
/Resilient	*mode*	Resilient integration of inserts, updates and deletes. This modifies the behavior of integration if a change cannot be integrated normally. If a row already exists then an insert is converted to an update, an update or a non-existent row is converted to an insert, and a delete of a non-existent row is discarded. Existence is checked using the replication key known to HVR (rather than checking the actual indexes or constraints on the target table). Resilience is a simple way to improve replication robustness but the disadvantage is that consistency problems can go undetected. Value *mode* controls whether an error message is written to when this occurs. Available options for *mode* are: • **SILENT** • **SILENT_DELETES** • **WARNING**

/OnErrorSaveFailed		On integrate error, write the failed change into 'fail table' *tbl*__**f** and then continue integrating other changes. Changes written into the fail table can be retried afterwards (see command **Hvrretryfailed**). If this parameter is not defined, the **default** behavior if an integrate error occurs is to write a fatal error and to stop the job. If certain errors occur, then the integrate will no longer fail. Instead, the current file's data will be 'saved' in the file location's state directory, and the integrate job will write a warning and continue processing other replicated files. The file integration can be reattempted (see command **Hvrretryfailed**). Note that this behavior affects only certain errors, for example, if a target file cannot be changed because someone has it open for writing. Other error types (e.g. disk full or network errors) are still fatal. They will just cause the integrate job to fail. If data is being replicated from database locations and this parameter is defined for any table, then it affects all tables integrated to that location. This parameter cannot be used with parameter **/Burst**. For Salesforce, this parameter causes failed rows/files to be written to a fail directory (**$HVR_CONFIG/work/***hub***/***chn***/***loc***/sf**) and an error message to be written which describes how the rows can be retried.
/DbProc		Apply database changes by calling integrate database procedures instead of using direct SQL statements (insert, update and delete). The database procedures are created by **hvrinit** and called *tbl*__**ii**, *tbl*__**iu**, *tbl*__**id**. This parameter cannot be used on tables with long column data types. **/DbProc** cannot be used on tables with long data types, for example **long varchar** or **blob**.
/TxBundleSize	*int*	Bundle small transactions together for improved performance. The **default** transaction bundle size is 100. For example, if the bundle size is 10 and there were 5 transactions with 3 changes each, then the first 3 transactions would be grouped into a transaction with 9 changes and the others would be grouped into a transaction with 6 changes. Transaction bundling does not split transactions. If this parameter is defined for any table, then it affects all tables integrated to that location. **/TxBundleSize** can only be used in combination with the continuous integration mode (without **/Burst**) for performance tuning. This parameter is disabled when **/Burst** is selected.
/TxSplitLimit	*int*	Split very large transactions to limit resource usage. The **default** is 0, which means transactions are never split. For example, if a transaction on the master database affected 10 million rows and the remote databases has a small rollback segment then if the split limit was set to 1000000 the original transaction would split into 10 transactions of 1 million changes each. If this parameter is defined for any table, then it affects all tables integrated to that location.

/NoTriggerFiring		Enable or disable database triggers during integrate. This parameter is supported only for certain location classes. For the list of supported location classes, see Disable/enable database triggers during integrate (**/NoTriggerFiring**) in Capabilities. For Ingres, this parameter disables the firing of all database rules during integration. This is done by performing SQL statement **set norules** at connection startup. For SQL Server, this parameter disables the firing of database triggers, foreign key constraints and check constraints during integration if those objects were defined with **not for replication**. This is done by connecting to the database with the SQL Server **Replication** connection capability. A disadvantage of this connection type is that the database connection string must have form *host,port* instead of form *\\host\instance*. This *port* needs to be configured in the **Network Configuration** section of the **SQL Server Configuration Manager**. Another limitation is that encryption of the ODBC connection is not supported if this parameter is used for SQL Server. For Oracle and SQL Server, **Hvrrefresh** will automatically disable triggers on target tables before the refresh and re-enable them afterwards, unless option **-q** is defined. Other ways to control trigger firing are described in Managing Recapturing Using Session Names.
/SessionName	*sess_name*	Integrate changes with specific session name. The **default** session name is **hvr_integrate**. Capture triggers/rules check the session name to avoid recapturing changes during bidirectional replication. For a description of recapturing and session names see Managing Recapturing Using Session Names and Capture. If this parameter is defined for any table, then it affects all tables integrated to that location.
/Topic **Kafka**	*expression*	Name of the Kafka topic. You can use strings/text or expressions as Kafka topic name. Following are the expressions to substitute capture location or table or schema name as topic name: • **{hvr_cap_loc}** - for capture location name. • **{hvr_tbl_name}** - for current table name. This is only allowed if the channel is defined with tables. • **{hvr_schema}**- for schema name of the table. This is only allowed if the channel is defined with tables and this can only be used when an explicit **TableProperties /Schema**=*my_schema* is defined for these tables on the target file location. If this parameter is not defined, the messages are sent to the location's **Default Topic** field. The default topic field may also contain the above expressions. The Kafka topics used should either exist already in the Kafka broker or it should be configured to auto-create Kafka topic when HVR sends a message.

/MessageBundling **Kafka**	*mode*	Number of messages written (bundled) into single Kafka message. Regardless of the file format chosen, each Kafka message contains one row by default. Available options for *mode* are: • **ROW** (**default**): Each Kafka message contains a single row; this mode does not support bundling of multiple rows into a single message. Note that this mode causes a key-update to be sent as multiple Kafka messages (first a 'before update' with **hvr_op 3**, and then an 'after update' with **hvr_op 2**). • **CHANGE**: Each Kafka message is a bundle containing two rows (a 'before update' and an 'after update' row) whereas messages for other changes (e.g. insert and delete) contain just one row. During refresh there is no concept of changes, so each row is put into a single message. Therefore in that situation, this behavior is the same as mode **ROW**. • **TRANSACTION**: During replication, each message contains all rows in the original captured transaction. An exception is if the message size looks like it will exceed the bundling threshold (see parameter **/MessageBundlingThreshold**). During refresh, all changes are treated as if they are from a single capture transaction so this mode behaves the same as bundling mode **THRESHOLD**. • **THRESHOLD**: Each Kafka message is bundled with rows until it exceeds the message bundling threshold (see parameter **/MessageBundlingThreshold**). Note that Confluent's Kafka Connect only allows certain message formats and does not allow any message bundling, therefore **/MessageBundling** must either be undefined or set to **ROW**. Bundled messages simply consist of the contents of several single-row messages concatenated together. When the *mode* is **TRANSACTION** or **THRESHOLD** and if the name of the Kafka topic contains an expression such as **{hvr_tbl_name}** then the rows of different tables will not be bundled into the same message.
/MessageBundlingThreshold **Kafka**	*int*	The threshold for bundling rows in a Kafka message. The **default** value is 800,000 bytes. This parameter is applicable only for the message bundling modes **TRANSACTION** and **THRESHOLD**. For those bundling modes, rows continue to be bundled into the same message until after this threshold is exceeded. After that happens the message is sent and new rows are bundled into the next message. By default the maximum size of a Kafka message is 1,000,000 bytes; HVR will not send a message exceeding this size and will instead give a fatal error. You can change the maximum Kafka message size that HVR will send by defining **$HVR_KAFKA_MSG_MAX_BYTES** but ensure not to exceed the maximum message size configured in Kafka broker (settings **message.max.bytes**). If the message size exceeds this limit then message will be lost.
/MessageKey **Since** v5.6.0/0 **Kafka**	*expression*	Expression to generate user-defined key in a Kafka message. An expression can contain constant strings mixed with substitutions.

> When **/MessageKey** is not defined and **FileFormat /Json** or **FileFormat /Avro** (option **Schema Registry (Avro)** in Kafka location connection is not selected) is defined, the default value for **/MessageKey** is **{"schema":string, "payload": {hvr_key_hash}}**

Possible substitutions include:

- **{***colname* [*spec*]**}** is replaced/substituted with the value of current table's column *colname*. If the column has a date and time data type, the default format is **%[utc] %Y%m%d%H%M%S**, but this can be overridden using the timestamp substitution format specifier *spec.* For more information, see Timestamp Substitution Format Specifier.
- **{hvr_cap_loc}** is replaced with the name of the source location where the change occurred.
- **{hvr_cap_subdirs}** and **{hvr_cap_filename}** are also allowed if the file was captured from a file location.
- **{hvr_cap_subdirs_sharepoint}** and **{hvr_cap_filename_sharepoint}** can be used to rename files whose original name is not legal for SharePoint file systems. Each illegal character is mapped to a hex code, so a file called **sugar&spice** is renamed to **sugarx26spice**.
- **{hvr_cap_tstamp** [*spec*]**}** is replaced with the moment (time) that the change occurred in source location. The default format is **%[utc] %Y%m%d%H%M%S**, but this can be overridden using the timestamp substitution format specifier *spec*. For more information, see Timestamp Substitution Format Specifier. This substitution is supported only during **5702553** and not supported during **Refresh**.
- **{hvr_cap_user}** is replaced with the name of the user who made the change.
- **{hvr_chn_name}** is replaced with the name of the channel.
- **{hvr_integ_key}** is replaced with a 16 byte string value (hex characters) which is unique and continuously increasing for all rows integrated into the target location. The value is calculated using a high precision timestamp of the moment that the row is integrated. This means that if changes from the same source database are captured by different channels and delivered to the same target location then the order of this sequence will not reflect the original change order. This contrasts with substitution **{hvr_integ_seq}** where the order of the value matches the order of the change captured. Another consequence of using a (high precision) integrate timestamp is that if the same changes are sent again to the same target location (for example after option 'capture rewind' of **hvrinit**, or if a Kafka location's integrate job is restarted due to interruption) then the 're-sent' change will be assigned a new value. This means the target databases cannot rely on this value to detect 're-sent' data.
- **{hvr_integ_seq}** is replaced with a 36 byte string value (hex characters) which is unique and continuously increasing for a specific source location. If the channel has multiple source locations then this substitution must be combined with **{hvr_cap_loc}** to give a unique value for the entire target location. The value is derived from source database's DBMS logging sequence, e.g. the Oracle System Change Number (SCN).

- **{hvr_integ_tstamp** [*spec*]**}** is replaced with the moment (time) that the file was integrated into target location. The default format is **%[utc] %Y%m%d%H%M%SSSS**, but this can be overridden using the timestamp substitution format specifier *spec*. For more information, see Timestamp Substitution Format Specifier. If a format is supplied with *spec*, then HVR will throttle file integration to no faster than one file per second.
- **{hvr_key_hash}** is replaced with the hashed value of the key of current row This is only allowed if the channel is a Kafka location and **FileFormat /Json** or **/Avro** (without using Schema Registry) is defined.
- **{{hvr_key_names** *sep***}}** is replaced with the values of table's key columns concatenated together with separator *sep*.
- **{hvr_key_names** *sep***}** is replaced with the names of table's key columns concatenated together with separator *sep*.
- **{hvr_op}** is replaced with the HVR operation type. Values are **0** (delete), **1** (insert), **2** (after update), **3** (before key update), **4** (before non–key update) or **5** (truncate table). See also Extra Columns for Capture, Fail and History Tables.
- **{hvr_schema}** is replaced with the schema name of the table. This is only allowed if the channel is defined with the tables. This can only be used when a explicit **TableProperties /Schema**=*my_schema* is defined for these tables on the target file location.
- **{hvr_tbl_base_name}** is replaced with the base name of the current table. This is only allowed if the channel is defined with the tables.
- **{hvr_tbl_name}** is replaced with the name of the current table. This is only allowed if the channel is defined with the tables.
- **{hvr_tx_countdown}** is replaced with countdown of changes within transaction, for example if a transaction contains three changes the first change would have countdown value **3**, then **2**, then **1**. A value of **0** indicates that commit information is missing for that change.
- **{hvr_tx_scn}** is replaced with the source location's SCN (Oracle). This substitution can only be used if the channel has a single source location.
- **{hvr_tx_seq}** is replaced with a hex representation of the sequence number of transaction. For capture from Oracle this value can be mapped back to the SCN of the transaction's commit statement. Value **[**

		It is recommended to define **/Context** when using the substitutions **{hvr_var_*xxx*}**, **{hvr_slice_num}**, **{hvr_slice_total}**, or **{hvr_slice_value}**, so that it can be easily disabled or enabled. **{hvr_slice_num}**, **{hvr_slice_total}**, **{hvr_slice_value}** cannot be used if the one of the old slicing substitutions **{hvr_var_slice_condition}**, **{hvr_var_slice_num}**, **{hvr_var_slice_total}**, or **{hvr_var_slice_value}** is defined in the channel/table involved in the compare /refresh.
/MessageKeySerializer **Since** v5.6.5/10 **Kafka**	*format*	HVR will encode the generated Kafka message key in a string or Kafka Avro serialization *format*. Available options for the *format* are: • **KAFKA_AVRO** (**default** if option **Schema Registry (Avro)** in Kafka location connection is selected). The **KAFKA_AVRO** format is compatible with the Confluent Kafka Schema Registry requirements. • **STRING** (**default** if option **Schema Registry (Avro)** in Kafka location connection is not selected).
/MessageCompress **Since** v5.6.5/7 **Kafka**	*algorithm*	HVR will configure the Kafka transport protocol to compress message sets transmitted to Kafka broker using one of the available algorithms. The compression allows to decrease the network latency and save disk space on the Kafka broker. Each message set can contain more than one Kafka message and the size of the message set will be less than **$HVR_KAFKA_MSG_MAX_BYTES**. For more information, see Kafka Message Bundling and Size. Available options for the *algorithm* are: • **NONE** (recommended if **FileFormat** **/Compress** or **/AvroCompression** parameters are defined) • **LZ4** • **GZIP** (**default**) • **SNAPPY** **LZ4** compression is not supported on the Windows platform if Kafka broker version is less than 1.0.
/RenameExpression	*expression*	Expression to name new files. A rename expression can contain constant strings mixed with substitutions.

When **/RenameExpression** is not defined -

- The new files are named **{hvr_cap_subdirs}/{hvr_cap_filename}** if they are captured from another file location.
- The new files are named **{hvr_integ_tstamp}.xml** if they are for database changes and the channel is defined with tables.

Possible substitutions include:

- **{***colname* [*spec*]**}** is replaced/substituted with the value of current table's column *colname*. This can be used to partition data into subdirectories based on column values. For example, if you have a column named "states", your rename expression could include states='**{***states***}**', which creates a dynamic folder, for example, states='CA' if the row value was CA for the state. The [*spec*] is usually associated with date/datetime formats. The default format is **%[utc] %Y%m%d%H%M%S**, but this can be overridden using the timestamp substitution format specifier *spec* if the column has a date and time data type. For more information, see Timestamp Substitution Format Specifier.
  Example: /RenameExpression="**{hvr_tbl_name}**/state='**{***state***}**'/**{hvr_integ_tstamp}.csv**"
- **{hvr_cap_loc}** is replaced with the name of the source location where the change occurred.
- **{hvr_cap_subdirs}** and **{hvr_cap_filename}** are also allowed if the file was captured from a file location.
- **{hvr_cap_subdirs_sharepoint}** and **{hvr_cap_filename_sharepoint}** can be used to rename files whose original name is not legal for SharePoint file systems. Each illegal character is mapped to a hex code, so a file called **sugar&spice** is renamed to **sugarx26spice**.
- **{hvr_cap_tstamp** [*spec*]**}** is replaced with the moment (time) that the change occurred in source location. The default format is **%[utc] %Y%m%d%H%M%S**, but this can be overridden using the timestamp substitution format specifier *spec*. For more information, see Timestamp Substitution Format Specifier. This substitution is supported only during **5702553** and not supported during **Refresh**.
- **{hvr_cap_user}** is replaced with the name of the user who made the change.
- **{hvr_chn_name}** is replaced with the name of the channel.
- **{hvr_integ_key}** is replaced with a 16 byte string value (hex characters) which is unique and continuously increasing for all rows integrated into the target location. The value is calculated using a high precision timestamp of the moment that the row is integrated. This means that if changes from the same source database are captured by different channels and delivered to the same target location then the order of this sequence will not reflect the original change order. This contrasts with substitution **{hvr_integ_seq}** where the order of the value matches the order of the change captured. Another consequence of using a (high precision) integrate timestamp is that if the same changes are sent again to the same target location (for example after option 'capture rewind' of **hvrinit**, or if a Kafka location's integrate job is restarted due to interruption) then the 're-sent' change will be assigned a new value. This means the target databases cannot rely on this value to detect 're-sent' data.

- **{hvr_integ_seq}** is replaced with a 36 byte string value (hex characters) which is unique and continuously increasing for a specific source location. If the channel has multiple source locations then this substitution must be combined with **{hvr_cap_loc}** to give a unique value for the entire target location. The value is derived from source database's DBMS logging sequence, e.g. the Oracle System Change Number (SCN).
- **{hvr_integ_tstamp** [*spec*]**}** is replaced with the moment (time) that the file was integrated into target location. The default format is **%[utc] %Y%m%d%H%M%SSSS**, but this can be overridden using the timestamp substitution format specifier *spec*. For more information, see Timestamp Substitution Format Specifier. If a format is supplied with *spec* , then HVR will throttle file integration to no faster than one file per second.
- **{{hvr_key_names** *sep***}}** is replaced with the values of table's key columns concatenated together with separator *sep*.
- **{hvr_key_names** *sep***}** is replaced with the names of table's key columns concatenated together with separator *sep*. **Since** v5.5.5/8
- **{hvr_op}** is replaced with the HVR operation type. Values are **0** (delete), **1** (insert), **2** (after update), **3** (before key update), **4** (before non–key update) or **5** (truncate table). See also Extra Columns for Capture, Fail and History Tables.
- **{hvr_schema}** is replaced with the schema name of the table. This is only allowed if the channel is defined with the tables. This can only be used when a explicit **TableProperties /Schema**=*my_schema* is defined for these tables on the target file location.
- **{hvr_tbl_base_name}** is replaced with the base name of the current table. This is only allowed if the channel is defined with the tables. **Since** v5.6.0/0
- **{hvr_tbl_name}** is replaced with the name of the current table. This is only allowed if the channel is defined with the tables.
- **{hvr_tx_countdown}** is replaced with countdown of changes within transaction, for example if a transaction contains three changes the first change would have countdown value **3**, then **2**, then **1**. A value of **0** indicates that commit information is missing for that change.
- **{hvr_tx_scn}** is replaced with the source location's SCN (Oracle). This substitution can only be used if the source location database is Oracle. This substitution can only be used for ordering if the channel has a single source location.
- **{hvr_tx_seq}** is replaced with a hex representation of the sequence number of transaction. For capture from Oracle this value can be mapped back to the SCN of the transaction's commit statement. Value **[hvr_tx_seq, -hvr_tx_countdown]** is increasing and uniquely identifies each change. **{hvr_tx_seq}** gets value only if **Select Moment** (option **-M**) is selected while performing HVR Refresh.
- **{hvr_var**_*xxx***}** is replaced with value of 'context variable' *xxx*. The value of a context variable can be supplied using option **–V***xxx*=*val* to command **hvrrefresh** or **hvrcompare**.
- **{hvr_slice_num}**: is replaced with the current slice number if slicing is defined with **Count** (option **-S** *num*) in **hvrrefresh** or **hvrcompare**. **Since** v5.6.5/0
- **{hvr_slice_total}**: is replaced with the total number of slices if slicing is defined with **Count** (option **-S** *num*) in **hvrrefresh** or **hvrcompare**. **Since** v5.6.5/0
- **{hvr_slice_value}**: is replaced with the current slice value if slicing is defined with **Series** (option **-S** *val1*[**;***val2*]...) in **hvrrefresh** or **hvrcompare** . **Since** v5.6.5/0

		• If the file was captured with a 'named pattern' (see **Capture /Pattern**), then the string that matched the named pattern can be used as a substitution. So if a file was matched with **/Pattern**=**"{office}.txt"** then it could be renamed with expression **hello_{office}.data**. It is recommended to define **/Context** when using the substitutions **{hvr_var_*xxx*}**, **{hvr_slice_num}**, **{hvr_slice_total}**, or **{hvr_slice_value}**, so that it can be easily disabled or enabled. **{hvr_slice_num}**, **{hvr_slice_total}**, **{hvr_slice_value}** cannot be used if the one of the old slicing substitutions **{hvr_var_slice_condition}**, **{hvr_var_slice_num}**, **{hvr_var_slice_total}**, or **{hvr_var_slice_value}** is defined in the channel/table involved in the compare /refresh.
/ComparePattern **Since** v5.5.5/6	*patt*	Perform direct file compare. During compare, HVR reads and parses (deserializes) files directly from a file location instead of using HIVE external tables (even if it is configured for that location). While performing direct file compare, the files of each table are distributed to pre-readers. The file 'pre-read' subtasks generate intermediate files. The location for these intermediate files can be configured using **LocationProperties /IntermediateDirectory**. To configure the number of pre-read subtasks during compare use **hvrcompare** with option **-w**. HVR can parse only CSV or XML file formats and does not support Avro, Parquet or JSON. This parameter can only be used on file locations. Example: **{hvr_tbl_name}/**/*.csv** To perform direct file compare, option **Generate Compare Event** (**-e**) should be selected in command **hvrcompare**.
/ErrorOnOverwrite		Error if a new file has same name as an existing file. If data is being replicated from database locations and this parameter is defined for any table, then it affects all tables integrated to that location.
/MaxFileSize	*int*	The threshold for bundling rows in a file. This parameter cannot be used for 'blob' file channels which contain no table information and only replicated files as 'blobs'. The rows are bundled into the same file until after this threshold is exceeded. After that happens, the file is sent and HVR will start writing rows to a new file whose name is found by re-evaluating parameter **/RenameExpression** (or **{hvr_integ_tstamp}.xml** if that parameter is not specified). XML files written by HVR always contain at least one row, which means that specifying a number between 1 and 500 will cause each file to contain a single row. For efficiency reasons HVR's decision to start writing a new file depends on the XML length of the previous row, not the current row. This means that sometimes the actual file size may slightly exceed the value specified. If data is being replicated from database locations and this parameter is defined for any table, then it affects all tables integrated to that location.

/Verbose		The file integrate job will write extra information, including the name of each file which is replicated. Normally, the job only reports the number of files written. If data is being replicated from database locations and this parameter is defined for any table, then it affects all tables integrated to that location.
/TableName	*userarg*	API name of Salesforce table into which attachments should be uploaded. See section 5702553 below.
/KeyName	*userarg*	API name of key in Salesforce table for uploading attachments. See section 5702553 below.
/CycleByteLimit	*int*	Maximum amount of compressed router files to process per integrate cycle. The default **CycleByteLimit** is **10 Mb**. Since HVR 5.6.5/13, if **/Burst** parameter is defined, the default **CycleByteLimit** is **100 Mb**. Value **0** means unlimited, so the integrate job will process all available work in a single integrate cycle. If more than this amount of data is queued for an integrate job, then it will process the work in 'sub cycles'. The benefit of 'sub cycles' is that the integrate job won't last for hours or days. If the **/Burst** parameter is defined, then large integrate cycles could boost the integrate speed, but they may require more resources (memory for sorting and disk room in the burst tables *tbl*_ _**b**). If the supplied value is smaller than the size of the first transaction file in the **router** directory, then all transactions in that file will be processed. Transactions in a transaction file will never be split between cycles or sub-cycles. If this parameter is defined for any table, then it affects all tables integrated to that location.
/JournalRouterFiles		Move processed transaction files to journal directory **$HVR_CONFIG/jnl/***hub /chn/YYYYMMDD* on the hub machine. Normally an integrate job would just delete its processed transactions files. The journal files are compressed, but their contents can be viewed using command **hvrrouterview**. Old journal files can be purged with command **hvrmaint journal_keep_days**=*N*. If this parameter is defined for any table, then it affects all tables integrated to that location.
/Delay	*N*	Delay integration of changes for *N* seconds
/Context	*ctx*	Ignore action unless refresh/compare context *ctx* is enabled. The value should be the name of a context (a lowercase identifier). It can also have form **!**_ctx_, which means that the action is effective unless context *ctx* is enabled. One or more contexts can be enabled for HVR Compare or Refresh (on the command line with option **–C***ctx*). Defining an action which is only effective when a context is enabled can have different uses. For example, if action **Integrate /RenameExpression /Context=qqq** is defined, then the file will only be renamed if context **qqq** is enabled (option **-Cqqq**).

Columns Changed During Update

If an SQL update is done to one column of a source table, but other columns are not changed, then normally the **update** statement performed by HVR integrate will only change the column named in the original update. However, all columns will be overwritten if the change was captured with **Capture /NoBeforeUpdate**.

There are three exceptional situations where columns will never be overwritten by an update statement:

- If **ColumnProperties /NoUpdate** is defined;
- If the column has a LOB data type and was not change in the original update;
- If the column was not mentioned in the channel definition.

Controlling Trigger Firing

Sometimes during integration it is preferable for application triggers not to fire. This can be achieved by changing the triggers so that they check the integrate session (for example **where userenv('CLIENT_INFO') <>'hvr_integrate')**. For more information, see section Application triggering during integration in **Capture**.

For Ingres target databases, database rule firing can be prevented by specifying **Integrate /NoTriggerFiring** or with **hvrrefresh** option **–f**.

SharePoint Version History

HVR can replicate to and from a SharePoint/WebDAV location which has versioning enabled. By default, HVR's file integrate will delete the SharePoint file history, but the file history can be preserved if action **LocationProperties /StateDirectory** is used to configure a state directory (which is the then on the HVR machine, outside SharePoint). Defining a **/StateDirectory** outside SharePoint does not impact the 'atomicity' of file integrate, because this atomicity is already supplied by the WebDAV protocol.

Salesforce Attachment Integration

Attachments can be integrated into Salesforce.com by defining a 'blob' file channel (without table information) which captures from a file location and integrates into a Salesforce location. In this case, the API name of the Salesforce table containing the attachments can be defined either with **Integrate /TableName** or using 'named pattern' **{hvr_sf_tbl_name}** in the **Capture /Pattern** parameter. Likewise, the key column can be defined either with **Integrate /KeyName** or using 'named pattern' **{hvr_sf_key_name}**. The value for each key must be defined with 'named pattern' **{hvr_sf_key_value}**.

For example, if the photo of each employee is named *id***.jpg**, and these need to be loaded into a table named **Employee** with key column **EmpId**, then action **Capture /Pattern="{hvr_sf_key_value}.jpg"** should be used with action **Integrate /TableName="Employee" /KeyName="EmpId"**.

- All rows integrated into Salesforce are treated as 'upserts' (an update or an insert). Deletes cannot be integrated.
- Salesforce locations can only be used for replication jobs; **HVR Refresh** and **HVR Compare** are not supported.

Timestamp Substitution Format Specifier

Timestamp substitution format specifiers allows explicit control of the format applied when substituting a timestamp value. These specifiers can be used with **{hvr_cap_tstamp**[*spec*]**}**, **{hvr_integ_tstamp**[*spec*]**}**, and **{***colname* [*spec*]**}** if *colname* has timestamp data type. The components that can be used in a timestamp format specifier *spec* are:

Component	Description	Example
%a	Abbreviate weekday according to current locale.	Wed
%b	Abbreviate month name according to current locale.	Jan
%d	Day of month as a decimal number (01–31).	07

%H	Hour as number using a 24–hour clock (00–23).	17
%j	Day of year as a decimal number (001–366).	008
%m	Month as a decimal number (01 to 12).	04
%M	Minute as a decimal number (00 to 59).	58
%s	Seconds since epoch (1970–01–01 00:00:00 UTC).	1099928130
%S	Second (range 00 to 61).	40
%T	Time in 24–hour notation (%H:%M:%S).	17:58:40
%U	Week of year as decimal number, with Sunday as first day of week (00 – 53).	30
%V **Linux**	The ISO 8601 week number, range 01 to 53, where week 1 is the first week that has at least 4 days in the new year.	15
%w	Weekday as decimal number (0 – 6; Sunday is 0).	6
%W	Week of year as decimal number, with Monday as first day of week (00 – 53)	25
%y	Year without century.	14
%Y	Year including the century.	2014
%[localtime] **Since** v5.5.5/xx	Perform timestamp substitution using machine local time (not UTC). This component should be at the start of the specifier (e.g. **{{hvr_cap_tstamp %[localtime]%H}}**).	
%[utc] **Since** v5.5.5/xx	Perform timestamp substitution using UTC (not local time). This component should be at the start of the specifier (e.g. **{{hvr_cap_tstamp %[utc]%T}}**).	

LocationProperties

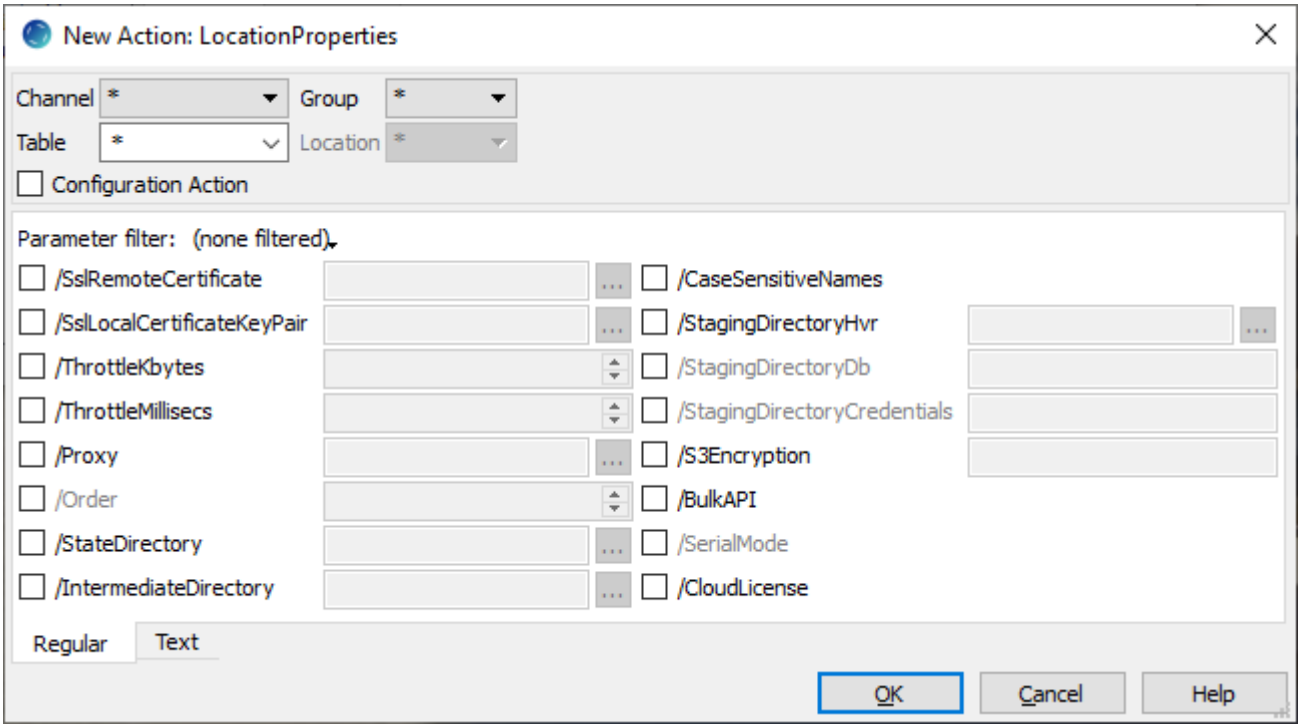
Contents

Description

Action **LocationProperties** defines properties of a remote location. This action has no affect other than that of its parameters. If this action is defined on a specific table, then it affects the entire job including data from other tables for that location.

Parameters

This section describes the parameters available for action **LocationProperties**. By default, only the supported parameters available for the selected location class are displayed in the **LocationProperties** window.



Parameter	Argument	Description
/SslRemoteCertificate	*pubcert*	Enable Secure Socket Layer (SSL) network encryption and check the identity of a remote location using *pubcert* file. Encryption relies on a public certificate which is held on the hub and remote location and a corresponding private key which is only held on the remote location. New pairs of private key and public certificate files can be generated by command **hvrsslgen** and are supplied to the remote **hvr** executable or **Hvrremotelistener** service with option **–K**. The argument *pubcert* points to the public certificate of the remote location which should be visible on the hub machine. It should either be an absolute pathname or a relative pathname (HVR then looks in directory **$HVR_HOME/lib**). A typical value is **hvr** which refers to a standard public certificate **$HVR_HOME/lib/cert/hvr.pub_cert.**

/SslLocalCertificateKeyPair	*pair*	Enable Secure Socket Layer (SSL) network encryption and allow the remote location to check the hub's identity by matching its copy of the hub's public certificate against *pair* which points to the hub machine's private key and public certificate pair. New pairs of private key and public certificate files can be generated by command **hvrsslgen** and are supplied to the remote **hvr** executable or **hvrremotelistener** service using an XML file containing the HVR access list. The argument *pair* points to the public certificate of the remote location which should be visible on the hub machine. It should either be an absolute pathname or a relative pathname (HVR then looks in directory **$HVR_HOME/lib**). It specifies two files: the names of these files are calculated by removing any extension from *pair* and then adding extensions **.pub_cert** and **.priv_key**. For example, value **hvr** refers to files **$HVR_HOME/lib/cert/hvr.pub_cert** and **$HVR_HOME/lib/cert/hvr.priv_key.**
/ThrottleKbytes	*int*	Restrain network bandwidth usage by grouping data sent to/from remote connection into packages, each containing *int* bytes, followed by a short sleep. The duration of the sleep is defined by **/ThrottleMillisecs**. Carefully setting these parameters will prevent HVR being an 'anti–social hog' of precious network bandwidth. This means it will not interfere with interactive end–users who share the link for example. For example if a network link can handle 64 KB/sec then a throttle of 32 KB with a 500 millisecond sleep will ensure HVR would be limited to no more than 50% bandwidth usage (when averaged–out over a one second interval).While using this parameter ensure to provide value in the dependent parameter **/ThrottleMillisecs**.
/ThrottleMillisecs	*int*	Restrict network bandwidth usage by sleeping *int* milliseconds between packets. While using this parameter ensure to provide value in the dependent parameter **/ThrottleKbytes** to define the package size.
/Proxy	*url*	URL of proxy server to connect to the specific location. Proxy servers are supported when connecting to HVR remote locations, for remote file access protocols (FTP, SFTP, WebDAV) and for Salesforce locations. The proxy server will be used for connections from the hub machine. If a remote HVR location is defined, then HVR will connect using its own protocol to the HVR remote machine and then via the proxy to the FTP /SFTP/WebDAV/Salesforce machine. Example, value *url* can be **hvr://***host name***:***port number*
/Order	*N*	Specify order of proxy chain from hub to location. Proxy chaining is only supported to HVR remote locations, not for file proxies (FTP, SFTP, WebDAV) or Salesforce proxies.
/StateDirectory	*path*	Directory for internal files used by HVR file replication state. By **default** these files are created in subdirectory **_hvr_state** which is created inside the file location top directory. If *path* is relative (e.g. **../work**), then the path used is relative to the file location's top directory. The state directory can either be defined to be a path inside the location's top directory or put outside this top directory. If the state directory is on the same file system as the file location's top directory, then HVR integrates file move operations will be 'atomic', so users will not be able to see the file partially written. Defining this parameter on a SharePoint/WebDAV integrate location ensures that the SharePoint version history is preserved.

/IntermediateDirectory **Since** v5.5.5/6	*dir*	Directory for storing 'intermediate files' that are generated during compare. I ntermediate files are generated by file 'pre-read' subtasks while performing d irect file compare. If this parameter is not defined, then by **default** the intermediate files are stored in *integratedir***/_hvr_intermediate** directory. The *integratedir* is the replication **Directory** defined in the **New Location** screen while creating a file location.
/CaseSensitiveNames		DBMS table names and column names are treated case sensitive by HVR. Normally HVR converts table names to lowercase and treats table and column names as case insensitive. Settings this parameter allows the replication of tables with mixed case names or tables whose names do not match the DBMS case convention. For example, normally an Oracle table name is held in uppercase internally (e.g. **MYTAB**), so this parameter is needed to replicate a table named **mytab** or **MyTab**. This parameter is supported only for certain location classes. For the list of location classes that supports this parameter, see Treat DBMS table names and columns case sensitive in Capabilities.
/StagingDirectoryHvr	*URL*	Directory for bulk load staging files. For certain databases (Hive ACID, Redshift, and Snowflake), HVR splits large amount data into multiple staging files, to optimize performance. This parameter is supported only for certain location classes. For the list of supported location classes, see Bulk load requires a staging area in Capabili ties. For Hive ACID, this should be an S3 or HDFS location.For Greenplum and HANA, this should be a local directory on the machine where HVR connects to the DBMS. For MySQL/MariaDb, when direct loading by the MySQL/MariaDB server option is used, this should be a directory local to the MySQL/MariaDB server which can be written to by the HVR user from the machine that HVR uses to connect to the DBMS. And when initial loading by the MySQL /MariaDB client option is used, this should be a local directory on the machine where HVR connects to the DBMS. For Redshift and Snowflake, this should be an S3 location.

/StagingDirectoryDb	*URL*	Location for the bulk load staging files visible from the database. This should point to the same files as **/StagingDirectoryHvr**. This parameter is enabled only if **/StagingDirectoryHvr** is selected. This parameter is supported only for certain location classes. For the list of supported location classes, see Bulk load requires a staging area in Capabilities. For Greenplum, this should either be a local directory on the Greenplum head-node or it should be a URL pointing to **/StagingDirectoryHvr**, for example a path starting with **gpfdist:** or **gpfdists:**. For HANA, this should be a local directory on the HANA machine which is configured for importing data by HANA. For Hive ACID, this should be the S3 or HDFS location that is used for **StagingDirectoryHvr**. For MySQL/MariaDb, when direct loading by the MySQL/MariaDB server option is used, this should be the directory from which the MySQL/MariaDB server should load the files. And when initial loading by the MySQL /MariaDB client option is used, this should be left empty. For Redshift and Snowflake, this should be the S3 location that is used for **StagingDirectoryHvr**.
/StagingDirectoryCredentials	*credentials*	Credentials to be used for S3 authentication and optional encryption during Hive ACID, RedShift, and Snowflake bulk load. This parameter is enabled only if **/StagingDirectoryDb** is selected. This parameter is supported only for certain location classes. For the list of supported location classes, see Bulk load requires a staging area in Capabilities. The supported formats for providing the credentials are: • Use credentials to retrieve from the AWS or Azure console, For AWS, **aws_access_key_id=***access_key_id***;** **aws_secret_access_key=***secret_access_key***;** For Azure, **azure_account=***azure_account***;azure_secret_access_key=***secret_key***;** • Use temporary credentials from the role of the current AWS EC2 node, **role=***AWS_IAM_role_name***;** • Optionally a static symmetric key for AES256 encryption of staged files can be provided with the above options for AWS, **aws_access_key_id=***access_key_id***;aws_secret_access_key=***secret_access_key***;master_symmetric_key=***32_hex_digits***;** or **role=***AWS_IAM_role_name***;master_symmetric_key=***32_hex_digits***;**
/S3Encryption	*keyinfo*	Enable client or server side encryption for uploading files into S3 locations. When client side encryption is enabled, any file uploaded to S3 is encrypted by HVR prior to uploading. With server side encryption, files uploaded to S3 will be encrypted by the S3 service itself. Value *keyinfo* can be:

- **sse_s3**
- **sse_kms**
- **master_symmetric_key=***64_hex_digits*
- **kms_cmk_id=***aws_kms_key_identifier*
  If only **kms_cmk_id** (without **sse_kms**) is specified, the following optional values can be specified with it:
  - **kms_region=***kms_key_region*
  - **access_key_id=***kms_key_user_access_key_id*
  - **secret_access_key=***kms_key_user_secret_access_key*
  - **role=***AWS_IAM_role_name*
- **matdesc**=*json_key_description* - This optional value can be provided only with the *keyinfo* values (**sse_s3**, **sse_kms**, **master_symmetric_key** or **kms_cmk_id**) to specify encryption materials description. If KMS is used (**kms_cmk_id** or **sse_kms**) then **matdesc** must be a JSON object containing only string values.

> Only the combination **sse_kms** with **kms_cmk_id=***aws_kms_key_identifier* is allowed, otherwise only one of the *keyinfo* value must be specified.

For client side encryption, each object is encrypted with an unique AES256 data key. If **master_symmetric_key** is used, this data key is encrypted with AES256, and stored alongside S3 object. If **kms_cmk_id** is used, encryption key is obtained from AWS KMS. By default, HVR uses S3 bucket region and credentials to query KMS. This can be changed by **kms_region**, **access_key_id** and **secret_access_key**. **matdesc**, if provided, will be stored unencrypted alongside S3 object. An encrypted file can be decrypted only with the information stored alongside to the object, combined with master key or AWS KMS credentials; as per Amazon S3 Client Side Encryption specifications. Examples are:

- **master_symmetric_key=123456789ABCDEF123456789ABCDEF123456789ABCDEF123456789ABCDEF**
- **master_symmetric_key=123456789ABCDEF123456789ABCDEF123456789ABCDEF123456789ABCDEF;matdesc={"hvr":"example"}**
- **kms_cmk_id=1234abcd-12ab-34cd-56ef-1234567890ab**
- **kms_cmk_id=1234abcd-12ab-34cd-56ef-1234567890ab; kms_region=us-east-1;access_key_id=AKIAIOFSODNN7EXAMPLE; secret_access_key=wJalrXUtnFEMI/K7DMENG /bPxRfiCYEXAMPLEKEY**
- **kms_cmk_id=1234abcd-12ab-34cd-56ef-1234567890ab;matdesc= {"hvr":"example"}**

For server side encryption, each object will be encrypted by the S3 service at rest. If **sse_s3** is specified, HVR will activate SSE-S3 server side encryption. If **sse_kms** is specified, HVR will activate SSE-KMS server side encryption using the default aws/s3 KMS key. If additionally **kms_cmk_id=***aws_kms_key_identifier* is specified, HVR will activate SSE-KMS server side encryption using the specified KMS key id. **matdesc**, if provided, will be stored unencrypted alongside S3 object. Examples are:

- **sse_s3;matdesc={"hvr":"example"}**
- **sse_kms**
- **sse_kms;kms_cmk_id=1234abcd-12ab-34cd-56ef-1234567890ab; matdesc={"hvr":"example"}**

/BulkAPI		Use Salesforce Bulk API (instead of the SOAP interface). This is more efficient for large volumes of data, because less roundtrips are used across the network. A potential disadvantage is that some Salesforce.com licenses limit the number of bulk API operations per day. If this parameter is defined for any table, then it affects all tables captured from that location.
/SerialMode		Force serial mode instead of parallel processing for Bulk API. The **default** is parallel processing, but enabling **/SerialMode** can be used to avoid some problems inside Salesforce.com. If this parameter is defined for any table, then it affects all tables captured from that location.
/CloudLicense		Location runs on cloud node with HVR on-demand licensing. HVR with on-demand licensing can be purchased on-line, for example in Amazon or Azure Marketplace. This form of run-time licensing checking is an alternative to a regular HVR license file (file **hvr.lic** in directory **$HVR_HOME/lib** on the hub), which is purchased directly from HVR-Software Corp. HVR checks licenses at run-time; if there is no regular HVR license on the hub machine which permits the activity then it will attempt to utilize any on-demand licenses for locations defined with this parameter. Note that if HVR's hub is on the on-demand licensed machine then its on-demand license will automatically be utilized, so this parameter is unnecessary.

Restrict

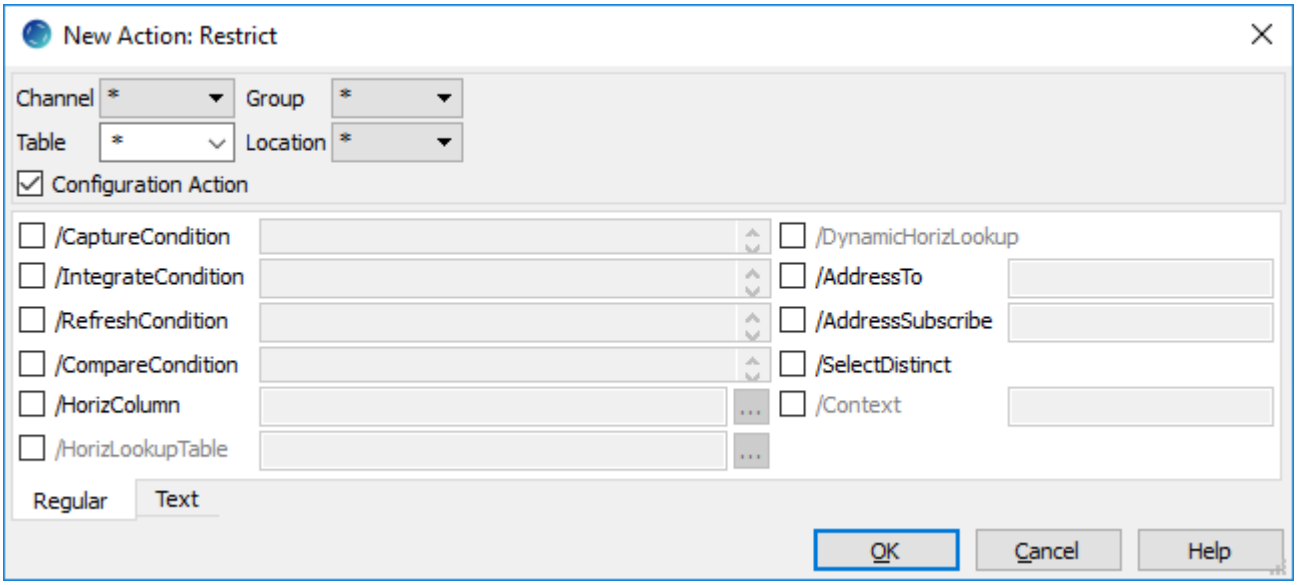
Contents

Description

Action **Restrict** defines that only rows that satisfy a certain condition should be replicated. The restriction logic is enforced during capture and integration and also during compare and refresh.

Parameters

This section describes the parameters available for action **Restrict**.



Parameter	Argument	Description

/CaptureCondition	*sql_expr*	Only rows where the condition *sql_expr* is TRUE are captured. The SQL expression *sql_expr* can contain the following substitutions: • **{***colname***}** is replaced with the value of current table's column *colname*. • **{hvr_cap_loc}** is replaced with the location name. • **{hvr_cap_tstamp}** is replaced with the moment (time) that the change occurred in source location. • **{hvr_cap_user}** is replaced with the name of the user who made the change. A subselect can be supplied, for example **exists (select 1 from lookup where id={id})**. The capture condition is embedded inside the trigger–based capture procedures. This parameter does 'update conversion'. Update conversion is when (for example) an update changes a row which did satisfy a condition and makes it into a row that does not satisfy the condition; such an update would be converted to a delete. If however the update changes the row from not satisfying the condition to satisfying it, then the update is converted to an insert. Parameter **Capture /IgnoreCondition** has a similar effect to this parameter but does not do update conversion. Parameter **/CaptureCondition** can also be defined on a Salesforce location; in this case it should be an Salesforce Object Query Language (SOQL) expression which can be put into the **WHERE** clause of the **SELECT**. Brace substitutions (e. g. **{prod_id} < 100**) are here not performed, but columns can be specified without braces (e.g. **Prod_id < 100**).
/IntegrateCondition	*sql_expr*	Only rows where the condition *sql_expr* is TRUE are integrated. The SQL expression *sql_expr* can contain the following substitutions: • **{***colname***}** is replaced with the value of current table's column *colname*. • **{hvr_cap_loc}** is replaced with the location where the capture was changed. • **{hvr_cap_tstamp}** is replaced with the moment (time) that the change occurred in source location. • **{hvr_cap_user}** is replaced with the name of the user who made the change. A subselect can be supplied, for example **exists (select 1 from lookup where id={id})**. This parameter does 'update conversion'. Update conversion is when (for example) an update changes a row which did satisfy a condition and makes it into a row that does not satisfy the condition; such an update would be converted to a delete. If however the update changes the row from not satisfying the condition to satisfying it, then the update is converted to an insert.

/RefreshCondition	*sql_expr*	During refresh, only rows where the condition *sql_expr* evaluates as TRUE are affected. If **/CompareCondition** is not defined then during compare this parameter also affects which rows are compared. This parameter should not be defined with **/SliceCondition**. This parameter is only supported for DB locations or for File locations with Hive External Tables. For refresh, the effect of this parameter depends on whether it is defined on the source or on the target side. • If defined on the source side, it affects which rows are selected for refreshing (**select * from** *source* **where** *condition*). • If defined on the target side, during bulk refresh it protects non–matching rows from bulk delete (**delete from** *target* **where** *condition*, instead of just **tr uncate** *target*). • If defined for row–wise refresh, it prevents some rows from being selected for comparison with the source rows (**select * from** *target* **where** *condition*). The SQL expression *sql_expr* can contain the following substitutions: • **{***colname***}** is replaced with the value of current table's column *colname*. • **{hvr_var_***xxx***}** is replaced with value of 'context variable' *xxx*. The value of a context variable can be supplied using option **–V***xxx***=***val* to command **Hvrref resh** or **Hvrcompare**. • **{hvr_local_loc}** is replaced with the current location name. • **{hvr_schema}** is replaced with the schema name of the table. • **{hvr_tbl_base_name}** is replaced with the base name of the current table. • **{hvr_opposite_loc}** on the source database is replaced with the target location name and on the target database it is replaced with the source location name. This feature allows compare and refresh to be made aware of horizontal partitioning. • **{hvr_var_slice_condition}** is used (mandatory) if slicing is defined with a boundary expression (option **-S***col***<***b1*[**<***b2*]...[**<***bN*]) or a modulo expression (option **-S***col%N*). **Since** v5.3.1/6 • **{hvr_var_slice_num}** contains current slice number (starting from **0**), if slicing is defined (option **-S**). **Since** v5.3.1/6 • **{hvr_var_slice_total}** contains total number of slices, when slicing is defined (option **-S**). **Since** v5.3.1/6 • **{hvr_var_slice_value}** is used (mandatory) if slicing is defined with a list of values (option **-S***v1*[**;***v2*]...[**;***vN*]). **Since** v5.3.1/6

/CompareCondition	*sql_expr*	Only rows where the condition *sql_expr* evaluates as TRUE are compared. Only these rows are selected for comparison (it can be defined on both databases or just on one). If this parameter is not defined but **/RefreshCondition** is defined then **Hvrcompare** will use **/RefreshCondition** for comparing. This parameter sh ould not be defined with **/SliceCondition**. The SQL expression can contain substitutions: • **{***colname***}** is replaced with the value of current table's column *colname*. • **{hvr_var_***xxx***}** is replaced with value of 'context variable' *xxx*. The value of a context variable can be supplied using option **–V***xxx=val* to command **Hvrref resh** or **Hvrcompare**. • **{hvr_local_loc}** is replaced with the current location name. • **{hvr_opposite_loc}** on the source database is replaced with the target location name and on the target database it is replaced with the source location name. This feature allows compare to be made aware of horizontal partitioning. • **{hvr_schema}** is replaced with the schema name of the table. • **{hvr_tbl_base_name}** is replaced with the base name of the current table. • **{hvr_integ_seq}**, **{hvr_tx_seq}**, **{hvr_tx_scn}** are replaced with values corresponding to SCN of the Oracle 'flashback moment' on the Oracle source database. These substitutions are only available on the target when option **Select Moment** (option **-M**) is supplied to **HVR Refresh**. For example, if the target is defined with actions **ColumnProperties /Name=***tkey* **/TimeKey /IntegrateExpression="{hvr_integ_seq}"** and **Restrict /CompareCondition="***tkey* **<= {hvr_integ_seq}"** then an "on-line compare" can be done by supplying a **Select Moment** (option **-M**) with time or SCN older than the current latency. • **{hvr_var_slice_condition}** is used (mandatory) if slicing is defined with a boundary expression (option **-S***col***<***b1*[**<***b2*]...[**<***bN*]) or a modulo expression (option **-S***col***%***N*). **Since** v5.3.1/6 • **{hvr_var_slice_num}** contains current slice number (starting from **0**), if slicing is defined (option **-S**). **Since** v5.3.1/6 • **{hvr_var_slice_total}** contains total number of slices, when slicing is defined (option **-S**). **Since** v5.3.1/6 • **{hvr_var_slice_value}** is used (mandatory) if slicing is defined with a list of values (option **-S***v1*[**;***v2*]...[**;***vN*]). **Since** v5.3.1/6 For DB locations, more than one substitutions can be supplied using the logical operators **AND**, and **OR**. For example, **{hvr_var_slice_conditio n} AND customer_name = 'Abc'**. For File locations, more than one substitutions can be supplied using the logical operator **&&**. The **OR** operation is not supported for file locations. For example, **{hvr_var_slice_condition} && customer_name = 'Abc'**.

/SliceCondition
Since v5.6.5/0
	sql_expr	During sliced (option **-S**) **refresh** or **compare**, only rows where the condition *sql_expr* evaluates as TRUE are affected. This parameter is allowed and required only for the **Count** (option *num*) and **Series** (option *val1*[**;***val2*]...) type of slicing.
		
		The effect of this parameter depends on whether it is defined on the source or on the target side.
		
		• If defined on the source side, it affects which rows are selected for refreshing or comparing (**select * from** *source* **where** *condition*).
		• If defined on the target side,
		  ○ during bulk refresh it protects non–matching rows from bulk delete (**delete from** *target* **where** *condition*, instead of just **truncate** *target*).
		  ○ during row–wise refresh it prevents some rows from being selected for comparison with the source rows (**select * from** *target* **where** *condition*).
		  ○ during compare it affects which rows are selected (**select * from** *source* **where** *condition*).
		
		The SQL expression *sql_expr* can contain the following substitutions:
		
		• **{hvr_slice_num}** contains current slice number (starting from **0**) if slicing is defined with a **Count** (option **-S***num*).
		• **{hvr_slice_total}** contains total number of slices if slicing is defined with a **Count** (option **-S***num*).
		• **{hvr_slice_value}** contains current slice value if slicing is defined with a **Series** of values (option **-S***v1*[**;***v2*]...[**;***vN*]).
		
		For examples, see **hvrrefresh** or **hvrcompare** option **-S**.
		
		It is recommended to define **/Context** parameter when using these substitutions so it can be easily disabled or enabled.
		
		/SliceCondition cannot be used if any of the following slicing substitution **{hvr_var_slice_condition}**, **{hvr_var_slice_num}**, **{hvr_var_slice_total}**, or **{hvr_var_slice_value}** is mentioned in any of the actions defined on the channel/table involved in the current compare /refresh. In this case, slicing can be achieved using **/RefreshCondition** and **/CompareCondition** using these four substitutions.

/HorizColumn	*col_name*	Horizontal partitioning column. The contents of the column of the replicated table is used to determine the integrate address. If parameter **/HorizLookupTable** is also defined then the capture will join using this column to that table. If it is not defined then the column's value will be used directly as an integrate address. An integrate address can be one of the following: • An integrate location name, such as **dec01**. • A location group name containing integrate locations, such as **DECEN**. • An alias for an integrate location, defined with **/AddressSubscribe** (see below), for example **22**. • A pattern to match one of the above, such as **dec***. • A list of the above, separated by a semicolon, colon or comma, such as **cen, 22**. This parameter should be defined with **Capture /TriggerBased**. When used with trigger–based capture, this parameter does 'update conversion'. Update conversion is when (for example) an update changes a row which did satisfy a condition and makes it into a row that does not satisfy the condition; such an update would be converted to a delete. If however the update changes the row from not satisfying the condition to satisfying it, then the update is converted to an insert. No update conversion is done if this parameter is used with log–based capture.
/HorizLookupTable	*tbl_name*	Lookup table for value in column specified by parameter **/HorizColumn**. The lookup table should have a column which has the name of the **/HorizColumn** parameter. It should also have a column named **hvr_address**. The capture logic selects rows from the lookup table and for each row found stores the change (along with the corresponding **hvr_address**) into the capture table. If no rows match then no capture is done. And if multiple rows match then the row is captured multiple times (for different destination addresses). This parameter is not supported if **Capture /TriggerBased** is not defined. A possible alternative for log–based capture channels is **Restrict /AddressTo** and **Restrict /AddressSubscribe**.
/DynamicHorizLookup		Dynamic replication of changes to lookup table. Normally only changes to the horizontally partitioned table are replicated. This parameter causes changes to the lookup table to also trigger capture. This is done by creating extra rules /triggers that fire when the lookup table is changed. These rules/triggers are name *tbl*__**li**, *tbl*__**lu**, *tbl*__**ld**. Changes are replicated in their actual order, so for example if a transaction inserts a row to a lookup table and then a matching row to the main replicated table, then perhaps the lookup table's insert would not cause replication because it has no match (yet). But the other insert would trigger replication (because it now matches the lookup table row). This dynamic lookup table replication feature is suitable if the lookup table is dynamic and there are relatively few rows of the partitioned replicated table for each row of the lookup table. But if for example a huge table is partitioned into a few sections which each correspond to a row of a tiny lookup table then this dynamic feature could be expensive because an update of one row of the lookup table could mean millions of rows being inserted into the capture table. A more efficient alternative could be to perform an HVR Refresh whenever the lookup table is changed and use parameter **/RefreshCondition** with pattern **{hvr_opposite_loc}** in the condition so that the refresh is aware of the partitioning. This parameter is not supported if **Capture /TriggerBased** is not defined. A possible alternative for log–based capture channels is **Restrict /AddressTo** and **Restrict /AddressSubscribe**.

Column **hvr_address** can contain a location name (lowercase), a location group name (UPPERCASE) or an asterisk (*). An asterisk means send to all locations with **Integrate** defined. It can also contain a comma separated list of the above.

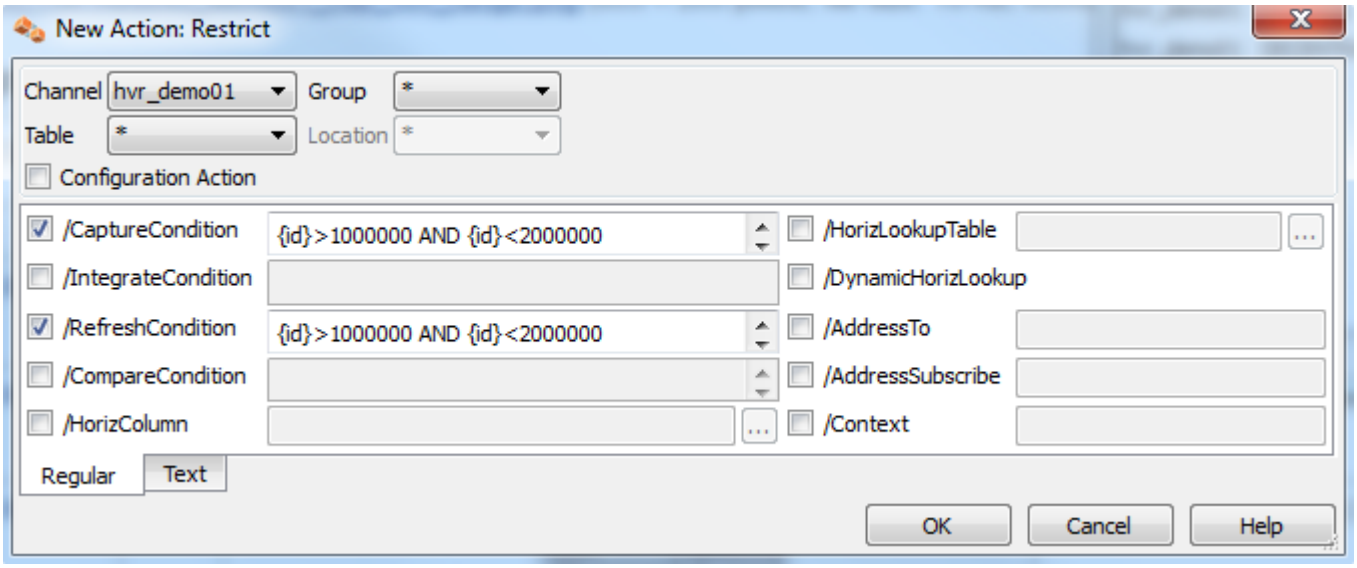
Examples

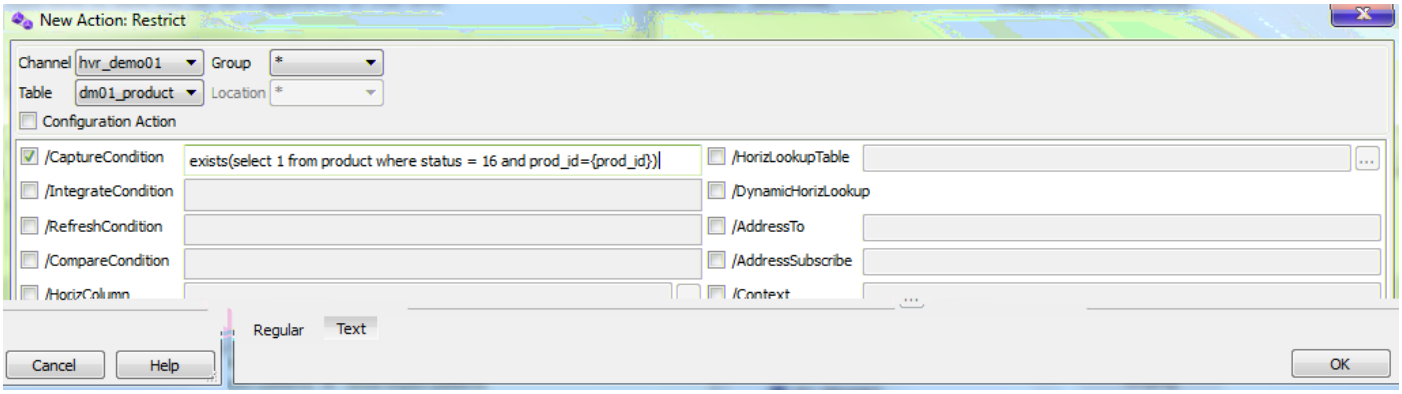
This section describes examples of using the following parameters of **Restrict**:

- **/CaptureCondition** and **/RefreshCondition**
- **/AddressTo** and **/AddressSubscribe**

Using /CaptureCondition and /RefreshCondition

To replicate only rows of table **product** having *id* between 1000000 and 2000000, use parameters **/CaptureCondition** and **/RefreshCondition**. Also only rows of table **order** for products which are in state 16 need to be captured. This is implemented with another **/CaptureCondition** parameter.





Using /AddressTo and /AddressSubscribe

This section describes the examples of using the **Restrict /AddressTo** and **Restrict /AddressSubscribe** actions.

The following replication configuration will be used:

- location group **SRCGRP** having 1 source location **src**
- location group **TGTGRP** having 2 target locations **tgt1** and **tgt2**

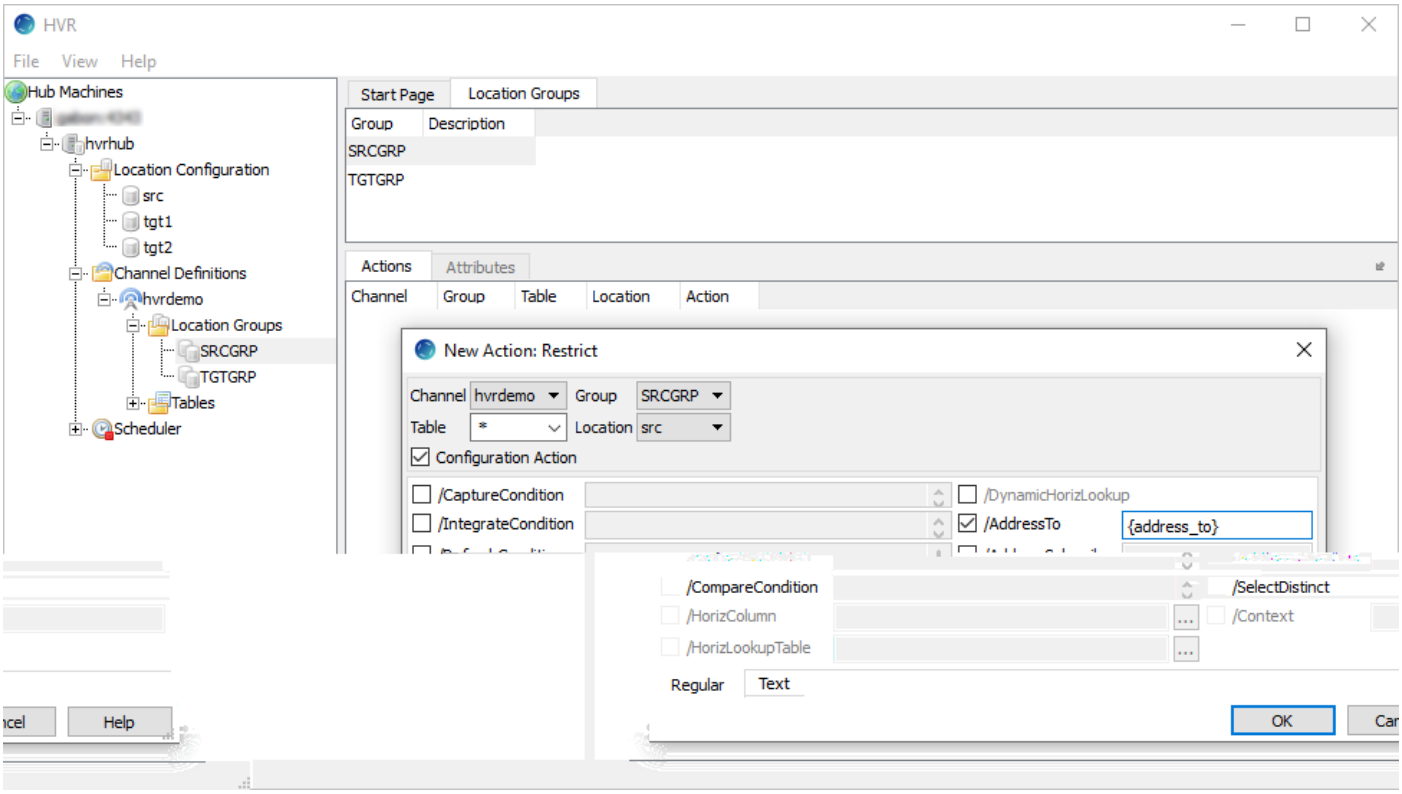
Create table (**t_order**) on source location:

```
create table t_order (
id number primary key,
subid number,
name varchar2(15),
street varchar2(15),
address_to varchar(20));
```

The **address_to** column will serve as a field for enforcing the restriction logic during replication, i.e. captured changes will be replicated to one of the target locations based on the values inserted in this column.

Scenario 1

This example requires changes captured from location **src** to be replicated only to location **tgt1**. In this case, the integrate address is restricted by the content of the **Restrict /AddressTo** field being the **{address_to}** column defined on source location **src** as shown in the screenshot below.



When value 'tgt1' is inserted in the **address_to** column on source location, the change should be replicated only to target location **tgt1**.

```
SQL> insert into t_order values (1,1,'Tester','Boardwalk','tgt1');
```

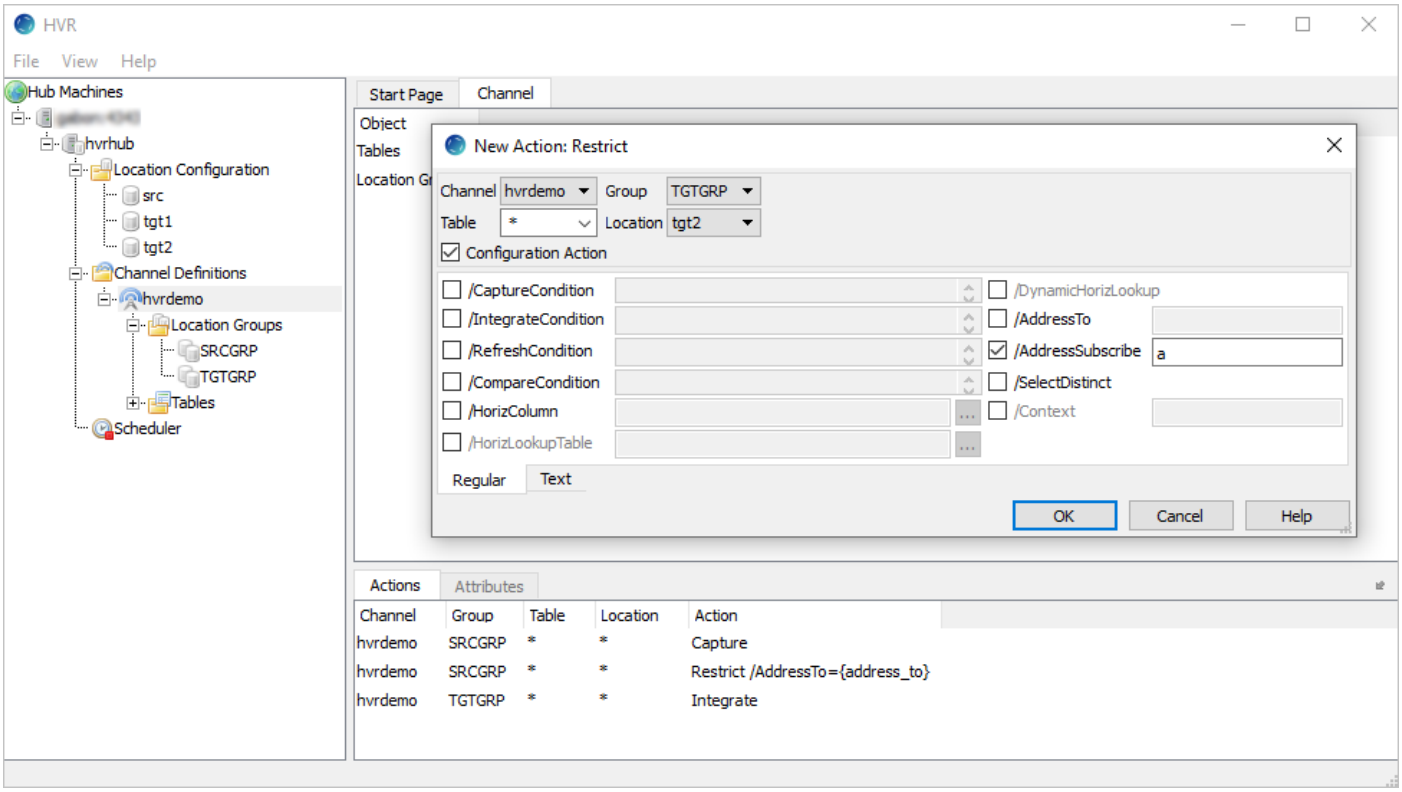
To verify that the change was replicated correctly, make a selection from both **tgt1** and **tgt2**.

```
SQL> select * from tgt1.t_order;

        ID      SUBID NAME            STREET          ADDRESS_TO
---------- ---------- --------------- --------------- ----------
         1          1 Tester          Boardwalk       tgt1
```

```
SQL> select * from tgt2.t_order;
no rows selected
```

Scenario 2

This example requires changes captured from location **src** to be replicated to target group **TGTGRP**, but only to target location **tgt2**, even though location **tgt1** is also a part of **TGTGRP**. In this case, the **Restrict /AddressTo** should be defined on **SRCGRP** with value {address_to} and the integrate address is restricted by the content of the **Restrict /AddressSubscribe** field being alias **a** defined on target location **tgt2** as shown in the screenshot below.



When value 'a' is inserted in the **address_to** column on source location, the change should be replicated only to target location **tgt2**, omitting **tgt1**:

```
SQL> insert into t_order values (5,5,'Tester','Boardwalk','a');
```

To verify that the change was replicated correctly, make a selection from both **tgt1** and **tgt2**.

```
SQL> select * from tgt2.t_order where id = 8;


        ID      SUBID NAME            STREET          ADDRESS_TO
---------- ---------- --------------- --------------- ----------
         8          6 Tester          Boardwalk       a
```

```
SQL> select * from tgt1.t_order where id = 8;
no rows selected
```

Using Subselect on Non-replicated Table in Refresh Condition

This is an example of using **Restrict /RefreshCondition** for updating key values from a source table to a target table based on a subset of values from a table in the source that is not included in the channel.

Prerequisites

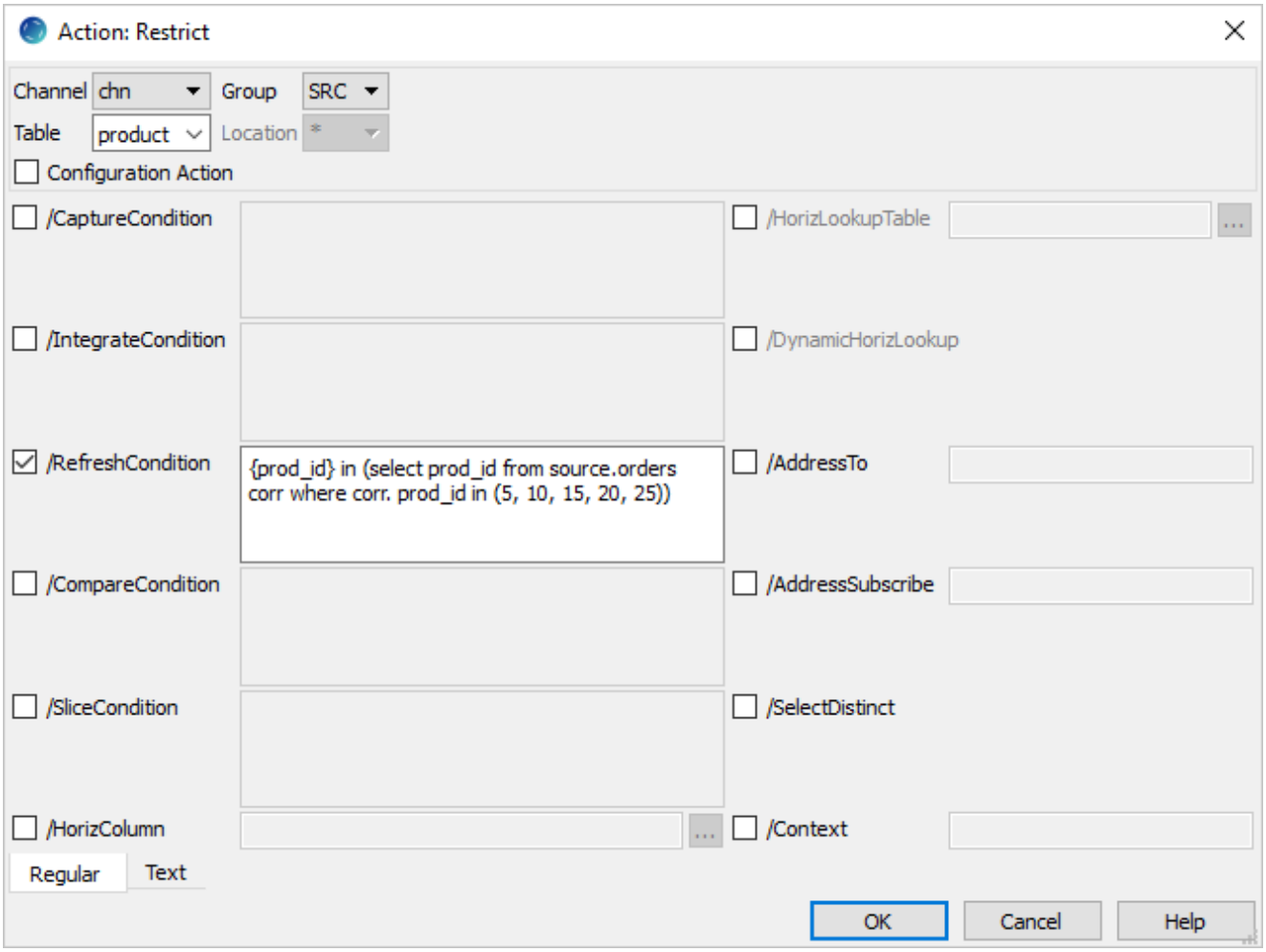
1. An Oracle-to-Oracle channel **chn** with the **Product** table on both source and target locations.
2. The **Orders** table on the source location that is not in the channel.

Scenario

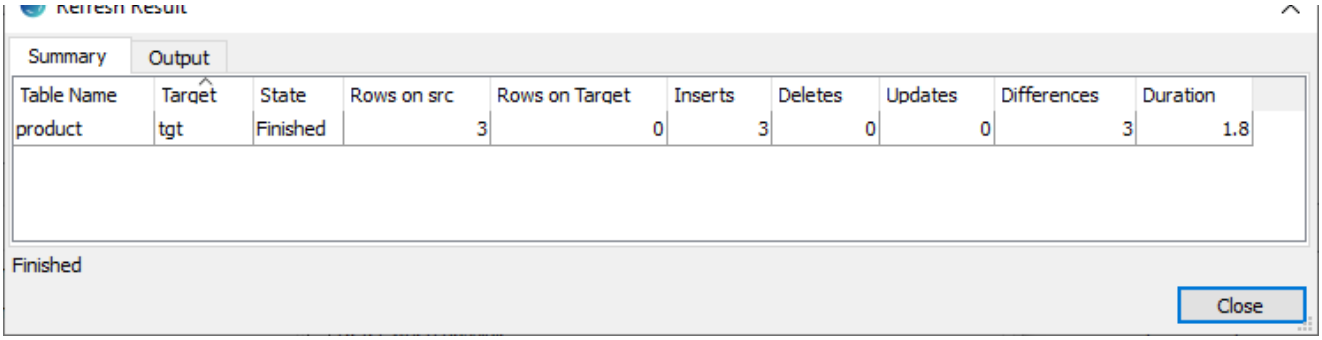
Suppose we need to update the values of the **Prod_ID** column in the **Product** table with only values falling under a certain subset, such as 5, 10, 15, 20, and 25, from another non-replicated table **Orders**. To achieve this, we will define a subquery expression for the **/RefreshCondition** parameter, that will select specific values from the **Orders** table on the source even though this table is not in the channel definition.

Steps

1. In the HVR GUI, define the **Capture** action on the source table: right-click the source group, navigate to **New Action** and click **Capture**. Click **OK**.
2. Define the **Integrate** action on the target table: click the target group, navigate to **New Action** and click **Integrate**. Click **OK**.
3. Define the **Restrict** action on the source table: right-click the source group, navigate to **New Action** and click **Restrict**.
4. In the **New Action: Restrict** dialog, select the **/RefreshCondition** checkbox and type in the following expression: **{prod_id} in (select prod_id from source.orders corr where corr.prod_id in (5, 10, 15, 20, 25))**. Click **OK**.



5. Right-click the channel **chn** and click **Refresh**. Select the **Product** table.
6. Under the **Options** tab, select the refresh method: either **Bulk Granularity** or **Row by Row Granularity**. Optionally, configure other parameters. Click **OK**.
7. The **Refresh Result** dialog shows the summary statistics of the refresh operation. In this case, three rows were inserted into the target database.



Scheduling

Contents

Description

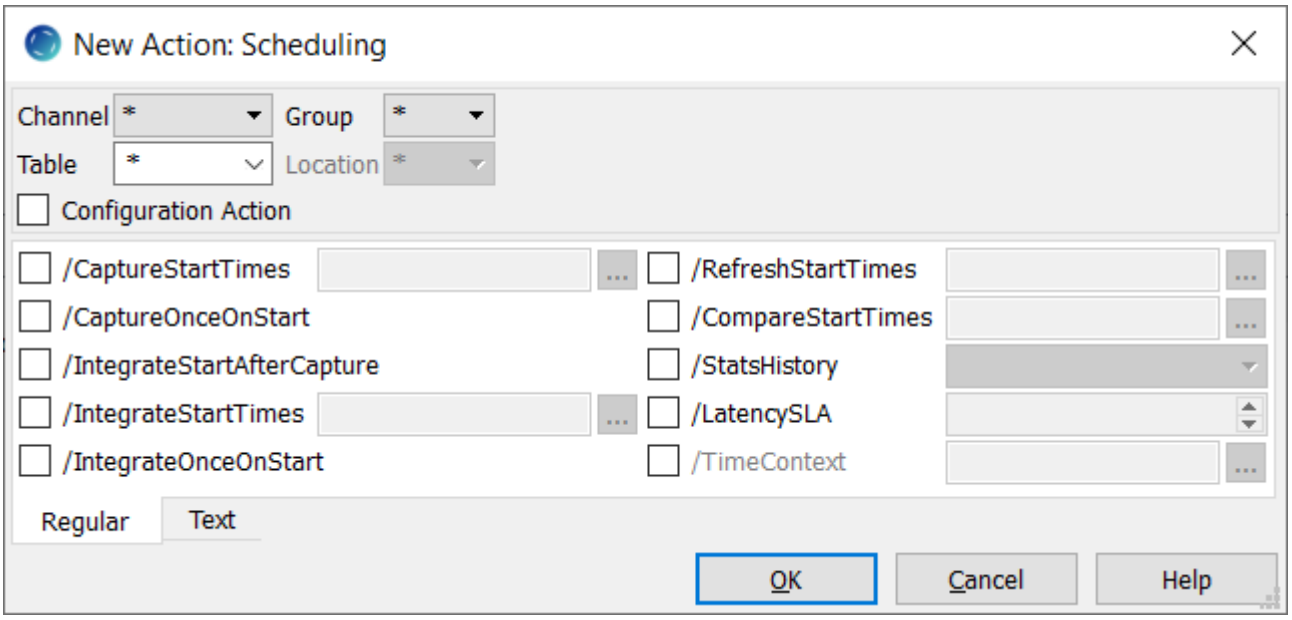
Action **Scheduling** controls how the replication jobs generated by **Hvrinit** and **Hvrrefresh** will be run by **Hvrscheduler**. By default, (if this **Scheduling** action is not defined) HVR schedules capture and integrate jobs to run continuously. This means after each replication cycle they will keep running and wait for new data to arrive. Other parameters also affect the scheduling of replication jobs, for example **Capture /ToggleFrequency**.

If this action is defined on a specific table, then it affects the entire job including data from other tables for that location.

A **Scheduling** action is only effective at the moment that the job is first created, i.e. when **HVR Initialize** creates capture or integrate jobs or when **HVR Refresh** creates a refresh job. After this moment, redefining this action has no effect. Instead, the scheduler's job attributes (such as **trig_crono**) can be manipulated directly by clicking the **Attributes** tab while inspecting a job in the HVR GUI.

Parameters

This section describes the parameters available for action **Scheduling**.



Parameter	Argument	Description
/CaptureStartTimes	*times*	Defines that the capture jobs should be triggered at the given times, rather than cycling continuously. Example, **/CaptureStartTimes**="0 * * * 1–5" specifies that capture jobs should be triggered at the start of each hour from Monday to Friday. For the format of *times* see section Start Times below.
/CaptureOnceOnStart		Capture job runs for one cycle after trigger. This means that the job does not run continuously, but is also not triggered automatically at specified times (the behavior of **/CaptureStartTimes**). Instead, the job stays **PENDING** until it is started manually with command **Hvrstart**.
/IntegrateStartAfterCapture		Defines that the integrate job should run after a capture job routes new data.
/IntegrateStartTimes	*times*	Defines that the integrate jobs should be triggered at the given times, rather than cycling continuously. For the format of *times* see section Start Times below.

<table>
<tr><td>/IntegrateOnceOnStart</td><td></td><td>Integrate job runs for one cycle after trigger. This means that the job does not run continuously, but is also not triggered automatically at specified times (the behavior of /IntegrateAfterCapture or /IntegrateStartTimes). Instead, the job stays PENDING until it is started manually with command Hvrstart.</td></tr>
<tr><td>/RefreshStartTimes</td><td>times</td><td>Defines that the refresh jobs should be triggered at the given times.

By default they must be triggered manually.

This parameter should be defined on the location on the 'write' side of refresh.

For the format of times see section Start Times below.</td></tr>
<tr><td>/CompareStartTimes</td><td>crono</td><td>Defines that the compare jobs should be triggered at the given times.

By default they must be triggered manually.

This parameter should be defined on the location on the 'right' side of compare.

For the format of times see section Start Times below.</td></tr>
<tr><td>/StatsHistory</td><td>size</td><td>Size of history maintained by hvrstats job, before it purges its own rows.

Available options for size are:

<table>
<tr><td>NONE</td><td>History is not maintained by hvrstats job. Does not add history rows to hvr_stats.</td></tr>
<tr><td>SMALL</td><td>History rows for per-table measurements at 1min/10min/1hour/1day granularity are purged after 1hour /4hours/1day/7days respectively.

History rows for all tables (table=*) at 1min/10 min/1hour/1 day granularity are purged after 4hours/1day /7days/30days respectively.</td></tr>
<tr><td>MEDIUM (default)</td><td>History rows for per-table measurements at 1min/10min/1hour/1day granularity are purged after 4hours /1day/7days/30days respectively.

History rows for all tables (table=*) at 1min/10min/1hour/1day granularity are purged after 1day/7days /30days/never respectively.</td></tr>
<tr><td>LARGE</td><td>History rows for per-table measurements at 1min/10min/1hour/1day granularity are purged after 1day /7days/30days/never respectively.

Rows for all tables (table=*) at 1min/10min/1hour/1day granularity are purged after 7days/30days/never /never respectively.</td></tr>
<tr><td>UNBOUNDED</td><td>Never purge history rows. Rows continue to grow in hvr_stats.</td></tr>
</table>
A smaller policy will reduce the amount of disk space needed for the hub database. For example, if a hub has 2 channels with same locations (1 capture and 2 targets) and each has 15 busy tables measured using 10 status measurements, then the following is the approximate number of rows in hvr_stats after 1 year;SMALL : 207 K rowsMEDIUM : 1222 K rowsLARGE : 7 M rowsUNBOUNDED : 75 M rows


To purge the statistics data immediately (as a one-time purge) from the hvr_stats table, use the command hvrstats (with option -p).

Related topic, Statistics</td></tr>
</table>

Start Times

Argument *times* uses a format that closely resembles the format of Unix's **crontab** and is also used by scheduler attribute **trig_crono**. It is composed of five integer patterns separated by spaces. These integer patterns specify:

- minute (0–59)
- hour (0–23)
- day of the month (1–31)
- month of the year (1–12)
- day of the week (0–6 with 0=Sunday)

Each pattern can be either an asterisk (meaning all legal values) or a list of comma–separated elements. An element is either one number or two numbers separated by a hyphen (meaning an inclusive range). All dates and times are interpreted using the local–time. Note that the specification of days can be made by two fields (day of the month and day of the week): if both fields are restricted (i.e. are not *), the job will be started when either field matches the current time. Multiple start times can be defined for the same job.

TableProperties

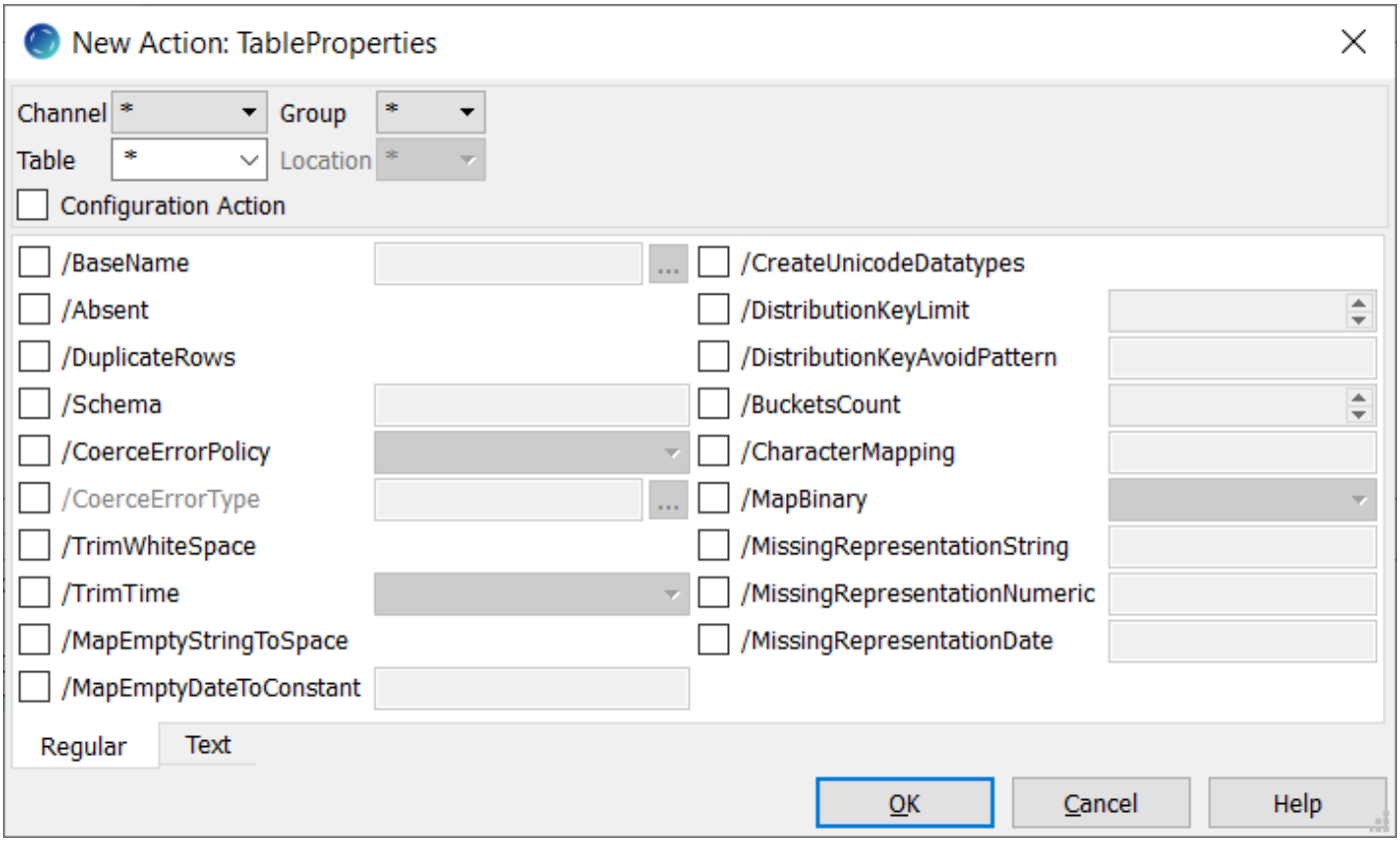
Contents

Description

Action **TableProperties** defines properties of a replicated table in a database location. The action has no effect other than that of its parameters. These parameters affect both replication (on the capture and integrate side) and HVR refresh and compare.

Parameters

This section describes the parameters available for action **TableProperties**.



Parameter	Argument	Description

/BaseName	*tbl_name*	This action defines the actual name of the table in the database location, as opposed to the table name that HVR has in the channel. This parameter is needed if the 'base name' of the table is different in the capture and integrate locations. In that case the table name in the HVR channel should have the same name as the 'base name' in the capture database and parameter **/BaseName** should be defined on the integrate side. An alternative is to define the **/BaseName** parameter on the capture database and have the name for the table in the HVR channel the same as the base name in the integrate database. Parameter **/BaseName** is also necessary if different tables have the same table name in a database location but have different owners (**/Schema** parameter). Or if a table's base name is not allowed as an HVR name, e.g. if it contains special characters or if it is too long. If this parameter is not defined then HVR uses the base name column (this is stored in **tbl_base_name** in catalog **hvr_table**). The concept of the 'base name' in a location as opposed to the name in the HVR channel applies to both columns and tables, see **/BaseName** in **ColumnProperties** . Parameter **/BaseName** can also be defined for file locations (to change the name of the table in XML tag) or for Salesforce locations (to match the Salesforce API name).
/Absent		Table does not exist in database. Example: parameter **/Absent** can be defined if a table needs to be excluded from one integrate location but included in another integrate location for a channel with parallel integration jobs.
/DuplicateRows		Replication table can contain duplicate rows. This parameter only has effect if no replication key columns are defined for the table in **hvr_column** . In this case, all updates are treated as key updates and are replicated as a delete and an insert. In addition, each delete is integrated using a special SQL subselect which ensures only a single row is deleted, not multiple rows.
/Schema	*schema*	Name of database schema or user which owns the base table. By default the base table is assumed to be owned by the database username that HVR uses to connect to the database.
/IgnoreCoerceError		No error will be reported if an error is outside the boundary of a destination data type or if a conversion is impossible because of a data type difference (e.g. string '**hello**' must be converted into an integer). Instead HVR will silently round an integer or date up or down so it fits within the boundary, truncate long strings or supply a default value if a value cannot be converted to a target data type. The default value used for date is **0001 01 01**. ⚠ Since HVR 5.3.1/16, **/IgnoreCoerceError** is replaced with **/CoerceErrorPolicy**.

/CoerceErrorPolicy **Since** v5.3.1/16	*policy*	Defines a *policy* to handle type coercion error (an error which occurs while converting a value from one data type to a target data type). This *policy* typically affects all types of coercion errors, unless parameter **/CoerceErrorType** is defined in the same action. Multiple actions with **/CoerceErrorPolicy** can be defined to apply different policies to different coercion error types. Available options for *policy* are: • **FATAL** (**default**): The replication job fails and displays an error message mentioning the table and column name where the bad values are encountered. • **SILENT** : The bad value is silently (without notification) coerced /replaced with a value that is legal. The coercion details are not written into the job's log file. For details about the how bad values are replaced/coerced, see **/CoerceErrorType** below. • **WARNING** : A warning message is written into the job's log file. The warning message contains the table and column name, the type of coercion (see **/CoerceErrorType** below), and the number of rows affected. • **WARNING_FILE** : The rows with bad values and the values in the key columns are written into a binary file (with extension **.coererr**) on the hub machine and also a warning message is written into the job's log file. This warning contains the table and column name, the type of coercion (see **CoerceErrorType** below), the number of rows affected, and the binary file name. The binary file has the same format as HVR's transaction files. To view its contents, use **Hvrrouterview** command: `hvrrouterview hubdb chn $HVR_CONFIG/router/hubdb/chn/coerceerror/YYYYMMDD/YYYYMMMDDHHmmSS-jobname.coererr`

/CoerceErrorType **Since** v5.3.1/16	*types*	This parameter defines which types of coercion errors are affected by **/CoerceErrorPolicy**. The default (if only **/CoerceErrorPolicy** is defined) is for all the below coercion errors to be affected. When multiple *types* are selected, it should be a comma-separated list. Available options for *types* are: • **NUMERIC_RANGE** : When value exceeds the minimum or maximum value allowed in the target numeric data type and the **/CoerceErrorPolicy** is not **FATAL**, the bad value will be replaced with the minimum or maximum legal value. • **DATE_RANGE** : When value exceeds the minimum or maximum value allowed in the target date data type and the **/CoerceErrorPolicy** is not **FATAL**, the bad value will be replaced with the minimum or maximum legal value. • **STRING_TRUNCATION** : When value exceeds the number of bytes or characters allowed in the target string data type and the **/CoerceErrorPolicy** is not **FATAL**, the bad value will be replaced with a truncated value. • **ROUNDING** : When value's precision exceeds the precision allowed in the target date or numeric data type and the **/CoerceErrorPolicy** is not **FATAL**, the bad value will be rounded to a legal value. • **ENCODING** : When invalid sequences are encountered during encoding from source to target string data type and the **/CoerceErrorPolicy** is not **FATAL**, the the sequence will be sanitized with a replacement sequence. • **NULLS** : When null value is encountered for non nullable target data type and the **/CoerceErrorPolicy** is not **FATAL**, the bad value this will be replaced with default value (a zero or an empty string depending on the data type). • **OTHER** : Anything that does not match any of the above *types* (e.g. non-number string '**hello**' being coerced to a target numeric data type), and the **/CoerceErrorPolicy** is not **FATAL**, this will be replaced with NULL or a default value (zero or empty string) depending on the data type.
/TrimWhiteSpace		Remove trailing whitespace from **varchar**.
/TrimTime	*policy*	Trim **time** when converting **date** from Oracle and SQL Server. Available options for *policy* are: • **YES** : Always trim **time** • **NO** (**default**) : Never trim **time** • **MIDNIGHT** : Only trim if value has **time** as **00:00:00**
/MapEmptyStringToSpace		Convert empty Ingres or SQL Server **varchar** values to an Oracle **varchar2** containing a single space and vice versa.
/MapEmptyDateToConstant	*date*	Convert between Ingres empty date and a special constant *date*. Value *date* must have form *DD**MMM**YYYY*.
/CreateUnicodeDatatypes		On table creation use unicode data types for string columns, e.g. map **varchar** to **nvarchar**

/DistributionKeyLimit	*int*	Maximum number of columns in the implicit distribution key. The **default** value is **1** (just one column). Value **0** means all key columns (or regular columns) can be used. A table's distribution key can be set explicitly or implicitly. An explicit distribution key can be set by clicking the checkboxes in the table's dialog, or by defining parameter **ColumnProperties /DistributionKey**. If no explicit distribution key is defined, then HVR uses the implicit distribution key rule. The implicit rule is to use the first N columns; either from the replication key, or (if the table has no replication key) from the regular columns which do not have a LOB data type. Some DBMS's (such as Redshift) are limited to only one distribution key column.
/DistributionKeyAvoidPattern	*patt*	Avoid putting given columns in the implicit distribution key. For a description of the implicit distribution key, see parameter **/DistributionKeyLimit** above. The **default** value is **''** (no column is avoided). If this parameter is defined then HVR will avoid adding any columns whose name matches to the implicit distribution key. So if the table has replication key columns (*k1 k2 k3 k4*) and **/DistributionKeyAvoidPattern**=*'k2|k3'* and **/DistributionKeyLimit**=*2* then the implicit distribution key would be (*k1 k4*). But if **/DistributionKeyAvoidPattern**=*'k2|k3'* and **/DistributionKeyLimit**=*4* then the implicit distribution key would be (*k1 k2 k3 k4*). For SAP databases, column **'mandt'** is often constant, so parameter **/DistributionKeyAvoidPattern**=*mandt* should be used.
/BucketsCount **Hive ACID**		Number of buckets to be specified while creating a table in Hive ACID. If this parameter is not selected, the **default** value is **1**.
/CharacterMapping	*rules*	Allows to replace some characters (potentially unsupported) in string columns with a replacement sequence. Value *rules* should be a semicolon separated list of elements, each with form *char***>***chars*. Each *char* be a literal character or have form **\n**, **\r**, **\t**, ****, **\x***NN*, **\u***NNNN*, **\U***NNNNNNNN* (where *N* is a hex digit). Example;**"\n>\\n;\r>\\r;\x00>\\0"**. Note that the mapping is performed during integration, refresh and also compare, so the HVR compare does not show an earlier mapping as a difference.
/MapBinary	*policy*	Controls the way binary columns are mapped to a string. This parameter is relevant only if the location does not support any binary data type (e.g. Redshift) (or) **FileFormat /Csv** or **FileFormat /Json** is defined for the location (or) a binary column is explicitly mapped to a string column using **ColumnProperties /DatatypeMatch /Datatype**. Available options for *policy* are: • **COPY** (**default** for CSV and databases): Memory copy of the binary data. This can cause invalid characters in the output. • **HEX** : The binary value is represented as HEX string. • **BASE64** (**default** for Json): The binary value is represented as Base64 string.

/MissingRepresentationString **Since** v5.3.1/5 **File** **Kafka**	*str*	Inserts value *str* into the string data type column(s) if value is missing /empty in the respective column(s) during integration. The value *str* defined here should be a valid input for the column(s) in target database. When **/MissingRepresentationNumeric** or **/MissingRepresentationDate** is used without defining **/MissingRepresentationString** then a default value (for example, empty string) is inserted into the string data type column(s) in which the value is missing/empty. Defining **/MissingRepresentationString** enables HVR to use **ColumnProperties /TimeKey** without requiring supplemental logging all.
/MissingRepresentationNumeric **Since** v5.3.1/5 **File** **Kafka**	*str*	Inserts value *str* into the numeric data type column(s) if value is missing /empty in the respective column(s) during integration. The value *str* defined here should be a valid input for the column(s) in target database. When **/MissingRepresentationString** or **/MissingRepresentationDate** is used without defining **/MissingRepresentationNumeric** then a default value (for example, 0) is inserted into the numeric data type column(s) in which the value is missing/empty. Defining **/MissingRepresentationNumeric** enables HVR to use **Column Properties /TimeKey** without requiring supplemental logging all.
/MissingRepresentationDate **Since** v5.3.1/5 **File** **Kafka**	*str*	Inserts value *str* into the date data type column(s) if value is missing /empty in the respective column(s) during integration. The value *str* defined here should be a valid input for the column(s) in target database. When **/MissingRepresentationNumeric** or **/MissingRepresentationString** is used without defining **/MissingRepresentationDate** then a default value is inserted into the date data type column(s) in which the value is missing/empty. Defining **/MissingRepresentationDate** enables HVR to use **ColumnProperties /TimeKey** without requiring supplemental logging all.

Transform

Contents

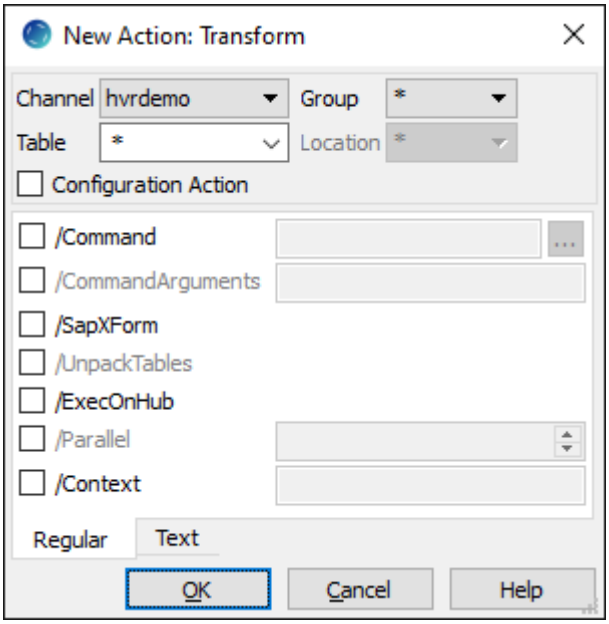
Description

Action **Transform** defines a data mapping that is applied inside the HVR pipeline. A transform can either be a command (a script or an executable), or it can built into HVR (such as **/SapAugment**). These transform happens after the data has been captured from a location and before it is integrated into the target, and is fed all the data in that job's cycle. To change the contents of a file as HVR reads it or to change its contents as HVR writes it, use **FileFormat /CaptureConverter** or **/IntegrateConverter**. This **Transform** action happens between the changes from those converters.

A command transform is fed data in XML format. This is a representation of all the data that passes through HVR's pipeline. A definition is in **$HVR_HOME/lib/hvr_private.dtd**.

Parameters

This section describes the parameters available for action **Transform**.



Parameter	Argument	Description

/Command	*path*	Name of transform command. This can be a script or an executable. Scripts can be shell scripts on Unix and batch scripts on Windows or can be files beginning with a 'magic line' containing the interpreter for the script e.g. **#!perl**. A transform command should read from its **stdin** and write the transformed bytes to **stdout**. Argument *path* can be an absolute or a relative pathname. If a relative pathname is supplied the agents should be located in **$HVR_HOME/lib/transform**. This parameter can either be defined on a specific table or on all tables (*****). Defining it on a specific table could be slower because the transform will be stopped and restarted each time the current table name alternates. However, defining it on all tables (*****) requires that all data must go through the transform, which could also be slower and costs extra resource (e.g. diskroom for a **/Command** transform).
/CommandArguments	*userarg*	Value(s) of parameter(s) for transform (space separated).
/SapAugment		Capture job selecting for de-clustering of multi-row SAP cluster tables. ⚠ This parameter is not supported since HVR version 5.3.1/19.
/SapXForm		Invoke SAP transformation for SAP pool and cluster tables. This parameter requires **Integrate /Burst** defined in the channel. This parameter should not be used together with **AdaptDDL** or **Capture /Coalesce**. ⚠ This parameter requires SAP decluster engine, which is included in HVR distributions since HVR 5.3.1/4. Prior to HVR 5.3.1/4, this parameter requires **/ExecOnHub**.
/UnpackTables		Transform will map *_**pack** tables into ***_*_unpack** tables.
/ExecOnHub		Execute transform on hub instead of location's machine.
/Parallel	*N*	Run transform in *N* multiple parallel branches. Rows will be distributed using hash of their distribution keys, or round robin if distribution key is not available. Parallelization starts only after first 1000 rows.

/Context	*context*	Ignore action unless refresh or compare *ctx* is enabled. The value should be the name of a context (a lowercase identifier). It can also have form **!***ctx*, which means that the action is effective unless context ctx is enabled. One or more contexts can be enabled for HVR Compare or Refresh (on the command line with option **–C***ctx*). Defining an action which is only effective when a context is enabled can have different uses. For example, if action **Transform /Command="C:/hvr/script/myscriptfile" /Context=qqq** is defined, then during replication no transform will be performed but if a refresh is done with context **qqq** enabled (option **–Cqqq**), then the transform will be performed. If a 'context variable' is supplied (using option **–V***xxx=val*) then this is passed to the transform as environment variable **$HVR_VAR_***XXX*.

Command Transform Environment

A transform command should read from its **stdin** and write the transformed bytes to **stdout**. If a transform command encounters a problem, it should write an error to **stderr** and return with exit code **1**, which will cause the replication jobs to fail. The transform command is called with multiple arguments, which should be defined using parameter **/CommandArguments**.

A transform inherits the environment from its parent process. On the hub, the parent of the transform's parent process is the HVR Scheduler. On a remote Unix machine, it is the **inetd** daemon. On a remote Windows machine it is the HVR Remote Listener service. Differences with the environment process are as follows:

- Environment variables **$HVR_CHN_NAME** and **$HVR_LOC_NAME** are set.
- Environment variable **$HVR_TRANSFORM_MODE** is set to either value **cap**, **integ**, **cmp**, **refr_read** or **refr_write** .
- Environment variable **$HVR_CONTEXTS** is defined with a comma–separated list of contexts defined with HVR Refresh or Compare (option **–C***ctx*).
- Environment variables **$HVR_VAR_***XXX* are defined for each context variable supplied to HVR Refresh or Compare (option **–V***xxx=val*).
- Any variable defined by action **Environment** is also set in the transform's environment, unless **/ExecOnHub** is defined.
- The current working directory is **$HVR_TMP**, or **$HVR_CONFIG/tmp** if this is not defined.
- **stdin** is redirected to a socket (HVR writes the original file contents into this), whereas **stdout** and **stderr** are redirected to separate temporary files. HVR replaces the contents of the original file with the bytes that the transform writes to its **stdout**. Anything that the transform writes to its **stderr** is printed in the job's log file on the hub machine.

Commands

HVR can be configured and controlled either by using the Graphical User Interface (GUI) or the Command Line Interface (CLI). This chapter describes all HVR commands and their parameters.

Calling HVR on the Command Line

Many HVR commands take a hub database name (e.g., **myhubdb**) as their first argument. HVR supports the creation of a hub database on certain databases (location classes) only. For the list of supported location classes, see section Hub Database in Capabilities.

Along with the hub database name argument (*hubdb*), the location class of the hub database can be explicitly specified in the command line using option **-h***class*. Valid values for *class* are **db2**, **db2i**, **ingres**, **mysql**, **oracle**, **postgresql**, **sqlserver**, or **teradata**.

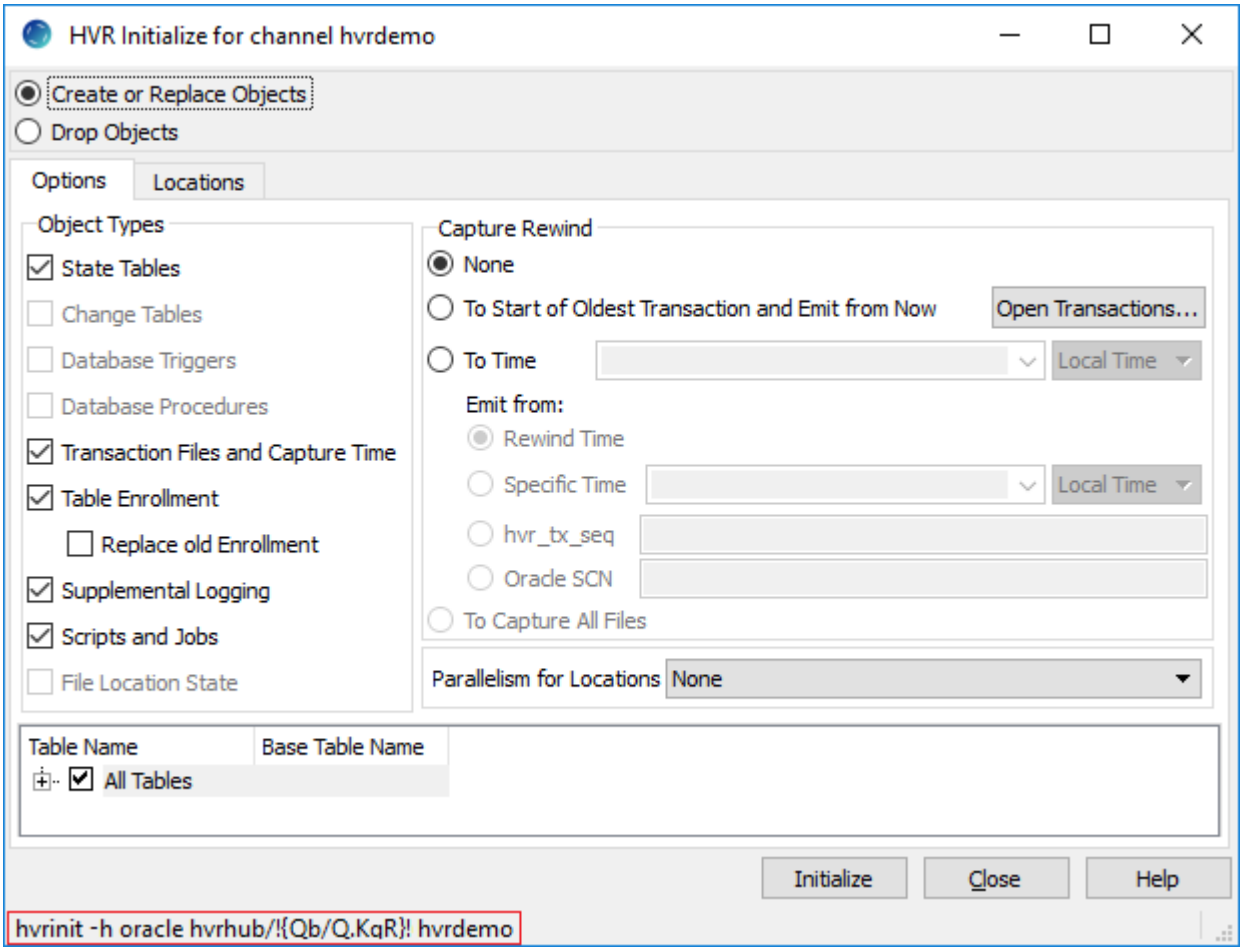
> Alternatively, the location class of the hub database can be set by defining the environment variable **HVR_HUB_CLASS**. Valid values for this environment variable are **db2**, **db2i**, **ingres**, **mysql**, **oracle**, **postgresql**, **sqlserver**, or **teradata**. Refer to the operating system documentation for the steps to set the environment variables.

The following table lists (location class wise) the syntax for using the hub database argument (*hubdb*) in the command line interface (CLI). Sometimes a DBMS password may be required as a command line argument. These passwords can be supplied in an encrypted form using the command **hvrcrypt** to prevent them being visible in the process table (for example with Unix command **ps**).

Syntax	Example	Description
hvrcommand **-h db2** *hubdb channel*	**hvrinit -h db2 myhubdb hvr_demo**	DB2 hub database as **myhubdb**.
hvrcommand **-u** *username**/**password hubdb channel*	**hvrinit -u myuser/mypwd myhubdb hvr_demo**	A username and password (of the hub database) can be supplied with option **–u**.
hvrcommand **-h ingres** *hubdb channel*	**hvrinit -h ingres myhubdb hvr_demo**	Ingres or Vectorwise hub database as **myhubdb**.
hvrcommand **-h mysql -u** *username**/**password node~port~hubdb channel*	**hvrinit -h mysql -u myuser /mypwd mynode~3306~myhubdb hvr_demo**	MySQL hub database as **myhubdb**. A username and password (of the hub database) is supplied with option **–u**.
*hvrcommand hubdb**/**password channel* *hvrcommand hubdb**/**password @tnsname channel* *hvrcommand* **-h oracle** *hubdb**/**password channel*	**hvrinit myhubdb/mypwd hvr_demo** **hvrinit myhubdb /mypwd@mytnsname hvr_demo** **hvrinit -h oracle myhubdb/mypwd hvr_demo**	Oracle schema as **myubdb** with password **mypwd**. Note that in the first example HVR recognizes this as Oracle, because of the forward slash.
hvrcommand **-h postgresql -u** *username**/**password node~port~hubdb channel*	**hvrinit -h postgresql -u myuser /mypwd mynode~5432~myhubdb hvr_demo**	PostgreSQL hub database as **myhubdb** . A username and password (of the hub database) is supplied with option **–u**.

hvrcommand *****\hubdb channel* *hvrcommand instance*\hubdb channel* *hvrcommand* node\instance\hubdb channel* *hvrcommand* **-u** *username*/password* *****\hubdb channel* *hvrcommand* **-h sqlserver** *hubdb channel*	**hvrinit \myhubdb hvr_demo** **hvrinit inst\myhubdb hvr_demo** **hvrinit mynode\myinst\myhubdb hvr_demo** **hvrinit -u myuser/mypwd \myhubdb hvr_demo** **hvrinit -h sqlserver myhubdb hvr_demo**	SQL Server hub database **myhubdb**. Note that HVR recognizes this as SQL Server because of the back slash. A SQL Server node and SQL Server instance can be added with extra back slashes. A username and password (of the hub database) can be supplied with option **–u**.
hvrcommand **-h teradata** node~*username*/password* *channel*	**hvrinit -h teradata mynode~myuser/mypwd hvr_demo**	Teradata schema as **myuser** on **mynode**.

Certain HVR commands can also be performed inside HVR's Graphical User Interface (**HVR GUI**). In such command's GUI dialog, the equivalent command is displayed at the bottom of the dialog window.



Command Reference

This section lists the HVR commands with short description. For more details about a command, click on the command name.

Command	Description
hvr	HVR runtime engine.
hvradapt	Select base table definitions and compare with channel information.
hvrcatalogcreate	Create catalog tables in hub database.
hvrcatalogdrop	Drop catalog tables from hub database.
hvrcatalogexport	Export from hub database into HVR catalog document.
hvrcatalogimport	Import from HVR catalog document into hub database.
hvrcompare	Compare data in tables.
hvrcontrol	Send and manage internal control files.
hvreventtool	Manage (add/edit/change state) events.
hvrcrypt	Encrypt passwords.
hvreventview	Displays events and their results.
hvrfailover	Failover between Business Continuity nodes using replication.
hvrfingerprint	Display host fingerprint.
hvrgui	HVR Graphical User Interface.
hvrinit	Load a replication channel.
hvrlivewallet	Set passwords to the HVR Live Wallet port.
hvrlogrelease	Manage DBMS logging files when not needed by log–based capture.
hvrmaint	Housekeeping script for HVR on the hub machine.
hvrproxy	HVR proxy.
hvrrefresh	Refresh the contents of tables in the channel.
hvrremotelistener	HVR Remote Listener.
hvrretryfailed	Retry changes saved in fail tables or directory due to integration errors.
hvrrouterconsolidate	Merge small tx files in router directory.
hvrrouterview	View or extract contents from internal router files.
hvrscheduler	HVR Scheduler server.
hvrsslgen	Generate a private key and public certificate pair.
hvrstart	Start HVR jobs.
hvrstatistics	Extract statistics from HVR scheduler log files.
hvrstats	Gather or output statistics information.
hvrstrip	Purge/remove HVR installation files.

hvrsuspend	Suspend (or unsuspend) HVR jobs.
hvrswitchtable	Schedule merge of one channel's tables into another channel without interrupting replication.
hvrtestlistener	Test listening on TCP/IP port for HVR remote connection.
hvrtestlocation	Test connection to HVR location.
hvrtestscheduler	Test (ping) that the HVR Scheduler is running.
hvrvalidpw	Authentication plugin to validate the username/password of incoming connections.
hvrwalletconfig	Configure HVR hub wallet.
hvrwalletopen	Open, close a hub encryption wallet and verify the wallet password.

Hvr

Contents

Name

hvr - HVR runtime engine.

Synopsis

hvr [**-E***n=v*]... [**-t***x*] [*script* [**-s***cropts*] [*scrargs*]]

hvr -r [**-A**] [**-E***n=v*]... [**-K***pair*] [**-N**] [**-p***pamsrv*] [**-U***user*]... [**-a***accessxml*]

hvr -s*lbl* [**-E***n=v*]...

hvr -x -a*accessxml* [**-E***n=v*]... [**-K***pair*]

Description

Command **hvr** is an interpreter for HVR's internal script language. These scripts are generated by HVR itself. Inspection of these scripts can improve transparency and assist debugging, but it is unnecessary and unwise to use the internal script language directly because the syntax is liable to change without prior notice between HVR versions.

If no arguments are supplied or the first argument is '**-**' then input is read from **stdin**. Otherwise *script* is taken as input. If *script* begins with '**.**' or '**/**' it is opened as an absolute pathname, otherwise a search for the hvr script is done in the current directory '**.**' and then in **$HVR_HOME/script**.

Command **hvr** with option **-r** is used to provide an HVR child process on a remote machine. Its validation of passwords at connection time is controlled by options **-A**, **-p**, **-N** and **-U**.

Command **hvr** with option **-x** is used to provide an HVR proxy. For more information, see **Hvrproxy**.

Options

This section describes the options available for command **hvr**.

Parameter	Description

-a*accessxml* **Unix & Linux**	Access control file. This is an XML file for remote connections (option **-r**) and proxy mode (option **-x**) which controls from which nodes connections will be accepted, and also the encryption for those connections. To enable 2-way SSL authentication the public certificate of the hub should be given with XML **<ssl remote_cert="***mycloud***"/>** inside the **<from/>** element of this access control file. Also the public certificate private key pair should be defined on the hub with **LocationProperties /SslLocalCertificateKeyPair**. In proxy mode (option **-x**) this option is mandatory and is also used to control to which nodes connections can be made using XML **<to/>** tags. If *accessxml* is a relative pathname, then the file should be in **$HVR_HOME/lib** and if a SSL certificate is a relative pathname then the file should be in **$HVR_HOME/lib/cert**.
-A **Unix & Linux**	Remote HVR connections should only authenticate login/password supplied from hub, but should not change from the current operating system username to that login. This option can be combined with the **-p** option (PAM) if the PAM service recognizes login names which are not known to the operating system. In that case the daemon service should be configured to start the HVR child process as the correct operating system user (instead of **root**).
-E*n=v*	Set environment variable *n* to value *v* for this process and its children.
-K*pair* **Unix & Linux**	SSL public certificate and private key of local machine. This should match the hub's certificate supplied by **/SslRemoteCertificate**. If *pair* is relative, then it is found in directory **$HVR_HOME/lib/cert**. Value *pair* specifies two files; the names of these files are calculated by removing any extension from *pair* and then adding extensions **.pub_cert** and **.priv_key**. For example, option **-Khvr** refers to files **$HVR_HOME/lib/cert/hvr.pub_cert** and **$HVR_HOME/lib/cert/hvr.priv_key.**
-N **Unix & Linux**	Do not authenticate passwords or change the current user name. Disabling password authentication is a security hole, but may be useful as a temporary measure. For example, if a configuration problem is causing an 'incorrect password' error, then this option will bypass that check.
-p*pamsrv* **UNIX & Linux**	Use Pluggable Authentication Module *pamsrv* for login password authentication of remote HVR connections. PAM is a service provided by several operation systems as an alternative to regular login/password authentication, e.g. checking the **/etc/passwd** file. Often **-plogin** will configure HVR child process to check passwords in the same way as the operating system. Available PAM services can be found in file **/etc/pam.conf** or directory **/etc/pam.d**.
-r	HVR child process to service remote HVR connections. On Unix/Linux, the **hvr** executable is invoked with this option by the configured daemon. On Windows, **hvr.exe** is invoked with this option by the HVR Remote Listener Service. Remote HVR connections are authenticated using the login/password supplied for the **connect to HVR on a remote machine** information in the location dialog window.
-s*lbl*	Add label *lbl* to HVR's internal child co-processes. HVR sometimes uses child co-processes internally to connect to database locations. Value *lbl* has no effect other than to appear next to the process id in the process table (e.g. from **ps -ef**) so that users can distinguish between child co-processes.
-t*x*	Timestamp prefix for each line. Value *x* can be either **s** (which means timestamps in seconds) or **n** (no timestamp). The default is to only prefix a timestamp before each output line if **stderr** directs to a TTY (interactive terminal).

-U*user*	Limits the HVR child process to only accept connections which are able to supply operating system password for account *user*. This reduces the number of passwords that must be kept secret. Multiple **-U** options can be supplied.
-x	HVR proxy mode. In this mode the HVR process will accept incoming connections a reconnect through to other nodes. This requires option **-a**. For more information, see section **Hvrproxy**.

Example

To run hvr script **foo** with arguments **-x** and **bar** and to redirect **stdout** and **stderr** to file log:

```
$ hvr foo -x bar >log 2>&1
```

Custom HVR Password Validation

When **hvr** is used for remote connections (option **-r**) it must validate passwords. This can be customized if an executable file is provided at **$HVR_HOME/lib/hvrvalidpw**. HVR will then invoke this command without arguments and will supply the login and password as **stdin**, separated by spaces. If **hvrvalidpw** returns with exit code **0**, then the password is accepted.

A password validation script is provided in **$HVR_HOME/lib/hvrvalidpw_example**. This script also has options to manage its password file **$HVR_HOME/lib/hvrpasswd**. To install custom HVR password validation,

1. Enable custom password validation.

   ```
   $ cp $HVR_HOME/lib/hvrvalidpw_example $HVR_HOME/lib/hvrvalidpw
   ```

2. Add option **-A** to **Hvrremotelistener** or to the **hvr -r** command line. This prevents an attempt to change the user. Also change **Hvrremotelistener** or the daemon configuration so that this service runs as a non-root user.
3. Add users to the password file **hvrpasswd**.

   ```
   $ $HVR_HOME/lib/hvrvalidpw newuser # User will be prompted for password
   $ $HVR_HOME/lib/hvrvalidpw -b mypwd newuser # Password supplied on command line
   ```

Files

HVR_HOME	
bin	
hvr	HVR executable (Unix and Linux).
hvr.exe	HVR executable (Windows).
hvr_ii*N*.dll	Ingres version *N*shared library (Windows).
hvr_or*N*.dll	Oracle version *N*shared library (Windows).
hvr_ms*N*.dll	SQL Server version *N*shared library (Windows).
lib	
cert	
hvr.priv_key	Default SSL encryption private key, used if **hvr**is supplied with option **-Chvr** or **-Khvr** (instead of absolute path). Must be created with command **hvrsslgen**.
hvr.pub_cert	Default SSL encryption public certificate, used if **hvr**is supplied with option **-Chvr** or **-Khvr**or **/SslRemoteCertficate=hvr** (instead of absolute path). Must be created with command **hvrsslgen**.

ca-bundle.crt	Used by HVR to authenticate SSL servers (FTPS, secure WebDAV, etc). Can be overridden by creating new file *host*.**pub_cert** in this same certificate directory. No authentication done if neither file is found. So delete or move both files to disable FTPS authentication. This file can be copied from e.g. **/usr/share/ssl/certs/ca-bundle.crt** on Unix/Linux.
host.**pub_cert**	Used to override **ca-bundle.crt** for server verification for *host*.
hvr_ii*N*.**sl** *or*.**so**	Ingres shared library (Unix and Linux).
hvr_or*N*.**sl** *or* .**so**	Oracle shared library (Unix and Linux).
hvrpasswd	Password file employed by **hvrvalidpwfile**.
hvrvalidpw	Used by HVR for user authentication.
hvrvalidpwfile	The plugin file for private password file authentication.
hvrvalidpwldap	The plugin file for LDAP authentication.
hvrvalidpwldap.conf	Configuration for LDAP authentication plugin.
hvrvalidpwldap.conf_example	Example configuration file for LDAP authentication plugin.
hvrscripthelp.html	Description of HVR's internal script syntax and procedures.
HVR_CONFIG	
job	HVR scripts generated by **hvrinit**.
files	
[*hubnode*]-*hub-chn-loc*.**logrelease**	Status of HVR log-based capture jobs, for command **hvrlogrelease**.
tmp	Temporary files if **$HVR_TMP** is not defined.
HVR_TEMP	Temporary files for sorting and large objects.

Hvradapt

Contents

Name

hvradapt - Explore base table definitions in the database(s) and adapt them into channel information

Synopsis

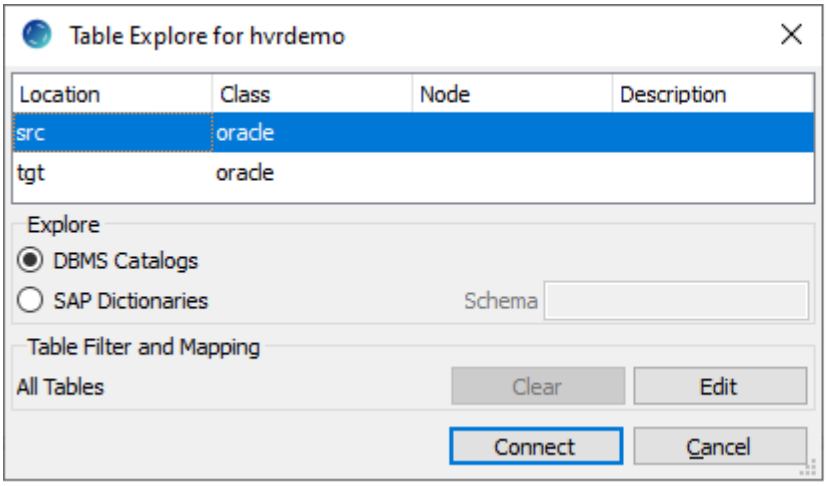
hvradapt [*-options*] **-l***oc hubdb chn*

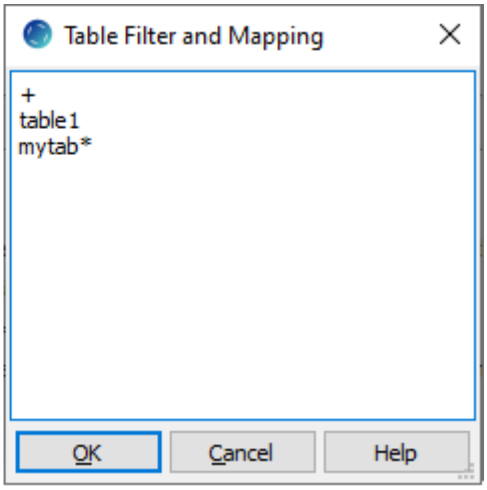
Description

Command **hvradapt** compares the base tables in the database with the table information for a channel. It will then either add, replace or delete table information in the catalog tables (**hvr_table** and **hvr_column**) so this information matches.

- If the location (**-l***oc*) from where the **hvradapt** explores the base table definitions contains a table which is not present in the channel but is matched by the table filter statement then it is added to the channel.
- If a table is in the channel but is not matched by the table filter statement then it is deleted from the channel.
- If a table is both matched by the table filter statement and included in the channel, but has the wrong column information in the channel, then this column information is updated.
- If table filter statement is not supplied (no **-n** or **-N** option), then tables are not added or deleted; only existing column information is updated where necessary.

Hvradapt is equivalent to the "**Table Explore**" dialog along with the **Table Filter** dialog in HVR GUI.





Options

This section describes the options available for command **hvradapt**.

Parameter	Description
-d*tblname...*	Delete specified table from channel. No other tables are compared or changed.
-f*fname*	Write (append) list of modified or added HVR table names to file *fname*. This can be useful in a script which calls **hvradapt** and then does extra steps (e.g. **hvrrefresh**) for tables which were affected (see example below).
-h*class*	Location class of the hub database. For valid values and format for specifying *class*, see Calling HVR on the Command Line.
-i*letters*	Ignore certain differences. Value *letters* can contain: • **c** - Column was dropped. • **C** - Column was added. • **d** - Data type changed. • **D** - Data type family changed. • **f** - Column range became smaller. • **F** - Column range became bigger. • **h** - Distribution key removed. • **H** - Distribution key added. • **k** - Unique index removed. • **K** - Unique index added. • **n** - Nullability removed. • **N** - Nullability added. • **r** - Column was renamed. • **s** - Encoding changed.
-I	Controls whether HVR should convert catalog data types into data types that could be created in the DBMS. If not supplied then the data types are converted before they are compared. Otherwise the actual catalog data type is compared without any conversion.
-l*loc*	Specifies the adapt location *loc*, typically the channel's capture location.
-n*tablefilterfile*	Specifies a table filter file *tablefilterfile* for the channel. This file can contain 'table filter' statement(s) to define which base tables in the database should be included (or excluded) in the channel. The *tablefilterfile* can contain names of the schema, table, column, and/or a pattern (such as **mytbl***). Multiple table filter statements can be supplied in HVR GUI and CLI. For more information, see section Table Filter. In HVR GUI, the contents in table filter file can only be copy pasted into the **Table Filter** dialog (click **Edit** in the **Table Explore** dialog).
-N*tablefilterstmt*	Specifies a table filter statement (pattern) *tablefilterstmt*. This statement defines which base tables in the database should be included (or excluded) in the channel. The *table filterstmt* can contain names of the schema, table, column, and/or a pattern (such as **m ytbl***). Multiple table filter statements can be supplied in HVR GUI and CLI. For more information, see section Table Filter. In HVR GUI, to specify the table filter statement, click **Edit** in the **Table Explore** dialog. In CLI, the table filter statement can also be supplied in a table filter file using option **-n**.
-R	Do not re-describe tables.
-r*tbls*	Re-describe only specific tables *tbls*.

-s*scope*	Add **TableProperties /Schema** to *scope.* Valid values for *scope* are: • **a** (**default**) - Add action to all (*) location(s). • **g** - Add action onto location's group • **l** - Add action onto the specific location).
-S*scope*	Add **ColumnProperties /Absent** to *scope* instead of updating the column information. The **default** is not to add any **ColumnProperties /Absent** to *scope*, but instead to delete the column information from the channel. This affects how the channel is changed when a column does not exist in the database but exist in the channel. If this option is supplied, a **ColumnProperties /Absent** is created. Valid values for *scope* are: • **g** - Add action onto location's group. • **l** - Add action onto the specific location.
-u*user*[*/pwd*]	Connect to hub database using DBMS account *user*. For some databases (e.g. SQL Server) a password must also be supplied.
-U*dictsch*	SAP dictionary tables are owned by DB schema *dictsch*. If this option is not supplied then HVR will look for the SAP dictionary tables in the HVR location's default schema, and also in schemas indicated in any **TableProperties /Schema** parameter. This option can only be used with option **-X**.
-V **Since** v5.6.0/0	Show views and materialized views. For Oracle, the materialized views are always shown.
-x	Check only mode. Do not apply any changes to the catalogs.
-X	Explore table and column information from SAP dictionaries, instead of DBMS catalogs. For SAP "transparent" tables this can give more correct data type information. It will also show the logical tables inside SAP "cluster" and "pool" tables. These are shown as 'unpacked' tables.

Table Filter

Hvradapt supplied with table filter statement (option **-n** or **-N**) allows you to define which base tables in the adapt location should be included in or excluded from the channel. Only tables matching any of the given statement will be included or excluded.

Syntax for Table Filter

This section describes the syntax for the classic table filter statement that can be used with options **-n** or **-N**.

```
[schema.]tablename [-T target_schema] [-K (col_list)]
```

```
![schema.]tablename or
```

```
+
```

- Value *schema* or *tablename* can be a literal (optionally enclosed in double quotes) or a pattern matching can be done (only for tables or columns) using the special symbols *****, **?** or **[***characters***]**.
- Option **-K** marks the listed columns as distribution key columns in the 'column catalog', instead of defining a new **ColumnProperties /DistributionKey** action. In HVR GUI, marking a column as distribution key can be done from the table **Properties** dialog. For more information, see Marking a Column as Distribution Key in HVR GUI.
- Option **-T** defines the target schema into which the table should be replicated. **Hvradapt** will automatically define a new **TableProperties /Schema** action in this case.
- Options **-K** and **-T** only have an effect at the moment a table is added to channel, but are ignored otherwise.
- Special symbol **!** (NOT) is used to define negative patterns. This type of pattern can only be used after a regular /positive pattern and therefore cannot be used as an orphan (without other patterns) or the first pattern in the adapt template. Tables matching the preceding pattern and the negative pattern are excluded from the channel. For example,

```
*                # Match all tables in default schema.
!tmp_*              # Exclude all tables whose name begin with 'tmp_' in default
schema.
```

- Special symbol **+** matches all tables already in the channel.

Empty lines and comments (e.g. **#Test**) are ignored.

Example for Table Filter

Hvradapt can filter tables using a table filter file (option **-n**) or on the command line (option **-N**).

- Examples for filtering tables or schemas.
  The following is an example of table filter file (e.g. **/tmp/adapt.tmpl**):

```
+                      # Match all tables already in channel.
tbl1                   # Match table named 'tbl1' in default schema.
schema1.*              # Match all tables in 'schema1'.
schema2.tbl1           # Match table named 'tbl1' in 'schema2'.
history_*                 # Match all tables whose name begin with 'history_' in
default schema.
"my table"             # Match table named "my table" in default schema. Use double
quotes if there is a space in table name.
schema2."tbl.tab*"        # Match all tables whose name begin with 'tbl.tab' in
'schema2'. Use double quotes if there is a special character in the table name.
schema3.xx* -T schema4  # Match all tables whose name begin with 'xx' in schema3 for
replicating them to schema4.
tbl2 -K(col1 col2)     # Match table named 'tbl2' in default schema and add columns
named 'col1' and 'col2' as distribution key columns.
```

  This table filter pattern can be supplied in the command line as:

```
hvradapt -N + -N tbl1 -N schema1.* -N schema2.tbl1 -N history_* -N "my table" -N
schema2."tbl.tab*" -N schema3.xx* -T schema4 -N tbl2 -K(col1 col2) -l mylocation
hvrhub/hvrhub hvrdemochn
```

A shell script as shown in section Shell Script to Run Hvradapt can be created to run **hvradapt** for checking new or modified tables in a location.

Shell Script to Run Hvradapt

A shell script can be created to run **hvradapt** for checking new or modified tables in a location.

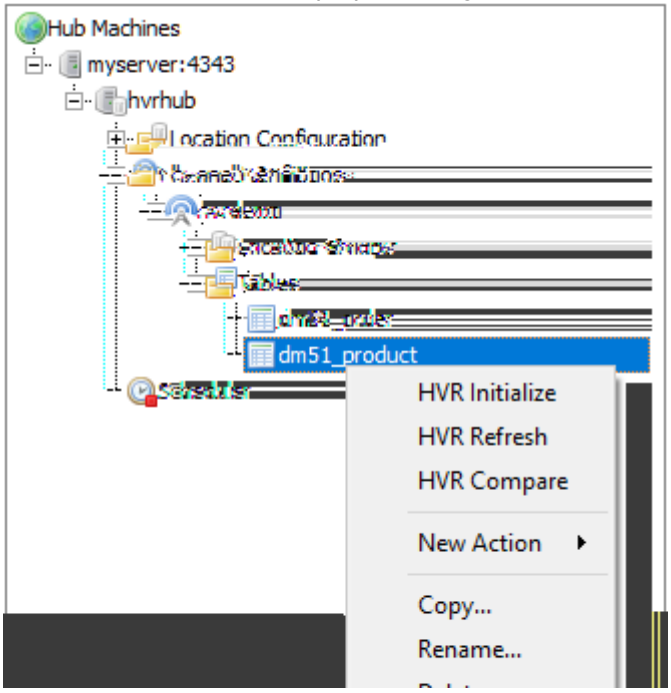
The following example demonstrates the use of a shell script to run **hvradapt** for checking new or modified tables in location **loc1** and if any new or modified tables are found in location **loc1**, the script executes the necessary commands to enroll the tables into the channel **mychn**.

```
#!/bin/sh
hub=myhub/passwd                                        # Hub database and password (if
required)
chn=mychn                                              # Channel
src=loc1                                               # Source location
F=/tmp/adapt_$chn.out                                   # File where hvradapt writes list of
new or changed tables
hvradapt -f$F -n/tmp/adapt.tmpl -l$src $hub $chn     # Add new or changed tables from
source to channel based on the patterns defined in the pattern file.
if test -f $F                                           # If file $F exists then new or
changed tables were detected
then
    hvrsuspend $hub $chn-integ                         # Stop integrate jobs
    hvrinit -oelj $hub $chn                            # Regenerate supplemental-logging,
jobs and enroll info
    hvrrefresh -r$src -t$F -qrw -cbkr $hub $chn     # Re-create and online refresh tables
in file $F
    hvrstart -r -u $hub $chn-inget                     # Re-trigger stopped jobs
    hvradapt -x $hub $chn                              # Double-check channel now matches
target location (optional)

    rm $F                                              # Remove file with list of new or
changed tables
fi
```

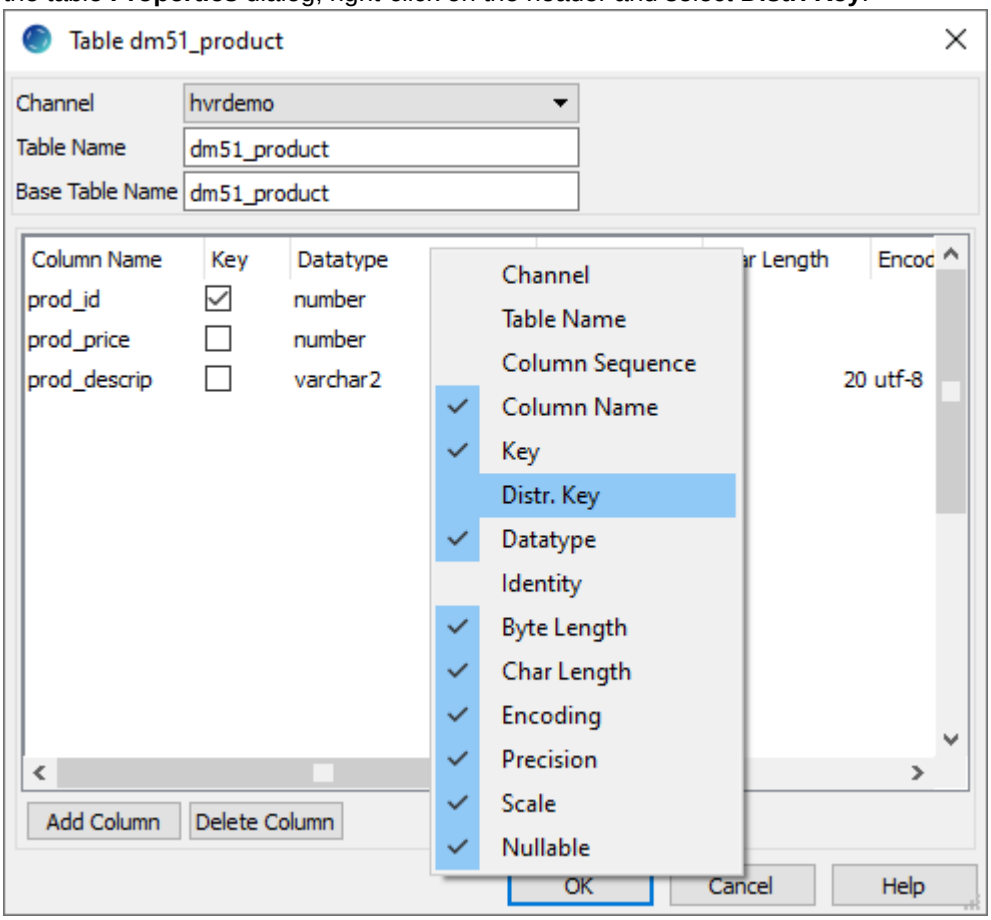
Marking a Column as Distribution Key in HVR GUI

Perform the following steps to mark a column as distribution key in HVR GUI:

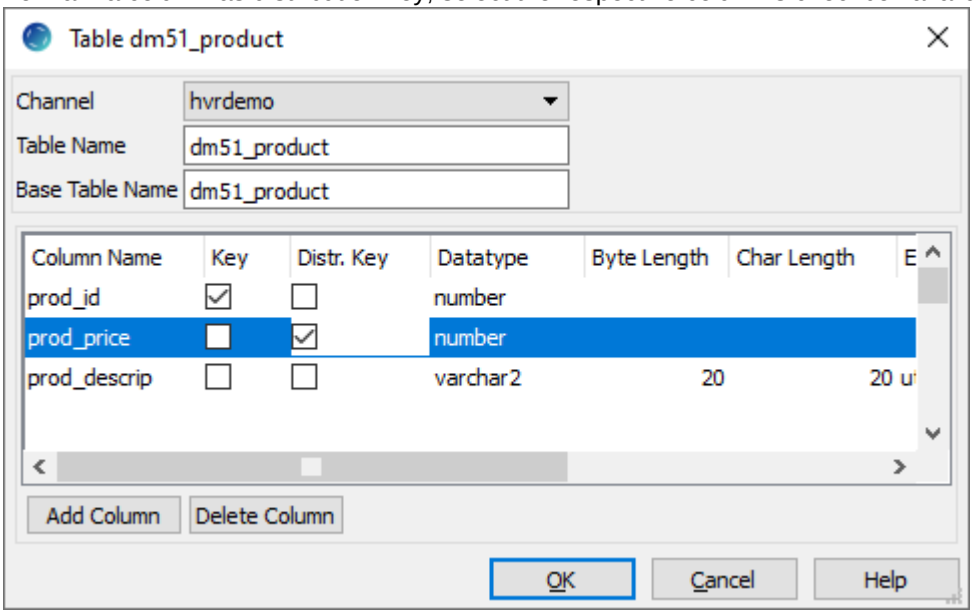
1. To view a table's details or properties, right-click on a table in the channel and select **Properties**.



2. By default, the distribution key column is not displayed in the table **Properties** dialog. To display this column in the table **Properties** dialog, right-click on the header and select **Distr. Key**.



3. To mark a column as distribution key, select the respective column's checkbox available under **Distr. Key**.



Hvrcatalogcreate, Hvrcatalogdrop

Contents

Name

hvrcatalogcreate - Create catalog tables in hub database.

hvrcatalogdrop - Drop catalog tables from hub database.

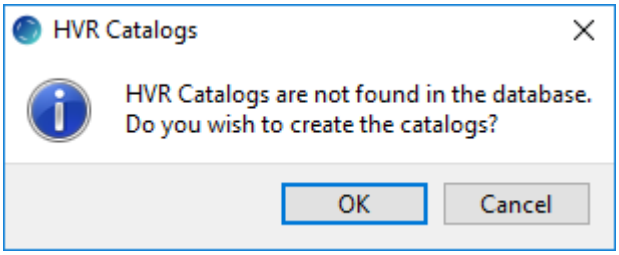
Synopsis

hvrcatalogcreate [*options*] *hubdb*

hvrcatalogdrop [*options*] *hubdb*

Description

Command **hvrcatalogcreate** allows you to create the catalog tables in hub database. This is equivalent to the **HVR Catalogs** dialog in the HVR GUI asking to create catalog tables in the HVR hub database. This prompt is displayed only if the catalog tables are not found in the hub database.



Command **hvrcatalogdrop** allows you to drop the catalog tables from hub database. Executing this command removes all locations, channels (including location groups and tables), and all actions defined on the channel or location. It is recommended to backup the catalogs (using **hvrcatalogexport**) before executing this command.

The argument *hubdb* specifies the connection to the hub database. For more information about supported hub databases and the syntax for using this argument, see Calling HVR on the Command Line.

Options

This section lists and describes all options available for **hvrcatalogcreate** and **hvrcatalogdrop**.

-h*class*	Specify hub database. For valid values for class, see Calling HVR on the Command Line.
-u*user*[**/***pwd*]	Connect to hub database using DBMS account *user*. For some databases (e.g. SQL Server) a password must also be supplied.

Hvrcatalogexport, hvrcatalogimport

Contents

Name

hvrcatalogexport - Export from hub database into HVR catalog document.

hvrcatalogimport - Import from HVR catalog document into hub database.

Synopsis

hvrcatalogexport [**-c***chn*...] [**-C**] [**-d**] [**-g**] [**-h***class*] [**-l**] [**-u**user] *hubdb catdoc*

hvrcatalogimport [**-h**class] [**-u***user*] *hubdb catdoc*

Description

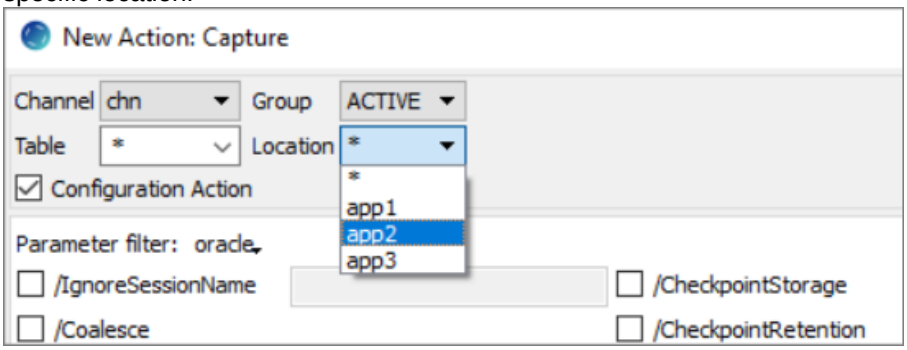
Command **hvrcatalogexport** extracts information from the HVR catalog tables in the hub database and writes it into file *catdoc*. HVR catalog document *catdoc* is an XML file which follows the HVR catalog Document Type Definition (DTD).

You can export the information about:

- Locations (see the **hvr_location** catalog table)
- Channel Definitions (see the **hvr_channel**, **hvr_table**, **hvr_column**, **hvr_loc_group** and **hvr_action** catalog tables)
- Group Membership (see the **hvr_loc_group_member** catalog table)
- Configuration Action (see the **hvr_config_action** catalog table).

Configuration Action

Configuration Actions are the actions defined at a location level. In HVR GUI, when an action is defined on a location (by right-clicking the location), the option **Configuration Action** is automatically selected in the **New Action** dialog. However, when an action is defined for a channel, location group or table (by right-clicking the channel, location group or table), the option **Configuration Action** needs to be manually selected in the **New Action** dialog to apply this action to a specific location.



You can select to export information from all the catalog tables or separately for each of the items from the above list.

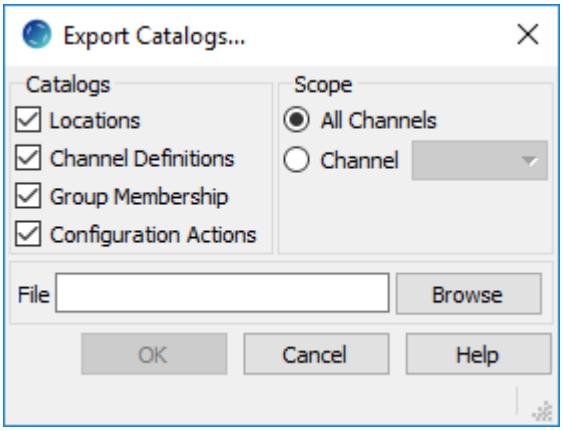
Command **hvrcatalogimport** loads the information from the supplied HVR catalog document into the HVR catalogs in the hub database.

The first argument *hubdb* specifies the connection to the hub database. For more information about supported hub databases and the syntax for using this argument, see Calling HVR on the Command Line.

An HVR catalog document file can either be created using command **hvrcatalogexport** or it can be prepared manually, provided it conforms to the HVR catalog DTD.

Options

This section describes the options available for the commands **hvrcatalogexport** and **hvrcatalogimport**.



Parameter	Description
-c*chn*	Only export catalog information for channel *chn*.
-C	Only export information from configuration action catalog **hvr_config_action**.
-d	Only export information from channel definition catalogs **hvr_channel**, **hvr_table**, **hvr_column**, **hvr_loc_group** and **hvr_action**.
-g	Only extract information from group membership catalog **hvr_loc_group_member**.
-h*class*	Specify hub database. Valid values are **oracle**, **ingres**, **sqlserver**, **db2**, **db2i**, **postgresql**, and **teradata**. See also section Calling HVR on the Command Line.
-l	Only export information from location catalog **hvr_location**.
-u*user*[*/pwd*]	Connect to hub database using DBMS account *user*. For some databases (e.g. SQL Server) a password *pwd* must also be supplied.

HVR Catalog DTD

HVR catalog documents are XML files that must conform to the HVR catalog Document Type Definition (DTD). A formal specification of this DTD can be found in file **HVR_HOME/lib/hvr_catalog.dtd**. Most XML tags in the DTD are directly equivalent to a table or row of the HVR Catalog Tables.

The root tag of the HVR catalog DTD is **<hvr_catalog>**. This root tag contains "table" tags named **<hvr_channels>**, **<hvr_tables>**, **<hvr_columns>**, **<hvr_loc_groups>**, **<hvr_actions>**, **<hvr_locations>**, **<hvr_loc_group_members>** and **<hvr_config_actions>**. Most table tags contain a special optional attribute **chn_name**. This special attribute controls the amount of data that is deleted and replaced as the HVR catalog document is loaded into the catalog tables. For example, a document that contains **<hvr_actions chn_name="hvr_demo01">** would imply that only rows for channel **hvr_demo01** should be deleted when the document is imported. If the special attribute **chn_name** is omitted then all rows for that catalog table are deleted.

Each table tag contains tags that correspond to rows in the catalog tables. These 'row' tags are named **<hvr_channel>**, **<hvr_table>**, **<hvr_column>**, **<hvr_loc_group>**, **<hvr_action>**, **<hvr_location>**, **<hvr_loc_group_member>** and **<hvr_config_action>**. Each of these row tags has an attribute for each column of the table. For example, tag **<hvr_tables>** could contain many **<hvr_table>** tags, which would each have attributes **chn_name**, **tbl_name** and **tbl_base_name**.

Some attributes of a row tag are optional. For example, if attribute **col_key** of **<hvr_column>** is omitted it defaults to **0** (false), and if attribute **tbl_name** of tag **<hvr_action>** is omitted then it defaults to '*****' (affect all tables).

Example

```
 <syntaxhighlight source lang="xml">
 <?xml version="1.0" encoding="UTF-8"?>
  <!DOCTYPE  hvr_catalog  SYSTEM  "<nowiki>http://www.hvr-software.com/dtd/1.0/hvr_catalog.
dtd</nowiki>">
 <hvr_catalog version="1.0">
  <hvr_channels chn_name="hvr_demo01">
    <hvr_channel chn_name="hvr_demo01" chn_description="Simple reference channel."/>
  </hvr_channels>
  <hvr_tables chn_name="hvr_demo01">
    <hvr_table chn_name="hvr_demo01" tbl_name="dm01_order" tbl_base_name="dm01_order"/>
     <hvr_table chn_name="hvr_demo01" tbl_name="dm01_product" tbl_base_name="dm01_product"
/>
  </hvr_tables>
  <hvr_columns chn_name="hvr_demo01">
           <hvr_column   chn_name="hvr_demo01"   tbl_name="dm01_order"   col_sequence="
1"                  col_name="prod_id" col_key="1" col_datatype="integer4"/>
           <hvr_column   chn_name="hvr_demo01"   tbl_name="dm01_order"   col_sequence="
2"                  col_name="ord_id" col_key="1" col_datatype="integer4"/>
           <hvr_column   chn_name="hvr_demo01"   tbl_name="dm01_order"   col_sequence="
3"                  col_name="cust_name" col_datatype="varchar" col_length="100"/>
           <hvr_column   chn_name="hvr_demo01"   tbl_name="dm01_order"   col_sequence="
4"                   col_name="cust_addr" col_datatype="varchar" col_length="100"
col_nullable="1"/>
          <hvr_column   chn_name="hvr_demo01"   tbl_name="dm01_product"   col_sequence="
1"                  col_name="prod_id" col_key="1" col_datatype="integer4"/>
          <hvr_column   chn_name="hvr_demo01"   tbl_name="dm01_product"   col_sequence="
2"                  col_name="prod_price" col_datatype="float8"/>
          <hvr_column   chn_name="hvr_demo01"   tbl_name="dm01_product"   col_sequence="
3"                  col_name="prod_descrip" col_datatype="varchar" col_length="100"/>
  </hvr_columns>
  <hvr_loc_groups chn_name="hvr_demo01">
    <hvr_loc_group chn_name="hvr_demo01" grp_name="CEN" grp_description="Headquarters"/>
    <hvr_loc_group chn_name="hvr_demo01" grp_name="DECEN" grp_description="Decentral"/>
  </hvr_loc_groups>
  <hvr_actions chn_name="hvr_demo01">
    <hvr_action chn_name="hvr_demo01" grp_name="CEN" act_name="DbCapture"/>
    <hvr_action chn_name="hvr_demo01" grp_name="DECEN" act_name="DbIntegrate"/>
  </hvr_actions>
 </hvr_catalog>
 </syntaxhighlight>
```

Files

- **HVR_HOME**
  - **demo**
    - **hvr_demo01**
      - **hvr_demo01_def.xml** — Catalog document for channel definition.
      - **hvr_demo01_cnf_gm_example.xml** — Catalog document for group membership.
      - **hvr_demo01_cnf_loc_oracle_example.xml** — Catalog document for HVR locations.
  - **HVR_HOME**
    - **hvr_catalog.dtd** — Catalog Document Type Definition.

Hvrcheckpointretention

Contents

Name

hvrcheckpointretention - Displays the checkpoint files.

Synopsis

hvrcheckpointretention [-*option*] *checkpointfilepath*

Description

Command **hvrcheckpointretention** displays the checkpoint files available in *checkpointfilepath* directory. Also, this command with option **-p** allows you to purge/delete the checkpoints available in *checkpointfilepath* directory.

Options

This section describes the options available for command **hvrcheckpointretention**.

Parameter	Description
-p*purge_secs*	Purge checkpoints older than the specified time in *purge_secs*. The format for *purge_secs* can be in any of the following time formats - **HH:mm:ss** or **mm:ss** or **ss** Checkpoints are purged only if there is a checkpoint newer than *purge_secs* available in *checkpointfilepath* directory. Example: **-p 86400** (purge checkpoints older than 86400 seconds) **-p 24:00:00** (purge checkpoints older than 24 hours)

Example

```
$ hvrcheckpointretention /hvr/hvr_config/capckpretain/myhub/mychannel/src
hvrcheckpointretention: HVR 5.5.6/0 (linux_glibc2.17-x64-64bit)
hvrcheckpointretention:  Found  2  checkpoints  in  /hvr/hvr_config/capckpretain/myhub
/mychannel/src suitable for a hvrinit re-initialize.
hvrcheckpointretention: Oldest checkpoint needs hvrinit rewind timestamp less than 2019-
05-17T11:45:49+02:00 (hvr_tx_seq=0x19eb29a00004), and emit timestamp after 2019-05-17T11:
45:49+02:00 (hvr_tx_seq=0x19eb29a20001).
hvrcheckpointretention: Most recent checkpoint needs hvrinit rewind timestamp less than
2019-05-17T11:46:46+02:00 (hvr_tx_seq=0x19eb2faf0001), and emit timestamp after 2019-05-
17T11:46:46+02:00 (hvr_tx_seq=0x19eb2fb10001).
```

Hvrcompare

Contents

Name

hvrcompare - Compare data in tables.

Synopsis

hvrcompare [*-options*] *hubdb chn*

Description

Command **hvrcompare** compares the data in different locations of channel *chn*. The locations must be databases, not file locations.

The first argument *hubdb* specifies the connection to the hub database. For more information about specifying value for *hubdb* in CLI and the supported hub databases, see Calling HVR on the Command Line.

HVR Compare Operation Type

Table compare can be performed as a bulk or row-by-row operation, depending on the option **-g** supplied with command **hvrcompare**.

Bulk Compare

During bulk compare, HVR calculates the checksum for each tables in the channel and compares these checksum to report whether the replicated tables are identical.

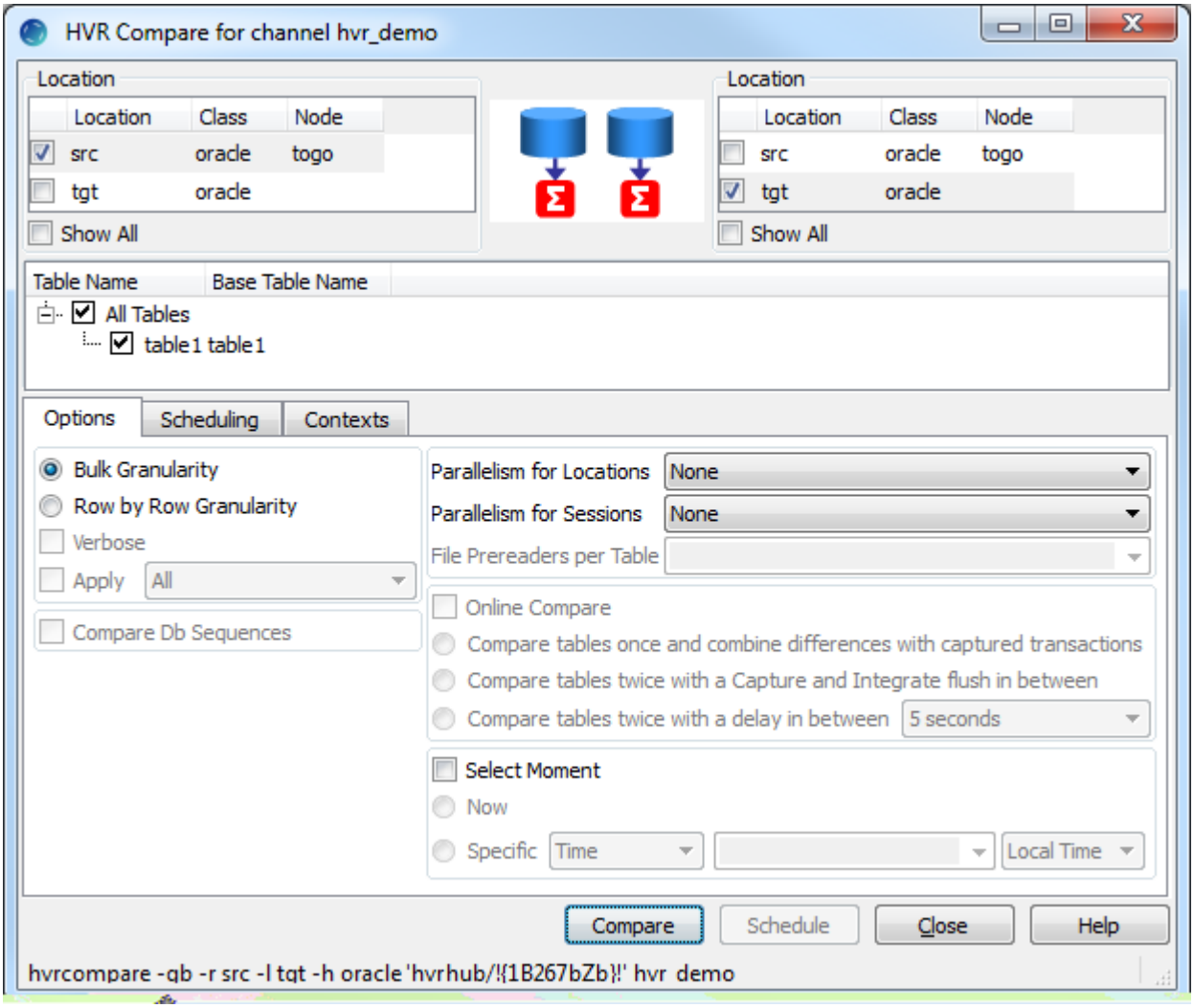
Row by Row Compare

During row by row compare, HVR extracts the data from a source (read) location, compresses it and transfers the data to a target (write) location(s) to perform row by row compare. Each individual row is compared to produce a 'diff' result. For each difference detected an SQL statement is written: an insert, update or delete.

If **hvrcompare** is connecting between different DBMS types, then an ambiguity can occur because of certain data type coercions. For example, HVR's coercion maps an empty string from other DBMS's into a **null** in an Oracle **varchar**. If Ingres location **ing** contains an empty string and Oracle location **ora** contains a **null**, then should HVR report that these tables are the same or different? Command **hvrcompare** allows both behaviors by applying the sensitivity of the 'write' location, not the 'read' location specified by option **-r**. This means that comparing from location **ing** to location **ora** will report the tables as identical, but comparing from **ora** to **ing** will say the tables are different.

Options

This section describes the options available for command **hvrcompare**.



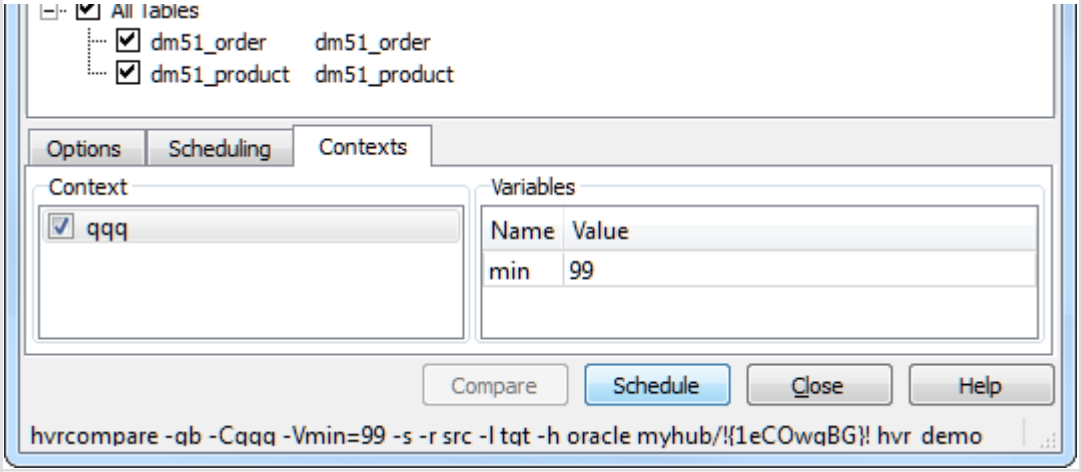
Parameter	Description

-C*context*

Enable *context*. This controls whether actions defined with parameter **Context** are effective or are ignored.

Defining an action with **Context** can have different uses. For example, if action **Restrict** **/CompareCondition**="**{id}>22" /Context=qqq** is defined, then normally all data will be compared, but if context **qqq** is enabled (**-Cqqq**), then only rows where **id>22** will be compared. Variables can also be used in the restrict condition, such as **"{id}>{hvr_var_min}"**. This means that **hvrcompare -Cqqq -Vmin=99** will compare only rows with **id>99**. To supply variables for restrict condition use option **-V**.

Parameter **/Context** can also be defined on action **ColumnProperties**. This can be used to define **/CaptureExpression** parameters which are only activated if a certain context is supplied. For example, to define a context for case-sensitive compares.



-D

Since v5.5.5/6

Duplicate an existing compare event. This option is used for repeating a compare operation, using the same arguments.

-d

Remove (drop) scripts and scheduler jobs & job groups generated by previous **hvrcompare** command.

When this option is used with option **-e** it cancels (**FAILED**) events that are in **PENDING** or **RUNNING** state. **Since** v5.5.5/6

-e

Since v5.5.5/6

Perform event driven compare. In event driven compare, the compare operation is managed using HVR's event system. For each compare operation performed, HVR maintains a record in the following catalog tables - **hvr_event** and **hvr_event_result**. In HVR GUI, the option to perform event driven compare (**Generate Compare Event**) is available in the **Scheduling** tab.

HVR creates a compare job in the **HVR Scheduler** and the compare operation is started under a compare event. While performing event driven compare, if a compare event with the same job name exists in **PENDING** or **RUNNING** s tate then it is automatically cancelled (**FAILED**) by the new compare event.

The results for event driven compare is displayed in Insight Events.

In HVR GUI, by default, the event driven compare is scheduled to **Start Immediately**. In CLI, to start the event driven compare immediately, execute the command **hvrstart** immediately after executing the command for event driven compare.

This option is also required to run the following compare operations:

1. Online compare. For more information, see option **-o** below.
2. Direct file compare. For more information, see section Direct File Compare.

-g*x*

Granularity of compare operation in database locations.

Valid values of *x* are:

- **b** (**default**): Bulk compare using checksums.
- **r** : Row-wise compare of tables.

-h*class*

Specify hub *class*, for connecting to hub database. For supported values, see Calling HVR on the Command Line.

-i*x*

Since v5.6.5/0

Retain and reuse intermediate files. This option allows you to retain the intermediate files that are generated during direct file compare. It also allows you to reuse the retained intermediate files that were generated earlier by a similar compare. A compare is similar if the location, channel, tables, and **Restrict** **/SliceCondition** (if any) are identical.

In HVR GUI, the default value for this option (**File Prereader Intermediate Files**) is **No reuse, delete afterwards**.

This option is applicable only for direct file compare.

Valid values of *x* are:

- **k**: Keep/retain intermediate files generated during this compare. In HVR GUI, this option is displayed as **No reuse, Keep afterwards**.
- **r**: Reuse the retained intermediate files that were generated earlier by a similar compare and after the compare operation is completed, the reused intermediate files and the intermediate files generated during this compare are deleted. In HVR GUI, this option is displayed as **Reuse old, delete afterwards**.
- **kr**: Reuse retained intermediate files and retain the intermediate files generated during this compare. In HVR GUI, this option is displayed as **Reuse old, keep afterwards**.

-l*srange*

Since v5.5.5/6

Compare event only perform subset of slices implied by **-S** (table slices) option.

This option is only allowed with options **-e** and **-S**.

Value *srange* should be a comma-separated list of one of the following:

- *N* : Only perform 'sub slices' number *N*. Note that these slices are numbered starting from zero.
- *N-M* : Perform from slices from *N* to *M* inclusive.
- *N-* : Perform from slices from *N* onwards.
- *-M* : Perform from slices from the first slices until slice *M*.

-j*num_jobs* **Since** v5.3.1/6

Sets **quota_run** compare job group attribute. It defines a number of jobs which can be run simultaneously. The option cannot be used without scheduling turned on (**-s**)

-l*x*

Target location of compare. The other (read location) is specified with option **-r**. If this option is not supplied then all locations except the read location are targets.

Values of *x* maybe one of the following:

- *loc* : Only location *loc*.
- *l1-l2* : All locations that fall alphabetically between *l1* and *l2* inclusive.
- !*loc* : All locations except *loc*.
- !*l1-l2* : All locations except for those that fall alphabetically between *l1* and *l2* inclusive.

Several **-l***x* instructions can be supplied together.

-m*mask*

Mask (ignore) some differences between the tables that are being compared.

Valid values of *mask* can be:

- **d** : Mask out delete differences.
- **i** : Mask out insert differences.
- **u** : Mask out update differences.

Letters can be combined, for example **-mid** means mask out inserts and deletes. If a difference is masked out, then the verbose option (**-v**) will not generate SQL for it. The **-m** option can only be used with row-wise granularity (option **-gr**).

-M*moment*

Select data from each table of source from same consistent *moment* in time.

Value *moment* can be one of the following:

- *time* : Flashback query with **select … as of timestamp**. Valid formats are *YYYY-MM-DD [HH:MM:SS]* (in local time) or *YYYY-MM-DDTHH:MM:SS+TZD* or *YYYY-MM-DDTHH:MM:SSZ* or **today** or **now**[[+|-]*SECS*] or an integer (seconds since **1970-01-01 00:00:00 UTC**). Note that if a symbolic time like **-Mnow** is supplied then a new "SCN time" will be retrieved each time the compare job is run (not only when the **hvrcompare** command is called. So if **hvrcompare -Mnow** is run on Monday, and the compare job it creates starts running at 10:00 Tuesday and runs again 10:00 on Wednesday, then the first compare will do a flashback query (for all tables) with an SCN corresponding to Tuesday at 10:00 and the second job run will use flashback query with an SCN corresponding to Wednesday at 10:00.
- **scn**=*val* : Flashback query with **select … as of scn**. Value is an Oracle SCN number, either in decimal or in hex (when it starts with **0x** or contains hex digits).
- **hvr_tx_seq**=*val* : Value from HVR column **hvr_tx_seq** is converted back to an Oracle SCN number (by dividing by **65536**) and used for flashback query with **select … as of scn**. Value is either in decimal or in hex (when it starts with **0x** or contains hex digits).

-n*numtabs*

Since v5.3.1/6

Create 'sub-jobs' which each compare a bundle of no more than *numtabs* tables. In HVR GUI, this option is displayed as **Limit Tables per Job** in the **Scheduling** tab.

For example, if a channel contains 6 tables then option **-n1** will create 6 jobs whereas were option **-n4** to be used on the same channel then only 2 jobs will be created (the first with 4 tables, the last with just 2). If tables are excluded (using option -t) then these will not count for the bundling.

Jobs are named by adding a number (starting at 0) to the task name which defaults **cmp** (although the task name can always be overridden using option -T). Normally the first slice's job is named **chn-cmp0-x-y** but numbers are left-padded with zeros, so if 10 slices are needed the first is named **chn-cmp00-x-y** instead.

One technique is to generate lots of jobs for compare of big channel (using this option and option **-s**) and add 'scheduler attribute' **quota_run** to the job group (named **CHN-CMP**) so that only a few (say 3) can run simultaneously. Scheduler attributes can be added by right-clicking on the job group and selecting **Add Attribute**.

Another technique to manage the compare of a channel with thousands of tables is use this option along with options **-R** (ranges) and **-T** (task name) to do 'power of ten' naming and bundling, in case a single table encounters a problem. The following illustrates this technique;

First use **[-n100]** so each job tries to compare 100 tables.

If one of these jobs fails (say job **chn-cmp03-x-y**) then use options **[-n10 -R30-39 -Tcmp03]** to replace it with 10 jobs which each do 10 tables.

Finally if one of those jobs fail (say **chn-cmp037-x-y**) then then use options **[-n1 -R370-379 -Tcmp037]** to replace it with 10 'single table' jobs.

-N*secs* **Since** v5.5.5/8	Compare tables twice with a delay in between. In CLI, this option can only be used along with option **-o diff_diff**. Capture and Integrate jobs are not required for performing this mode of online compare. This online compare mode is similar to **Compare tables twice with a Capture and Integrate flush in between** (option **-o diff_diff**) however, with one difference. HVR performs a regular compare which produces a result (also known as diff). HVR then waits for *secs* seconds after which it again performs the regular compare which produces a second result (diff). The compare results generated in the first and second compare are combined to produce a final compare result. Example: **hvrcompare -gr -o diff_diff -N 5 -e -r src -l tgt -h oracle 'myhub/myhub' mychannel**
-o*mode* **Since** v5.5.5/8	Online compare. Performs live compare between locations where data is rapidly changing. This option can only be used if **Row by Row Granularity** (option **-gr**) and **Generate Compare Event** (option **-e**) is selected (or supplied). The results of online compare are displayed in Insight Events. Value *mode* can be either: • **diff_cap** (**default** in HVR GUI): Compare tables once and combine differences with captured transactions. Performs a regular compare which produces a result (also known as diff). This result (diff) is compared with the transaction files (which are continuously created by the capture job) to identify and remove the pending transaction from result (diff) for producing the final compare result. This compare mode is supported for comparing databases and files. • **diff_diff**: Compare tables twice with a Capture and Integrate flush in between. Performs a regular compare which produces a result (also known as diff). HVR then waits for the completion of a full Capture and Integrate cycle after which it again performs the regular compare which produces a second result (diff). The compare results generated in the first and second compare are combined to produce a final compare result. This compare mode is only supported for comparing databases. If **Compare tables twice with a delay in between** (option **-N***secs*) is selected (or supplied with **diff_diff**), HVR waits for a fixed amount of time (seconds) defined, instead of waiting for the completion of Capture and Integrate cycle in between compares. For more information, see option **-N***secs* above. Capture and Integrate jobs are not required for performing this mode of online compare. If a running **diff_cap** online compare job is suspended/deleted, HVR will continue to accumulate tx files with each capture cycle, which will lead to higher disk space usage. For the list of pending events associated with the compare job, use **hvreventview**. To cancel the job, use **hvreventtool**. Scheduling a new event driven compare with the same task name will also cancel the previous event automatically.

-O
Since v5.3.1/6
	Only show OS command implied by options **-n** (jobs for bundles of tables) or **-S** (table slices), instead of executing them. This can be used to generate a shell script of 'simpler' **hvrcompare** commands;
	
	For example if channel only contains tables **tab1**, **tab2**, **tab3** and **tab4** then this command;
	
	`hvrcompare -rcen -O -n2 myhub/pwd mychn`
	
	will only generate this output;
	
	`hvrcompare -rcen -Tcmp0 -ttab1 -ttab2 myhub/pwd mychn`
	
	`hvrcompare -rcen -Tcmp1 -ttab3 -ttab4 myhub/pwd mychn`

-p*N*	Parallelism for Locations. Perform compare on different locations in parallel using *N* sub-processes. This cannot be used with option **-s**.
-P*M*	Parallelism for Sessions. Perform compare for different tables in parallel using *M* sub-processes. The compare will start processing *M* tables in parallel; when the first of these is finished the next table will be processed, and so on.
-Q	No compare of database sequences matched by action **DbSequence**. If this option is not specified, then the database sequence in the source database will be compared with matching sequences in the target database. Sequences that only exist in the target database are ignored.
-r*loc*	Read location. This means that location *loc* is passive; the data is piped from here to the other location(s) and the work of comparing the data is performed there instead.
-R*rangeexpr* **Since** v5.3.1/6	Only perform certain 'sub jobs' implied by either options **-N** (job for bundles of tables) or **-S** (table slices). This option cannot be used without one of those options. Value *rangeexpr* should be a comma-separated list of one of the following: • *N* : Only perform 'sub job' number *N*. Note that these jobs are numbered starting from zero (e.g the first is *chn*-**cmp0**-*rloc*-*wloc*). • *N*-*M* : Perform from jobs from *N* to *M* inclusive. • *N*- : Perform from jobs from *N* onwards. • -*M* : Perform from jobs from the first job until job *M*. For example, if a channel contains 20 tables then option **-n1** would cause 20 jobs to be created (with names **chn-cmp00-x-y**, **chn-cmp01-x-y**, **chn-cmp02-x-y… chn-cmp19-x-y**) but options **-n1 -R0,10** would restrict job creation to only 11 jobs (named **chn-cmp00-x-y**, then **chn-cmp10-x-y**, **chn-cmp11-x-y … chn-cmp19-x-y**).

-s

Schedule invocation of compare scripts using the **HVR Scheduler**. In HVR GUI, this option is displayed as **Schedule Classic Job** in the **Scheduling** tab.

Without this option the **default** behavior is to perform the compare immediately (in HVR GUI, **Run Interactively**).

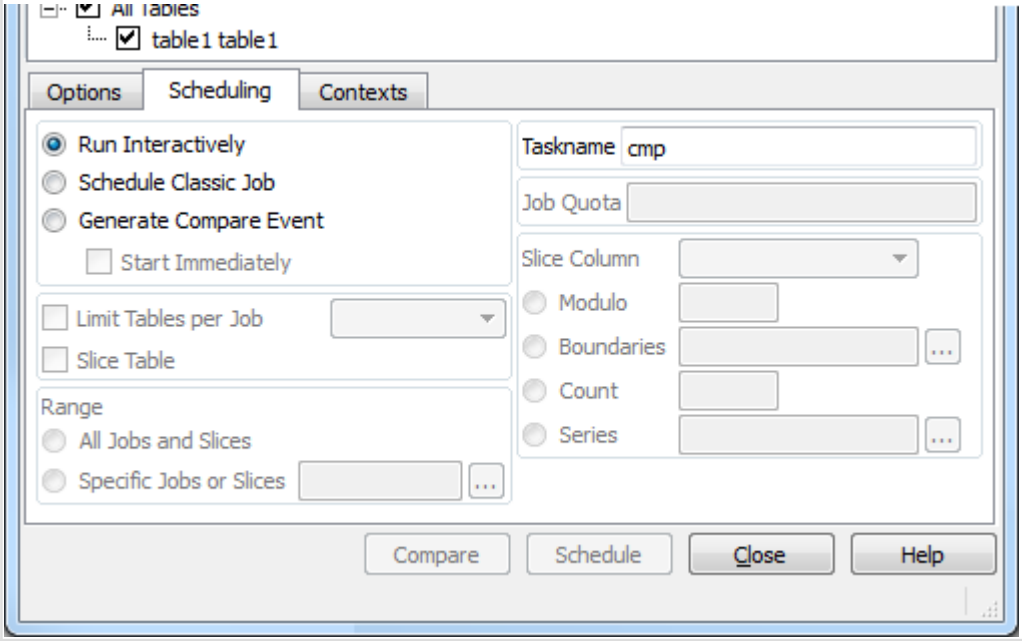
This option creates compare job for comparing the tables. By default, this compare job is created in **SUSPEND** state and they are named *chn*-**cmp**-*source-target*. This compare job can be invoked using command **Hvrstart** as in the following example:

```
hvrstart -u -w hubdb chn-cmp
```

Executing the above command unsuspends (moves to **PENDING** state) the jobs and instructs the scheduler to run them. Output from the jobs is copied to the **Hvrstart** command's **stdout** and the command finishes when all jobs have finished. Jobs created are cyclic which means that after they have run they go back to **PENDING** state again. They are not generated by a **trig_delay** attribute which means that once they complete they will stay in **PENDING** state without getting retriggered.

Once a compare job has been created with option **-s** then it can also be run manually on the command line (without using **HVR Scheduler**) as follows:

```
hvrstart -i hubdb chn-cmp-source-target
```



-S*sliceexpr*

Since v5.3.1/6

Compare large tables using slicing. Value *sliceexpr* can be used to split table into multiple slices. In HVR GUI, this option is displayed as **Slice Table** in the **Scheduling** tab.

If performing **Schedule Classic Job** (option **-s**), per slice a compare job is created for comparing only rows contained in the slice. These compare jobs can be run in parallel to improve the overall speed of the compare. Slicing can only be used for a single table (defined with option **-t**).

If performing an event driven compare - **Generate Compare Event** (option **-e**), only a single compare job is created for all slices.

The value *sliceexpr* affects action **Restrict /CompareCondition**. That action must be defined on the table (at least on the read location) and must contain a relevant **{hvr_var_slice_*}** substitution.

If performing event driven compare (option **-e**), multiple tables can be sliced simultaneously (same job) by supplying the table name with the *sliceexpr*. The format is *tablename.sliceexpr*. **Since** v5.5.5/6

As with option **-n** (bundles of tables), jobs are named by adding a number (starting at 0) to the task name which defaults **cmp** (although this task name can always be overridden using option **-T**). Normally the first slice's job is named *chn*-**cmp0**-*source-target* but numbers are left-padded with zeros, so if 10 slices are needed the first is named *chn*-**cmp00**-*source-target* instead.

The column used to slice a table must be 'stable', because if it is updated then a row could 'move' from one slice to another while the compare is running. The row could be compared in two slices (which will cause errors) or no slices (data-loss). If the source database is Oracle then this problem can be avoided using a common **Select Moment** (option **-M**).

For more information on slicing limitations, see section Slicing Limitations below.

Value *sliceexpr* must have one of the following forms:

col%*num*

Slicing using modulo of numbers. In HVR GUI, this option is displayed as **Modulo**.

Since HVR 5.6.5/0, it is not required to define **Restrict /CompareCondition** to use this type of slicing. However, prior to HVR 5.6.5/0, this slicing form affects the substitution **{hvr_var_slice_condition}** which must be mentioned in **Restrict /CompareCondition** defined for the slice table.

It is recommend that any **Restrict /CompareCondition** defined for slicing is also given a **/Context** parameter so it can be easily disabled or enabled.

If **-Sabc%3** is supplied then the conditions for the three slices are:

```
mod(round(abs(coalesce(abc, 0)), 0), 3)= 0
mod(round(abs(coalesce(abc, 0)), 0), 3)= 1
mod(round(abs(coalesce(abc, 0)), 0), 3)= 2
```

Note that the use of extra SQL functions (e.g. **round()**, **abs()** and **coalesce()**) ensure that slicing affect fractions, negative numbers and NULL too. Modulo slicing can only be used on a column with a numeric data type.

col<b1[<b2]… [<bN]	Slicing using boundaries. In HVR GUI, this option is displayed as **Boundaries**.
	If *N* boundaries are defined then *N+1* slices are implied.
	Since HVR 5.6.5/0, it is not required to define **Restrict /CompareCondition** to use this form of slicing. However, prior to HVR 5.6.5/0, this slicing affects the substitution **{hvr_var_slice_condition}** which must be mentioned in **Restrict /CompareCondition** defined for the slice table.
	
	It is recommend that any **Restrict /CompareCondition** defined for slicing is also given a **/Context** parameter so it can be easily disabled or enabled.
	
	If **-Sabc<10<20<30** is supplied then the conditions for the four slices are:
	`abc <= 10`
	`abc > 10 and abc <= 20`
	`abc > 20 and abc <= 30`
	`abc > 30 or abc is null`
	
	Note that strings can be supplied by adding quotes around boundaries, i.e. **-Sabc<'x'<'y'<'z'**.
	
	For very large tables consider the DBMS query execution plan. If the DBMS decides to 'walk' an index (with a lookup for each matched row) but this is not optimal (i.e. a 'serial-scan' of the table would be faster) then either use DBMS techniques (**$HVR_SQL_SELECT_HINT** allows Oracle optimizer hints) or consider modulo slicing (*col%inum*) instead.
	
	For this type of slicing, HVR can suggest boundaries by using the Oracle's **dbms_stats** package. Click the browse ("...") button for **Boundaries** type of slicing and then click **Suggest Values** in **Boundaries for Table** dialog. Number of slices can be also specified.
	
	Gathering column histogram statistics is required for this functionality to work. This can be done by calling the **dbms_stats.gather_table_stats** stored procedure.
	
	Examples:
	1. Gathers statistics including column histograms, for table 'table_name', using all table rows, for all columns, and maximum of 254 histogram buckets (therefore up to 254 slice boundaries can be suggested).
	`exec dbms_stats.gather_table_stats('schema_name', 'table_name', estimate_percent=>100, method_opt=>'for all columns size 254);`
	2. Gathers statistics including column histograms, for table 'table_name', using all table rows, for all indexed columns, and default number of histogram buckets.
	`exec dbms_stats.gather_table_stats('schema_name', 'table_name', estimate_percent=>100, method_opt=>'for all indexed columns);`
	3. Gathers statistics including column histograms, for table 'table_name', using 70% of table rows, for column 'table_column', and maximum of 150 histogram buckets (therefore up to 150 slice boundaries can be suggested).
	`exec dbms_stats.gather_table_stats('schema_name', 'table_name', estimate_percent=>70, method_opt=>'for columns table_column size 150);`
	4. Gathers statistics including column histograms, for table 'table_name', for all columns, and maximum 254 histogram buckets. This is an obsolete way to generate statistics and there are much less options supported.
	`analyze table table_name compute statistics for all columns size 254;`

	num Numbered slices. In HVR GUI, this option is displayed as **Count**. Since HVR 5.6.5/0, the number of each slice is assigned to substitution **{hvr_slice_num}** which must be mentioned in **Restrict /SliceCondition** defined for the slice table. Substitution **{hvr_slice_total}** is also assigned to the total number of slices. However, prior to HVR 5.6.5/0, the substitution **{hvr_var_slice_num}** must be mentioned in **Restrict /CompareCondition** defined for the slice table. Substitution **{hvr_var_slice_total}** is also assigned to the total number of slices. It is recommend that any **Restrict /CompareCondition** defined for slicing is also given a **/Context** parameter so it can be easily disabled or enabled. **Example:** In heterogeneous environments doing normal modulo slicing is not always possible because of different syntax of modulo operation. For example, in Oracle modulo operation is **mod(x,y)** and in Teradata it is **x mod y**. Also negative numbers are handled differently on these two databases. For this scenario, two **Restrict** actions can be defined, one for the capture location (Oracle) and other for the integrate location (Teradata): **Location** / **Action** ora: **Restrict /SliceCondition="abs(mod({hvr_var_slice_col}, {hvr_slice_total})) = {hvr_slice_num}" /Context=slice** terad: **Restrict /SliceCondition="abs({hvr_var_slice_col} mod {hvr_slice_total}) = {hvr_slice_num}" /Context=slice** Prior to HVR 5.6.5/0, using **Restrict /CompareCondition**: **Location** / **Action** ora: **Restrict /CompareCondition="abs(mod({hvr_var_slice_col}, {hvr_var_slice_total})) = {hvr_var_slice_num}" /Context=slice** terad: **Restrict /CompareCondition="abs({hvr_var_slice_col} mod {hvr_var_slice_total}) = {hvr_var_slice_num}" /Context=slice** If options **-S3 -Vslice_col=abc** are supplied then the conditions for the three slices are: **Capture Location**: `abs(mod(abc, 3))= 0` `abs(mod(abc, 3))= 1` `abs(mod(abc, 3))= 2` **Integrate Location**: `abs(abc mod 3)= 0` `abs(abc mod 3)= 1` `abs(abc mod 3)= 2`
	val1[,*val2*]... Slicing using a list of values. In HVR GUI, this option is displayed as **Series**. Values are separated by semicolons. Since HVR 5.6.5/0, each slice has its value assigned directly into substitution **{hvr_slice_value}** must be mentioned in **Restrict /SliceCondition** defined for the sliced table. However, prior to HVR 5.6.5/0, the substitution **{hvr_var_slice_value}** must be mentioned in **Restrict /CompareCondition** defined for the slice table. It is recommend that any **Restrict /CompareCondition** defined for slicing is also given a **/Context** parameter so it can be easily disabled or enabled. **Example:** A large table with column **country** having values **US**, **UK**, and **NL** can be sliced in the following manner: • If option **-S "'US';'UK'"** is supplied with action **Restrict /CompareCondition="country ={hvr_var_slice_value}" /Context=slice** then **HVR Compare** creates 2 slices (2 compare jobs) - 1 for **US** and 1 for **UK**. • If option **-S "'US';'UK'"** is supplied with action **Restrict /CompareCondition="country IN ({hvr_var_slice_value})" /Context=slice** then **HVR Compare** creates 2 slices (2 compare jobs) - 1 for **US** and 1 for **UK**. • If option **-S "('US','UK');('NL')"** is supplied with action **Restrict /CompareCondition="country IN ({hvr_var_slice_value})" /Context=slice** then **HVR Compare** creates 2 slices (2 compare jobs) - 1 for **US**, **UK** and 1 for **NL**. The double quotes (") supplied with option **-S** is not required in HVR GUI.
-t*y*	Only compare tables specified by *y*. Values of *y* may be one of the following: • *tbl* : Only compare table name *tbl*. • *t1*-*t2* : Compare all table codes that fall alphabetically between *t1* and *t2* inclusive. • !*tbl* : Compare all table names except *tbl*. • !*t1*-*t2* : Compare all table codes except for those that fall alphabetically between *t1* and *t2* inclusive. Several **-t***y* instructions can be supplied together.
-T*tsk*	Specify alternative task for naming scripts and jobs. The **default** task name is **cmp**, so for example without this **-T** option jobs are named *chn*-**cmp**-*l1*-*l2*.
-u*user*[/*pwd*]	Connect to hub database using DBMS account *user*. For some databases (e.g. SQL Server) a password must also be supplied.
-v	Verbose. This causes row-wise compare to display each difference detected. Differences are presented as SQL statements. This option requires that option **-gr** (row-wise granularity) is supplied.
-V*nm*=*val*	Supply variable for restrict condition. This should be supplied if a **Restrict /CompareCondition** parameter contains string **{hvr_var_***name***}**. This string is replaced with *val*. In HVR GUI, the option to supply variable for restrict condition is available under **Contexts** tab.
-w*N* **Since** v5.5.5/6	File prereaders per table. Define the number of prereader subtasks per table while performing direct file compare. This option is only allowed if the source or target is a file location.

Direct File Compare

Since v5.5.5/6

A direct file compare is a compare performed against a file location. This compare method is a faster alternative for file compare via Hive External Tables and also helps to avoid compare mismatches caused by data type coercion through Hive deserializer.

To perform a direct file compare:

- against a source file location, **Capture /Pattern** should be defined.
- against a target file location, **Integrate /ComparePattern** should be defined.

During direct file compare, HVR reads and parses (deserialize) files directly from the file location instead of using the HIVE external tables (even if it is configured for that location). In direct file compare, the files of each table are sliced and distributed to prereader sub tasks. Each prereader subtasks reads, sorts and parses (deserialize) the files to generate compressed(encrypted) intermediate files. These intermediate files are then compared with the database on the other side.

The number of prereader subtasks used during direct file compare can be configured using the compare option **-w**.

The location to store the intermediate files generated during compare can be configured using **LocationProperties /IntermediateDirectory**.

Direct file compare does not support Avro, Parquet or JSON file formats.

Direct file compare is not supported if **Restrict /RefreshCondition** is defined on a file location involved in the compare.

Slicing Limitations

This section lists the limitations of slicing when using **hvrcompare**.

Modulo

Following are the limitations when using slicing with modulo of numbers (*col%num*):

1. Only works on numeric data types. It may work with binary **float** values depending on DBMS data type handling (e.g. works on MySQL but not on PostgreSQL).
2. A diverse **Modulo** syntax on source and target (e.g. "**where col%5=2**" and "**where mod(col,5)=2**") may produce inaccurate results. This limitation applies only to classic compare and refresh. Since HVR 5.5.5/6, the event-driven compare can handle it.
3. Heterogeneous compare (between different DBMSes or file locations) has a limitation with **Modulo** slicing on the Oracle's **NUMBER(*)** column: if a value has an exponent larger than 37 (e.g. if a number is larger than 1E+37 or smaller than -1E+37), then this row might be associated with a wrong slice. This column should not be used for **Modulo** slicing. (The exact limits of the values depend on the number of slices).
4. For some supported DBMSes (SQL Server, PostgreSQL, Greenplum, Redshift), **Modulo** slicing on a **float** column is not allowed (may result in SQL query error).
5. For some DBMSes, **float** values above the limits of DBMS's underlying precision ("big **float** values") may produce inaccurate results during **Modulo** slicing. This affects only heterogeneous environments.
6. Compare with **Modulo** slicing on a column with "big **float** values" may produce inaccurate results in HANA even in a homogeneous environment (HANA-to-HANA).

A workaround for the above limitations is to use **Boundaries** slicing or **Count** slicing with custom SQL expressions.

Boundaries

Boundaries slicing of dates does not work in heterogeneous DBMSes.

Slicing with Direct File Compare

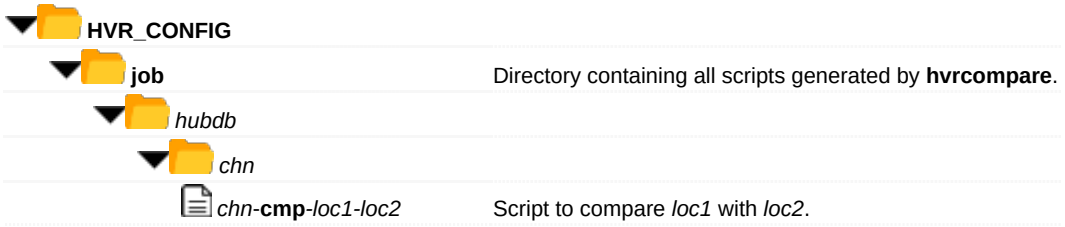
An Oracle's **NUMBER(*)** column or a similar column with big numbers (namely, if a number is larger than 1E+37 or smaller than -1E+37) cannot be used in direct file compare with slicing. A workaround is to use another column for slicing.

Example

For bulk compare of table **order** in location **src** and location **tgt**:

```
$ hvrcompare -rsrc -ltgt -torder hubdb/pwd sales
```

Files



HVR_CONFIG	
job	Directory containing all scripts generated by **hvrcompare**.
hubdb	
chn	
chn-**cmp**-*loc1*-*loc2*	Script to compare *loc1* with *loc2*.

See Also

Commands Hvrcrypt, Hvrgui and Hvrproxy.

Hvrcontrol

Contents

Name

hvrcontrol - Send and manage internal control files.

Synopsis

hvrcontrol [*-options*] *hubdb chn*

Description

Command **hvrcontrol** either sends HVR 'controls' to replication jobs, or removes them. A 'control' is a message file which can serve two functions;

1. To tell a job to do something else when it is already running. For example, wakeup or change its default behavior.
2. To instruct a job to treat certain rows in a special way, e.g. skip an old or 'bad' row, send a certain change straight to a 'fail table', or be resilient for some rows during an online refresh.

Correct use of command **hvrcontrol** requires understanding of undocumented internals of HVR. For this reason this command should only be used after consultation with HVR Technical Support or when its use is recommend by an HVR error message.

HVR sends control files internally in these areas;

- Command **Hvrstart** tells the **Hvrscheduler** to send a **trigger** control file. Jobs which are in a 'cycle loop' will detect this file and do an extra cycle even if they are still running. When this cycle is done they will delete this control file, so **Hvrstart** **-w** commands will terminate (otherwise they would keep hanging).
- Online refresh jobs (**Hvrrefresh** **-q**) sends *refresh taskname*_**online** (default is **refr**_**online**) control files to instruct capture and integrate jobs to skip changes made to the base tables before the refresh and to treat changes made while the refresh is running with resilience.

Options

This section describes the options available for command **hvrcontrol**.

Parameter	Description
-c	Only send control to capture jobs. By **default**, the control is sent to both capture and integrate jobs.
-d	Delete older control files while creating the new control, so that the new control replaces any old controls. The older control is deleted if it was for the same job and it had the same control name (see option **-n**).

-D	Delete control files and do not create a new control. All control files for the channel are deleted unless options **-c**, **-i**, **-l** or **-n** are supplied.
-E*name=value*	Set environment variable *name* to *value* in affected job.
-f	Affected changes should be sent directly to the 'fail table' instead of trying to integrate them. All changes are failed unless options **-w** or **-t** are supplied. This option can only be used on an integrate job and cannot be combined with options **-m**, **-r** or **-s**.
-F	Affected jobs should finish at the end of the next replication cycle.
-h*class*	Specify hub database. For supported values, see Calling HVR on the Command Line.
-i	Only send control to integrate jobs. By **default**, the control is send to both capture and integrate jobs.
-l*x*	Only send controls to jobs for locations specified by *x*. Values of *x* may be one of the following: Values of *x* maybe one of the following: • *loc* : Only location *loc*. • *l1*-*l2* : All locations that fall alphabetically between *l1* and *l2* inclusive. • **!***loc* : All locations except *loc*. • **!***l1*-*l2* : All locations except for those that fall alphabetically between *l1* and *l2* inclusive. Several **-l***x* instructions can be supplied together.
-m*col*	For affected changes value of column *col* should be set to missing. Setting the value to missing means that the change will not have data for *col* anymore. The column value is set to missing for all changes unless options **-w** or **-t** are supplied. It is not recommended to use this option on a key column. This option cannot be combined with options **-f**, **-r** or **-s**.
-n*ctrlname*	Name of control. This is part of the file name of the control created for each job and it also affects which old control files are deleted if option **-d** or **-D** are supplied. The **default** is **adhoc**.
-r	Affected changes should be treated with resilience (as if action **/Resilient=SILENT** is defined) during integration. All changes are resilient unless options **-w** or **-t** are supplied. This option can only be used on an integrate job and cannot be combined with options **-f**, **-m** or **-s**.
-s	Affected changes should be skipped. All changes are skipped unless options **-w** or **-t** are supplied. This option cannot be combined with options **-f**, **-m** or **-r**.
-t*y*	Only filter rows for tables specified by *y*. Values of *y* may be one of the following: • *tbl* : Only table *tbl*. • *t1*-*t2* : All tables that fall alphabetically between *t1* and *t2* inclusive. • **!***tbl* : All tables except *tbl*. • **!***t1*-*t2* : All tables except for those that fall alphabetically between *t1* and *t2* inclusive. Several **-t***y* instructions can be supplied together to **hvrcontrol**. This option must be used with either options **-f**, **-m**, **-r** or **-s**.
-u*user*[*/pwd*]	Connect to hub database using DBMS account *user*. For some databases (e.g. SQL Server) a password *pwd* must also be supplied.

-w*where*	Where condition which must have form *columname operator value*. The *operator* can be either **= != <> > < >=** or **<=**. The *value* can be a number, '*str*', **X**'*hex*', or a date. Valid date formats are *YYYY-MM-DD [HH:MM:SS]* in local time or *YYYY-MM-DDTHH:MM:SS+TZD* or *YYYY-MM-DDTHH:MM:SSZ* or **today** or **now** [[±]*SECS*] or an integer (seconds since **1970-01-01 00:00:00 UTC**). For some operators (**= != <>**) the value can be a list seperated by '**|**'. If multiple **-w** options are supplied then they are AND-ed together. For an OR condition multiple control files may be used. This option must be used with either options **-f**, **-m**, **-r** or **-s**.
-x*expire*	Expiry. The affected job should expire the control file and delete it when this time is reached. Valid date formats are *YYYY-MM-DD [HH:MM:SS]* in local time or *YYYY-MM-DDTHH:MM:SS+TZD* or *YYYY-MM-DDTHH:MM:SSZ* or **today** or **now** [[±]*SECS*] or an integer (seconds since **1970-01-01 00:00:00 UTC**). Option **-x0** therefore means that the control will be removed by the job after its first cycle.
-X*expire*	Receive expiry. The affected job should expire the control file and delete it after it has processed all changes that occurred before this time. Valid date formats are *YYYY-MM-DD [HH:MM:SS]* in local time or *YYYY-MM-DDTHH:MM:SS+TZD* or *YYYY-MM-DDTHH:MM:SSZ* or **today** or **now** [[±]*SECS*] or an integer (seconds since **1970-01-01 00:00:00 UTC**).

Examples

To instruct all jobs in channel **sales** to skip rows for table **x** with **prod_id<5** use:

```
hvrcontrol -s -tx "-wprod_id<5" hubdb/pwd sales
```

To instruct the capture job to skip all changes before a certain date and delete any old control files use;

```
hvrcontrol -d -s "-whvr_cap_tstamp<2010-07-07 12:00:00" hubdb/pwd sales
```

In HVR, each change has a unique **hvr_tx_seq** and *hvr_tx_countdown *combination, with these values acting as major and minor numbers respectively. Note also that **hvr_tx_countdown** has reverse ordering (i.e. for a big transaction the first change has countdown 100 and the last has countdown 1). The following command will send everything before the change with **hvr_tx_seq ffff** and **hvr_countdown 3** into the fail tables. Note the use of comparison operator **<<** for major /minor ordering.

```
hvrcontrol -f "-whvr_tx_seq<<X'ffff'" "-whvr_tx_countdown>3" hubdb/pwd sales
```

To instruct an integrate job for location **q** to be resilient for all changes where **(prod_id=1 and prod_price=10) or (prod_id=2 and (prod_price=20 or prod_price=21))** use two HVR controls:

```
hvrcontrol -i -lq -r -wprod_id=1 -wprod_price=10 hubdb/pwd sales
hvrcontrol -i -lq -r -wprod_id=2 "-wprod_price=20|21" hubdb/pwd sales
```

To make a running log-based capture job write a dump of its state (including all open transactions) into its log file (**$HVR_CONFIG/log/***hubdb***/***chn***-cap-***loc***.out**), use the following command:

```
hvrcontrol -c hubdb/pwd sales TxDump
```

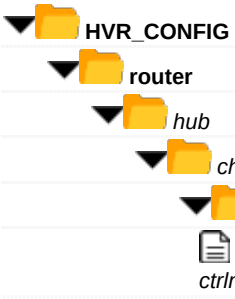
To view the contents of all control files affecting a channel, use the following command that converts the internal format into a readable XML format;

```
hvrrouterview -s hubdb/pwd sales
```

To delete all controls affecting a channel use;

```
hvrcontrol -D hubdb/pwd sales
```

Files



HVR_CONFIG	
router	
hub	
chn	
control	
tstamp.**ctrl-***jobname*-*ctrlname*	Control file containing instructions for a replication job. The contents of the file can be inspected using command **hvrrouterview**.

Hvrcrypt

Contents

Name

hvrcrypt - Encrypt passwords.

Synopsis

hvrcrypt *key* [*pwd*]

hvrcryptdb [*-options*] *hubdb*

Description

Command **hvrcrypt** can be used to interactively encrypt a password for a hub database when starting HVR on the command line. The second argument *pwd* is optional. If not specified **hvrcrypt** will prompt for it on the command line, not echoing the input. Using **hvrcrypt** is not needed for commands started with the HVR GUI.

Command **hvrcryptdb** will encrypt all unencrypted passwords in column **loc_remote_pwd** and **loc_db_user** in catalog **hvr_location** of the hub database, using column **loc_name** as key. Passwords entered using the HVR GUI will already be encrypted.

The first argument *hubdb* specifies the connection to the hub database, this can be an Ingres, Oracle or SQL Server database depending on its form. See further section Calling HVR on the Command Line.

Passwords are encrypted using an encryption key. Each password is encrypted using a different encryption key, so that if two passwords are identical they will be encrypted to a different value. The encryption key used for hub database passwords is the name of the hub database, whereas the key used to encrypt the login passwords and database passwords for HVR location sis the HVR location name. This means that if an HVR location is renamed, the encrypted password becomes invalid.

Regardless of whether **hvrcrypt** is used, **Hvrgui** and **Hvrinit** will always encrypt passwords before saving them or sending them over the network. The passwords will only be decrypted during authorization checking on the remote location.

Options

This section describes the options available for command **hvrcrypt**.

Parameter	Description
-h*class*	Specify hub database. Valid values are **oracle**, **ingres**, **sqlserver**, **db2**, **db2i**, **postgre sql**, and **teradata**. For more information, see section Calling HVR on the Command Line.
-u_user[*/pwd*]	Connect to hub database using DBMS account *user*. For some databases (e.g. SQL Server) a password must also be supplied.

Example

To start the HVR Scheduler at reboot without the password being visible:

Unix & Linux

```
$ DBUSER=<span style="color:blue;"><i>hvrhubaw</i></span>
$ DBPWD=<span style="color:blue;"><i>mypassword</i></span>
$ DBPWD_CRYPT=`hvrcrypt $DBUSER $DBPWD`
$ hvrscheduler $DBUSER/$DBPWD_CRYPT
```

Use of Unix command **ps|grep hvrscheduler** will give the following:

```
hvr 21852 17136 0 15:50:59 pts/tf 00:03 hvrscheduler -i hvrhubaw/!\{CLCIfCSy6Z7AUUya\}!
```

The above techniques also work for the hub database name supplied to **Hvrinit**.

Notes

Although the password encryption algorithm is reversible, there is deliberately no decryption command supplied.

Secure network encryption of remote HVR connections is provided using command **hvrsslgen** and action **LocationProperties /SslRemoteCertificate**.

Hvreventtool

Since v5.6.0/0

Contents

Name

hvreventtool - Manage HVR events

Synopsis

hvreventtool [*-options*] [**-h** *class*] [**-u** *user*] *hubdb*

Description

Command **hvreventtool** allows you to manage (add/edit/change state) events. The primary use of this command is to cancel events which are long running/not responding.

Events are any user action/activity that makes changes in HVR. HVR events are maintained in catalog tables **hvr_event** and **hvr_event_result** in hub database. Following are the different states of HVR Events -

- **PENDING** - Indicates an event is in pending state.
- **DONE** - Indicates an event is completed/finished.
- **FAILED** - Indicates an event is cancelled/failed.

Sample event:

```
"2019-06-07T12:31:46.475Z": {
    "type": "Refresh_Classic_Command",
    "user": "admin",
    "description": "HVR Refresh",
    "channel": "hvr_demo",
    "state": "DONE",
    "start_tstamp": "2019-06-07T12:31:46.475Z",
    "finish_tstamp": "2019-06-07T12:31:46.475Z",
        "body":  {"os_user":"admin","channel":"hvr_demo","source_loc":"src","target_locs":
["tgt"],"tables":["dm_order","dm_product"],"options":["-g b","-P 2","-s","-r src"]}
  }
```

Options

This section lists and describes all options available for **hvreventtool**.

Parameter	Description
-a*ev_type*	Add or create a new event. Event type *ev_ype* should be specified with this option. The default state of a new event is **PENDING**.

-b*ev_body*	Add event *body* in JSON format for an event. This can be used only when adding (option **-a**) a new event.
-c*chn_name*	Assign channel *chn* for an event. This can be used only when adding (option **-a**) a new event.
-d	Set event state as **DONE** without a response text when adding (option **-a**) or editing (o ption **-i**) an event. • When adding a new event, the new event is created in **DONE** state. • When editing an event, change the state from **PENDING** to **DONE**.
-f	Set event state as **FAILED** when adding (option **-a**) or editing (option **-i**) an event. • When adding a new event, the new event is created in **FAILED** state. • When editing an event, change the state from **PENDING** to **FAILED**.
-h*class*	Specify hub *class*, for connecting to hub database. For supported values, see Calling HVR on the Command Line.
-i*ev_id*	Edit or change event. Event ID *evid* should be specified with this option.
-lresult_loc_source	Add source location name in event result. This option can only be used when adding result (option **-R**) to an event.
-Lresult_loc_target	Add target location name in event result. This option can only be used when adding result (option **-R**) to an event.
-r*ev_responsetext*	Add response text *resptext* for an event. This option can only be used when changing the event state using options **-d** and **-f**.
-R*result*	Add *result* for an event.
-t*result_table*	Add the table in event result. This option can only be used when adding result (option **-R**) to an event.
-T*ev_textdescr*	Add description for an event. This can be used only when adding (option **-a**) a new event.
-u*user*[/*pwd*]	Connect to hub database using DBMS account *user*. For some databases (e.g. SQL Server) a password must also be supplied.

Example

- To change an event's state from **PENDING** to **FAILED** :

```
hvreventtool -i -f -r "Manually cancelling because of ..." myhub/myhubpwd 2018-12-
12T23:59:59.000Z
```

- To change an event's state from **PENDING** to **DONE**:

```
hvreventtool -i -d -r "Took 1 hour" myhub/myhubpwd 2018-12-12T23:59:59.000Z
```

- To add a custom event (default is **PENDING** state):

```
hvreventtool -a myhub/myhubpwd TakingBackup
```

See Also

Events, Hvreventview

Hvreventview

Since v5.5.5/2

Contents

Name

hvreventview - Display events from **hvr_event** and **hvr_event_result**

Synopsis

hvreventview [**-h** *class*] [**-u** *user*] [*-opts*] *hubdb*

Description

Command **hvreventview** displays events and their results from the catalog tables - **hvr_event** and **hvr_event_result**.

Options

This section lists and describes all options available for **hvreventview**.

Parameter	Description
-b*begin_id*	Display only events with event ids newer than *begin_id*. Value *begin_id* must have form *YYYY-MM-DD HH:MM:SS[.MSECS], YYYY-MM-DD***T***HH:MM:SS[.MSECS]+TZD* or *YYYY-MM-DD***T***HH:MM:SS***Z***[.MSECS]*.
-B*begin_updated* **Since** v5.5.5/5	Display only events which were updated since *begin_updated*. Value *begin_updated* m ust have form YYYY-MM-DD HH:MM:SS[.MSECS], *YYYY-MM-DD***T***HH:MM:SS[. MSECS]+TZD* or *YYYY-MM-DD***T***HH:MM:SS***Z***[.MSECS]*.
-c*chn*	Display only events for channel *chn*. This option can be supplied multiple times.
-C	Display only the current event. This is either the earliest event with state **PENDING** or, if no such event exists, the latest event with state **DONE** or **FAILED**. This option requires **-j**.
-e*end_id*	Display only events with event ids older than *end_id*. Value *end_id* must have form *YY YY-MM-DD HH:MM:SS[.MSECS]*, *YYYY-MM-DD***T***HH:MM:SS[.MSECS]+TZD* or *YYYY -MM-DD***T***HH:MM:SS***Z***[.MSECS]*.
-E*end_updated* **Since** v5.5.5/5	Display only events which were updated before *end_updated*. Value *end_updated* mus t have form *YYYY-MM-DD HH:MM:SS[.MSECS]*, *YYYY-MM-DD***T***HH:MM:SS[.MSECS] +TZD* or *YYYY-MM-DD***T***HH:MM:SS***Z***[.MSECS]*.
-f	Follow mode. Displays events that get updated after **hvreventview** was invoked. Runs in an endless loop. If specific event ids are specified with **-i**, then it terminates as soon as all of these events have state **DONE**.
-h*class*	Hub class, for connecting to hub database.

-i*ev_id*	Display only event with event id *ev_id*. This option can be supplied multiple times.
-j*job_name*	Display only events for job *job_name*. This option can be supplied multiple times.
-l*loc* **Since** v5.5.5/5	Display only event results for either source or target location *loc*. This option can be supplied multiple times. This option requires **-r** or **-R**.
-L*loc*	Display only event results for target location *loc*. This option can be supplied multiple times. This option requires **-r** or **-R**. ⚠ This option is not available since HVR 5.5.5/5.
-n*res*	Display only latest event plus results where one of the results matches *res*.
-N*num* **Since** v5.5.5/5	Display only the number of events specified in *num*. This option displays only the latest events. For example, if **-N5** is supplied then it displays the latest 5 events.
-o*ofile*	Write output to file *ofile* instead of displaying it.
-r	Also display results from events. All results are shown except ones starting whose name starts with '_' (these are advanced/internal).
-R*res_patt*	Display only event results with result names matching *res_patt*. Value *res_patt* may contain alphanumerics and any of the symbols '*', '?', '|', '_' and '-'. This option implies **-r**. To see all results (including advanced/internal results starting with '_') use **-R ***.
-s*state*	Display only events with state *state*. Value *state* may be **PENDING**, **DONE** or **FAILED**. This option can be supplied multiple times.
-S*body_patt*	Display only events with contents of body matching *body_patt*. Value *body_patt* may contain alphanumerics and any of the symbols '*', '?', '|', '_' and '-'.
-t*tbl*	Display only event results for table *tbl*. This option can be supplied multiple times.
-T*type_patt*	Display only events with event types matching *type_patt*. Value *type_patt* may contain alphanumerics and any of the symbols '*', '?', '|', '_' and '-'.
-u*user*[/*pwd*]	Connect to hub database using DBMS account *user*. For some databases (e.g. SQL Server) a password must also be supplied.

Example

To display all pending **refresh** events for channel test_channel for the 12th of december 2018, use the following;

```
hvreventview -b "2018-12-12T00:00:00.000Z" -e "2018-12-12T23:59:59.000Z" -c test_channel -
s PENDING -T Refresh myhub/myhubpwd
```

To monitor all **compare** events as they run, including their results, use the following;

```
hvreventview -f -T Compare myhub/myhubpwd
```

Hvrfailover

Ingres | Unix & Linux

Contents

⚠ Command **hvrfailover** applies only to Ingres locations.

Name

hvrfailover - Failover between Business Continuity nodes using replication

Synopsis

hvrfailover [**-r**] [**-v**] **start**

hvrfailover [**-r**] [**-v**] **stop**

hvrfailover boot

hvrfailover check

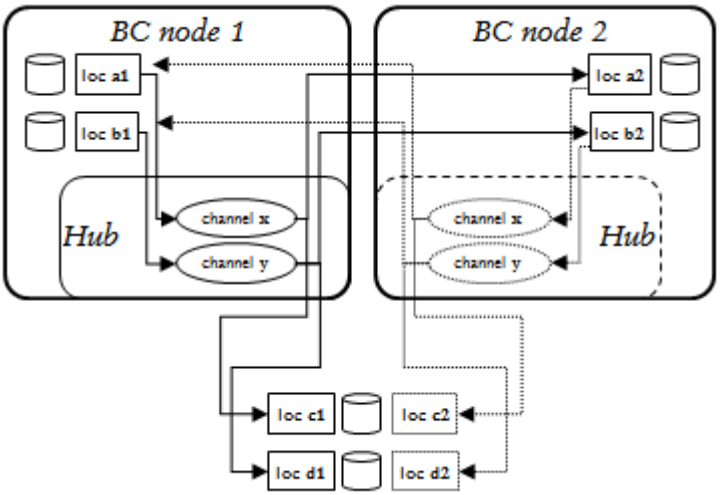
Description

hvrfailover manages the 'failover' of a database service between two nodes. These are called 'Business Continuity (BC) nodes'. They do not need to share a disk. At any moment only one of these BC nodes is active, and HVR is replicating database changes across to the other node, so that if an error occurs on the active node then **hvrfailover** can switch-over to a 'virtual IP address'. At this point existing database connections are broken (an error message will be returned immediately or they may have to timeout first) and new database connections are sent to the other node instead. This ensures 'high availability' for the database service, and means a replicated system has no 'single point of failure'. Each BC node contains one or more replicated databases plus a HVR hub with channels to replicate changes from them to the other BC node.

It is important to distinguish between 'graceful failover' and 'abrupt failover'. Graceful failover is when a proper 'stop' was performed on the first node (so that all replicated changes are flushed to the other node) before switching to the virtual IP address and starting on the other node. In this case, the first node is still consistent and a 'failback' is possible without causing consistency problems. Abrupt failover is when a proper stop was not performed (machine crashed?) or did not fully complete (network timeout?). In this case there could be unreplicated changes left in the first node (called 'phantom changes'), so a 'failback' to it is impossible because it could give database inconsistency. Resetting a database after an abrupt failover typically requires an **Hvrrefresh** and a new **Hvrinit**. The unreplicated changes are lost.

The replication inside the **hvrfailover** nodes must conform to the diagram on the right. The arrows show the direction of replication; the dotted arrows are replication after failover has occurred. The names of the hub database and channels are

identical in both nodes. All locations in one BC node must end with suffix **1** and in the other with suffix **2**. Channels in one node must capture changes from one location inside that node and integrate them to one on the other node.



So for example if a channel named **mychannel** in the hub database on the first BC node capture location is **xxx1** and an integrate location **xxx2**, then channel **mychannel** also exists in other BC node with capture location **xxx2** and integrate location **xxx1**.

Channels may also integrate changes to extra databases outside either BC nodes. This can be used to add an off-site 'disaster recovery' database to the channel for a 'one-way' switch-over after a real emergency, with the two BC nodes providing bi-directional switch-over to maintain business continuity for smaller failures or during batchwork. These databases must be added with a different location name in each hub; with suffix **1** in the first node's hub and with suffix **2** in the other hub.

Command **hvrfailover start** first checks that the other BC node is not active by doing a **ping** of virtual IP address (unless option **-r** is supplied) and **Hvrtestlistener** to the other hub's scheduler (unless option **-v** is supplied). It then begins replication to the other node by starting the HVR Scheduler. Finally (if option **-r** is not supplied), it activates the virtual IP address, so new database connections will be redirected to databases on this node. Option **-r** therefore means that replication should be started, but database connections via the virtual IP address should not be affected. Option **-v** means that only the virtual IP address is started or stopped. Command **hvrfailover start** should never be done on one node while either replication or the virtual IP address is active on the other node.

Command **hvrfailover stop** breaks all connections into the local databases by stopping and starting the DBMS communication server (e.g. **ingstop/start -iigcc**, not if **-v** is supplied), and deactivating the virtual IP address (not if **-r** is supplied). It also attempts to flush replication so that both databases are identical.

Configuration Options

File **$HVR_CONFIG/files/hvrfailover.opt** contains the following options:

Parameter	Description
-env=*NAME*=*VALUE*	Environment variables. Set e.g. **$II_SYSTEM** and **$HVR_PUBLIC_PORT**.
-hubdb=*dbname*	Hub database name. Should be the same name on both BC nodes. Mandatory.
-virtual_ip=*ipaddress*	Virtual IP address.

Configuring Hvrfailover

1. Configure the DBMS (Ingres) so it is restarted at boot time on both nodes.
2. Configure the **hvrfailover.opt** file in **$HVR_CONFIG/files**. This file should be identical on both BC nodes.
   Example:

   ```
   -hubdb=hvrhub
   -virtual_ip=192.168.10.228
   -env=II_SYSTEM=/opt/Ingres/IngresII
   -env=HVR_PUBLIC_PORT=50010
   ```

3. Create a new interface for the virtual IP address on both BC nodes. This can be done, as **root** or by copying an interface file into directory **/etc/sysconfig/network-scripts**.
   Example:

```
$ cp ifcfg-eth0 ifcfg-eth0:1
```

Then edit this file and change the directives:

```
DEVICE=eth0:1    #(for this example)
IPADDR=<virtual ip>
ONBOOT=no
USERCTL=yes
```

4. Configure **Hvrremotelistener** and **Hvrfailover** so they are restarted at boot time by adding a line to **/etc/hvrtab** on both nodes.
   Example:

```
root 4343 /opt/hvr/hvr_home /opt/hvr/hvr_config -EHVR_TMP=/tmp
ingres hvrfailover /opt/hvr/hvr_home /opt/hvr/hvr_config -EHVR_TMP=/tmp
```

5. Configure **Hvrmaint** to run frequently, for example by adding a line to **crontab**.
6. Optionally a crontab may be added to run **Hvrlogrelease** on both BC nodes. **Hvrlogrelease** option (**-logrelease_expire**) should be used to ignore logrelease files older than a certain period. This is necessary to ensure **Hvrlogrelease** cleans up redo/journal files on the BC nodes.

Files

HVR_HOME	
bin	
hvrfailover	Perl script.
HVR_CONFIG	
files	
hvrfailover.opt	Option file.
hvrfailover.state	File created automatically by **hvrfailover**. It contains string **START** or **STOP**. This file is used by **hvrfailover boot** at machine reboot to decide if the scheduler and virtual IP address should be reallocated.
hvrfailover.stop_done	File created automatically by **hvrfailover**.
log	
hvrfailover	
hvrfailover.log	Log file.

Hvrfingerprint

Contents

Name

hvrfingerprint - Display host fingerprint

Synopsis

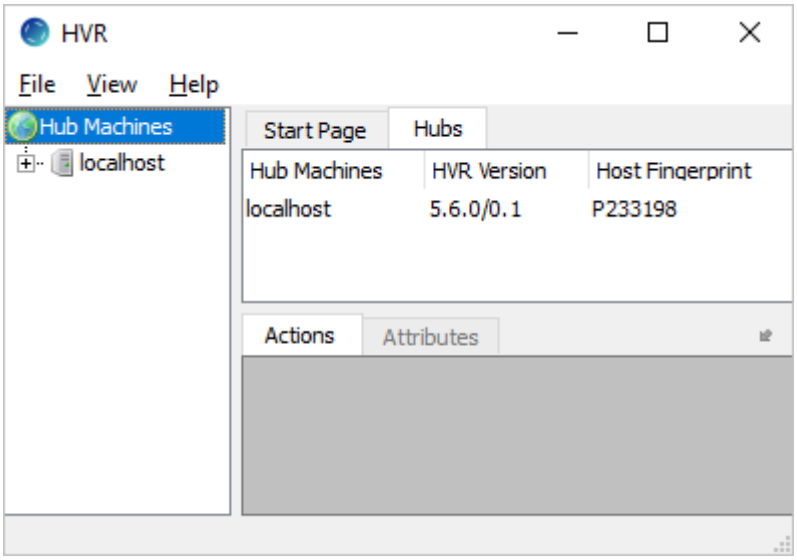
hvrfingerprint

Description

Command **hvrfingerprint** causes HVR to calculate and display the hardware fingerprint of the current host it is running on. A host fingerprint is an upper case letter followed by 6 decimal digits, such as **P233198**. This command should be used on the hub host to learn its fingerprint.

```
C:\hvr>hvrfingerprint
P233198

C:\hvr>
```

Host fingerprint of the hub host is also displayed in **hvrgui**.



Hardware fingerprint of an Amazon EC2 virtual machine will be reported with an 'A' letter, such as **A654321**. Only EC2 instance ID is used in this case for fingerprinting. The fingerprint stays the same even if EC2 virtual machine migrates to a different hardware.

Hvrgui

Contents

Name

HVR GUI - HVR Graphical User Interface.

Synopsis

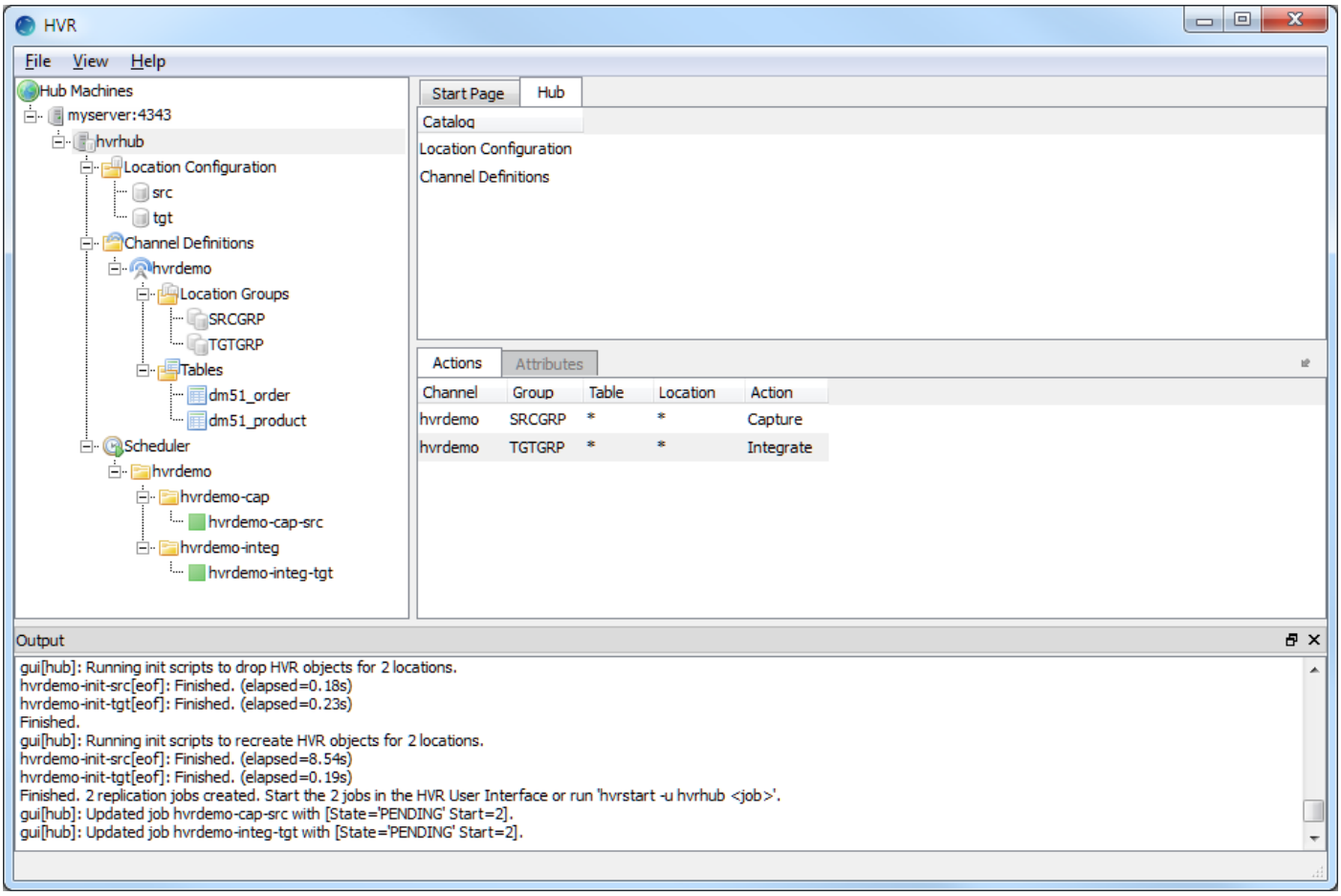
hvrgui

Description

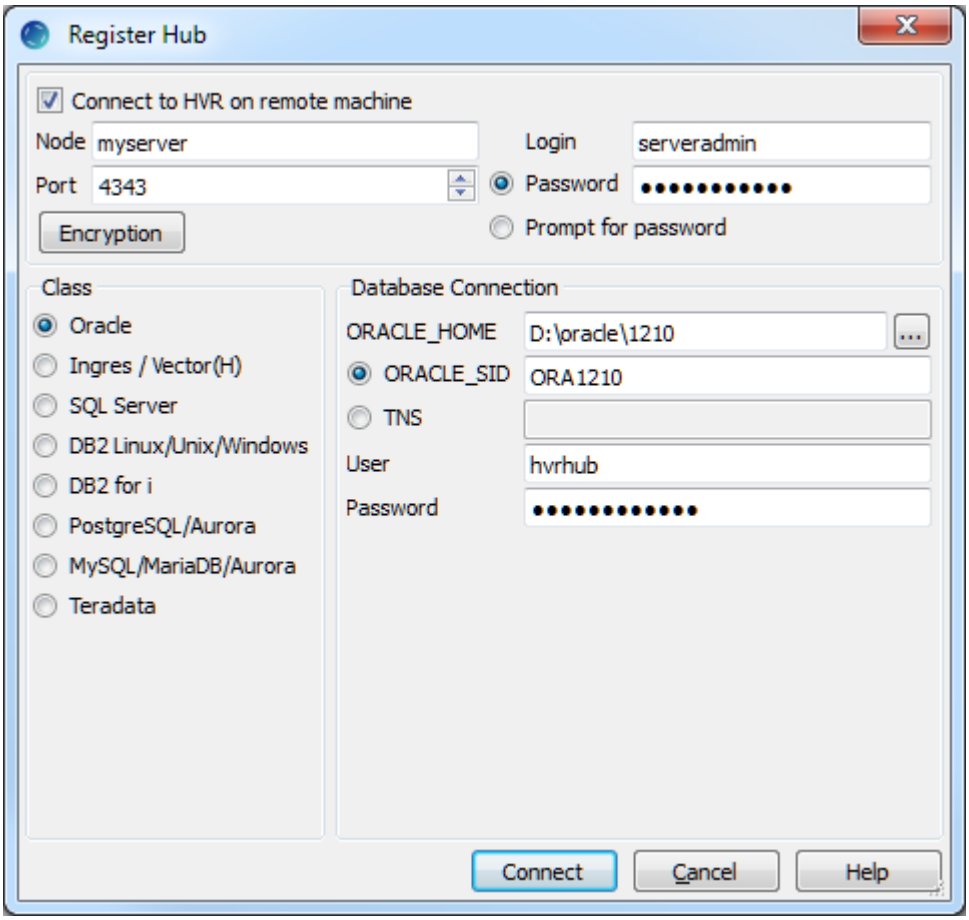
HVR GUI is a Graphical User Interface used to configure replication. The GUI can just be run on the hub machine, but it can also run on the user's PC and connect to a remote hub machine. To start the GUI double-click on its Windows shortcut or execute command **hvrgui** on Linux. HVR GUI does not run on Unix machines; instead it must connect from the user's PC to such a hub machine.

The main window consists of four panes.

1. The top-left pane contains a treeview with hub(s), location(s), channel(s) and its definitions.
2. The top-right pane displays details from the node selected in the treeview.
3. The **Actions** pane lists the HVR actions configured for the channel selected in the treeview. The **Attributes** pane displays the attributes of the scheduler jobs. This pane is enabled only when **Scheduler** or any node below scheduler is selected in the treeview.
4. The bottom pane displays the logfile and error information.



When HVR's GUI is launched for the first time **Register Hub** window is displayed automatically to input details required for connecting to the hub database. **Register Hub** can also be accessed from menu **File Register Hub**.



After registering a hub, you can see folders for **Location Configuration** and **Channel Definitions**. Right-clicking on these (or on the actual channels, locations, etc. under it) reveals a menu that allows you to do things like:

- Create a new instance of that object
- Perform commands like an **HVR Initialize**, **Export**, **Import**
- Create an action for that object

⚠ In the HVR GUI you can change an action that is defined for all tables (**Table="*"**) with a set of actions (each for a specific table) as follows. Right-click in the action's **Table field Expand**. You can expand actions on any field that has a **"*"** value.

Hvrinit

Contents

Name

hvrinit - Load a replication channel.

Synopsis

hvrinit [*options*] *hubdb chn*

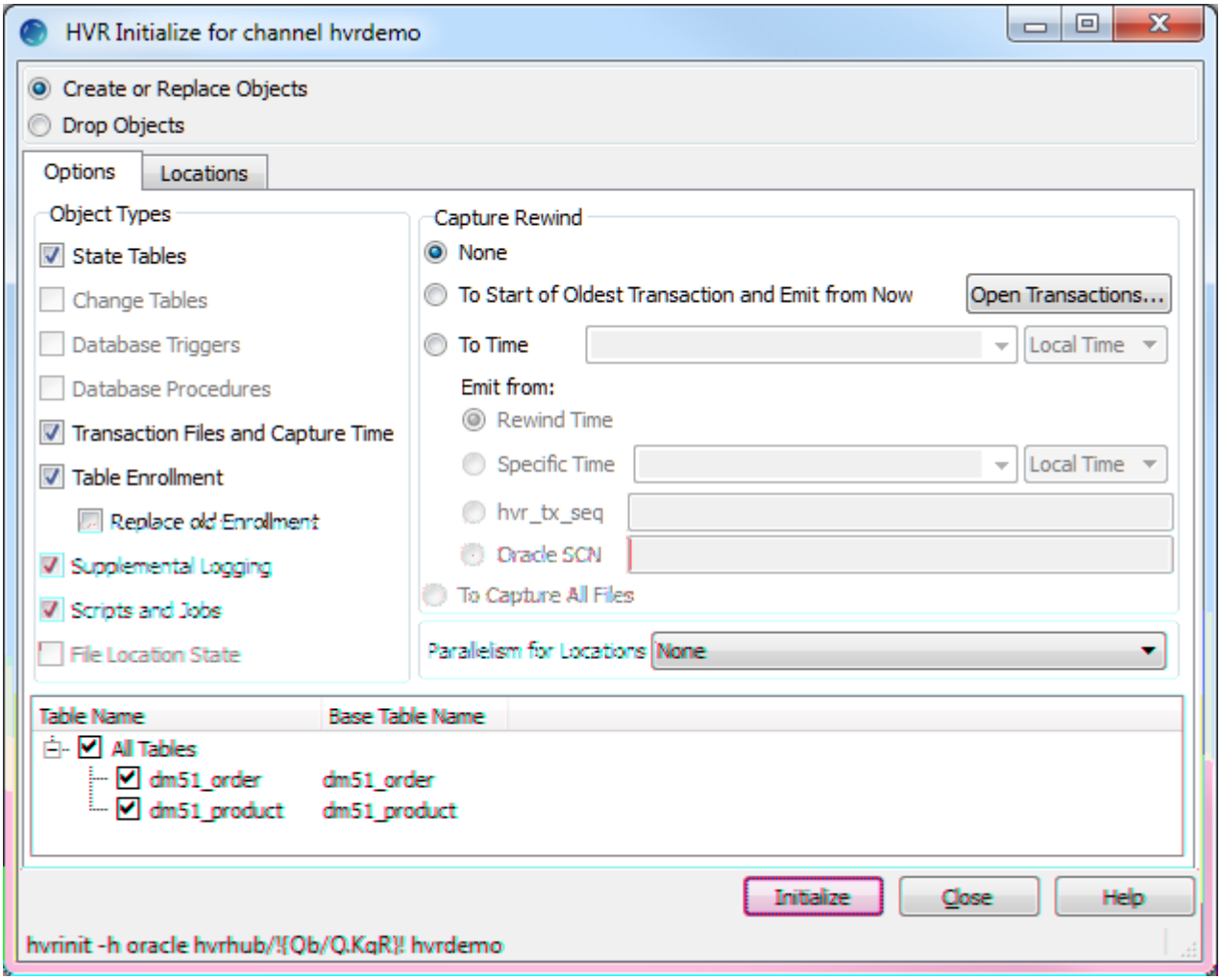
Description

hvrinit encapsulates all steps required to generate and load the various objects needed to enable replication of channel *chn*. These objects include replication jobs and scripts as well as database triggers/rules for trigger-based capture and table enrollment information for log-based capture.

The first argument *hubdb* specifies the connection to the hub database. For more information about supported hub databases, see Calling HVR on the Command Line.

Options

This section describes the options available for command **hvrinit**.

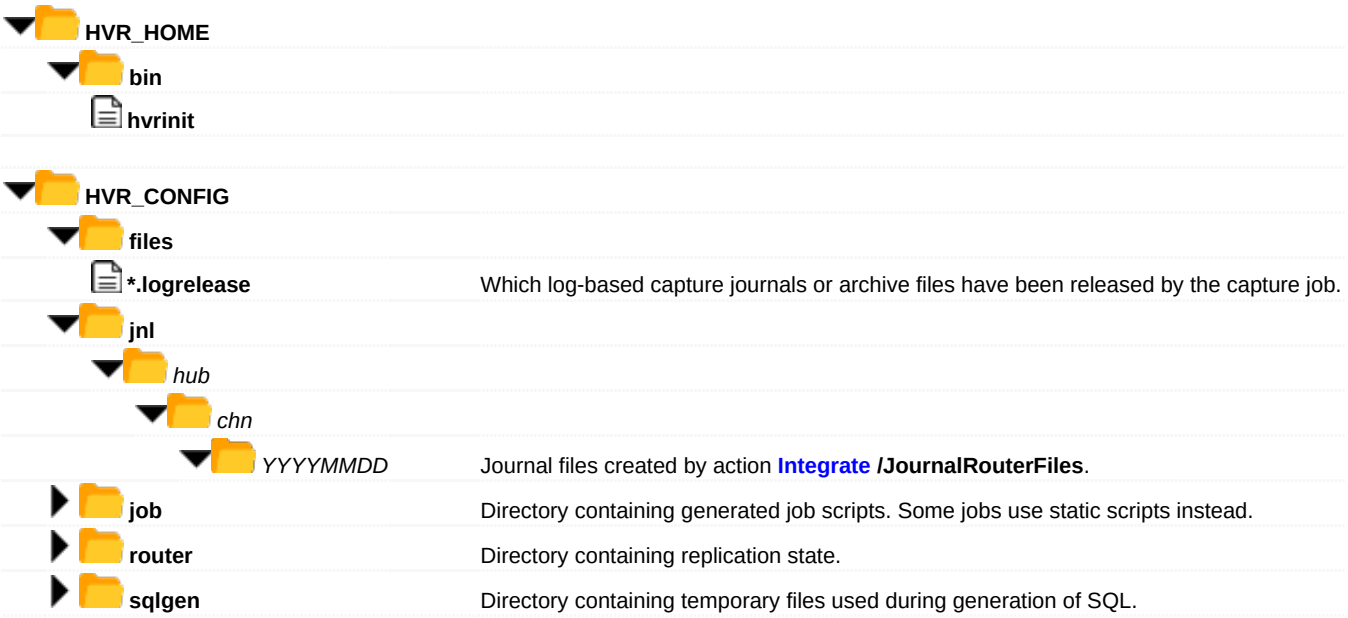


Parameter	Description
-d	Drop objects only. If this option is not supplied, then **hvrinit** will drop and recreate the objects associated with the channel such as HVR scripts, internal tables and any transaction files containing data in the replication pipeline. Only a few objects are preserved such as job groups in the scheduler catalogs; these can be removed using **hvrinit -d**.
-E Since v5.3.1/1	Recreates (replace) enroll file for all tables present in the channel. ⓘ In HVR versions released between 5.3.1/5 and 5.5.0/2, enroll file is recreated only for the tables that are selected during **hvrinit**. Using **hvrinit -E** is same as **hvrinit -osctprEljf** (in HVRGUI it is same as selecting all options under **Object Types**).
-h*class*	Specify hub database. Valid values are **oracle**, **ingres**, **sqlserver**, **db2**, **db2i**,**postgre sql**, or **teradata**. For more information, see section Calling HVR on the Command Line.

-i*x*	Capture rewind. Initialize channel to start capturing changes from a specific time in the past, rather than only changes made from the moment the **hvrinit** command is run. Capture rewind is only supported for database log-based capture (not for trigger-based capture i.e. **/TriggerBased** parameter) and for capture from file locations when parameter **/DeleteAfterCapture** is not defined. Values of *x* may be one of the following: • *oldest_tx* : Capture changes from the beginning of the oldest current (not closed) transaction (transactions that do not affect channel's tables will not be considered) and emit from **now**. • *time* : Valid formats are *YYYY-MM-DD [HH:MM:SS]* (in local time) or *YYYY-MM-DD***T***HH:MM:SS+TZD* or *YYYY-MM-DD***T***HH:MM:SSZ* or **today** or **now***[±SECS]* or an integer (seconds since **1970-01-01 00:00:00 UTC**). For example, **-i "2017-11-01 12:52:46"** or **-i 2017-11-01T12:52:46-06:30** or **-i 2017-11-01T12:52:46Z** or **-i now-3600** (for one hour ago). • *min* : Capture all available changes in file location. The following should be noted when executing **Integrate** after running **HVR Initialize** with **Capture Rewind**: • When continuous **Integrate** (without **/Burst**) is executed then HVR automatically skips the rows that are already processed by a previous integrate cycle (the transaction sequence number for these rows are already available in the state tables). To forcefully Integrate these rows, the state tables should be cleared (select **State Tables** (option **-os**) in **HVR Initialize**). • If **State Tables** (option **-os**) is selected (to clear the state tables) while performing **HVR Initialize** and when continuous integrate (**Integrate** without **/Burst**) is executed then **Integrate /Resilient** should be defined to avoid the error triggered due to the rows that are already available (processed by a previous integrate cycle) in the target location. • When integrate with **/Burst** is executed then **Integrate /Resilient** should be defined to avoid the error triggered due to the rows that are already available (proc essed by a previous integrate cycle) in the target location.
-l*x*	Defines start emitting time. This option requires **-i**. • *time* : Emit changes from the specified moment of time. The time formats are the same as for **-i** option. • **hvr_tx_seq**=*number* : Emit from a specific HVR transaction sequence number. The *number* can be given in a decimal or a hexadecimal form. If *number* contains decimal digits only then it is decimal. Otherwise, if it starts from prefix **0x** or contains hexadecimal digits **A**,**B**,**C**,**D**,**E** or **F** then it is treated as hexadecimal. • **scn**=*number* : Emit changes from a specified **SCN**. For Oracle, this is equivalent to Emit from HVR transaction sequence number where **hvr_tx_seq**=**scn*65536**. The *number* can be in a decimal or a hexadecimal form.
-l*x*	Only affect objects for locations specified by *x*. Values of *x* may be one of the following: • *loc* : Only affect location *loc*. • *l1*-*l2* : Affects all locations that fall alphabetically between *l1* and *l2* inclusive. • **!***loc* : Affect all locations except *loc*. • **!***l1*-*l2* : Affects all locations except for those that fall alphabetically between *l1* and *l2* inclusive. Several **-l***x* instructions can be supplied together to **hvrinit**.

-o*S*	Operations limited to objects indicated by *S*. Values of *S* may be one of the following: • **s** : State tables in database locations, e.g. the toggle table. • **c** : Tables containing changes, e.g. burst, history and fail tables. • **t** : Database triggers/rules for tables with trigger-based capture. • **p** : Database procedures in database locations. • **r** : Router directory's transaction files. Also capture start time for log-based capture and file copy channels. • **e** : Extend enroll information for tables with log-based capture. • **E** : Recreate (replace) enroll file for all tables present in the channel. In HVR versions released between 5.3.1/5 and 5.5.0/2, enroll file is recreated only for the tables that are selected during **hvrinit**. • **l** : Supplemental logging for log-based capture tables. • **j** : Job scripts and scheduler jobs and job groups. • **f** : Files inside file location's state directory. Several **-o***S* instructions can be supplied together (e.g. **-octp**) which causes **hvrinit** to effect all object types indicated. Not specifying a **-o** option implies all objects are affected (equivalent to **-osctpreljf**).
-p*N*	Indicates that SQL for database locations should be performed using *N* sub-processes running in parallel. Output lines from each subprocess are preceded by a symbol indicating the corresponding location. This option cannot be used with option **-S**.
-r*checkpointfilepath* **Since** v5.5.5/6	Adopt most suitable retained checkpoint from checkpoint files available in *checkpointfilepath*. The *checkpointfilepath* should be the exact path for the directory containing checkpoint files. For more information about saving checkpoint files in a directory, see **Capture /CheckpointStorage**. A checkpoint file is considered not suitable when the checkpoints available in it are beyond the rewind or emit times requested for **hvrinit**. This option can be supplied more than one time with **hvrinit** to use multiple checkpoint file paths.
-S	Write SQL to **stdout** instead of applying it to database locations. This can either be used for debugging or as a way of generating a first version of an SQL include file (see action **DbObjectGeneration /IncludeSqlFile**), which can later be customized. This option is often used with options **-l***loc* **-otp**.
-t*y*	Only affect objects referring to tables specified by *y*. Values of *y* may be one of the following: • *tbl* : Only affects table *tbl*. • *t1*-*t2* : Affects all tables that fall alphabetically between *t1* and *t2* inclusive. • **!***tbl* : Affects all tables except *tbl*. • **!***t1*-*t2* : Affects all tables except for those that fall alphabetically between *t1* and *t2* inclusive. Several **-t***y* instructions can be supplied together to **hvrinit**.
-u*user*[*/pwd*]	Connect to hub database using DBMS account *user*. For some databases (e.g. SQL Server) a password must also be supplied.

Files



See Also

Commands Hvrproxy, Hvrrouterview and Hvrscheduler.

Hvrlivewallet

Oracle

Contents

Name

hvrlivewallet - HVR Live Wallet.

Synopsis

hvrlivewallet [**-***v*] *hubdb locuser/locpwd*

hvrlivewallet [**-***v*] *portnum locuser/locpwd*

Description

hvrlivewallet sets a password for the Oracle TDE wallet associated to the location specified by *locuser* on the HVR Live Wallet port. The HVR Live Wallet port is an additional listening port started whenever a **HVR Scheduler** or **HVR Remote Listener** is started. It is designed to remember encrypted passwords in the process memory, which will be lost whenever the process terminates. Passwords set with HVR Live Wallet will never be written to disk nor leave the machine's memory.

hvrlivewallet [**-***v*] *hubdb locuser/locpwd* sets a password for a local connection on the HVR Scheduler's Live Wallet port.

hvrlivewallet [**-***v*] *portnum locuser/locpwd* sets a password for a remote location on the HVR Remote Listener's Live Wallet port.

Running either a **HVR Scheduler** or **HVR Remote Listener** is mandatory to use **hvrlivewallet**.

For remote locations, **hvrlivewallet** needs to be run on the remote machine.

⚠ **hvrlivewallet** is not supported for Unix **inetd** or Linux **xinetd**.

Options

This section describes the options available for command **hvrlivewallet**.

Parameter	Description
-v	Validate the provided password.

Notes

Despite the fact that the Oracle database is provided, the password is remembered in association with the location of the wallet. Hence, if multiple locations using the same Oracle wallet share a HVR Scheduler or HVR Remote Listener, the wallet password must be set only once using **hvrlivewallet**.

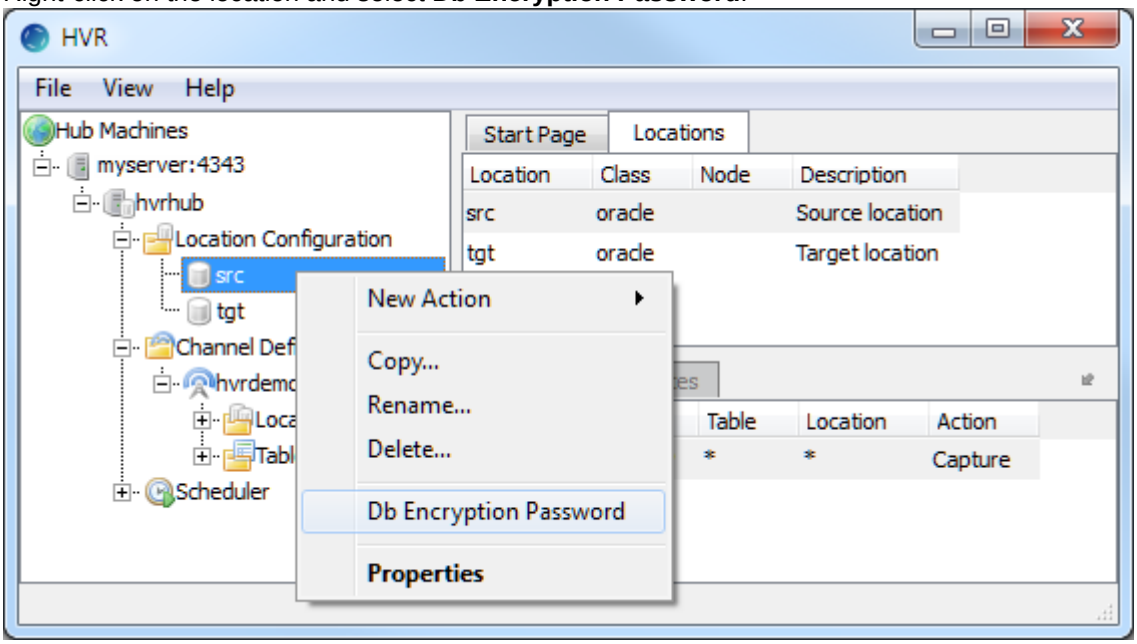
One part of **hvrlivewallet** is a password validation step to see that the correct password is given. Therefore it is not possible to set a wrong password into the HVR Live Wallet port. Additionally, **hvrlivewallet** will report if an existing password is going to be overwritten. The report message will contain the timestamp when the previous password was set.

If the HVR Live Wallet port cannot remember a requested password, **hvrlivewallet** will report the timestamp when the HVR Live Wallet port was last restarted.

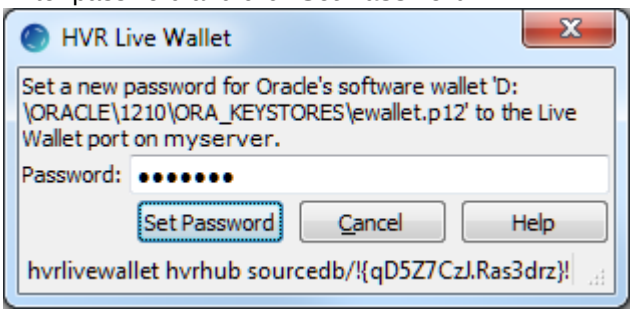
GUI

For Oracle locations that have a **Capture** action defined, the HVR Live Wallet functionality can be used in the GUI.

1. Right-click on the location and select **Db Encryption Password**.



2. Enter password and click **Set Password**.



Examples

For a local location **hvrcen** on the hub **hvrhub**.

```
$ hvrlivewallet hvrhub hvrcen/hvrcen
```

For a remote location **hvrcen** with the HVR Remote Listener on port **4343**.

```
$ echo "my_secret" | hvrlivewallet 4343 hvrcen/hvrcen
```

See Also

Command Hvrscheduler and Hvrremotelistener.

Hvrlogrelease

Contents

Name

hvrlogrelease - Manage DBMS logging files when not needed by log-based capture.

Synopsis

hvrlogrelease [*optfile*] [*-options*]

Description

hvrlogrelease works differently depending on database class applied: it either manages log files (for Oracle and Ingres) or frees log file blocks (for SQL Server). Database class is given by one of **-oracle_sid**, **-ingres_db** or **-sqlserver_db** options, which must be provided.

Functionality for Oracle

hvrlogrelease can be used to manage Oracle archive files so that they are available if needed by HVR's log-based capture jobs. The default behavior is that **hvrlogrelease** monitors the 'primary' archive directory where the Oracle archiver process writes these files and makes a copy in a private directory whenever one appears. It then removes these copies when the HVR capture jobs will no longer need their contents.

An alternative behavior is activated by option **-purge**; in this case no copies are made and **hvrlogrelease** simply deletes Oracle archive file from the 'primary' archive directory once it sees these files will no longer be needed by HVR capture jobs. This is useful if archiving was only enabled for HVR's log-based capture and are not needed for backup and recovery.

It relies on 'log release' files that are re-created by log-based capture jobs after each replication cycle. These files are in **$HVR_CONFIG/files** and contain timestamps which allows **hvrlogrelease** to see which archive files could still be needed by HVR replication.

When **hvrlogrelease** is making private copies of archive files (option **-purge** not defined), they are placed in a special directory using a file system 'hard link' to avoid i/o overhead. The path is defined by action **Capture /ArchiveLogPath**. If that is not defined then a default path is derived by adding the suffix **_hvr** to the Oracle archive destination directory.

The files copied by **hvrlogrelease** keep their original 'base name', so parameter **/ArchiveLogFormat** cannot be used.

Command **hvrlogrelease** with option **-purge** can be used with action **Capture /ArchiveLogPath /ArchiveLogOnly** to remove files which the capture job no longer needs. In this case, an additional option **-force** must be used.

Functionality for Ingres

hvrlogrelease can be used to manage Ingres journal files so that they are available if needed by HVR log-based capture jobs. The default behavior is that **hvrlogrelease** monitors the directory where the DBMS creates these files, and makes a copy in a private directory whenever one appears. It then removes these copies when the HVR capture jobs will no longer need their contents.

An alternative behavior is activated by option **-purge**; in this case no copies are made and **hvrlogrelease** simply deletes DBMS logging files from the archive or journal directory once it sees these files will no longer be needed by HVR capture jobs. This is only useful if journaling was enabled for HVR's log-based capture and are not needed for backup and recovery.

It relies on 'log release' files that are re-created by log-based capture jobs after each replication cycle. These files are in **$HVR_CONFIG/files** and contain timestamps which allows **hvrlogrelease** to see which DBMS logging files could still be needed by HVR replication.

When **hvrlogrelease** makes private copies of Ingres journal files (option **-purge** not defined) they are placed in special directory using a file system 'hard link' to avoid i/o overhead. The path is defined by environment variable **$HVR_LOG_RELEASE_DIR**. If that is not defined then a default path is derived by adding the suffix **_hvr** to the database's journal directory.

Functionality for SQL Server

hvrlogrelease resets log file blocks from 'taken' into 'free' state. Such blocks can be later reused to prevent excessive log file growth. That log 'truncation' may be needed in the following situations:

- coexistence, when several brands of replication tools need to read same log file;
- multi-capture, when several capture jobs read the same log file;
- in order to reset the bunch of used log file blocks when log has not being truncated for a long period.

First two scenarios require configuring HVR capture job with automatic log truncation turned off. Then log file needs to be truncated by executing **hvrlogrelease** command from according job ran by schedule or any other means. For more information, see Managing SQL Server Log file truncation.

Notes

- **hvrlogrelease** must be scheduled to run under the DBMS owner's login (e.g. **oracle** or **ingres)** , whereas **hvrmaint** must run under the HVR's login.
- When **hvrlogrelease** is installed on an Oracle RAC, then the **$HVR_CONFIG** directory must be shared between all nodes. Directory **$HVR_TMP** (if configured) should not be shared. Command **hvrlogrelease** should then be scheduled to run on all nodes, but with 'interleaved' timings. For example 0, 20, 40 minutes after each hour on one node and 10, 30, 50 minutes after each hour on the other node.
- Command **hvrlogrelease** does not support archive files located inside Oracle ASM. In this situation the RMAN must be configured to retain the archive files for sufficient time for HVR.

Options

This section describes the options available for command **hvrlogrelease**.

Parameter	Description
-email=*addr1*[*;addr2*]	Email output from **hvrlogrelease** output to *addr1* [and *addr2*]. Requires either option **-smtp_server** or option **-mailer**. Multiple email addresses can be specified, either using a single **-email** option with values separated by a semicolon or using multiple **-email** options.
-email_only_errors	Only send an email if **hvrlogrelease** encounters an error.

-email_activity	Only send an email if **hvrlogrelease** encounters an error or purged or compressed any file.
-email_from=*from*	Specify a *from* address in email header.
-env=*NAME=VALUE*	Set environment variable, such as **$HVR_HOME**. This option can be repeated to set different variables. Values for **$ORACLE_SID** and **$HVR_CONFIG** should be defined with special options **-oracle_sid** and **-hvr_config** respectively.
-help	Provide options list with service information to create UI dialog.
-hvr_config=*dir*	Check for 'log release' files in this **$HVR_CONFIG** directory. This option must be supplied at least once. It can also be supplied several times if multiple HVR installations capture log changes from a single database. **hvrlogrelease** will then purge DBMS logging files only after they have been released by all the HVR installations. If value *dir* contains an asterisk (*) then all matching directories are searched.
-ingres_db=*db*	Only check 'log release' files for Ingres database *db*. If value db contains an asterisk (*) then all Ingres databases are matched. This option can be supplied several times for different databases. Note that **hvrlogrelease** extracts the path **$II_SYSTEM** from matching log release files, so it could affect journal files which are not in the current Ingres installation.
-logrelease_expire=*Nunits*	Instruct command **hvrlogrelease** to ignore any 'log release file' that are too old. Value *units* can be **days**, **hours** or **minutes**. For example, value **4days** could be defined because if the capture job has not run for four days then the replication backlog is so great that a refresh will be needed and the DBMS logging files can be discarded.
-mailer=*cmd*	Mailer command to use for sending emails, instead of sending them via an SMTP server. Requires option **-email**. String **%s** contained in *cmd* is replaced by the email subject and string **%a** is replaced by the intended recipients of the email. The body of the email is piped to *cmd* as **stdin**.
-oracle_sid=*sid*	Only check for 'log release' files for Oracle instance *sid*. If value *sid* contains an asterisk (*) then all Oracle instances are matched. This option can be supplied several times for different instances. Note that **hvrlogrelease** extracts the value of **$ORACLE_HOME** from matching log release files, so it could affect archived redo files which are not in the current Oracle installation.
-output=*fil*	Append **hvrlogrelease** output to file *fil*. If this option is not supplied then output is sent to **stdout**. Output can also be sent to an operator using option **-email**.
-state_dir=*dir*	Create files **hvrlogrelease.pid** and **hvrlogrelease.dirs** in provided directory *dir*. If not supplied these files will be created in directory set in environment variable **$HVR_CONFIG**.
-purge	Purge (i.e. delete) old DBMS logging files (Ingres journals and Oracle archived redo logfiles) and backup files (e.g. Ingres checkpoints) once the 'log release' files indicate that they are no longer needed for any HVR replication. If the 'log release' file is absent or unchanged then nothing is purged, so operators must purge journals/archived redo files manually. This is only useful if journaling /archiving was only enabled for HVR's log-based capture and are not needed for backup and recovery.
-smtp_server=*server*	SMTP server to use when sending email. Value *server* can be either a node name or IP address. Requires option **-email**.
-smtp_user=*user*	Username user for authentication SMTP server if needed.
-smtp_pass=*pass*	Password pass used for authentication on the SMTP server if needed.
-user=*login/pass*	Database authentication login and password.

-sqlserver_db=*db*	Mark SQL Server log file blocks free for database db.
-verbose	Write extra logging to output file.

Deleting Ingres Journals with CKPDB

Command **hvrlogrelease** can encounter problems if Ingres command **ckpdb** is used with option **-d** (delete journal files). This is because Ingres will create new journal files when 'draining' the log file and then delete them so quickly that **hvrlogrelease** does not have time to make a copy. The solution is to create a customized Ingres 'checkpoint template' file which calls **hvrlogrelease** from within **ckpdb**. Two lines of this file need to be changed: those labeled **PSDD** and **PSTD**. The following steps use a Perl expression to change them to call **hvrlogrelease** with specific option file (this file must also be created).

```
$ cd $II_SYSTEM/ingres/files
$  perl  -pe  's/(PS[DT]D:\s*)/$1hvrlogrelease  \$HVR_CONFIG\/files\/hvrlogrelease.opt;/'
cktmpl.def >cktmpl_hvr.def
$ ingsetenv II_CKTMPL_FILE $II_SYSTEM/ingres/files/cktmpl_hvr.def
```

Examples

Example 1 - manage Oracle archive file copies
The following can be saved in option file **/opt/hvrlogrelease.opt** so that private copies of any Oracle archive files from instance **ORA1020** are available when needed by HVR log-based capture:

```
-env=HVR_HOME=/opt/hvr410/hvr_home
-hvr_config=/opt/hvr410/hvr_config
-oracle_sid=ORA1020
-logrelease_expire=3days
-email_from=hvr@prod.mycorp.com
-email=bob@mycorp.com;jim@mycorp.com
-email_only_errors
```

If Oracle command **rman** is also configured to remove old redo logfiles, then **hvrlogrelease** must be scheduled to run first so that it sees every file before that file is removed by **rman**. This can be done by scheduling both **hvrlogrelease** and **rman** in a single crontab line.

```
00,15,30,45 * * * * su oracle -c "/opt/hvr_home/bin/hvrlogrelease  /opt/hvrlogrelease.opt
>> /tmp/hvrlogrelease.log 2>&1 && rman @delete_archives >>  /tmp/delete_archives.log 2>&1"
```

Example 2 - manage Ingres journal file copies
The following option file can be used to maintain private copies of any Ingres journal files for database **mydb** so that they are available when needed by HVR log-based capture:

```
-env=HVR_HOME=/opt/hvr410/hvr_home
-hvr_config=/opt/hvr410/hvr_config
-ingres_db=mydb
-email_from=hvr@prod.mycorp.com
-email=bob@mycorp.com;jim@mycorp.com
-email_only_errors
```

Example 3 - mark SQL Server log blocks as free
The following option file can be used to move Log Sequence Number replication pointer for database **mydb** to mark virtual log file blocks as free:

```
-sqlserver_db=mydb
-user=my_login/my_pwd
-output=c:\hvr\hvr_config\log\hvrlogrelease-hub-mydb.out
```

Files

- **HVR_CONFIG**
  - **files**

hvrlogrelease.dir	Cache of search directories.
hvrlogrelease.pid	Process ID of current **hvrlogrelease** command.
[*node-*]*hub-chn-loc*.**logrelease**	Log release file, containing the time. Recreated by each log-based capture job cycle and used by **hvrlogrelease**.

Hvrmaint

Contents

Name

hvrmaint - Housekeeping script for HVR on the hub machine.

Synopsis

hvrmaint [*optfile*] [*-options*]

Description

Command **hvrmaint** is a script for regular housekeeping of the HVR on the hub machine. The script can run on the hub machine and can be scheduled on Unix using crontab or as a Windows scheduled task.

The **hvrmaint** can be used for three main purposes:

1. **Maintenance:** Schedule **hvrmaint** nightly (or weekly) with options **-stop** and **-start.** These options instruct **hvrmaint** to restart the **HVR Scheduler**. Often other options can be used, such as **-scan_hvr_out** (scan log files for HVR errors) or **-archive_files** (move old log files to archive directory **$HVR_CONFIG/logarchive/***hub_name***/***day*). **Email alerts** can be used to send an email with the status summary to operator(s). When used in this way, **hvrmaint** could be scheduled on Unix using crontab, and on Windows as a Windows Scheduled Task.
2. **Monitoring:** Run **hvrmaint** frequently (e.g. every 15 minutes) with options **-scan_hvr_out**, **-test_scheduler**, and **-check_logfile_growth** to check if the **HVR Scheduler** is running and to scan the HVR log files for errors. Running **hvrmaint** this way does not interrupt the **HVR Scheduler**. There is option **-email_only_when_errors** to send emails only if an error has occurred.
3. **Backup:** The last way to use **hvrmaint** is as part of a larger nightly or weekly batch script, which halts all server processes (including the DBMS), does a system backup and then restarts everything again. In this case, **hvrmaint** would be called at the top of the batch script with option **-stop** (stop the **HVR Scheduler**) and would then be called again near the bottom with option **-start** (restart the **HVR Scheduler**).

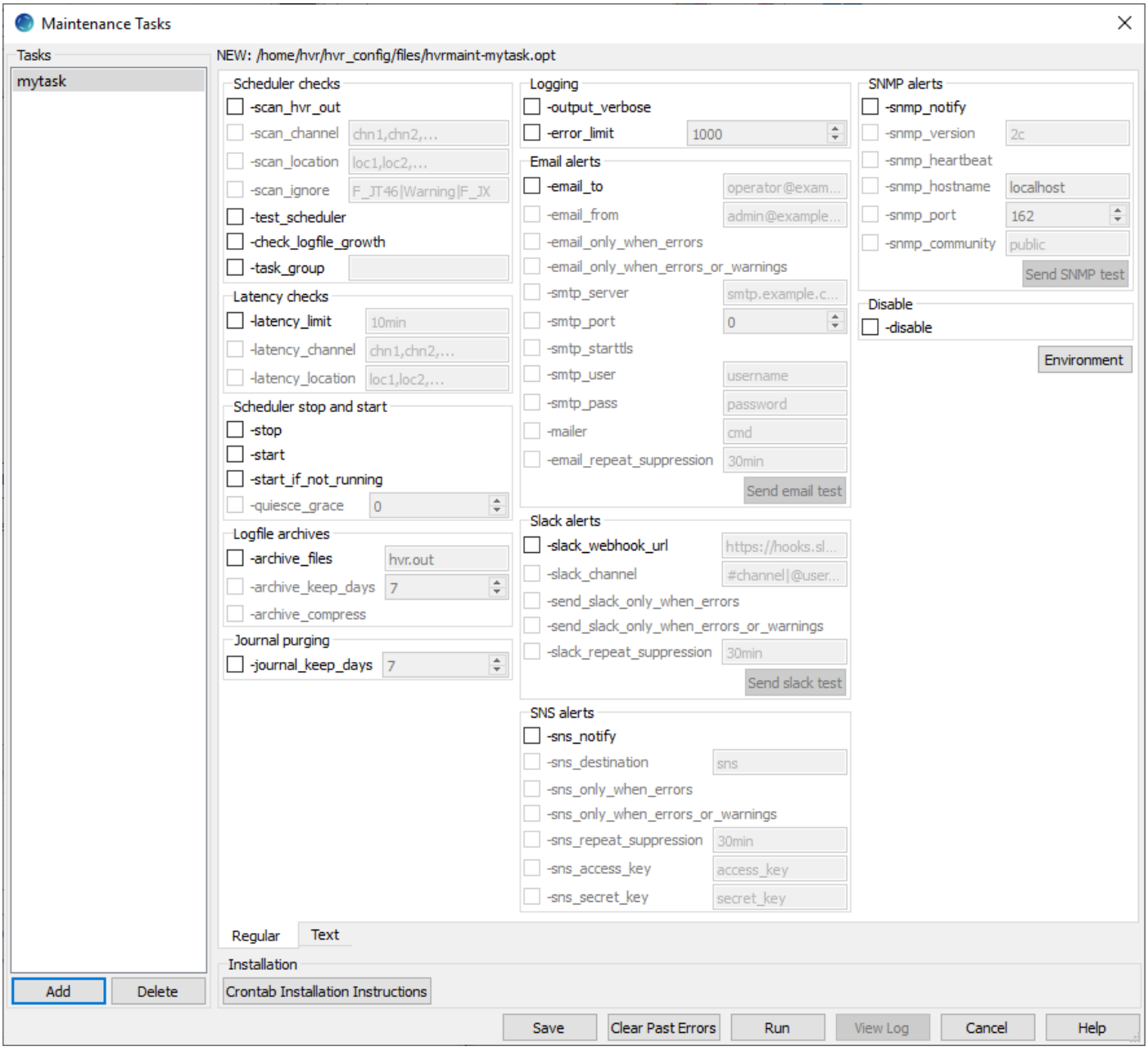
⚠ Command **hvrmaint** cannot process log files containing more than 12 months of data.

Creating Maintenance Task

The behavior of **hvrmaint** is controlled by an option file, which can be generated using the HVR **Maintenance Tasks** dialog available from the context menu of the **Scheduler**.

1. Right-click the **Scheduler** node and select **Maintenance Tasks** from the context menu. The **Maintenance Tasks** dialog will open containing the list of tasks on the left pane (if they were previously created) and configuration options on the right pane (see the description of options available for command **hvrmaint** below).
2. Click the **Add** button at the bottom of the left pane to create a new maintenance task (option file). Type the name of the task and click **OK**.
3. Select the required options, specify parameters for them, where needed and click **Save**.
4. Click **Run** to run the **hvrmaint** script you created against the hub. You can click **View Log** to watch the output of the script.
5. The time options on the bottom pane allow you to schedule the task to run at a specific time, namely at regular intervals, daily or weekly.

> ⚠ Select **Highest Privileges** option to run the task with administrative permissions.



Options

This section describes the options available for command **hvrmaint**.

Parameter	Description
-task_name=*task*	Task name is used internally by **hvrmaint** to locate its option file and name its offset files. This allows different tasks defined in the GUI to have a different state. e.g. so that a when a task for one channel has processed today's files a different task for a different channel still remembers to process today's files.
Scheduler checks	
-scan_hvr_out	Scan Scheduler log file **hvr.out**. Command **hvrmaint** writes a summary of HVR errors detected in this file to its output and to any emails that it sends.
-scan_channel=*chn*	Only scan for general errors and errors in specified channel(s) *chn*. Requires option **-scan_hvr_out**.
-scan_location=*loc*	Only scan for general errors and errors in specified locations(s) *loc*. Requires option **-scan_hvr_out**.
-scan_ignore=*patt*	Ignore log records which match specified pattern *patt* (can be regular expression). Requires option **-scan_hvr_out**.
-test_scheduler	Check that HVR Scheduler is actually running using **hvrtestscheduler**. If option **-stop** is also defined then this test is performed before the **HVR Scheduler** is stopped. If option **-start** is supplied then **hvrmaint** always checks that the **HVR Scheduler** is running using a test, regardless of whether or not option **-test_scheduler** is defined.
-check_logfile_growth	Check that logfile **hvr.out** has grown in size since the last time **hvrmaint** was run. If this file has not grown then an error message will be written. This option should be used with **-scan_hvr_out**.
-task_group=*group*	Task group allows different **hvrmaint** tasks to share the same state. So a nightly task that processes log files and gives a warning if the latency is >1 hour can use the same 'offset state' as a task that runs during the day which gives a warning if latency is >1 minute.
Latency checks	
-latency_limit=*dur*	Check for replication latencies and consider jobs over the limit erroneous. Value for *dur* can be specified in one of the following formats: • *N* **d** or *N* **day** or *N* **days** - Indicates number of days • *N* **h** or *N* **hour** or *N* **hours** - Indicates number of hours • *N* **m** or *N* **min** or *N* **mins** or *N* **minute** or *N* **minutes** - Indicates number of minutes • *N* **s** or *N* **sec** or *N* **secs** or *N* **second** or *N* **seconds** - Indicates number of seconds • *HH:MM:SS*[*.SSS*] - Indicates time format with hour, minute, second, and millisecond • *MM:SS*[*.SSS*] - Indicates time format with minute, second, and millisecond ⚠ Milliseconds [*.SSS*] is ignored by this parameter.

-latency_channel=*chn*	Only check latencies of jobs in specified channel(s) *chn*.
-latency_location=*loc*	Only check latencies of jobs in specified location(s) *loc*.
Scheduler stop and start	
-stop	Stop **HVR Scheduler**.
-start	Start **HVR Scheduler**.
-quiesce_grace=*secs*	If jobs are still running when the **HVR Scheduler** must stop, allow seconds *secs* grace before killing them. The **default** is 60 seconds. This parameter is passed with the **HVR Scheduler** using the **-q** option.
-start_if_not_running **Since** v5.6.5/13	Start **HVR Scheduler** if it is not already running. ⚠ This option will automatically start the **HVR Scheduler**, if the hub wallet is enabled and the method to supply wallet password is either **Auto-Open Password** or **Auto-Open Plugin.** However, if the method is **Manual**, the wallet password needs to be supplied by the user manually in the command line using the command **hvr walletopen** to start the **HVR Scheduler**. If the wallet password is not supplied within 30 seconds, then HVR prints error in the log stating that it tried to start **HVR Scheduler** but wallet password was not supplied. Until the wallet password is supplied, the error message is repeated each time the **hvrmaint** tries to start the **HVR Scheduler**.
Logfile archives	
-archive_files=*patt*	Move any files in directory **$HVR_CONFIG/log/***hub_name* matching pattern *patt* to the archive directory (**$HVR_CONFIG/logarchive/***hub_name***/***day*). Files that do not match pattern *patt* are deleted. Pattern *patt* is a regular expression. For example: • **.*\.out** – matches all files ending with **.out**. • **hvr.*** - matches all files starting with 'hvr'. • **.*\.(out|err)** - multiple patterns may be specified. In this case, all **.out** files and all **.err** files will be moved to the archive directory.
-archive_keep_days=*N*	Retain files in archive directory (**$HVR_CONFIG/logarchive/***hub_name***/***day*) for *N* number of days. The retained files are deleted after *N* number of days. Requires option **-archive_files**. If this option is not specified, then archived files are kept indefinitely.
-archive_compress	Compress HVR Scheduler log files while moving them to the archive directory (**$HVR_CONFIG/logarchive/***hub_name***/***day*). For a Windows hub, this option can only be used if command **gzip** has been installed.
Journal purging	
-journal_keep_days=*n*	Retain HVR journal files in directory **$HVR_CONFIG/jnl** for *n* number of days. The files are deleted after *n* number of days. These files are written by integrate jobs if parameter **Integrate /JournalRouterFiles** is defined.
Logging	

-output_verbose	Prints latency information and error summaries to **hvrmaint** output.
Email alerts	
-email_to=*addr1*[*;addr2*]	Send the output from **hvrmaint** as email to the specified email address *addr1* [and *addr2*]. Requires either option **-smtp_server** or option **-mailer**. Multiple email addresses can be specified with values separated by a semicolon or using multiple **-email_to** options (only in CLI).
-email_from=*addr*	Specify a sender address *addr* in email header.
-email_only_when_errors	Send an email if **hvrmaint** encountered an error itself or detected an HVR error while scanning **hvr.out** or the latency limit is exceeded.
-email_only_when_errors_or_warnings	Send an email if **hvrmaint** encountered an error itself or detected an HVR error or warning while scanning **hvr.out** or the latency limit is exceeded.
-error_limit=*N*	HVR errors reported is limited to *N* number. **Default** is 1000. This option prevents the generated emails becoming too large.
-smtp_server=*server*	SMTP server to use when sending an email. Value *server* can be either a node name or IP address. Requires option **-email**.
-smtp_port **Since** v5.6.5/2	SMTP port to use when sending an email.
-smpt_starttls **Since** v5.6.5/2	Use the STARTTLS method to communicate with the SMTP server.
-smtp_user=*user*	Username *user* for authentication SMTP server if needed.
-smtp_pass=*pass*	Password *pass* used for authentication on the SMTP server if needed.
-mailer=*cmd*	Mailer command to use for sending emails, instead of sending them via an SMTP server. Requires option **-email**. String **%s** contained in *cmd* is replaced by the email subject and string **%a** is replaced by the intended recipients of the email. The body of the email is piped to *cmd* as **stdin**. E. g. on Linux: **-mailer**=**/bin/mail -s %s %a**
-email_repeat_supression=*dur* **Since** v5.6.5/11	Suppress repetition of the same email alert for the specified duration *dur*. By default, each time when **hvrmaint** encounters an error itself or detects an HVR error or warning while scanning **hvr.out** or the latency limit is exceeded, the **hvrmaint** sends out an alert until the issue is fixed. The number of the alerts sent depends on the frequency in which **hvrmaint** runs. As long as the issue is not resolved or the error/warning has not changed, **hvrmaint** will repeatedly send alerts for the same issue. To avoid repeatedly sending alerts for the same issue, this option forces **hvrmaint** to remain silent for specified duration *dur* after the first alert is sent out. Value for *dur* can be specified in one of the following formats: • *N* **d** or *N* **day** or *N* **days** - Indicates number of days • *N* **h** or *N* **hour** or *N* **hours** - Indicates number of hours • *N* **m** or *N* **min** or *N* **mins** or *N* **minute** or *N* **minutes** - Indicates number of minutes • *N* **s** or *N* **sec** or *N* **secs** or *N* **second** or *N* **seconds** - Indicates number of seconds

	• *HH:MM:SS*[*.SSS*] - Indicates time format with hour, minute, second, and millisecond • *MM:SS*[*.SSS*] - Indicates time format with minute, second, and millisecond ⚠ Milliseconds [*.SSS*] is ignored by this parameter.
Slack alerts	
-slack_webhook_url=*url*	A webhook for a Slack channel in company **MyCorp** looks like **https://hooks.slack.com/services/***xxxx/yyyy*. To generate a Slack webhook, sign into Slack, then navigate to **Apps Manage apps Custom Integrations Incoming WebHooks Add Configuration**.
-slack_channel=*chn*	**Hvrmaint** will send the message to the specified Slack user (@username) or channel *chn*. This optional field can be used to override the Slack user or channel defined in the Slack webhook (**-slack_webhook_url**).
-send_slack_only_when_errors	Send a Slack message if **hvrmaint** encountered an error itself or detected an HVR error while scanning **hvr.out** or the latency limit is exceeded.
-send_slack_only_when_errors_or_warnings	Send a Slack message if **hvrmaint** encountered an error itself or detected an HVR error or warning while scanning **hvr.out** or the latency limit is exceeded.
-slack_repeat_supression=*dur* **Since** v5.6.5/11	Suppress repetition of the same Slack alert for the specified duration *dur*. By default, each time when **hvrmaint** encounters an error itself or detects an HVR error or warning while scanning **hvr.out** or the latency limit is exceeded, the **hvrmaint** sends out an alert until the issue is fixed. The number of the alerts sent depends on the frequency in which **hvrmaint** runs. As long as the issue is not resolved or the error/warning has not changed, **hvrmaint** will repeatedly send alerts for the same issue. To avoid repeatedly sending alerts for the same issue, this option forces **hvrmaint** to remain silent for specified duration *dur* after the first alert is sent out. Value for *dur* can be specified in one of the following formats: • *N* **d** or *N* **day** or *N* **days** - Indicates number of days • *N* **h** or *N* **hour** or *N* **hours** - Indicates number of hours • *N* **m** or *N* **min** or *N* **mins** or *N* **minute** or *N* **minutes** - Indicates number of minutes • *N* **s** or *N* **sec** or *N* **secs** or *N* **second** or *N* **seconds** - Indicates number of seconds • *HH:MM:SS*[*.SSS*] - Indicates time format with hour, minute, second, and millisecond • *MM:SS*[*.SSS*] - Indicates time format with minute, second, and millisecond ⚠ Milliseconds [*.SSS*] is ignored by this parameter.

SNS alerts Since v5.6.5/2	
-sns_notify	Send notification to Amazon Simple Notification Service (SNS).
-sns_destination	Amazon Resource Name (ARN) of the SNS topic.
-sns_only_when_errors	Send a notification to Amazon SNS if **hvrmaint** encountered an error itself or detected an HVR error while scanning **hvr.out** or the latency limit is exceeded.
-sns_only_when_errors_or_warnings	Send a notification to Amazon SNS if **hvrmaint** encountered an error itself or detected an HVR error or warning while scanning **hvr.out** or the latency limit is exceeded.
-sns_repeat_supression=*dur* Since v5.6.5/11	Suppress repetition of the same SNS alert for the specified duration *dur*. By default, each time when **hvrmaint** encounters an error itself or detects an HVR error or warning while scanning **hvr.out** or the latency limit is exceeded, the **hvrmaint** sends out an alert until the issue is fixed. The number of the alerts sent depends on the frequency in which **hvrmaint** runs. As long as the issue is not resolved or the error/warning has not changed, **hvrmaint** will repeatedly send alerts for the same issue. To avoid repeatedly sending alerts for the same issue, this option forces **hvrmaint** to remain silent for specified duration *dur* after the first alert is sent out. Value for *dur* can be specified in one of the following formats: • *N* **d** or *N* **day** or *N* **days** - Indicates number of days • *N* **h** or *N* **hour** or *N* **hours** - Indicates number of hours • *N* **m** or *N* **min** or *N* **mins** or *N* **minute** or *N* **minutes** - Indicates number of minutes • *N* **s** or *N* **sec** or *N* **secs** or *N* **second** or *N* **seconds** - Indicates number of seconds • *HH:MM:SS*[*.SSS*] - Indicates time format with hour, minute, second, and millisecond • *MM:SS*[*.SSS*] - Indicates time format with minute, second, and millisecond ⚠ Milliseconds [*.SSS*] is ignored by this parameter.
-sns_access_key	Access key ID of the AWS IAM user. For more information about access key, refer to Managing Access Keys for IAM Users in AWS documentation.
-sns_secret_key	Secret access key of the AWS IAM user. For more information about secret key, refer to Managing Access Keys for IAM Users in AWS documentation.
SNMP alerts	
-snmp_notify	Send SNMP v1 traps or v2c notifications. The **-snmp_community** option is required. See **$HVR_HOME/lib/mibs/HVR-MIB.txt**
-snmp_version=*vers*	Specify SNMP version. Value *vers* can be **1** or **2c** (**default**).

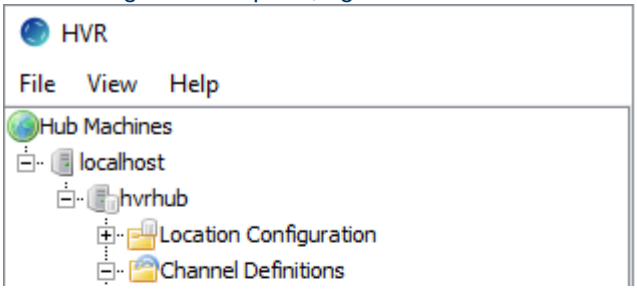
-snmp_heartbeat	Send a **hvrMaintNotifySummary** notification, even if there was nothing to report.
-snmp_hostname=*host*	SNMP agent hostname *host*. **Default** is localhost.
-snmp_port=*port*	SNMP agent trap *port*. **Default** is port 162.
-snmp_community=*str*	Community string *str* for SNMPv1/v2c transactions.
Disable	
-disable	Disable **hvrmaint** alerts. This option allows to disable the alerts without stopping the **hvrmaint**. This can be useful during a maintenance window when channels are being modified or stopped. An alternative is to stop running **hvrmaint** during the maintenance window and restart it after, but this can generate a lot of alerts caused by the maintenance.
-env=*NAME=VALUE*	Set environment variable. This option can be repeated to set multiple variables such as **$HVR_HOME**, **$HVR_CONFIG**, **$HVR_TMP**, **$II_SYSTEM**, **$ORACLE_HOME** etc. In HVRGUI, to view this option, click on the **Environment** button.
-hub=*hub*	Hub database for **HVR Scheduler**. This value has form *user/pwd* (for an Oracle hub), *inghub* (for an Ingres hub database), or *hub* for a (SQL Server hub database). For Oracle, passwords can be encrypted using command **hvrcrypt**. In HVRGUI, this option is only available inside the **Text** tab.
-sched_option=*schedopt*	Extra startup parameters for the HVR Scheduler service. Possible examples are **-u***user/pwd* (for a username), -**hsqlserver** (for the hub class) or **-c***clus/clusgrp* (for Windows cluster group). In HVRGUI, this option is only available inside the **Text** tab.
-output=*fil*	Append **hvrmaint** output to file *fil*. If this option is not supplied, then output is sent to **stdout**. Output can also be sent to an operator using option **-email**. In HVRGUI, this option is only available inside the **Text** tab.

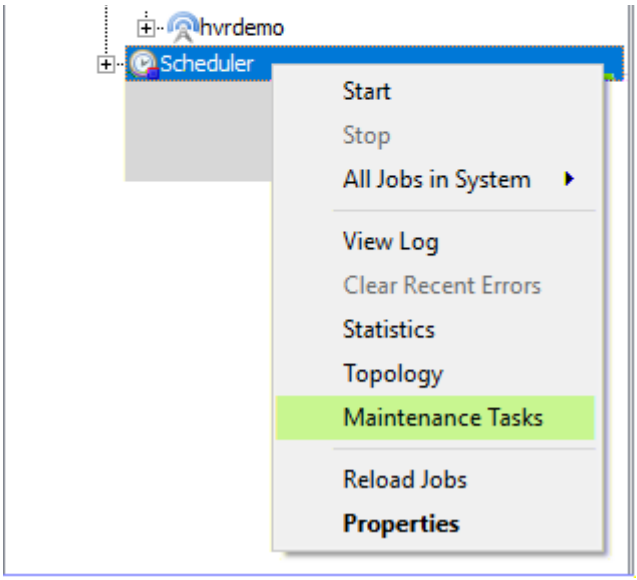
Configuring HVR Email Alerts Using Gmail SMTP

Following are the steps to set up HVR to send **hvrmaint** email alerts via Gmail SMTP server.

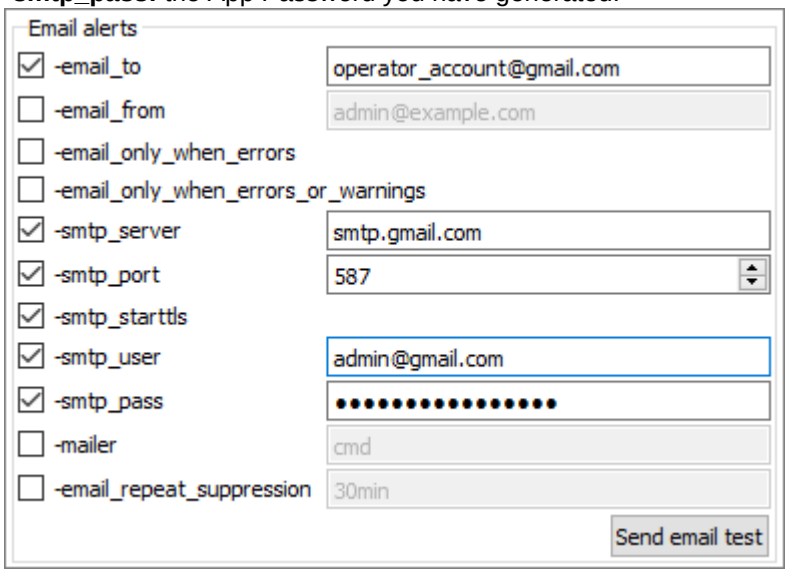
Before proceeding, you need to generate App Password (**-smtp_pass**) for the Gmail account (**-smtp_user**) that will be used to authenticate with the Gmail SMTP server (**-smtp_server**). Also, ensure that the two-factor authentication is activated for the Gmail address (**-smtp_user**). After generating the App Password, perform the following steps in HVR GUI to create a maintenance task:

1. In the navigation tree pane, right-click **Scheduler** ▸ **Maintenance Tasks**.





2. In the **Maintenance Tasks** dialog, click **Add** in the left bottom. Type the name of the task and click **OK**.
3. Enter the following values under the **Email alerts** section:
   a. **-email_to:** Email address(es) to which **hvrmaint** alerts will be sent.
   b. **-smtp_server:** this is the address of the Gmail SMTP server - **smtp.gmail.com**.
   c. **-smtp_port:** the Gmail SMTP server port for using TLS/STARTTLS - **587**.
   d. Select **-smtp_starttls** to enable STARTTLS for secure connection.
   e. **-smtp_user:** the Gmail address to authenticate with the Gmail SMTP server. This is the Gmail account, from which the **hvrmaint** email alerts will be sent.
   f. **-smtp_pass:** the App Password you have generated.



4. Click **Save** and the task will be added to the list of tasks on the left panel.
5. To run the task manually, select the task in the list and click **Run**. This task will also run automatically if one of the conditions were defined: **-email_only_when_errors** or **-email_only_when_errors_or_warnings**.

Examples

Unix & Linux

- On Unix, **hvrmaint** could be scheduled to monitor the status of HVR every hour and also to restart the HVR Scheduler and rotate log files at 21:00 each Saturday. The environment for such batch programs is very limited, so many **-env** options are needed to pass it sufficient environment variables.

  Two option files are prepared. The first option file **/usr/hvr/hvr_config/files/hvrmaint_hourly.opt** will just check for errors and contains the following:

```
-hub=hvr/!\{s8Dhx./gsuWHUt\}!   # Encrypted Oracle password
-sched_option=-h oracle
-env=HVR_HOME=/usr/hvr/hvr_home
-env=HVR_CONFIG=/usr/hvr/hvr_config
-env=HVR_TMP=/tmp
-env=ORACLE_HOME=/distr/oracle/OraHome817
-env=ORACLE_SID=ORA817
-email_from=hvr@prod.mycorp.com
-email_to=bob@mycorp.com;jim@mycorp.com
-email_only_when_errors
-snmp_server=snmp.mycorp.com
-output=/usr/hvr/hvr_config/files/hvrmaint.log
-scan_hvr_out
```

The second option file **/usr/hvr/hvr_config/files/hvrmaint_weekly.opt** will restart the HVR Scheduler and rotate the log files each week.

```
-hub=hvr/!\{s8Dhx./gsuWHUt\}!    # Encrypted Oracle password
-sched_option=-h oracle
-env=HVR_HOME=/usr/hvr/hvr_home
-env=HVR_CONFIG=/usr/hvr/hvr_config
-env=HVR_TMP=/tmp
-env=ORACLE_HOME=/distr/oracle/OraHome817
-env=ORACLE_SID=ORA817
-email_from=hvr@prod.mycorp.com
-email_to=bob@mycorp.com;jim@mycorp.com
-email_only_when_errors
-snmp_server=snmp.mycorp.com
-output=/usr/hvr/hvr_config/files/hvrmaint.log
-scan_hvr_out
-stop
-archive_files=hvr.out    # Only archive log file hvr.out
-archive_compress
-archive_keep_days=14      # Delete files after 2 weeks
-journal_keep_days=4
-start
```

The following lines are added to **crontab** for user **hvr** (these should be single lines without wrapping):

```
0 * * * * /usr/hvr/hvr_home/bin/hvrmaint /usr/hvr/hvr_config/files/hvrmaint_hourly.
opt
0 21 * * * /usr/hvr/hvr_home/bin/hvrmaint /usr/hvr/hvr_config/files/hvrmaint_weekly.
opt
```

Alternatively the following line could be added to **crontab** for **root**:

```
0 21 * * 6 su hvr -c /usr/hvr/hvr_home/bin/hvrmaint /usr/hvr/hvr_config/files
/hvrmaint_weekly.opt
```

Instead of scheduling **hvrmaint** on its own, it could also be used as part of a larger nightly batch script run by **root** which halts the HVR Scheduler and DBMS before doing a system backup. This batch script would roughly look like this:

```
su hvr -c /usr/hvr/hvr_home/bin/hvrmaint /opt/hvrmaint.opt -stop -scan_hvr_out -
archive_files=hvr.out<br />
su ingres -c /opt/ingres/utility/ingstop  # Stop DBMS
backup -f/dev/rmt/0m    # Perform system backup<br />
```

```
su ingres -c /opt/ingres/utility/ingstart  # Restart DBMS<br />
su hvr -c /usr/hvr/hvr_home/bin/hvrmaint /opt/hvrmaint.opt -start
```

Windows

- On Windows, **hvrmaint** can be run as a Windows Scheduled Task.

  The configuration steps are as follows:

  1. Click **Start Settings Control Panel Scheduled Tasks**.
  2. Create a new Scheduled Task by clicking **File New Scheduled Task**.
  3. Double-click **Add Scheduled Task**.
  4. On the **Tasks** tab enter the **hvrmaint** command line in the **Run** field, for example:

     ```
     c:\hvr\hvr_home\bin\hvrmaint.exe
     c:\hvr\hvr_config\files\hvrmaint_hourly.opt
     ```

  5. On the **Schedule** tab, configure when the **hvrmaint** script should run.
  6. When ready click the **OK** button.
  7. A dialog now appears requesting Windows account information (username and passwords). Enter this information as requested.

  A sample option file for hourly monitoring could be:

  ```
  -hub=hvr/!{s8Dhx./gsuWHUt}!   # Encrypted Oracle password
  -sched_option=-h oracle
  -env=HVR_HOME=c:\\opt\\hvr_home
  -env=HVR_CONFIG=c:\\opt\\hvr_config
  -env=HVR_TMP=c:\\temp
  -env=ORACLE_HOME=c:\\distr\\oracle\\OraHome817
  -env=ORACLE_SID=ORA817
  -email_from=hvr@prod.mycorp.com
  -email=bob@mycorp.com;jim@mycorp.com
  -email_only_when_errors
  -snmp_server=snmp.mycorp.com
  -output=c:\\opt\\hvr_config\\files\\hvrmaint.log
  -scan_hvr_out<br /><br />
  ```

  A sample option file for weekly restart of HVR on Windows would be:

  ```
  -hub=hvr/!{s8Dhx./gsuWHUt}!    # Encrypted Oracle password
  -sched_option=-h oracle
  -env=HVR_HOME=c:\\opt\\hvr_home
  -env=HVR_CONFIG=c:\\opt\\hvr_config
  -env=HVR_TMP=c:\\temp
  -env=ORACLE_HOME=c:\\distr\\oracle\\OraHome817
  -env=ORACLE_SID=ORA817
  -email_from=hvr@prod.mycorp.com
  -email=bob@mycorp.com;jim@mycorp.com
  -email_only_when_errors
  -snmp_server=snmp.mycorp.com
  -output=c:\\opt\\hvr_config\\files\\hvrmaint.log
  -scan_hvr_out
  -stop
  -archive_files=hvr.out    # Only archive log file hvr.out
  -archive_keep_days=14      # Delete files after 2 weeks
  -journal_keep_days=4
  -start
  ```

Sample Output

```
From: root@bambi.mycorp.com
To: bob@mycorp.com; jim@mycorp.com
Subject: hvrmaint detected 7 errors (323 rows in fail tables) for hub hvr/ on bambi
2017-11-01T21:00:01-06:30 hvrmaint: Starting hvrmaint c:\tools\hvrmaint.opt -hub=hvr/ -
stop -start
2017-11-01T21:10:21-06:30 hvrmaint: Stopping HVR Scheduler 4.4.4/5 (windows-x64-64bit).
2017-11-01T21:10:33-06:30 hvrmaint: Scanning d:\hvr_config\log\hvr\hvr.out (2017-11-01T21:
00:03-06:30).
2017-11-01T21:11:13-06:30 hvrmaint: 7 errors (323 rows in fail tables) were detected
during scan.
2017-11-01T21:12:33-06:30 hvrmaint: 3 capture jobs for 1 location did 606 cycles.
2017-11-01T21:12:59-06:30 hvrmaint: 6 integrate jobs for 2 locations did 400 cycles and
integrated 50 changes for 3 tables.
2017-11-01T21:13:53-06:30 hvrmaint: Archiving 9 log files to d:
\hvr\archivelog\hvr_20050209.
2017-11-01T21:16:23-06:30 hvrmaint: Purging 0 archive directories older than 14 days.
2017-11-01T21:18:29-06:30 hvrmaint: Starting HVR Scheduler 4.4.4/5 (windows-x64-64bit).

-------------------- Summary of errors detected during scan-----------------
F_JD1034_RAISE_ERROR_P3 occurred 6 times between 2017-11-01T19:43:52-06:30 and 2017-11-
01T20:14:24-06:30
F_JJ106E_TIMEO_DB occurred 1 time at 2017-11-01T21:10:03-06:30

-------------------------- Errors detected during scan--------------------------
 2017-11-01T19:43:52-06:30: channel-cap-d01: F_JD1034_RAISE_ERROR_P3: Error as raised by
user during pl/sql procedure statement on Oracle SID.

---------------------- End of errors detected during scan ----------------------
 2017-11-01T21:19:01 hvrmaint: Sending e-mail to bob@mycorp.com; jim@mycorp.com
```

Files

HVR_CONFIG	
log	
hubdb	HVR Scheduler log files.
hvr.out	Main Scheduler log file.
.hvrmaint_state	**hvrmaint** state file.
logarchive	
hubdb	
YYYYMMDD	
hvr.out	Archived Scheduler log file. These files are created if **hvrmaint** option **-archive_files** is defined and deleted again if option **-archive_keep_days** is defined.
hvr.out.gz	Archived Scheduler log file if -**archive_compress** is defined.

Hvrproxy

Contents

Name

hvrproxy - HVR proxy.

Synopsis

hvrproxy [*-options*] *portnum access_conf.xml*

Description

HVR Proxy listens on a TCP/IP port number and invokes an **hvr** process with option **-x** (proxy mode) for each connection. The mechanism is the same as that of configuring an HVR proxy with the Unix daemon **inetd**.

On Windows, HVR Proxy is a Windows Service which is administered with option **-a**. The account under which it is installed must be member of the Administrator group, and must be granted privilege to act as part of the Operating System (**SeTcbPrivilege**). The service can either run as the default system account, or (if option **-P** is used) can run under the HVR account which created the Windows Service.

On Unix and Linux, HVR Proxy runs as a daemon which can be started with option **-d** and killed with option **-k**.

After the port number *portnum* an access configuration file *access_conf.xml* must be specified. This file is used to authenticate the identity of incoming connections and to control the outgoing connections. If the access file is a relative pathname, then it should be located in **$HVR_HOME/lib**.

⚠
- HVR Proxy is supported on Unix and Linux but it is more common on these machines to configure proxies using the **inetd** process to call executable **hvr** with options **-a** (access control file) and **-x** (proxy mode).
- When running as a Windows service, errors are written to the Windows event logs (**Control Panel Adminitrative Tools Event Viewer Windows Logs Application**).

Options

This section describes the options available for command **hvrproxy**.

Parameter	Description

-a*x* **Windows**	Administration operations for Microsoft Windows system service. Values of *x* can be: • **c** : Create the HVR Proxy system service. • **s** : Start the HVR Proxy system service. • **h** : Halt (stop) the system service. • **d** : Destroy the system service. Several **-a***x* operations can be supplied together; allowed combinations are e.g. **-acs** (create and start) or **-ahd** (halt and destroy). HVR Proxy system service can be started (**-as**) and halted (**-ah**) from Windows Services (**Control Panel Administrative Tools Computer Management Services and Applications Services**).
-c*clus**clusgrp* **Windows**	Enroll the HVR Proxy service in a Windows cluster named *clus* in the cluster group *clusgrp*. Once the service is enrolled in the cluster it should only be stopped and started with the Windows cluster dialogs instead of the service being stopped and started directly (in the Windows Services dialog or with options **-as** or **-ah**). In Windows, failover clusters *clusgrp* is the network name of the item under **Services and Applications**. The group chosen should also contain the remote location; either the DBMS service for the remote database or the shared storage for a file location's top directory and state directory. The service needs to be created (with option **-ac**) on each node in the cluster. This service will act as a 'Generic Service' resource within the cluster. This option must be used with option **-a**.
-d	Start **hvrproxy** as a daemon process.
-E*name=value*	Set environment variable *name* to value *value* for the HVR processes started by this service.
-i	Interactive invocation. HVR Proxy stays attached to the terminal instead of redirecting its output to a log file.
-k	Stop **hvrproxy** daemon using the process-id in **$HVR_CONFIG/files/hvrproxy***port***.pid**.
-K*pair*	SSL encryption using two files (public certificate and private key) to match public certificate supplied by **/SslRemoteCertificate**. If *pair* is relative, then it is found in directory **$HVR_HOME/lib/cert**. Value *pair* specifies two files; the names of these files are calculated by removing any extension from *pair* and then adding extensions **.pub_cert** and **.priv_key**. For example, option **-Khvr** refers to files **$HVR_HOME/lib/cert/hvr.pub_cert** and **$HVR_HOME/lib/cert/hvr.priv_key.**
-P*pwd* **Windows**	Configure HVR Proxy service to run under the current login HVR account using password *pwd*, instead of under the default system login account. May only be supplied with option **-ac**. Empty passwords are not allowed. The password is kept (hidden) within the Microsoft Windows OS and must be re-entered if passwords change.

Examples

The following access control file will restrict access to only connections from a certain network and to a pair of hosts.

```
<hvraccess>
  <allow>
     <from>
       <network>123.123.123.123/4</network>  <ssl remote_cert="cloud"/>
     </from>
     <to> <host>server1.internal</host> <port>4343</port> </to>
     <to> <host>server2.internal</host> <port>4343</port> </to>
  </allow>
</hvraccess>
```

If this XML is written to the default directory **$HVR_HOME/lib**, then a relative pathname can be used (e.g. **hvrproxy.xml**).

Windows

To create and start a Windows proxy service to listen on port number 4343:

```
c:\> hvrproxy -acs -Kproxy 4343 hvrproxy.xml
```

Windows

To configure an HVR proxy on Unix, add the following line to the **xinetd** configuration.

```
server_args= -x -Kproxy -ahvrproxy.xml
```

This connection can be tested with the following command:

```
$ hvrtestlistener -Kcloud -Cproxy -Rproxy-host:4343 server1.internal 4343
```

Files

HVR_HOME	
bin	
hvr	Executable for remote HVR service.
hvrproxy	HVR Proxy executable.
lib	
hvrproxy_example.xml	Sample proxy access file.
HVR_CONFIG	
files	
hvrproxy*port*.pid	Process-id of daemon started with option **-d**.
log	
hvrproxy	
hvrproxy*port*.log	Logfile for daemon started with **-d**.

See Also

Command Hvr.

Hvrrefresh

Contents

Name

hvrrefresh - Refresh the contents of tables in the channel.

Synopsis

hvrrefresh [*-options*] *hubdb chn*

Description

Command **hvrrefresh** copies tables (available in a channel) from a source location to target location(s). The source must be a database location, but the targets can be databases or file locations.

The argument *hubdb* specifies the connection to the hub database. For more information about using this argument in the command line, see Calling HVR on the Command Line.

Hvrrefresh from a source location is supported only on certain location classes. For the list of supported source location classes, see Hvrrefresh and Hvrcompare from source location in Capabilities.

An HVR channel can be defined purely for hvrrefresh, instead of being used for replication (capture and integrate jobs). In this case, the channel must still be defined with actions **Capture** and **Integrate**, even though **hvrnit** will never be called.

Since v5.5.0/0

Integrate and refresh jobs cannot be run simultaneously because it can lead to data inconsistency. Therefore, when a refresh job is started, HVR forces the integrate job into **SUSPEND** state and creates a control file to block the integrate job from running. When the refresh job is completed, HVR automatically removes the control file and unsuspends the integrate job. Note that the integrate job is restored to its previous state before the **hvrrefresh** was executed.

The control files are created on the hub server in the directory **$HVR_CONFIG/router/***hubname***/***channelname***/control**. Multiple control files will be created if slicing (option **-S**) is used.

In case the refresh job fails and the block control files are not removed automatically, the integrate job cannot be restarted (or unsuspended); an error message is displayed when this happens. To resolve this error, remove the control files with names matching ***.ctrl-***channelname***-integ-***targetlocation***-*_block** from the hub directory **$HVR_CONFIG /router/***hubname***/***channelname***/control** and then manually **Unsuspend** the integrate job.

HVR Refresh Operation Type

For database and file targets, the refresh can be performed row-by-row or as a bulk operation, depending on which **-g** option is supplied.

Bulk Refresh

Bulk refresh means that the target object is truncated, and then bulk copy is used to refresh the data from the read location. During bulk refresh table indexes and constraints will be temporarily dropped or disabled and will be reset after the refresh is complete.

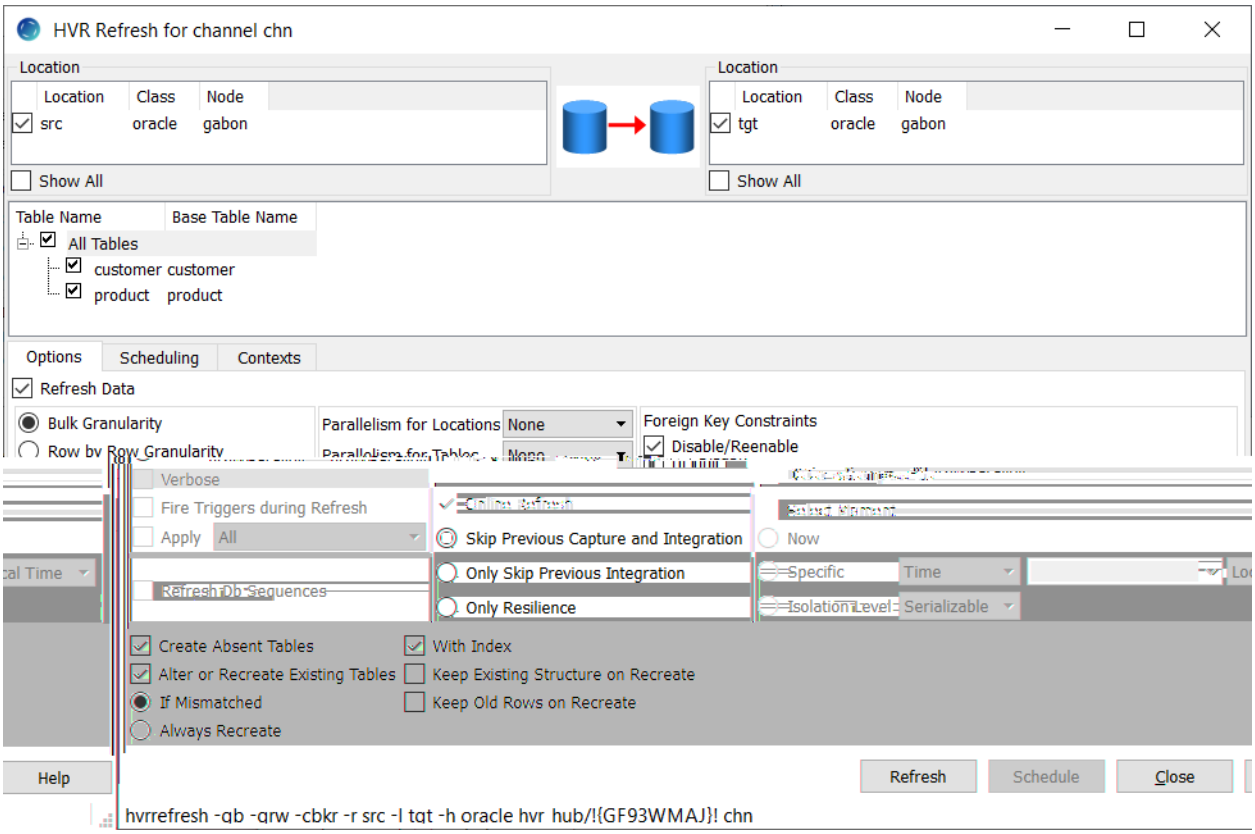
During bulk refresh, HVR typically streams data directly over the network into a bulk loading interface (e.g. direct path load in Oracle) of the target database. For DBMSs that do not support a bulk loading interface, HVR streams data into intermediate temporary staging files (in a staging directory) from where the data is loaded into the target database. For more information about staging files/directory, see section "Burst Integrate and Bulk Refresh" in the respective location class requirements.

Row by Row Refresh

Row-wise refresh compares data on read and write locations and produces a 'diff' result based on which only rows that differ are updated on the write location, each row is refreshed individually. This results in a list of a minimal number of inserts, updates or deletes needed to re-synchronize the tables.

Options

This section describes the options available for command **hvrrefresh**.



Parameter	Description

-c*S*	Instruct **hvrrefresh** to create new tables. Only 'basic' tables are created, based on the information in the channel. A basic table just has the correct column names and data types without any extra indexes, constraints, triggers or tables spaces. Value *S* can be one of the following: • **b** : Create basic (absent) tables only (mandatory). • **e** : Keep existing structure. • **f** : Force recreation (always recreate) of tables. • **k** : Create index (unique key or non-unique index). If the original table does not have a unique key, then a non-unique index is created instead of a unique key. • **o** : Only create tables, do not refresh any data into them. • **p** : Preserve (keep) existing data on recreate. • **r** : Recreate (drop and create) if the column names do not match. • **u** : Use Unicode data types such as **nvarchar** instead of **varchar**. Several **-cS** instructions can be supplied together, e.g. **-cbkr** which causes **hvrrefresh** to create new tables in the target database if they do not already exist and re-create if they exist but have the wrong column information. **DbObjectGeneration** **/RefreshCreateTableClause** can be used to add extra SQL to the **Create Table** statement which HVR will generate.
-C*context*	Enable *context*. This option controls whether actions defined with parameter **Context** are effective or are ignored. Defining an action with parameter **Context** can have different uses. For example, if action **Restrict** **/RefreshCondition="{id}>22" /Context=qqq** is defined, then normally all data will be refreshed, but if context **qqq** is enabled (**-Cqqq**), then only rows where **id>22** will be refreshed. Variables can also be used in the restrict condition, such as **"{id}>{hvr_var_min}"**. This means that **hvrrefresh -Cqqq -Vmin=99** will only refresh rows with **id>99**. Parameter **Context** can also be defined on action **ColumnProperties**. This can be used to define **/CaptureExpression** parameters which are only activated if a certain context is supplied. For example, to define a bulk refresh context where SQL expressions are performed on the source database (which would slow down capture) instead of the target database (which would slow down bulk refresh).
-d	Remove (drop) scripts and scheduler jobs & job groups generated by previous **hvrrefresh** command.
-f	Fire database triggers/rules while applying SQL changes for refresh. Normally for Oracle and SQL Server, HVR disables any triggers on the target tables before the refresh and re-enables them afterwards. On Ingres, the refresh avoids firing databases rules using statement **set no rules** . This option prevents this, so if refresh does an **insert** statement then it could fire a trigger. But note that HVR's refresh often uses a bulk-load method to load data, in which case database triggers will not be fired anyway. Other ways to control trigger firing are described in Managing Recapturing Using Session Names. For integration jobs into Ingres and SQL Server, action **Integrate** **/NoTriggerFiring** can also be used.
-F*k*	Behavior for foreign key constraint in the target database which either reference or are referenced by a table which should be refreshed. Value for *k* is one or more of these letters: • **i** : Ignore foreign key constraints. Normally this would cause foreign key constraint errors. This cannot be combined with other letters. • **x** : Disable all such constraints before refresh and re-enable them at the end. If the DBMS does not support disable/re-enable syntax (e.g. Ingres) then constraints are instead dropped before refresh and recreated at the end. Note that for on-line refresh (option **-q**) without a select moment supplied (option **-M**) the actual re-enabling of disabled foreign key constraints is not done by the refresh itself but is instead delayed until the end of next cycle of integration. • **d** : For a target with 'deferred' or 'deferrable' foreign key constraints, perform entire refresh in a single (very large) transaction which is committed right at the end of refresh. This can use a lot of DBMS resources. It is also slower because HVR uses SQL **delete** and **insert** statements, because both **truncate** and 'direct path load' both imply a commit. This is only supported for Oracle (because other DBMSs do not support deferring of foreign key constraints). If this letter is combined with letter **x** (disable) then only non-deferrable constraints are disabled and then re-enabled. Deferred constraint handling cannot be used with table parallelism (option **-P**) If this option (**-F**) is not supplied then all foreign-key constraints will be disabled before refresh and re-enabled afterwards (letter **x**), unless HVR has no capability for foreign keys at all (letter **i**).
-g*x*	Granularity of refresh in database locations. Valid values of *x* are: • **b** (**default**): Bulk refresh using bulk data load. • **r** : Row-wise refresh of tables.
-h*class*	Location *class* of the hub database. Valid values for *class* are **db2**, **db2i**, **ingres**, **mysql**, **oracle**, **postgresql**, **sqlserver**, or **teradata**. For more information, see Calling HVR on the Command Line.
-i*num_jobs* **Since** v5.3.1/25	Sets **quota_run** refresh job group attribute. It defines a number of jobs which can be run simultaneously. The option cannot be used without scheduling turned on (**-s**)
-l*x*	Target location of refresh. The other (read location) is specified with option **-r**. If this option is not supplied then all locations except the read location are targets. Values of *x* maybe one of the following: • *loc* : Only location *loc*. • *l1*-*l2* : All locations that fall alphabetically between *l1* and *l2* inclusive. • !*loc* : All locations except *loc*. • !*l1*-*l2* : All locations except for those that fall alphabetically between *l1* and *l2* inclusive. Several **-l***x* instructions can be supplied together.
-m*mask*	Mask (ignore) some differences between the tables that are being refreshed. Valid values of *mask* can be: • **d** : Mask out delete differences. • **i** : Mask out insert differences. • **u** : Mask out update differences. Letters can be combined, for example **-mid** means mask out inserts and deletes. If a difference is masked out, then the verbose option (**-v**) will not generate SQL for it and **hvrrefresh** will not rectify it. The **-m** option can only be used with row-wise granularity (option **-gr**).
-M*moment*	Select data from each table from same consistent *moment* in time. Value *moment* can be one of the following: • *time* : Flashback query with **select … as of timestamp**. Valid formats are *YYYY-MM-DD [HH:MM:SS]* (in local time) or *YYYY-MM-DDTHH:MM:SS+TZD* or *YYYY-MM-DDTHH:MM:SSZ* or **today** or now[[±]*SE CS*] or an integer (seconds since **1970-01-01 00:00:00 UTC**). Note that if a symbolic time like **-Mnow** is supplied then a new "SCN time" will be retrieved each time the refresh job is run (not only when the **hvr refresh** command is called. So if **hvrrefresh -Mnow** is run on Monday, and the refresh job it creates starts running at 10:00 Tuesday and runs again 10:00 on Wednesday, then the first refresh will do a flashback query (for all tables) with an SCN corresponding to Tuesday at 10:00 and the second job run will use flashback query with an SCN corresponding to Wednesday at 10:00. • **scn**=*val* : Flashback query with **select … as of scn**. Value is an Oracle SCN number, either in decimal or in hex (when it starts with **0x** or contains hex digits). • **hvr_tx_seq**=*val* : Value from HVR column **hvr_tx_seq** is converted back to an Oracle SCN number (by dividing by **65536**) and used for flashback query with **select … as of scn**. Value is either in decimal or in hex (when it starts with **0x** or contains hex digits). • **serializable** : Select all data from source database using a single transaction (therefore a single session) which has SQL isolation level **serializable**. This cannot be used with table parallelism (option **-P**). • **snapshot** : Select all data from source database using a single transaction (therefore a single session) which has SQL isolation level **snapshot** (SQL Server only). Using **snapshot** isolation level requires enabling **ALLOW_SNAPSHOT_ISOLATION** database option in SQL Server. This cannot be used with table parallelism (option **-P**). This parameter only affects the selects of the leftmost (source) database, not any selects on the rightmost (target) database.
-n*numtabs* **Since** v5.3.1/6	Create 'sub-jobs' which each refresh a bundle of no more than *numtabs* tables. In HVR GUI, this option is displayed as **Limit Tables per Job** in the **Scheduling** tab. For example, if a channel contains 6 tables then option **-n1** will create 6 jobs whereas were option **-n4** to be used on the same channel then only 2 jobs will be created (the first with 4 tables, the last with just 2). If tables are excluded (using option **-t**) then these will not count for the bundling. Jobs are named by adding a number (starting at 0) to the task name which defaults **refr** (although the task name can always be overridden using option **-T**). Normally the first slice's job is named **chn-refr0-x-y** but numbers are left-padded with zeros, so if 10 slices are needed the first is named **chn-refr00-x-y** instead. One technique is to generate lots of jobs for refresh of big channel (using this option and option **-s**) and add 'scheduler attribute' **quota_run** to the job group (named *CHN*-**REFR**) so that only a few (say 3) can run simultaneously. Scheduler attributes can be added by right-clicking on the job group and selecting **Add Attribute**. Another technique to manage the refresh a channel with thousands of tables is use this option along with options **-R** (ranges) and **-T** (task name) to do 'power of ten' naming and bundling, in case a single table encounters a problem. The following illustrates this technique; First use **[-n100]** so each job tries to refresh 100 tables. If one of these jobs fails (say job **chn-refr03-x-y**) then use options **[-n10 -R30-39 -Trefr03]** to replace it with 10 jobs which each do 10 tables. Finally if one of those jobs fail (say **chn-refr037-x-y**) then use options **[-n1 -R370-379 -Trefr037]** to replace it with 10 'single table' jobs.

-O
Since v5.3.1/6
	Only show OS command implied by options **-n** (jobs for bundles of tables) or **-S** (table slices), instead of executing them. This can be used to generate a shell script of 'simpler' **hvrrefresh** commands;
	
	For example, if a channel only contains tables **tab1**, **tab2**, **tab3** and **tab4** then this command;
	
	`hvrrefresh -rcen -O -n2 myhub/pwd mychn`
	
	will only generate this output;
	
	`hvrrefresh -rcen -Trefr0 -ttab1 -ttab2 myhub/pwd mychn`
	`hvrrefresh -rcen -Trefr1 -ttab3 -ttab4 myhub/pwd mychn`

-p*N*	Perform refresh on different locations in parallel using *N* sub-processes. This cannot be used with option **-s**.
-P*M*	Perform refresh for different tables in parallel using *M* sub-processes. The refresh will start by processing *M* tables in parallel; when the first of these is finished the next table will be processed, and so on.
-q*d*	Online refresh of data from a database that is continuously being changed. This requires that capture is enabled on the source database. The integration jobs are automatically suspended while the online refresh is running, and restarted afterwards. The target database is not yet consistent after the online refresh has finished. Instead, it leaves instructions so that when the replication jobs are restarted, they skip all changes that occurred before the refresh and perform special handling for changes that occurred during the refresh. This means that after the next replication cycle consistency is restored in the target database. If the target database had foreign key constraints, then these will also be restored. Valid values for *d* are: • **wo** : Write only. Changes before the online refresh should only be skipped on the write side (by the integrate job), not on the read side (by the capture job). If changes are being replicated from the read location to multiple targets, then this value will avoid skipping changes that are still needed by the other targets. • **rw** : Read/Write. Changes before the online refresh should be skipped both on the read side (by the capture job) and on the write side (by the integrate job). There are two advantages to skipping changes on the capture side; performance (those changes will not be send over the network) and avoiding some replication errors (i.e. those caused by an alter table statement). The disadvantage of skipping changes on the capture side is that these changes may be needed by other replication targets. If they were needed, then these other integration locations need a new 'online' refresh, but without **-qrw**, otherwise the original targets will need yet another refresh. • **no** : No skipping. Changes that occurred before the refresh are not skipped, only special handling is activated for changes that occurred during the refresh. This is useful for online refresh of a context-sensitive restriction of data (**-C***context* and **Restrict** **/RefreshCondition /Context**). Internally online refresh uses 'control files' to send instructions to the other replication jobs (see command **hvrcontrol**). These files can be viewed using command **hvrrouterview** with option **-s**. Online refresh (with option **-q**) can give errors if duplicate rows (**/DuplicateRows**) are actually changed during the online refresh.
-Q	No refresh of database sequences matched by action **DbSequence**. If this option is not specified, then the database sequence in the source database will be refreshed with matching sequences in the target database. Sequences that only exist in the target database are ignored.
-r*loc*	Read location. This means that data will be read from location *loc* and written to the other location(s).

-R*rangeexpr*

Since v5.3.1/6

Only perform certain 'sub jobs' implied by either options **-N** (job for bundles of tables) or **-S** (table slices). This option cannot be used without one of those options.

Value *rangeexpr* should be a comma-separated list of one of the following:

- *N* : Only perform 'sub job' number *N*. Note that these jobs are numbered starting from zero (e.g the first is *chn*-**refr0**-*rloc*-*wloc*).
- *N*-*M* : Perform from jobs from *N* to *M* inclusive.
- *N*- : Perform from jobs from *N* onwards.
- -*M* : Perform from jobs from the first job until job *M*.

For example, if a channel contains 20 tables then option **-n1** would cause 20 jobs to be created (with names **chn-refr00-x-y**, **chn-refr01-x-y**, **chn-refr02-x-y… chn-refr19-x-y**) but options **-n1 -R0,10** would restrict job creation to only 11 jobs (named **chn-refr00-x-y**, then **chn-refr10-x-y**, **chn-refr11-x-y … chn-refr19-x-y**).

-s

Schedule invocation of refresh scripts using the **HVR Scheduler**. In HVR GUI, this option is displayed as **Schedule Classic Job** in the **Scheduling** tab.

Without this option the **default** behavior is to perform the refresh immediately (in HVR GUI, **Run Interactively**).

This option creates refresh job for performing the refresh of tables in a channel from a source location to target location. By default, this refresh job is created in **SUSPEND** state and are named *chn*-**refr**-*source*-*target*. This refresh job can be invoked using command **Hvrstart** as in the following example:

```
hvrstart -u -w hubdb chn-refr
```

Executing the above command unsuspends (moves to **PENDING** state) the jobs and instructs the scheduler to run them. Output from the jobs is copied to the **hvrstart** command's **stdout** and the command finishes when all jobs have finished. Jobs created are cyclic which means that after they have run they go back to **PENDING** state again. They are not generated by a **trig_delay** attribute which means that once they complete they will stay in **PENDING** state without getting retriggered.

Once a refresh job has been created with option **-s** then it can only be run manually on the command line (without using **HVR Scheduler**) as follows:

```
hvrstart -i hubdb chn-refr-source-target
```

-S*sliceexpr*

Since v5.3.1/6

Refresh large tables using slicing. Value *sliceexpr* can be used to split table into multiple slices. In HVR GUI, this option is displayed as **Slice Table** in the **Scheduling** tab.

A refresh job is created per slice for refreshing only rows contained in the slice. These refresh jobs can be run in parallel to improve the overall speed of the refresh. Slicing can only be used for a single table (defined with option **-t**).

The value *sliceexpr* affects action **Restrict /RefreshCondition**. That action must be defined on the table (at least on the read location) and must contain a relevant **{hvr_var_slice_*}** substitution.

As with option **-n** (bundles of tables), jobs are named by adding a number (starting at 0) to the task name which defaults **refr** (although this task name can always be overridden using option **-T**). Normally the first slice's job is named *chn*-**refr0**-*source*-*target* but numbers are left-padded with zeros, so if 10 slices are needed the first is named *chn*-**refr00**-*source*-*target* instead.

Note that if an on-line refresh is done (option **-q**) and no **Select Moment** is specified (option **-M**) then only value **no** (resilience) is allowed, not **rw** (skip during capture and integrate) or **wo** (skip during integrate only).

The column used to slice a table must be 'stable', because if it is updated then a row could 'move' from one slice to another while the refresh is running. The row could be refreshed in two slices (which will cause errors) or no slices (data-loss). If the source database is Oracle then this problem can be avoided using a common 'select moment' (option **-M**).

Running bulk refresh (option **-gb**) for multiple slices in parallel is not supported for relational database targets. Run them one at the time instead and use **Restrict /RefreshCondition** with a filter such as **{hvr_var_slice_condition}** to protect rows on the target that will not be refreshed.

Value *sliceexpr* must have one of the following forms:

col%*num*

Slicing using modulo of numbers. In HVR GUI, this option is displayed as **Modulo**.

Since HVR 5.6.5/0, it is not required to define **Restrict /RefreshCondition** to use this type of slicing. Prior to HVR 5.6.5/0, this slicing form affects the substitution **{hvr_var_slice_condition}** which must be mentioned in **Restrict /RefreshCondition** defined for the slice table.

> It is recommend that any **Restrict /RefreshCondition** defined for slicing is also given a **/Context** parameter so it can be easily disabled or enabled.

If **-Sabc%3** is supplied then the conditions for the three slices are:

```
mod(round(abs(coalesce(abc, 0)), 0), 3)= 0
mod(round(abs(coalesce(abc, 0)), 0), 3)= 1
mod(round(abs(coalesce(abc, 0)), 0), 3)= 2
```

Note that the use of extra SQL functions (e.g. **round()**, **abs()** and **coalesce()**) ensure that slicing affect fractions, negative numbers and NULL too. Modulo slicing can only be used on a column with a numeric data type.

col<b1[<b2]... [<bN]]

Slicing using boundaries. In HVR GUI, this option is displayed as **Boundaries**.
If *N* boundaries are defined then *N+1* slices are implied.

Since HVR 5.6.5/0, it is not required to define **Restrict /RefreshCondition** to use this form of slicing. Prior to HVR 5.6.5/0, this slicing affects the substitution **{hvr_var_slice_condition}** which must be mentioned in **Restrict /RefreshCondition** defined for the slice table.

> It is recommend that any **Restrict /RefreshCondition** defined for slicing is also given a **/Context** parameter so it can be easily disabled or enabled.

If **-Sabc<10<20<30** is supplied then the conditions for the four slices are:

```
abc <= 10
abc > 10 and abc <= 20
abc > 20 and abc <= 30
abc > 30 or abc is null
```

Note that strings can be supplied by adding quotes around boundaries, i.e. **-Sabc<'x'<'y'<'z'**.

For very large tables consider the DBMS query execution plan. If the DBMS decides to 'walk' an index (with a lookup for each matched row) but this is not optimal (i.e. a 'serial-scan' of the table would be faster) then either use *DBMS techniques* (**$HVR_SQL_SELECT_HINT** allows Oracle optimizer hints) or consider modulo slicing (*col%num*) instead.

For this type of slicing, HVR can suggest boundaries by using the Oracle's **dbms_stats** package. Click the browse ("...") button for **Boundaries** type of slicing and then click **Suggest Values** in **Boundaries for Table** dialog. Number of slices can be also specified.

Gathering column histogram statistics is required for this functionality to work. This can be done by calling the **dbms_stats.gather_table_stats** stored procedure

Examples:

1. Gathers statistics including column histograms, for table 'table_name', using all table rows, for all columns, and maximum of 254 histogram buckets (therefore up to 254 slice boundaries can be suggested).

```
exec dbms_stats.gather_table_stats('schema_name',
'table_name', estimate_percent=>100, method_opt=>'for
all columns size 254);
```

2. Gathers statistics including column histograms, for table 'table_name', using all table rows, for all indexed columns, and default number of histogram buckets.

```
exec dbms_stats.gather_table_stats('schema_name',
'table_name', estimate_percent=>100, method_opt=>'for
all indexed columns);
```

3. Gathers statistics including column histograms, for table 'table_name', using 70% of table rows, for column 'table_column', and maximum of 150 histogram buckets (therefore up to 150 slice boundaries can be suggested).

```
exec dbms_stats.gather_table_stats('schema_name',
'table_name', estimate_percent=>70, method_opt=>'for
columns table_column size 150);
```

4. Gathers statistics including column histograms, for table 'table_name', for all columns, and maximum 254 histogram buckets. This is an obsolete way to generate statistics and there are much less options supported.

```
analyze table table_name compute statistics for all
columns size 254;
```

num

Numbered slices. In HVR GUI, this option is displayed as **Count**.

Since HVR 5.6.5/0, the number of each slice is assigned to substitution **{hvr_slice_num}** which must be mentioned in **Restrict /SliceCondition** defined for the slice table. Substitution **{hvr_slice_total}** is also assigned to the total number of slices. However, prior to HVR 5.6.5/0, the substitution **{hvr_var_slice_num}** must be mentioned in **Restrict /RefreshCondition** defined for the slice table. Substitution **{hvr_var_slice_total}** is also assigned to the total number of slices.

> It is recommend that any **Restrict /RefreshCondition** defined for slicing is also given a **/Context** parameter so it can be easily disabled or enabled.

Example:

In heterogeneous environments doing normal modulo slicing is not always possible because of different syntax of modulo operation. For example, in Oracle modulo operation is **mod(x,y)** and in Teradata it is **x mod y**. Also negative numbers are handled differently on these two databases. For this scenario, two **Restrict** actions can be defined, one for the capture location (Oracle) and other for the integrate location (Teradata):

Location	Action
ora	**Restrict /SliceCondition="abs(mod({hvr_var_slice_col}, {hvr_slice_total})) = {hvr_slice_num}" /Context=slice**
terad	**Restrict /SliceCondition="abs({hvr_var_slice_col} mod {hvr_slice_total}) = {hvr_slice_num}" /Context=slice**

Prior to HVR 5.6.5/0, using **Restrict /RefreshCondition**:

Location	Action
ora	**Restrict /RefreshCondition="abs(mod({hvr_var_slice_col}, {hvr_var_slice_total})) = {hvr_var_slice_num}" /Context=slice**
terad	**Restrict /RefreshCondition="abs({hvr_var_slice_col} mod {hvr_var_slice_total}) = {hvr_var_slice_num}" /Context=slice**

If options **-S3 -Vslice_col=abc** are supplied then the conditions for the three slices are:

Capture Location	Integrate Location
`abs(mod(abc, 3))= 0 abs(mod (abc, 3))= 1 abs(mod(abc, 3))= 2`	`abs(abc mod 3)= 0 abs (abc mod 3)= 1 abs(abc mod 3)= 2`

	val1[,*val2*]...	Slicing using a list of values. In HVR GUI, this option is displayed as **Series**. Values are separated by semicolons. Since HVR 5.6.5/0, each slice has its value assigned directly into substitution **{hvr_slice_value}** must be mentioned in **Restrict /SliceCondition** defined for the sliced table. However, prior to HVR 5.6.5/0, the substitution **{hvr_var_slice_value}** must be mentioned in **Restrict /RefreshCondition** defined for the slice table. It is recommend that any **Restrict /RefreshCondition** defined for slicing is also given a **/Context** parameter so it can be easily disabled or enabled. **Example:** A large table with column **country** having values **US**, **UK**, and **NL** can be sliced in the following manner: • If option **-S "'US','UK'"** is supplied with action **Restrict /RefreshCondition="country ={hvr_var_slice_value}" /Context=slice** then **HVR Refresh** creates 2 slices (2 refresh jobs) - 1 for **US** and 1 for **UK**. • If option **-S "'US','UK'"** is supplied with action **Restrict /RefreshCondition="country IN ({hvr_var_slice_value})" /Context=slice** then **HVR Refresh** creates 2 slices (2 refresh jobs) - 1 for **US** and 1 for **UK**. • If option **-S "{'US','UK'};('NL')"** is supplied with action **Restrict /RefreshCondition="country IN ({hvr_var_slice_value})" /Context=slice** then **HVR Refresh** creates 2 slices (2 refresh jobs) - 1 for **US**, **UK** and 1 for **NL**. The double quotes (") supplied with option **-S** is not required in HVR GUI.
-t*y*	Only refresh objects referring to table codes specified by *y*. Values of *y* may be one of the following: • *tbl* : Only refresh table name *tbl*. • *t1*-*t2* : Refresh all table codes that fall alphabetically between *t1* and *t2* inclusive. • !*tbl* : Refresh all table names except *tbl*. • !*t1*-*t2* : Refresh all table codes except for those that fall alphabetically between *t1* and *t2* inclusive. Several **-t***y* instructions can be supplied together.
-T*tsk*	Specify alternative task for naming scripts and jobs. The **default** task name is **refr**, so for example without this **-T** option the generated jobs and scripts are named *chn*-**refr**-*l1*-*l2*.
-u*user*[/*pwd*]	Connect to hub database using DBMS account *user*. For some databases (e.g. SQL Server) a password *pwd* must also be supplied.
-v	Verbose. This causes row-wise refresh to display each difference detected. Differences are presented as SQL statements. This option requires that option **-gr** (row-wise granularity) is supplied.
-V*name*=*value*	Supply variable into refresh restrict condition. This should be supplied if a **/RefreshCondition** parameter contains string **{hvr_var_***name***}**. This string is replaced with *value*.

The effects of **hvrrefresh** can be customized by defining different actions in the channel. Possible actions include **Integrate /DbProc** (so that row-wise refresh calls database procedures to make its changes) and **Restrict /RefreshCondition** (so that only certain rows of the table are refreshed). Parameter **/Context** can be used with option **-C** to allow restrictions to be enabled dynamically. Another form of customization is to employ SQL views; HVR Refresh can read data from a view in the source database and row-wise refresh can also select from a view in the target database, rather than a real table when comparing the incoming changes.

If row-wise **hvrrefresh** is connecting between different DBMS types, then an ambiguity can occur because of certain data type coercions. For example, HVR's coercion maps an empty string from other DBMS's into a **null** in an Oracle **varchar**. If Ingres location **ing** contains an empty string and Oracle location **ora** contains a null, then should HVR report that these tables are the same or different? Command **hvrrefresh** allow both behavior by applying the sensitivity of the 'write' location, not the 'read' location specified by **-r**. This means that row-wise refreshing from location **ing** to location **ora** will report the tables were identical, but row-wise refreshing from **ora** to **ing** will say the tables were different.

Slicing Limitations

This section lists the limitations of slicing when using **hvrrefresh**.

Modulo

Following are the limitations when using slicing with modulo of numbers (*col%num*):

1. It only works on numeric data types. It may work with binary **float** values depending on DBMS data type handling (e.g. works on MySQL but not on PostgreSQL).
2. A diverse **Modulo** syntax on source and target (e.g. "**where col%5=2**" and "**where mod(col,5)=2**") may produce inaccurate results. This limitation applies only to classic refresh. Since HVR 5.5.5/6, the event-driven refresh can handle it.
3. Heterogeneous refresh (between different DBMSes or file locations) has a limitation with **Modulo** slicing on the Oracle's **NUMBER(*)** column: if a value has an exponent larger than 37 (e.g. if a number is larger than 1E+37 or smaller than -1E+37), then this row might be associated with a wrong slice. This column should not be used for **Modulo** slicing. (The exact limits of the values depend on the number of slices).
4. For some supported DBMSes (SQL Server, PostgreSQL, Greenplum, Redshift), **Modulo** slicing on a **float** column is not allowed (may result in SQL query error).

5. For some DBMSes, **float** values above the limits of DBMS's underlying precision ("big **float** values") may produce inaccurate results during **Modulo** slicing. This affects only heterogeneous environments.
6. Refresh with **Modulo** slicing on a column with "big **float** values" may produce inaccurate results in HANA even in a homogeneous environment (HANA-to-HANA).

A workaround for the above limitations is to use **Boundaries** slicing or **Count** slicing with custom SQL expressions.

Boundaries

Boundaries slicing of dates does not work in heterogeneous DBMSes.

> For databases that require staging during bulk refresh, like Redshift, Snowflake, Greenplum, etc. in order to manage a large refresh operation, multiple slicing jobs should be scheduled one after another to avoid a risk of corruption on a tartget location.

Examples

For bulk refresh of table **order** from location **cen** to location **decen**:

```
hvrrefresh -rcen -ldecen -torder hubdb/pwd sales
```

To only send updates and insert to a target database without applying any deletes use the following command:

```
hvrrefresh -rcen -md -gr hubdb/pwd sales
```

Files

HVR_CONFIG	
job	Directory containing all scripts generated by **hvrrefresh**.
hubdb	
chn	
chn-**refr**-*loc1*-*loc2*	Script to refresh *loc1* with *loc2*.

See Also

Commands Hvrcompare, Hvrgui and Hvrcrypt.

Hvrremotelistener

Contents

Name

hvrremotelistener - HVR Remote Listener.

Synopsis

hvrremotelistener [**-***options*] *portnum* [*access_conf.xml*]

Description

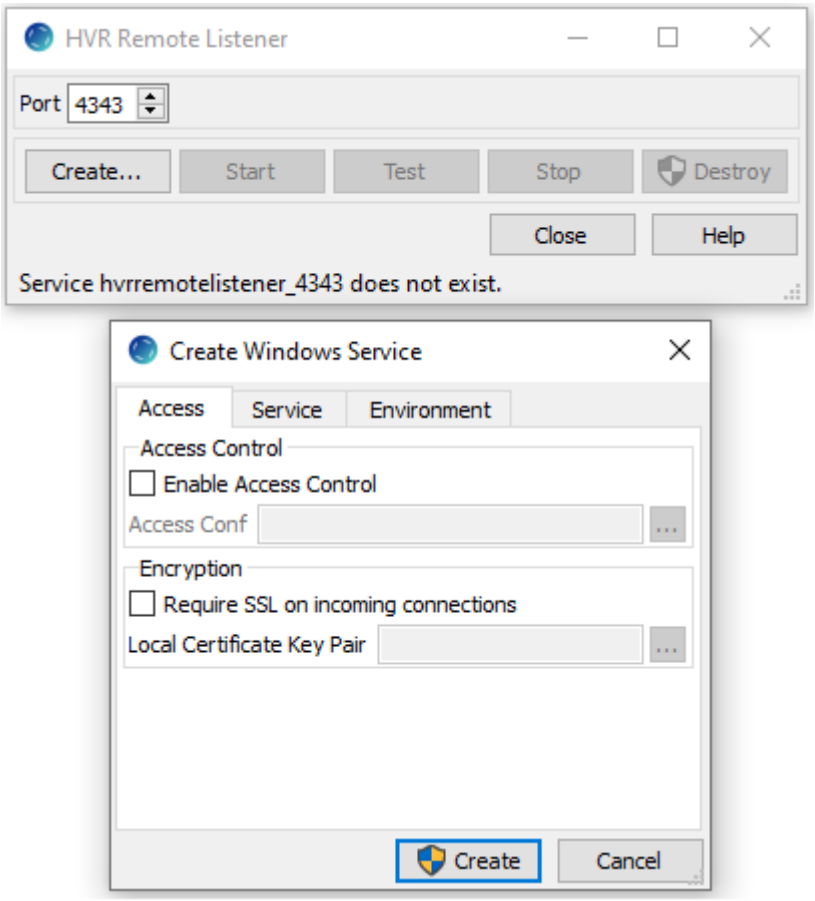
HVR Remote Listener listens on a TCP/IP port number and invokes an **hvr** process for each connection. The mechanism is the same as that of the Unix/Linux daemon **inetd**, **xinetd** or **systemd**.

On Windows, HVR Remote Listener is a Windows Service which is administered with option **-a**. The account under which it is installed must be member of the Administrator group, and must be granted privilege to act as part of the operating system (**SeTcbPrivilege**). The service can either run as the default system account, or (if option **-P** is used) can run under the HVR account which created the Windows Service.

On Unix and Linux, HVR Remote Listener runs as a daemon which can be started with option **-d** and killed with option **-k**.

Optionally, after the port number *portnum* an access configuration file *access_conf.xml* can be specified. This can be used to authenticate the identity of incoming connections using SSL. For example, the following contents will restrict access to only connections from a certain hub machine:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hvraccess SYSTEM "hvraccess.dtd">
<hvraccess>
<allow>
<from> <host>myhub</host> <ssl remote_cert="hub"/> </from>
</allow>
</hvraccess>
```



⚠

- HVR Remote Listener is supported on Unix and Linux but it is more common on these machines to start remote HVR executables using the system process (**inetd**, **xinetd** or **systemd**). For more information, see Configuring Remote Installation of HVR on Unix or Linux.
- When HVR Remote Listener is executed as a Windows service the errors are written to the Windows event log (**Control Panel Adminitrative Tools Event Viewer Windows Logs Application**).

Options

This section describes the options available for command **hvrremotelistener**.

Parameter	Description
-a*x* **Windows**	Administration operations for Microsoft Windows system service.Values of *x* can be: • **c** : Create the HVR Remote Listener system service. • **s** : Start the HVR Remote Listener system service. • **h** : Halt (stop) the system service. • **d** : Destroy the system service. Several **-a***x* operations can be supplied together; allowed combinations are e.g. **-acs** (c reate and start) or **-ahd** (halt and destroy). HVR Remote Listener system service can be started (**-as**) and halted (**-ah**) from Windows Services (**Control Panel Administrative Tools Computer Management Services and Applications Services**).

-A **Unix & Linux**	Remote HVR connections should only authenticate login/password supplied from hub, but should not change from the current operating system username to that login. This option can be combined with the **-p** option (PAM) if the PAM service recognizes login names which are not known to the operating system. In that case the daemon service should be configured to start the HVR child process as the correct operating system user (instead of **root**).
-c*clus**clusgrp* **Windows**	Enroll the Remote Listener Service in a Windows cluster named *clus* in the cluster group *clusgrp*. Once the service is enrolled in the cluster it should only be stopped and started with the Windows cluster dialogs instead of the service being stopped and started directly (in the Windows Services dialog or with options **-as** or **-ah**). In Windows, failover clusters *clusgrp* is the network name of the item under **Services and Applications**. The group chosen should also contain the remote location; either the DBMS service for the remote database or the shared storage for a file location's top directory and state directory. The service needs to be created (with option **-ac**) on each node in the cluster. This service will act as a 'Generic Service' resource within the cluster. This option must be used with option **-a**.
-d **Unix & Linux**	Start **hvrremotelistener** as a daemon process.
-E*name=value*	Set environment variable *name* to value *value* for the HVR processes started by this service.
-i	Interactive invocation. HVR Remote Listener stays attached to the terminal instead of redirecting its output to a log file.
-k **Unix & Linux**	Stop **hvrremotelistener** daemon using the process-id available in **$HVR_CONFIG /files/hvrremotelistener***port***.pid**.
-K*pair*	SSL encryption using two files (public certificate and private key) to match public certificate supplied by **/SslRemoteCertificate**. If *pair* is relative, then it is found in directory **$HVR_HOME/lib/cert**. Value *pair* specifies two files; the names of these files are calculated by removing any extension from *pair* and then adding extensions **. pub_cert** and **.priv_key**. For example, option **-Khvr** refers to files **$HVR_HOME/lib /cert/hvr.pub_cert** and **$HVR_HOME/lib/cert/hvr.priv_key.**
-N	Do not authenticate passwords or change the current user name. Disabling password authentication is a security hole, but may be useful as a temporary measure. For example, if a configuration problem is causing an 'incorrect password' error, then this option will bypass that check.
-p*pamsrv*** **Unix & Linux**	Use Pluggable Authentication Module *pamsrv* for login password authentication of remote HVR connections. PAM is a service provided by several Operation Systems as an alternative to regular login/password authentication, e.g. checking the **/etc /passwd** file. Often **-plogin** will configure HVR child processes to check passwords in the same way as the operating system. Available PAM services can be found in file **/et c/pam.conf** or directory **/etc/pam.d**.
-P*pwd* **Windows**	Configure HVR Remote Listener service to run under the current login HVR account using password *pwd*, instead of under the default system login account. May only be supplied with option **-ac**. Empty passwords are not allowed. The password is kept (hidden) within the Microsoft Windows operating system and must be re-entered if passwords change.
-U*user*	Limits the HVR child process to only accept connections which are able to supply the password for account *user*. Multiple **-U** options can be supplied.

Examples

Windows

To create and start a Windows listener service to listen on port number 4343:

```
c:\> hvrremotelistener -acs 4343
```

Unix & Linux

To run **hvrremotelistener** interactively so that it listens on a Unix machine, use the following command. Note that option **-N** is used to disable password authentication; this is necessary when running as an unprivileged user because only **root** has permission to check passwords.

```
$ hvrremotelistener -i -N 4343
```

Files

HVR_HOME	
bin	
hvr	Executable for remote HVR service.
hvrremotelistener	HVR Remote Listener executable.
lib	
hvrpasswd	Password file employed by **hvrvalidpwfile**.
hvrvalidpw	Used by HVR for user authentication.
hvrvalidpwfile	The plugin file for private password file authentication.
hvrvalidpwldap	The plugin file for LDAP authentication.
hvrvalidpwldap.conf	Configuration for LDAP authentication plugin.
hvrvalidpwldap.conf_example	Example configuration file for LDAP authentication plugin.
HVR_CONFIG	
files	
hvrremotelistener*port_node***.pid**	Process-id of daemon started with option **-d**.
log	
hvrremotelistener	
hvrremotelistener*port***.log**	Logfile for daemon started with **-d**.

See Also

- Command Hvr
- Configuring Remote Installation of HVR on Unix or Linux
- Configuring Remote Installation of HVR on Windows

Hvrretryfailed

Contents

Name

hvrretryfailed - Retry changes saved in fail tables or directory due to integration errors.

Synopsis

hvrretryfailed [**-d**] [**-h***class*] [**-t***tbl*]... [**-u***user*] [**-w***sqlrestr*] [**-v**] *hubdb chn loc*

Description

Command **hvrretryfailed** causes HVR to reattempt integration of changes which gave an error during integration into location *loc*. For integration into a database these changes were written to fail tables in the target database. For file integration these are unsuccessful files which are moved into the file location's state directory. HVR integration jobs save changes in the fail tables or directory if action **/OnErrorSaveFailed** or **/OnErrorBlockLoc** is defined. The integration is retried immediately, instead of being delayed until the next integrate job runs.

The first argument *hubdb* specifies the connection to the hub database. This can be an Oracle, Ingres, SQL Server, DB2, DB2 for I, PostgreSQL or Teradata database depending on its form. See further section Calling HVR on the Command Line.

Options

This section describes the options available for command **hvrretryfailed**.

Parameter	Description
-d	Only delete rows, do not retry them. If no **-w** option is supplied then the fail table is also dropped. This is only allowed for database locations.
-f*freq*	Commit frequency. By **default** a commit is done after every 100 deletes. This is only allowed for database locations.
-h*class*	Specify hub *class*, for connecting to hub database. For supported values, see Calling HVR on the Command Line.

-t*tbl*	Only failed rows from table *tbl*. If this option is not supplied, rows from all fail tables will be processed. Value *tbl* may be one of the following: • *tbl* : Only affects table *tbl*. • *t1*-*t2* : Affects all tables that fall alphabetically between *t1* and *t2* inclusive. • **!***tbl* : Affects all tables except *tbl*. • **!***t1*-*t2* : Affects all tables except for those that fall alphabetically between *t1* and *t2* inclusive. Several **-t***y* instructions can be supplied together. This is only allowed for database locations.
-u*user*[*/pwd*]	Connect to hub database using DBMS account *user*. For some databases (e.g. SQL Server) a password must also be supplied.
-v	Verbose output.
-w*sqlrestrict*	Where clause. Only failed rows where *sqlrestr* is true will be processed. For example to only retry recent changes for a certain column, the SQL restriction would be **-w "hvr_cap_tstamp >= '25/5/2007' and col1=22".**This is only allowed for database locations.

Hvrrouterconsolidate

Contents

Name

Hvrrouterconsolidate - Merge small tx files in router directory.

Synopsis

hvrrouterconsolidate [-*options*]... *hub* [*chn*]

Description

If capture jobs run without integrate jobs then many transaction files (matching ***.tx**_*integloc*) will accumulate in **$HVR_CONFIG/router/***hub***/***chn*. These files can cause extra CPU load in other jobs. Small transaction files will also accumulate in **$HVR_CONFIG/jnl/***hub***/***chn* if action **Integrate /JournalRouterFiles** is defined. Command **hvrrouterconsolidate** will merge many of these smaller files together.

Consolidation is unsafe while an integrate job is running; for this reason **hvrrouterconsolidate** will only consolidate router files for integrate jobs which have scheduler **job_state** as **SUSPEND** or **FAILED** or **PENDING** (unless option **-f** supplied).

If the channel name is not supplied then router consolidation will be done for all channels.

Note that consolidation will not merge all small files; the algorithm instead processes files in batches of 10, 100, 1000 etc...

Options

This section describes the options available for command **hvrrouterconsolidate**.

Parameter	Description
-b*base*	Base/radix. Otherwise 10 files are grouped.
-f	Consolidate files for all jobs, not just ones where job_state is inactive. Only used when the scheduler is not running.
-j	Journals only, ignore /router directory.
-h*class*	Class for hub database.
-l*locscope*	Specific locations only. Value can have form *loc* or **!***loc*
-N*nrows*	Max num rows. Larger files won't be joined.
-p	Only print OS commands, no execution.
-r	Router files only, ignore **/jnl** directory.

-s*fsize*	No consolidate if new file would exceed *fsize*. **Default** is 20Mb.
-u*dbuser*	Database user name.

Hvrrouterview

Contents

Name

hvrrouterview - View or extract contents from internal router files.

Synopsis

hvrrouterview [*-restrict opts*] [*-xml opts*] [**-F**] *hubdb chn* [*txfile*]…

hvrrouterview -xtgt [*-restrict opts*] [*-extract opts*] [**-F**] *hubdb chn* [*txfile*]…

hvrrouterview -s [**-c***loc*] [**-i***loc*] [*-xml opts*] *hubdb chn*

hvrrouterview [*-xml opts*] *hubdb chn hvrfile…*

Description

This command can be used to view or extract data from internal HVR files such as transaction and journal files in the router directory on the hub machine.

The first form (in the Synopsis above) shows the contents of any transaction files currently in the channel's router directory. Options **-b**, **-c**, **-e**, **-f**, **-i**, **-n**, **-t** and **-w** can be used to restrict the changes shown. Option **-j** shows journal files instead of transaction files. The output is shown as XML, which is sent to **stdout**.

The second form (with option **-x**) extracts the data from the transaction files into a target, which should be either a database (for database changes) or a directory (for a blob file channel).

The third form (with option **-s**) shows the contents of control files as XML.

The fourth form can be used to view the contents of many internal HVR files, such as a ***.enroll** or ***.cap_state** file in a **router** directory, any of the files in a file location's **_hvr_state** directory, a control file (in directory **$HVR_CONFIG/router/***hub***/***chn***/control**) or the GUI preferences file (**$HVR_CONFIG/files/hvrgui.ini**).

The first argument *hubdb* specifies the connection to the hub database. For more information about supported hub databases, see Calling HVR on the Command Line.

General Options

Parameter	Description
-F	Identifies transaction file as captured by a 'blob file' channel (these use a different decompression algorithm). This is seldom necessary, because HVR should deduce the decompression algorithm from the basename of the transaction file.

-s	View contents of control files instead of transaction files. Control files are created by **Hvrrefresh** with option **-q**, or by command **Hvrcontrol**.
-u*user*[*/pwd*]	Connect to hub database using DBMS account *user*. For some databases (e.g. SQL Server) a password *pwd* must also be supplied.

Restrict Options

Parameter	Description
-b*time*	Only changes after capture time. Valid formats are *YYYY-MM-DD [HH:MM:SS]* in local time or *YYYY-MM-DDTHH:MM:SS+TZD* or *YYYY-MM-DDTHH:MM:SSZ* or **today** or **now**[[±]SECS] or an integer (seconds since **1970-01-01 00:00:00 UTC**). For example, **-b "2010-07-29 10:30:21"** or **-b now-3600** (changes in the last hour). This option is equivalent to **-whvr_cap_tstamp>=***time*.
-c*loc*	Only changes from specific capture location.
-e*time*	Only changes before capture time. Valid formats are *YYYY-MM-DD [HH:MM:SS]* in local time or *YYYY-MM-DDTHH:MM:SS+TZD* or *YYYY-MM-DDTHH:MM:SSZ* or **today** or **now**[[±]SECS] or an integer (seconds since **1970-01-01 00:00:00 UTC**). For example, **-b "2010-07-29 10:30:21"** or **-b now-3600** (changes in the last hour). This option is equivalent to **-whvr_cap_tstamp>=***time*.
-f*txfile*	Contents of a specific transaction file. This option can be specified multiple times. Another way to see the contents of a specific transaction file is to list the file(s) after the channel name (as a third positional parameter). The advantage is that 'globbing' can be used for a list of files (e.g. ***.tx_l***) whereas **-f** only accepts one file (although it can be supplied multiple times).
-i*loc*	Only changes for a specific integrate location.
-j	Show the contents of HVR journal files in directory **$HVRCONFIG/jnl/***hubdbl**chn*. These files are kept if parameter **/Journal** is defined for **Integrate**.
-n	Newest. Only most recent transaction file(s).
-t*y*	Only rows for tables specified by *y*. Values of *y* may be one of the following: • *tbl* : Only table *tbl*. • *t1***-***t2* : All tables that fall alphabetically between *t1* and *t2* inclusive. • **!***tbl* : All tables except *tbl*. • **!***t1***-***t2* : All tables except for those that fall alphabetically between *t1* and *t2* inclusive.Several **-t***y* instructions can be supplied together.
-w*where*	*Where* condition which must have form *columname operator value*. The *operator* can be either **= != <> > < >=** or **<=**. The *value* can be a number, '*str*', **X**'*hex* ', or a date. Valid date formats are *YYYY-MM-DD [HH:MM:SS]* in local time or *YYYY-MM-DDTHH:MM:SS+TZD* or *YYYY-MM-DDTHH:MM:SSZ* or **today** or **now** [[±]*SECS*] or an integer (seconds since **1970-01-01 00:00:00 UTC**). For some operators (**= != <>**) the value can be a list seperated by '**|**'. If multiple **-w** options are supplied then they are AND-ed together.

XML Options

Parameter	Description
-d	Show column data type information.
-h	Print data values in hexadecimal format.

Extract Options

Parameter	Description
-x*tgt*	Extract data from transaction files into a target. Value *tgt* should be either an actual database name (e.g. **myuser/pwd**) or a directory (e.g. **/tmp**) and not a HVR location name. By default, this target should be on the hub machine, unless option **-R** is specified, in which case **hvrrouterview** will extract the data to a target on a different machine.
-C*pubcert*	SSL public certificate of remote location. This must be used with options **-x** and **-R**.
-E*n=v*	Set environment variable *n* to value *v* for the HVR processes started on the remote node.
-K*pair*	SSL public certificate and private key pair of hub location.
-L*login/pwd*	Login and password for remote node. This must be used with options **-x** and **-R**.
-R*node:port*	Remote node name and HVR port number so data is extracted to remote target. This must be used with option **-x**.

Examples

To show the contents of certain transaction files:

```
cd $HVR_CONFIG/router/hubdb/hvr_demo01/loc_cen
```

```
hvrrouterview hubdb chn *.tx_dec01
```

To show the contents of file **4a82a8e8_c31e6.cap_state**:

```
hvrrouterview hubdb chn 4a82a8e8_c31e6.cap_state
```

To show any changes for table **cust** from the journals:

```
hvrrouterview -j -tcust hubdb chn
```

To retrieve all files moved by a blob file channel in the last hour use the following command. The data is read from the channel's journals and the extracted files are written into **/tmp**.

```
hvrrouterview -F -j -bnow-3600 -x/tmp hubdb chn
```

Files

router	Directory containing replication state.

hub	
chn	
catalog	
timestamp.**cache**	Cache of HVR catalogs used for routing. This is refreshed if option **-or** is supplied.
control	
tstamp.**ctrl-***jobname-ctrlname*	Control file containing instructions for a replication job. The contents of the file can be inspected using command **hvrrouterview**.
loc_*caploc*	
timestamp.**tx**_*intloc*	Data captured from location *caploc* that has been routed to location *intloc*. The contents of this file can be viewed using command **hvrrouterview**. If the base name of this file (*timestamp*) is bigger than the base name of any ***.cap_state** file, then this router data is not yet revealed and is still invisible to integrate locations.
timestamp.**cap_state**	Timestamps and capture status of capture job. The contents of this file can be viewed with command **hvrrouterview**.

See Also

Command Hvrinit.

Hvrscheduler

Contents

Name

hvrscheduler - HVR Scheduler server.

Synopsis

hvrscheduler [*-options*] *hubdb*

Description

The HVR Scheduler is a process which runs jobs defined in the catalog table **HVR_JOB**. This catalog table can be found in the hub database.

These jobs are generated by commands **Hvrinit**, **Hvrrefresh** and **Hvrcompare**. After they have been generated these jobs can be controlled by attributes defined by the jobs themselves and on the job groups to which they belong. These attributes control when the jobs get scheduled.

The first argument *hubdb* specifies the connection to the hub database. This can be an Oracle, Ingres, SQL Server, DB2, DB2 for I, PostgreSQL, or Teradata database depending on its form. See further section Calling HVR on the Command Line.

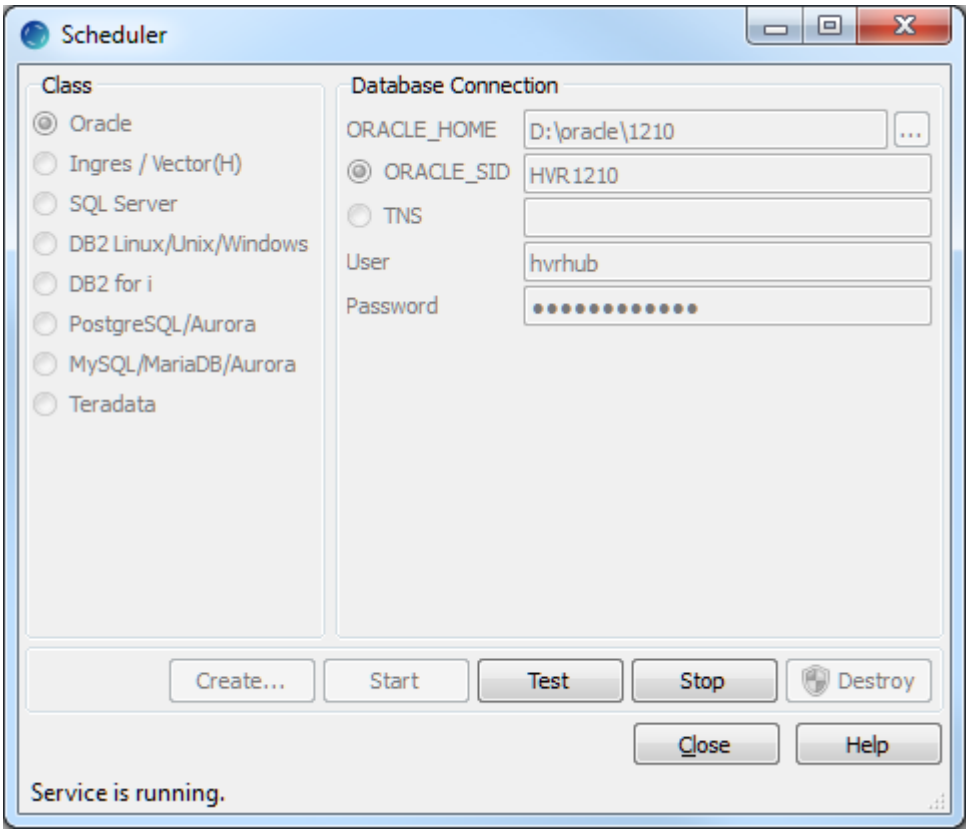
On Unix, the HVR Scheduler runs as a daemon. It can be started and stopped within the HVR GUI. Alternatively, on the Unix command line, it can be started using command **hvrscheduler** *hubdb* (no options) and stopped using **hvrscheduler -k**.

On Windows, the HVR Scheduler runs as a system service. It can be started and stopped within the HVR GUI. Alternatively, on the Windows command line, it can be created using command **hvrscheduler -ac**, started with **hvrscheduler -as** and stopped with **hvrscheduler -ah**.

Internally the HVR Scheduler uses a concept of 'Job Space', a two-dimensional area containing jobs and job groups. A job group may contain jobs and other job groups. In Job Space, jobs are represented as points (defined by X and Y coordinates) and job groups are represented as boxes (defined by four coordinates minimum X, maximum X, minimum Y and maximum Y). All jobs and job groups are contained within the largest job group, which is called **system**.

Options

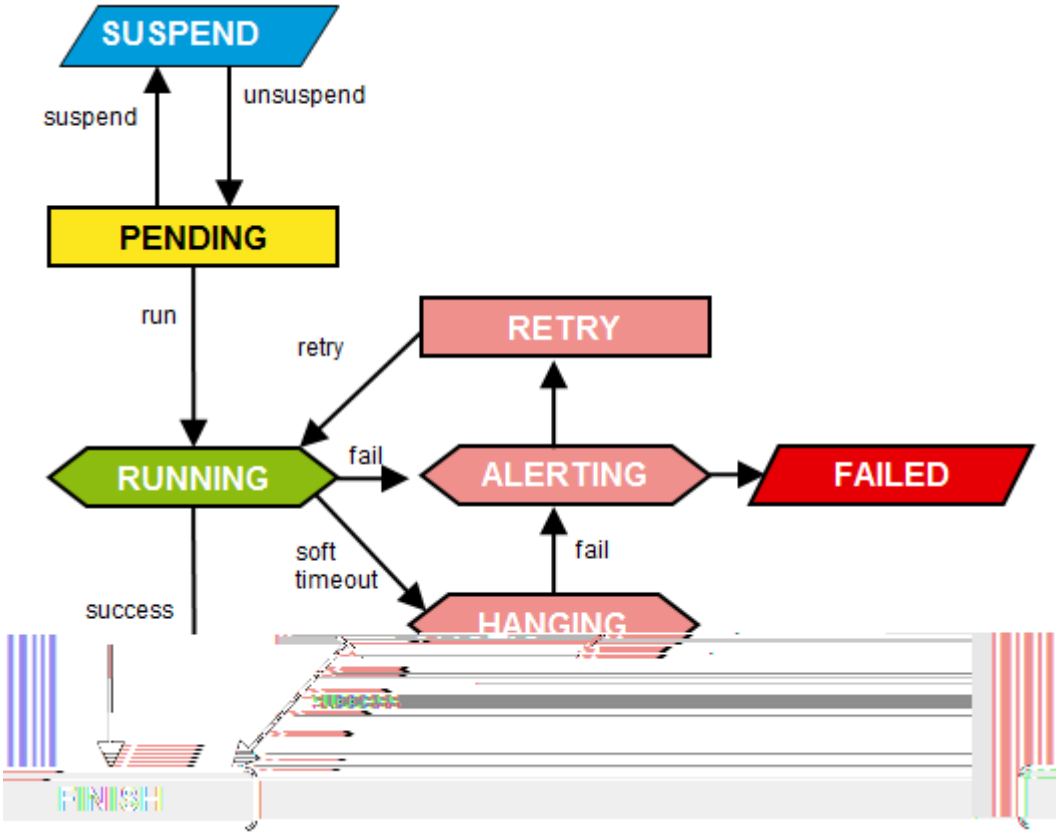
This section describes the options available for command **hvrscheduler**.



Parameter	Description
-a*x* **Windows**	Administration operations for the HVR Scheduler Microsoft Windows system service. Allowed values of *x* are: • **c** : Create the HVR Scheduler service and configured it to start automatically at system reboot. The service will run under the default system unless **-P** option is given. • **s** : Start the HVR Scheduler service. • **h** : Halt (stop) the system service. • **d** : Destroy the system service. Several **-a***x* operations can be supplied together, e.g. **-acs** (create and start) and **-ahd** (halt and destroy). Operations **-as** and **-ah** can also be performed from the window **Settings ControlPanel Services** of Windows.
-c*clus**clusgrp* **Windows**	Enroll the Scheduler Service in a Windows cluster named *clus* in the cluster group *clusgrp*. Once the service is enrolled in the cluster it should only be stopped and started with the Windows cluster dialogs instead of the service being stopped and started directly (in the Windows Services dialog or with options **-as** or **-ah**). In Windows failover clusters *clsgrp* is the network name of the item under **Services and Applications**. The group chosen should also contain the DBMS service for the hub database and the shared storage for **HVR_CONFIG**. The service needs to be created (with option **-ac**) on each node in the cluster. If this option is used to create the scheduler service in a cluster group, then it should also be added to option **-sched_option** of command **Hvrmaint**. This service will act as a 'Generic Service' resource within the cluster. This option must be used with option **-a**.
-E*n=v*	Set environment variable *n* to value *v* for this process and its children.
-F	Force start the **HVR Scheduler** process. This overrides certain checks that the scheduler does before starting. This is an internal option used by HVR.
-h*class*	Specify hub database. Valid values are **oracle**, **ingres**, **sqlserver**, **db2**, **db2i**, **postgresql**, and **teradata**. See also section Calling HVR on the Command Line.

-i	Interactive invocation. **HVR Scheduler** does not detach itself from the terminal and job output is written to **stdout** and **stderr** as well as to the regular logfiles.
-k **Unix & Linux**	Kill currently running **HVR Scheduler** process and any jobs which it may be running at that moment. When this option is used, it contacts the **HVR Scheduler** process to instruct it to terminate by itself. If this does not happen, it will kill the **HVR Scheduler** pr ocess using its process id (PID).
-K **Unix & Linux**	Kill immediately the currently running **HVR Scheduler** process and any jobs which it may be running at that moment. This is a variant of option **-k** except it skips the initial step of contacting the **HVR Scheduler** process and instructing it to terminate by itself.
-P*pwd* **Windows**	Configure the **HVR Scheduler** system service to run under the current login account using password *pwd*, instead of under the default system login account. May only be supplied with option **-ac**. Empty passwords are not allowed. The password is stored (hidden) within the Microsoft Windows OS and must be re-entered if passwords change.
-q*secs* **Unix & Linux**	Kill the **HVR Scheduler** process and any jobs which it may be running at that moment. This option allows *secs* seconds grace time before terminating the **HVR Scheduler** process. The **default** is 60 seconds. This parameter can also be passed from **hvrmaint** using the **-quiesce_grace** option.
-s*lbl*	Add label *lbl* to HVR's internal child co-processes. The scheduler uses two co-processes at runtime; one to make SQL changes to the hub database (**-swork**), and the other for listening for database events (**-slisten**).
-t*secs*	Connection timeout after *secs* seconds to the old **HVR Scheduler** process. The **default** is 10 seconds.
-u*user*[*/pwd*]	Connect to hub database using DBMS account *user*. For some databases (e.g. SQL Server) a password *pwd* must also be supplied.

Job States



The HVR Scheduler schedules jobs. Each job performs a certain task. At any moment a job is in a certain state. For instance, when a job is waiting to be run, it is in state **PENDING**; when a job is running, it is in state **RUNNING**.

Jobs can be either acyclic or cyclic. Acyclic jobs will only run once, whereas cyclic jobs will rerun repeatedly. When a cyclic job runs, it goes from state **PENDING** to **RUNNING** and then back to state **PENDING**. In this state it waits to receive a signal (trigger) in order to run again. When an acyclic job runs, it goes from state **PENDING** to **RUNNING** and then disappears.

If for some reason a job fails to run successfully the scheduler will change its state first to **ALERTING**, then **RETRY** and will eventually run again. If a job stays in state **RUNNING** for too long it may be marked with state **HANGING**; if it finishes successfully it will just become **PENDING**.

Output Redirection

Each message written by an HVR job is redirected by the scheduling server to multiple logfiles. This means that one logfiles exists with all output from a job (both its **stdout** and **stderr**). But another file has the **stderr** from all jobs in the channel.

Scheduler Attributes

Scheduler attributes are a component which is used to internally communicate (at the moment that HVR Initialize is run) the definition of features such as **Scheduling /CaptureStartTimes** to the run-time system. They are exposed in the HVR User Interface to allow the verification that these **Scheduling** actions have been propagated to the run-time system.

These scheduler attributes will be redesigned in a future HVR version. It is recommended not to change scheduler attributes that HVR generates automatically or not to create new ones.

Attribute	Arg1	Arg2	Description
quota_run	*n*		Maximum number of jobs which can be in **RUNNING** or **HANGING** state in job group. So if attribute **quota_run 2** is added to job groups **CHN1**, **CHN2** and **quota_run 3** is added to job group **SYSTEM**, then only two jobs can run for each channel (**CHN1** and **CHN2**) and only three jobs can run in the whole system.

quota_children	*n*		Maximum number of child processes associated with jobs in job group, including running jobs, but excluding the scheduler's own child processes.
quota_speed	*q*	*secs*	Limit the speed with which the scheduler starts jobs to no more than *q* job executions inside *secs* seconds. For example, **quota_speed 20 1** means that if there are lots of job ready to be started, then the scheduler will only start 20 jobs per second.
trig_delay	*secs*		Trigger cyclic job *secs* seconds after it last finished running (column **job_last_run_end** of **HVR_JOB**). If a cyclic job is not affected by any **trig_delay** attribute, it will remain **PENDING** indefinitely.
trig_crono	*crono*		Trigger job at *crono* moment. For the format of crono see **Scheduling /CaptureStartTimes**. After applying attribute **trig_crono**, the job needs to be in a **PENDING** state for the **HVR Scheduler** to trigger the job.
trig_at	*time*		Trigger job at specific (*time*) moment. Valid formats are *YYYY-MM-DD [HH:MM:SS]* (in local time) or *YYYY-MM-DDTHH:MM:SS+TZD* or *YYYY-MM-DDTHH:MM:SSZ* or **today** or **now***[±SECS]* or an integer (seconds since **1970-01-01 00:00:00 UTC**).
retry_max	*n*		Allow *n* retries for an unsuccessful job. If *n* is zero no retry is allowed. Jobs unaffected by this attribute are not retried: on error they become **FAILED** directly.
retry_delay	*isecs*	*fsecs*	Initially retry job after *isecs* and double this delay for each unsuccessful retry until *fsecs* is reached. The default value for *isecs* is **60** seconds and *fsecs* is **3600** seconds.
timeo_soft	*secs*		After job has been in **RUNNING** state for *secs* seconds, write time-out message and change its job state to **HANGING**. If *secs* is zero then no time-out will occur.
timeo_hard	*secs*		Terminates the job in **RUNNING** or **HANGING** state after it has run *secs* seconds. If *secs* is zero then no time-out will occur.
set	*name*	*val*	Set variable name to value *val* in job's runtime environment.

Environment Variables

Variable Name	Description
HVR_ITO_LOG	Causes the HVR Scheduler to write a copy of each critical error tot the file named in the variable's value. This can be used to ensure all HVR error messages from different hub databases on a single machine can be seen by scanning a single file. Long messages are not wrapped over many lines with a backslash '****', but instead are written on a single line which is truncated to 1024 characters. Each line is prefixed with "**HVR_ITO_AIC** *hubnode:hubdb locnode*", although **HVR** is used instead of **$HVR_ITO_AIC** if that variable is not set.
HVR_PUBLIC_PORT	Instructs the HVR Scheduler to listen on an additional (public) TCP/IP port number.

Examples

In Unix, start **HVR Scheduler** as a Unix daemon.

```
hvrscheduler hubdb
```

In Windows, create and start **HVR Scheduler** as a Windows Service.

```
hvrscheduler -acs hubdb
```

⚠ When starting the **HVR Scheduler** it is important that a database password is not exposed to other users. This can be encrypted using command **hvrcrypt**.

Files

HVR_HOME	
bin	
hvralert	Perl script used by scheduler to decide if jobs should be retried.
lib	
retriable.pat	Patterns indicating which errors can be handled by retrying a job.
HVR_CONFIG	
files	
sched*db_node***.pid**	Process-id file.
sched*db***.host**	Current node running scheduler.
log	
hubdb	
*job***.out**	All messages for job *jobname*.
*job***.err**	Only error messages for job *jobname*.
*chn***.out**	All messages for channel *chn*.
*chn***.err**	Only error messages for channel *chn*.
hvr.out	All messages for all jobs.
hvr.err	Only error messages for all jobs.
hvr.ctrl	Log file for actions from control sessions.
HVR_TMP	Working directory for executing jobs.

See Also

Commands Hvrcrypt, Hvrsuspend, Hvrstart.

Hvrsslgen

Contents

Name

hvrsslgen - Generate a private key and public certificate pair.

Synopsis

hvrsslgen [*-options*] *basename subj*

Description

Command **hvrsslgen** generates a private key and public key pair required for SSL Connection. These key files together are required for establishing a secure encrypted connection between HVR hub and remote HVR locations. Both files (private key and public key) are needed on the remote machine, however, only the public key file must be copied to the hub machine.

By default, the generated key's length is **2048** bits, and the private key is encrypted using **aes-256-cbc** algorithm and the SSL certificate is signed using **sha256** hash algorithm. This can be customized by using the *options* available for **hvrsslgen**.

Command argument *basename* is used for naming the key files. The private key file is named *basename***.priv_key** and the corresponding public key file is named *basename***.pub_cert**.

The second argument *subj* is written as plain text into the subject field of the X509 public certificate file and serves for reference purposes only. If argument *subj* contains two or more words with space between them, then it must be enclosed in double quotes. For example, "Certitficate for Cloud".

Options

This section describes the options available for command **hvrsslgen**.

Parameter	Description
-a*bits*	Generate an asymmetric (RSA) key pair with length *bits*. The **default** is **2048** bits.
-d*dir*	Generate files in directory *dir* instead of current directory.

-e*enc_alg*	Encrypt the private key using an internal password with encryption algorithm *enc_alg*. Valid values for *enc_alg* are: • **aes-128-cbc** • **aes-192-cbc** • **aes-256-cbc** (**default**) • **aes-128-cfb** • **aes-192-cfb** • **aes-256-cfb** • **aes-128-ecb** • **aes-192-ecb** • **aes-256-ecb** • **des-56-cbc** • **des-168-cbc**
-h*hash_alg*	Sign the SSL certificate using hash algorithm *hash_alg*. Valid values for *hash_alg* are: • **sha1** • **sha256** (**default**) • **sha512** • **md5**

Example

Run the following command to generate the private key and public certificate key pair:

```
hvrsslgen -a2048 -eaes-256-cfb -hsha512 MyCertificate "Certificate for Cloud"
```

The output will be as follows:

```
hvrsslgen: Generating SSL key pair...
hvrsslgen: Generating SSL key pair... completed.
hvrsslgen: Private key written to 'MyCertificate.priv_key'.
hvrsslgen: Public Certificate written to 'MyCertificate.pub_cert'.
hvrsslgen: Certificate subject: 'HVR Certificate for Cloud'
hvrsslgen: Certificate contains 2048 bit RSA Public Key.
hvrsslgen: Certificate valid from Nov  4 10:11:57 2015 GMT
hvrsslgen: Certificate valid until Oct 30 10:11:57 2035 GMT
hvrsslgen: Finished. (elapsed=1.85s)
```

Hvrstart

Contents

Name

hvrstart - Start jobs.

Synopsis

hvrstart [*-options*] *hubdb jobs…*

Description

Command **hvrstart** causes HVR jobs to be run. The jobs are either run via the **HVR Scheduler** or they are run directly by using **hvrstart -i** command. Jobs can either be specified explicitly (e.g. *chn*-**cap**-*locx*) or they can be partially specified (e.g. *chn*-**cap** which matches all capture jobs). If only a channel (*chn*) name is specified, then **hvrstart** runs the *chn*-**cap** jobs and then the *chn*-**integ** jobs.

If the jobs are run via the scheduler (no **-i** option), then jobs in a **SUSPEND** state are immune unless option **-u** is used. Jobs in state **FAILED** or **RETRY** are also immune unless option **-r** is used. In this mode, the **HVR Scheduler** process must already be running. If the job is already running, then the Scheduler will force the job to wake up and perform a new replication cycle.

The first argument *hubdb* specifies the connection to the hub database. For more information about supported hub databases and the syntax for using this argument, see Calling HVR on the Command Line.

Options

This section describes the options available for command **hvrstart**.

Parameter	Description
-C*pub_cert*	Public certificate for encrypted connection to hub machine. This must be used with option **-R**.
-E*nm=val*	Set environment variable *nm* to value *val*.
-h*class*	Specify hub database *class*. For supported values, see Calling HVR on the Command Line. This option is only required when option **-i** is used for running compare or refresh jobs.
-i	Interactive. The HVR job is run directly instead of via HVR Scheduler. The job's output and errors are sent to **stdout** and **stderr**.
-K*pair*	SSL public certificate and private key of local machine. If *pair* is relative, then it is found in directory **$HVR_HOME/lib/cert**. Value *pair* specifies two files; the names of these files are calculated by removing any extension from *pair* and then adding extensions.**pub_cert** and.**priv_key**. For example, option **-Khvr** refers to files **$HVR_HOME/lib/cert/hvr.pub_cert** and **$HVR_HOME/lib/cert/hvr.priv_key.**

-L*login/pwd*	Login/password for remote hub machine. Must be used with option **-R** *node:port*.
-r	Retry **FAILED** or **RETRY** jobs by triggering them with value **2** in column **job_trigger** of catalog **hvr_job**. This option cannot be used with option **-i**.
-R*node:port*	Remote hub machine node name and TCP/IP port number. Must be used with option **-L** *login/pwd*.
-t*N*	Time-out after *N (> 0)* seconds if scheduler or job takes too long, or if network is hanging. Job execution is not interrupted by this client timeout. If no **-t** option is supplied then **hvrstart** will wait indefinitely. This option cannot be used with option **-i**.
-u	Unsuspend. This option cannot be used with option **-t***N* or **-i**.
-U*user*[*/pwd*]	Connect to hub database using DBMS account *user*. For some databases (e.g. SQL Server) a password */pwd* must also be supplied, but only when running compare or refresh jobs. This option is only required when option **-i** is used for running compare or refresh jobs.
-w	Wait until all triggered jobs which were selected have finished running or have completed a full replication cycle. While **hvrstart** is waiting for a job, its output is carbon-copied to the **hvrstart** command's **stdout** and **stderr**. This option cannot be used with option **-i**.

Exit Codes

0	Success. If option **-i** is not used, then a success just means that the HVR Scheduler was able to run the jobs, not that the job succeeded.
1	Failure. If option **-i** is not used, then this means error sending instruction to HVR Scheduler server, job did not exist, etc.
2	Time-out.
3	Jobs existed in state **FAILED** , **RETRY** , **ERROR** , **ALERTING** or **SUSPEND** which were not affected by the command.

Example

- Run a capture job from the command line, without the **HVR Scheduler**:

```
hvrstart -i hubdb chn-cap-loc
```

- Run all integrate jobs via the HVR Scheduler:

```
hvrstart hubdb chn-integ
```

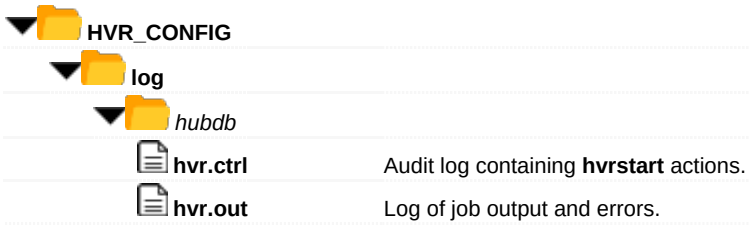
- Run all capture jobs and then all integrate jobs from the command line:

```
hvrstart -i hubdb chn
```

- Run a refresh job from the command line:

```
hvrstart -i -h sqlserver -U user/pwd hubdb chn-refr-loc1-loc2
```

Files



Hvrstatistics

Contents

Name

hvrstatistics - Extract statistics from HVR scheduler logfiles.

⚠ Since HVR 5.3.1/25, **hvrstatistics** is replaced with **Hvrstats**.

Synopsis

hvrstatistics [*-options*]... [*hubdb*]

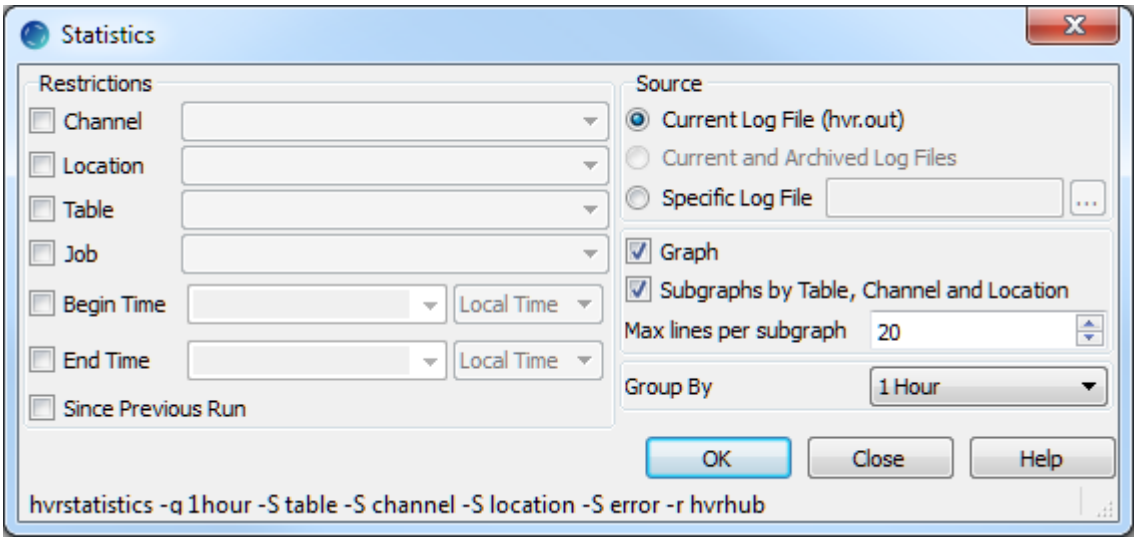
Description

hvrstatistics displays the statistics from HVR scheduler logfiles. The first argument *hubdb* specifies the connection to the hub database. For more information about supported hub databases, see Calling HVR on the Command Line.

⚠ **hvrstatistics** cannot process files containing more than 12 months of data.

Options

This section describes the options available for command **hvrstatistics**.

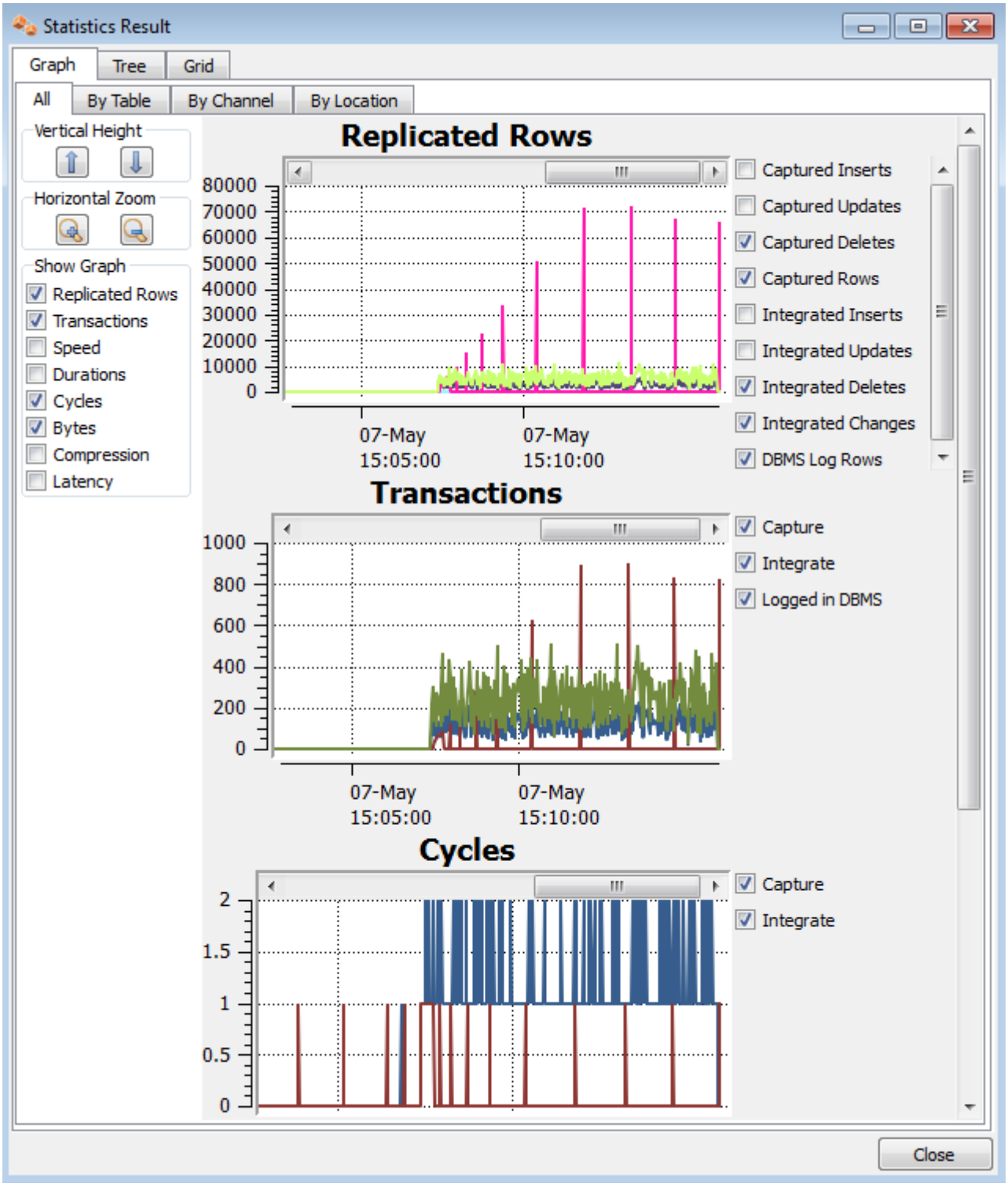


Parameter	Description
-b*time*	Only lines after (begin) *time*. Argument must have one of the following formats *YYYY-MM-DD HH:MI:SS* or *YYYY-MM-DDTHH:MM:SS+TZD* or *YYYY-MM-DDTHH:MM:SSZ*.
-c*chn*	Only parse output for specific channel *chn*. Alternatively, a specific channel can be omitted using form **-c!***chn*. This option can be specified multiple times.
-e*time*	Only lines until (end) *time*. Argument must have one of the following formats *YYYY-MM-DD HH:MI:SS* or *YYYY-MM-DDTHH:MM:SS+TZD* or *YYYY-MM-DDTHH:MM:SSZ*.
-f*file*	Parse scheduler log file *file*, instead of default **$HVR_CONFIG/log/***hubdb***/hvr.out**.
-g*col*	Summarize totals grouped by *col* which can be either **channel**, **location**, **job**, **table**, **year**, **month**, **day**, **hour**, **minute** or **second**. This option can be specified multiple times, which will subdivide the results by each column. Additionally, a reasonable subdivision of time-based columns can be specified, e.g.: **-s "10 minutes"** or **-s "6 hours"**
-i	Incremental. Only lines added since previous run of **hvrstatistics**. The position of the last run is stored in file **$HVR_CONFIG/files/hvrstatistics.offset**.
-l*name*	Incremental with variable status file. Only lines added since previous run of **hvrstatistics**. The position of the last run is stored in file **$HVR_CONFIG/files/hvrstatistics**_*name*__**offset**.
-l*loc*	Only parse output for specific location *loc*. Alternatively, a specific location can be omitted using form **-l!***loc*. This option can be specified multiple times.
-r	Resilient. Do not show log file output lines which cannot be matched.
-s*sep*	Print parsed output in CSV-matrix with field separator *sep*. This allows the input to be imported into a spreadsheet.
-S*col*	Summarize totals grouped by *col* which can be either **channel**, **location**, **job**, **table**, **year**, **month**, **day**, **hour**, **minute** or **second**. This option can be specified multiple times, which will cause the same data to be repeated in multiple blocks, but with each block divided by a different column.
-v	Verbose trace messages.

⚠ For list of all statistics metrics, see Metrics for Statistics.

Examples

HVR Statistics can be run from inside the HVR GUI, or it can be run on the command line. The following screenshot shows an example of the HVR Statistics inside the GUI.



On the command line, to count total rows replicated for each location and table, use the following:

```
hvrstatistics -g location -g table myhub
```

Sample output:

```
Location cen
  Table order
     Captured rows           :     25
  Table product
     Captured rows           :    100
  Capture cycles             :      6
  Routed bytes               :  12053
Location dec01
  Table order
     Integrated updates      :     25
     Integrated changes      :     25
  Table product
     Integrated updates      :    100
     Integrated changes      :    100
  Integrate cycles           :      7
  Integrate transactions     :      4
```

To create a CSV file with the same data use option **-s** as follows:

```
hvrstatistics -g location -g table -s"," myhub > stats.csv
```

If this CSV file was imported into a spreadsheet (e.g. Excel) it would look this:

Location	Table	Capture cycles	Captured rows	Routed bytes	Integrate cycles	Integrate transactions	Integrated updates	Integrated changes
cen	order		25					
cen	product		100					
cen		6		12053				
decen	order						25	25
decen	product						100	100
decen					7	4		

Files

HVR_CONFIG

files

hvrstatistics.offset State file for incremental statistics (option **-i**).

Hvrstats

Since v5.3.1/25

Contents

Name

hvrstats - Gather or output statistics information.

⚠ Statistics generation for HVR version 5.3.1/24 and older, see **Hvrstatistics**.

Synopsis

hvrstats [**-h** *class*] [**-u** *user*] **-C***lett hubdb*

hvrstats [**-h** *class*] [**-u** *user*] **-f***logf* [**-f***logf*]... [**-T***gran*] *hubdb*

hvrstats [**-h** *class*] [**-u** *user*] **-g***N* [**-T***gran*] [**-G***typ*] *hubdb*

hvrstats [**-h** *class*] [**-u** *user*] **-o***fname* [*-outopts*] *hubdb*

hvrstats [**-h** *class*] [**-u** *user*] **-p***pol hubdb*

Description

Command **hvrstats** can be invoked in five distinct ways:

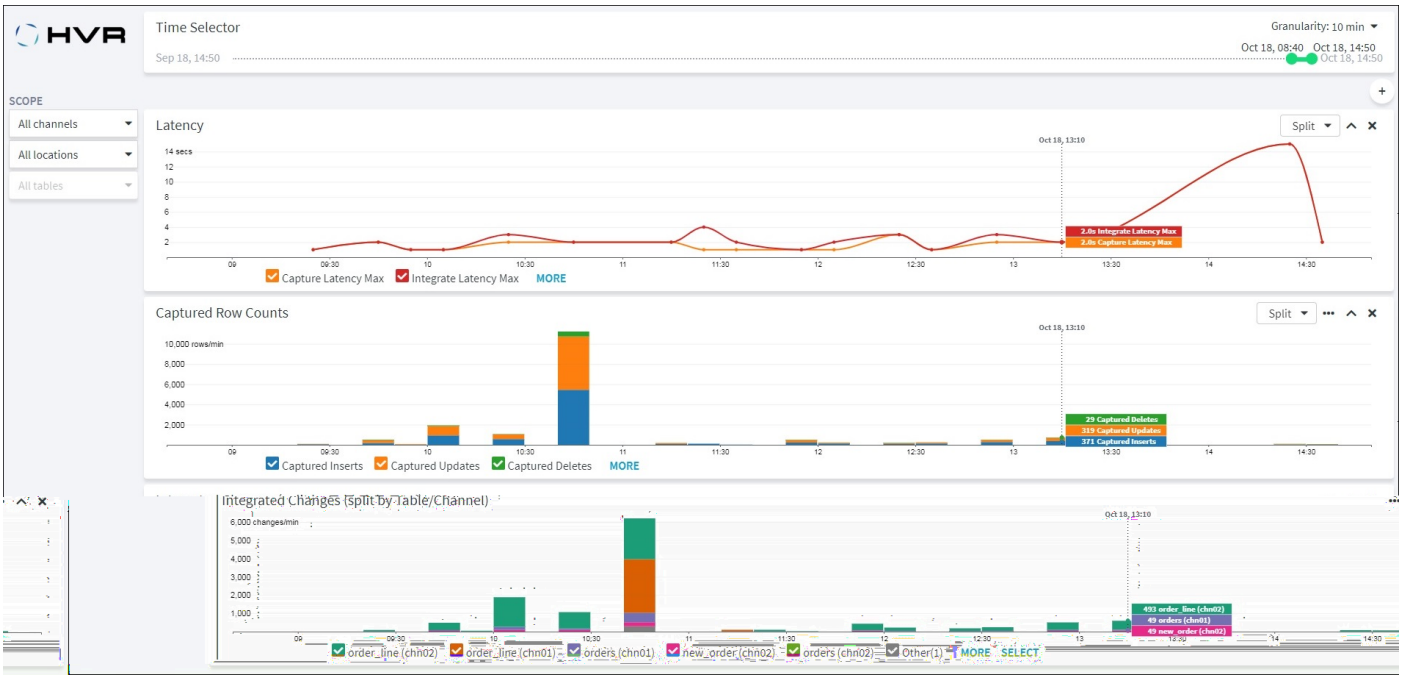
Option **-C** creates database objects.

Option **-f** gathers statistics information from a static log file.

Option **-g** gathers live information into table **hvr_stats**.

Option **-o** outputs statistics information from table **hvr_stats** to a file.

Option **-p** purges old data from table **hvr_stats**.



Regular Options

Parameter	Description
-C*letters*	Create database objects for **hvrstats**. Value *letters* can be one or more of the following: • **t** : Create tables **hvr_stats** and **hvr_stats_staging**. • **j** : Create **hvrstats** job in scheduler. Once created, the job can be started and suspended using commands **Hvrstart** and **Hvrsuspend** respectively. This option cannot be used with **-g -f -o** or **-p**.
-f*logf*	Gather statistics measurements from HVR log file *logf*. This option can be supplied multiple times. Examples of use are to catch-up with the current log file (**$HVR_CONFIG/log/***hubdb***.out**) or to consume archived log files (in **$HVR_CONFIG/logarchive**) . This option does not change the statistics offset state file. This option cannot be used with **-C -g -o** or **-p**.
-g*bool*	Gather information from runtime; normal run-time **hvrstats** processing. Value *bool* should either be **0** (run continuously in a loop until terminated) or **1** (perform just one [full] cycle, then stop). This option cannot be used with **-C -f -o** or **-p**.
-G*letters*	Type of information to gather. By **default** all types of information is gathered (but not at same frequency). Value _*letters*_can one of the following: • **j** : Job information, including latency (from **$HVR_CONFIG/router**) and log files sizes (from **$HVR_CONFIG/log**). • **s** : Statistics metrics from live HVR log file's contents. This option requires option **-g** (gather). If this option is not supplied then all types of information is gathered (**-Gjs**).
-h*class*	Hub class, for connecting to hub database.

-o*fname*	Writes statistics information (fetched from table **hvr_stats**) into file *fname*. The default file format is JSON, for other file formats see output option **-V**. To filter the output written into file *fname*, you can use the output options along with **-o**.
-p*policy*	Purge old records immediately from the catalog table **hvr_stats**. Value *policy* can be one of the following: • **SMALL** : Per-table measurements at 1min/10 min/1hour/1day granularity are purged after 1hour /4hours/1day/7days respectively. Rows for all tables (table=*) at 1min/10 min/1hour/1day granularity are purged after 4hours/1day/7days/30days respectively. • **MEDIUM** : Per-table measurements at 1min/10 min/1hour/1day granularity are purged after 4hours/1day/7days/30days respectively. Rows for all tables (table=*) at 1min/10min/1hour/1day granularity are purged after 1day/7days/30days/never respectively. • **LARGE** : Per-table measurements at 1min/10min/1hour/1day granularity are purged after 1day /7days/30days/never respectively. Rows for all tables (table=*) at 1min/10min/1hour/1day granularity are purged after 7days/30days/never/never respectively. Values **NONE** and **UNBOUNDED** are not allowed here but are valid for action **Scheduling /StatsHistory**. This option cannot be used with **-C -g -f** or **-o**.
-T*gran*	Time granularity of data to gather or to output. Value *gran* must be only one of the following: • **m** : Minute granularity • **t** : Ten (10) minutes granularity • **h** : Hour granularity • **d** : Day granularity • **c** : Current granularity. This letter is allowed with option **-o**(view output), not option **-g**(gather from runtime). This option can only be used with **-f** (gather from file), **-g**(gather from runtime) or **-o**(view output). When gathering (option **-g**) if this option is omitted the default is **m** (minute granularity). Also, when gathering (but not when showing) if a small granularity is supplied then large granularities (e.g. **m > t > h > d**) will also be calculated. For example for option **-T t** (for 10 minutes) is supplied then aggregate values are also calculated for hour and day granularity. With option **-o**(view output), multiple letters can be supplied and the default is to return all time granularities (**-T mthd**).
-u*user*[/pwd]	Username for hub database.

Output Options

The following options (*-outopts*) can only be used with option **-o**.

⚠ For list of all statistics metrics, see Metrics for Statistics.

Parameter	Description
-b*begin_time*	Only write statistics information since *begin_time*. Value *begin_time* must have form *YYYY-MM-DD HH:MM:SS*, *YYYY-MM-DD***T***HH:MM:SS+TZD* or *YYYY-MM-DD***T***HH:MM:SS***Z**.
-c*chn*	Only write statistics information for channel *chn*. This option can be supplied multiple times.
-e*end_time*	Only write statistics information upto *end_time*. Value *end_time* must have form *YYYY-MM-DD HH:MM:SS*, *YYYY-MM-DD***T***HH:MM:SS+TZD* or *YYYY-MM-DD***T***HH:MM:SS***Z**.
-l*loc*	Only write statistics information for location *loc*. This option can be supplied multiple times.
-m*mchoice*	Only write statistics information for specific metrics. Values *mchoice* can be either a metric name (e.g. **Integrated Updates**), a group of metrics (e.g. **Latency**) or a named label of metrics (__kpi_lines). This option can be supplied multiple times; if it is not supplied then all metrics are displayed.

-s*scope*	Only write statistics information for metric with specific *scope*. A scope is identified by three letters for channel, location and table The first letter of *scope* is either **c** if the value is for a specific channel or ***** if it is associated (an aggregate) for all channels. The second is either **l** if the value is for a specific location or ***** if it is associated (an aggregate) for all locations. The third is either **t** if the value is for a specific table or ***** if it is associated (an aggregate) for all tables. Value *scope* must be one of the following: • **clt** : Specific (named) channel, location and table. • **cl*** : Specific channel and location, but for all tables (e.g. table=*). • **c*t** : Specific channel and table, but for all locations. • ***l*** : Specific locations, but for all channels and tables. • **c**** : Specific channel, but for all locations and tables. • ******* : General values, which apply to all channels, locations and tables. If this option is not supplied then measurements for all scope are shown. This option can be supplied multiple times. **cl*** ***l*** **clt** ******* **c*t** **c**** Note that two combinations (***lt** and ****t**) are not supported.
-t*tbl*	Only write statistics information for table *tbl*. This option can be supplied multiple times.
-V*fmt*	Format of the output file *fname*. Value *fmt* can be one of the following: • **json** (**default**). This format is available only in CLI. • **csv** • **xlsx** **Since** v5.6.0/1
-w*time*	Only write statistics information that was updated after *time*. Value *time* must have form *YYYY-MM-DD HH:MM:SS*, *YYYY-MM-DD***T***HH:MM:SS+TZD* or *YYYY-MM-DD***T***HH:MM:SS***Z**.

Example

This command will create the **hvrstats** catalog tables (if necessary), gather all data from a log file (**-f <hvr_log>**), select data for time granularity '10 minutes' (**-Tt**) into a file (**-o <ofile>**) and purge (**-p**) old rows according to the **SMALL** purge policy. Note that these actions will be performed in that exact order.

hvrstats -Ct -f <hvr_log> -o <ofile> -Tt -pSMALL <hub_db>

Files

- **HVR_CONFIG**
  - **files**
    - **hvrstatistics-stats-***hubdb***.offset** Statistics state file.
    - **hvr_stats_staging_***hubdb***.xml**

See Also

- Statistics

Hvrstrip

Contents

Name

hvrstrip - Purge HVR installation files.

Synopsis

hvrstrip [*-options*]

Description

Command **hvrstrip** allows you to purge HVR installation files. This command is used to remove either the old /unnecessary HVR files after upgrading HVR or the HVR installation files (on a remote server) that are not required for the HVR remote agent.

Options

This section lists and describes all options available for **hvrstrip**.

Parameter	Description
-p	Displays the list of files that will be purged using option **-P**. This option cannot be used with options **-P**, and **-r**.
-P	Purge HVR installation files that not required after performing an HVR upgrade. The list of files that will be purged can be viewed using option **-p**. This option cannot be used with options **-p**, and **-r**.
-r	Purge all HVR installation files that are not required for the HVR remote agent. This option cannot be used with options **-p**, and **-P**. This option should not be used on a server running HVR hub.

Hvrsuspend

Contents

Name

hvrsuspend - Suspend (or un-suspend) jobs

Synopsis

hvrsuspend [*-options*] *hubdb jobs*...

Description

In the first form **hvrsuspend** will force jobs in the HVR Scheduler into **SUSPEND** state. The second form (with option **-u**) will un-suspend jobs, which means that they will go into **PENDING** or **RUNNING** state.

Jobs can either be specified explicitly (e.g. *chn*-**cap**-*locx*) or they can be partially specified (e.g. *chn*-**cap** which matches all capture jobs). If only a channel name is specified, then **hvrsuspend** suspends all jobs in the channel.

The first argument *hubdb* specifies the connection to the hub database. For more information about supported hub databases, see Calling HVR on the Command Line.

This command connects to the **HVR Scheduler** so the scheduler must be already running.

Options

This section describes the options available for command **hvrsuspend**.

Parameter	Description
-C*pub_cert*	Public certificate for encrypted connection to hub machine. This must be used with option **-R**.
-E*nm=val*	Set environment variable *nm* to value *val*.
-K*pair*	SSL public certificate and private key of local machine. If *pair* is relative, then it is found in directory **$HVR_HOME/lib/cert**. Value *pair* specifies two files; the names of these files are calculated by removing any extension from *pair* and then adding extensions **.pub_cert** and **.priv_key**. For example, option **-Khvr** refers to files **$HVR_HOME/lib/cert/hvr.pub_cert** and **$HVR_HOME/lib/cert/hvr.priv_key.**
-L*login/pwd*	Login/password for remote hub machine. Must be used with option **-R***node:port*.
-R*node:port*	Remote hub machine node name and TCP/IP port number. Must be used with option **-L***login/pwd*.
-u	Unsuspend.

Example

A change has been made to the HVR catalogs for location **d01** but for the change to take effect the job's script must be regenerated.

```
$ hvrsuspend hubdb chn              # Halt all replication jobs
$ hvrinit -oj -ld01 hubdb chn       # Regenerate script for location d01
$ hvrsuspend -u hubdb chn           # Reactivate replication
```

•
•
•
•
•
  •
  •
  •
  •

-h*class*	Specify hub database. For supported values, see Calling HVR on the Command Line.
-i*latency*	Specify an initial minimum *latency*. Switch if tables will not be performed if one of the involved jobs has a initial latency higher than *latency*. Valid inputs are an integer (in seconds) or a time duration (e.g. **10s, 3m, 1h**).
-k	Keep *chn1* integrate jobs. Do not delete integrate jobs of *chn1* after successful switch.
-m*moment*	Specify future switch moment relative to **hvr_cap_tstamp**. Valid formats are *YYYY-MM-DD [HH:MM:SS]* (in local time) or *YYYY-MM-DD***T***HH:MM:SS+TZD* or *YYYY-MM-DD***T***HH:MM:SS***Z** or **now***[±SECS]* (current time, not **hvr_cap_tstamp**) or an integer (seconds since **1970-01-01 00:00:00 UTC**).
-s	Do not suspend and restart integrate jobs when aborting.
-t*table*	Specify table scope. Only switch table *table* from *chn1* to *chn2*. This option can be supplied multiple times. If not specified all tables will be switched and integrate jobs of *chn1* will be deleted afterward (unless **-k** is specified). It is not possible to switch all tables using option **-t**.
-u*user*[*/pwd*]	Connect to hub database using DBMS account *user*. For some databases (e.g. SQL Server) a password must also be supplied.

Examples

Example 1: Adding tables to a running channel via a temporary channel

Consider a running channel **chn_main** replicating tables **tab_main1** and **tab_main2** from capture location **cap** to integrate location **integ**. Location **cap** is member of location group **CAP** and location **integ** is member of location group **INTEG**. **chn_main** is running with a low latency and without errors. Now **hvrswitchtable** can be used to add new tables **tab_new1** and **tab_new2** without interrupting capture or integrate. First a new channel **chn_tmp** is created with the same location groups **CAP** and **INTEG**. Then new tables **tab_new1** and **tab_new2** are added using **hvradapt,chn_tmp** is initialize using **hvrinit** and after starting the capture job the new tables are refreshed with an online **hvrrefresh**. After **hvrrefresh** completed successfully the integrate job can be started and HVR will start replicating all changes made to the new tables. After ensuring that **chn_tmp** runs fine with a low latency, a switch if the tables of **chn_tmp** into **chn_main** can be prepared.

Executing the following command will first check that all capture and integrate jobs in **chn_tmp** and **chn_main** run with a latency lower than 10 seconds. If so it adds tables **tab_new1** and **tab_new2** to **chn_main** and prepares to switch the tables in 1 minute.

```
$ hvrswitchtable -m now+60 -l 10s hubdb chn_tmp chn_main
```

Note that the 'now' in the command line refers to the current time (not relative to **hvr_cap_tstamp**). After **chn_tmp** has captured and integrated all changes that were made to **tab_new1** and **tab_new2** up to 1 minute after **hvrswitchtable** was run, **chn_main** will capture and integrate all future changes and the integrate job of **chn_tmp** will be deleted. Now all tables have been switched and **chn_tmp** can be deleted.

Example 2: Schedule and abort a merge of two channels

The scenario is similar to Example 1 but now location group **INTEG** contains a second integrate location **intg2** and **chn_tmp** is already running fine with a low latency. In this example the current time is '2017-12-10T14:30:00+01:00'.

Executing the following command prepares the switch for '2017-12-10T15:00:00+01:00', i.e., in 30 minutes and the integrate jobs of **chn_tmp** will not be deleted after the switch:

```
$ hvrswitchtable -m 2017-12-10T15:00:00+01:00 -k hubdb chn_tmp chn_main
```

If you want to abort the switch before '2017-12-10T15:00:00+01:00', execute the following command:

```
$ hvrswitchtable -a -s hubdb chn_tmp chn_main
```

Aborting the switch removes tables **tab_new1** and **tab_new2** from **chn_main** and the replication of both channels will continue without any interruption.

Assume the switch was not aborted and at '2017-12-10T14:58:00+01:00' the integrate job for location **intg2** in channel **chn_tmp** (**chn_tmp-integ-intg2**) failed due to some integration error. At '2017-12-10T15:00:00+01:00' tables have been switched successfully for location **integ**, but **chn_tmp-integ-intg2** is still failing. The integrate job for **intg2** in **chn_main** (**chn_main-integ-intg2**) will wait for **chn_tmp-integ-intg2** to integrate all changes up to '2017-12-10T15:00:00+01:00', so now it is hanging. To abort the switch only for location **intg2**, execute the following command:

```
$ hvrswitchtable -A intg2 hubdb chn_tmp chn_main
```

This will not remove tables **tab_new1** and **tab_new2** from **chn_main**, but will restart **chn_main-integ-intg2** and ensure that **chn_main** will not replicate any changes for **tab_new1** and **tab_new2**. After integration issues in **chn_tmp** have been resolved, integration of **tab_new1** and **tab_new2** into **intg2** will continue in **chn_tmp**.

Example 3: Switch some tables between two running channels

Consider a situation where two identically configured channels **chn_from** and **chn_to** are moving tables at a low latency. For this example **chn_from** is replicating tables **tab1**, **tab2** and **tab3** and **chn_to** is replicating tables **tab4** and **tab5**. To switch tables **tab2** and **tab3** from **chn_from** to **chn_to** in 1 hour execute the following command.

```
$ hvrswitchtable -m now+3600 -t tab2 -t tab3 hubdb chn_to chn_from
```

This will add **tab2** and **tab3** and all their associated actions to **chn_to**. After all involved integrate jobs are passed the given switch moment (i.e. **hvr_cap_tstamp** has gone passed now +1 hour), **chn_to** will replicate all changes for **tab2** and **tab3** and the tables and their associated actions will be removed from **chn_from**. That means **chn_from** will now only replicate **tab1** and **chn_to** will replicate **tab2**, **tab3**, **tab4** and **tab5**. No further actions are required and **chn_from** and **chn_to** will continue with their replication. Note that aborting of the switch works as in Example 2, only the table scope (option **-t**) must be provided for the abort command as well.

Example 4: Moving a portion of tables to a new secondary channel

Consider a situation where the main channel **chn_main** is running with many tables being replicated from capture location **cap** to integrate location **integ**. Location **cap** is a member of location group **CAP** and location **integ** is a member of location group **INTEG**. Channel **chn_main** is running with low latency and without errors. A new secondary channel **chn_sec** needs to be created that would take 50% of the load from the main channel **chn_main**. It is assumed that the source is always active/live and that the main channel **chn_main** has been running for some time. All checkpoints, including the capture and integrate checkpoints, should be maintained.

To implement this, first a secondary channel **chn_sec** is created with the same location groups **CAP** and **INTEG**. Then a baseline table is added to the secondary channel using **hvradapt**, **chn_sec** is initialized using **hvrinit**, and after starting the capture job, the baseline table is refreshed with an online **hvrrefresh**. After **hvrrefresh** completed successfully, the integrate job can be started.

To move half of the tables from **chn_main** to **chn_sec** in 1 hour, execute the following command:

```
$ hvrswitchtable -m now+3600 -t tab1 -t tab2 – tab3 hubdb chn_sec chn_main
```

If there are multiple tables in **chn_main**, each table to be moved to **chn_sec** needs to be listed with **-t** in the command

After the switch completed successfully, the baseline can be deleted from **chn_sec** (optional).

Checkpoints

No manual action is required to maintain correct checkpoints for the new secondary channel **chn_sec**. Up to the switch point, the capture state, capture checkpoints and integrate state for the tables being moved are handled by the original channel **chn_main**. After the switch (which is always on the transaction boundary), the capture state, capture checkpoints and integrate state will automatically be handled by the secondary channel **chn_sec**.

Hvrtestlistener, hvrtestlocation, hvrtestscheduler

Contents

Name

hvrtestlistener - Test listening on TCP/IP port for HVR remote connection

hvrtestlocation - Test connection to HVR location

hvrtestscheduler - Test (ping) that the HVR Scheduler is running.

Synopsis

hvrtestlistener [**-C***pubcert*] [**-K***pair*] [**-L***login/pwd*] [**-t***N*] *node port*

hvrtestlocation [**-h***class*] [**-l***loc*]... [**-t***N*] [**-u***user*] *hubdb chn*

hvrtestscheduler [**-n***node*][**-t***N*] *hubdb*

Description

Command **hvrtestlistener** tests that an HVR process is listening on a TCP/IP port for a HVR remote connection. If option **-L** is supplied then it also tests the authentication for that login and password.

Command **hvrtestlocation** tests a connection to an HVR location.

Command **hvrtestscheduler** checks if the HVR scheduler is running.

Options

This section describes the options available for command **hvrtestlistener**.

Parameter	Description
-C*pubcert*	Remote public certificate for testing encrypted SSL connection.
-h*class*	Specify hub database. For supported values, see Calling HVR on the Command Line.
-K*pair*	SSL public certificate and private key of local machine. If *pair* is relative, then it is found in directory **$HVR_HOME/lib/cert**. Value *pair* specifies two files; the names of these files are calculated by removing any extension from *pair* and then adding extensions **.pub_cert** and **.priv_key**. For example, option **-Khvr** refers to files **$HVR_HOME/lib/cert/hvr.pub_cert** and **$HVR_HOME/lib/cert/hvr.priv_key.**

-l*x*	Test locations specified by *x*. If this option is not supplied, then **hvrtestlocation** will test all locations within the channel. Values of *x* may be one of the following: • *loc* : Only test location *loc*. • *l1*-*l2* : Test all locations that fall alphabetically between *l1* and *l2* inclusive. • **!***loc* : Test all locations except *loc*. • **!***l1*-*l2* : Test all locations except for those that fall alphabetically between *l1* and *l2* inclusive. Several **-l***x* instructions can be supplied together to **hvrtestlocation**.
-L*login/pwd*	Test authentication of login/password on remote machine.
-n*node*	Connect to HVR Scheduler running on node.
-R*node:port*	Connect to node as a proxy. This option can be supplied multiple time for a chain of proxies. For more information, see section **Hvrproxy**.
-t*N*	Time-out after *N* seconds if network is hanging or HVR Scheduler takes too long to reply.
-u*user*[*/pwd*]	Connect to hub database using DBMS account *user*. For some databases (e.g. SQL Server) a password must also be supplied.

Hvrvalidpw

Contents

Hvrvalidpw allows customization of how the HVR executable validates the username/password of incoming connections. This overrides the default behavior, which is to validate username/password as operating system credentials. **hvrvalidpw** is not a command to be executed manually in the command line to authenticate a user; it is only a plugin which is invoked by HVR for authentication. For more information about authentication modes and access control in HVR, see Authentication and Access Control.

HVR distribution includes the following authentication plugins:

- LDAP Authentication - Hvrvalidpwldap Plugin
- Private Password File Authentication - Hvrvalidpwfile Plugin
- Custom Hvrvalidpw Authentication

> For HVR to invoke either of the above mentioned authentication plugins, the respective plugin file should be copied as **hvrvalidpw** in **HVR_HOME/lib/** directory.

LDAP Authentication - Hvrvalidpwldap Plugin

HVR authenticates the incoming username/password by invoking its **hvrvalidpwldap** plugin.This plugin authenticates a user by validating the credentials stored on LDAP server. This authentication is achieved by using the command file **hvrvalidpwldap** available in **HVR_HOME/lib** directory.
This plugin connects to the LDAP server with a search username and password. For Active Directory, it can connect using NTLM authentication. The search connection should have privileges to perform search operations. After establishing a connection with search user, an LDAP search is performed to validate the HVR user. User groups of the validated user also can be fetched from the LDAP server; these groups can be used inside the access control file.

> **hvrvalidpwldap** is not a command to be executed manually in the command line to authenticate a user; it is only a plugin which is invoked by HVR for LDAP based authentication.

Installing Python Environment

HVR requires the LDAP python client module installed for using the LDAP authentication. Perform the following on HVR hub machine:

1. Install Python (only 2.7.x version is supported). Skip this step if the mentioned python version is already installed in the machine.
2. Install the following python client module:

```
pip install ldap3
```

Enabling LDAP Authentication

To enable LDAP authentication:

1. Create file **HVR_HOME/lib/hvrvalidpwldap.conf** to supply the configuration required for connecting to the LDAP server. The configuration file parameters are described in section LDAP Configuration File. An example

configuration file **hvrvalidpwldap.conf_example** is available in **HVR_HOME/lib** directory.

2. HVR should use the username/password only for authentication, but must not change from the current operating system user to that login. To achieve this;

   - In Linux or Unix,

     systemd

     a. Set **user=** with a non-root operating system user.
     b. Update the **ExecStart** from **-r** to **-r -A** to prevent changing of user.

     xinetd

     a. Set **user=** with a non-root operating system user.
     b. Update the **server_args** from **-r** to **-r -A** to prevent changing of user.

     inetd

     a. Change the user from **root** to a non-root operating system user.
     b. Update **-r** in the command as **-r -A** to prevent changing of user.

     hvrremotelistener

     a. Execute **Hvrremotelistener** with option **-A** along with **-d** or **-i** options.
   - In Windows,

     a. Execute **Hvrremotelistener** with option **-A** along with **-ac** option in the command line. Option **-P** can also be used along with this command to create the service as non administrator operating system user.
3. Copy **HVR_HOME/lib/hvrvalidpwldap** to **HVR_HOME/lib/hvrvalidpw**.

> HVR only uses a plugin-based authentication system if it detects file **hvrvalidpw** in directory **HVR_HOME/lib**. This step activates **hvrvalidpwldap** plugin for user authentication.

LDAP Configuration File

This section lists and describes the parameters required for configuring the connection to the LDAP server in **hvrvalidpwldap.conf**.

Parameter	Description
LDAP_Server	The hostname or address of the LDAP server. Possible values are: • *hostname* • **ldap://***hostname* • **ldap://***hostname***:** *port* • **ldaps://***hostname* • **ldaps://***hostname***:** *port*

LDAP_Search_User	The credential to perform LDAP search. Possible values are: • **anonymous** : Do not bind to an LDAP user, perform search anonymously. • **self** : Bind to input HVR username/password. LDAP search will still be performed if the **LDAP_User_Method** is not set as **none**. • **self_ntlm** : Bind to input HVR username/password using the Active Directory NTLM authentication. LDAP search will still be performed if the **LDAP_User_Method** is not set as **none**. **Since** v5.6.5/1 • **user:***username* : Bind to LDAP user or distinguished name (DN) *username*. Requires **LDAP_Search_Password**. • **ntlm:***username* : Authenticate using the Active Directory NTLM. The format for *username* is *domain**user*. Requires **LDAP_Search_Password**.
LDAP_Search_Password	The password of the **LDAP_Search_User**.
LDAP_User_Method	The method to find (search) LDAP users. Possible values are: • **none** : To disable searching for the users from LDAP server. This can be used only if **LDAP_Search_User** is **self** or **self_ntlm**. Note that user groups cannot be searched in this case, so **LDAP_Group_Method** must also be set as **none**. **Since** v5.6.5/1 • *search_type***/***base_dn***/***filter* : A separate LDAP search is performed. Here, *search_type* is either **search_one** or **search_subtree**; *base_dn* is the starting point of search in LDAP tree; *filter* is an LDAP filter. Separator **/** can be replaced with any other non-alphanumeric character. In the example below, **%u** is replaced with the actual username.
LDAP_Group_Method	The method to find (search) LDAP user groups. Possible values are: • **none** : To disable searching for the user groups from LDAP server. • **user_attribute/***attr_name* : *attr_name* is an attribute of previously found user. • *search_type***/***base_dn***/***filter***/***attr_name* : A separate LDAP search is performed. Here, *search_type* is either **search_one** or **search_subtree**; *base_dn* is the starting point of search in LDAP tree; *filter* is an LDAP filter; *attr_name* is an attribute of found group entities to use as group name. Separator **/** can be replaced with any other non-alphanumeric character. In the example below, **%U** is replaced with found user's distinguished name (DN).
LDAP_Timeout	Timeout (in seconds) for the LDAP connections and queries.

LDAP_Error_Trace **Since** v5.6.5/1	Show/hide detailed information (to help diagnose configuration problems or error messages) in case of authentication failure. Possible values are: **0** : To disable error tracing. **1** : To enable error tracing. Enabling error tracing will allow unauthorized users to see details of authentication failure.

Examples

For basic Active Directory setup,

```
LDAP_Server=localhost

LDAP_Search_User=self_ntlm

LDAP_User_Method=none

LDAP_Group_Method=none

LDAP_Timeout=10
```

For basic LDAP setup,

```
LDAP_Server=localhost

LDAP_Search_User=self

LDAP_User_Method=none

LDAP_Group_Method=none

LDAP_Timeout=10
```

For basic setup with a dedicated lowest-privileged search user,

```
LDAP_Server=localhost

LDAP_Search_User=user:CN=SearchUser,CN=Users,DC=organization,DC=local

LDAP_Search_Password=password

LDAP_User_Method=search_subtree/CN=Users,DC=organization,DC=local/(&(objectClass=person)(|
(cn=%u)(sAMAccountName=%u)(uid=%u)))

LDAP_Group_Method=search_subtree/DC=organization,DC=local/(&(objectClass=group)(member=%
U))/CN

LDAP_Timeout=10
```

Specify the **LDAP_User_Method** and **LDAP_Group_Method** that is appropriate for your LDAP setup.

Files

HVR_HOME	
lib	
hvrvalidpwldap	The plugin file for LDAP authentication. To use authentication through LDAP this file should be copied to **hvrvalidpw**.
hvrvalidpwldap.conf	Configuration for this plugin.
hvrvalidpwldap.conf_example	Example configuration file for this plugin.
hvrvalidpw	Used by HVR for user authentication. For LDAP authentication, this should be a copy of **hvrvalidpwldap**.

Private Password File Authentication - Hvrvalidpwfile Plugin

HVR authenticates incoming username/password by invoking its **hvrvalidpwfile** plugin. This plugin authenticates a user by validating the credentials stored in a private password file. This authentication is achieved by using the command file **hvrvalidpwfile** available in **HVR_HOME/lib** directory.
For authentication, this plugin is invoked by HVR without any arguments and supplies the login and password (space separated) on the standard input.

Enabling Private Password File Authentication

To enable Private Password File Authentication:

1. Create user:
    a. Execute command **hvrvalidpwfile** *username*. For example,

    ```
    perl hvrvalidpwfile user1
    ```

    b. Enter password at the prompt and press **Enter** key.
    c. Repeat steps a and b to create multiple users.

    The username and password are stored in **HVR_HOME/lib/hvrpasswd**. For more information, see Managing Usernames and Passwords.

2. HVR should use the username/password only for authentication, but must not change from the current operating system user to that login. To achieve this;
    - In Linux or Unix,

    systemd

    a. Set **user=** with a non-root operating system user.
    b. Update the **ExecStart** from **-r** to **-r -A** to prevent changing of user.

    xinetd

    a. Set **user=** with a non-root operating system user.
    b. Update the **server_args** from **-r** to **-r -A** to prevent changing of user.

    inetd

a. Change the user from **root** to a non-root operating system user.
b. Update **-r** in the command as **-r -A** to prevent changing of user.

hvrremotelistener

a. Execute **Hvrremotelistener** with option **-A** along with **-d** or **-i** options.

- In Windows,

a. Execute **Hvrremotelistener** with option **-A** along with **-ac** option in the command line. Option **-P** can also be used along with this command to create the service as non administrator operating system user.

3. Copy **HVR_HOME/lib/hvrvalidpwfile** to **HVR_HOME/lib/hvrvalidpw**.

> HVR only uses a plugin-based authentication system if it detects file **hvrvalidpw** in directory **HVR_HOME/lib** . This step activates **hvrvalidpwfile** plugin for user authentication.

Managing Usernames and Passwords

The command **hvrvalidpwfile** allows you to manage usernames and passwords. The password is always encrypted and stored in the custom password file **hvrpasswd** available in **HVR_HOME/lib** directory.

- To create new user or update the password of an existing user:
hvrvalidpwfile *username*

> This command prompts to enter password. The password entered in this command is saved for the respective *username*.

- To create new user or update the password of an existing user without displaying prompt to enter password:
hvrvalidpwfile -b *username password*

- To delete an existing user:
hvrvalidpwfile -D *username*

Files

HVR_HOME

lib

hvrvalidpwfile	The plugin file for private password file authentication. This file should be copied to **hvrvalidpw**.
hvrpasswd	Used by **hvrvalidpwfile** for storing the username and password.
hvrvalidpw	Used by HVR for user authentication. For local password file authentication, this should be a copy of **hvrvalidpwfile**.

Custom Hvrvalidpw Authentication

HVR also allows you to supply your own **hvrvalidpw** authentication plugin. This plugin can be a modified version of **hvrvalidpwfile** plugin or else you can create your own plugin. The custom plugin file should be named **hvrvalidpw** and saved in **HVR_HOME/lib** directory. It should obey the following calling conventions:

- It should read a line of input which will contain the username and password.
- It should exit with code 0 if the username and password is valid. Otherwise, it should exit with code 1.

Hvrwalletconfig

Since v5.6.5/5

Contents

Name

hvrwalletconfig - Configure HVR hub wallet.

Synopsis

hvrwalletconfig -*options hubdb* [*properties*]

Description

Command **hvrwalletconfig** configures the hub encryption wallet.

This command is used to enable/disable the hub wallet, set wallet password, auto open hub wallet, rotate the hub wallet encryption key, change wallet password, and delete hub wallet.

The first argument *hubdb* specifies the connection to the hub database. For more information about supported hub databases and the syntax for using this argument, see Calling HVR on the Command Line.

The second argument *properties* specifies the properties that define the hub wallet type and configuration. For more information, see section Properties.

Options

This section describes the options available for command **hvrwalletconfig**.

-d*arg*	Delete wallet. Valid values for *arg* are:
	
	• **a**: Delete wallet but retain the artifacts (encryption key sequence and key history). This requires option **-p** and **Encryption** to be set to **NONE**.
	• **A**: Delete wallet and artifacts. This requires option **-p** and **Encryption** to be set to **NONE**.
	• **f**: Force wallet deletion even if the wallet is in use. This option can only be used in combination with the above options :
	  • **af** : Delete wallet but retain the artifacts such as historical keys (the wallet will be removed even if the encryption is not disabled). This can be used for example if wallet password is lost. Keeping artifacts requires access to the wallet (must be open and accessible). Historical keys will be lost. If **Encryption=SECRETS_ONLY** is not set before, encrypted passwords in a hub database will remain if wallet is not open or accessible. These passwords need to be manually fixed by a user by re-entering the passwords in the **HVR GUI** and saving them.
	  • **Af** : Delete wallet, remove artifacts such as historical keys (the wallet will be removed even if the encryption is not disabled). This can be used if the wallet password is lost. If **Encryption=SECRETS_ONLY** is not set before, encrypted passwords in a hub database will remain if wallet is not open or accessible. These passwords need to be manually fixed by a user by re-entering the passwords in the **HVR GUI** and saving them.
	
	Retaining artifacts is good to handle the transition, so that service passwords, jobs mentioning encrypted password, etc continue to work as normal. However, when the wallet is deleted, those artifacts are not protected anymore (they were protected with wallet), so the historical keys become unprotected. This might compromise your previously encrypted values.

-h*class*	Location *class* of the hub database. Valid values for *class* are **db2**, **db2i**, **ingres**, **mysql**, **oracle**, **postgresql**, **sqlserver**, or **teradata**. For more information, see Calling HVR on the Command Line.
-m	Migrate a hub wallet to different storage instead of modifying its configuration in place. Wallet migration moves the encryption key from one wallet configuration file to another. The encryption key does not change, but its encrypted storage is first decrypted by the old wallet and then encrypted by a new wallet. For more information, see section Hub Wallet Migration in Hub Wallet and Encryption. In software wallet, this option is used to get a new password to change a wallet password to a new password. This option is mandatory when changing the wallet password (e.g. it protects against unintended password changes when setting up auto-open password option). A new password must be provided using option **-p**. The old password must be available either via auto-open password feature, or wallet must be opened using **hvrwalletopen** (through a running **HVR Scheduler**). In KMS wallet, this option is used to migrate a hub wallet from a previous KMS account/settings to new KMS account/settings or a user switches to a non-KMS wallet. This option is mandatory when migrating to another KMS wallet.
-p	Ask for a password of the hub wallet after command **hvrwalletconfig** is run. The following operations require providing the existing or a new password: • Operations that can lock the user out (such as removing **Wallet_Auto_Open_Password**) require the existing password. • Operations that install a new wallet, migrate a wallet to another device (to a different **Wallet_Type** or to the same **Wallet_Type** with a different account) require a new password.

-P	Enable automatic wallet open feature. This option saves the provided password into the **Wallet_Auto_Open_Password** property. This requires option **-p**. For more information about wallet auto-open, see section Methods to Supply Wallet Password in Hub Wallet and Encryption.
-r	Rotate (retire and regenerate) the encryption key. This option creates a new encryption key, encrypts it, and stores it in the wallet. The previous encryption key is moved to the history (encrypted with the new key) for the cases when HVR needs it to decrypt data encrypted with it. Then HVR decrypts the hub catalogs with the old key and re-encrypts them with the new key. During this key rotation process, both the old and new keys are available in the history. Historical keys are kept in the wallet configuration file each encrypted with the latest key. TX/Log files do not undergo key rotation. Instead, the old key is left in the history protected by the latest key. Existing password (**-p**) of the hub wallet is required if the wallet is not already open by the **HVR Scheduler** and if the **Wallet_Auto_Open_Password** property is not set. This option can be used alone or with other options that change the **Wallet_*** properties. It cannot be combined with the other options such as getting wallet configuration or removing historical keys.
-S*sequence*	Delete historical keys older than sequence number *sequence*. This option cannot be combined with others.

-T*tstamp*	Delete historical keys rotated before timestamp *tstamp*.
	
	This option cannot be combined with others.
	
	Valid values for *tstamp* can be an absolute timestamp or as a relative timestamp using seconds. Following are examples:
	
	`hvrwalletconfig -T 2019-11-26T10:54:59Z myhubuser/myhubpassword`
	
	The following example will remove keys rotated older than the last 86400 seconds (or 24 hours).
	
	`hvrwalletconfig -T now-86400 myhubuser/myhubpassword`

-u*user*[*/pwd*]	A hub database *user* name. For some databases (e.g. SQL Server) a password must also be supplied. For more information, see Calling HVR on the Command Line.

Properties

This section describes the properties that can be defined in the hub wallet configuration file.

Property	Description

Encryption	The category of data that should be encrypted using the hub wallet. Valid values are (case-sensitive): • **NONE** (**default**) - turns off the encryption. When setting up the hub wallet encryption without specifying **Encryption=SECRETS_ONLY** or **Encryption=ALL_CONFIDENTIAL**, then it remains as **Encrption=NONE**, and the previous behaviour remains. Also, to remove the hub wallet (without force), you need to set **Encryption=NONE** first. • **SECRETS_ONLY** - includes secret keys and passwords used for accessing/connecting to a database. For more information, refer to section Classification of Data on page Hub Wallet and Encryption. • **ALL_CONFIDENTIAL** - includes values in a user table and key-values exposed in the error message.
Wallet_Type	Type of the hub wallet. Valid values are (case-sensitive): • **SOFTWARE** is a file that stores the hub encryption key. • **KMS** is a network service (KMS) that encrypts the hub encryption key. For a detailed description on the wallet types, refer to section Hub Wallet Types on page Hub Wallet and Encryption.
Wallet_Auto_Open_Plugin	A user-supplied plugin that runs command **hvrwalletopen**. The **HVR Scheduler** can execute this plugin to obtain the wallet password. For example: */home/user/myplugin*.sh
Wallet_Auto_Open_Password	Remove a wallet auto-open password. This property is used only to disable the auto-open hub wallet feature. It does not accept any value. Just set it to blank for removing the auto-open password. For example: **Wallet_Auto_Open_Password=** For security reasons, "**Wallet_Auto_Open_Password=**" will work to unset the password, but "**Wallet_Auto_Open_Password**=*mypassword*" will not work. This is the only way to set it. For more information, refer to section Auto-Open Hub Wallet on page Configuring and Managing Hub Wallet.
Wallet_KMS_Region **KMS Wallet**	KMS region where the KMS server is located. For example: **Wallet_KMS_Region**=*eu-west-1* For more information, refer to section Creating and Enabling a KMS Wallet on page Configuring and Managing Hub Wallet.
Wallet_KMS_Access_Key_Id **KMS Wallet**	KMS access key ID of the AWS user to access KMS. The corresponding AWS Secret Access Key should be used as a password of the HVR hub wallet. For example: **Wallet_KMS_Access_Key_Id**=*AKIAJDRSJY123QWERTY* This property cannot be used with **Wallet_KMS_IAM_Role** For more information, refer to section Creating and Enabling a KMS Wallet on page Configuring and Managing Hub Wallet.

Wallet_KMS_Customer_Master_Key_Id **KMS Wallet**	Customer Master Key (CMK) ID that uniquely identifies CMK within your KMS region. CMK is used for encryption and decryption of the hub encryption key. For more information, refer to the AWS Documentation. For example: **Wallet_KMS_Customer_Master_Key_Id**=*1234abcd-12ab-1234590ab* For more information, refer to section Creating and Enabling a KMS Wallet on page Configuring and Managing Hub Wallet.
Wallet_KMS_IAM_Role **KMS Wallet**	KMS IAM role. This defines how to retrieve Access Key ID/Secret Access Key from an EC2 node. Using an IAM role does not require a wallet password. HVR fetches AWS credentials from the EC2 instance HVR hub is running on. This property cannot be used with **Wallet_KMS_Access_Key_Id**. For more information, refer to section Creating and Enabling a KMS Wallet on page Configuring and Managing Hub Wallet.
Encryption_Key_Filename **Software Wallet**	The name of the software wallet file (.p12) that stores the hub encryption key. The hub wallet file is a password-encrypted (using the PKCS#12 standard) file which is supplied by a user when creating the software wallet. For example: **hvrwallet-5e9f3869.p12** This property is automatically defined by HVR and cannot be manually configured by a user. For more information, refer to section Creating and Enabling a Software Wallet on page Configuring and Managing Hub Wallet.
Encryption_Key_Encrypted **KMS Wallet**	This defines the hub encryption key encrypted using the KMS wallet and stored encrypted in the HVR wallet configuration file. This property is automatically defined by HVR and cannot be manually configured by a user.
Encyption_Key_Sequence	Defines a unique sequence number of the hub encryption key. Every hub encryption key has a unique sequence number. At the same time, e ach encrypted secret contains its hub encryption key's sequence number. This sequence number is used to easily find the correct encryption key for the encrypted secret. This property is automatically defined by HVR and cannot be manually configured by a user.

Encyption_Key_History	Defines a history file that holds the historical record of old hub encryption keys (encrypted with the latest hub encryption key) in case they are needed for decrypting data encrypted with the old encryption keys. This property is automatically defined by HVR and cannot be manually configured by a user. For more information, refer to section History on page Hub Wallet and Encryption.

Hvrwalletopen

Since v5.6.5/5

Contents

Name

hvrwalletopen - Open, close a hub encryption wallet and verify the wallet password.

Synopsis

hvrwalletopen -*options hubdb*

Description

Command **hvrwalletopen** opens or closes a hub encryption wallet and verifies the wallet password. If *options* are not supplied, **hvrwalletopen** opens the wallet (only if the **HVR Scheduler** is running) by providing the password to the jobs running under the **HVR Scheduler**.

The first argument *hubdb* specifies the connection to the hub database. For more information about supported hub databases and the syntax for using this argument, see Calling HVR on the Command Line.

Command **hvrwalletopen** can be executed either by a user or a plugin:

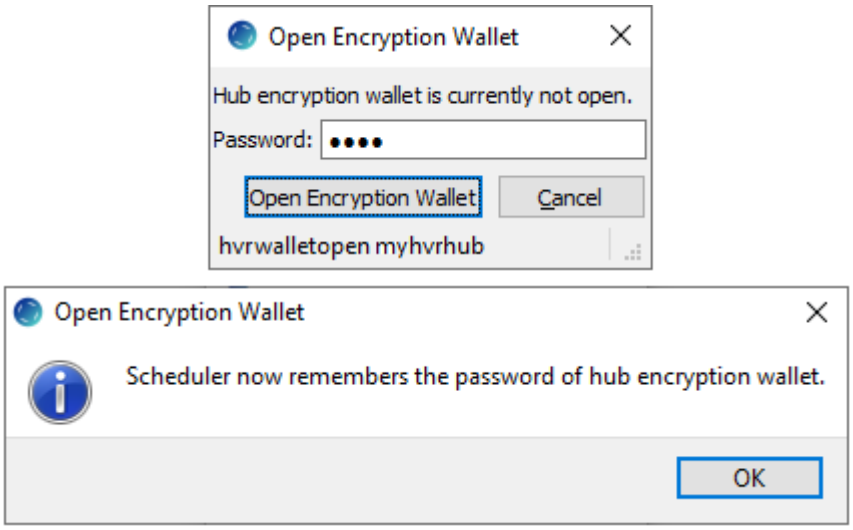
- If a user is running the command, then argument *hubdb* is mandatory (e.g. myhub).
- If a plugin is running the command, then argument *hubdb* is optional. When HVR is running a plugin defined in **Wallet_Auto_Open_Plugin**, it is sufficient for the plugin to only execute **$HVR_HOME/bin/hvrwalletopen** without any options or arguments.

  An example plugin:

  ```
  #!/bin/sh
  echo mywalletpassword | $HVR_HOME/bin/hvrwalletopen
  ```

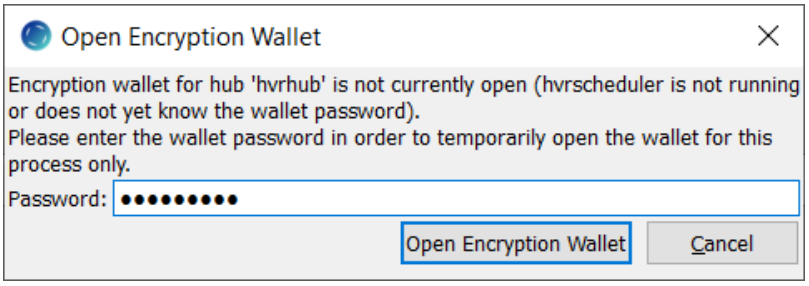
  In this case, command **hvrwalletopen** automatically picks up the hub name and optionally value for **-p***port*.

Command **hvrwalletopen** can also be executed in **HVRGUI** to open the wallet. However, the command *options* cannot be supplied in **HVRGUI**. In the navigation tree pane, right-click *hubname* **Open Encryption Wallet**.



Options

This section describes the options available for the command **hvrwalletopen**.



-c	Close the hub wallet by removing the wallet password from the **HVR Scheduler** memory. The hub wallet can be closed only if **HVR Scheduler** is running. Closing a hub wallet does not affect any running jobs which already have acce ss to the hub encryption key.
-h*class*	Location *class* of the hub database. Valid values for *class* are **db2**, **db2i**, **ingres**, **mys ql**, **oracle**, **postgresql**, **sqlserver**, or **teradata**. For more information, see Calling HVR on the Command Line.
-o	Check if the wallet is open (only if the **HVR Scheduler** is running). This option checks if the **HVR Scheduler** knows the password or not. A message will notify if the wallet is not open or if the wallet does not require setting a password via **hvrwalletopen**.
-p*port*	Port number to connect to the HVR hub. If an HVR process except the **HVR Scheduler** (such as a command line process, a job, HVRGUI hub side processes) needs to execute **hvrwalletopen**, then it requires the hub port number for **hvrwalletop en** to find that HVR process listening on that port. In this case, HVR prompts a message, e.g: "Run this command: hvrwalletopen -p 1234".
-v	Verify given password. This option does not require the **HVR Scheduler**.

HVR Insights

Since v5.3.1/25

HVR **Insights** is a real-time, web-based graphical interface that provides a comprehensive visualization of replication (capture and integrate) along with dashboard and event log viewer. It includes the following interfaces:

Topology

Visualization of capture and integrate activity which includes the volum6.64nIich eawhiloclizatire ar.

- **Show URL only** - A prompt is displayed with an option to copy the **Insights** URL, which can be then pasted into any web browser's address bar to view the respective **Insights** interface. This option is used in any of the following situations:
  - If the machine on which the HVR GUI is executed does not have a web browser installed.
  - To share the URL with all other users. All other users can access the **Insights** interface even if they are not connected to the machine on which the HVR GUI is executed.

For **Insights** to work, HVR GUI should be running. If HVR GUI is closed with the **Insights** window open, then an error message is displayed in the **Insights** window.

Topology

Since v5.5.5/6

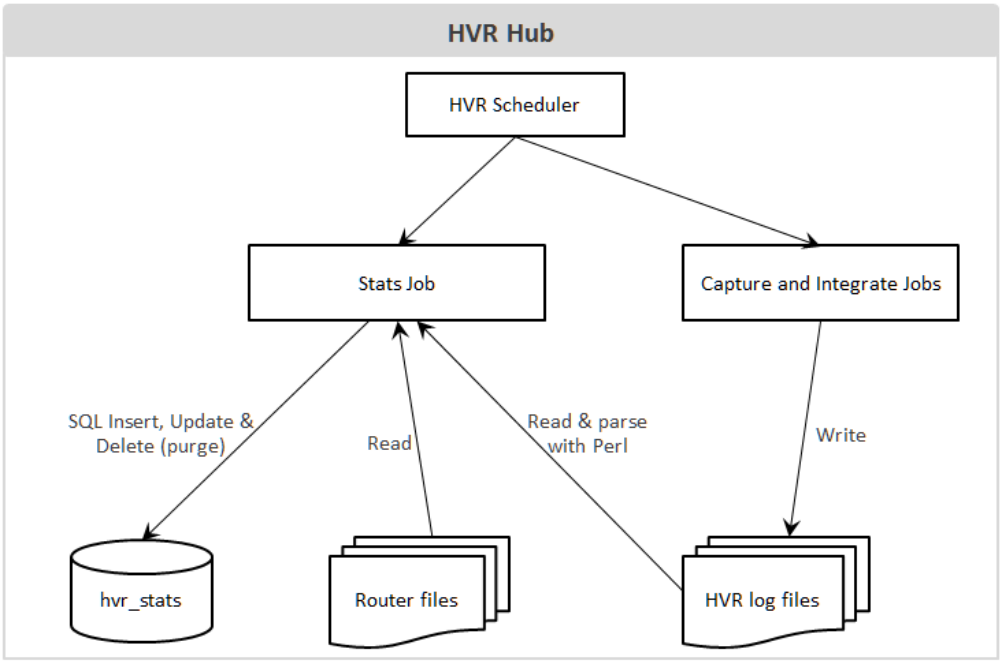
Contents

HVR's **Topology** view provides a real-time, web-based graphical visualization of how the source and target locations are connected using channels defined in HVR. It also graphically represents various elements in replication like source and target locations, volume of data in each locations, channels involved in replication, direction of replication, volume of replication, latency in replication, and problems like latency threshold exceeded or job failure etc. This helps to see at a glance where trouble lies in the replication: locations and channels displayed in the **Topology** view are color-coded based on the data volume and latency. Locations or channels alerting red can be immediately clicked into for diagnosis and troubleshooting.

In **Topology**, a specific location or channel can be selected to view the **statistics** related to that specific selection. For more information, see section Live Status Card.

Stats Job

The stats job (**hvrstats**) generates the information required for **Topology** and **Statistics** and saves it into the catalog table (**hvr_stats**) which is responsible for maintaining the statistics data. So, the stats job must be running to display live details in **Topology** or **Statistics**. The stats job is created along with the catalog tables during HVR's installation and automatically started when starting the **HVR Scheduler**.



To generate the information required for **Topology** and **Statistics**,

1. The stats job reads data from the HVR log files and the router transaction files.
2. The stats job then modifies (using **insert**, **update**, and **delete** SQL statements) the catalog table - **hvr_stats** based on the data read from the HVR log files and the router transaction files. It also aggregates the statistics data that are written in the **hvr_stats** table.

> The **hvr_stats** table consists of a number of columns that store statistical information about data replication. In particular, the **hvr_stats** table include the **metric_name** and **metric_value** columns storing data on a variety of metrics captured by HVR, such as capture/integrate latency, captured row counts, integrated change counts. For more information, see Metrics for Statistics.

Aggregation of Statistics Data

The stats job performs two types of aggregations (grouping) when writing statistics data to the **hvr_stats** table:

1. **Scope aggregation**

   The metrics information received from the HVR log files are written into the **hvr_stats** table based on the scope defined by a channel name (column **chn_name**), location name (column **loc_name**), and table name (column **tbl_name**), which can be either named explicitly or regarded as '*****' (which means applies to all channels, locations, tables).
   For example, if there are **5** 'captured inserts' with **chn_name**='*chn1*', **loc_name**='*src*' and **tbl_name**='*tbl1*' and **5** 'captured inserts' with **chn_name**='*chn1*', **loc_name**='*src*' and **tbl_name**='*tbl2*'. The **hvr_stats** table will store these values, but it will also store value **10** for **tbl_name**='*****', the sum of both values ('captured inserts').

   For more information on various scopes that can be defined, see **hvrstats** (option **-s**).
2. **Time granularity aggregation**

   Metrics are gathered/output with a per-minute granularity. For example, the value of 'captured inserts' for one-minute granularity means the number of rows inserted within that minute. These values can be aggregated up to 10 minutes, 1 hour and 1 day. For more information on the time granularity, see **hvrstats** (option **-T**).

Launching Topology from HVR GUI

Topology can only be launched from HVR GUI, and it can only be viewed within the **Insights** interface in a web browser.

Click here - Settings for Viewing Insights
Settings for Viewing Insights

The **Insights** interface for **Topology**, **Statistics**, and **Events** can be viewed only in the web browser.

Following are the two available options that can be configured (**View Insights Web App**) for viewing the **Insights** interface:

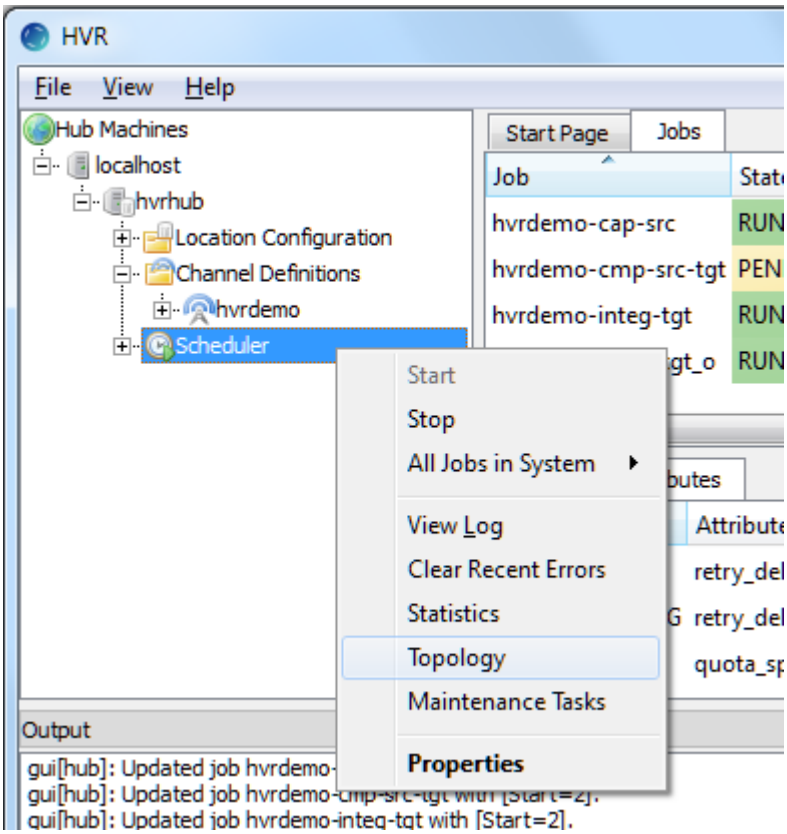
- **Open in local web browser** (default) - Automatically open the respective **Insights** interface in the default web browser.

> When this option is used and if the URL is shared with other users, only the users who are connected to the machine on which the HVR GUI is executed can access the **Insights** interface.

- **Show URL only** - A prompt is displayed with an option to copy the **Insights** URL, which can be then pasted into any web browser's address bar to view the respective **Insights** interface. This option is used in any of the following situations:
  - If the machine on which the HVR GUI is executed does not have a web browser installed.
  - To share the URL with all other users. All other users can access the **Insights** interface even if they are not connected to the machine on which the HVR GUI is executed.

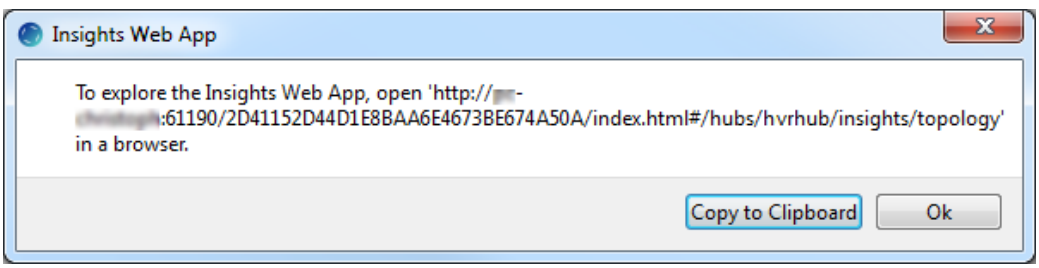
> For **Insights** to work, HVR GUI should be running. If HVR GUI is closed with the **Insights** window open, then an error message is displayed in the **Insights** window.

- To launch **Topology**, right-click **Scheduler** and select **Topology**.



- To launch **Topology** when the viewing option for **Insights Web App** is set as **Show URL only**, the following needs to be performed:

1. Right-click **Scheduler** and select **Topology**
2. Click **Copy to Clipboard**.



3. Paste the URL in the web browser's address bar and press **Enter**.

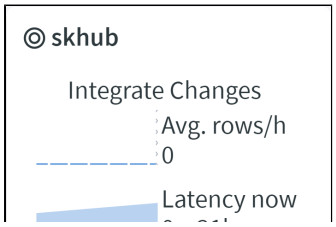
Topology View

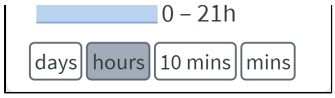
The **Topology** view contains the following user interface components:

1. **Hub** - name of the hub for which the topology is displayed.
2. **Scope** - highlights the selected location or channel in the topology. This is same as clicking on a location or channel.
3. **Topology** - graphical representation of locations, channel, direction of replication, volume of data in locations, volume of data being replicated, and latency in replication.
4. **Legend** - show/hide description for the colors and icons displayed in topology.
5. **Settings** - show/hide configuration for the topology view.
6. **Live status card** - displays the performance metrics for the respective element selected in topology. For more information, see Live Status Card.

Live Status Card

Live status card displays the live statistics metrics for the elements selected in topology. Live status card is supported for the following elements - Hub, Channel, Source Location, Capture Job, Integrate Job, and Integrate Location. By default, only the live status card for Hub is displayed in the **Topology** view. The statistics displayed in the live status card can be filtered based on the granularity (days/hours/10 mins/mins) selected. The default granularity is an **hours**.



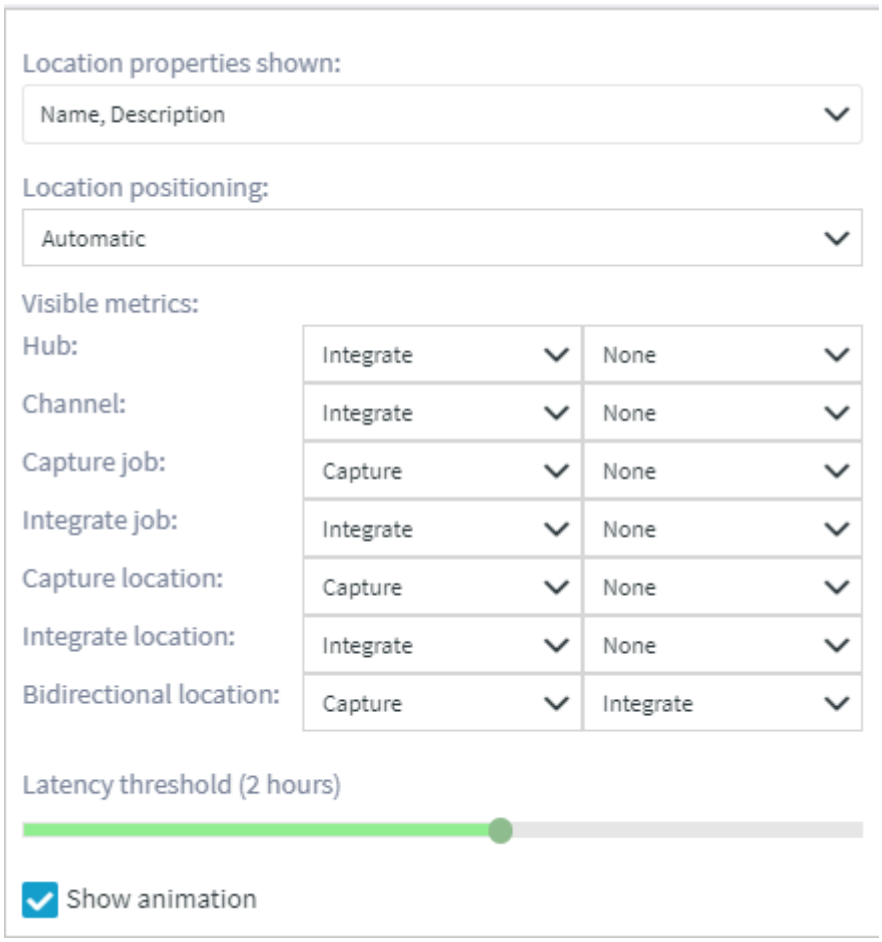


Only two metrics can be displayed at a time in live status card and this can be configured from **Settings** **Visible metrics**.

Clicking on the live status card opens **statistics** view for the respective element.

Settings

The settings screen allows you to configure/customize the topology view.



The options available in settings screen are:

Settings	Options
Location properties shown	Configure the location properties to be displayed next to a location icon. Only the option selected first is displayed and the remaining are only visible on mouse hover. Options available are: • **Name** (default): Displays the location name. • **Description** (default - visible on mouse hover): Displays the location description. • **Number of tables**: Displays the total number of tables available in the location. • **Class**: Displays the location class/type. • **Remote Node**: Displays the node name where the HVR Remote Listener for this location is running. **Node** name is defined in the **Connection** tab of the **New Location** screen.
Location positioning	Configure the visual layout of locations over left and right edges of the Topology area (source and target locations respectively). Options available are: • **Automatic** (default): 'Ungrouped' view. Source locations and target locations are spread over left and right edges of the Topology area respectively. • **Group by Class**: Source and target locations are grouped by location class. The location class name is displayed as the group name. • **Group by Remote Node**: Source locations and target locations are grouped by remote node. The remote node name is displayed as the group name.
Visible metrics	Configure the metrics to be displayed in live status card for the following elements of replication: **Hub** — Available options are -

		• **None** • **Capture**: Displays the captured changes graph for the hub. • **Integrate** (default): Displays the integrated changes graph for the hub.
	Channel	Available options are - • **None** • **Number of tables** : Displays the number of tables in the channel. • **Description**: Displays the channel description. • **Capture**: Displays the captured changes graph for the channel. • **Integrate** (default): Displays the integrated changes graph for the channel.
	Capture job	Available options are - • **None** • **Number of tables**: Displays the total number of tables captured by the capture job. • **Capture** (default): Displays the captured changes graph for the capture job.
	Integrate job	Available options are - • **None** • **Number of tables**: Displays the total number of tables captured by the integrate job. • **Integrate** (default): Displays the integrated changes graph for the integrate job.
	Capture location	Available options are - • **None** • **Number of tables**: Displays the total number of tables available in the capture location. • **Class**: Displays the capture location class/type. • **Remote Node**: Displays the node name where the HVR Remote Listener for this location is running. • **Description**: Displays the capture location description. • **Capture** (default): Displays the captured changes graph for the capture location.
	Integrate location	Available options are - • **None** • **Number of tables**: Displays the total number of tables available in the integrate location. • **Class**: Displays the integrate location class/type. • **Remote Node**: Displays the node name where the HVR Remote Listener for this location is running. • **Description**: Displays the integrate location description. • **Integrate** (default): Displays the integrated changes graph for the integrate location.
	Bidirectional location	This is applicable only for bidirectional locations. Available options are - • **None** • **Number of tables**: Displays the total number of tables available in the location. • **Class**: Displays the location class/type. • **Remote Node**: Displays the node name where the HVR Remote Listener for this location is running. • **Description**: Displays the location description. • **Capture** (default): Displays the captured changes graph for the location. • **Integrate** (default): Displays the integrated changes graph for the location.
Latency threshold	Slider to configure the threshold for latency.	
Show animation	Show/hide animation for data movement in channel.	

Icons in Topology

In the Topology view, elements involved in replication are represented by various icons.

Icon	Description
	Indicates HVR hub.
	Indicates channel.
	Indicates database location.
	Indicates file location.
	Indicates Kafka location.
	Indicates capture job.
	Indicates integrate job.
	Indicates the direction of replication. The color of this icon turns red if a job failed or the latency threshold is exceeded.

Statistics

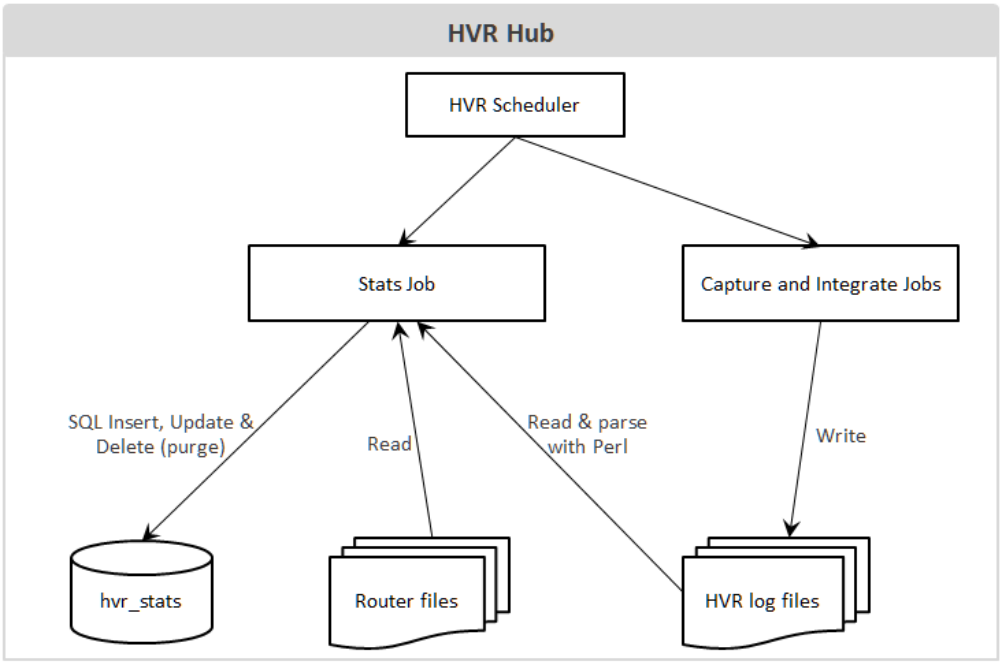
Since v5.31/25

Contents

Insight **Statistics** is a real-time, web-based dashboard which allows you to monitor replication performance. HVR's statistics dashboard helps you stay in control of your replication activity by giving you visibility into your most important data and metrics. It can display several types of statistical information such as number of rows replicated, time taken for capture and integrate, compression rates, and size of data processed on the replication system(s). The dashboard displays statistics for data updates very frequently, sometimes even on a minute-by-minute or second-by-second basis.

Stats Job

The stats job (**hvrstats**) generates the information required for **Topology** and **Statistics** and saves it into the catalog table (**hvr_stats**) which is responsible for maintaining the statistics data. So, the stats job must be running to display live details in **Topology** or **Statistics**. The stats job is created along with the catalog tables during HVR's installation and automatically started when starting the **HVR Scheduler**.



To generate the information required for **Topology** and **Statistics**,

1. The stats job reads data from the HVR log files and the router transaction files.
2. The stats job then modifies (using **insert**, **update**, and **delete** SQL statements) the catalog table - **hvr_stats** based on the data read from the HVR log files and the router transaction files. It also aggregates the statistics data that are written in the **hvr_stats** table.

> The **hvr_stats** table consists of a number of columns that store statistical information about data replication. In particular, the **hvr_stats** table include the **metric_name** and **metric_value** columns storing data on a variety of metrics captured by HVR, such as capture/integrate latency, captured row counts, integrated change counts. For more information, see Metrics for Statistics.

Aggregation of Statistics Data

The stats job performs two types of aggregations (grouping) when writing statistics data to the **hvr_stats** table:

1. **Scope aggregation**

   The metrics information received from the HVR log files are written into the **hvr_stats** table based on the scope defined by a channel name (column **chn_name**), location name (column **loc_name**), and table name (column **tbl_name**), which can be either named explicitly or regarded as '*****' (which means applies to all channels, locations, tables).
   For example, if there are **5** 'captured inserts' with **chn_name**='*chn1*', **loc_name**='*src*' and **tbl_name**='*tbl1*' and **5** 'captured inserts' with **chn_name**='*chn1*', **loc_name**='*src*' and **tbl_name**='*tbl2*'. The **hvr_stats** table will store these values, but it will also store value **10** for **tbl_name**='*****', the sum of both values ('captured inserts').

   For more information on various scopes that can be defined, see **hvrstats** (option **-s**).
2. **Time granularity aggregation**

   Metrics are gathered/output with a per-minute granularity. For example, the value of 'captured inserts' for one-minute granularity means the number of rows inserted within that minute. These values can be aggregated up to 10 minutes, 1 hour and 1 day. For more information on the time granularity, see **hvrstats** (option **-T**).

Statistics History Retention

By default, the old statistics data in the **hvr_stats** table is purged automatically to save the disk space needed for the hub database. The default statistics history retention size is **MEDIUM** (described in **Scheduling /StatsHistory**).

To change default statistics history retention size, define action **Scheduling /StatsHistory**.

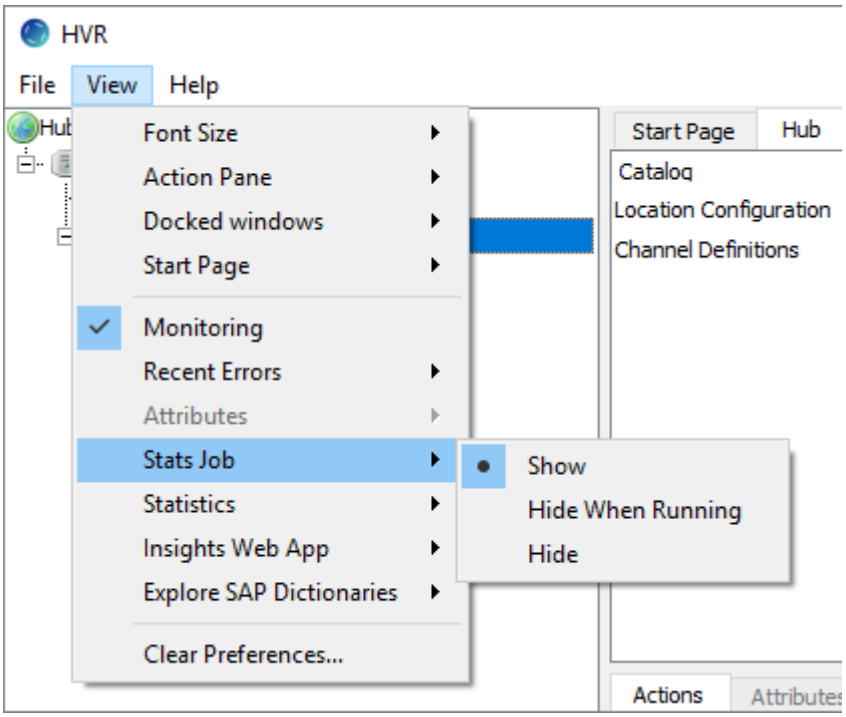
> To purge the statistics data immediately (as a one-time purge) from the **hvr_stats** table, use the command **hvrstats** (option **-p**).

Viewing Stats Job and Stat Job Log

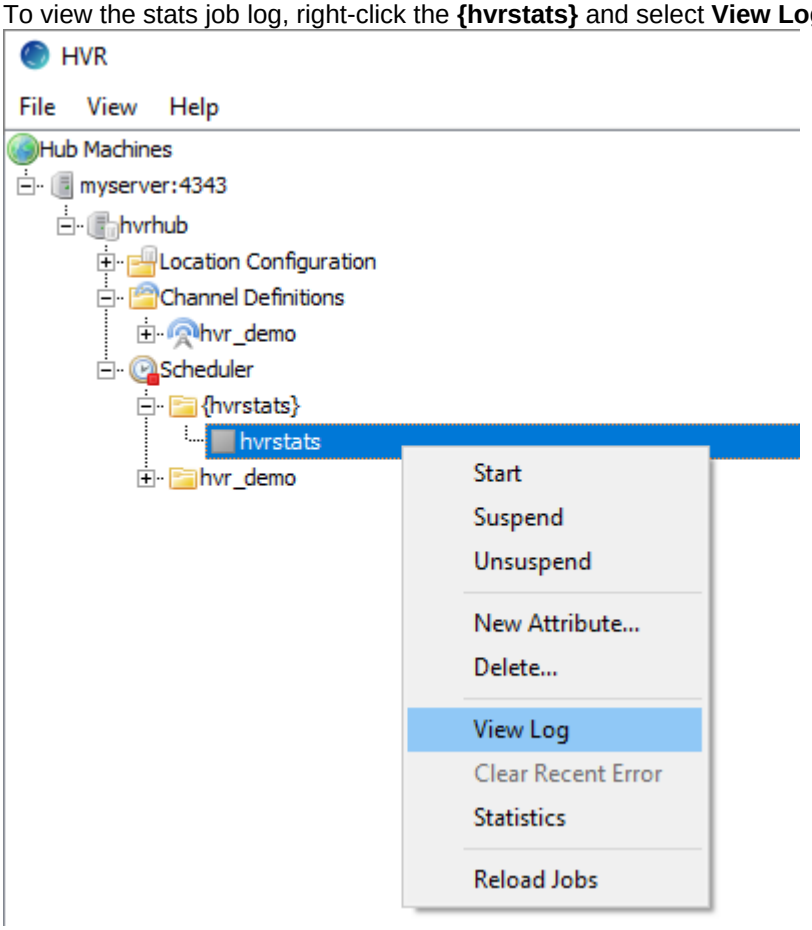
In HVR GUI, the stats job is displayed under the **Scheduler** node.

By default, if the stats job is running, it is not displayed under the **Scheduler** node. The display settings for the stats job can be configured from the menu bar.

- In the menu bar, click **View Stats JobShow**, to always display the stats job under the **Scheduler** node.



To view the stats job log, right-click the **{hvrstats}** and select **View Log**.



Launching Statistics from HVR GUI

Statistics can be launched only from HVR GUI, and it can only be viewed within the **Insights** interface in a web browser.

Click here - Settings for Viewing Insights
Settings for Viewing Insights

The **Insights** interface for **Topology**, **Statistics**, and **Events** can be viewed only in the web browser.

Following are the two available options that can be configured (**View Insights Web App**) for viewing the **Insights** interface:

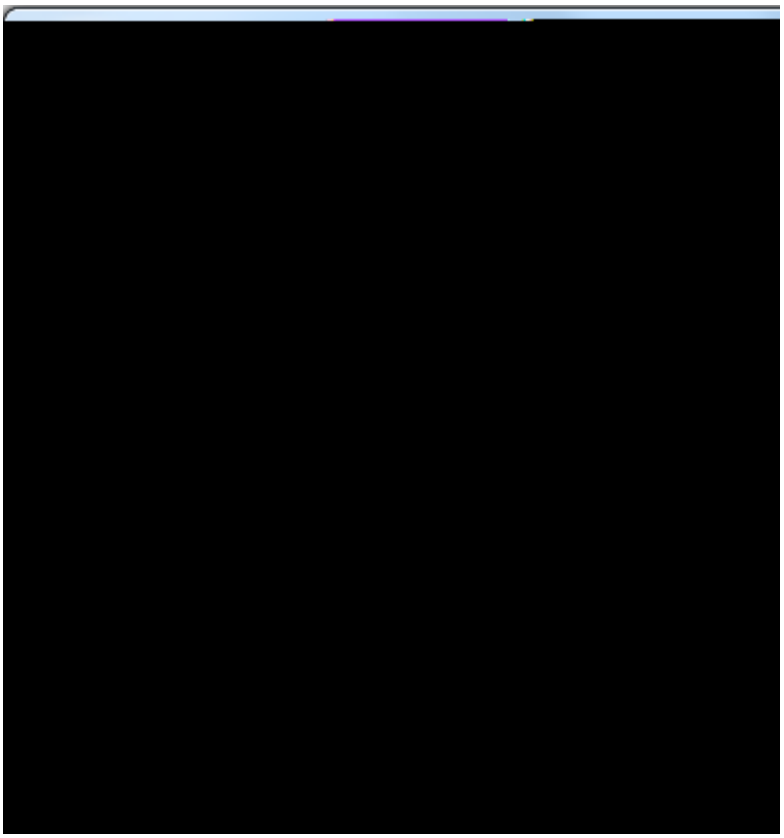
- **Open in local web browser** (default) - Automatically open the respective **Insights** interface in the default web browser.

  > When this option is used and if the URL is shared with other users, only the users who are connected to the machine on which the HVR GUI is executed can access the **Insights** interface.

- **Show URL only** - A prompt is displayed with an option to copy the **Insights** URL, which can be then pasted into any web browser's address bar to view the respective **Insights** interface. This option is used in any of the following situations:
  - If the machine on which the HVR GUI is executed does not have a web browser installed.
  - To share the URL with all other users. All other users can access the **Insights** interface even if they are not connected to the machine on which the HVR GUI is executed.

> For **Insights** to work, HVR GUI should be running. If HVR GUI is closed with the **Insights** window open, then an error message is displayed in the **Insights** window.

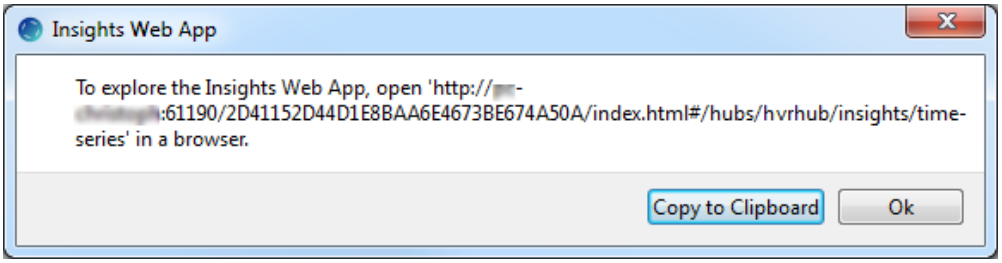
- To launch **Statistics**, right-click **Scheduler** an select **Statistics**.



- To launch **Statistics**, when the viewing option for **Insights Web App** is set to **Show URL only**, the following needs to be performed:

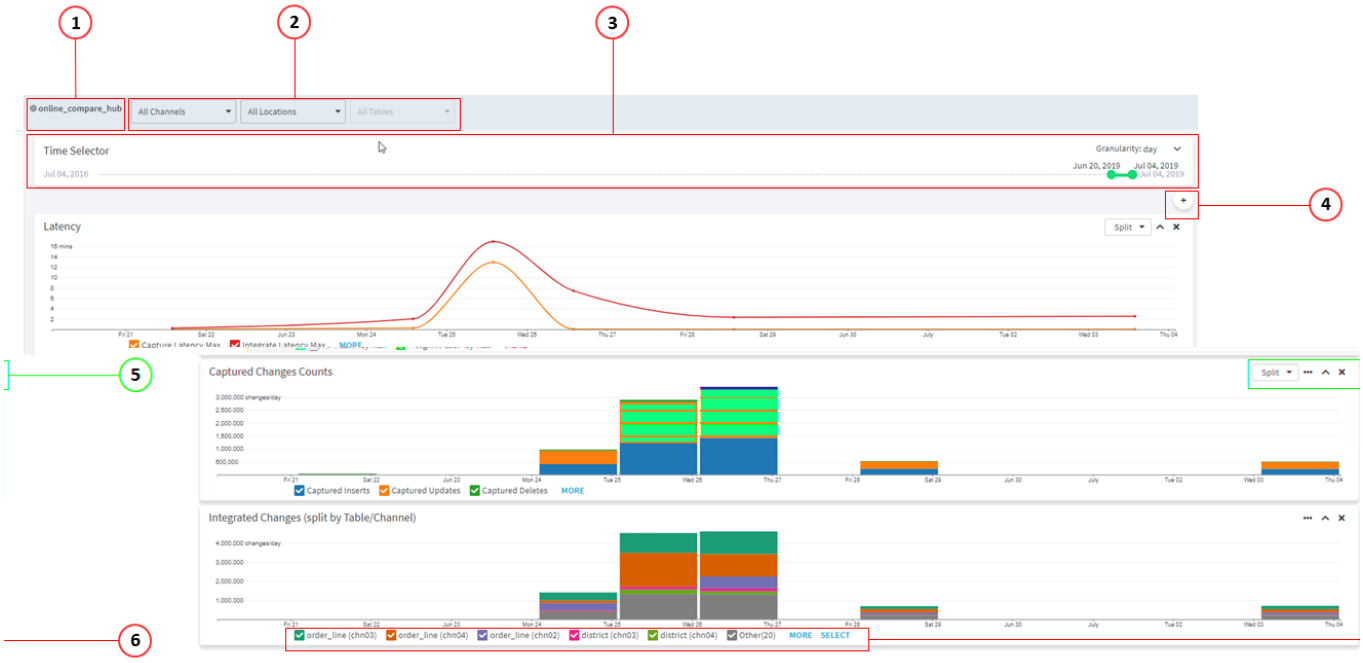
  1. Right-click **Scheduler** and select **Statistics**

2. Click **Copy to Clipboard**.



3. Paste the URL in the web browser's address bar and press **Enter**.

Statistics View



The **Statistics** view contains the following user interface components:

1. **Hub** - name of the hub for which the statistics is displayed.
2. **Scope** - filter the dashboard based on the selected location, channel, or table.
3. **Time Selector** - filter the dashboard based on the selected start date/time and end date/time. The other method to select time range is to zoom in on a custom time range on the graph. Click-and-drag the mouse across the chart to select a time range. The newly selected time range is applied to all graphs available in the dashboard. For example, to view statistics from 25th to 27th June, click-and-drag the mouse across the chart as shown below.

a. **Granularity** - Sets the aggregation of data for the dashboard. The granularity of time refers to the size in which data fields are sub-divided. The default granularity is an **hour**. However, this aggregation can be changed to a 'fine granularity' level like **1 minute** or to a 'coarse granularity' level like **1 day**. Changing the granularity is immediately reflected in the x-axis of all charts displayed in the dashboard. Finer granularity has overheads like excess computation time, memory, or other resources that are required to generate statistics dashboard. For this reason the data stored for each granular level is limited/purged at certain point of time - **Scheduling /StatsHistory**.

   The available options are -
   - **minute**: Indicates 1 minute granularity. The period displayed in the **Time Selector** is limited to the last 14 days.
   - **10 min**: Indicates 10 minute granularity. The period displayed in the **Time Selector** is limited to the last 30 days.
   - **hour**: Indicates 1 hour granularity. The period displayed in the **Time Selector** is limited to the last 6 months.
   - **day**: Indicates 1 day granularity. The period displayed in the **Time Selector** is limited to the last 3 years.

4. **Add Graph**- Add graph to the dashboard. For information about the available graphs in **Statistics**, see section Graphs.
5. Options to customize each graph.
   - **Split**: create a graph using the selected graph's **Metric** and **Scope** (Location, Channel, or Table and Channel). In **Split** drop-down, you can select a **Metric** available in the graph and split it by **Scope**. The split graph will be inserted next to the graph containing a metric you want to split.
   - **Rate unit**: change the Y-axis unit (per second, per minute, per hour, or per day). By default, the rate unit matches the **Granularity**: per minute for **minute** and **10 minute** granularity, per hour for **hour** granularity and per day for **day** granularity.
   - : Expand/collapse graph (when collapsed, only the graph header is displayed).
   - **X**: Remove graph from the dashboard.
6. **Legend/Metrics** - Indicates the metrics for graph. Each graph displays different metrics. For more information about metrics displayed in Statistics, see Metrics for Statistics.
   - **MORE**: displays more metrics in the graph.
   - **LESS**: displays less metrics.
   - **SELECT**: allows you to select particular metrics that needs to be displayed in a graph. This option is displayed only when a graph has more than 5 metrics associated with it.

Graphs in Statistics

The dashboard contains a set of graphs that indicate the state of different aspects of the replication. Following are the default graphs displayed in **Statistics**:

Graph Name	Description
Latency	Displays the latency information for capturing and integrating changes. Latency is the time (in seconds) taken for a transaction committed on the source system to be replicated (or committed) on the target system. This graph allows you to analyze the delay in data replication. The following Latency Stats Metrics are displayed in this graph: • **Capture Latency Min** • **Capture Latency Max** (default) • **Integrate Latency Min** • **Integrate Latency Max** (default) • **Capture Rewind Interval**

Captured Changes Counts	Displays the total number of changes captured. The following Captured Row Counts Stats Metrics are displayed in this graph: • **Captured Inserts** (default) • **Captured Updates** (default) • **Captured Deletes** (default) • **Captured Changes** • **Captured Changes Backdated** • **Captured Skipped Rows** • **Augmented Rows** • **SAP Augment Selects** • **Captured DDL Statements**
Integrated Changes (split by Table /Channel)	Displays the total number of changes integrated. This is basically a **Integrated Change Counts** graph which is split using the metric **Integrated Changes** and scope **Table/Channel**.

The following graphs can be added by clicking on the **Add Graph** button.

Graph Type	Description
Integrated Change Counts	Displays the total number of changes integrated. The following Integrated Change Counts Stats Metrics are displayed in this graph: • **Integrated Inserts** (default) • **Integrated Updates** (default) • **Integrated Deletes** (default) • **Integrated Changes** • **Integrated Skipped Changes** • **Changes Coalesced Away** • **Failed Inserts Saved** • **Failed Updates Saved** • **Failed Deletes Saved** • **Failed Changes Saved** • **Collision Changes Discarded** • **Empty Updates Discarded**
Transactions	Displays the total number of transactions. The following Transactions Stats Metrics are displayed in this graph: • **Captured Transactions** • **Captured Transactions Backdated** • **Integrated Transactions**
Durations	Displays the time taken for replication in seconds. The following Duration Stats Metrics are displayed in this graph: • **Capture Duration Total** • **Capture Duration Max** • **Capture Duration Average** • **Integrate Duration Total**

Speed	Displays the speed of replication. The unit of speed is indicated by the number of captured and integrated rows per selected granularity of time. The following Speed Stats Metrics are displayed in this graph: • **Capture Speed Max** (default) • **Capture Speed Average** • **Integrate Speed Max** (default) • **Integrate Speed Average** • **Integrate Burst Load Speed Max** • **Integrate Burst Load Speed Average** • **Integrate Burst Setwise Speed Max** • **Integrate Burst Setwise Speed Average**
Cycles	Displays the number of Capture and Integrate cycles. The following Cycles Stats Metrics are displayed in this graph: • **Capture Cycles** • **Integrate Cycles**
Byte I/O	Displays the size of captured row data and files in bytes before and after compression. The following Byte I/O Stats Metrics are displayed in this graph: • **Routed Bytes Written** • **Routed Bytes Written Uncompressed** • **Captured File Size** • **Capture DbmsLog Bytes** • **Capture DbmsLog Bytes Backdated**
Compression	Displays the compression ratio for row data. Captured row data is compressed when sent from the capture location to the hub and from the hub to the integrate location. The following Compression Stats Metrics are displayed in this graph: • **Compression Ratio Max** • **Compression Ratio Average**
Replicated Files	Displays the number of rows captured and integrated into file locations. The following Replicated Files Stats Metrics are displayed in this graph: • **Captured Files** • **Integrated Files** • **Failed Files Saved**
Errors/Warnings	Displays the total number of errors and warnings. The following Errors/Warnings Stats Metrics are displayed in this graph: • **Errors** • **^Errors** • **Errors F_J*** • **^Errors F_J*** • **Warnings** • **^Warnings** • **Warnings W_J*** • **^Warnings W_J***

Router Rows	Displays the total number of rows in the transaction files. The following Router Rows Stats Metrics are displayed in this graph: • **Capture Router Rows** • **Integrate Router Rows**
Router Bytes	Displays the total size of the transaction files in bytes. The following Router Bytes Stats Metrics are displayed in this graph: • **Capture Router Bytes** • **Integrate Router Bytes**
Router Files	Displays the total number of transaction files. The following Router Files Stats Metrics are displayed in this graph: • **Capture Router Files** • **Integrate Router Files**

Events

Since v5.5.5/6

Contents

HVR's event system is a part of the **HVR Insights** web-based application that maintains records for certain changes that a user makes in HVR. These records are called events, which are maintained in the catalog tables - **hvr_event** and **hvr_event_result**. The event system allows for accurate tracking of events and provides better visibility of real-time activities in HVR.

The two key purposes of using HVR's event-based system are:

1. Maintaining the audit trail for user activities that make certain changes in HVR. Records contain details that include event type, date/time, and user information associated with the event. This allows to monitor and analyze all user activities for establishing what events occurred and who caused them at a certain point of time.

   The following activities lead to the creation of events in HVR:
   - CLI: Executing commands like **hvrinit**, **hvrrefresh**, **hvrcompare**, **hvrswitchtable**, **hvrstart**, **hvrsuspend**, **hvrscheduler**, **hvrstats**, **hvradapt**, **hvrcontrol**, **hvrmaint**, and **hvrcatalogimport**.
   - GUI: Changing a channel definition or starting/stopping any jobs, etc.

   The following activities do not create events in HVR:
   - Automated occurrences inside HVR, e.g. activity of replication jobs. This is logged in job's log files already.
   - Read-only activities, such as opening/closing GUI, executing commands like **hvrrouterview** or **hvrcatalogexport**.
   - Running jobs directly from command-line. The output logging of such jobs is also lost.
   - Operations on remote HVR machine, such as starting **hvrremotelistener**.

2. Holding state for long-running operations (e.g. 'event-driven' compare) and storing the results of each compare and refresh operation. The event-driven compare operation is controlled by the state of an HVR event (**PENDING**, **RUNNING**, **DONE**, **FAILED**) in the HVR's event system. HVR creates a compare job in the **HVR Scheduler** and the compare operation is started under a compare event. When a compare is restarted it will continue from where it was interrupted, not start again from the beginning. While performing event-driven compare, if there is an existing compare event with the same job name in **PENDING** or **RUNNING** state then it is canceled (**FAILED**) by the new compare event.

Launching Events from HVR GUI

The **Events** page can be launched only from HVR GUI, and by default, and it can only be viewed within the **Insights** interface in a web browser.

Click here - Settings for Viewing Insights
Settings for Viewing Insights

The **Insights** interface for **Topology**, **Statistics**, and **Events** can be viewed only in the web browser.

Following are the two available options that can be configured (**View Insights Web App**) for viewing the **Insights** interface:

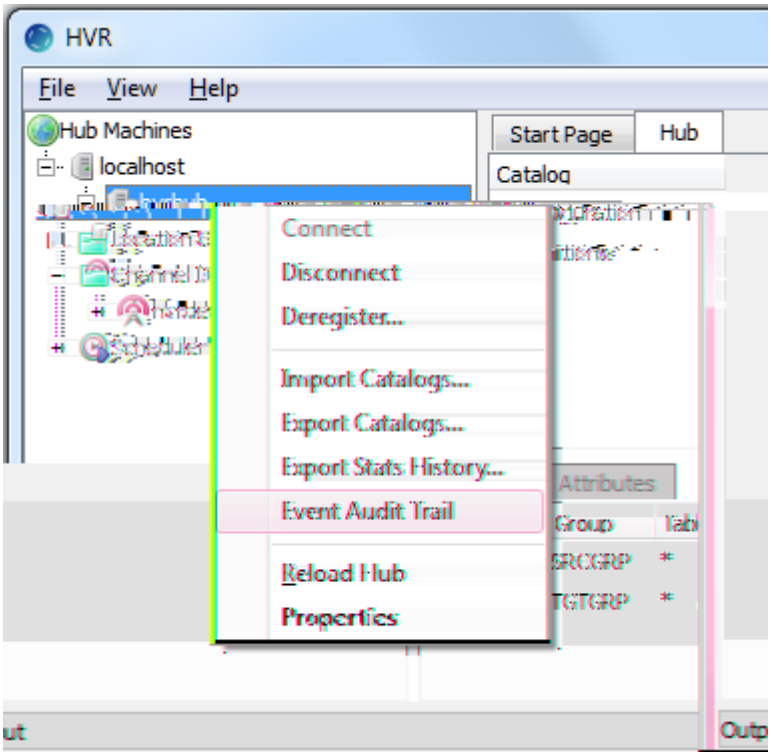
- **Open in local web browser** (default) - Automatically open the respective **Insights** interface in the default web browser.

  > When this option is used and if the URL is shared with other users, only the users who are connected to the machine on which the HVR GUI is executed can access the **Insights** interface.

- **Show URL only** - A prompt is displayed with an option to copy the **Insights** URL, which can be then pasted into any web browser's address bar to view the respective **Insights** interface. This option is used in any of the following situations:
  - If the machine on which the HVR GUI is executed does not have a web browser installed.
  - To share the URL with all other users. All other users can access the **Insights** interface even if they are not connected to the machine on which the HVR GUI is executed.

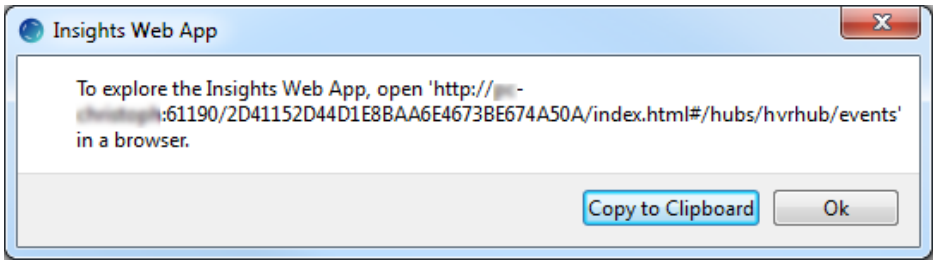
> For **Insights** to work, HVR GUI should be running. If HVR GUI is closed with the **Insights** window open, then an error message is displayed in the **Insights** window.

- To launch **Events**, right-click the hub name and select **Event Audit Trail**.



- To launch **Events** when the viewing option for **Insights Web App** is set to **Show URL only**, the following needs to be performed:

  1. Right-click the hub name and select **Event Audit Trial**.
  2. Click **Copy to Clipboard**.



3. Paste the URL in the web browser's address bar and press **Enter**.

Events Page

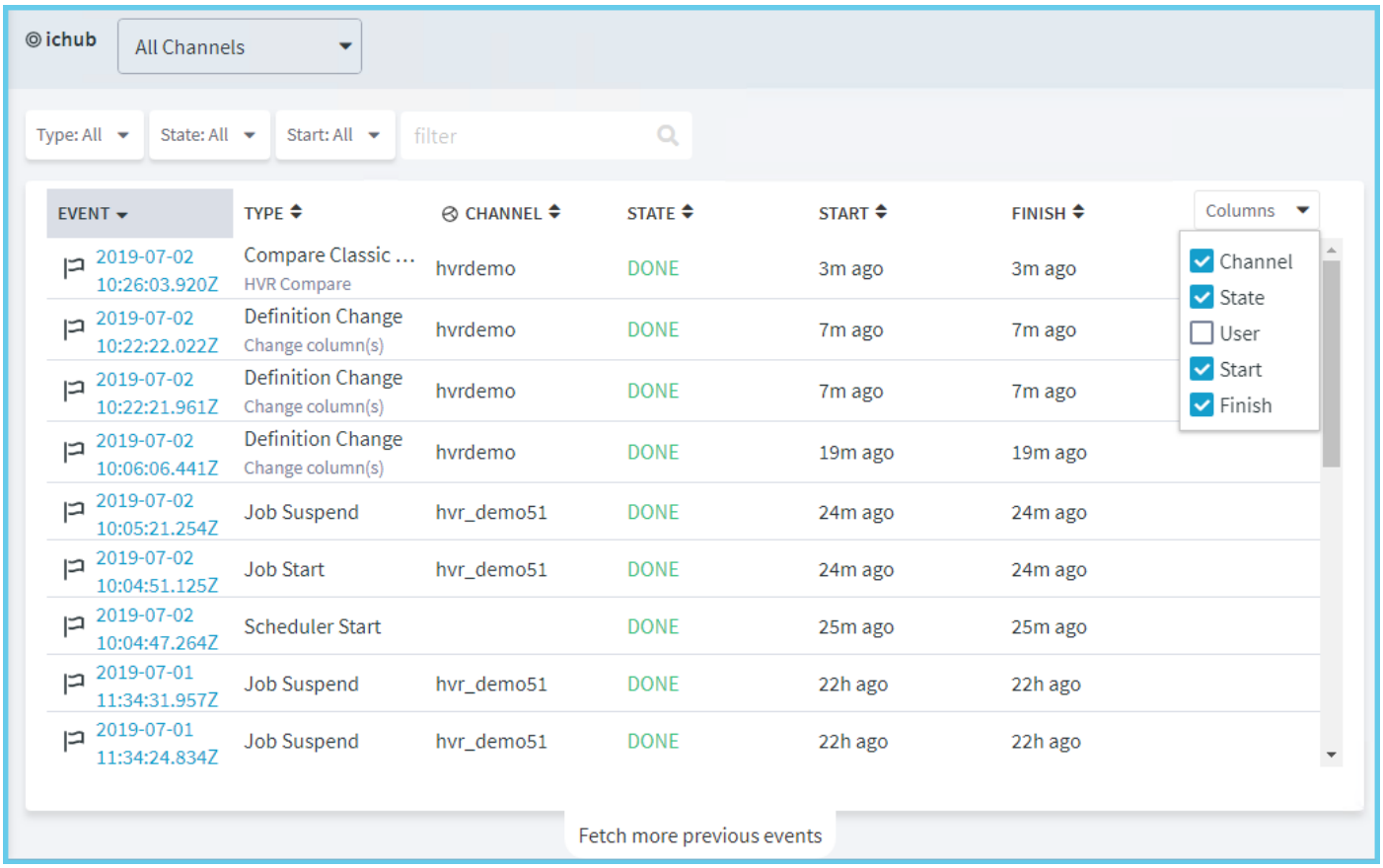
The **Events** page shows a list of all HVR events related to the hub or selected channel.

The following fields/columns are displayed in the **Events** page:

Field	Description
EVENT	Indicates the event ID.
TYPE	Indicates the type of event.
STATE	Indicates the status of event. • **PENDING** - indicates the event is yet to be executed. • **RUNNING** - indicates the event execution is in progress. • **DONE** - indicates the event execution is completed. • **FAILED** - indicates the event execution is canceled.
CHANNEL	Indicates the channel name associated with the event.
USER	Indicates the user name associated with the event.
START	Indicates the start time of the event.
FINISH	Indicates the finish time of the event.

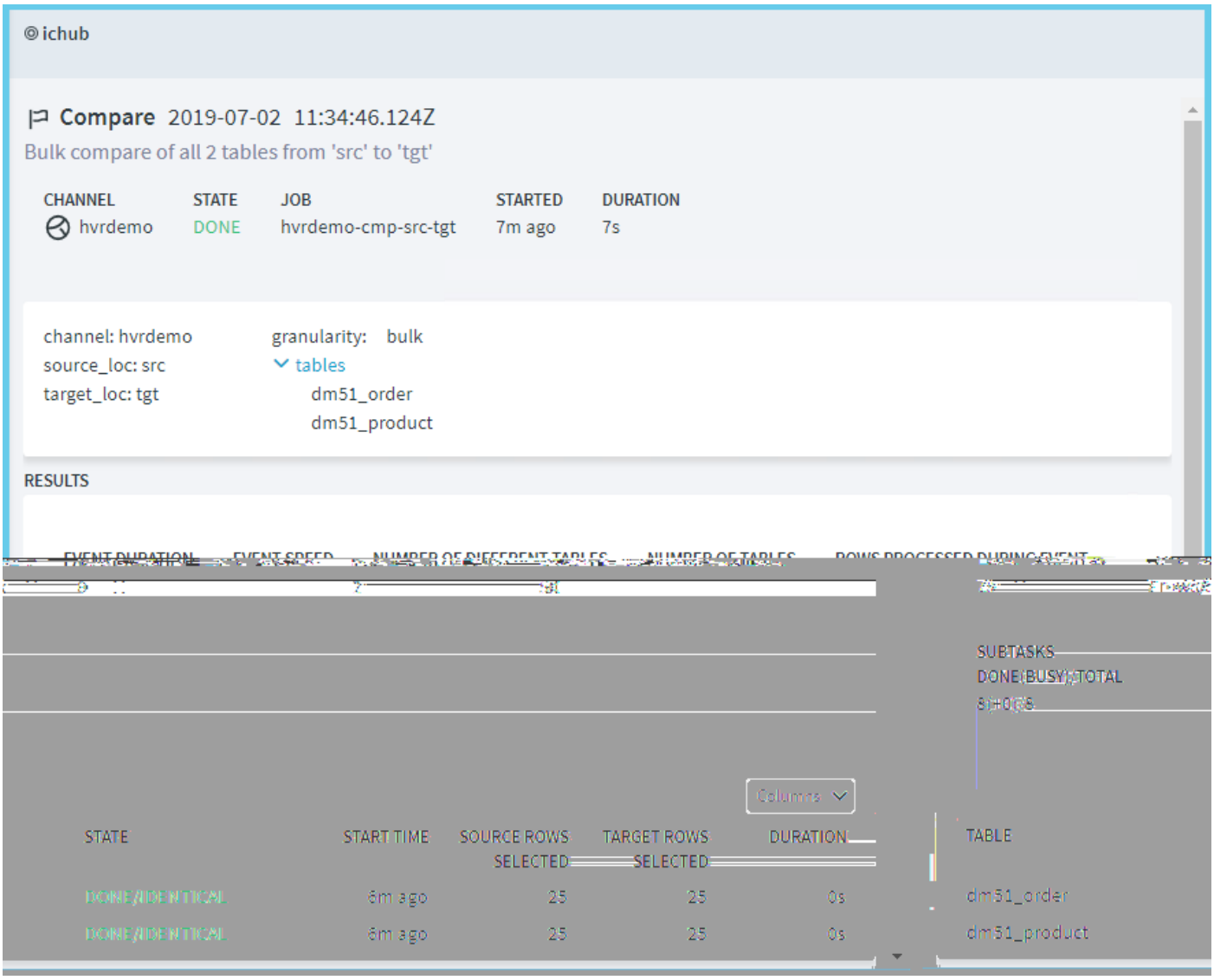
You can browse the list more efficiently using the **Type**, **State**, **Start** filters or the custom filter search field. To view more events, click **Fetch more previous events** at the bottom to expand the list.

The event list can be sorted by columns **Event ID**, **Type**, **Channel**, **State**, **User**, **Start time**, **Finish time**. You can select the columns to be displayed/hidden in the list in the **Columns** drop-down menu on the right.



Event Details

The **Event Details** page displays the details of a selected event. For example, the event details may include a channel and job associated with the event, the status of the event, when the event was started and its duration, as well as the names of source and target locations, tables involved, a user who initiated the event, and various parameters of the event, which may differ depending on the type of event. To open the **Event Details** page, click the ID of the required event.



Event-Driven Compare Results

As shown in the above screenshot, the **Results** pane of the **Event Details** page displays the results for the event-driven compare. Following are the fields/columns displayed for event-driven compare results:

Field	Description
COMPRESSED BYTES	Indicates the number of bytes HVR transferred (after compression)
COMPRESSION RATIO (BY MEMORY)	Indicates the number of compressed bytes transmitted over network compared with Hvr's representation of that row in memory.
DIFF FILE	Name of the file which contains the compare result in verbose - displays each difference detected as a BINARY file which can be read in XML format using **hvrrouterview** command. This is displayed only if **Verbose** (**–v**) option in HVR Compare is selected.
DURATION	Indicates the time taken to compare the specific table.
EVENT DURATION	Indicates the time taken to finish the compare operation.
EVENT SPEED	Indicates the speed of compare operation in rows per seconds.
NUMBER OF DIFFERENT TABLES	Indicates the number of tables that are not identical.

NUMBER OF TABLES	Indicates the total number of tables compared.
NUMBER OF INCONCLUSIVE TABLES	(Applicable only for Online Compare) indicates the total number of inconclusive tables.
ROWS IN MOTION IDENTICAL	(Applicable only for Online Compare) indicates the number of "in-flight" differences that are about to be replicated from source to target.
ROWS IN MOTION INCONCLUSIVE	(Applicable only for Online Compare) indicates the number of rows that are changing which are not identified as real differences or "in-flight" differences.
ROWS ONLY ON SOURCE	Indicates the total number of rows available only in source location that need to be inserted to target.
ROWS ONLY ON TARGET	Indicates the total number of rows available only in target location that need to be deleted from target.
ROWS PROCESSED DURING EVENT	Indicates the total number of rows compared.
ROWS WHICH DIFFERS	Indicates the total number of rows that need to be updated on target.
SOURCE ROWS SELECTED	Indicates the total number of rows selected in source table(s).
TARGET ROWS SELECTED	Indicates the total number of rows selected in target table(s).
SOURCE ROWS USED	Indicates the total number of rows that were actually compared in source.
TARGET ROWS USED	Indicates the total number of rows that were actually compared in target.
SPEED	Indicates the speed of compare operation while comparing the specific table in rows/secs.
START TIME	Indicated the time when the compare operation started.
STATE	Indicates the status of the table after the compare operation is completed: • **DONE/IDENTICAL** - indicates that the **Compare** event is completed and the tables are identical in source and target locations. • **DONE/DIFFERENT** - indicates that the **Compare** event is completed and the tables are NOT identical in source and target locations. • **DONE/INCONCLUSIVE** - (Applicable only for Online Compare) indicates that the **Online Compare** event is completed and tables are inconclusive, i.e. not leading to a firm conclusion. • **BUSY** - indicates that the table is processing right now (if the event is running). • **PENDING** - indicates that the table did not start processing yet. • **RETRY** - indicated that the Table is processing right now, but the event failed and restarted at least once during this table processing.
SUBTASKS DONE(BUSY)/TOTAL	Indicates the number of subtasks that are done(busy) and the total number of them.
TABLE	Indicates the name of the tables compared.

See Also

Command hvreventview, hvreventool

Advanced Operations

This section provides information on various operational features of HVR.

Configuring Multi-Directional Replication

A bi-directional topology assumes that data is replicated between two locations in both directions, and end users (applications) modify data on both sides. A multi-directional active/active replication involves more than two locations that stay synced up. For more information on different types of HVR replications, refer to section Replication Topologies.

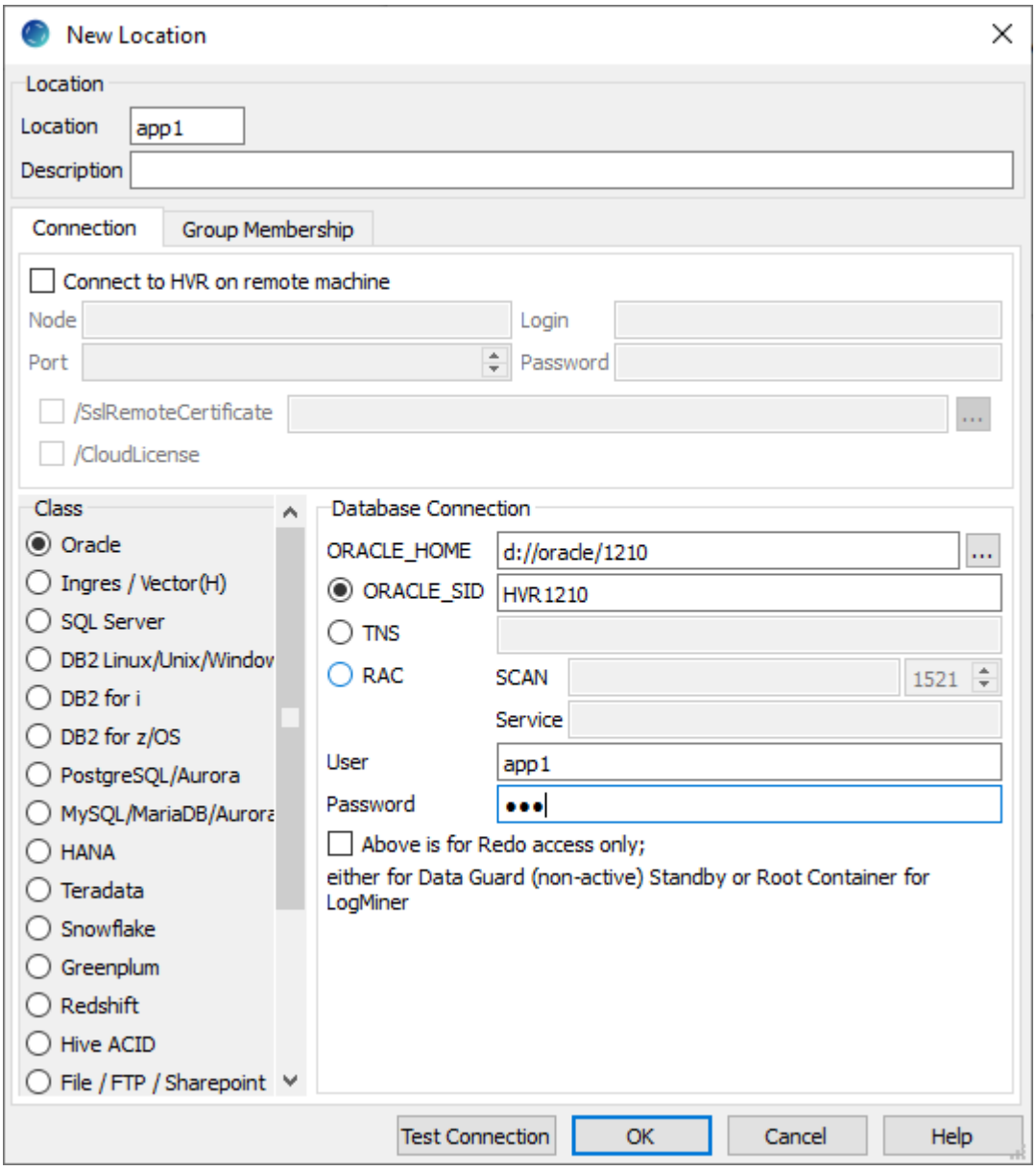
For the list of locations, on which bi-directional replication is supported, refer to the relevant section of Capabilities.

Channel Setup for Multi-Directional Replication

The steps described in this section are applicable for both bi-directional and multi-directional replication.

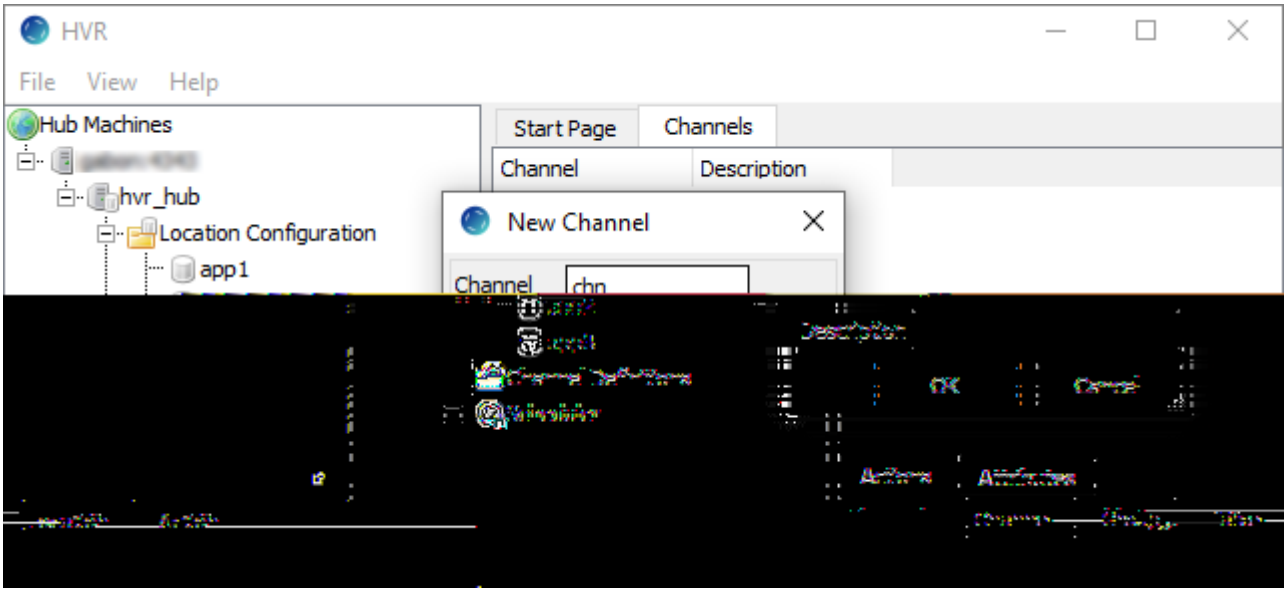
The following steps are to configure a multi-directional replication with three locations on Oracle machines.

1. In HVR GUI, after registering the hub, create three physical locations **app1**, **app2**, **app3**:
   a. In the navigation tree pane, right-click **Location Configuration** and select **New Location**.
   b. Fill in the appropriate fields to connect to location app1 (e.g. as shown on the image below) and click **Test Connection** to verify the connection to the database and then click **OK**.

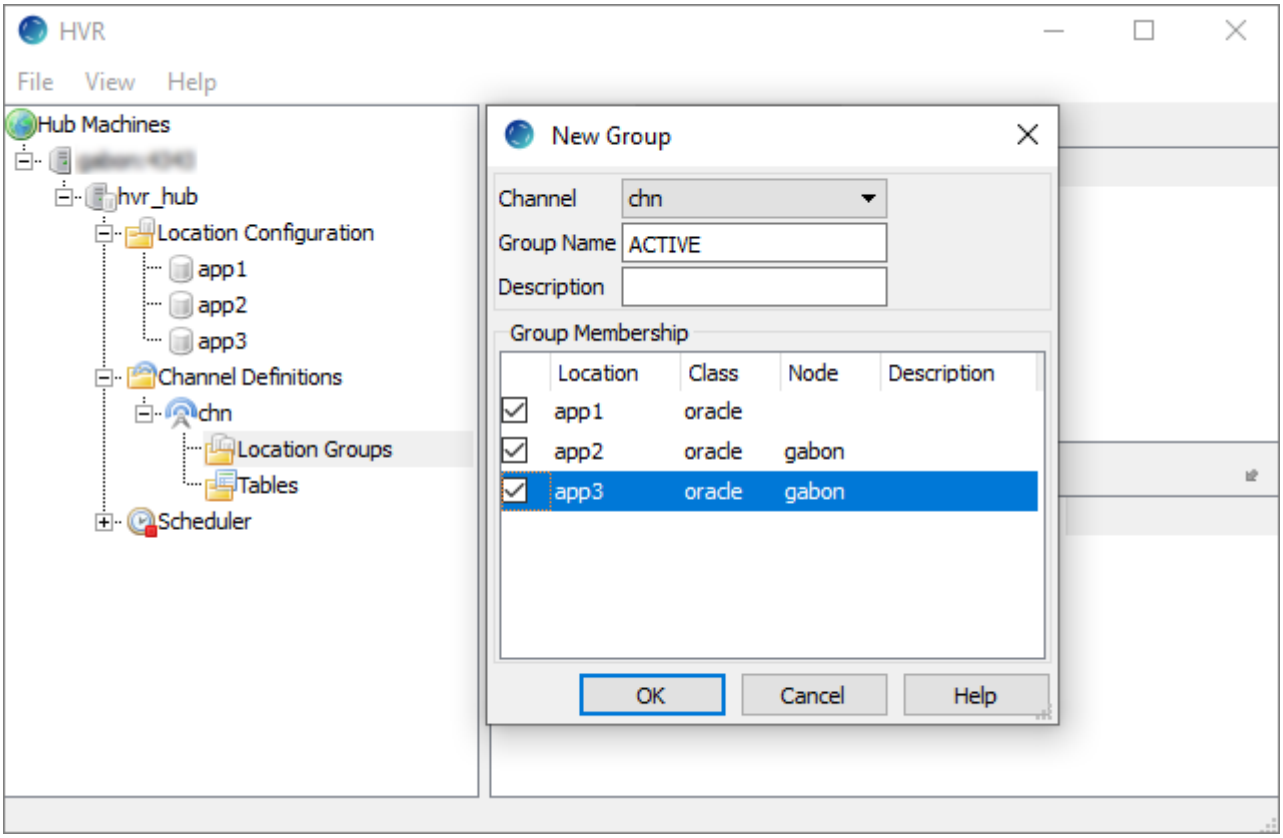


2. Repeat the above steps **a** and **b** to create locations **app2** and **app3**.

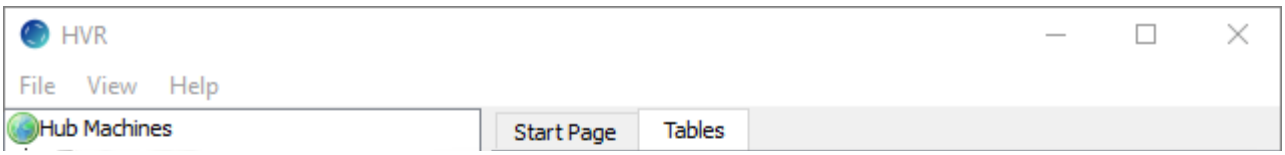
3. Define a channel for replication: right-click **Channel Definitions** and select **New Channel** and enter the name of the channel **chn**. Click **OK**.

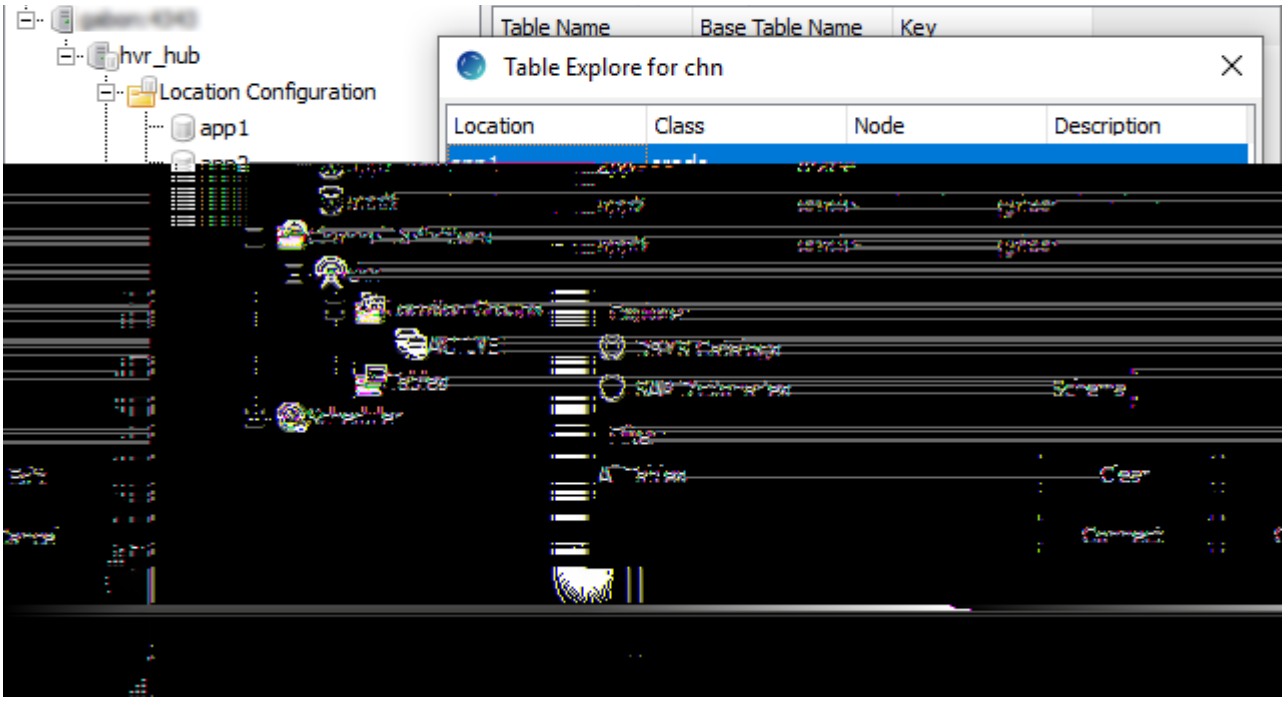


4. Define logical locations (location groups) for replication. Since this is a multi-directional configuration, a single location group will be defined including all physical locations.
   a. Click the plus sign (**+**) next to the channel **chn**, right-click **Location Groups** and select **New Group**.
   b. Enter the name of the group **Active** and select all of the locations **app1**, **app2**, **app3**. Click **OK**.



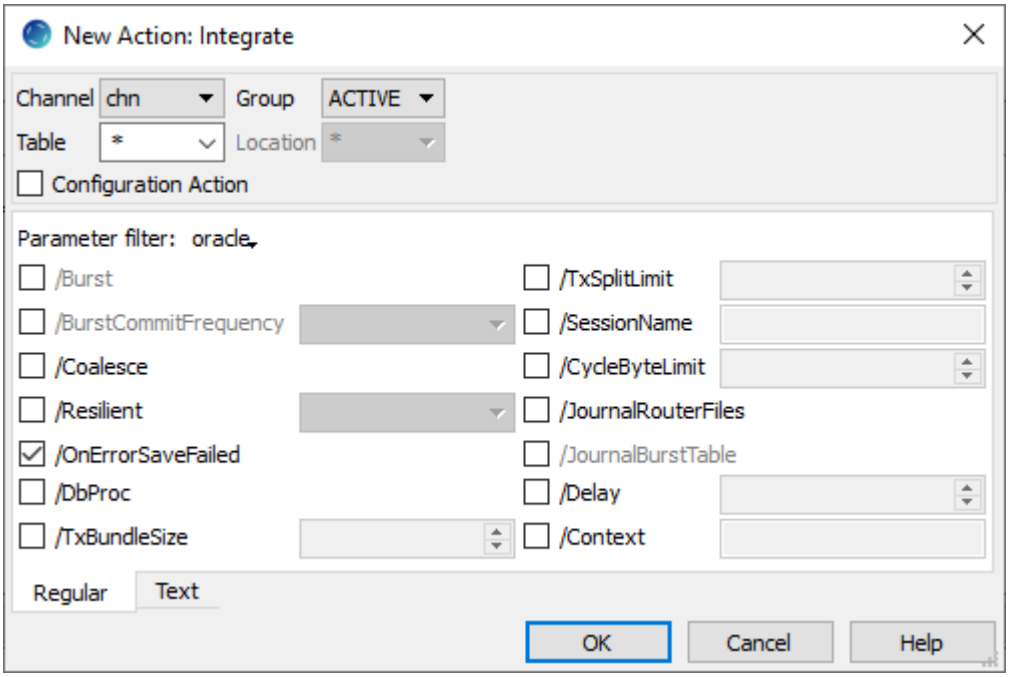
5. Location **app1** includes table **Product**. Right-click **Tables** and select **Table Explorer**. In the **Table Explorer** dialog, select **app1** and click **Connect**. Select the table and click **Add**.





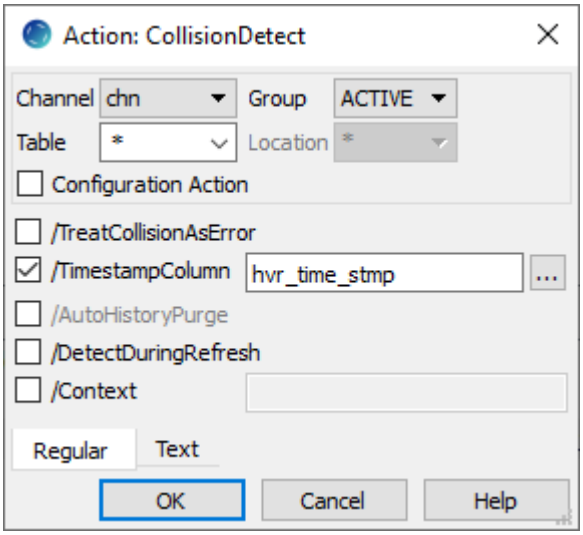
6. At least two actions must be defined on the channel:
   a. Right-click group **ACTIVE**, click **New Action** and select **Capture**. While adding action **Capture**, ensure that field **Table** has value '*****' and field **Group** has value '**ACTIVE**'.
   b. Right-click group **ACTIVE**, click **New Action** and select **Integrate**. In the Integrate dialog, select the **/OnErrorSaveFailed** parameter and click **OK**. Ensure that field **Table** has value '*****' and field **Group** has value '**ACTIVE**'.

> ⚠ Parameter **/OnErrorSaveFailed** allows HVR to continue replication when errors occur. The error message including the row causing the failure is stored in the fail table *tbl*_**f** and can be investigated afterwards (see command **Hvrretryfailed**).The errors will also be written to the log file.

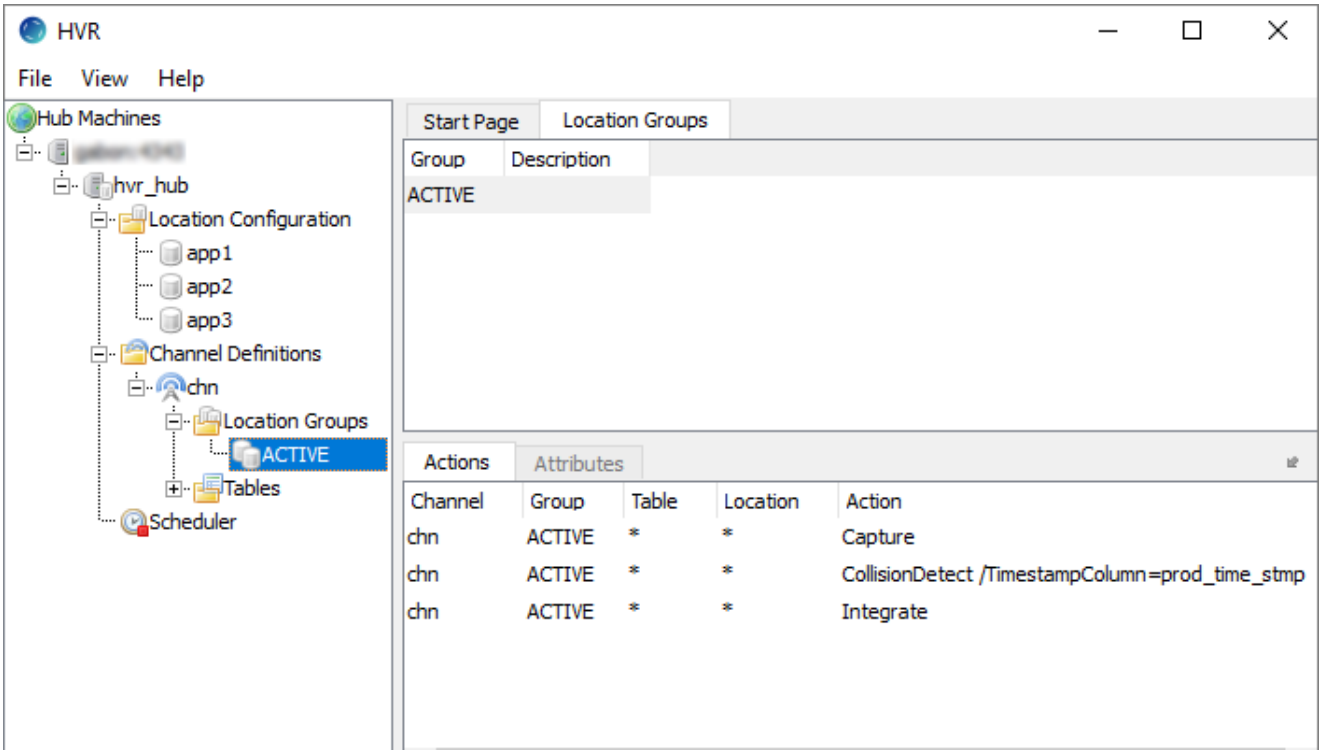


7. In a multi-directional replication environment, there can always be collisions. For example, users in different databases may update the same row at the same time, or a row may be updated in one database and deleted in another database. If collisions happen, then the end result may be that your databases get out of sync. HVR provides a sophisticated collision detect capability (**CollisionDetect**) that will ensure that the most recent change to a row always wins and systems remain in sync. Right-click group **ACTIVE**, click **New Action** and select **CollisionDetect**.
8. In the **CollisionDetect** dialog select option **/TimestampColumn** and select the timestamp column in the replicated tables for collision detection. A timestamp column should be manually added beforehand to each replicated table in the multi-directional channel. Consider using action **CollisionDetect** only on tables with a reliable timestamp column that accurately indicates when the data was last updated.

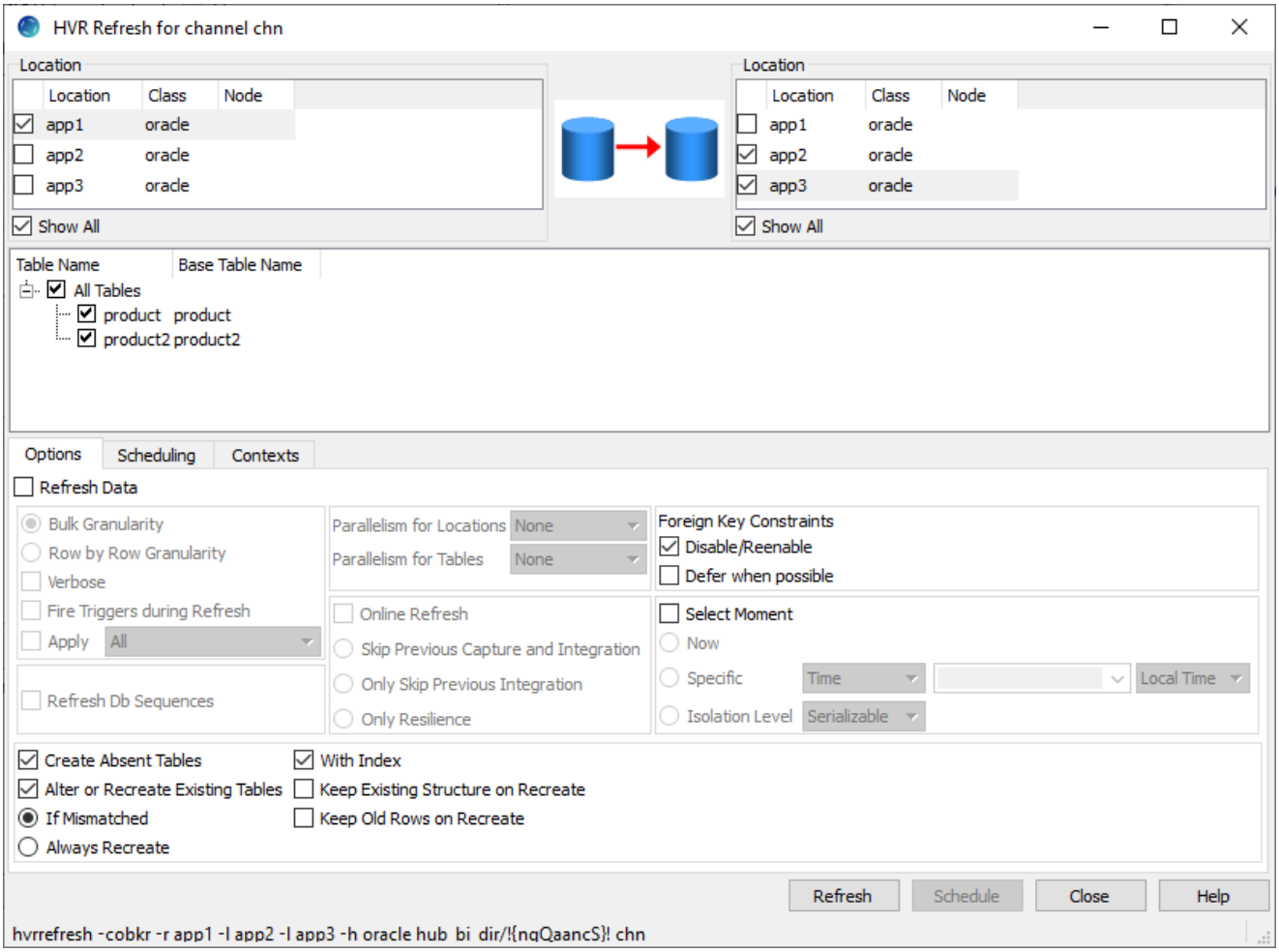
> ⚠ In some multi-directional replication environments, the collisions may be prevented at the application level (using customers' own application partitioning), thus avoiding the need to use action **CollisionDetect** .



The resulting channel definition is shown on the image below:

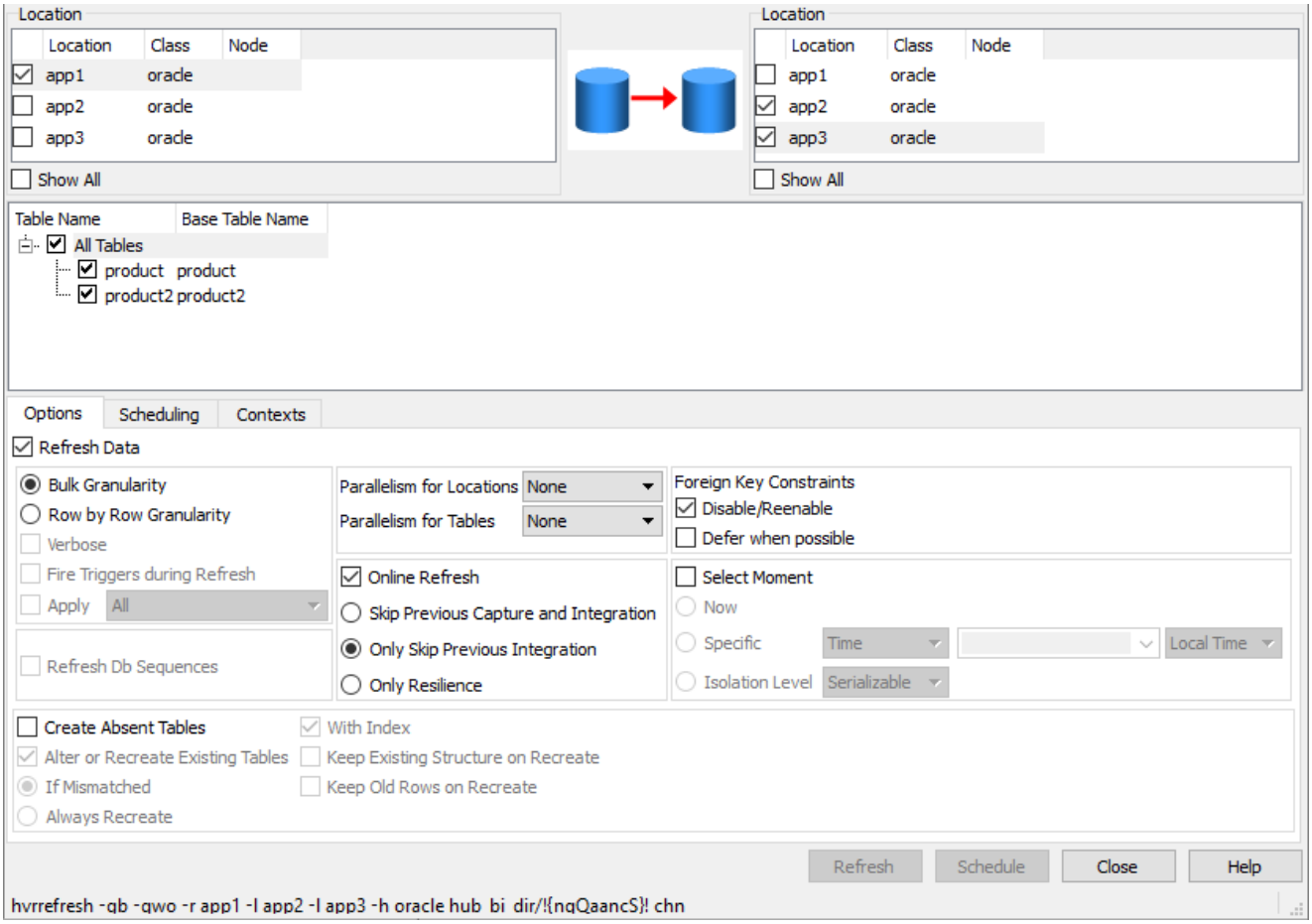


9. At the initial stage of setting up the multi-directional replication, some of the locations may not have the tables existing in another location. In this case, you need to create the tables in all locations in the channel and ensure they are all in sync. You can use the **HVR Refresh** capability for this. For example, only location **app1** has table **Product** at this point. To create the table in other two locations **app2** and **app3**:
   a. In the HVR GUI, right-click channel **chn** and select **HVR Refresh**. In the HVR Refresh dialog, select **app1** as a source and **app2** and **app3** as targets.
   b. Below, ensure that the required table is selected.
   c. Under the **Options** tab, select **Create Absent Tables**. Click **Refresh**.

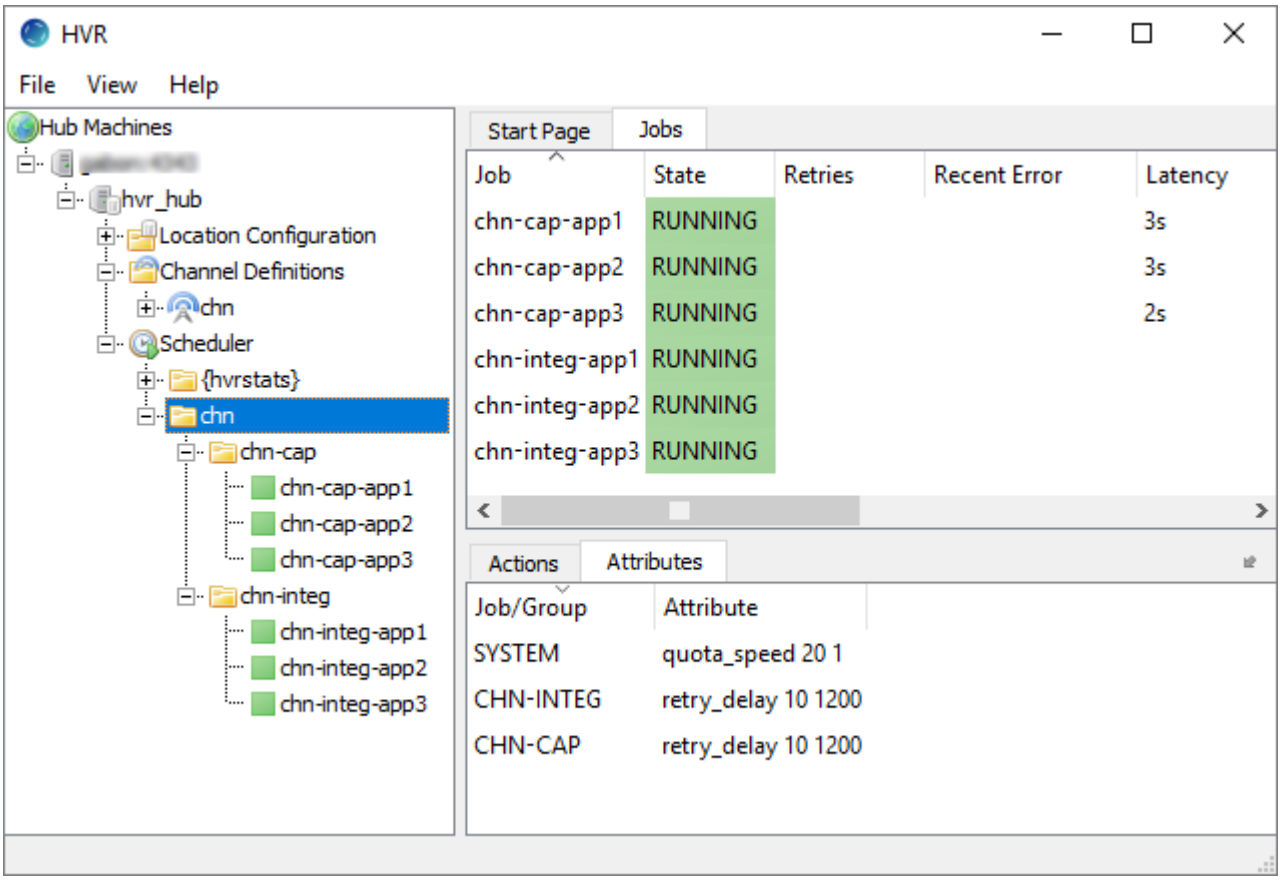


10. Once the absent tables are created in the other two locations, right-click channel **chn** and select **HVR Initialize**. HVR will create the corresponding jobs under the **Scheduler**.
11. After that, run **HVR Refresh** once again to sync the data in all the locations to ensure that no data changes made during the refresh are lost.

⚠ You can run the **Refresh** operation in parallel across all locations (tables) to speed up the execution. For this select the number of parallel refresh processes in the **Parallelism for Locations** and **Parallelism for Tables** fields.



12. Right-click the **Scheduler** and click **Start**. On Windows, the **HVR Scheduler** runs as a system service. Create a new **HVR Scheduler** service using the HVR GUI dialog.
13. Right-click the **Scheduler**, select **All Jobs in Systems**, and click **Start**. Once the jobs are started, they will go from the **SUSPEND** state to the **RUNNING** state. The setup is now complete.



Extended Data Type Support

Since v5.3.1/19

Contents

There are database-specific data types that are not natively supported by HVR. These data types are called "extended data types" in HVR. Different extended data types need different capture and integrate expressions to be defined on a channel to convert them to the ones supported by HVR. Generally, a cast to a **varchar** or **clob** column works well, though some types might be better represented by a numeric data type. For nested object types, a stored procedure to serialize the object may be defined. For examples, see section Expression Library below.

When a table with an extended data type is added to a channel, HVR's **Table Explore** displays the extended data types as a data type name enclosed in special markers: **<<***datatype***>>**. The *datatype* is the name of the data type as defined in a particular database and can be used in data type pattern matching similar to the regular data types.

Configuring HVR for Extended Data Types

Action **ColumnProperties** with parameters **/DataTypeMatch**, **/CaptureExpression**, **/CaptureExpressionType**, **/IntegrateExpression**, and **/DataType** should be defined for converting extended data types to supported data types during **capture**, **compare** or **refresh** (source) and back to extended data types during **integrate**, **compare** or **refresh** (target).

1. Right-click a channel, navigate to **New Action** and click **ColumnProperties**.
2. In the **New Action: ColumnProperties** window, select **/DatatypeMatch** and choose the required <<*datatype*>> from the drop-down list. The **/DatatypeMatch** parameter in combination with the **{{hvr_col_name}}** pattern in the capture and integrate expressions allows you to define capture expression based on the data type of a column, rather than the name of the column. This parameter fully supports extended data types, allowing you to specify a single expression for all columns of a specific extended data type in all tables in all channels.
3.
4.

-
-
-

Example:

The parameters defined as: **/CaptureExpression=cast(col as varchar) /CaptureExpressionType=SQL_WHERE_ROW** cause HVR to generate the following SQL expression against a table being replicated:

```
select (cast(col as varchar)) from schema.table where key1=key1, ...
```

5. Select the **/IntegrateExpression** option and type in the required expression depending on the database involved in the replication. See section Expression Library below for the extended data types in different databases and their relevant capture and integrate expressions.

The following screenshot demonstrates an example setup of the **ColumnProperties** action to capture an extended data type.

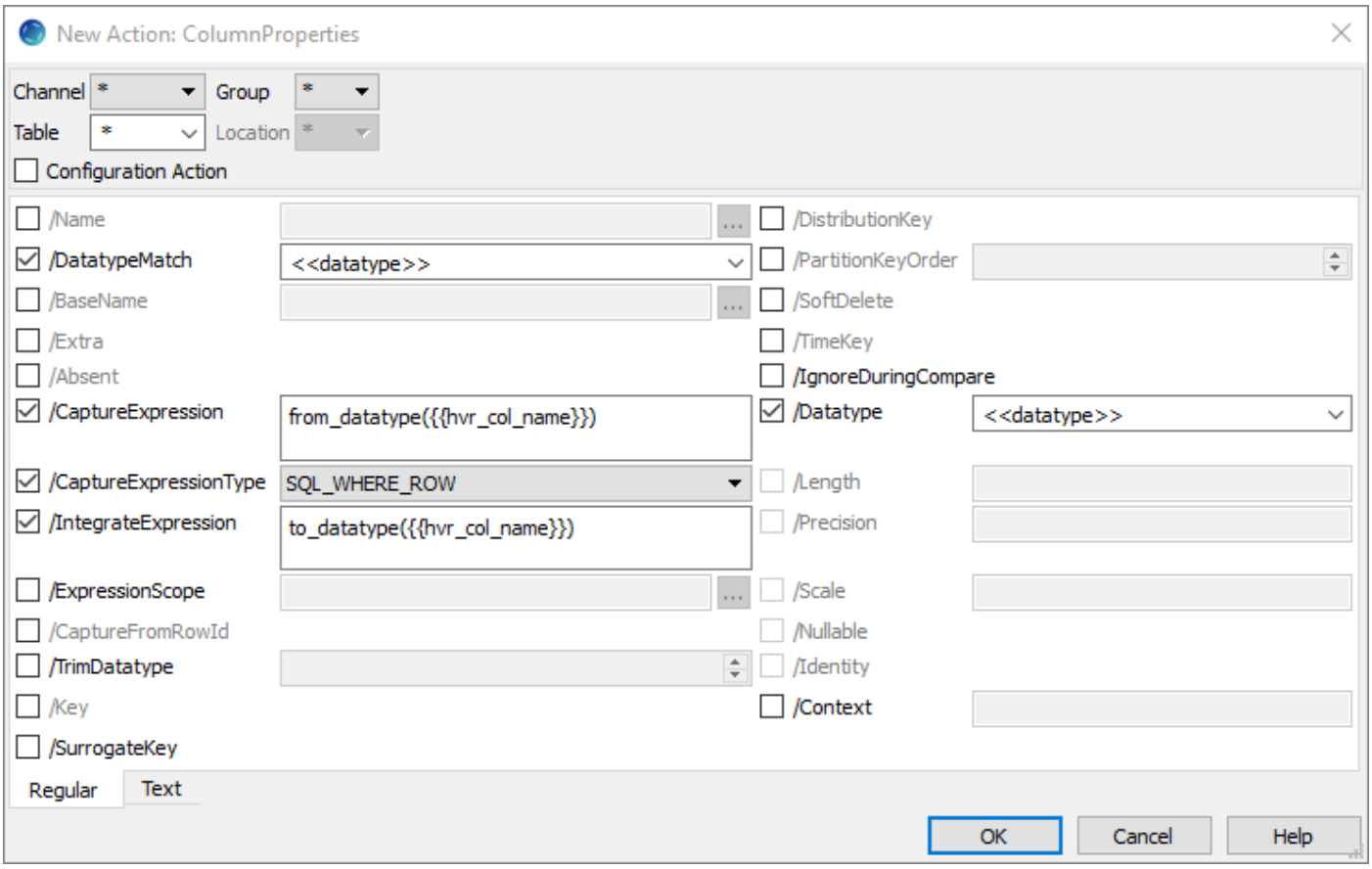
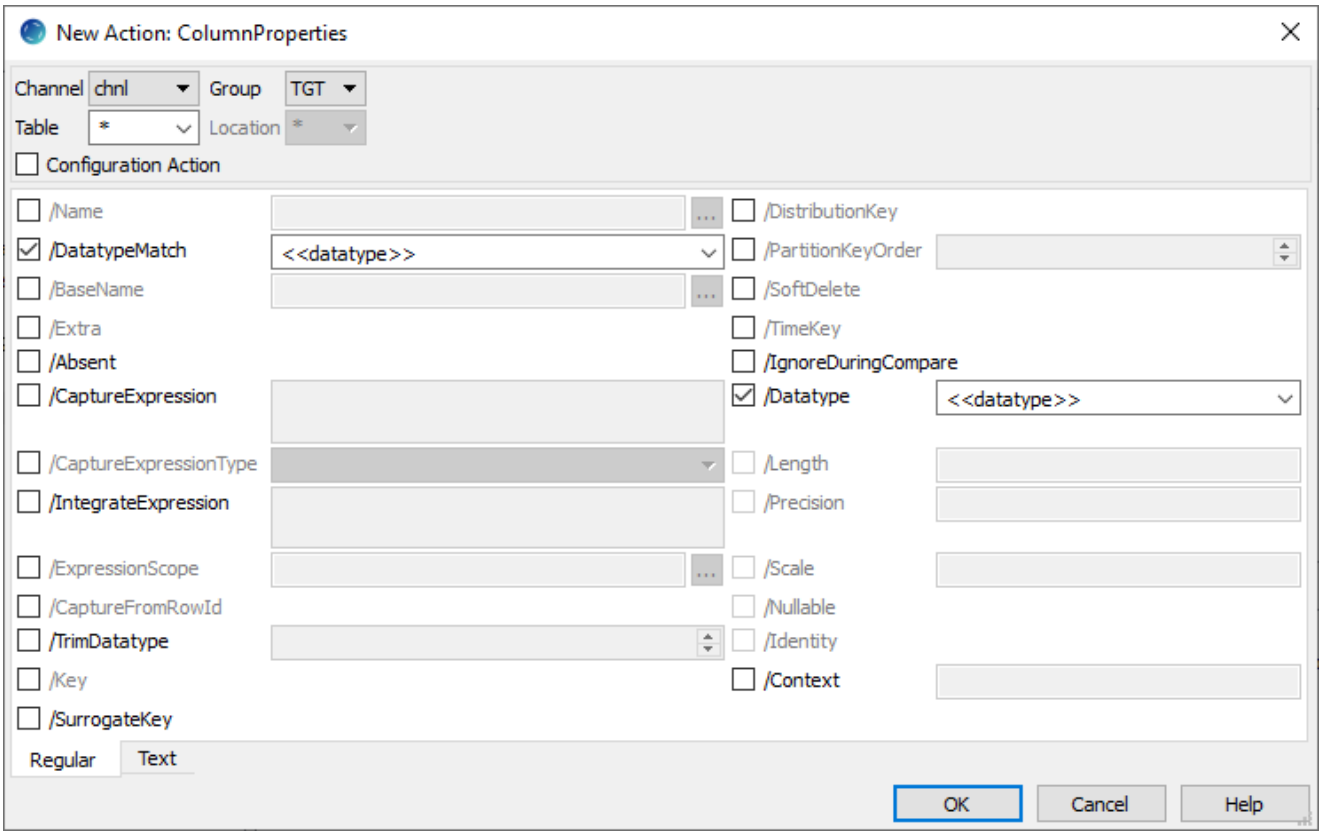


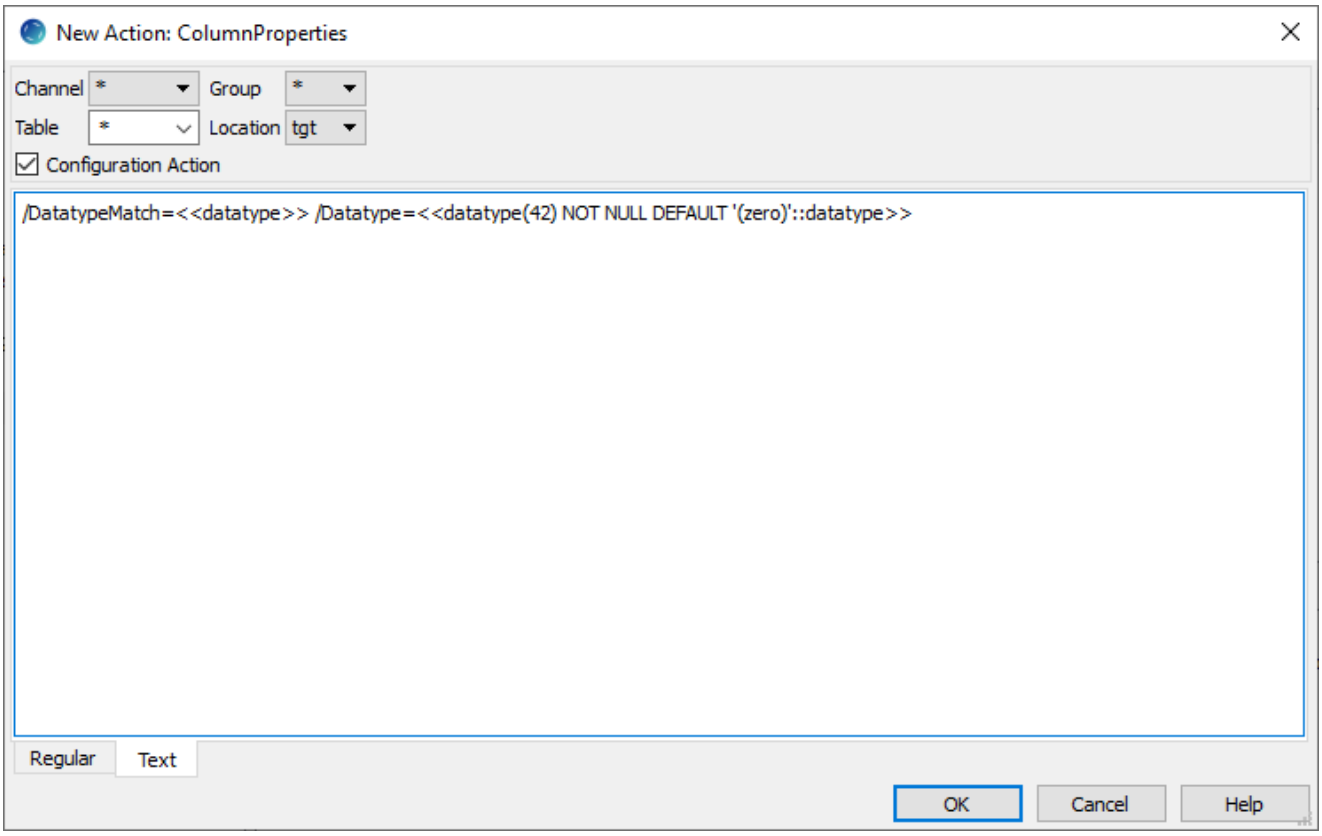
Table Create

The extended data type defined as **<<datatype>>** in HVR is just a base name of a specific data type without any attributes like **NOT NULL**, **DEFAULT**, or allowed values in enumeration-like data types. This might not be sufficient to create a table on a target side. So, to enable table creation on the target side, you need to set up **ColumnProperties** **/DatatypeMatch** and **/DataType** parameters on the target location.

1. Right-click a target location, navigate to **New Action** and select **ColumnProperties** from the list.
2. In the **New Action: ColumnProperties** window, select the **/DatatypeMatch** parameter and choose the required extended data type from the drop-down list.
3. Select the **/Datatype** parameter and select the same from the drop-down list.



4. Click the **Text** tab in the bottom left corner. In the text editor, type the necessary attributes that HVR will put in the **CREATE TABLE** statement on the target side. An example expression for the **/Datatype** parameter can be **/DataType=<<data type(42) NOT NULL DEFAULT '(zero)'::datatype>>**.



Final Expression

An example expression for the entire channel with columns of extended data type **<<datatype>>** for table creation can be as follows:

Group	Table	Action
*	*	**ColumnProperties /DatatypeMatch=<<datatype>> /CaptureExpression="from_datatype({{hvr_col_name}})" /CaptureExpressionType=SQL_WHERE_ROW /IntegrateExpression="to_datatype({{hvr_col_name}})" /DataType="<<datatype(42) NOT NULL DEFAULT '(zero)'::datatype>>"**

Excluding Extended Data Types from Replication

To ignore replication of an extended data type that is enrolled in the table definitions of the HVR catalogs, define the **ColumnProperties /Absent** parameter. Since capture of **/Absent** columns requires a capture expression, you can use a dummy expression to satisfy that requirement. If you add the **/CaptureExpression** of type **SQL_PER_CYCLE**, the number of executions of this expression is reduced to one per cycle negating the performance cost. Reasons for doing this include the convenience of enrolling tables as they are in the source database when a target database cannot accept this data type.

Group	Table	Action
*	*	**ColumnProperties /DatatypeMatch=<<datatype>> /CaptureExpression="0" /CaptureExpressionType=SQL_PER_CYCLE /Absent**

Advantages and Disadvantages

The primary advantage is that the extended data type feature allows HVR to access data types that are not supported, and allow full flexibility to tune the use of otherwise unsupported or totally custom data types, even in heterogeneous replication scenarios. The downside includes performance cost and the requirement to add custom expressions.

Capture Performance

Since HVR can not process the native representation of extended data types, there is a performance cost of capturing these types. For each row, HVR will need to do a query to the source database to augment in the value as a supported type. This also causes the consistency to change to eventual consistency since there will be a time discrepancy between the commit and the execution of the capture expression in the order of the capture latency.

Integration has no noticeable overhead.

Bulk Refresh Performance

In general, during bulk refresh, HVR uses the bulk load APIs of the target database platform. These, however, require native support for all data types of the interface, and generally, do not allow expressions. If extended data types are present in a table, this table drops down to batched SQL statements for insertion.

Bulk compare or row-wise refresh and compare have no noticeable overhead.

Coercion

HVR coerces data types from the source database to the target database, but can not do this for extended data types. However, HVR ensures that the data type returned by the capture expression is localized to a data type supported by the integrate location. It is the responsibility of the integrate expression to deal with possible incompatibilities that might result from interpreting a value. This means that features like **TableProperties /CoerceErrorPolicy** only apply to the localization of the data type, not to the processing of the integrate expression.

AdaptDDL

While using **AdaptDDL** in combination with extended data types the following should be noted:

- Tables with extended data type require expressions, and **AdaptDDL** can add tables. If **AdaptDDL** adds tables to your channel, but they do not have any expressions defined beforehand, the channel will fail. If you use **/DatatypeMatch** to define the expressions of data types that will be adapted in the future, they will be used.
- There is a restriction in comparing extended data types. HVR does not assign meaning to the name the database gives the data type, that is located in the type inside markers **<< >>**. For comparing, HVR considers all extended data types equal to each other and can not detect differences for the purpose of updating channel definitions or executing **ALTER TABLE** statements on the target.

Restrictions

- Executing capture expressions during **Capture** requires a **WHERE** clause containing key information, so a key column with an extended data type cannot be used during **Capture**.
- Extended data types cannot be used on a primary key column.

Expression Library

MySQL

- Extended Data Type: **set**
  /CaptureExpression=cast({{hvr_col_name}} as char)
  /IntegrateExpression={{hvr_col_name}}

Oracle

- Extended Data Type: **xmltype**
  /CaptureExpression=xmltype.getClobVal({{hvr_col_name}})
  /IntegrateExpression=xmltype.createXml({{hvr_col_name}})
- Extended Data Type: **SDO_GEOMETRY**
  /CaptureExpression=SDO_UTIL.TO_WKTGEOMETRY({{hvr_col_name}})
  /IntegrateExpression=SDO_UTIL.FROM_WKTGEOMETRY({{hvr_col_name}})

PostgreSQL

- Extended Data Type: **interval**
  /CaptureExpression=cast({{hvr_col_name}} as varchar(100))
  /IntegrateExpression=cast({{hvr_col_name}} as interval)

SQL Server

- Extended Data Type: **sql_variant**
  /CaptureExpression=convert(nvarchar,{{hvr_col_name}}, 1)
  /IntegrateExpression=cast({{hvr_col_name}} as nvarchar)

> ⚠ Using these expressions integrates each values with a **BaseType** of **nvarchar**.

- Extended Data Type: **geometry**
  /CaptureExpression={{hvr_col_name}}**.ToString()**
  /IntegrateExpression=geometry::STGeomFromText({{hvr_col_name}},0)

- Extended Data Type: **geography**
  /CaptureExpression={{hvr_col_name}}**.STAsText()**
  /IntegrateExpression=geography::STGeomFromText({{hvr_col_name}},4326)

- Extended Data Type: **hierarchyid**
  /CaptureExpression={{hvr_col_name}}.ToString()
  /IntegrateExpression=hierarchyid::Parse({{hvr_col_name}})

Managing Recapturing Using Session Names

Contents

In HVR, session name is the name of a transaction performed by the integrate job. In all DBMSes, for which capture is supported, transactions performed by the integrate job also contain an update to the integrate state table (named **hvr_stin*** or **hvr_stis***) which has column **session_name** containing the session name. This allows log-based capture jobs to recognize the session name defined by the integrate job, but this is ignored by the trigger-based capture jobs.

The default session name is **hvr_integrate**. Normally, HVR capture avoids recapturing changes made on the integration side by ignoring any changes made by sessions named **hvr_integrate**.

Replication recapturing is when changes captured on a source location are captured again on the integration side. Recapturing may be controlled using session names and actions **Capture /IgnoreSessionName** and **Integrate /SessionName**. Depending on the situation recapturing can be useful or unwanted.

The following are typical cases of how recapturing is managed using session names.

Cascade Replication

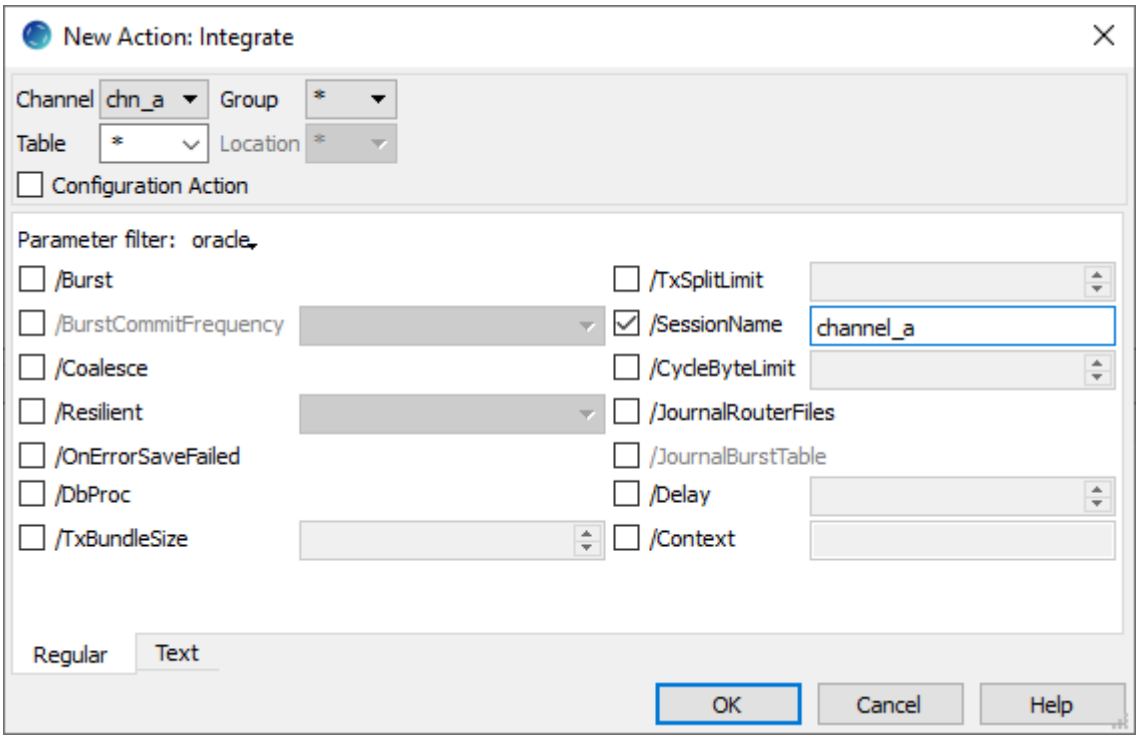
Cascade replication is when changes from one channel are captured again by another channel and replicated to a different group of databases.

Recapturing is necessary for cascade replication. In this case, recapturing can be configured using action **Integrate /SessionName** so that integration is not recognized by the capture triggers in the cascade channel. This action allows to integrate changes with specific session name.

Example

Changes made by **channel A** do not get captured by **channel B** by default. The default session name of **channel A** in database **MID** is **hvr_integrate** and the capture job of **channel B** ignores sessions with the **hvr_integrate** name.

To capture changes, the default session name of **channel A** must be changed from the default value to a different name. Do this by setting the **/SessionName** parameter to another value, for example **channel_a**.

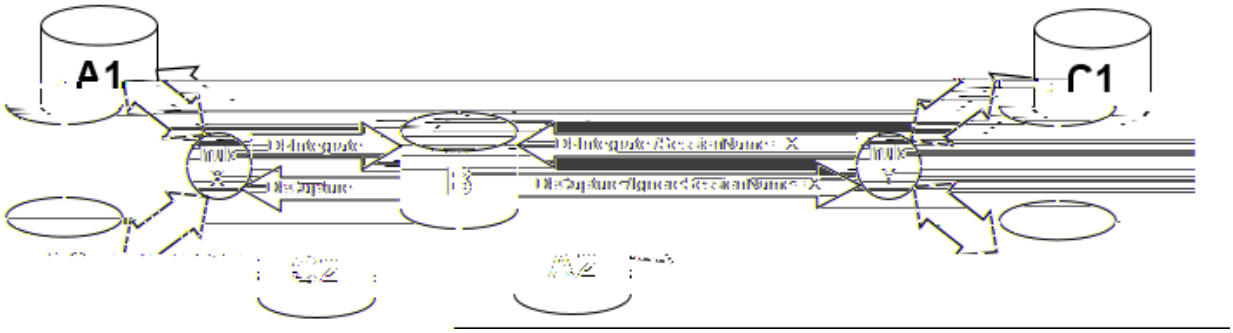


Bi-directional Replication

During bidirectional replication, integrated changes can be accidentally captured again boomeranging back to the original capture database. This will form an infinite loop. For this reason, action **Capture /IgnoreSessionName** should be used to check the session name and avoid recapturing on the integration side. This action instructs the capture job to ignore changes performed by the specified session name.

Example

Changes made to databases **A1** and **A2** must be replicated via **hub X** to database **B** and then cascade via **hub Y** to databases **C1** and **C2**. Changes made to **B** must replicate to **A1** and **A2** (bi-directional replication) and to **C1** and **C2**, but must not boomerang back to **B**. Normally changes from **A1** and **A2** to **B** would not be cascade replicated onto **C1** and **C2** because they all use the same session name (**X**). This is solved by adding parameters **/SessionName** and **/IgnoreSessionName** to the channel in **hub Y**.



Bi-directional Replication using MySQL

By default, during HVR Bulk Refresh, a target table is truncated and then a bulk copy of data from the source table is transmitted to the target table.

For a MySQL database, HVR does not associate a truncate table operation with a session name, i.e. HVR does not distinguish between the truncate table operation initiated by the HVR Bulk Refresh and the truncate table operation initialed by a third-party user application.

Therefore, in MySQL bi-directional setup, the HVR Bulk Refresh may cause looping truncates on either side of the bi-directional setup, which in turn will result in all data getting deleted.

One possible workaround to this is to define action **Restrict** with parameter **/RefreshCondition** set to '1=1' on a channel. This will delete all data from the table not truncate the table. Deleting the data will generate more transaction log output and take longer than truncating, but in the bi-directional setup, deleting will not cause the loopback issue.

Another approach for this would be to suspend the capture job when truncates happen (e.g. as part of the Bulk Refresh) and reset the capture time (using option Capture Rewind) beyond the time when the truncates happened before restarting the capture job.

Batch Work to Purge Old Data

Sometimes replication of large blocks of batch work is too expensive. Capturing of the batch work can be disabled by setting the session name so that the capture job ignores the changes done by that batch job. For log-based capture, the session name must be set using the 'current user name' approach (see below), as other approaches are not recognized for this.

Application Triggering During Integration

HVR has multiple implementations for session names because of variations in the DBMS features and because some implementations can only be recognized by trigger-based capture jobs and others can only be recognized by log-based capture jobs.

Sometimes an application will have database triggers on the replicated tables. These triggers have already been fired on the capture database so firing them again during HVR integration is unnecessary and can cause consistency problems. For Ingres and SQL Server databases, this rule firing can be avoided with action **Integrate /NoTriggerFiring**.

Current user name

In all DBMSes, for which capture is supported, trigger-based capture jobs also recognize the current DBMS user name as the session name. Therefore, end user sessions can make changes without activating the trigger-based capture by connecting to the database using a specially created user name (e.g. making database changes as user name **hvr_integrate**). An application database trigger can be prevented from firing on the integration side by changing it to include the following clause: **where user <> 'hvr_integrate'**.

Ingres role

For Ingres trigger-based capture, the integrate job always connects to the target database using an Ingres role, which is the name of the session (e.g. **hvr_integrate** or **hvr_refresh**). This is recognized by the trigger-based capture jobs. Therefore, end-user sessions can make changes without activating the trigger-based capture by connecting to the database with **SQL** option **–R**. An application database trigger can be prevented from firing on the integration side by changing it to include the following clause: **where dbmsinfo('role') != 'hvr_integrate'**.

SQL Server marked transaction name (log-based capture)

For SQL Server, log-based capture jobs also recognize the marked transaction name as the session name. Therefore, end user sessions can make changes without activating log-based capture by executing their DML statements inside a marked transaction with a specially created name (e.g. **hvr_integrate** or **hvr_refresh**). The following SQL statement can be used to start a marked named transaction:

```
begin transaction hvr_integrate with mark;
```

SQL Server application name (trigger-based capture)

For SQL Server trigger-based capture only, the integrate job always connects to the target dataset using an application name which is the session name (e.g. **hvr_integrate**). End user sessions can therefore make changes without trigger-based capture being activated by also connecting to the database using a specific application name. An application

database trigger can be prevented from firing on the integration side by changing it to include the following clause: **where app_name() <> 'hvr_integrate'**.

Manually Adapting a Channel for DDL Statements

Contents

A channel needs to be adapted only when new table(s) are added to an existing channel, or when the definition of table (s) that were already being replicated have changed, or when tables are removed from a channel.

For certain databases, HVR continuously watch for Data Definition Language (DDL) statements (using action **AdaptDDL**) and automatically performs the required steps to adapt a channel. For the list of supported databases, see Log-based capture of DDL statements using action AdaptDDL in Capabilities.

For all other database sources that do not support capture of DDL statements using action **AdaptDDL** or if the action **AdaptDDL** is not defined, HVR only captures the DML statements - **insert**, **update** and **delete** as well as **truncate** table (modify to truncated) and replicates these to integrate database. But DDL statements, such as **create**, **drop**, and **alter** table are not captured by HVR. In this scenario, to capture the DDL statements, the required steps to adapt the channel needs to be performed manually.

When DDL statements are used, the following must be considered:

- These statements are not replicated by HVR, so they must be applied manually on both the capture and integrate databases.
- The HVR channel which replicates the database must be changed ('adapted') so it contains the new list of tables and columns, and the enroll information contains the correct internal table id number.
- For Ingres log–based capture, after an **alter table** statement an extra modify statement is needed to convert all the rows which are stored with the old column format. The statement is **modify** *mytbl* **to reconstruct**, or (assuming the old structure was a unique **btree**) **modify to** *mytbl* **btree unique**.

There are two ways to manually adapt a channel:

1. **Online Manual Adapt**: This method is less disruptive because while performing this procedure users can still make changes to all tables. However this method requires you to perform more steps to adapt a channel.
2. **Offline Manual Adapt**: This method is more disruptive because while performing this procedure users are not allowed to make changes to any of the replicated tables. However this method requires you to perform fewer steps to adapt a channel. This method is preferred when the application (e.g. SAP or Oracle eBusiness Suite or any other similar major application) is not making any changes to the database because of a planned downtime (e.g for application upgrade).

The steps mentioned in the following sections do not apply for trigger-based capture and bi–directional replication (changes travelling in both directions). For such situations, contact HVR Technical Support for minimal–impact adapt steps.

Online Manual Adapt

Perform the following steps to manually adapt a channel using Online Manual Adapt method:

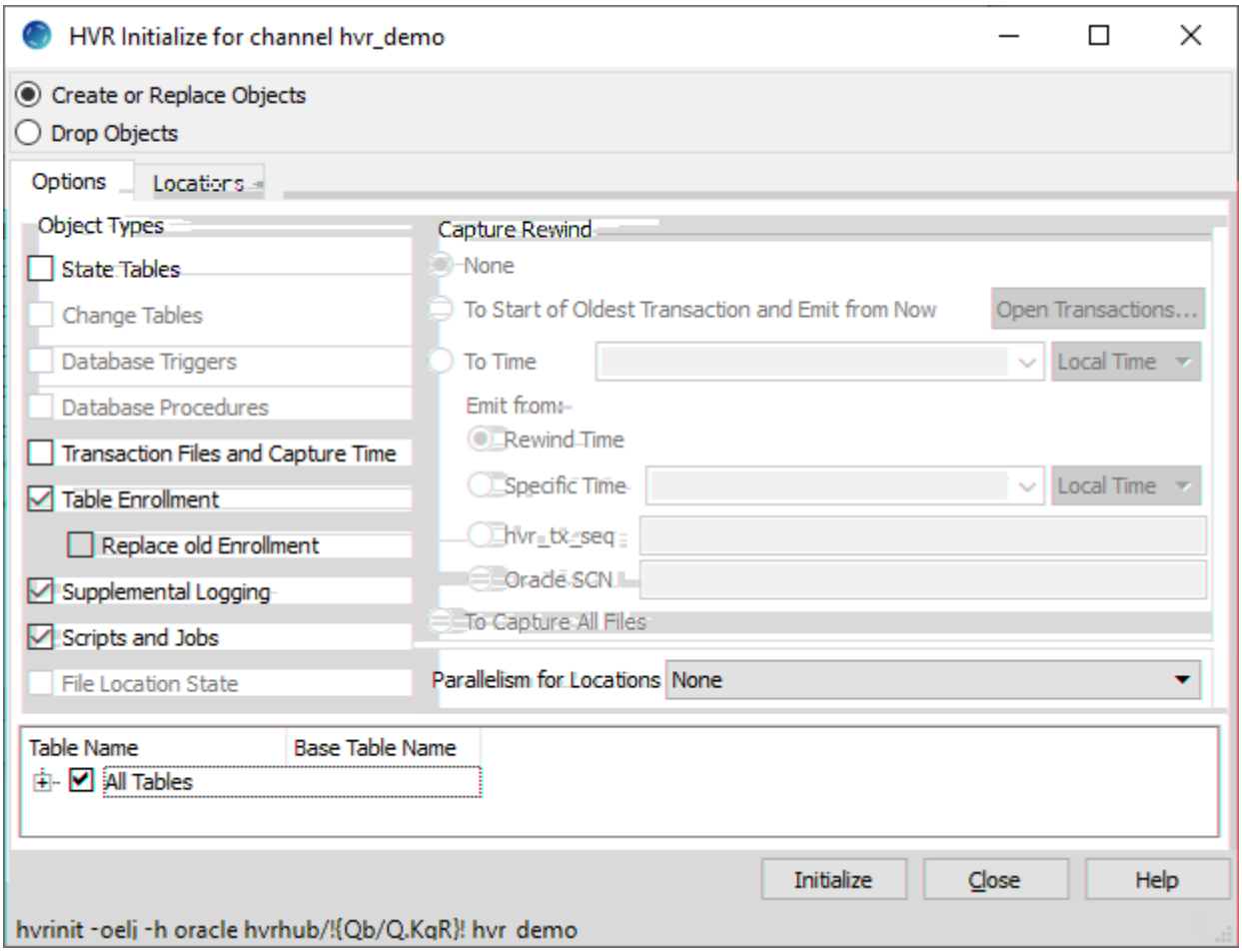
1. **Suspend** the capture jobs, wait until the integrate jobs have finished integrating all the changes (so no transaction files are left in the router directory) and then suspend them too.

```
hvrsuspend hubdb chn-cap
```

```
hvrstart -w hubdb chn-integ
```

```
hvrsuspend hubdb chn-integ
```

2. Run the SQL script with the DDL statements against both the source and target databases, so that database schemas become identical.
3. Manually adapt the channel definition so it reflects the DDL changes. This can be done in the HVR GUI or on the command line. In the HVR GUI select option **Table Explore** and connect to the capture database to incorporate the changes in the channel definition.
    a. Use **Add** to add the tables that are **Only in Database** or **In other Channel**.
    b. Use **Replace** to change the definition of the tables that have **Different Keys**, **Different Columns** or **Different Datatypes**.
    c. Use **Delete** for tables that are **Only in Channel**.
4. Use **Table Explore** to the integrate locations to check that all the tables have value **Same** in the **Match** column.
5. Execute **HVR Initialize** to regenerate the **Table Enrollment** information, the replication **Scripts and Jobs**, and enable DBMS logging (**Supplemental Logging**) for the new tables in the capture database.

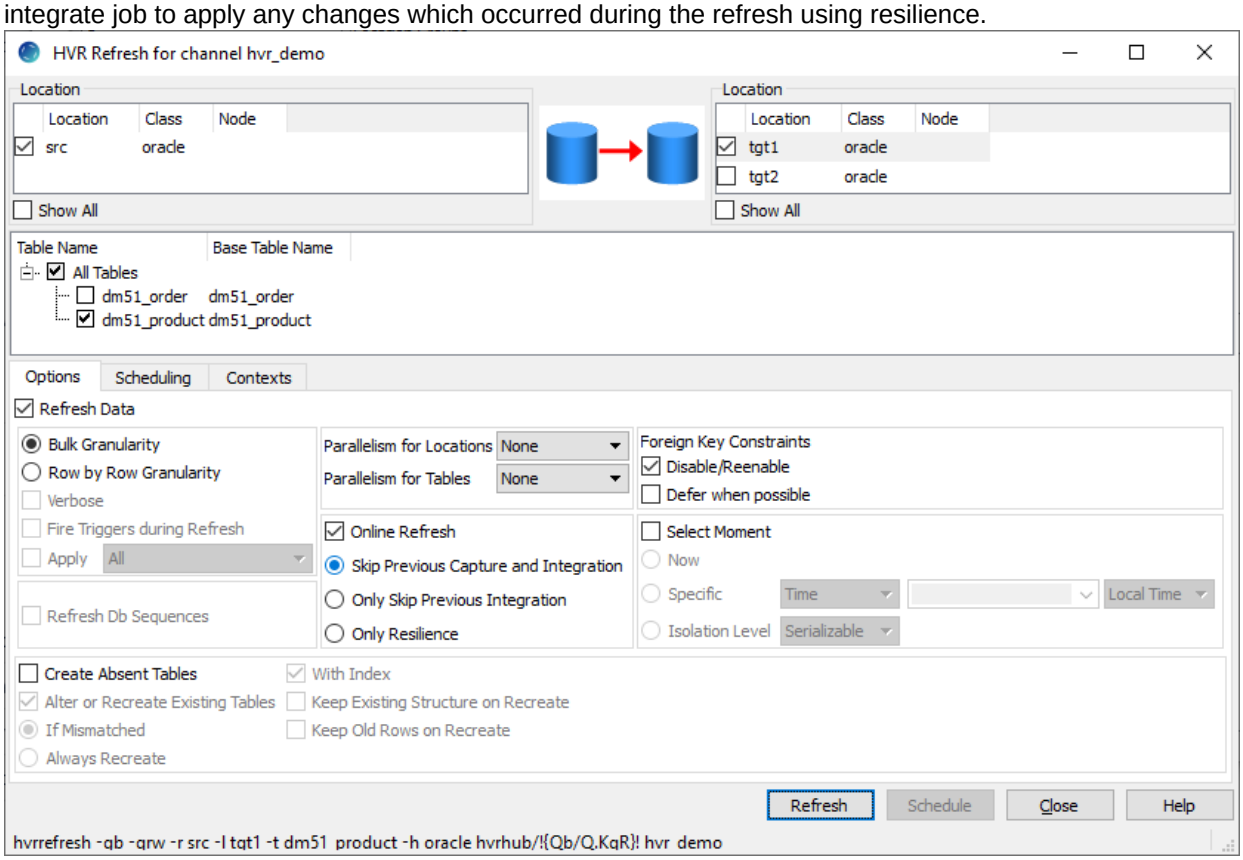


This can also be executed on the command line as follows:

```
hvrinit -oelj hubdb chn
```

6. Execute **HVR Refresh** to synchronize only the tables that are affected by DDL statements (except for tables that were only dropped) in the SQL script in step **2**. Tables which were only affected by DML statements in this script do not need to be refreshed. It is also not necessary to refresh tables which have only had columns added or removed.

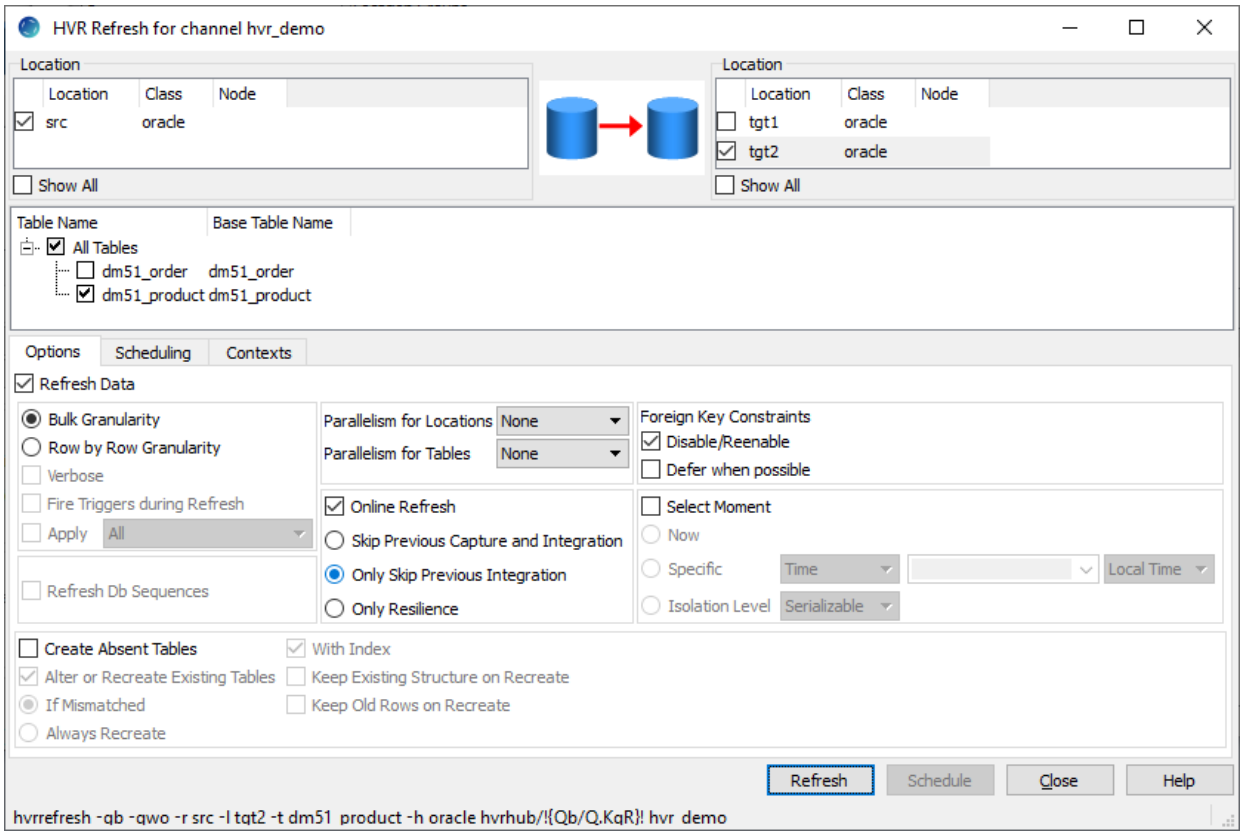
   - For the first target location, the **Online Refresh** option **Skip Previous Capture and Integration** (**–qrw**) should be used to instruct the capture job to skip changes which occurred before the refresh and the integrate job to apply any changes which occurred during the refresh using resilience.



   This can also be executed on the command line as follows:

```
hvrrefresh -gb -qrw -r src -l tgt1 -t dm51_product -h oracle hvrhub/!{Qb/Q.
KqR}! hvr_demo
```

   - For any extra target location(s), use the **Online Refresh** option **Only Skip Previous Integration** (**–qwo**) because the capture job should not skip any changes, but the integrate jobs should apply changes which occurred during the refresh using resilience.



This can also be executed on the command line as follows:

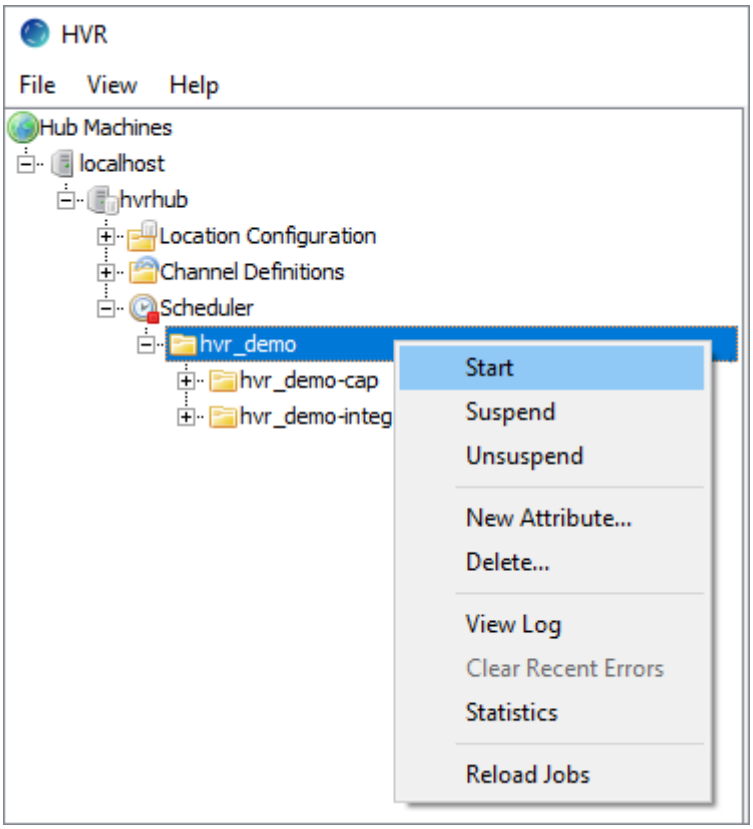
```
hvrrefresh -gb -qwo -r src -l tgt2 -t dm51_product -h oracle hvrhub/!{Qb/Q.
KqR}! hvr_demo
```

> For an Ingres target database, performing bulk **refresh** (option **-gb**) will sometimes disable journaling on affected tables. If **Hvrrefresh** had displayed a warning about disabling journaling then it is necessary to execute the Ingres command **ckpdb +j** on each target database to re-enable journaling.

7. If any fail tables exists in the integrate location(s) (**/OnErrorSaveFailed**) for the tables which have had columns added or dropped, then these fail tables must be dropped. For this, execute **HVR Initialize** with **Change Tables** option (**-oc**) selected.

```
hvrinit -oc -l int1 -l int2 -t tbl1 -t tbl2 hubdb chn
```

8. Start the capture and integrate jobs:



9. If the channel is replicating to a standby machine and that machine has its own hub with an identical channel running in the opposite direction, then that channel must also be adapted by repeating steps **3**, **5** and **7** on the standby machine.

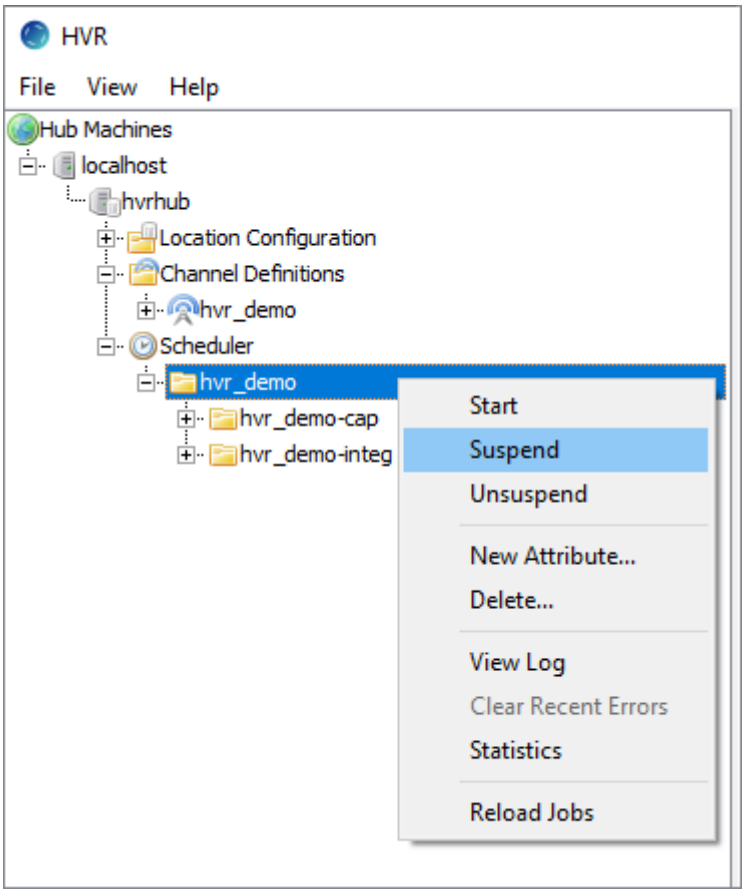
Offline Manual Adapt

Perform the following steps to manually adapt a channel using Offline Manual Adapt method:

1. Start downtime. Ensure that users cannot make changes to any of the replicated tables.

   > It is recommended to wait for the capture and integrate jobs to process all outstanding changes before performing **hvrsuspend** in the next step. If waiting is not feasible (in case long time is required for the capture and integrate jobs to process all outstanding changes), then any out of sync issues can be resolved with the **HVR Refresh** performed in step **8**.

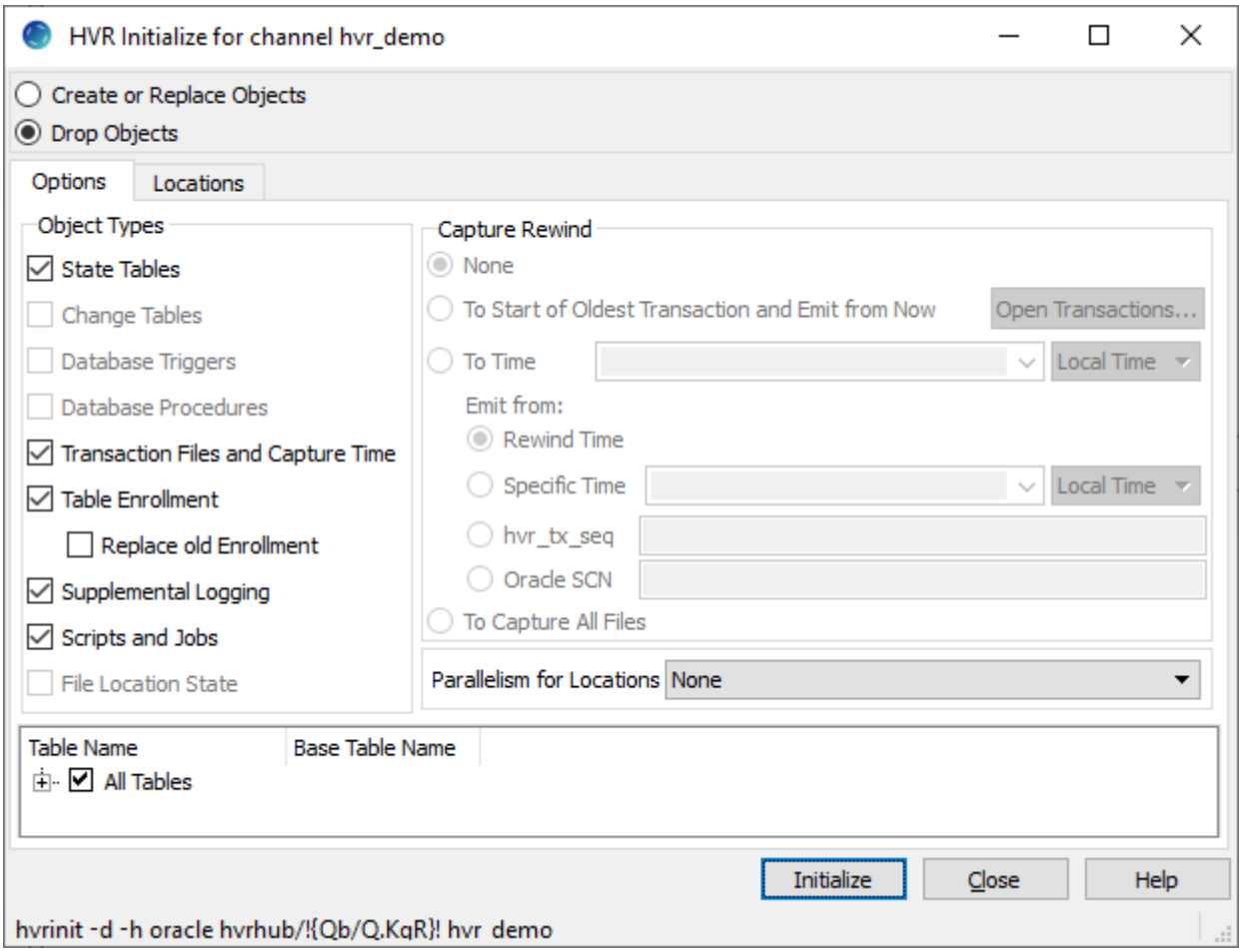
2. **Suspend** all jobs in the **HVR Scheduler**.



This can also be executed on the command line as follows:

```
hvrsuspend hubdb chn
```

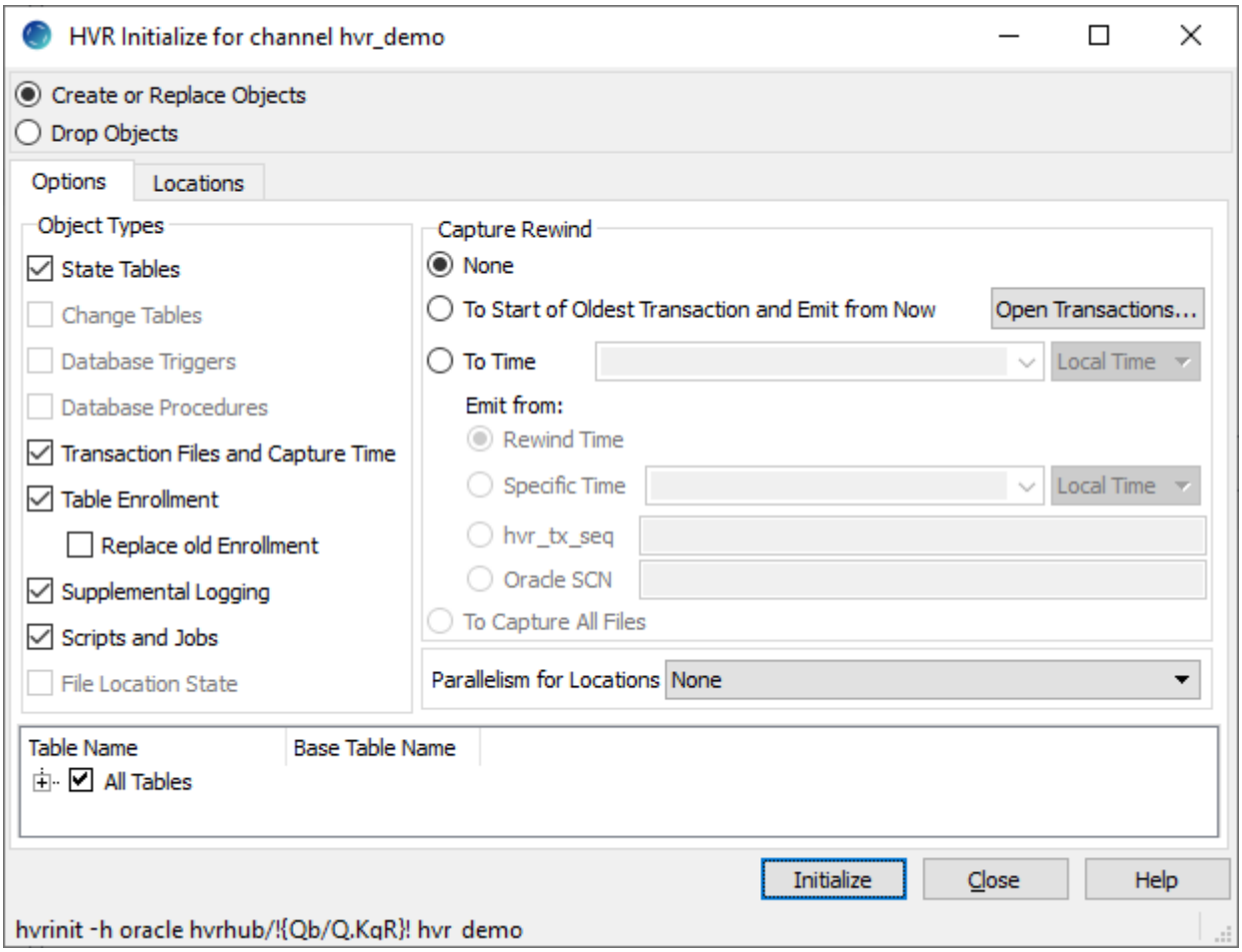
3. Execute **HVR Initialize** to deactivate the replication. Select **Drop Objects** and then select all options in **Object Types**.



This can also be executed on the command line as follows:

```
hvrinit -d hubdb chn
```

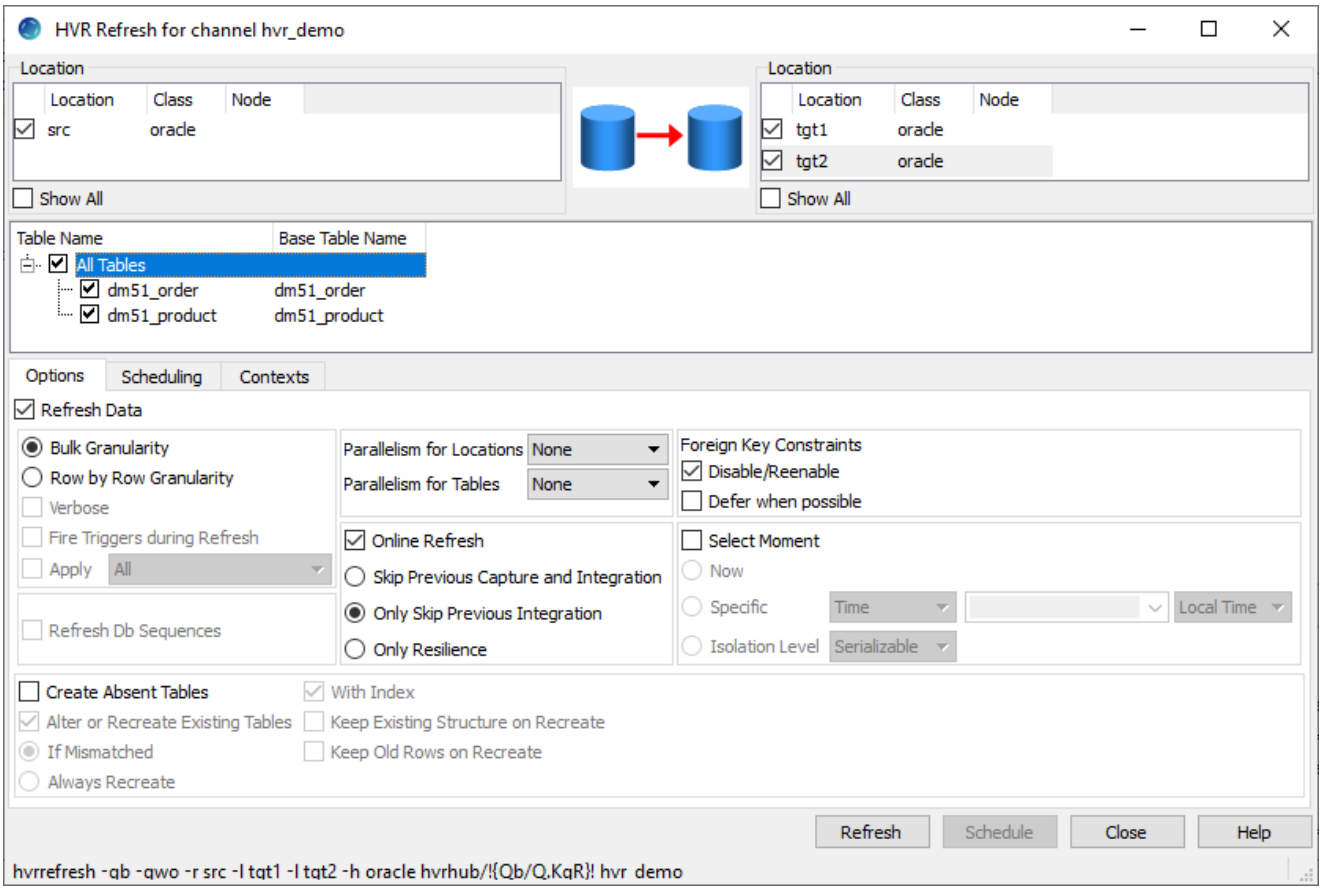
4. Run the SQL script with the DDL or DML statements against both the source and target databases.
5. Use the HVR GUI **Table Explore** connected to the captured database to incorporate the changes in the channel definition.
    a. Use **Add** to add the tables that are **Only in Database** or **In other Channel**.
    b. Use **Replace** to change the definition of the tables that have **Different Keys**, **Different Columns** or **Different Datatypes**.
    c. Use **Delete** for tables that have **Only in Channel**.
6. Use **Table Explore** to the integrate locations to check that all the tables have value **Same** in the **Match** column.
7. Execute **HVR Initialize** with all options in **Object Types** selected to reactivate the replication:



This can also be executed on the command line as follows:

```
hvrinit hubdb chn
```

8. Execute **HVR Refresh** to synchronize all tables that are affected by the SQL script in step **4** (except for the tables that were only dropped). This includes tables that were also affected by DML statements in this script.

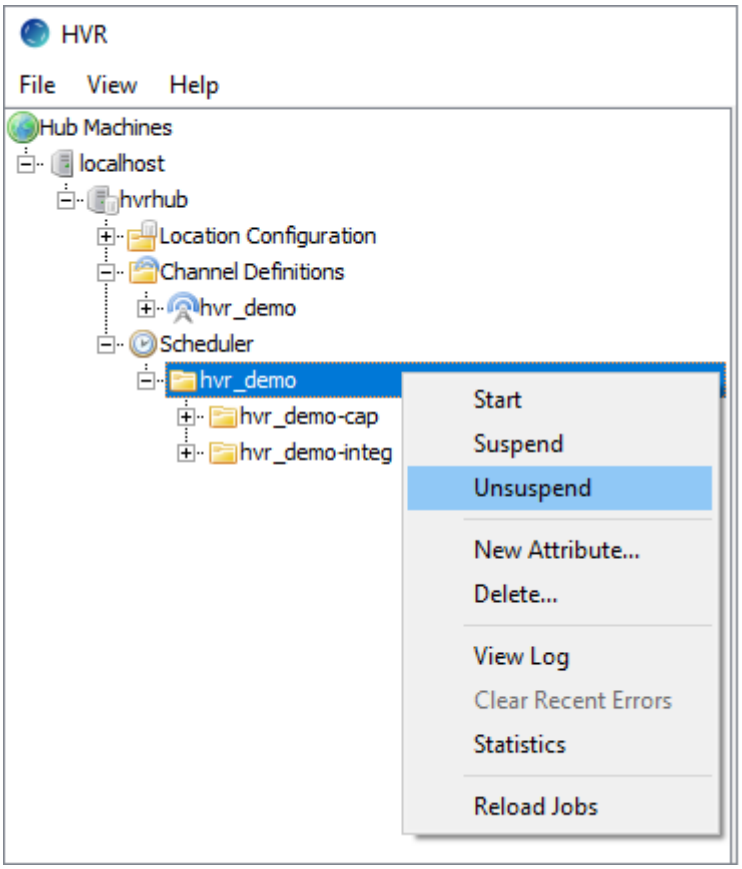



```
hvrrefresh -r src -t tbl1 -t tbl2 hubdb chn
```

The **–t** options can also just be omitted, which will cause all replicated tables to be refreshed.

> For an Ingres target database, performing bulk **refresh** (option **-gb**) will sometimes disable journaling on affected tables. If **Hvrrefresh** had displayed a warning about disabling journaling then it is necessary to execute the Ingres command **ckpdb +j** on each target database to re-enable journaling.

9. **Unsuspend** the capture and integrate jobs:



This can also be executed on the command line as follows:

```
hvrsuspend -u hubdb chn
```

10. Finish downtime.
11. If the channel is replicating to a standby machine and that machine has its own hub with an identical channel running in the opposite direction, then that channel must also be adapted by repeating steps **3**, **5**, **6**, **8** and **9** on the standby machine.

Trigger-Based Capture

Action **AdaptDDL** cannot be applied with trigger-based capture. In this case, the channel needs to be manually configured to perform trigger-based capture involving DDL statements. The steps defined for the **Offline Manual Adapt** method above are also applicable to the trigger-based capture with DDL statements involved. Note that while performing the steps for the trigger-based capture, in steps **3** and **7**, all options (in **Object Types**) should be selected in the **HVR Initialize** dialog.

Replication Transformations Between Non-Identical Tables

HVR can simply replicate between tables with an identical structure, but it can also perform transformations when it needs to replicate between tables which are not identical. Note that replication between different DBMS (e.g. between Oracle and SQL Server) does not necessarily count as a "transformation"; HVR will automatically convert between the data types as necessary.

Transformation logic can be performed during capture, inside the HVR pipeline or during integration. These transformations can be defined in HVR using different techniques:

- Declaratively, by adding special HVR actions to the channel. These declarative actions can be defined on the capture side or the integrate side. The following are examples:
  - Capture side action
    ColumnProperties /CaptureExpression="lowercase({ *colname* **})"** can be used to perform an SQL expression whenever reading from column *colname* . This SQL expression could also do a sub select (if the DBMS supports that syntax).
  - Capture side action
    Restrict /CaptureCondition="{ *colname* **}>22"** , so that HVR only captures certain changes.
  - Integrate side action
    ColumnProperties /IntegrateExpression="lowercase({ *colname* **})"** can be used to perform an SQL expression whenever writing into column *colname* . This SQL expression could also do a sub select.
  - Integrate side action
    Restrict /IntegrateCondition="{ *colname* **}>22"** , so that HVR only applies certain changes.
- Injection of blocks of business logic inside HVR's normal processing. For example, section **DbObjectGeneration** shows how a block of user supplied SQL can be injected inside the procedure which uses to update a certain table.
- Replacement of HVR's normal logic using user supplied logic. This is also called "override SQL" and is also explained in section **DbObjectGeneration** .
- Using an SQL view on the capture database. This means the transformation can be encapsulated in an SQL view, which HVR then replicates from. See example section **DbObjectGeneration** .
- In an HVR "agent". An agent is a user supplied program which is defined in the channel and is then scheduled by HVR inside its capture or integration jobs. See section **AgentPlugin** .

HVR does not only apply these transformations during replication (capture and integration). It also applies these transformations when doing a compare or refresh between the source and target databases.

Replication with File Locations

Contents

This section describes how to set up and use files as a source or target in HVR replication.

File Replication Scenarios

HVR supports three types of managed file transfers:

- File-to-File Replication
- File-to-Database Replication
- Database-to-File Replication

File-to-File Replication

An HVR file-to-file transfer will copy files from one source file location to one or more target file locations. A file location is a directory or a tree of directories, which can either be accessed through the local file system (Unix, Linux or Windows) or a network file protocol (FTP, FTPS, SFTP or WebDAV). Files can be copied or moved. In the latter case, the files on the source location are deleted after they have been copied to the target location.

To distribute sets of files, HVR provides the possibility to copy files selectively from the source location by matching their names to a predefined pattern. This feature also enables the routing of files within the same source location to different target locations based on their file names to enable selective file distribution scenarios.

⚠ In the file-to-file replication scenario, HVR treats each file as a sequence of bytes without making an assumption of their file format.

File-to-Database Replication

In a file-to-database transfer, data will be read from files in the source file location and replicated into one or more target databases. The source files are by default expected to be in a specific HVR XML format, which contains the table information required to determine to which tables and rows the changes should be written in the target database. It is also possible to use other input file formats by including an additional transformation step in the file capture.

Database-to-File Replication

In a database-to-file transfer, the data is read from a source database and copied into one or more files on the source file location. By default, the resulting files are in the HVR XML format preserving the table information. However, CSV is also supported out-of-the-box and other file formats can be created by including an additional transformation command definition in the file output. As in the continuous database replication between databases, it is possible to select specific tables and rows from the source database and convert names and column values.

⚠ In the file-to-database and database-to-file replication scenarios, CSV and XML are supported both for **Capture** and **Integrate**, and Avro, JSON, Parquet are only supported for **Integrate**.

Supported File Locations

HVR supports the following file storage locations: Azure BlobFS, Azure Data Lake Store, FTP, SFTP, SharePoint WebDAV, HDFS, and S3.

For the requirements, access privileges, and other features of HVR when using one of the listed locations, see the corresponding requirements pages:

- Requirements for Azure BlobFS
- Requirements for Azure Data Lake Store
- Requirements for FTP, SFTP, and SharePoint WebDAV
- Requirements for HDFS
- Requirements for S3

Location Connection

The locations can be local or remote. A local location is just a directory or a tree of directories on your file system.

There are two ways to connect to a remote location that can be used simultaneously:

1. Connect to a remote HVR agent through HVR's built-in protocol.
2. Connect through an external protocol (e.g. FTP, SFTP, WebDAV, HDFS or S3). For more information, see sections Requirements for FTP, SFTP, and SharePoint WebDAV, Requirements for HDFS, Requirements for S3. The advantage of using these protocols is the monitoring and managing capabilities HVR provides.

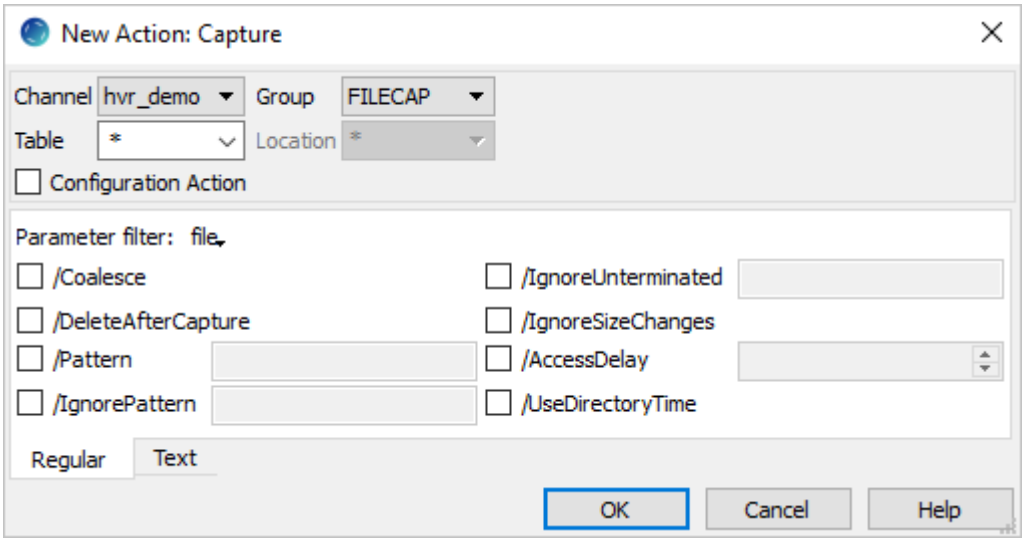
Channel Configuration

There are two types of channels that can be configured for file replication scenarios:

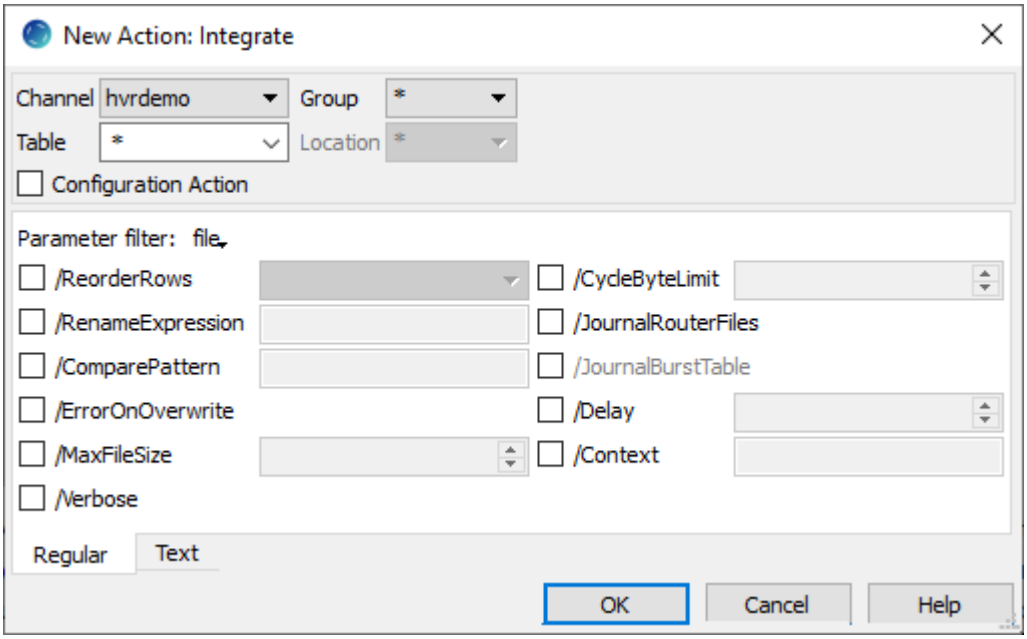
1. A channel containing only file locations (with no table information). In this case, HVR handles captured files as 'blobs' (a stream of bytes). Blobs can be of any format and can be integrated into any file locations. If only actions **Capture** and **Integrate** (no parameters) are defined for a file-to-file channel, then all files in the source directory (including files in sub-directories) are replicated to the target directory. The original files are not deleted from the source directory and the original file names and sub-directories are preserved in the target directory. New and changed files are replicated, but empty sub-directories and file deletions are not replicated.
2. A channel containing a file location as a source and a database table as a target or vice versa. HVR interprets each file as containing database changes in XML, CSV, Avro, JSON or Parquet formats. The default format for file locations is HVR's own XML format. In this case, HVR can manage data in the files.

Replication Options

Action **Capture** is sufficient to start replication, it instructs HVR to capture files from the source location. However, you can configure the capture behavior according to your needs by setting certain options **Capture**. For example, use **/DeleteAfterCapture** option to move files instead of copying them. The **/Pattern** and **/IgnorePattern** options control which files are captured and/or ignored during replication: you can specify to capture all files with the ***.xml** extension and ignore all files with ***tmp*** in their name. More powerful expressions are supported by HVR. For more options, see section **Capture**.



Action **Integrate** should be defined for the target location and it is sufficient to commence file transfer. However, as with action **Capture**, you can configure several options to impart specific behavior during integration. For example, the parameter **/ErrorOnOverwrite** controls whether overwrites are allowed or not. Overwrites usually happen when a source file is being altered and HVR has to transfer it. The parameter **/RenameExpression** allows you to rename files using regular expressions (e.g. **{hvr_op}**) . The **{hvr_op}** expression adds an operation field enabling deletes to be written as well, which is useful in database/file transactions. The parameter **/MaxFileSize** can be used on structured files to bundling rows in a file (split files). For more options, see section **Integrate**.



File Transformation

HVR supports a number of different built-in transformation mechanisms that are applied when data is captured from a source and before it is integrated into a target:

- Soft deletes (introduction of a logical delete column, which indicates whether a row was deleted on a source database)
- Transforming XML from/into CSV
- Tokenize (calling an external token service to encrypt values)
- File2Column (loading a file into a database column)

For more information, see section **Transform**.

Replicating CSV file to Database Table

This example describes how to replicate a CSV file from a source location to a database table on a target location using HVR.

Prerequisites

1. A CSV file resides on your local machine. Contents of the CSV file are as below:

	A	B
1	c1	c2
2	1	ABC
3	2	xyz
4	3	PQR
5	4	ghi
6	5	def
7		

2. Create a table in a target database, for example, as follows:

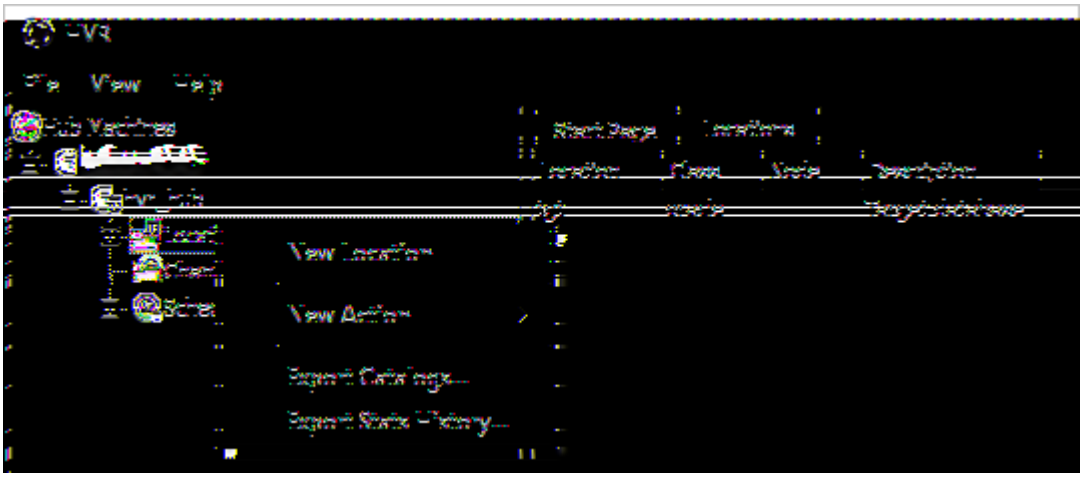
```
SQL > create table test_csv (c1 int, c2 varchar2(20));
```

⚠ The base name of the columns in the database should match the columns in the CSV file.

Replication Steps

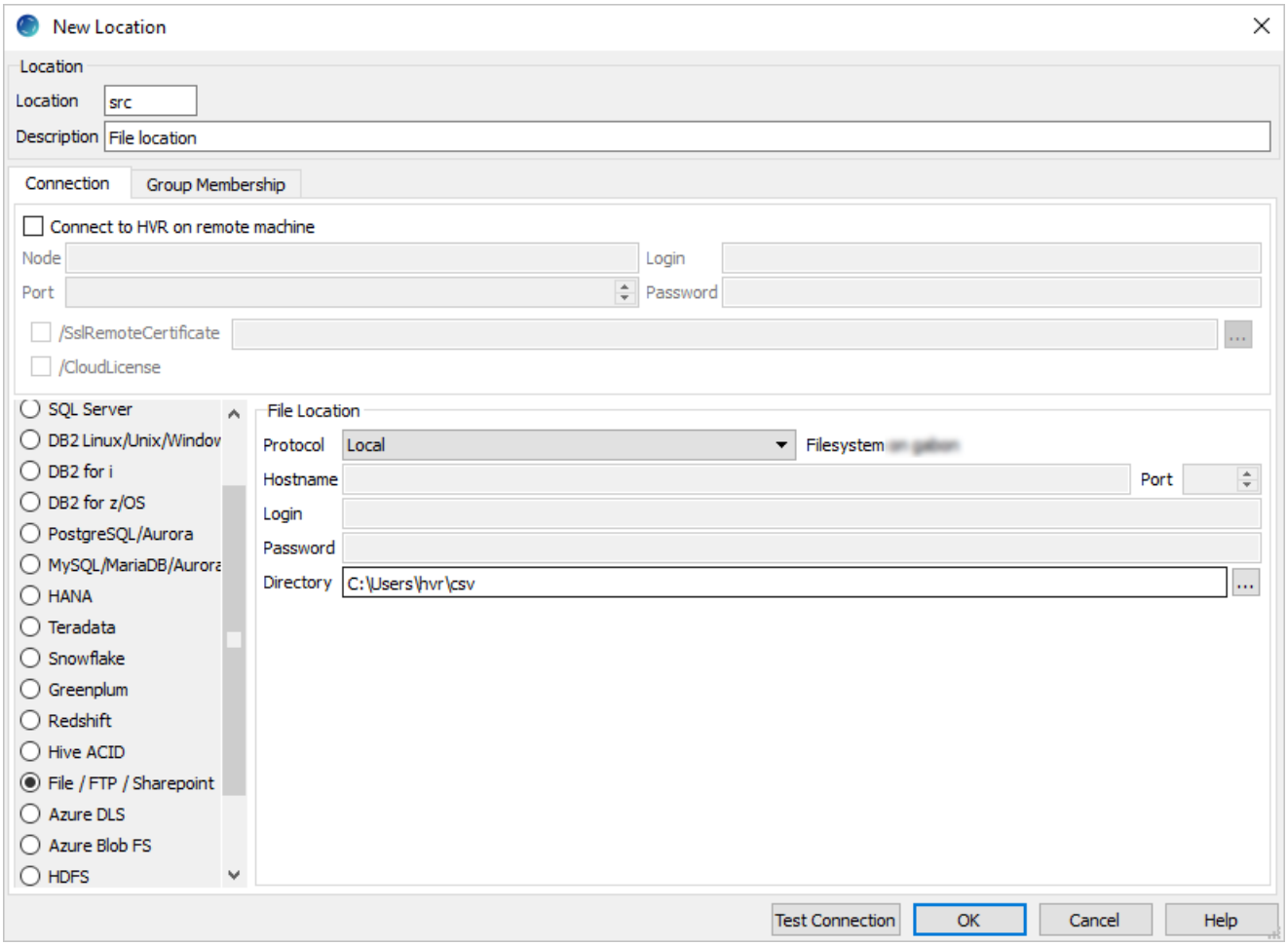
For the purpose of this example, it is assumed that the hub database already exists and a target database location **tgt** has been configured. For more information on how to register the HVR Hub and set up location configuration, see, for example, section Quick Start for HVR - Oracle.

1. Next, add a file directory to the source location: in HVR GUI, right-click **Location Configuration** and select **New Location**.

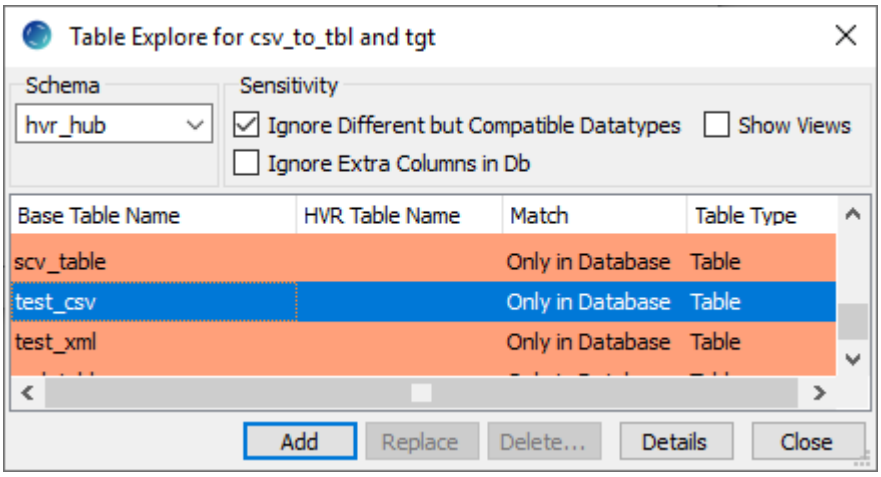


2. In the **New Location** window, specify the location name **src**.
3. In the **Class** pane, select **File/FTP/Sharepoint**.

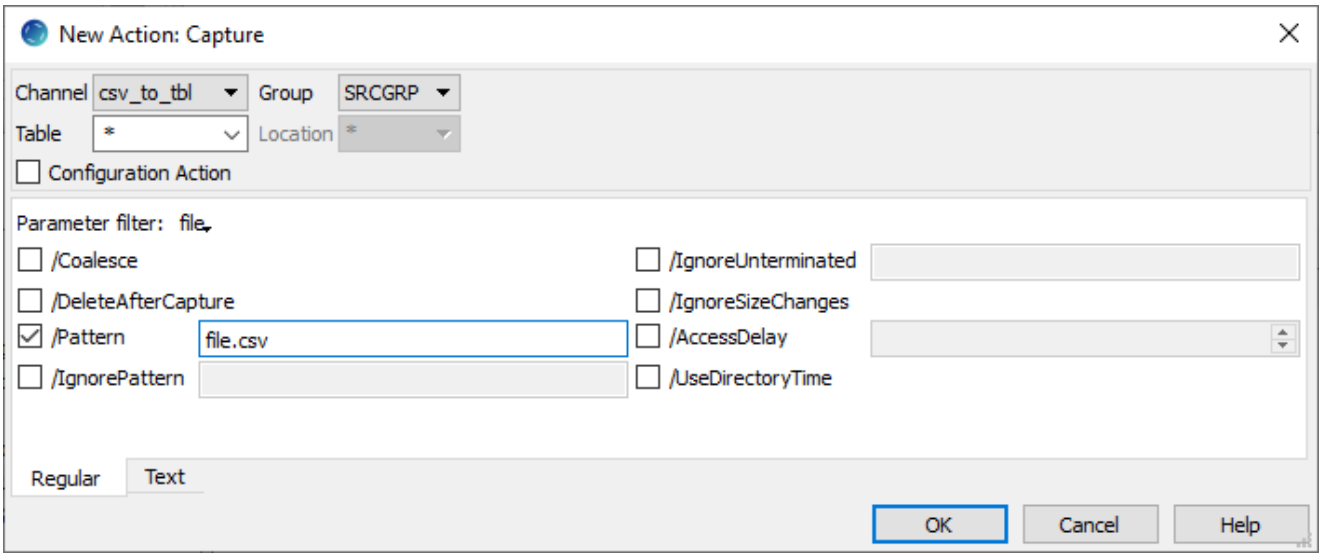
4. In this case, the XML file resides on a local machine, thus, select **Local** in the **Protocol** field. Alternatively, connect to a remote file location using one of the available file protocols (e.g. FTP, SFTP or WebDAV).
5. In the **Directory** field, select the directory, in which the CSV file will be later uploaded.



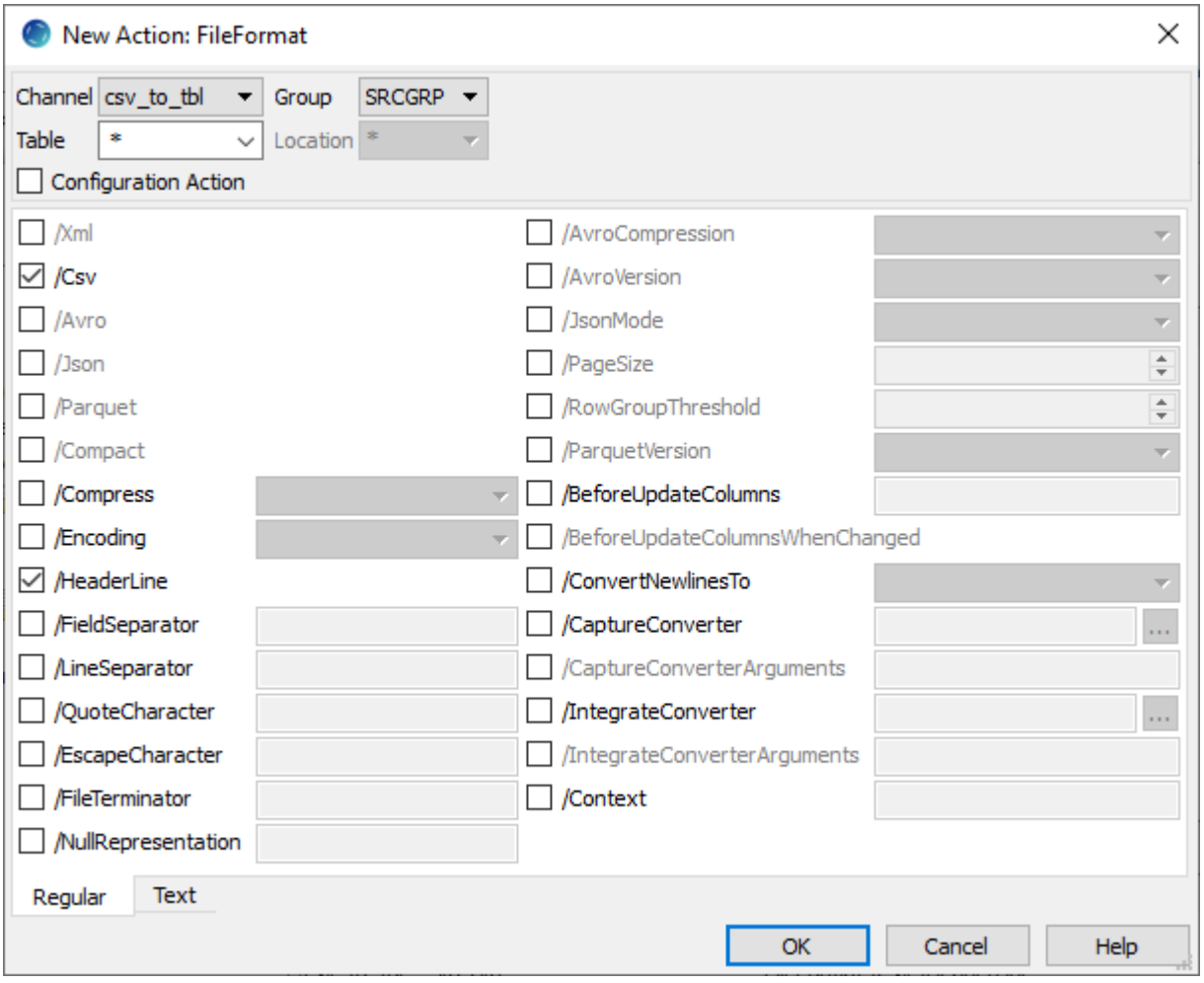
6. Click **Test Connection** to verify the connection and then click **OK**.
7. Now, you need to create a channel: right-click **Channel Definitions** and select **New Channel**. Specify the channel name, e.g. **csv_to_tbl**, and click **OK**.
8. Expand the channel node, right-click **Location Groups** and select **New Group**. Specify the name of the source group, e.g. **SRCGRP**, select **src** under the **Group Membership**.
9. Right-click **Location Groups** and select **New Group**. Specify the name of the target group, e.g. **TGTGRP**, select **tgt** under the **Group Membership**.
10. Right-click the **Tables** node, select **Table Explorer**. In the **Table Explorer** window, click **Connect**. Select table **test_csv** and click **Add**. Click **OK** in the **HVR Table Name** window and close the **Table Explorer** window.



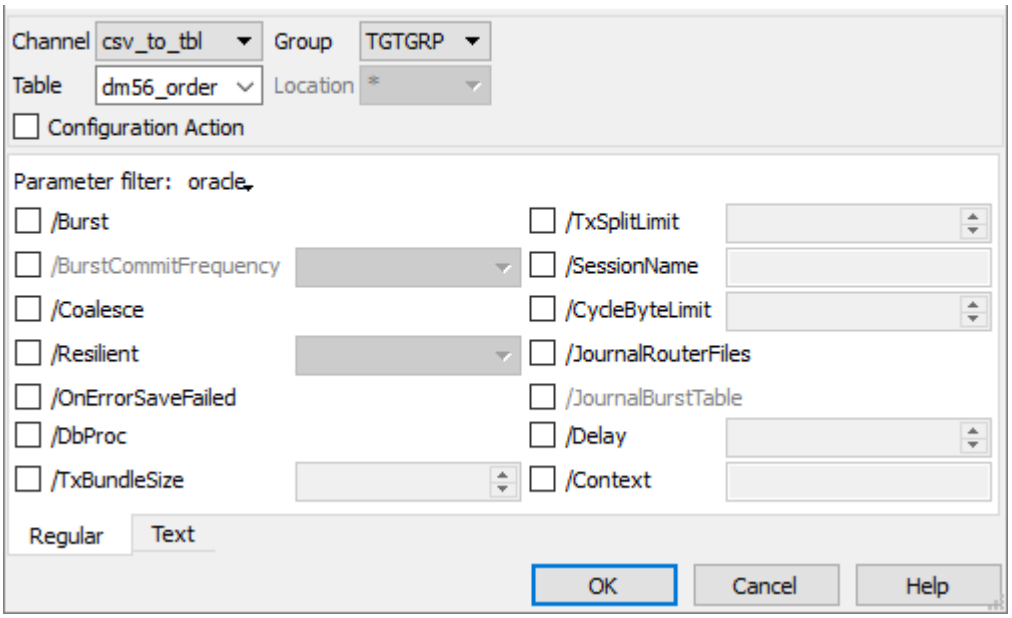
11. Add the **Capture** action to the source group: right-click source group **SRCGRP**, select **New Action Capture**.
12. In the **New Action: Capture** window, select **Pattern** and specify the name of the CSV file. This parameter controls to capture only the file that meets certain pattern. For example, if you specify '*.csv', HVR will capture all CVS files. Click **OK**.



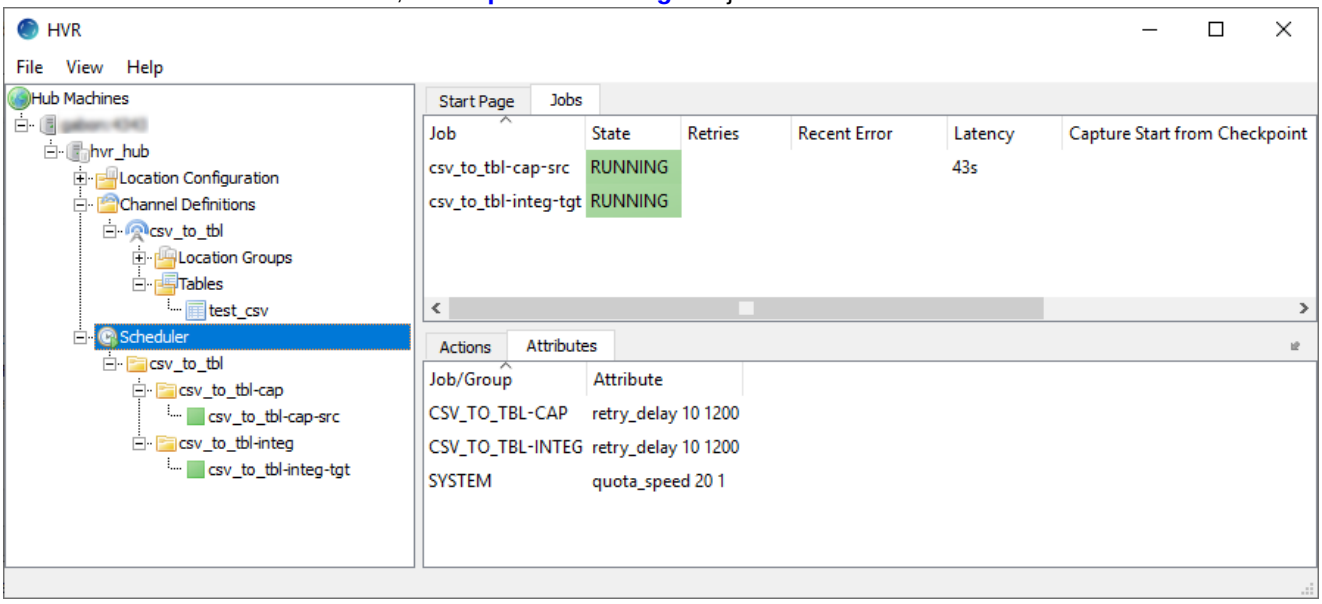
13. Right-click source group **SRCGRP**, select **New Action FileFormat**.
14. In the **New Action: FileFormat** window, select options **/Csv** and **/HeaderLine** and click **OK**.



15. Add the **Integrate** action to the target group: right-click target group **TGTGRP**, select **New Action Integrate**.
16. In the **New Action: Integrate** window, select the required table and click **OK**.



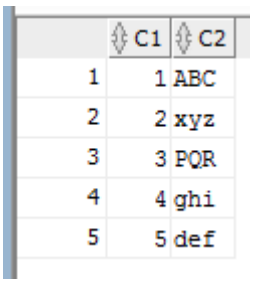
17. Now that the channel is configured, right-click the channel and select **HVR Initialize**. In the **HVR Initialize** window, click **Initialize**. HVR will create two replications jobs under the **Scheduler** node.
18. Right-click the **Scheduler** node and select **Start**.
19. Right-click the **Scheduler** node, navigate to **All Jobs in System** and click **Start**.
20. As shown in the screenshot below, the **Capture** and **Integrate** jobs are in the **RUNNING** state.



Test Replication

Add the CSV file to the directory defined on the source location. Right-click the **Scheduler** node and select **View Log** that will display the output of the **Capture** and **Integrate** jobs running.

If you look at the table, you will see that it got populated with the corresponding values from the CSV file.



	C1	C2
1	1	ABC
2	2	xyz
3	3	PQR
4	4	ghi
5	5	def

Replicating XML file to Database Table

This example describes how to move an XML file from a source location to a database table on a target location using HVR.

Prerequisites

1. Create a table in a target database, for example, as follows:

```
SQL > create table test_xml (c1 int, c2 varchar2(20));
```

2. Create an XML file with the HVR's XML format, for example, as shown below:

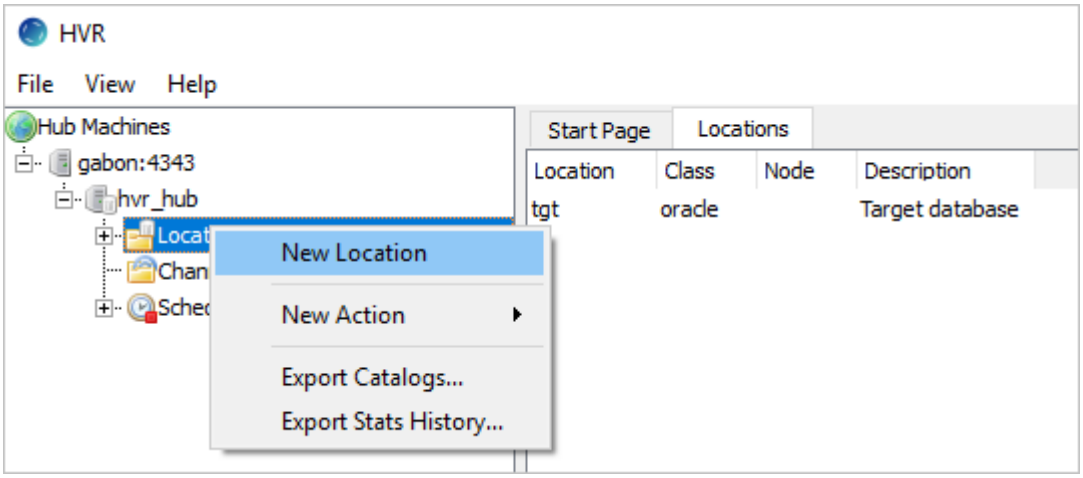
```
<?xml version=”1.0 encoding=”UTF-8?>

<!DOCTYPE hvr SYSTEM “lib/hvr.dtd”>

<hvr version=”1.0>

<table name=”test_xml>

<row>

<column name=”c1>1</column>

<column name=”c2>Hello</column>

</row>

<row>

<column name=”c1>2</column>

<column name=”c2>Hello World</column>

</row>

</table>

</hvr>
```

   Here, **c1** and **c2** match the base names of the columns and **test_xml** matches the table base name in the target database.

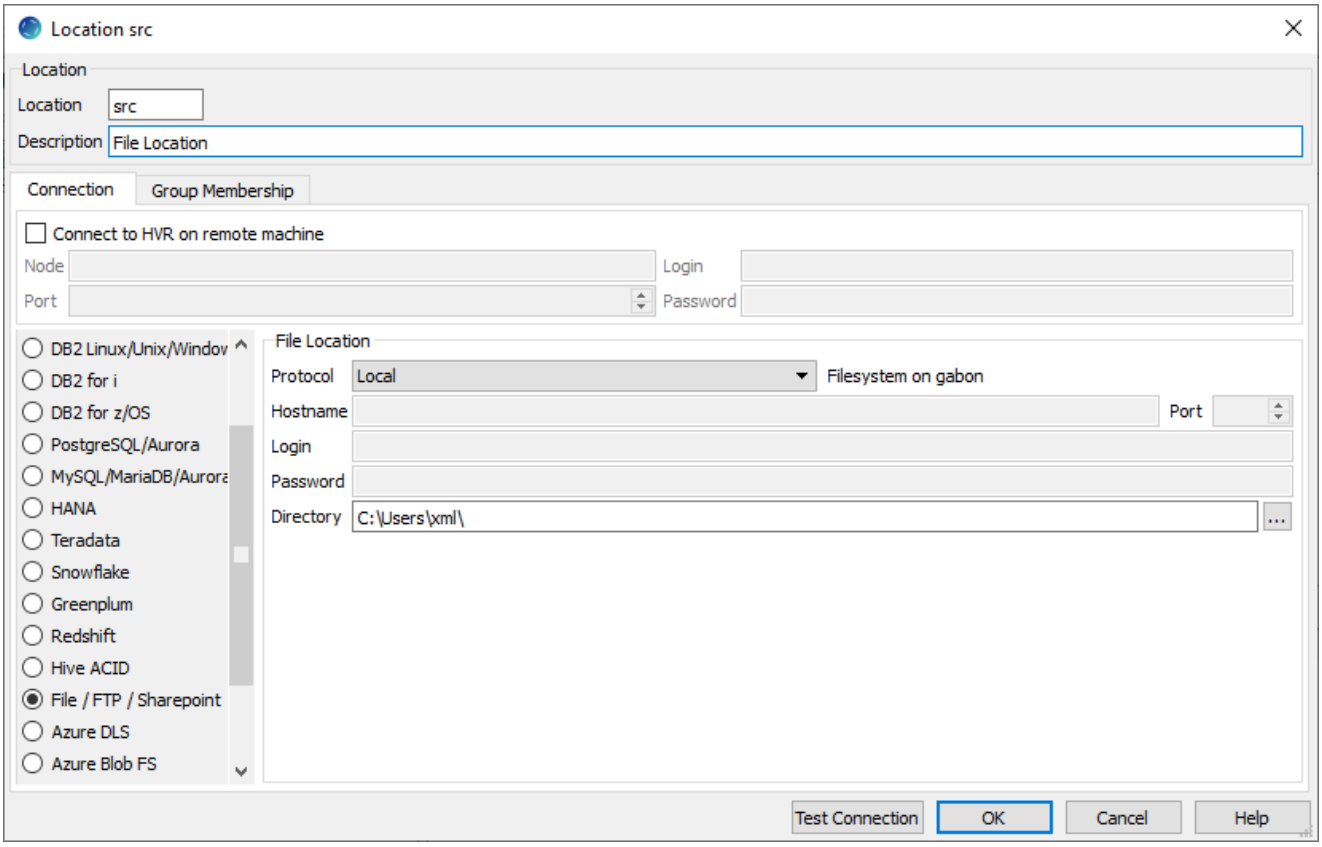
Configuration Steps

For the purpose of this example, it is assumed that the hub database already exists and a target database location **tgt** has been configured. For more information on how to register the HVR Hub and create locations, see, for example, section Quick Start for HVR - Oracle.

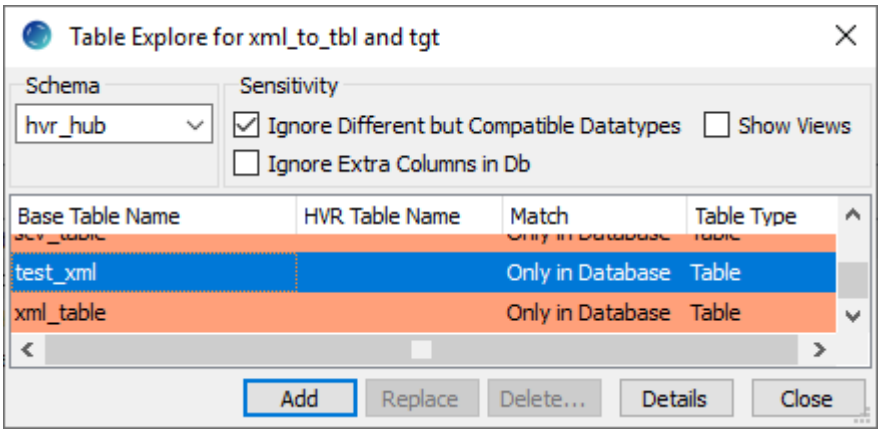
1. Next, add a file directory to the source location: in HVR GUI, right-click **Location Configuration** and select **New Location**.



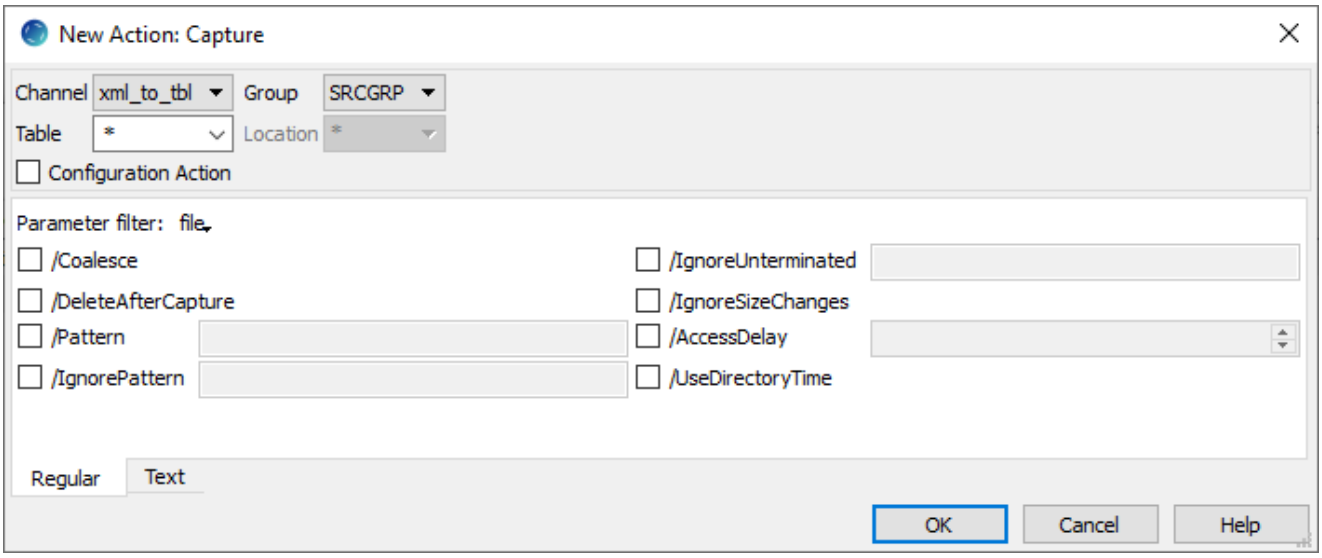
2. In the **New Location** window, specify the location name **src**.
3. In the **Class** pane, select **File/FTP/Sharepoint**.
4. In this case, the XML file resides on a local machine, thus, select **Local** in the **Protocol** field. Alternatively, connect to a remote file location using one of the available file protocols (e.g. FTP, SFTP or WebDAV).
5. In the **Directory** field, select the directory, in which the XML file will be uploaded.



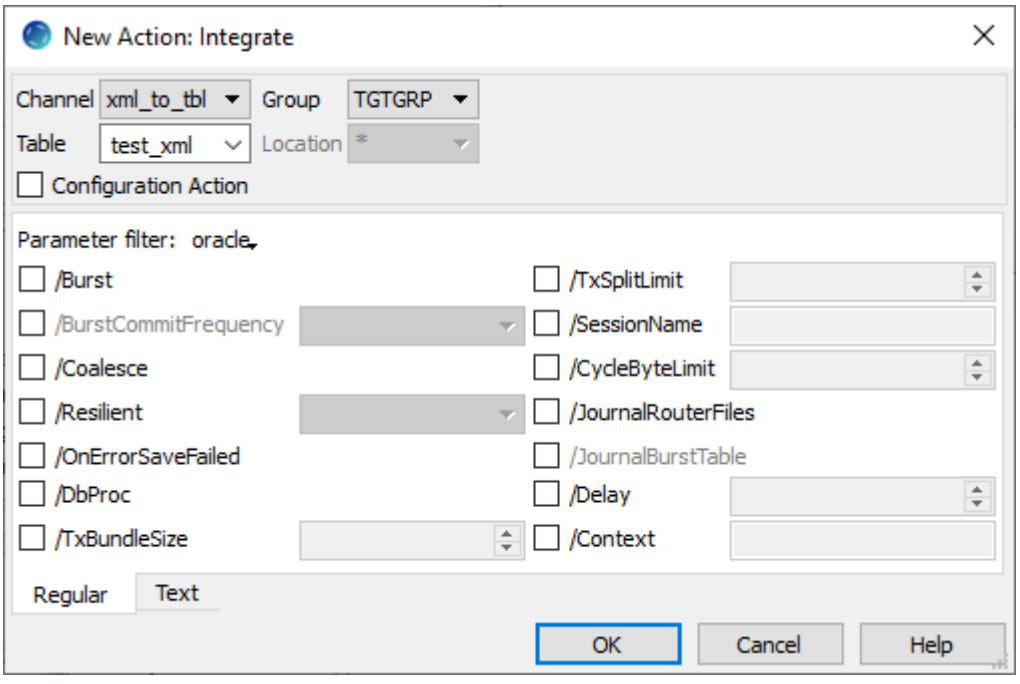
6. Click **Test Connection** to verify the connection and then click **OK**.
7. Now, you need to create a channel: right-click **Channel Definitions** and select **New Channel**. Specify the channel name, e.g. **xml_to_tbl**, and click **OK**.
8. Expand the channel node, right-click **Location Groups** and select **New Group**. Specify the name of the source group, e.g. **SRCGRP**, select **src** under the **Group Membership**.
9. Right-click **Location Groups** and select **New Group**. Specify the name of the target group, e.g. **TGTGRP**, select **tgt** under the **Group Membership**.
10. Right-click the **Tables** node, select **Table Explorer**. In the **Table Explorer** window, click **Connect**. Select the required table, in this case, **test_xml**, and click **Add**. Click **OK** in the **HVR Table Name** window and close the **Table Explorer** window.



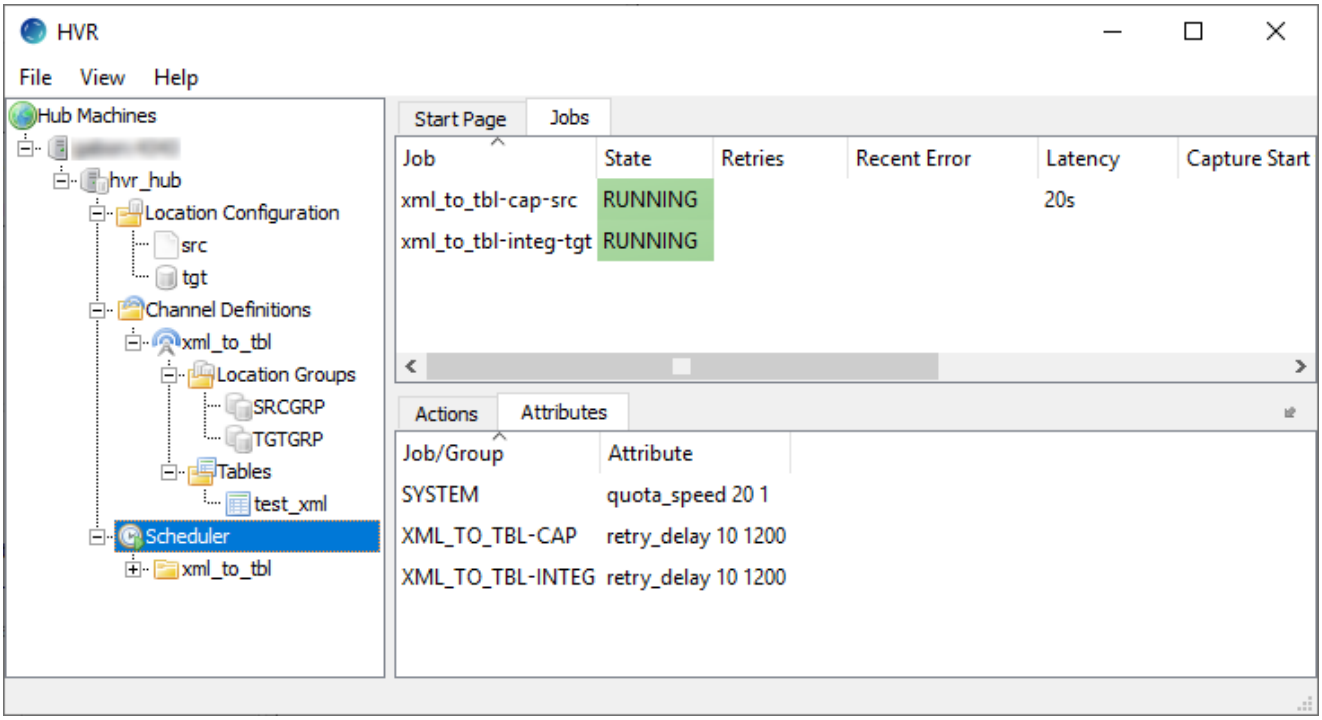
11. Add the **Capture** action to the source group: right-click source group **SRCGRP**, select **New Action Capture**. In the **New Action: Capture**, click **OK**.



12. Add the **Integrate** action to the target group: right-click target group **TGTGRP**, select **New Action Integrate**.
13. In the **New Action: Integrate** window, select table **test_xml** and click **OK**.



14. Now that the channel is configured, right-click the channel and select **HVR Initialize**. In the **HVR Initialize** window, click **Initialize**. HVR will create two replications jobs under the **Scheduler** node.
15. Right-click the **Scheduler** node and select **Start**.
16. Right-click the **Scheduler** node, navigate to **All Jobs in System** and click **Start**.
17. As shown in the screenshot below, the **Capture** and **Integrate** jobs are in the **RUNNING** state.



Test Replication

Add the XML file to the directory defined on the source location. Right-click the **Scheduler** node and select **View Log** that will display the output of the **Capture** and **Integrate** jobs running.

If you look at the table, you will see that it got populated with the corresponding values from the XML file.

Using Contexts Variables for Comparing Data Based on Datetime Column

This section describes the steps of setting up a channel for comparing a manageable subset of data in two Oracle tables based on a DateTime column. HVR allows you to implement this behavior using action **Restrict** with the **/CompareCondition** and **/Context** parameters using context variables.

For the initial steps to set up a channel, refer to Quick Start for HVR - Oracle.

In this example, a channel is configured with source and target locations residing on Oracle databases.

To set up HVR Compare based on a DateTime column using context variables, perform the following actions:

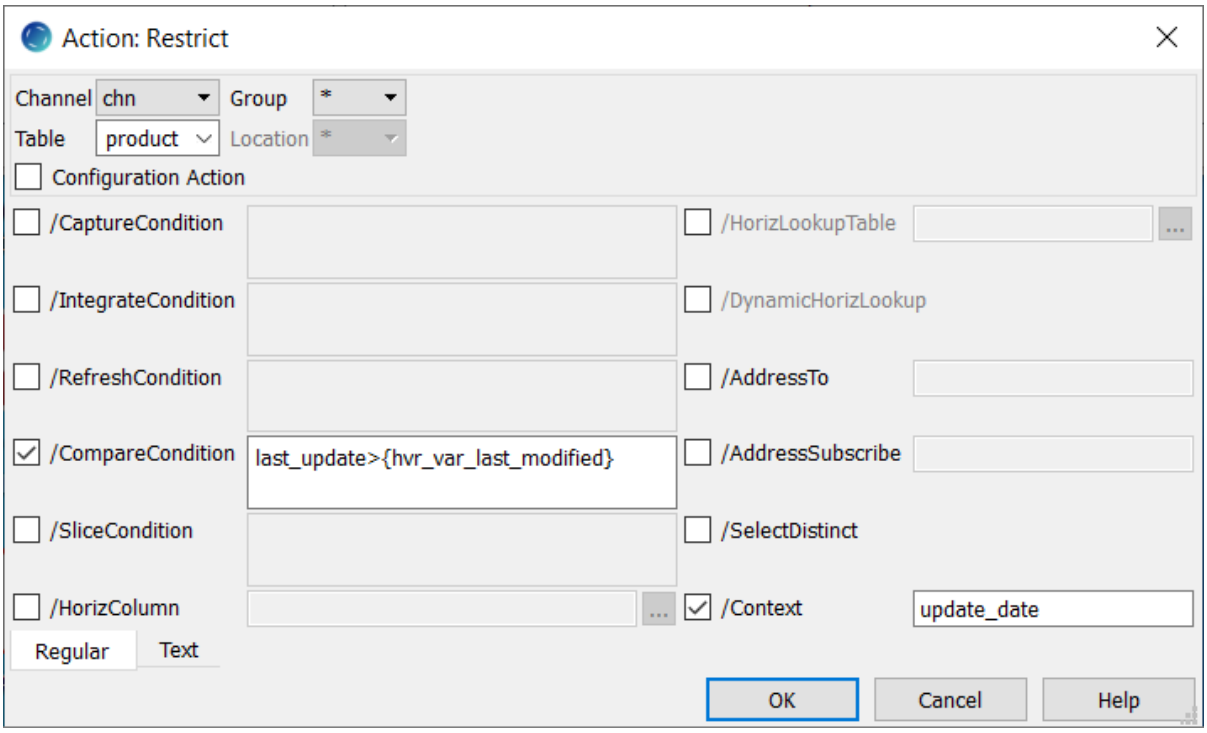
1. Define action **Restrict** with parameters **/CompareCondition** and **/Context** for both source and target locations. The compare condition allows to compare only rows that satisfy a certain condition. The condition may be defined using the following pattern **{hvr_var**_*xxx*}, where *xxx* is a value of the context variable. The **/Context** parameter allows to activate the **Restrict** action only if the context is enabled. For more information, refer to sections **/CompareCondition** and **/Context** on the **Restrict** page.

   a. In the HVR GUI, right-click **Location Groups** under the channel **chn** node, navigate to **New Action** and select **Restrict**.

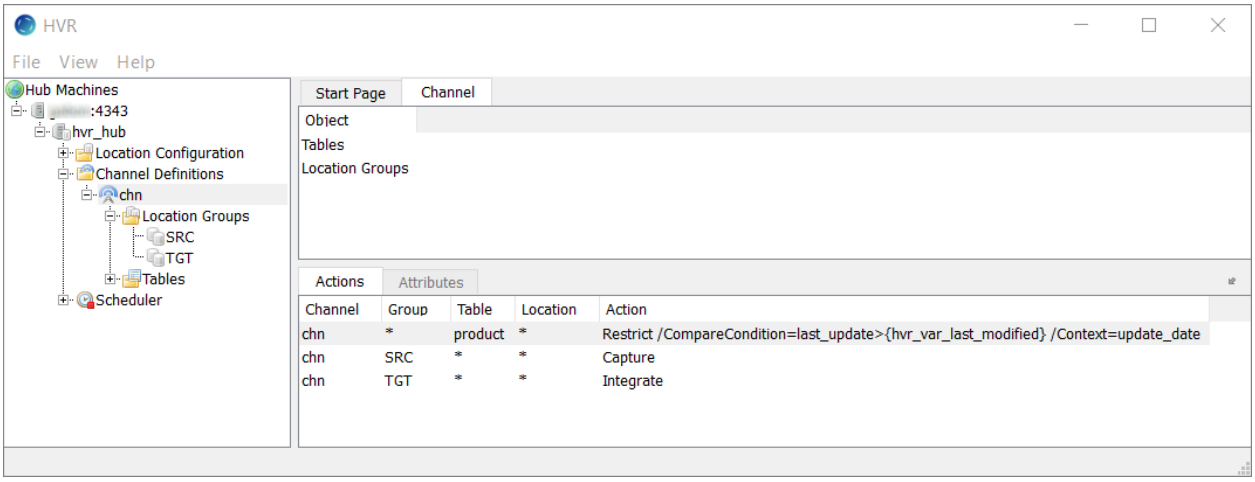
   b. Since the compare condition is defined for both source and target location groups, select '*****' in the **Group** field. Then select table 'product' in the **Table** field.

   c. Specify the following condition in the **CompareCondition** field: **last_update>{hvr_var_last_modified}**, where **last_modified** is a variable, the value for which can be defined in the **Contexts** tab of the **HVR Compare** dialog (see step 8 below). By defining different values/expressions for the variable, you can manage the subsets of data to be compared.

   b. Enter the context name, e.g. 'update_date' in the **/Context** field. Click **OK**. **HVR Compare** is effective only when the context is enabled. The context can be enabled in the **Contexts** tab of the **HVR Compare** dialog (see step 7 below).



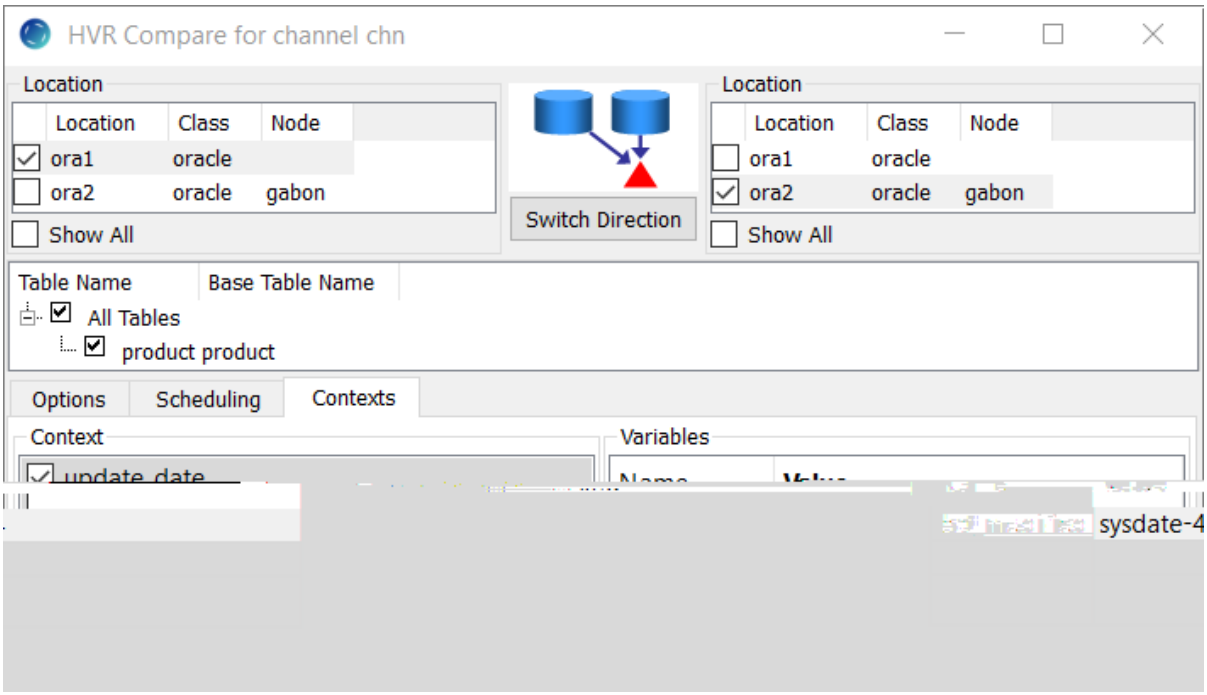
2. Additionally, define action **Capture** on source location group **SRC** and action **Integrate** on target location group **TGT**, which are mandatory for performing compare. For this, right-click source group **SRC**, navigate to **New Action** and select **Capture**. For location group **TGT**, select **Integrate**.
3. The resulting channel **chn** configuration will be as follows:



4. Set up HVR Compare: right-click channel **chn** and select **HVR Compare**. Select location **ora1** in the left **Location** pane and location **ora2** in the right **Location** pane.
5. Select table 'product' in the tree of tables below.
6. Under the **Options** tab, select the **Row by Row Granularity** compare method. Alternatively, you can select the **Bulk Granularity** compare method. For more information on the difference between the two compare methods, refer to section **Hvrcompare**.

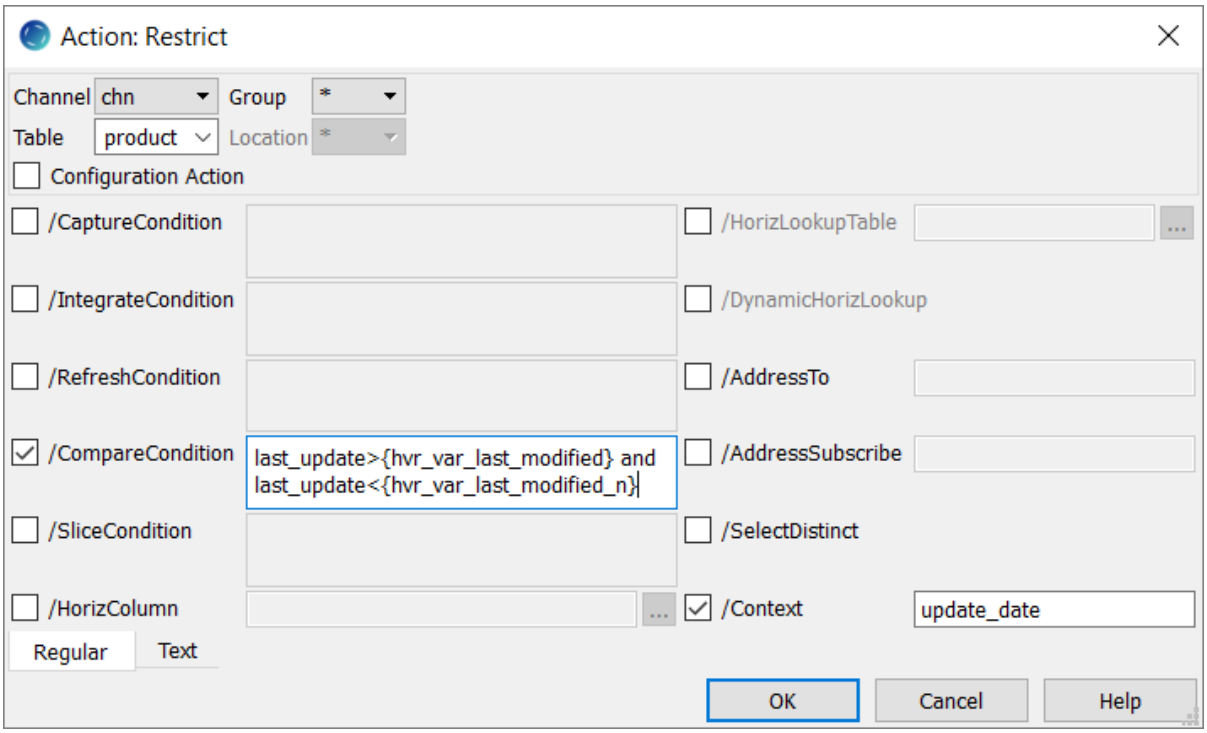
   SC-Hvr-OperationalTopics-UsingContextVariables_Compare_row_by_row
7. Click the **Contexts** tab. In the **Context** pane, select context 'update_date' that was defined earlier in the **Restrict** dialog for the **/Context** parameter.
8. In the **Variables** pane, specify value **sysdate-4** for variable **last_modified** defined on the source and target locations. Expression **sysdate-4** selects only data which is 4 days old. Click **Compare**.

> ⚠ SYSDATE is an Oracle function that returns the current date and time set for the operating system on which the database resides. For other DBMSs, the appropriate date/time functions should be used.

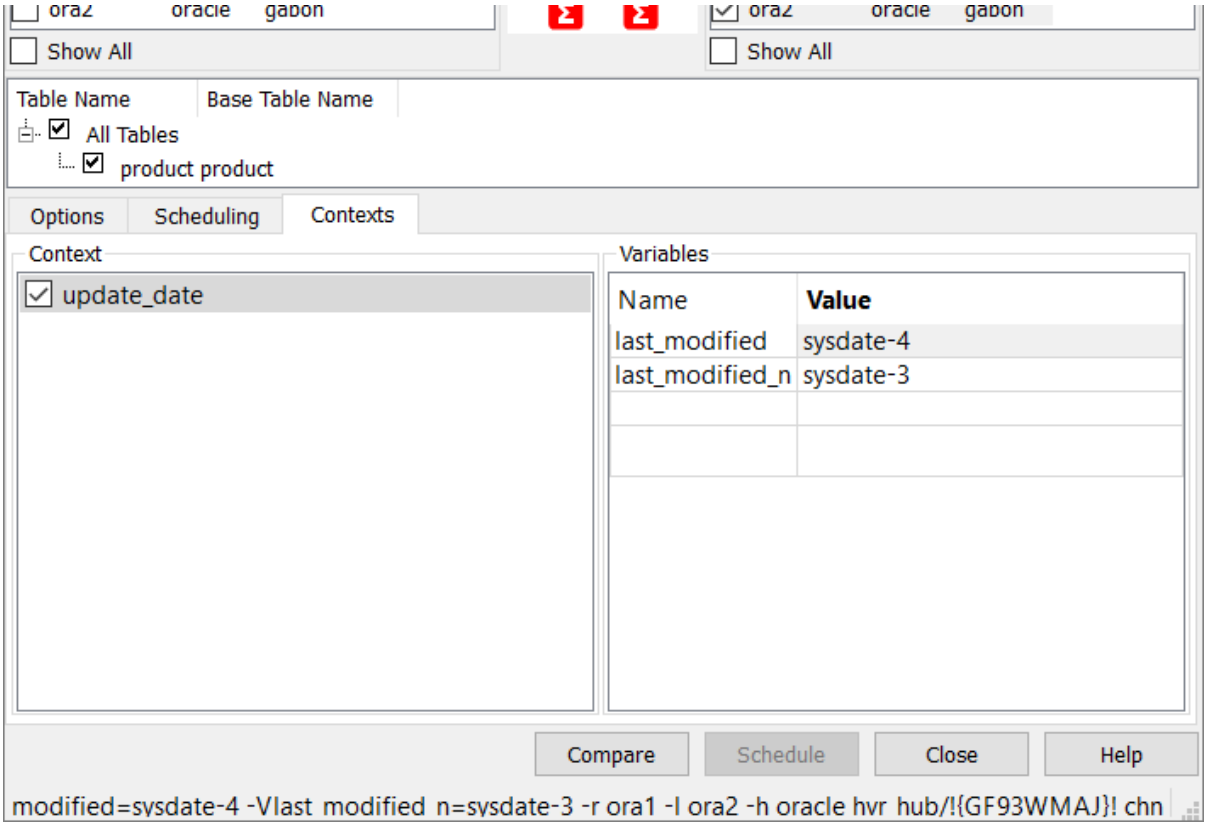


9. After the compare event is complete, the **Compare Result** dialog appears showing the comparison details.

You can change the date range for which you want to compare data in locations by specifying different values /expressions for the context variable. For example, if you want to compare data modified on a particular date, you can define the following compare condition for source and target:



In this case, the context expressions are defined as follows:



Internal HVR Objects

This section describes the database objects HVR uses inside its hub database and also the objects it create inside the replicated databases.

Catalog Tables

Contents

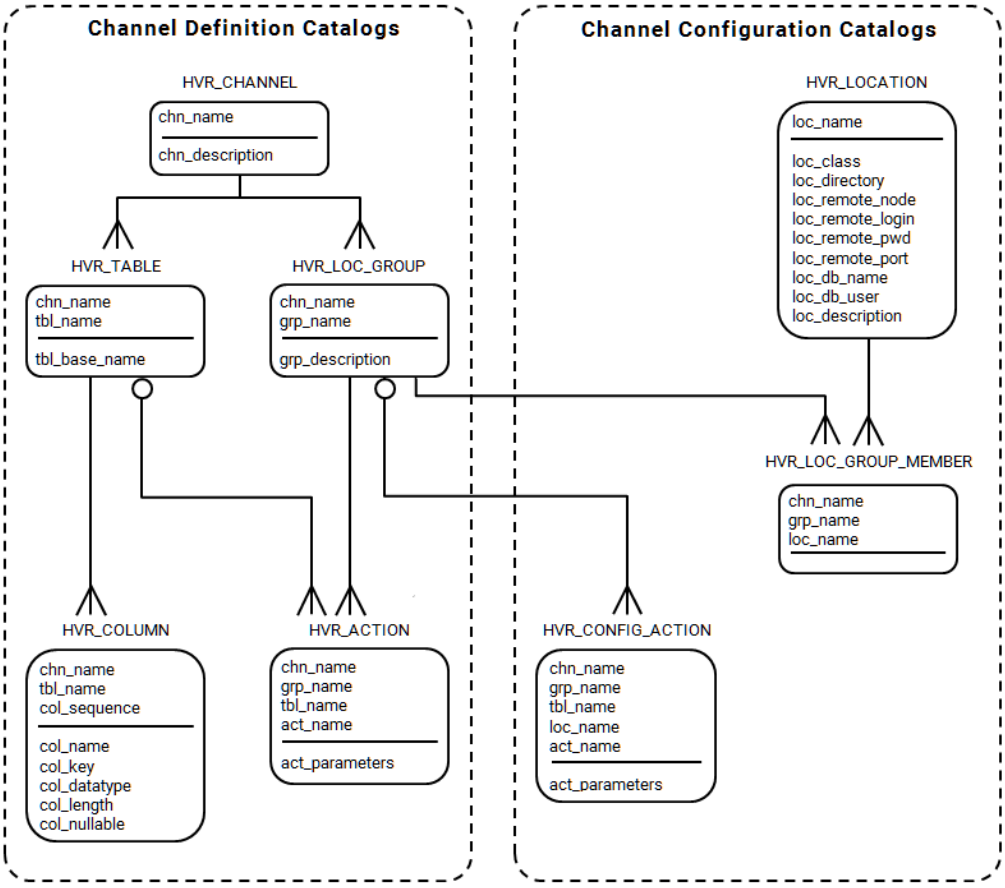
The catalog tables are tables inside the hub database that contain a repository for information about what must be replicated. They are normally edited using the HVR GUI.

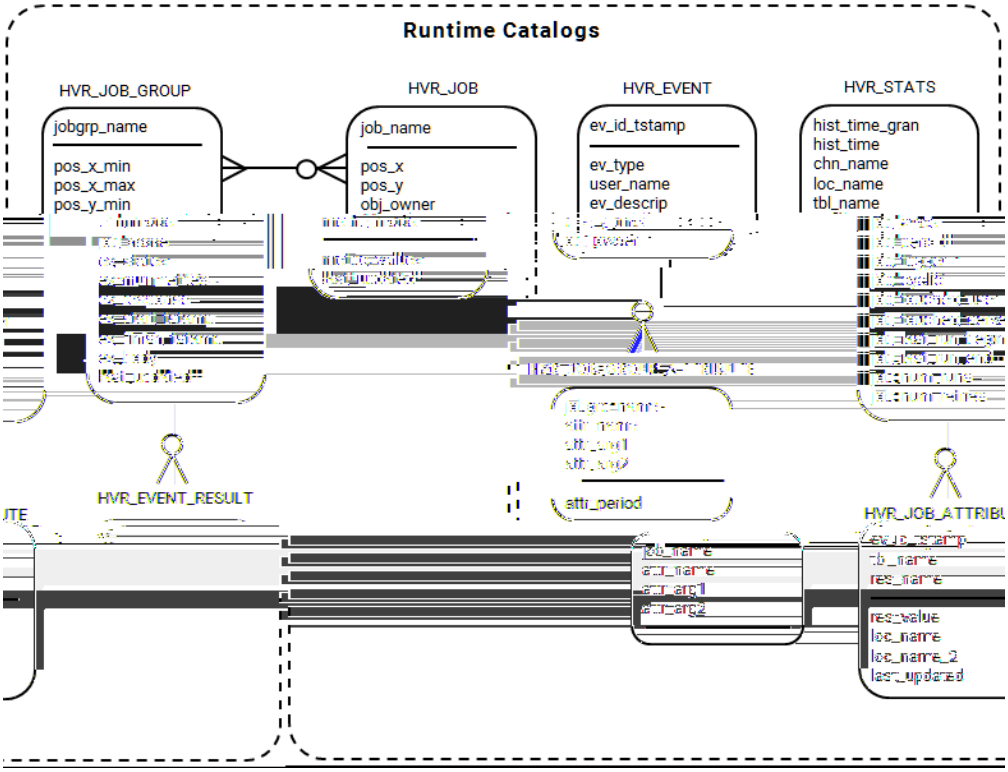
The HVR catalogs are divided into channel definition information (delivered by the developer) and location configuration information (maintained by the operator or the DBA). The HVR Scheduler catalogs hold the current state of scheduling; operators can control jobs by directly inserting, updating and deleting rows of these catalogs.

Hub database also contain few catalog tables that are used internally by HVR. Following are the internal catalog tables available in hub database:

- HVR_COUNTER
- HVR_JOB_RESOURCE
- HVR_JOB_RESOURCE_ATTRIBUTE
- HVR_JOB_PARAM
- HVR_STATS_STAGING

These crucial tables should not be modified/deleted manually. Modifying/deleting these tables without proper guidance from HVR's Technical Support can lead to disruption or data loss during replication.





HVR_CHANNEL

Column	Data type	Optional?	Description
chn_name	String 12 characters	No	Unique name for channel. Used as the parameter by most HVR commands, and also as a component for naming jobs, database objects and files. For example, an HVR capture job is named *chn*–**cap**–*loc* . Must be a lowercase identifier containing only alphanumerics and underscores. Because this value occurs so often in every logfile, program, database etc. it is recommenced that this name be kept as small and concise as possible. Values **hvr**_* and **system** are reserved.
chn_description	String 200 characters	Yes	Description of channel.

HVR_TABLE

Column	Data type	Optional?	Description
chn_name	String 12 characters	No	Name of channel to which this table belongs. Each table name therefore belongs to a single channel.
tbl_name	String 124 characters	No	Replication name for table. Typically this is the same as the name of the table in the database location, but it could differ. For example if the table's database name is too long or is not an identifier. It must be a lowercase identifier; an alphabetic followed by alphanumerics and underscores.
tbl_base_name	String 128 characters	Yes	Name of database table to which this replication table refers. If the table has different names in different databases then the specific value can also be set with action **TableProperties /BaseName** .

HVR_COLUMN

Column	Data type	Optional?	Description
chn_name	string 12 characters	No	Channel name.
tbl_name	string 124 characters	No	Table name.
col_sequence	number	No	Sequence of column in the table.
col_name	string 128 characters	No	If the column has a different name in different databases, this value can be overridden with action **ColumnProperties /BaseName** .

col_key	string 32 characters	Yes	Is column part of table's replication key and distribution key? Possible values are: • *bool*: Value **0** means column not in replication key, whereas value 1 means it is. • *bool.bool*: First boolean indicates whether column is in replication key, second indicates whether column is in its distribution key. Replication key information is needed to replicate updates and deletes and is used to create target tables. The replication key does not have to match a primary key or physical unique index in the replicated table. If a table has no columns marked as replication keys, then by default it will assume an 'implicit' replication key that consists of all non-lob columns will give uniqueness. If this is not the case then action **TableProperties /DuplicateRows** must be defined.
col_datatype	string 128 characters	No	Data type of column. Any database type can be used here, i.e. **varchar**, **varchar2**, **char**, **integer**, **integer4**, **number** or **date**.
col_length	string 128 characters	Yes	The meaning of this column depends on the data type: • For string data types such as **binary**, **byte**, **c**, **char**, **text**, **raw**, **varchar**, **varchar2** - It indicates the maximum length of string. Different formats are possible, to distinguish between byte length and character; a single integer is interpreted as byte length. The value can also have format [*len* **byte**] [*len* **char**] [*encoding*] where *encoding* can be values like **ISO-8859-1**, **WINDOWS-1252** or **UTF-8**. • For the data types **number** and **decimal** - It indicates scale and precision. Left of the decimal point is precision and right is scale. For example, value 3.2 indicates precision 3 and scale 2. Value –5.2 indicates precision 5 and scale –2. • For other data types, it is not used.
col_nullable	number	No	Is column data type nullable? Values are **0** (indicates not nullable) or **1** (indicates nullable).

HVR_LOC_GROUP

Column	Data type	Optional?	Description
chn_name	string 12 characters	No	Name of channel to which this location group belongs.
grp_name	string 11 characters	No	Unique UPPERCASE identifiers used as name of location group. Should begin with an alphabetic and contain only alphanumerics and underscores.
grp_description	string 200 characters	Yes	Description of location group.

HVR_ACTION

Column	Data type	Optional?	Description
chn_name	string 12 characters	No	Channel affected by this action. An asterisk '*' means all channels are affected.
grp_name	string 11 characters	No	Location group affected by this action. An asterisk '*' means all location groups are affected.
tbl_name	string 124 characters	No	Table affected by this action. An asterisk '*' means all tables are affected.
act_name	string 24 characters	No	Action name. See also section Action Reference for available actions and their parameters.

act_parameters	string 1000 characters	Yes	Each action has a list of parameters which change that action's behavior. Each parameter must be preceded by a '/'. If an action takes an argument it is given in the form */Param=arg*. Arguments that contain non–alphanumeric characters should be enclosed in double quotes (""). If an action needs multiple parameters they should be separated by a blank. For example action **Restrict** can have the following value for this column: **/CaptureCondition="{a}>3"**.

HVR_LOCATION

Column	Data type	Optional?	Description
loc_name	string 5 characters	No	A short name for each location. Used as a part of name of generated HVR objects as well as being used as an argument in various commands. A lowercase identifier composed of alphanumerics but may not contain underscores. Example: the location database in Amsterdam could be **ams**.
loc_class	string 10 characters	No	Class of location. Valid values are: • **oracle** : Oracle database. • **ingres** : Ingres database. • **sqlserver** : Microsoft SQL Server database. • **db2** : IBM DB2 database for Linux, UniDation.

chn_name	String 12 characters	No	Channel name for location group.
grp_name	String 11 characters	No	Name of location group defined in catalog **hvr_loc_group**.
loc_name	String 5 characters	No	Location belonging to this location group.

HVR_CONFIG_ACTION

Column	Data type	Optional?	Description
chn_name	string 12 characters	No	Channel affected by this action. An asterisk '*' means all channels are affected.
grp_name	string 11 characters	No	Location group affected by this action. An asterisk '*' means all location groups are affected.
tbl_name	string 124 characters	No	Table affected by this action. An asterisk '*' means all tables are affected.
loc_name	string 5 characters	No	Location affected by this action. An asterisk '*' means all locations are affected.
act_name	string 24 characters	No	Action name. See also section Action Reference for available actions and their parameters.
act_parameters	string 1000 characters	Yes	Each action has a list of parameters which change that action's behavior. Each parameter must be preceded by a '/'. If an action takes an argument it is given in the form */Param=arg*. Arguments that contain non–alphanumeric characters should be enclosed in double quotes (""). If an action needs multiple parameters they should be separated by a blank. For example action **Restrict** can have the following value in this column: **/CaptureCondition="{a}>3"**.

HVR_STATS

Since v5.5.0/1

Column	Data type	Optional?	Description
hist_time_gran	number	No	Granularity in minutes. Possible values are: • **0** : Current granularity (not historical). • **1** : Minute time granularity. • **10** : Ten (10) minutes granularity. • **60** : Hour granularity. • **1440** : Day granularity.
hist_time	number	No	Start time of measurement period as seconds since 1 Jan 1970. The length of the measurement period is equal to the value of **hist_time_gran** in minutes.
chn_name	string 12 characters	No	Channel name. An asterisk '*' means the value (sum, average, min or max) for all channels.
loc_name	string 5 characters	No	Location name. An asterisk '*' means the value (sum, average, min or max) for all locations.
tbl_name	string 124 characters	No	Table name. An asterisk '*' means the value (sum, average, min or max) for all tables.

metric_name	string 64 characters	No	Metric name.
metric_value	string 1024 characters	No	Value of metric.
metric_gatherer Since v5.6.5/11	string 4 characters	No	Name of the subsystem that gathered the metric. Values can be 'logs' (metric was gathered from the HVR log files) or 'glob' (metric was gathered from globbed router files).
metric_scope Since v5.6.5/11	string 3 characters	No	Scope of the current metric. First letter is '*' if **chn_name** is '*' and 'c' otherwise. Second letter is '*' if **loc_name** is '*' and 'l' otherwise. Third letter is '*' if **tbl_name** is '*' and 't' otherwise.
last_updated	number	No	Time when the metric was last updated, the value is in seconds since 1 Jan 1970.

HVR_JOB

Column	Data type	Optional?	Description
job_name	string 40 characters	No	Unique name of job. Case sensitive and conventionally composed of lowercase identifiers (alphanumerics and underscores) separated by hyphens. Examples: **foo** and **foo–bar**.
pos_x, pos_y	number	No	X and Y coordinates of job in job space. The coordinates of a job determines within which job groups it is contained and therefore which attributes apply.
obj_owner	string 24 characters	No	Used for authorization: only the HVR Scheduler administrator and a job's owner may change a jobs attributes or attributes.
job_state	string 10 characters	No	Valid values for cyclic jobs are **PENDING**, **RUNNING**, **HANGING**, **ALERTING**, **FAILED**, **RETRY** and **SUSPEND** are also allowed.
job_period	string 10 characters	No	Mandatory column indicating the period in which the job is currently operating. The job's period affects which job group attributes are effective. The typical value is **normal**.
job_trigger	number	Yes	0 indicates job is not triggered, 1 means it may run if successful, and 2 means it may run even if it is unsuccessful.
job_cyclic	number	Yes	0 indicates job is acyclic, and will disappear after running; 1 indicates job is cyclic.
job_touched_user	date	Yes	Last time user or **Hvrinit** (not **Hvrscheduler**) changed job tuple.
job_touched_server	date	Yes	Last time hvrscheduler changed job tuple.
job_last_run_begin	date	Yes	Last time job was started.
job_last_run_end	date	Yes	Last time job finished running.
job_num_runs	number	Yes	Number of times job has successfully run.
job_num_retries	number	Yes	Number of retries job has performed since last time job successfully ran. Reset to zero after job runs successfully.

HVR_JOB_ATTRIBUTE

Column	Data type	Optional?	Description
job_name	string 40 characters	No	Name of object on which attribute is defined.
attr_name	string 24 characters	No	Type of attribute. Case insensitive.
attr_arg1, 2	string 200 characters	Yes	Some attribute types require one or more arguments, which are supplied in these columns.

HVR_JOB_GROUP

Column	Data type	Optional?	Description
jobgrp_name	string 40 characters	No	Job group name. Case sensitive and conventionally composed of UPPERCASE identifiers (alphanumerics and underscores) separated by hyphens. Examples: **FOO** and **FOO–BAR**.
pos_x, y_min, max	number	No	These form coordinates of the job group's box in job space. Objects such as jobs, resources and other job groups whose coordinates fall within this box are contained by this job group and are affected by its attributes.
obj_owner	string 24 characters	Yes	Owner of a job group. Only a job group's owner and the HVR Scheduler administrator can make changes its coordinates or attributes.

HVR_JOB_GROUP_ATTRIBUTE

Column	Data type	Optional?	Description
jobgrp_name	string 40 characters	No	Name of job group on which attribute is defined. These also affect objects contained in job group.
attr_name	string 24 characters	No	Type of attribute. Case insensitive.
attr_arg1, 2	string 200 characters	Yes	Some attribute types require one or more arguments, which are supplied in these columns.
attr_period	string 10 characters	No	For which period does this attribute apply? Must be a lowercase identifier or an asterisks '*'.

HVR_EVENT

Since v5.5.0/3

Column	Data type	Optional?	Description

ev_id_tstamp	datetime with microsecond precision	No	Unique ID of this event. This is the time when the event was created. This timestamp is generated using **HVR_COUNTER**.
ev_type	string 64 characters	Yes	Name of this event. Some events are just audit records of system changes (e.g. **Catalog Change**) while other events (e.g. **Refresh** or **Compare**) are activities which could run for some time.
user_name	string 128 characters	Yes	Name of the user that created this event.
ev_descrip	string 1024 characters	Yes	Description of this event.
chn_name	string 12 characters	Yes	Name of the channel affected by this event.
job_name	string 40 characters	Yes	Name of the job associated to this event.
ev_state	string 10 characters	Yes	State of this event, either **PENDING**, **DONE** or **FAILED**.
ev_num_retries	int	Yes	Number of times event has been restarted.
ev_response	string 128 characters	Yes	Summary of the activity in this event; either written when the event finishes successfully or containing the error that caused it to fail or be cancelled.
ev_start_tstamp	datetime with microsecond precision	Yes	Time when event was last started (updated on each retry).
ev_finish_tstamp	datetime with microsecond precision	Yes	Time when event finished.
ev_body	clob	Yes	Event body string in JSON. Contains arguments for this event.
last_updated	datetime with microsecond precision	Yes	Time when event was last updated.

HVR_EVENT_RESULT

Since v5.5.0/3

Column	Data type	Optional?	Description
ev_id_tstamp	datetime with microsecond precision	No	Event ID of parent event (from **HVR_EVENT**).
tbl_name	string 128 characters	No	Name of table associated to this result.
res_name	string 64 characters	No	Name of this result.

res_value	clob	Yes	Value of this result.
loc_name	string 5 characters	Yes	Name of location associated to this result.
loc_name_2	string 5 characters	Yes	Name of second location associated to this result.
last_updated	datetime with microsecond precision	Yes	Time when event result was last updated.

HVR_EVENT_ARCHIVED

Since v5.6.0/0

> This table is generated only if HVR is upgraded to 5.6.0/0 from any of the HVR releases between 5.5.0/3 and 5.5.5/8.

Column	Data type	Optional?	Description
ev_id_tstamp	datetime with microsecond precision	No	Unique ID of this event. This is the time when the event was created. This timestamp is generated using **HVR_COUNTER**.
ev_type	string 64 characters	Yes	Name of this event. Some events are just audit records of system changes (e.g. **Catalog Change**) while other events (e.g. **Refresh** or **Compare**) are activities which could run for some time.
ev_body	clob	Yes	Event body string in JSON. Contains arguments for this event.
ev_descrip	string 1024 characters	Yes	Description of this event.
chn_name	string 12 characters	Yes	Name of the channel affected by this event.
ev_status	string 10 characters	Yes	State of this event, either **PENDING**, **DONE** or **FAILED**.
ev_response	string 128 characters	Yes	Summary of the activity in this event; either written when the event finishes successfully or containing the error that caused it to fail or be cancelled.
ev_finish_tstamp	datetime with microsecond precision	Yes	Time when event finished.

Extra Columns for Capture, Fail and History Tables

Each capture, fail or history table created by HVR contains columns from the replicated table it serves, plus extra columns. These columns contain information about what the captured operation was and where it must be sent.

The following extra columns can appear in capture, fail or history tables:

Column	Datatype	Description
hvr_seq	**float** or **byte10** on Ingres, **numeric** on Oracle and **timestamp** on SQL Server	Sequence in which capture triggers were fired. Operations are replayed on integrate databases in the same order, which is important for consistency.
hvr_tx_id	string	Transaction ID of captured change. This number is unique at any moment (each open transaction has a unique number) but may not be increasing.
hvr_tx_seq	string	Sequence number of transaction. For log–based capture this value can be mapped to the Ingres LSN or the Oracle SCN of the transaction's **commit** state ment.
hvr_tx_countdown	number	Countdown of change within transaction, for example if a transaction contains three changes the first change would have countdown value 3, then 2, then 1. A value of zero indicates that commit information is missing for that change.

hvr_op	number	Operation type. Values are • **0** - Delete • **1** - Insert • **2** - After update • **3** - Before key update • **4** - Before non–key update • **5** - Truncate table • **8** - Delete affecting multiple rows • **22** - Key update with missing values • **32** - Resilient variant of **22** • **42** - Key update whose missing values have been augmented • **52** - Resilient variant of **42** A key–update sometimes appears as a before update followed by an after update, but is sometimes converted into a delete followed by an insert. A before non–key update row (**hvr_op=4**) can be removed by adding **Capture /NoBeforeUpdate**. During an on-line refresh, a delete, insert and delete can be 'in-doubt'; these are shown as **10**, **11** and **12** respectively. To ignore this 'in-doubt' information, use **mod(10)** to convert these back to **0**, **1** or **2**. Note that **Integrate /Burst** will convert a key update (**hvr_op 3** then **2**) into a delete+insert pair (**hvr_op 0** t hen **1**).
hvr_cap_loc	string	Name of location on which the change was captured.
hvr_cap_tstamp	normally date, sometimes integer (number of seconds since 1st Jan 1970 GMT)	Timestamp of captured change. For log-based capture this is the time that the change was committed (it did not logically "exist" before that moment). For trigger-based capture it when the DML statement itself was triggered. If HVR refresh fills this value, it uses the time that the refresh job started.
hvr_cap_user	string	Name of user who performed captured change. Supported only on few DBMSs, see capabilities Populates column **hvr_cap_user** for use in **ColumnProperties {hvr_cap_user}** substitutions.
hvr_address	string	Address of target location for change. Only set if action **Restrict /HorizColumn** is defined.

hvr_err_tstamp	date	Time at which integration error occurred. Written into fail table.
hvr_err_msg	long string	Integration error message written into fail table.
hvr_colval_mask	string	Mask showing which column values were missing or not updated. For example value **––––** **m–** could mean that log–based capture of an update is missing a value for the second last column of a table.

Integrate Receive Timestamp Table

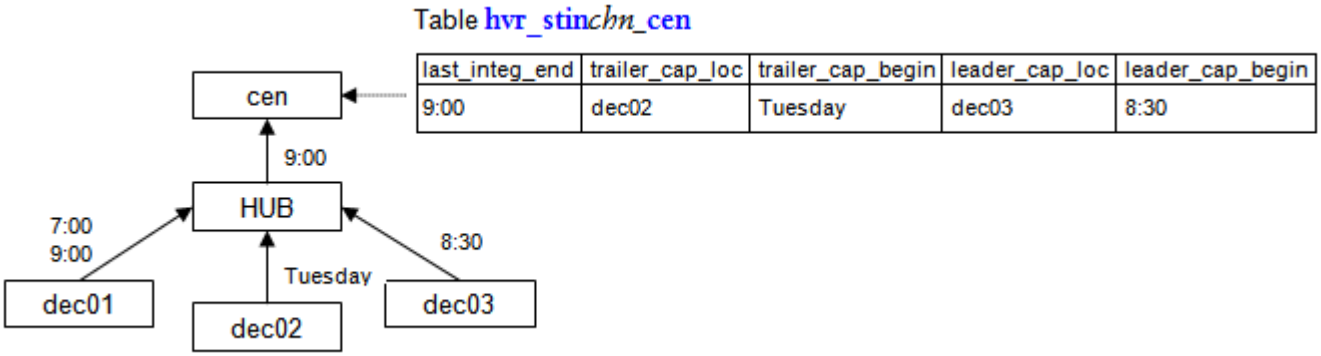
The receive stamp table in an integration database contains information about which captured changes have already been integrated. Because changes can be captured from more than one location, the table contains information about the 'leader' and 'trailer' location. If there is only one capture location then that location is both the leader and the trailer.

The leader location is the capture location whose changes have arrived most recently. Column **leader_cap_loc** contains the leaders's location name and column **leader_cap_begin** contains a time before which all changes captured on the leader are guaranteed to be already integrated. The trailer location is the location whose changes are oldest. Column **trailer_cap_loc** contains the trailer's location name and column **trailer_cap_begin** contains a timestamp. All changes captured on the trailer location before this time are guaranteed to be already integrated on the target machine. Receive timestamps are updated by HVR when the integrate jobs finishes running.

HVR accounts for the fact that changes have to be queued first in the capture database and then inside routing, before they are integrated. The receive stamp table is only updated if an arrival is guaranteed, so if a capture job was running at exactly the same time as an integrate job and the processes cannot detect whether a change 'caught its bus' then receive stamps are not reset. The receive stamp table is named **hvr_stin**_*chn_loc*. It is created the first time the integrate jobs run. The table also contains columns containing the date timestamps as the number of seconds since 1970 1st January GMT.

Example

Data was last moved to the hub from location **dec01** at 7:00 and 9:00, from **dec02** on Tuesday, from **dec03** at 8:30 and from the hub to central at 9:00. For the **cen** location, the leader location is **dec03** and the trailer location is **dec02**. The contents of the integrate receive timestamp table is shown in the diagram below. Note that location **dec01** is not the leader because its job ran at the same time as the central job, so there is no guarantee that all data available at **dec01** has arrived.



Metrics for Statistics

Contents

List of HVR Stats Metrics

Latency Stats Metrics

Metric Name	5 7 0 1 6 47	Remarks
Capture Latency Min	secs	Latency value reported by capture job ('Scanned X .... from 1 mins ago'). In older HVR versions this metric was called **Minimum Capture Latency** .
Capture Latency Max	secs	In older HVR versions this metric was called **Maximum Capture Latency** .
Integrate Latency Min	secs	In older HVR versions this metric was called **Minimum Integrate Latency** .
Integrate Latency Max	secs	In older HVR versions this metric was called **Maximum Integrate Latency** .
Capture Rewind Interval	secs	
Router Capture Latency	secs	

Router Integrate Latency	secs	

Captured Row Counts Stats Metrics

Metric Name	5 7 0 1 6 47	Remarks
Captured Inserts	rows	
Captured Updates	rows	Updates are captured as 2 rows not 1, unless Capture/NoBeforeUpdate defined. So captured inserts+updates+deletes can be less than captured rows
Captured Deletes	rows	
Captured Rows	rows	Sometimes changes (e.g. updates) are captured as 2 rows not 1. So captured inserts+updates+deletes can be less than captured rows
Captured Rows Backdated	rows	A 'backdated' version of 'Captured Rows'. Backdated means latency in the message is used to assign the value to an time earlier than the message's timestamp. For example if a message 'Scanned 100 changes from 30 mins ago' has timestamp '16:55:00' then 100 is both added to 'Captured Rows' for 16:55 and also to 'Captured Rows Backdated' for 16:25. Visual comparison of a back dated metric to its regularly dated one can display bottleneck pattern; the area under both graph lines should be identical (the total amount of work done) but if the backdated graph shows a peak which quickly subsides and the regularly dated line shows a smaller rise which subsided more slowly, then a bottleneck is visible. In older HVR versions this metric was called **DBMS Log Rows** .
Captured Skipped Rows	rows	Changes are skipped by Hvr 'controls', e.g. after an on-line refresh.
Augmented Rows	rows	This counts situations where capture job performs db query to fetch extra column value(s) to 'augment' other column values read from dbms logging, These operations are relatively slow (db query needed).
SAP Augment Selects	rows	Only occurs when SapXForm engine is used. Counts situations where capture job performs db query to fetch extra rows to 'augment' SAP cluster rows read from dbms logging, These operations are relatively slow (db query needed).
Captured DDL Statements	rows	Measures number of DDL statements processed by action AdaptDDL. These can be followed by on-line refreshes which can cause row skipping. In older HVR versions this metric was called **Captured DDL** .

Integrated Change Counts Stats Metrics

Metric Name	5 7 0 1 6 47	Remarks
Integrated Inserts	changes	

Integrated Updates	changes	Although updates can be captured as 2 rows not 1, this metric measures the updates changes, not underlying rows.
Integrated Deletes	changes	
Integrated Changes	changes	Sometimes changes (e.g. updates) are captured as 2 rows not 1. Such an update would count as 1 here (not 2). Note that inserts+updates+deletes equals number of changes can be less than number of rows. This count does NOT include DDL changes (or rows refreshed due to DDL changes).
Integrated Skipped Changes	changes	Changes are skipped by Hvr 'controls', e.g. during an on-line refresh or if a job is restarted after some interrupt.
Changes Coalesced Away	changes	Integrate /Burst and /Coalesce will both squeeze consecutive operations on same row (e.g. insert + 3 updates) into a single change.
Failed Inserts Saved	changes	Failed inserts written to <tbl>__f table after errors when Integrate/OnErrorSave is defined. possibly because row already existed.
Failed Updates Saved	changes	Failed updates written to <tbl>__f table after errors when Integrate/OnErrorSave is defined, possibly because row did not exist.
Failed Deletes Saved	changes	Failed delete written to <tbl>__f table after errors when Integrate/OnErrorSave is defined, possibly because row did not exist.
Failed Changes Saved	changes	Total number of changes (insert+update+delete) which failed and were written to <tbl>__f table after errors when Integrate/OnErrorSave is defined, possibly because row did not exist.
Collision Changes Discarded	changes	Change which was discarded due to collision detection (action CollisionDetect) which decided change was older then current row in target db. In older HVR versions this metric was called **Discarded Changes** .

Transactions Stats Metrics

Metric Name	5 7 0 1 6 47	Remarks
Captured Transactions	transaction	This metric counts number of COMMITs of transactions which contain changes to replicated tables. Note that its interesting to see if a system is dominated by large (e.g. 1000 row /commit) transactions, but comparing this filed with 'Captured Rows' only shows an average.
Captured Transactions Backdated	transaction	A 'backdated' version of 'Captured Transactions'. Backdated means latency in the message is used to assign the value to an time earlier than the message's timestamp. For example if a message 'Scanned 100 changes from 30 mins ago' has timestamp '16:55:00' then 100 is both added to 'Captured Transactions' for 16:55 and also to 'Captured Transactions Backdated' for 16:25. Visual comparison of a back dated metric to its regularly dated one can display bottleneck pattern; the area under both graph lines should be identical (the total amount of work done) but if the backdated graph shows a peak which quickly subsides and the regularly dated line shows a smaller rise which subsided more slowly, then a bottleneck is visible. In older HVR versions this metric was called **Transactions Logged in DBMS** .

Integrated Transactions	transaction	Hvr integrate will often bundle lots of smaller 'Captured Transaction' values into fewer 'Integrated Transactions' for speed. This metric measure these bundled commits, not the original ones.

Durations Stats Metrics

Metric Name	5 7 0 1 6 47	Remarks
Capture Duration Total	secs	In older HVR versions this metric was called **Total Capture Duration** .
Capture Duration Max	secs	In older HVR versions this metric was called **Maximum Capture Duration** .
Capture Duration Average	secs	In older HVR versions this metric was called **Average Capture Duration** .
Integrate Duration Total	secs	In older HVR versions this metric was called **Total Integrate Duration** .
Integrate Duration Max	secs	In older HVR versions this metric was called **Maximum Integrate Duration** .
Integrate Duration Average	secs	In older HVR versions this metric was called **Average Integrate Duration** .
Integrate Burst Load Duration Total	secs	In older HVR versions this metric was called **Tot Integ Burst Load Dur** .
Integrate Burst Load Duration Max	secs	In older HVR versions this metric was called **Max Integ Burst Load Dur** .
Integrate Burst Load Duration Average	secs	In older HVR versions this metric was called **Avg Integ Burst Load Dur** .
Integrate Burst Setwise Duration Total	secs	In older HVR versions this metric was called **Tot Integ Burst Setwise Dur** .
Integrate Burst Setwise Duration Max	secs	In older HVR versions this metric was called **Max Integ Burst Setwise Dur** .
Integrate Burst Setwise Duration Average	secs	In older HVR versions this metric was called **Avg Integ Burst Setwise Dur** .

Speed Stats Metrics

Metric Name	5 7 0 1 6 47	Remarks
Capture Speed Max	rows/sec	This is maximum speed reached by during this time period. For a 'cl*' value its the max speed for any job; if this is aggregated In older HVR versions this metric was called **Maximum Capture Change Speed** .
Capture Speed Average	rows/sec	In older HVR versions this metric was called **Average Capture Change Speed** .
Integrate Speed Max	rows/sec	In older HVR versions this metric was called **Maximum Integrate Change Speed** .
Integrate Speed Average	rows/sec	In older HVR versions this metric was called **Average Integrate Change Speed** .
Integrate Burst Load Speed Max	rows/sec	In older HVR versions this metric was called **Max Integ Burst Load Speed** .
Integrate Burst Load Speed Average	rows/sec	In older HVR versions this metric was called **Avg Integ Burst Load Speed** .
Integrate Burst Setwise Speed Max	rows/sec	In older HVR versions this metric was called **Max Integ Burst Setwise Speed** .
Integrate Burst Setwise Speed Average	rows/sec	In older HVR versions this metric was called **Avg Integ Burst Setwise Speed** .

Cycles Stats Metrics

Metric Name	5 7 0 1 6 47	Remarks
Capture Cycles	cycles	Count of capture cycles whose start occurred during this time period. This does not include 'sub-cycles' or 'silent cycles', but does include 'empty cycles'. A 'sub-cycle' happens when a busy capture job emits a block of changes but has not yet caught up to the 'top'. It can be recognized as an extra 'Scanning' message which is not preceded by a 'Cycle X' line'. A 'silent-cycle' is when a capture job sees no change activity and decides to progress its 'capture state' files but without writing an line in the log (above every 10 secs). A 'empty-cycle' is when a capture job sees no change activity and does write line in the log (above every 10 mins).

Integrate Cycles	cycles	Count of integrate cycles whose start occurred during this time period. Note that the integrate activity may fall into a subsequent period.

Byte I/O Stats Metrics

Metric Name	5 7 0 1 6 47	Remarks
Routed Bytes Written	bytes	In older HVR versions this metric was called **Routed Bytes** .
Routed Bytes Written Uncompressed	bytes	Refers to Hvr's representation of routed rows in memory. This is different from the DBMS's 'storage size' (DBMS's storage of that row on disk) Example: Say a table has a varchar(100) column containing 'Hello World' which Hvr manages to compress that down to 3 bytes. Hvr's memory representation of varchar(100) is 103 bytes, wheres the DBMS storage is 13 bytes. In older HVR versions this metric was called **Routed Bytes (uncompressed)** .
Captured File Size	bytes	In older HVR versions this metric was called **Captured File Size** .
Capture DbmsLog Bytes	bytes	In older HVR versions this metric was called **Capture Log Bytes Read** .
Capture DbmsLog Bytes Backdated	bytes	In older HVR versions this metric was called **DBMS Log Bytes Written** .

Compression Stats Metrics

Metric Name	5 7 0 1 6 47	Remarks
Compression Ratio Max	%	When transporting table rows, Hvr reports its 'memory compression' ratio; the number of compressed bytes transmitted over network compared with Hvr's representation of that row in memory. This different from the DBMS's 'storage compression ratio' (number of compressed bytes transmitted compared with DBMS's storage of that row on disk. Example: Say a table has a varchar(100) column containing 'Hello World' which Hvr manages to compress that down to 3 bytes. Hvr's memory representation of varchar(100) is 103 bytes, wheres the DBMS storage is 13 bytes. In this case Hvr's 'memory compression' ration is 97% (1-(3/103)) whereas the storage compression ratio would be 77% (1-(13/103)). In older HVR versions this metric was called **Maximum Compression Ratio** .
Compression Ratio Average	%	In older HVR versions this metric was called **Average Compression Ratio** .

Replicated Files Stats Metrics

Metric Name	5 7 0 1 6 47	Remarks
Captured Files	files	Counts rows captured from 'file' locations.
Integrated Files	files	
Failed Files Saved	files	

Errors/Warnings Stats Metrics

Metric Name	5 7 0 1 6 47	Remarks
Errors	lines	Counts number of errors (lines matching F_J*) in the job logfile. Such an error line could affect multiple rows, or could affect one row and be repeated lots of times (so it counts as multiple 'errors'). In older HVR versions this metric was called **Number of Errors** .
^Errors	string	Annotation: Text of the most recent error line as additional information for metric Errors. In older HVR versions this metric was called **Last Error** .
Errors F_J*	lines	Groups the appearing errors by up to the 2 most significant error numbers. Example: Say error F_JA1234 happened. Then the metric 'Errors F_JA1234' will increase by 1. If the errors F_JA1234; F_JB1234: The previous error occured ...; F_JC1234: The previous error occured ...; appear then 'Errors F_JA1234 F_JB1234' will increase by 1.
^Errors F_J*	string	Annotation: Groups the appearing errors messages by up to the 2 most significant error numbers. Example: Say error F_JA1234 happened. Then the metric '^Errors F_JA1234' will hold the message of F_JA1234. If the errors F_JA1234; F_JB1234: The previous error occured ...; F_JC1234: The previous error occured ...; appear then '^Errors F_JA1234 F_JB1234' will hold the messages of F_JA1234 and F_JB1234.
Warnings	lines	Counts number of warnings (lines matching W_J*) in the job logfile. In older HVR versions this metric was called **Number of Warnings** .
^Warnings	string	Annotation: Most recent warning line. In older HVR versions this metric was called **Last Warning** .
Warnings W_J*	lines	Groups the appearing warning by the warning number. Example: Say warning W_JA1234 happened. Then the metric 'Warnings W_JA1234' will increase by 1.
^Warnings W_J*	string	Annotation: Groups the appearing warning messages by the warning number. Example: Say warning W_JA1234 happened. Then the metric '^Warnings W_JA1234' will hold the message of W_JA1234.

Refresh and Compare Runs Stats Metrics

Metric Name	5 7 0 1 6 47	Remarks
Bulk Compare Job Runs	runs	
Bulk Compare Table Runs	runs	
Bulk Refresh Job Runs	runs	
Bulk Refresh Table Runs	runs	
Row-wise Compare Job Runs	runs	
Row-wise Compare Table Runs	runs	
Row-wise Refresh Job Runs	runs	
Row-wise Refresh Table Runs	runs	

Refresh and Compare Job Duration Stats Metrics

Metric Name	5 7 0 1 6 47	Remarks
Bulk Compare Job Duration Min	secs	In older HVR versions this metric was called **Bulk Compare Job Dur Min** .
Bulk Compare Job Duration Max	secs	In older HVR versions this metric was called **Bulk Compare Job Dur Max** .
Bulk Compare Job Duration Average	secs	In older HVR versions this metric was called **Bulk Compare Job Dur Avg** .

Row-wise Compare Job Duration Min	secs	In older HVR versions this metric was called **Row-wise Compare Job Dur Min** .
Row-wise Compare Job Duration Max	secs	In older HVR versions this metric was called **Row-wise Compare Job Dur Max** .
Row-wise Compare Job Duration Average	secs	In older HVR versions this metric was called **Row-wise Compare Job Dur Avg** .
Bulk Refresh Job Duration Min	secs	In older HVR versions this metric was called **Bulk Refresh Job Dur Min** .
Bulk Refresh Job Duration Max	secs	In older HVR versions this metric was called **Bulk Refresh Job Dur Max** .
Bulk Refresh Job Duration Average	secs	In older HVR versions this metric was called **Bulk Refresh Job Dur Avg** .
Row-wise Refresh Job Duration Min	secs	In older HVR versions this metric was called **Row-wise Refresh Job Dur Min** .
Row-wise Refresh Job Duration Max	secs	In older HVR versions this metric was called **Row-wise Refresh Job Dur Max** .
Row-wise Refresh Job Duration Average	secs	In older HVR versions this metric was called **Row-wise Refresh Job Dur Avg** .

Refresh and Compare Table Duration Stats Metrics

Metric Name	5 7 0 1 6 47	Remarks
Bulk Compare Table Duration Min	secs	In older HVR versions this metric was called **Bulk Compare Table Dur Min** .
Bulk Compare Table Duration Max	secs	In older HVR versions this metric was called **Bulk Compare Table Dur Max** .
Bulk Compare Table Duration Average	secs	In older HVR versions this metric was called **Bulk Compare Table Dur Avg** .

Row-wise Compare Table Duration Min	secs	In older HVR versions this metric was called **Row-wise Compare Table Dur Min** .
Row-wise Compare Table Duration Max	secs	In older HVR versions this metric was called **Row-wise Compare Table Dur Max** .
Row-wise Compare Table Duration Average	secs	In older HVR versions this metric was called **Row-wise Compare Table Dur Avg** .
Bulk Refresh Table Duration Min	secs	In older HVR versions this metric was called **Bulk Refresh Table Dur Min** .
Bulk Refresh Table Duration Max	secs	In older HVR versions this metric was called **Bulk Refresh Table Dur Max** .
Bulk Refresh Table Duration Average	secs	In older HVR versions this metric was called **Bulk Refresh Table Dur Avg** .
Row-wise Refresh Table Duration Min	secs	In older HVR versions this metric was called **Row-wise Refresh Table Dur Min** .
Row-wise Refresh Table Duration Max	secs	In older HVR versions this metric was called **Row-wise Refresh Table Dur Max** .
Row-wise Refresh Table Duration Average	secs	In older HVR versions this metric was called **Row-wise Refresh Table Dur Avg** .

Refresh and Compare Rows Stats Metrics

Metric Name	5 7 0 1 6 47	Remarks
Bulk Compare Rows	rows	
Bulk Compare Packed Rows	rows	
Row-wise Compare Rows	rows	

Row-wise Compare Packed Rows	rows	
Bulk Refresh Rows	rows	
Bulk Refresh Packed Rows	rows	
Row-wise Refresh Rows	rows	
Row-wise Refresh Packed Rows	rows	
Compare Differences Inserts	rows	
Compare Differences Updates	rows	
Compare Differences Deletes	rows	
Refresh Differences Inserts	rows	
Refresh Differences Updates	rows	
Refresh Differences Deletes	rows	

Router Rows Stats Metrics

Metric Name	5 7 0 1 6 47	Remarks
Capture Router Rows	rows	
Integrate Router Rows	rows	

Router Bytes Stats Metrics

Metric Name	5 7 0 1 6 47	Remarks
Capture Router Bytes	bytes	
Integrate Router Bytes	bytes	

Router Files Stats Metrics

Metric Name	5 7 0 1 6 47	Remarks
Capture Router Files	files	
Integrate Router Files	files	

Router Timestamps Stats Metrics

Metric Name	5 7 0 1 6 47	Remarks
Capture Rewind Time	time	
Capture Emit Time	time	
Capture Last Cycle Time	time	
Capture Router Timestamp	time	
Integrate Router Timestamp	time	

Description of Units

Unit	Description

%	Percent. When used for compression this means ratio of bytes left after compression. So if 100 bytes are compressed by 70% then 30 bytes remain.
bytes	
changes	Changes of tables affected by insert, update or delete statements. Updates are sometimes moved as 2 rows (before-update and after-update).
cycles	A capture or integrate cycle.
files	
int	Integer number
lines	Number of message lines written to log file. A single line could be an error message which mentions multiple failed changes
rows	Rows of tables affected by insert, update or delete statements.
rows/sec	Measures the average of 'rows' or 'changes' replicated per second.
runs	Number of times a job has been run.
secs	Seconds
string	
time	Timestamp
transaction	Unit 'transaction' means a group of changes terminated by a commit (not just a changed row).

Naming of HVR Objects Inside Database Locations

The following table shows the database objects which HVR can create to support replication. The name of each database object either begins with an **hvr_** prefix or consists of a replicated table name followed by two underscores and a suffix.

Name	Description
*tbl*_ _**c0** *tbl*_ _**c1**	Capture tables (trigger–based capture only).
*tbl*_ _**ci** *tbl*_ _**cd** *tbl*_ _**cu**	Capture database rules or triggers.
*tbl*_ _**c** *tbl*_ _**c0** *tbl*_ _**c1**	Capture database procedures (trigger–based capture only).
*tbl*_ _**l** *tbl*_ _**li** *tbl*_ _**ld** *tbl*_ _**lu**	Database procedures, rules and triggers for capture of dynamic lookup table. Created for action **Restrict /DynamicHorizLookup**.
hvr_sys_table	Temp table used for faster set-wise queries of DBMS catalogs (Oracle only)
hvr_tog*chn*	Capture toggle state table (trigger–based capture only).
hvr_qtog*chn*	Capture quick toggle state table (trigger–based capture only).
hvr_lktog*chn*	Capture toggle lock table (Ingres trigger–based capture only).
hvr_seq*chn*	Capture sequence number (Oracle trigger–based capture only).
*tbl**_ _ii * *tbl*_ _**id** *tbl*_ _**iu**	Integrate database procedures. Created for action **Integrate /DbProc**.
*tbl*_ _**b**	Burst tables. Used for staging if **Integrate /Burst** is defined.
*tbl** _ _f*	Integrate fail table. Created when needed, i.e. when an integrate error occurs.
*tbl*_ _**h**	Collision history table. Created if action **CollisionDetect** is defined.
*tbl*_ _**x**	External table. Used for external loading into Greenplum.
hvr_stbu*chn_loc*	Integrate burst state table. Created if action **Integrate /Burst** is defined.
hvr_stin*chn_loc*	Integrate receive timestamp table.
hvr_stis*chn_loc*	Integrate commit frequency table.
hvr_strr*chn_loc*	Bulk refresh recovery state table.
hvr_strs*chn_loc*	State table created by **Hvrrefresh** so that HVR capture can detect the session name.
hvr_integrate	Integrate role (Ingres only).
hvr_refresh	Refresh role (Ingres only).
hvr_scheduler	Scheduler role (Ingres only).

⚠
- If a table has no non–key columns (i.e. the replication key consists of all columns) then some update objects (e.g. *tbl*__**iu**) may not exist.
- Capture objects are only created for trigger–based capture; log–based capture does not use any database objects.
- Action **DbObjectGeneration** can be used to inhibit or modify generation of these database objects.