HVR SOFTWARE

REAL-TIME DATA REPLICATION

HVR User Manual

Version 4.7

23 September 2015

HVR User Manual

Version 4.7

23 September 2015

Contents

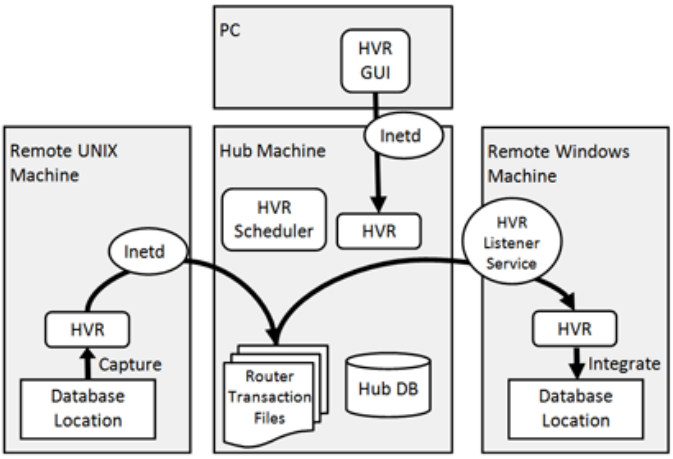
1 Introduction

HVR replicates transactions between databases that HVR calls 'locations'. Each change it captures is applied to the target locations. It can also replicate between directories (file locations) or replicate between databases and directories.

Terminology

The HUB DATABASE is a small database from which HVR controls replication of the other databases. It can be either an Oracle schema (a username within an instance), Ingres database, SQL Server database or a DB2 database. It is created especially for the HVR installation and can have any name (in most examples in this manual it is called *hubdb*). It contains HVR catalog tables that hold all specifications of replication such as the names of the replicated databases, the replication direction and the list of tables to be replicated. These catalog tables are created in the hub database during installation (see also sections New Installation on Unix or Linux and New Installation on Windows).
A LOCATION is a database that HVR will replicate to or from. A location can also be a directory (a 'file location') from which HVR replicates files or a Salesforce endpoint.
A CHANNEL is an object in HVR that groups together the locations that should be connected together and the tables that should be replicated. It also contains actions that control how replication should be done. For example, to capture changes a **DbCapture** action must be defined on a database location. Channels are defined in the hub database. They can be configured for replication between various databases, or between files, or between databases and files. As well as replicating changes, channels can also be used to refresh data. Refresh means all data is read from a source and loaded into another database, without replication.

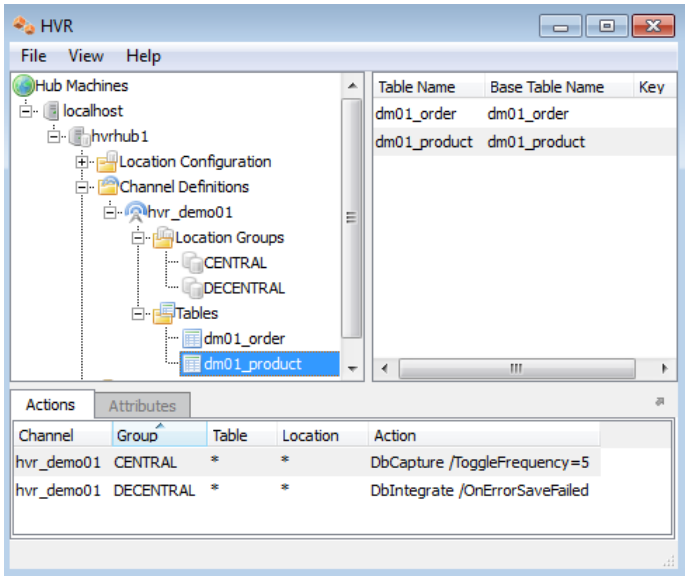
Process Architecture and Network Connections



The hub machine contains the HVR hub database and an HVR scheduler (which controls the replication jobs) and all the log files. Locations can be either local (i.e. on the hub machine) or remote.
To access a remote location HVR normally connects to an HVR installation on that remote machine using a special TCP/IP port number. If the remote machine is Unix then the INETD daemon is configured to listen

to this TCP/IP port. If it is a Windows machine then HVR listens with its own HVR Remote Listener (a Windows Service). Alternatively, HVR can connect to a remote database location using the DBMS protocol such as Oracle TNS.
The HVR Scheduler on the hub machine starts capture and integrate jobs that connect out to the remote location and either capture or apply ('integrate') changes to the remote location. HVR on a remote machine is quite passive; the executables are acting as slaves for the hub machine. Replication is entirely controlled from the hub machine.

Overview of Steps to Setup a Channel

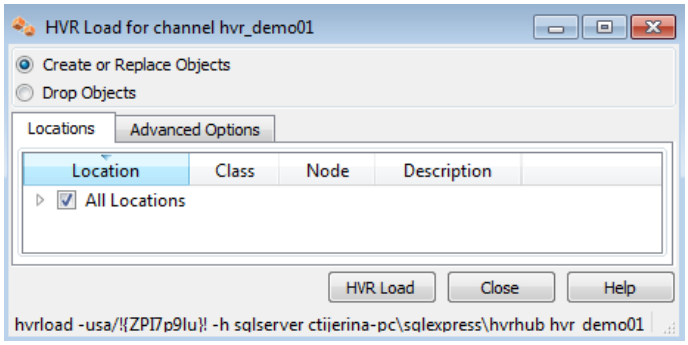


HVR must first be installed on the various machines involved. For the installation steps, see sections New Installation on Unix or Linux or New Installation on Windows. These installation steps also create a hub database containing empty catalog tables.
Once HVR is installed, it can be managed using a Graphical User Interface (GUI). The GUI can just 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.
To start the GUI double click on its shortcut or command **hvrgui** on Linux. See section hvrgui for more information. The HVR GUI allows a channel to be defined in the hub database. The channel must contain at least two locations (e.g. an Oracle schema, or an Ingres or SQL Server database). It must further contain location groups. This is a collection of locations belonging to a channel. HVR actions are typically defined on the channel's location group. HVR channels can be for database replication or for replicating files.
To make a channel for database replication, choose the **Table Select** option in the GUI to import a list of tables from a database location. Action **DbCapture** is defined on the source database to capture database changes and **DbIntegrate** is defined on the target database to apply ('integrate') changes. See appendix Quick Start for HVR on Oracle, Quick Start for HVR on Ingres, or Quick Start for HVR on SQL Server for quick start steps for setting up database replication.
HVR behavior can be reconfigured by specifying other parameters on the actions or by adding other actions.

Managed File Transfer

A file replication channel is built with file locations. A file location can be a directory or a tree of directories on a machine where HVR is installed. It can also be a location that HVR can access using FTP, SFTP or WebDAV protocols.
HVR can either replicate new files from one file location to a different file location or it can replicate between file locations and database locations. If HVR replicates between file locations it treats these files simply as a stream of bytes. But if a channel has database and file locations then each file is interpreted as containing database changes, by default in HVR's XML format.
When HVR is capturing changes from a file location's directory it can either move each file (delete it after it is captured) or make a copy to the other location.
Actions **FileCapture** and **FileIntegrate** must be applied for file replication. See appendix Quick Start for File Replication.

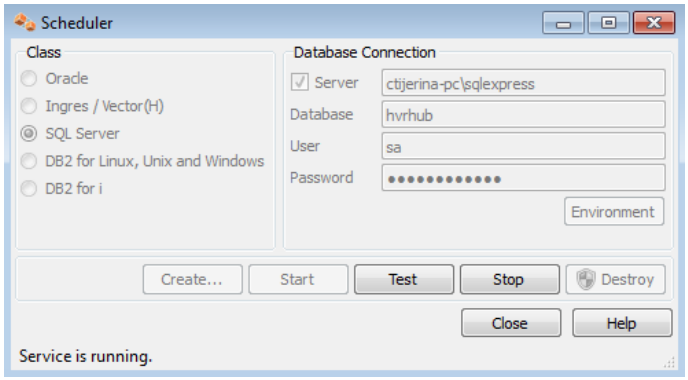
Starting Replication



The runtime replication system is generated by command **HVR Load** in the GUI or **hvrload** from the command line on the hub machine. **HVR Load** 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 Load** 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.

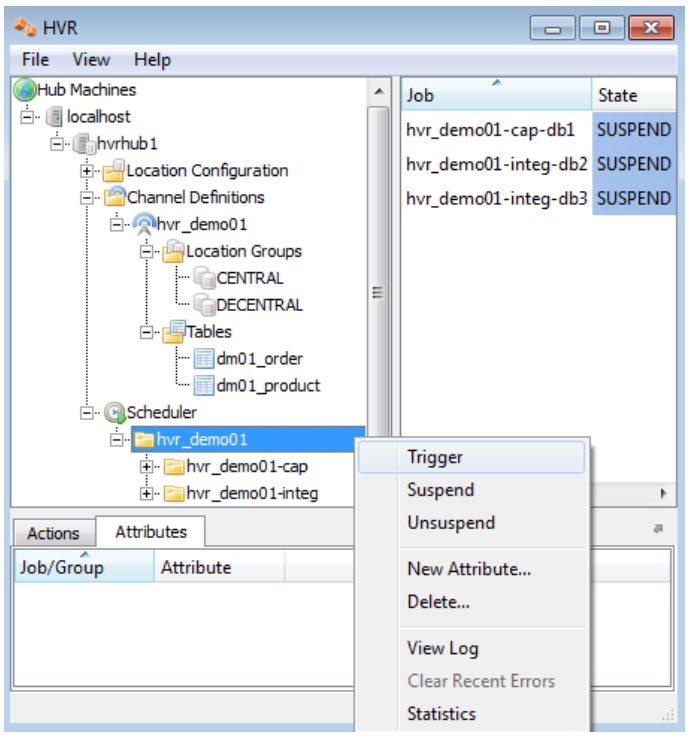


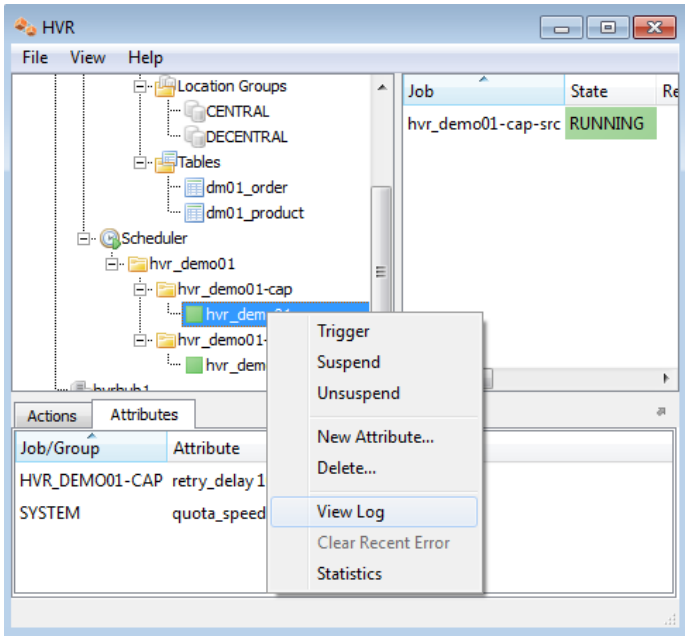
HVR Load creates jobs in suspended state. These can be activated using the GUI by right clicking on a channel and select **Trigger**.
Like other operations in the HVR GUI, starting jobs can also be done from the command line. See section hvrtrigger.
The HVR Scheduler collects output and errors from all 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.

A job's errors can be viewed by clicking on the job underneath the scheduler in the treeview and click **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*.

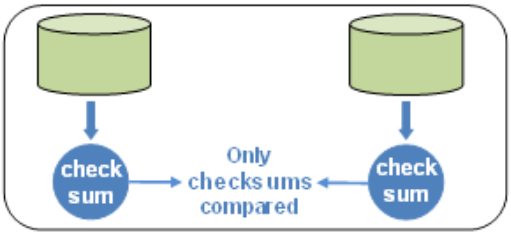
Refresh and Compare

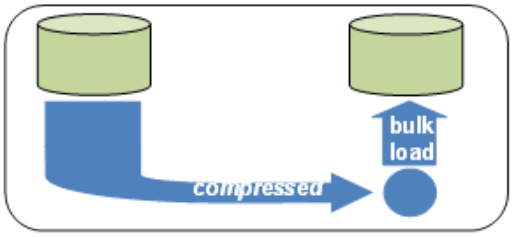
As well as actual replication (capturing each change and applying it to another database), HVR supports refresh and compare. HVR supplies two flavors of refresh and compare.





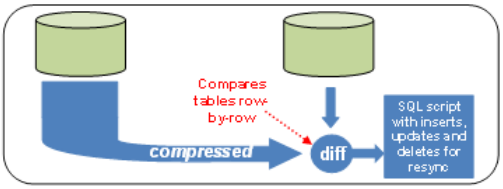
Bulk compare means that HVR performs checksums on each of the tables in the replication channel. The actual data does not travel across the network so this is very efficient for large databases over a WAN.

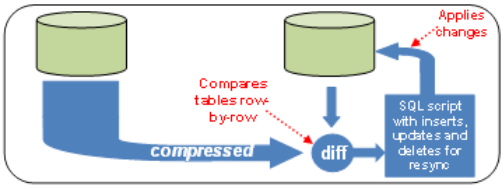




Bulk Refresh means that HVR extracts the data from all the tables in one database, compresses it, brings it across the network, uncompresses it and loads that data into the target database. After the data has been bulk loaded into the target database, indexes are reinitialized.

Row wise compare means that HVR extracts the data from one database, compresses it and on the target database it compares those changes row by row with the changes in the target database. For each difference detected an SQL statement is written; an insert, update or delete.





Row wise refresh is the same as row wise compare. But the resulting SQL statements are applied directly to the target database in order to resynchronize the tables.

1.1 Using HVR for Replication Transformations

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 datatypes 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. See also section Restrict.
  - 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.
- In the HVR pipeline using action **Transform**. These transformations can be written as a command or in the XSLT language. See section Transform.
- 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 Agent.

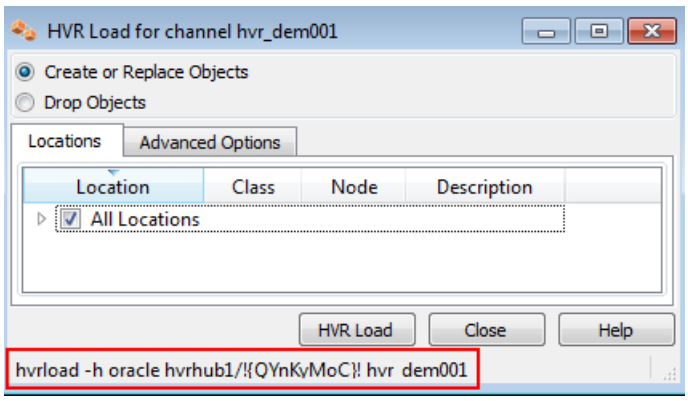
Note that HVR does not only apply these transformations during replication (capture and integration). It also applies these transformations when doing a compare or refresh between the source and target databases.

1.2 Calling HVR on the Command Line

Many HVR commands take a hub database name (*hubdb*) as their first argument. This is either an Oracle schema (username), Ingres or Vectorwise database, SQL Server database, or DB2 database, depending on the form:

Example	Description
hvrload hubdb/pwd hvr_demo01 **hvrload hubdb/pwd@tnsname hvr_demo01** **hvrload -horacle hubdb/pwd hvr_demo01**	Oracle schema **hubdb** with password **pwd**. Note that in the first example HVR recognizes this as Oracle, because of the forward slash.
hvrload hubdb hvr_demo01	Ingres or Vectorwise hub database **hubdb**, because the database name contains neither a forward nor backslash.
hvrload \hubdb hvr_demo01 **hvrload inst\hubdb hvr_demo01** **hvrload node\inst\hubdb hvr_demo01** **hvrload -uuser/pwd \hubdb hvr_demo01** **hvrload hsqlserver hubdb hvr_demo01**	SQL Server hub database **hubdb**. Note that HVR recognizes this as SQL Server due to the back slash. A SQL Server node and SQL Server instance can be added with extra back slashes. A username and password can be supplied with option **–u**.
hvrload -hdb2 hubdb hvr_demo01 **hvrload –hdb2 –uuser/pwd hubdb hvr_demo01**	DB2 hub database as **hubdb**.

The examples above are for command **hvrload**, but other commands like **hvrrefresh** and **hvrcompare** also behave the same.
The hub database class can be defined explicitly either using option **-h***class* or with environment variable **$HVR_HUB_CLASS**. Valid values are **oracle**,**ingres, sqlserver** or **db2**.
Sometimes a DBMS password may be required as a command line argument. These passwords can be provided in an encrypted form using command **hvrcrypt** to prevent them being visible in the process table (for example with Unix command **ps**). See section hvrcrypt.
HVR commands can be performed either inside the GUI or on the command line. Each GUI dialog displays the equivalent command line at the bottom of the dialog window.



2

Before Installing HVR

Before proceeding, check that the following is present:

- The HVR distribution, downloaded from the website. Read the COMPATIBILITY section of the accompanying RELEASE NOTES to ensure this version is compatible with the target Operating System and DBMS. The release notes are in **$HVR_HOME/hvr.rel**.
- An HVR license file (named **hvr.lic**) which is normally delivered separately to each customer by HVR Technical Support.
- A hub database. This database can be Oracle, Ingres, Microsoft SQL Server or DB2. Initially this 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, et cetera. For Oracle, the hub database is normally just an empty schema within the capture database or the target database.

2.1 Where Should HVR be Installed

The HVR distribution 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).
For an HVR remote location, the full HVR distribution should also be installed on the remote machine, although a few of its files are used. If the DBMS's own protocol is used then no HVR files need be installed on the remote 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 own login account (e.g. **hvr**) on the hub machine and each machine 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 replicate data between these locations without difficulty.
The HVR distribution files are installed under environment variable **$HVR_HOME**. HVR also creates files underneath **$HVR_CONFIG.** Both these environment variables must be configured. An extra environment variable **$HVR_TMP** is also recognized, but if it is not configured it will default to **$HVR_CONFIG/tmp**.

2.2 Requirements for Unix or Linux

UNIX and Linux

Installation of HVR's own protocol can require **root** permission. HVR's protocol is needed when connecting form 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** or **xinetd** configuration files. An alternative configuration method for HVR's protocol is to use command **hvrremotelistener** instead of **inetd** or **xinetd**; this alternative does not necessarily need **root** permission. See hvrremotelistener.

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). This is described in section Requirements for Oracle.

Ingres

Log based capture from Ingres requires that HVR has permission to read the internal DBMS logging files. Installing this requires a special step to create a trusted executable, which must be performed while logged in as **ingres**. See step 4 of section New Installation on Unix or Linux. This installation step is not needed if HVR runs under login ingres, and is also not needed if trigger based capture is used or on Microsoft Windows.

2.3 Requirements for Microsoft Windows

Windows

Perl (version 5.6 or higher) must be installed on the hub machine and also on the capture machine if utility **hvrlogrelease** is used (see section hvrlogrelease). It can be downloaded from `http://strawberryperl.com/`.
If Perl is installed after 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 command **secpol.msc**.

2.4 Requirements for Microsoft Azure

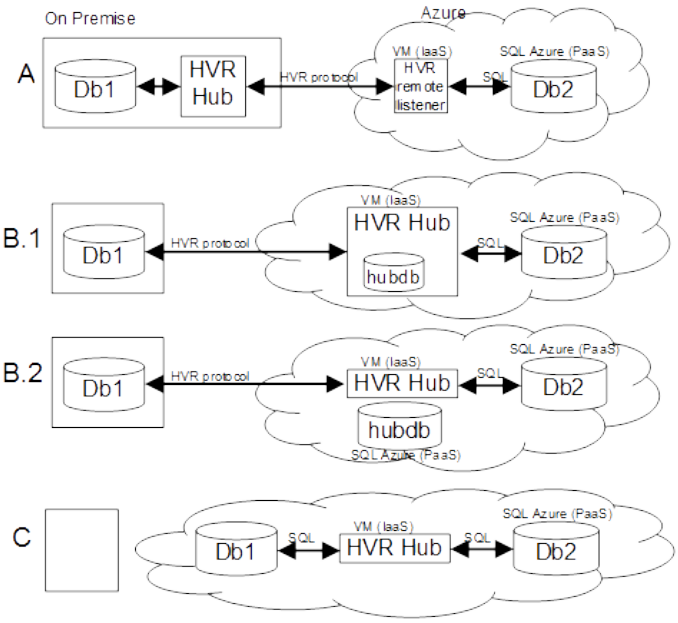
Azure is Microsoft's cloud platform providing the following components relevant for HVR:

- 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.
- Azure data services supported as a source or a target for HVR. HVR supports two Azure data services:

1. SQL Azure database as a subset of a full SQL Server database. This is "Platform as a Service" (PaaS). HVR can connect to Azure SQL as a source, as a target and as hub database. For details, see Requirements for Azure SQL
2. Azure HDInsight, HDFS on Azure. For details on Hadoop, see Requirements for Hadoop 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 Image for Azure" 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 New Installation of HVR Image for Azure for further details. Alternatively, an Agent or Hub can be set up by doing a manual installation as decribed in New Installation 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). 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.
- To use the instance as a hub install perl (Strawberry Perl or ActivePerl).
- 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.
- Consider the use of SSL encryption to securely transport data over public infrastructure to the cloud. See section Configuring_Encrypted_Network_Connections

2.5 Requirements for Oracle

Grants for Hub Schema

HVR needs its own schema (called the 'hub database') on the hub machine, where it keeps its control information. This schema can be inside an Oracle instance on the hub machine, or it can be on another machine using a TNS connection. This user name may need extra permissions to capture changes from a source database or integrate changes into a target database.
The following steps are sufficient to create a hub database schema called **hvrhub** with password **mypass**.

```
$ sqlplus system/manager
SQL> create user hvrhub identified by mypass
  2  default tablespace users
  3  temporary tablespace temp
  4  quota unlimited on users;
SQL> grant create session to hvrhub;
SQL> grant create table to hvrhub;
SQL> grant create trigger to hvrhub;
SQL> grant create procedure to hvrhub;
```

HVR's hub schema must also be allowed to execute package **dbms_alert**. This grant can only be given by a user with **sysdba** privilege (e.g. **oracle**).

```
$ sqlplus / as sysdba
SQL> grant execute on dbms_alert to hvrhub;
```

Grants for Log-Based Capture Database

HVR does log-based capture if action **DbCapture /LogBased** 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. **hvr**).
The database user that HVR uses must be granted **create session** privilege.
HVR needs a temporary table when enrolling tables from the Oracle dictionary. This means the HVR user should be granted create table privilege. An alternative is to create this table manually using SQL statement **create global temporary table hvr_sys_table (table_name varchar(128), table_owner varchar(128))**.
If HVR needs to replicate tables which are owned by other schemas (using action **TableProperties /Schema**) then the user must be granted **select any table** privilege.
If action **DbSequence** is used, then the user must be granted **select any sequence** privilege and given **grant select on sys.seq$**.
HVR needs grants for 'supplemental logging'. This is explained in more detail in the next section. This needs two privileges;

1. The very first time that HVR Load runs it will check if the database allows any supplemental at all. If it is not then **HVR Load** will attempt statement **alter database add supplemental log data**. This will succeed if the HVR user has **sysdba** privilege. Otherwise HVR will write an error message which requests that a different user (who does have this privilege) execute this statement. Note that this statement will hang if other users are changing tables. To see the status of supplemental logging, perform query **select log_group_type from all_log_groups where table_name='mytab'**.
2. If HVR needs to replicate tables which are owned by other schemas, then optionally the HVR user can also be granted **alter any table** privilege, so that **HVR Load** can enable 'supplemental logging' on each of the replicated tables. If this privilege is not granted then **HVR Load** 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.

HVR needs to read the data dictionaries in Oracle's **SYS** schema. This can be done by either giving the HVR user **sysdba** privilege or using statement **grant select any dictionary to** *hvruser*.
For Oracle 11.2, before Oracle 11.2.0.4, HVR will read the redo and archive files directly through its SQL connection, provided those files are on ASM storage or the connection to the source database is over TNS. This can be done by either giving the HVR user **sysdba** privilege or using statement **grant select any transaction to** *hvruser*.
NOTE: The steps to give **sysdba** privilege to an Oracle user (e.g. **hvr**) are as follows;

- On Unix and Linux this can be done by adding the user name used by HVR to the line in **/etc/group** for Oracle sysadmin group. Normally this is **dba** but it can be something else.
- On Windows, right click **My Computer** and select **Manage ▶ Local Users and Groups ▶ Groups ▶ ora_dba ▶ Add to Group ▶ Add**.

Configuring Supplemental Logging for Log-Based Capture

Background: HVR also needs Oracle's "supplemental logging" feature enabled on replicate tables that it must replicate. 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. Oracle supplemental logging can be set at database level and on specific tables.
The very first time that **HVR Load** runs it will check if the database allows any supplemental at all. If it is not then **HVR Load** will attempt statement **alter table database supplemental log data** (see previous section for the privileges that this requires). 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. Note that this statement hangs if other users are changing tables.
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'**].
HVR Load 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 Load 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 /SupplementalLogging**
- On any location:
  - **CollisionDetect**
  - **ColumnProperties** with **/IntegrateExpression** or **/Key**
  - **DbIntegrate** with **/Burst**, **/DbProc**, **/ResilientUpdate**, **/ResilientDeleteCondition** or **/Resilient**
  - **FileIntegrate /RenameExpression**
  - **Restrict** with **/AddressTo** or **/IntegrateCondition**
  - **SalesforceIntegrate**

Problems can occur if multiple HVR channels are capturing changes from the same Oracle base table. This is because both **HVR Load** commands will enable supplemental logging (**alter table...add**) but if one channel is dropped then it will disable supplemental logging (**alter table...drop**) which will accidentally affect the other channel. Such problems can be avoided either by (a) taking care with using **HVR Load** to drop a channel (e.g. when dropping, unselect option **–ol (supplemental logging)** or (b) overriding table level settings by enabling logging at the database level (e.g. **alter database add supplemental log data primary key columns**).
Supplemental logging can be easily disabled (**alter database drop supplemental log data**).

Configuring Redo Files and Archiving for Log-Based Capture

The Oracle instance must have archiving enabled, otherwise (a) HVR will lose changes if it falls behind or it is suspended for a time, and (b) HVR will not capture changes made with Oracle **insert** statements with 'append hints'.
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$database**.
The current archive destination can be checked with query **select destination, status from v$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) reconfiguring **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.

Oracle parameter **DB_BLOCK_CHECKSUM=OFF** is not supported by log-based capture. Values **TYPICAL** (the default) and **FULL** (unnecessarily high) are supported by HVR.
HVR's capture job needs permission to read Oracle's redo and archive files at the Operating System level. There are four different ways that this can be done;

1. For Oracle 11.2, before Oracle 11.2.0.4, HVR will read the redo and archive files directly through its SQL connection, provided those files are on ASM storage or the connection to the source database is over TNS. This requires that its user has been granted the privilege **select any transaction**.
2. Install HVR so it runs as Oracle's user (e.g. **oracle**).
3. Install HVR under a username (e.g. **hvr**) which is a member of Oracle's default Operating System group (typically either **oinstall** or **dba**).
   - On Unix and Linux the default group of user **oracle** can be seen in the 4th field of its line in **/etc/passwd**. The HVR user be made a member of that group by adding its name to file **/etc/group** (e.g. line **oinstall:x:101:oracle,hvr**).
   - On Windows, right click **My Computer** and select **Manage ▶ Local Users and Groups ▶ Groups ▶ ora_dba ▶ Add to Group ▶ Add**.

   Note that adding HVR's user to group **dba** will also give HVR **sysdba** privilege.
4. Create special Access Control Lists (ACLs) on these files so that HVR's user can read them.
   - On Linux the following commands can be run as user **oracle** to allow user **hvr** to see redo files in **$ORACLE_HOME/oradata/SID** and archive files in **$ORACLE_HOME/ora_arch**. Note that an extra "default ACL" is needed for the archive directory, so that future archive files will also inherit the directory's permissions.
   - On HP UX the commands are as follows;

```
$ setacl -m u:hvr:r x $ORACLE_HOME/oradata
$ setacl -m u:hvr:r x $ORACLE_HOME/oradata/SID
$ setacl -m u:hvr:r x $ORACLE_HOME/oradata/SID/*
$ setacl -m u:hvr:r x,d:u:hvr:r x $ORACLE_HOME/ora_arch
$ setacl -m u:hvr:r x,d:u:hvr:r x $ORACLE_HOME/ora_arch/*
$ setacl -m u:hvr:r x $ORACLE_HOME/ora_arch/*/*
```

   - On Solaris an extra command is needed to initialize the "default ACL";

```
$ setfacl -m u:hvr:r x $ORACLE_HOME/oradata
$ setfacl -m u:hvr:r x $ORACLE_HOME/oradata/SID
$ setfacl -m u:hvr:r x $ORACLE_HOME/oradata/SID/*
$ setfacl -m d:u::rwx,d:g::r x,d:o:   ,d:m:rwx $ORACLE_HOME/ora_arch
$ setfacl -m u:hvr:r x,d:u:hvr:r x $ORACLE_HOME/ora_arch
$ setfacl -m u:hvr:r x,d:u:hvr:r x $ORACLE_HOME/ora_arch/*
$ setfacl -m u:hvr:r x $ORACLE_HOME/ora_arch/*/*
```

Notes for ACL:

- 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 dynamically change this.
- Oracle permission **grant select any to** *hvruser* is required (see GRANTS FOR LOG-BASED CAPTURE DATABASE).

Configuring Oracle ASM

HVR supports log-based capture from Oracle databases whose redo and archive files are located on ASM storage. There are two ways to configure this:

1. For Oracle 11.2, before Oracle 11.2.0.4, HVR will read the redo and archive files directly through its SQL connection. This requires that its user has been granted the privilege **select any transaction**.
2. 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 the ASM is located under a different **$ORACLE_HOME** then that path should be defined with environment variable **$HVR_ASM_HOME**. These variables should be configured using environment actions on the Oracle location.

The password can be encrypted using the **hvrcrypt** command.

Unix and Linux

Example:

```
$ HVR_ASM_CONNECT="myuser/‘hvrcrypt myuser mypass‘@+ASM"
```

Capturing from Oracle RAC

When capturing from Oracle RAC, HVR will typically connect with its own protocol to an HVR listener. On Unix this could be configured using **inetd** daemon. This listener should be configured to run inside all cluster nodes simultaneously. The hub then connects to one of its remote locations by first interacting with the Oracle RAC ‘scan’ address.
The HVR channel only needs one location for its RAC and there is only one 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.

Grants for Trigger-Based Capture Database

HVR does trigger-based capture if action **DbCapture** is defined without parameter **/LogBased**. 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**). The database user must have the following privileges: **create session**, **create table**, **create trigger**, **create procedure** and **create sequence**.
If the HVR user needs to replicate tables which are owned by other schemas (using action **TableProperties /Schema**) then the following are also needed; **select any table**, **execute any procedure** and **create any trigger**.
Trigger-based capture will use package **dbms_alert**, unless action **DbCapture** is defined with parameter **/ToggleFrequency** or action **Scheduling** is defined with parameters **/CaptureStartTimes** or **/CaptureOnceOnTrigger**. This grant can only be given by a user with **sysdba** privilege (e.g. **oracle**).

```
$ sqlplus / as sysdba
SQL> grant execute on dbms_alert to hvruser;
```

Grants for HVR on Target Database

If HVR will integrate changes into a database, or if **HVR Refresh** will load data into a database, the HVR user will need following privileges; **create session** and **create table**. The user must be allocated a quota on its default tablespace. Use **alter user ... quota ... on ...** or **grant unlimited tablespace to ...**.
If the HVR user needs to change tables which are owned by other schemas (using action **TableProperties /Schema**) then the following grants are needed; **select any table**, **insert any table**, **update any table** and **delete any table**. Bulk refresh of tables in other schemas also needs **alter any table**, **lock any table** and **drop any table** (needed for **truncate** statements). If the target schema has triggers then **alter any trigger** privilege is also required to disable/re-enable triggers. If HVR Refresh will be used to create target tables, then the following grants are also needed: **create any table**, **create any index**, **drop any index**, **alter any table**, **drop any table**.
If action **DbIntegrate /DbProc** is defined, then **create procedure** privilege is needed.
If action **DbSequence** is defined with parameter **/Schema**, then the user should be granted **create any sequence**, **select any sequence** and **drop any sequence** privileges.

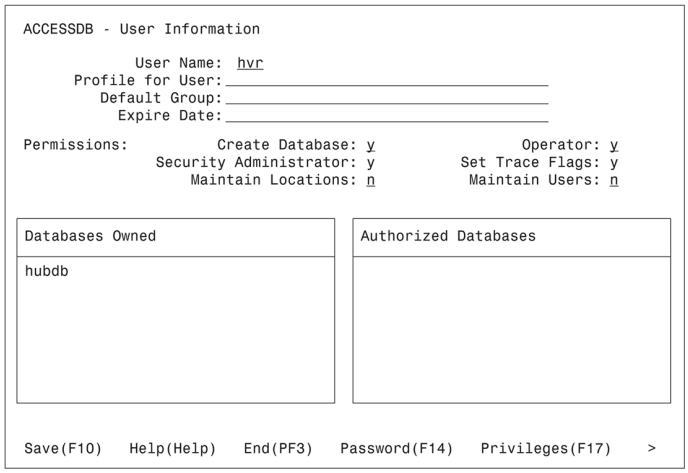
Grants for HVR on Compare or Refresh Source Database

The HVR user always needs **create session** privilege.
If **HVR Compare** or **HVR Refresh** needs to read from tables which are owned by other schemas (using action **TableProperties /Schema**) then HVR will also need **select any table** privilege.

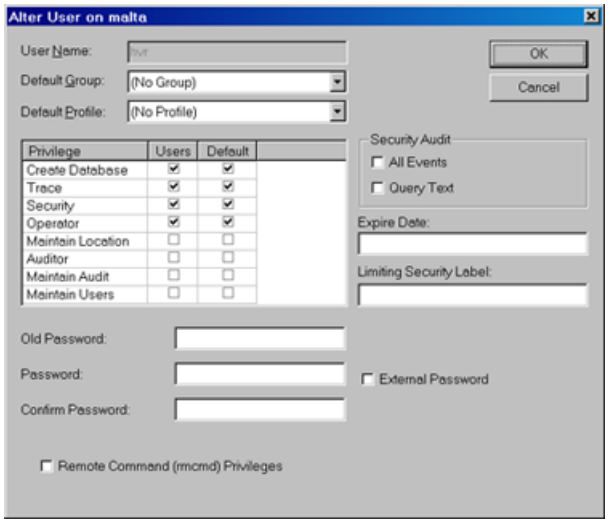
2.6 Requirements for Ingres or Vectorwise

Ingres

For an Ingres or Vectorwise 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 that 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 row wise refresh into such a database is not supported if database rules are defined on the target tables.
Accessdb permission screen:



DBA permission screen:

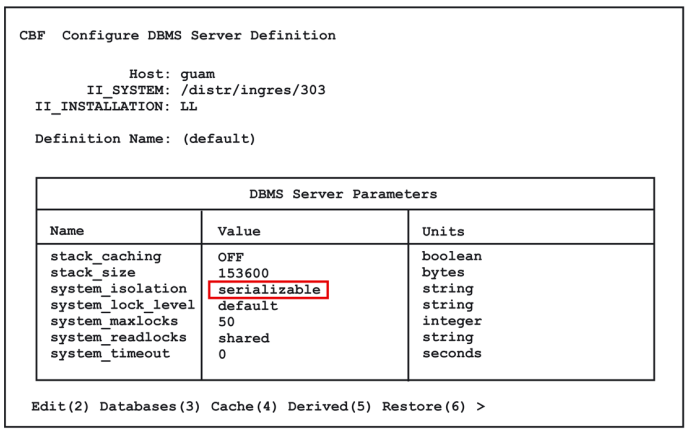


CBF screen. For trigger based capture from Ingres databases, the isolation level (parameter **system_isolation**) must be set to **serializable**. Other parameters (e.g. **system_readlocks**) can be anything.

Unix and Linux

Log based capture from Ingres requires that HVR has permission to read the internal DBMS logging files. Installing this requires a special step to create a trusted executable, which must be performed while logged in as **ingres**. See step 4 of section New Installation on Unix or Linux. This installation step is not needed if HVR runs under login **ingres**, and is also not needed if trigger based capture is used or on Microsoft Windows.
If HVR will be integrating changes into an Ingres installation on a remote machine, then a special database role must be created in that Ingres installation;

```
$ sql iidbdb < $HVR_HOME/sql/ingres/hvrrolecreate.sql
```


CBF Configure DBMS Server Definition
Host: guam
II_SYSTEM: /distr/ingres/303
II_INSTALLATION: LL
Definition Name: (default)
DBMS Server Parameters
Name
Value
Units
stack_caching
OFF
boolean
stack_size
153600
bytes
system_isolation
serializable
string
system_lock_level
default
string
system_maxlocks
50
integer
system_readlocks
shared
string
system_timeout
0
seconds
Edit(2) Databases(3) Cache(4) Derived(5) Restore(6) >

2.7 Requirements for Microsoft SQL Server

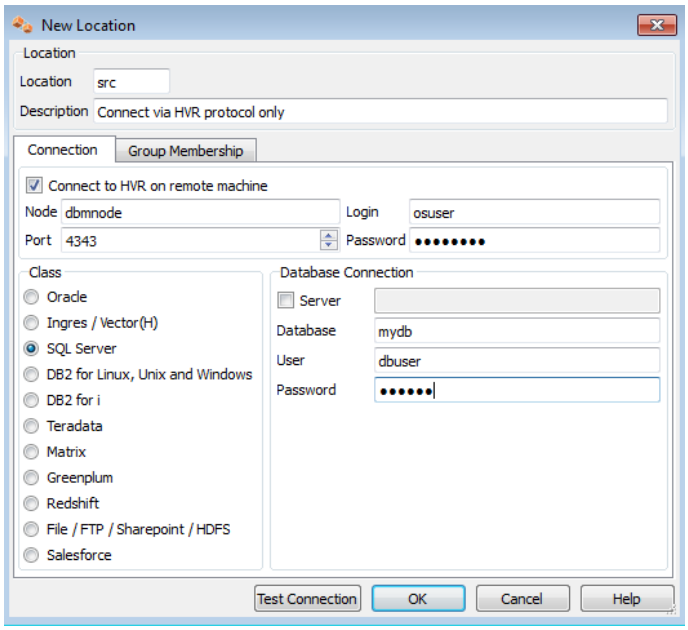
Authentication

The account used for HVR should be defined with **SQL Server authentication** or **Windows authentication**.

Connecting to SQL Server from HUB machine

HVR needs a database connection to the SQL Server source database itself. A connection to an AlwaysOn slave database is not enough.
To connect from a HVR hub machine to a remote SQL Server database there are three options:

1. 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.



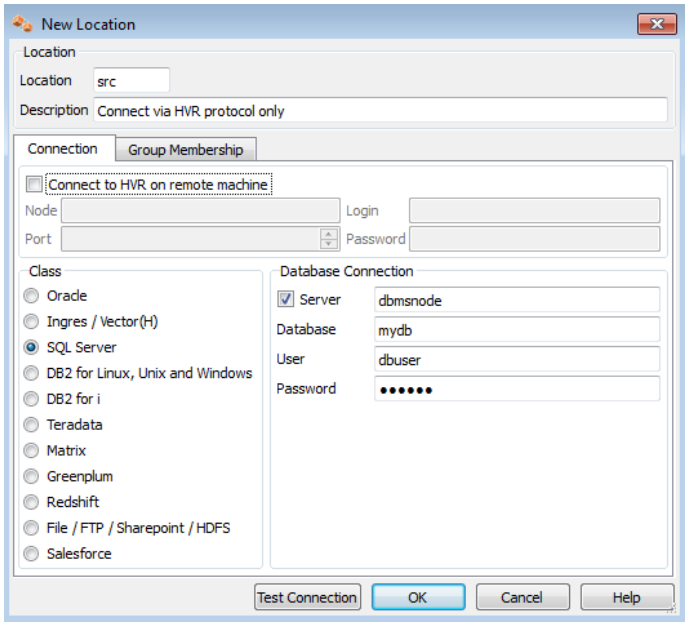
2. Connect to a SQL Server database using the SQL Server protocol (equivalent to TNS). This option is easy to setup, but cannot be used when connecting from a Unix/Linux hub to a SQL Server database because HVR's Unix/Linux release does not support direct connection to SQL Server.

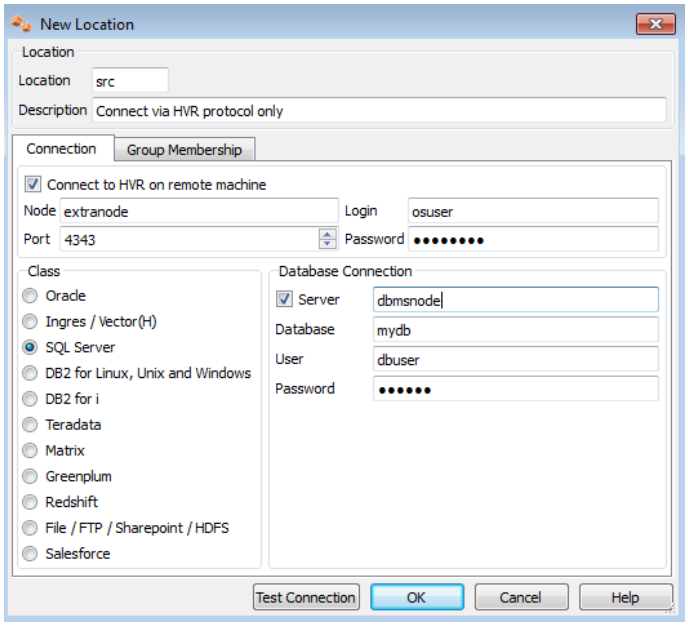
3. Connect first to a HVR installation on an extra machine using HVR's protocol (a sort of proxy) and then connect from there to a the SQL Server database machine using the SQL Server protocol (equivalent to TNS). This option is useful when connecting from a Unix/Linux hub to avoid an (intrusive) installation of HVR's software on the machine containing the SQL Server database.

All three of the above connections can be used for both capture and integration. Specifically: HVR's log-based capture can get changes from a database without HVR's executables being physically installed on the source machine.

Configuring failover for connections to AlwaysOn

If HVR is connecting using its own protocol to a database inside an AlwaysOn availability group, the following is needed:





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 AlwaysOn inside Azure an extra step is needed:
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.

Grants for SQL Server HUB database

Normally for a hub database a new SQL Server database is created. This is recommended.

Alternatively, an existing database can be used instead. In this case 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
```

Grants and steps for log-based capture database

HVR log-based capture supports three modes permissions: SysAdmin model, DbOwner model and Minimal model. The SysAdmin model is the easiest to setup and the Minimal model is the most difficult.

SysAdmin model

With this model, HVR's user should be granted a **sysadmin** role. There is no need for operators to perform special SQL statements manually.

For this model, perform only install steps 1 and 2 below; the others are unnecessary.

DbOwner model

With this model, HVR's user should be granted a **db_owner** role for the source database, but not **sysadmin** role. An operator must perform a few special SQL statements manually only when setting up a new database for capture.

For this model, perform only install steps 1-4 below; numbers 5-7 are unnecessary.

Minimal model

With this model, HVR's user needs neither **sysadmin** nor **db_owner** roles at runtime. But whenever an **HVR Load** command is run (for example to add a new table to a channel) then a user with **db_owner** privilege must perform SQL statements manually.

For this model, perform all the install steps below; numbers 1-7.

Installation steps

1. For log-based capture from SQL Server Enterprise Edition or Developer Edition, HVR requires that **SQL Server Replication Components** option is installed. This option is not required by HVR for log-based capture for SQL Server Standard Edition (but that component does not exist in Standard Edition anyway).

   This step is needed once when HVR is installed for an SQL Server instance.

2. A user with **sysadmin** must create a distribution database for the SQL Server instance, unless one already exists. To do this a user with **sysadmin** should run SQL Server wizard **Configure Distribution Wizard**. which can be run by clicking **Replication > Configure Distribution...** Any database name can be supplied (just click **Next > Next > Next**).

   This step is needed once when HVR is installed for an SQL Server instance.

   If AlwaysOn is installed then only one distribution database should be configured. Either this can be setup inside the first node and the other nodes get a distributor which points to it. Or the distribution database can be located outside the AlwaysOn cluster and each node gets a distributor which points to it there.

3. [For this step and subsequent steps the HVR binaries must already be installed].
   A user with **sysadmin** must create special 'wrapper' SQL procedures called **sp_hvr_dblog** and **sp_hvr_loghdr** so that the HVR can call SQL Server's read-only function **fn_dump_dblog** and perform SQL statement **restore headeronly**. This must be done inside SQL Server database's special database **msdb**, not the actual capture database.

The SQL to create these procedures is in file **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 grant must be done there:

```
use msdb
create user hvruser for login hvruser
grant execute on sp_hvr_dblog to hvruser
grant execute on sp_hvr_loghdr to hvruser
```

This step is needed once when HVR is installed for an SQL Server instance. But if AlwaysOn is installed then this step is needed on each AlwaysOn node.

4. A user with **sysadmin** must grant the HVR user 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 AlwaysOn is installed then this step is needed on each AlwaysOn node.

5. A user with **db_owner** (or **sysadmin**) must create three of 'wrapper' SQL procedures in each capture database so that HVR can call SQL Server's read-only procedures **sp_helppublication**, **sp_helparticle** and **fn_dblog**.

The SQL to create these three read-only procedures is in file **hvrcapdbowner.sql** in directory **%HVR_HOME%\sql\sqlserver**.

The HVR user must then be allowed to execute these procedures. The following grants must be done 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
```

This step is needed once when each new source database is being setup.

6. A user with **db_owner** (or **sysadmin**) must grant HVR a read-only privilege and also create a small state table (called **hvr_stcl**) that HVR will use during capture. A special schema is used so this table is hidden from other users.

```
use capdb
alter role db_datareader add member hvruser
```

```
create schema hvrschema
alter user hvruser with default_schema=hvrschema
create table hvrschema.hvr_stcl (x integer)
grant insert on hvrschema.hvr_stcl to hvruser
```

This step is needed once when each new source database is being setup.

7. When an **HVR Load** 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 Load** will write a script containing necessary SQL statements, and then show a popup asking for this file to be performed. The file will be written in directory **%HVR_CONFIG%\files** on the capture machine; its exact filename and the necessary permissions level is shown in the error message. The first time **HVR Load** gives this message then a user with **sysadmin** privilege must then perform these SQL statements. Subsequently these SQL statements can be performed by a user that just has **db_owner** privilege.

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 backup files to capture the older changes. HVR is not interested in full or incremental backups; it only reads backup transaction files. 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. This problem is called "transaction

archive retention". The amount of 'retention' needed (3 hours? 2 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 **msdb**. But if AlwaysOn is configured then this information source is not available on all nodes. So when HVR is used with AlwaysOn, the transaction log backups must be made on a directory which is both accessible from all AlwaysOn nodes and also from the machine where the HVR capture process is running (if this is different) via the same pathname. The HVR should be configured to find these files by defining two environment variables with HVR's **Environment** action. The first variable is **$HVR_TLOG_BACKUP_DIR** which points to a file system directory which HVR will search directly for the transaction log backup files, instead of querying **msdb**. The other variable is **$HVR_TLOG_BACKUP_FORMAT** which specifies a pattern for matching files in that directory. The pattern can contain these special characters;

```
*   Wildcard, to match zero or more characters
%d  Database name. Case insensitive.
%n  File number (in decimal) which is used by HVR to order files
%%  Matches %
```

All other characters must match exactly.

Grants for trigger-based capture database

HVR's user should be made a database owner (**db_owner** role).
For triggerbased capture, extended stored procedure **hvrevent** should normally be installed on the capture machine. This is not needed if parameters **DbCapture/LogBased** or **/ToggleFrequency** or **Scheduling/-CaptureStartTimes** or **/CaptureOnceOnTrigger** are defined. This step must be performed by a user that is a member of the system administrator role. See also section 3.2 New Installation on Windows.

Grants for HVR on target database

When HVR is making changes to a target database it supports two permission models; DbOwner privilege model, and Minimal privileges mode. But Minimal cannot use parameter **/Schema** to change tables with a different owner.

Grants for DbOwner model

In this model HVR's user should 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
```

Grants for Minimal model

In this model HVR's 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 **DbIntegrate /DbProc** is defined, then **create procedure** privilege is needed.

Grants for HVR on compare or refresh source database

When HVR is reading rows from a database (no capture) it supports two permission models: DbOwner model, and Minimal model.

Grants for DbOwner privilege mode

For this model, HVR's user should be made a database owner of the source database (**db_owner** role).

Grants for Minimal privilege mode

For this model, HVR's 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 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
```

Procedure for dropping the source database

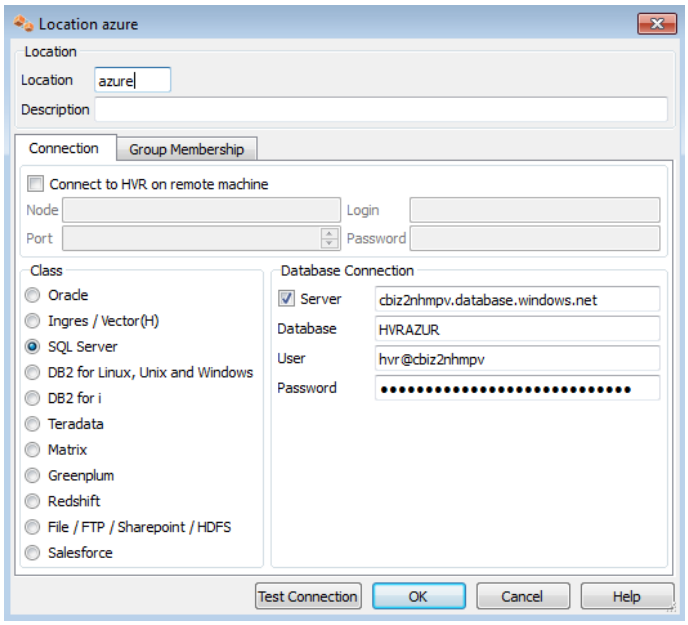
HVR log-based capture will enable the 'publish' replication option on each source database. This means that attempts to drop the database will give an SQL Server error. Additionally, HVR log-based capture may enable Change Data Capture (CDC) on source databases. This can happen when capturing data from tables with no primary key. When CDC is enabled for the database then attempts to drop the database may also give an SQL Server error because of the running CDC capture and cleanup jobs.
When command **HVR Load** 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 Load** first) the **sysadmin** must perform the following SQL statement:

```
exec sp_replicationdboption 'capdb', 'publish', 'false'
use [capdb]
exec sp_cdc_disable_db
```

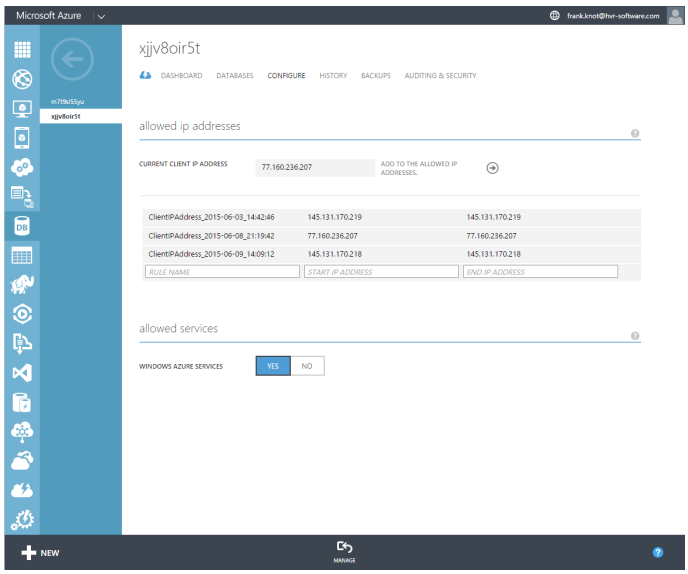
2.8 Requirements for Azure SQL

Azure SQL is the PAAS database of Microsofts Azure Cloud Platform. It is a limited version of SQL server. HVR supports Azure SQL through its regular SQL Server driver.

Configuration Notes



- When specifying the name of a Azure SQL "server name" use the full qualified name like **cbiz2nhmp.database.windows.net**.
- When specifiying the user name, add the short server name to the user name like **hvr@cbiz2nhmp**
- Log-based capture is not supported from Azure SQL. Use trigger-based capture instead.
- Burst integrate is not supported into Azure SQL. Use regular integrate.
- Azure SQL can be used for HVR's hub database
- 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 command **hvrmaint** to purge these files.
- When using HVR Refresh to HVR create tables ("**Create absent tables**") in an Azure SQL database, enable the option "**With Key**" because Azure doesn't support tables without Clustered Indexes.
- The Azure SQL database server has a default firewall preventing incoming connections. It's setting can be found under database server | configure. When connecting from an Azure VM (through an agent), enable "windows services" as "allowed services". When connecting directly from an on-premises hub, add its IP adress to the allowed range.

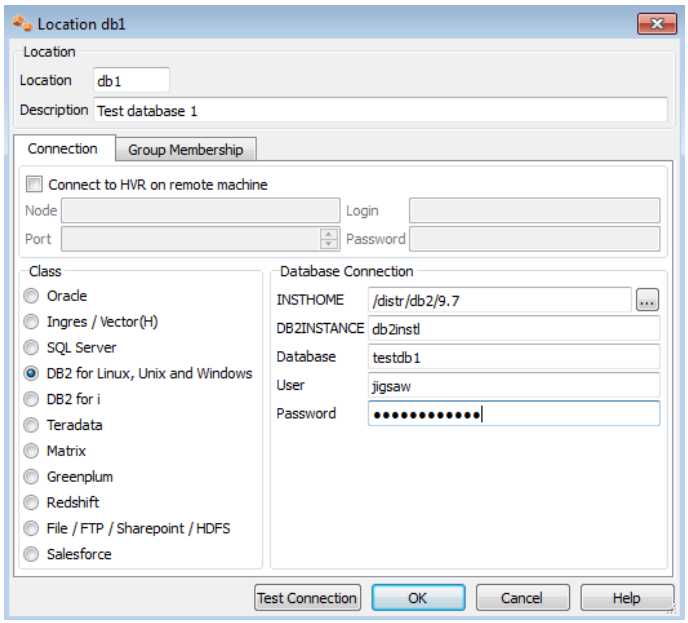

Microsoft Azure
frank.knot@hvr-software.com
xjjv8oir5t
DASHBOARD
DATABASES
CONFIGURE
HISTORY
BACKUPS
AUDITING & SECURITY
allowed ip addresses
CURRENT CLIENT IP ADDRESS
77.160.236.207
ADD TO THE ALLOWED IP ADDRESSES.
ClientIPAddress_2015-06-03_14:42:46
145.131.170.219
145.131.170.219
ClientIPAddress_2015-06-08_21:19:42
77.160.236.207
77.160.236.207
ClientIPAddress_2015-06-09_14:09:12
145.131.170.218
145.131.170.218
RULE NAME
START IP ADDRESS
END IP ADDRESS
allowed services
WINDOWS AZURE SERVICES
YES
NO
NEW
MANAGE

2.9 Requirements for DB2 for Linux, UNIX and Windows

Connection

HVR has native support for DB2 on Linux, Unix and Windows. The DB2 client should be installed on the machine from which HVR will connect to DB2.
For connections to DB2 on z/OS, the package DB2Connect should be installed on the machine from which HVR will connect to DB2.

Location Configuration



INSTHOME is the directory path of the DB2 installation.
DB2INSTANCE is the name of the DB2 instance.
Database is the name of the DB2 database.
User is the username as which to connect to DB2. This user should have the permissions specified below in order for HVR to connect, capture, compare, refresh and integrate data.
Password is the password of the user as which HVR connects to DB2.

Compare and Refresh Source

- The *hvruser* should have permission to read replicated tables

```
GRANT SELECT ON tbl TO USER hvruser
```

Integrate and Refresh Target

- The *hvruser* 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 *hvruser* should have permission to load data

```
GRANT LOAD ON DATABASE TO USER hvruser
```

- The *hvruser* should have permission to CREATE and DROP HVR state tables

```
GRANT CREATETAB ON DATABASE TO USER hvruser
```

Hub

- The *hvruser* should have permission to CREATE and DROP HVR catalog tables

```
GRANT CREATETAB ON DATABASE TO USER hvruser
```

2.10 Requirements for DB2 for i

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.

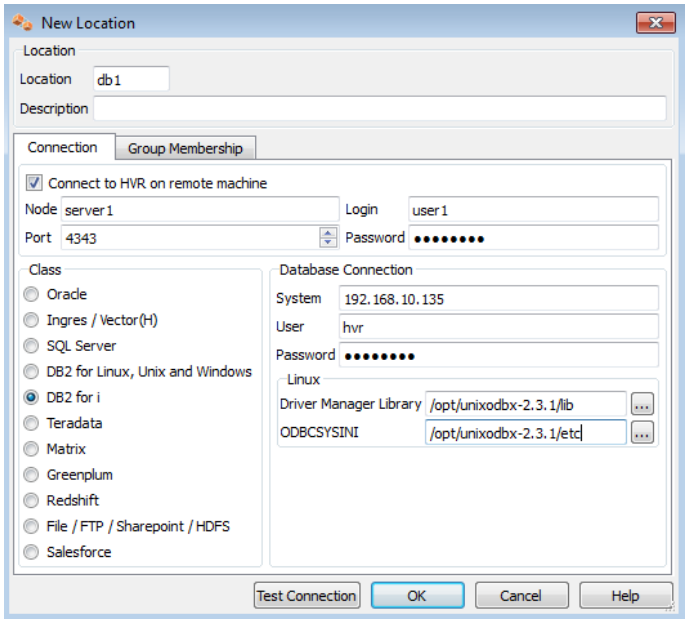
Linux

- IBM i Access Client Solutions ODBC Driver 64-bit
- ODBC driver manager UnixODBC 2.3.1

Windows

- IBM i Access Client Solutions ODBC Driver 13.64.11.00

Location Configuration



System is the hostname or ip-address of the DB2 for i system.
Named Database is the named database. 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 is the username as which to connect to the DB2 for i system. This user should have the permissions specified below in order for HVR to connect, capture, compare, refresh and integrate data.
Password is the password of the user as which HVR connects to the DB2 for i system.

Linux

Driver Manager Library is the optional directory path where the ODBC Driver Manager Library is installed. For a default installation this would be `/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 is the optional directory path where `odbc.ini` and `odbcinst.ini` are located. For a default installation this would be `/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 IBM i Access Client Solutions ODBC Driver under the heading `[IBM i Access ODBC Driver 64-bit]`

General

- The *hvruser* should have permission to read the DB2 for i system catalogs

```
GRANT SELECT ON system_catalog TO hvruser
```

Log based Capture

- The *hvruser* should have permission to select data from journal receivers

```
SELECT x.* FROM TABLE ( DISPLAY_JOURNAL (...) ) AS x
```

- All changes made to the replicated tables should be fully written to the journal receivers

IBM i Journal attribute *MINENTDTA* should be set to **NONE*
IBM i Table attribute *IMAGES* should be set to **BOTH*

- The journal receivers should not be removed before HVR has been able to process the changes written in them
- The sequence numbers used to number the changes written to the journal receivers should be increasing and should not be reset

IBM i Journal attribute *RCVSIZOPT* should contain **MAXOPT3*
When journal sequence numbers do have been reset, **hvrload** should be run to reset the capture start sequence. Optionally **hvrrefresh** could be used to repair any changes from before the new capture start that were missed

- Tables grouped in the same HVR channel, should be using the same journal (**DbCapture /LogJournal**)
- When Action **DbCapture /IgnoreSession** is used, the name of the user making a change should be logged

IBM i Journal attribute *FIXLENDTA* should contain **USR* in that case

Compare and Refresh Source

- The *hvruser* should have permission to read replicated tables

```
GRANT SELECT ON tbl TO hvruser
```

Integrate and Refresh Target

- The *hvruser* should have permission to read and change replicated tables

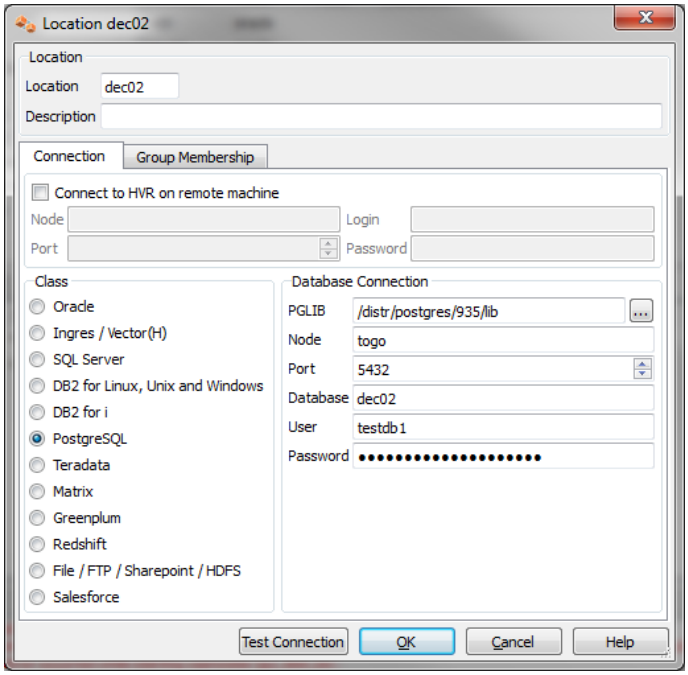
```
GRANT SELECT, INSERT, UPDATE, DELETE ON tbl TO hvruser
```

- The *hvruser* should have permission to CREATE and DROP HVR state tables
- The *current schema* of *hvruser* should have journaling enabled and tables created in that schema should automatically be journaled

Hub

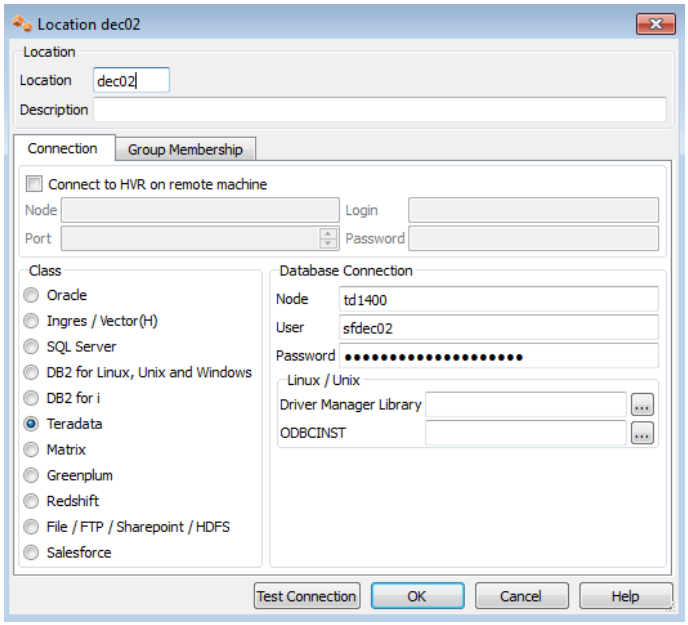
- The *hvruser* should have permission to CREATE and DROP HVR catalog tables

2.11 Requirements for PostgreSQL



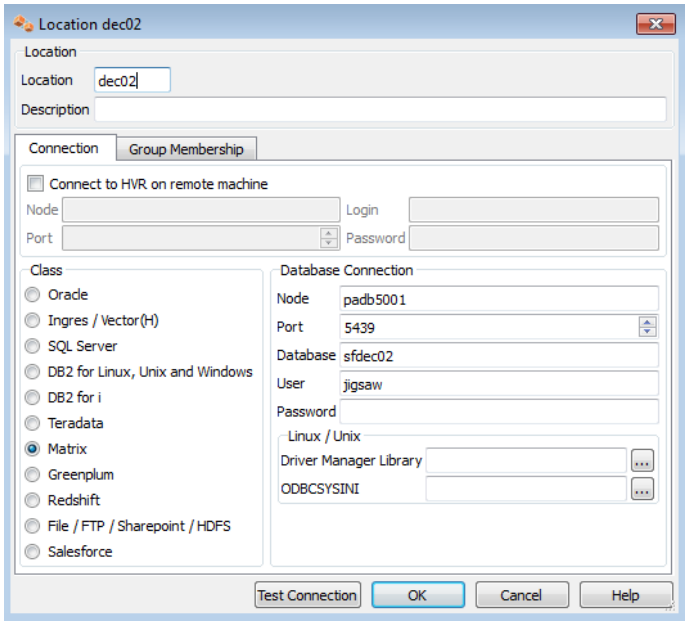
HVR supports PostgreSQL as a target database, but not as hub or capture database. HVR requires that the PostgreSQL 9 client (i.e. libpq.so.5 and its dependencies) is installed on the machine from which HVR will connect to PostgreSQL. This can be used to connect to both PostgreSQL 8 and 9 server versions.

2.12 Requirements for Teradata



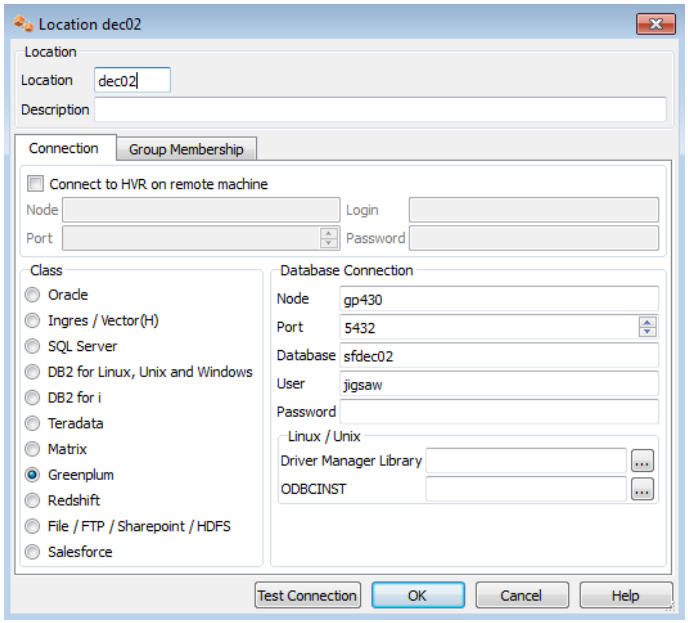
HVR requires that the ODBC driver provided by Teradata is installed on the machine from which HVR will connect to Teradata.

2.13 Requirements for Actian Matrix (Paraccel)



HVR requires that the ODBC driver provided by Matrix (Paraccel) is installed on the machine from which HVR will connect to Matrix.

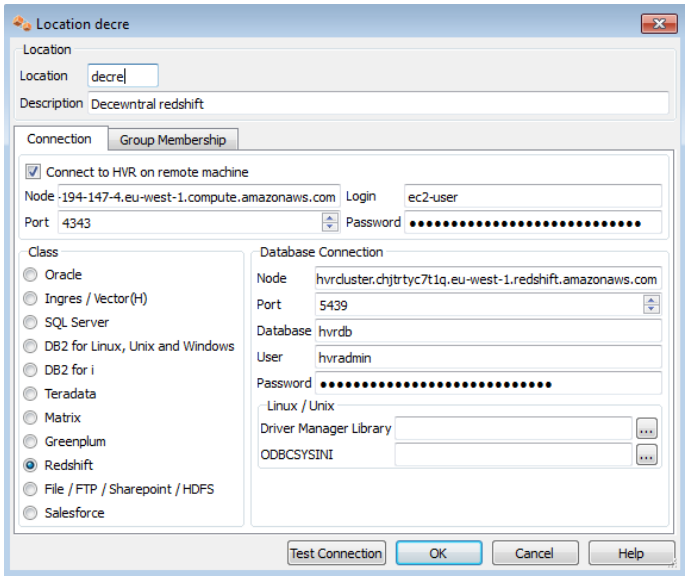
2.14 Requirements for Greenplum



HVR requires that the ODBC driver provided by Greenplum is installed on the machine from which HVR will connect to Greenplum.

2.15 Requirements for Redshift

HVR support connecting to Redshift though Redshifts Postgres ODBC interface. HVR requires that this ODBC driver is installed on the machine from which HVR will connect to Redshift. On Windows, HVR uses the Postgres 9 ODBC driver, on Linux it requires the Postgres 8 driver.



Linux Install Notes

To install the Postgres 8 driver on Linux, first install packages **unixODBC**, **unixODBC-devel** and **postgresql-libs** automatically (with **yum install** and then download and install package **postgres-odbc v8.04** manually. Postgres 8.04 can be obtained from this location:

```
ftp://bo.mirror.garr.it/pub/1/slc/slc62/x86_64/Packages/postgresql-odbc-08.04.0200-1.el6.
   x86_64.rpm
```

Check the **/etc/odbcinst.ini** file if the PostgreSQL driver is installed correctly:

```
# Driver from the postgresql-odbc package
# Setup from the unixODBC package
[PostgreSQL]
Description     = ODBC for PostgreSQL
Driver          = /usr/lib/psqlodbcw.so
Setup           = /usr/lib/libodbcpsqlS.so
Driver64        = /usr/lib64/psqlodbcw.so
Setup64         = /usr/lib64/libodbcpsqlS.so
FileUsage = 1
```

2.16 Requirements for FTP SFTP and SharePoint WebDAV

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 a network protocol for connecting 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 **FileCapture** and integration is defined with **FileIntegrate**. All other file location parameters are supported and behave normally.
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 **FileCapture** 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).
FTP and SFTP support certificates for authentication. These are held in the certificate directory **$HVR_HOME/lib/cert** (see paragraph FILES in section hvr runtime engine). The following files are included:

- **sftp_id.pub_key**: used if SFTP server wants to authenticate HVR client. New **sftp_id.*_key** files should be generated with command **ssh keygen**, not **hvrsslgen**.
- **sftp_id.priv_key** and**ca bundle.crt**:used by HVR to authenticate FTPS server. 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 **/usr/share/ssl/certs/ca bundle.crt** on Unix/Linux.
- *host***.pub_cert:** used to override **ca bundle.crt** during authentication to FTPS server *host*. These files must be created with command **hvrsslgen**.

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, SSL explicit encryption and TLS explicit encryption.
If HVR will be using a file protocol to connect to a file location (e.g. FTP, SFTP or WebDAV), then it can either connect with this protocol directly from the hub machine, or it first connect 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). 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.
An extra 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**.
A proxy server to connect to FTP, SFTP or WebDAV can be configured with action **LocationProperties /Proxy**.
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.

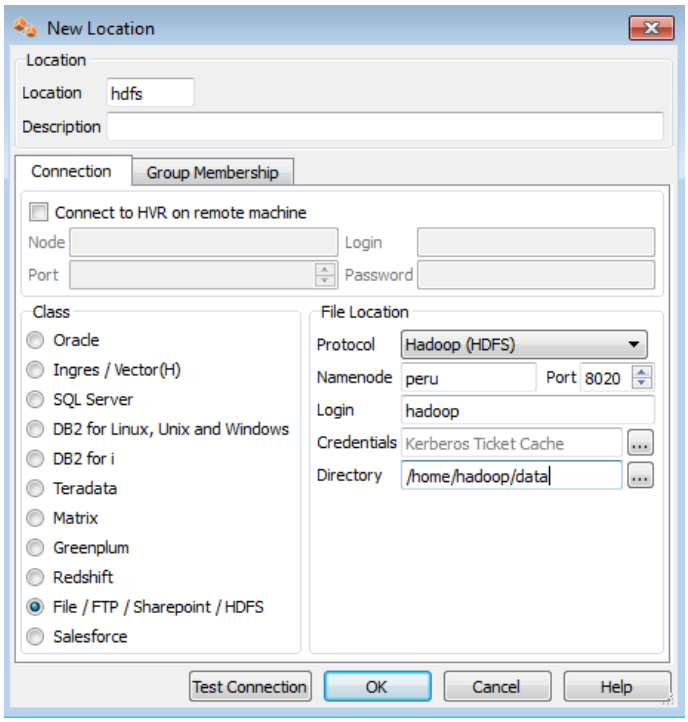
2.17 Requirements for Hadoop HDFS

HDFS locations can only be accessed through HVR running on Linux, either as the hub or as an HVR agent. Note that it is not required to run with HVR installed on the Hadoop name node although it is possible to do so.
In order to connect to HDFS HVR must have access to a Java 7 Runtime environment and Hadoop connectivity libraries. There are two ways to provide these:

- Download and install the extras package provided by HVR and install into **$HVR_HOME** separately (in the directory **$HVR_HOME/extras** that is created when you install the package):

```
$ cd $HVR_HOME
$ tar -zxvf /tmp/hvr_extras-4.7-linux_glibc2.5-x64-64bit.tar.gz
```

- If the system running HVR already contains a Java 7 Runtime environment and Hadoop connectivity libraries then these can be reused. For this scenario **$JAVA_HOME** must reference the JRE/JDK home and **$HADOOP_HOME** must point to the Hadoop installation.



2.18 Requirements for SalesForce

HVR SalesForce connection requires that the SalesForce dataloader is installed. The location of the dataloader is supplied with the SalesForce location (e.g. **C:\Program Files\salesforce.com\Data Loader\Dataloader.jar**).
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 defined with an **Environment** action.
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**.

2.19 How Much Disk Room is Needed

The full HVR distribution occupies 50Mb. For a simple situation such as channel **hvr_demo01** about 10 Mb, more disk room is needed, plus 2 Mb for the hub database and 2 Mb for each of the replicated databases. The following illustrates the disk usage on the hub machine:

HVR_HOME	Smaller than 50 Mb.
HVR_CONFIG	
files	Smaller than 1 Mb.
job	Smaller than 1 Mb.
jnl	Compressed data files, grows every time data is replicated.
log	Output from scheduled jobs containing a record of all events such as transport, routing and integration.
logarchive	Copies of logfiles 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_CONFIG	Temporary files and working directory of scheduled jobs.

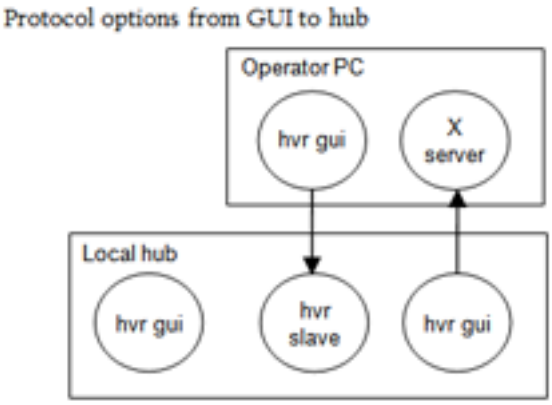
The following illustrates the disk usage on a remote location:

HVR_HOME	Smaller than 50 Mb. This can be reduced to only the commands needed on a remote location using command **hvrstrip**.
HVR_CONFIG	
files	Smaller than 1 Mb.
tmp	Temporary files if **$HVR_TMP** is not defined.
HVR_CONFIG	Temporary files.

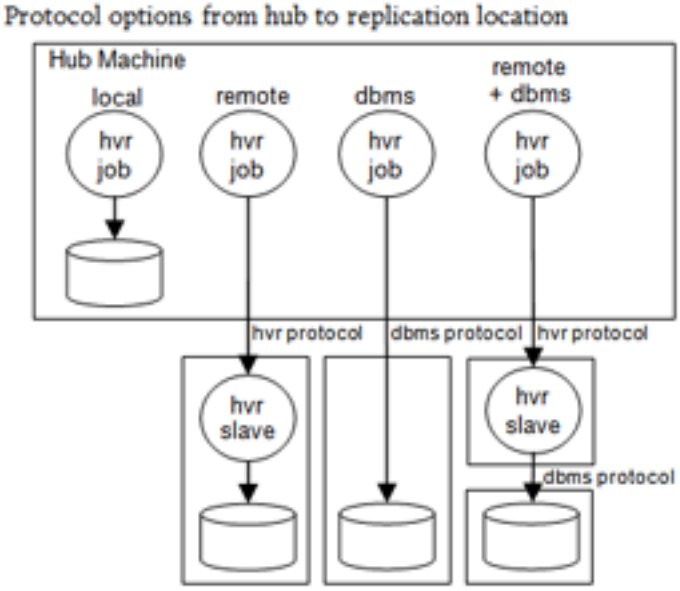
For replicating files, HVR also requires diskroom in its 'state directory'. This is normally located in subdirectory **_hvr_state** which is inside the file location's top directory, but this can be changed using parameter **LocationProperties/StateDirectory** (see section LocationProperties). When capturing 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.

2.20 Network Protocols, Port Numbers and Firewalls

To configure connections between the hub machine, the 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 his PC, in which case several TCP/IP ports must be opened from his PC to the hub machine.



To configure connections between the hub machine, the 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 his PC, in which case several TCP/IP ports must be opened from his PC to the hub machine.



Connection	TCP/IP Port Number	Network Protocol (unencrypted)	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	Not available.

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 keepalives (**SO_KEEPALIVE** is true). TCP keepalives can be disabled by setting environment variable **$HVR_TCP_KEEPALIVE** to value **0**.

On some platforms (Linux, Windows and AIX from version 5.3 onwards) 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.

2.21 How Much Network Bandwidth Will Be Used

HVR is extremely efficient over WAN and uses the minimum possible bandwidth. This is because HVR only sends changes, not the whole 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, because it knows the datatype information of each table that it replicates. 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**.

Action **LocationProperties /ThrottleKbytes** can be used to reduce the fraction of available bandwidth used. See also section LocationProperties, action **/ThrottleKbytes** for more information.

Note: The environment variable **HVR_NETSTATS_TRACE=1** traces the number of bytes sent and the number of packets sent/received. The bytes reported may be slightly higher than expected because they include internal 'control packets'. The number of such packets is constant for a given HVR script and does not depend on the number of rows transported. A typical HVR script may send 10 such packets, each containing about 20 bytes: so these 'control packets' can be ignored for network sizing purposes.

2.22 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. Three common scenarios are described below:

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.
Another way to configure high availability without relying on operating system clustering and shared discs, is to use HVR's own failover capability (see command hvrfailover).

3

Installing HVR

3.1 New Installation on Unix or Linux

Unix & Linux

The following steps should be performed as the user which HVR will run as, not as user **root**.

1. Choose directories for read only and read write parts of HVR.

```
$ HVR_HOME=/usr/hvr/hvr_home
$ HVR_CONFIG=/usr/hvr/hvr_config
$ HVR_TMP=/tmp
$ export HVR_HOME HVR_CONFIG HVR_TMP
```

   Modify your **$PATH**. For Bourne Shell and Korn Shell this is:

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

   If this installation is for a hub machine then this should be repeated in the shell startup script (e.g. **.profile**).

2. Read and uncompress the distribution file:

```
$ umask 0
$ mkdir $HVR_HOME $HVR_CONFIG
$ cd $HVR_HOME
$ gzip -dc /tmp/hvr-4.5.0-linux_glibc2.5-x64-64bit.tar.gz | tar xf -
```

3. If this is a hub machine, then install the HVR license file. This file is named **hvr.lic** and is normally delivered to each customer by HVR Technical Support.

```
$ cp /tmp/hvr.lic $HVR_HOME/lib
```

   This license file is only required on the hub machine.

4. If HVR must perform log-based capture from Ingres then a trusted executable must be created so it can read from the Ingres logging system. Perform the following steps while logged in as the DBMS owner (**ingres**):

   Ingres

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

   This step is not needed if capture will be trigger-based or if capture will be from another machine. It is also not needed if HVR is already running as the DBMS owner (**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;

   Ingres

```
$ chmod g+rX $HVR_HOME
$ chmod  R g+rwX $HVR_CONFIG
```

5. If this is a remote machine or a hub machine to which operators will be connecting using the HVR GUI, an HVR listen port must be configured. Pick an arbitrary TCP/IP port number between **1024** and **65535** which is not already in use. Number **4343** is recommended. Login as **root** and configure the **inetd** daemon to invoke HVR. For an HVR installed under **/usr/hvr/hvr_home** with TCP/IP port **4343**, add the following line to **/etc/services**:

```
hvr 4343/tcp
```

   For Unix the following line should be added to **/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
```

   On Linux however a new file should be created named **/etc/xinet.d/hvr** which should contain the following contents:

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

Now force the **inet** daemon to reread its configuration. On Linux this is done with command **/etc/init.d/xinetd force reload**. On other Unix machines, send signal **SIGHUP (kill 1)** to the process id of the **inetd** daemon.

Notes on setup:

- Option **–r** tells **hvr** to run as a remote slave process.
- The values used for **HVR_HOME**, **HVR_CONFIG** and **HVR_TMP** are for the current machine not the hub machine.
- For encryption add options **–C***pair* at the end of the **/etc/inetd.conf** line or to the end of the **server_args** line in file **/etc/xinet.d/hvr**. See section hvr runtime engine.
- For Solaris version 10 and higher, file **/etc/inetd.conf** must be imported into SMF using command **inetconv(8)**.
- For an alternative to connecting with the **inetd** daemon, see section hvrremotelistener. This is necessary if the Linux **XINETD** package is not installed (this is the case for RHEL5). It can also be used if **root** privilege is unavailable or if password authentication cannot be configured. The following command uses options **–i** (interactive: do not run as daemon) and option **–N** (skip password authentication) to listen on port 4343.

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

- If the HVR installation is not for a hub machine, then unnecessary files can be removed using command **hvrstrip**.

Notes on password validation:

- On some machines **hvr** needs the PAM option **–p** (Pluggable Authentication Module) so it can check the passwords properly. For example, add option **–plogin** at the end of the **/etc/inetd.conf** line or to the end of the **server_args** line in file **/etc/xinet.d/hvr**.
- Custom password validation can be configured by copying file **$HVR_HOME/lib/hvrvalidpw_example** to **$HVR_HOME/lib/hvrvalidpw**. Option **–A** should be added at the end of the **/etc/inetd.conf** line or to the end of the **server_args** line in file **/etc/xinet.d/hvr**. And the **inetd** should be reconfigured so that HVR is not run as **root**. Command **hvrvalidpw** can then be used to create private username passwords which are stored in **$HVR_HOME/lib/hvrpasswd**.

Notes on firewalls:

- On some machines 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 SELinux. To see the status use command **sestatus(8)**. To disable, edit **/etc/selinux.conf** so it contains **SELINUX=disable** and then reboot the machine.

Notes on testing:

- Test the connection from a different machine using command **hvrtestlistener**.

```
$ hvrtestlistener node 4343
```

A username and password can also be tested as follows;

```
$ hvrtestlistener -L user/pwd node 4343
```

6. HVR GUI can either run on a separate PC and connect using a HVR remote connection to the hub or it can run on the hub machine itself and connect using X to an X server on the user's pc. In the first case, HVR GUI will run on the user's PC and another hvr install must be performed there. HVR GUI will only run on Windows and Linux.

7. If HVR will be integrating changes into a Ingres installation on a remote machine, then a special database role must be created in that Ingres installation;

Ingres

```
$ sql iidbdb < $HVR_HOME/sql/ingres/hvrrolecreate.sql
```

8. If this is a hub machine, then define environment variable **$HVR_HUB_CLASS** as a default DBMS type for the hub database. Valid values are **oracle**, **ingres** and **db2**. This can be done editing the shell startup script (e.g. **.profile**). See also Calling HVR on the Command Line.

3.2 New Installation on Windows

Windows

1. If this is a hub machine, or if this is a machine where HVR will do log-based capture and will use command **hvrlogrelease**, then Perl must be installed. This can be downloaded from http://www.activestate.com/activeperl. If Perl is installed after HVR, then the HVR Scheduler service must be recreated.

2. Login to Microsoft Windows under a normal user account.

3. Install distribution. This can be done by running the installation wizard **hvr 4.5.0 windows x64 64bit setup.exe**.

Then answer the installation wizard's questions to install the runtime objects.

4. If this is a hub machine, then install an HVR license file. This file is named **hvr.lic** and is normally delivered to each customer by HVR Technical Support.

```
C:\> copy c:\tmp\hvr.lic %HVR_HOME%\lib
```

This file need only be installed in directory **%HVR_HOME%\lib** on the hub machine.

SQL Server

5. For trigger-based capture from SQL Server (not Microsoft Azure), a special extended stored procedure can be created called **hvrevent**. This procedure is not necessary for a capture machine if certain capture parameters are defined (**DbCapture/LogBased** or **/ToggleFrequency** or **Scheduling/CaptureStartTimes** or **/CaptureOnceOnTrigger**). 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  dmaster  Uuser  Ppwd
> exec sp_addextendedproc 'hvrevent', 'c:\hvr\hvr_home\bin\hvrevent.dll'
> go
> grant execute on hvrevent to public
> go
```

Notes:

- 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**.

6. If this is a hub machine, then define environment variable **HVR_HUB_CLASS** as a default DBMS type for the hub database. Valid values are **oracle**, **ingres**, **sqlserver** and **db2**. This can be done with **Start ▶ Control Panel ▶ System ▶ Advanced ▶ Environment Variables**. See also Calling HVR on the Command Line.

7. If this is for an HVR remote location, or if this is a hub machine which the HVR GUI will connect to from a different machine, then the HVR Remote Listener service must be installed.

   Pick an arbitrary TCP/IP port number between 1024 and 65535 which is not already in use. Number 4343 is recommended. Create and start the HVR Remote Listener service with this port number.

   Notes:

   - The HVR Remote Listener service can also be created on the command line. See section hvrremotelistener.
   - Custom password validation can be configured by copying file **%HVR_HOME%\lib\hvrvalidpw_example** to **%HVR_HOME%\lib\hvrvalidpw**. Option **–A** should be added to hvrremotelistener options and that service should not be run as **Administrator**. Command **hvrvalidpw** can then be used to create private username passwords which are stored in **%HVR_HOME%\lib\hvrpasswd**.
   - To test the connection from a different machine use command **hvrtestlistener**.

     ```
     $ hvrtestlistener node 4343
     ```

   - If the HVR installation is not for a hub machine, then unnecessary files can be removed using command **hvrstrip**.
   - If the HVR Remote Listener is for a remote location space which is part of a Windows cluster, then the HVR Remote Listener should be created with option **-c** to connect it to the cluster group.

3.3 New Installation of HVR Image for Azure

Azure

HVR provides **HVR Image for Azure**, a pre-configured Azure Windows image containing HVR's remote listener agent including necessary drivers to connect to SQL Server and Oracle databases. HVR Image for Azure only supports use as a remote HVR agent (for capture or integration). It does not support use as a hub machine. To use HVR as a hub in Azure follow the manual install steps for a New Installation on Windows.
The **HVR Image for Azure** needs to be prepared for use in your Azure environment once and can be used to create HVR remote listener agents indefinitely.

Getting and preparing the HVR for Azure Image

The **HVR for Azure Image** is made available by HVR Software as VHD file already in the Azure cloud. To get the Image available for use in your own Azure environment, perform the following steps once:

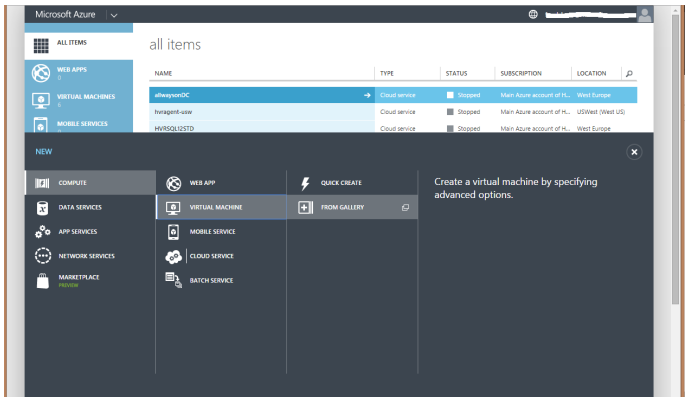
1. Contact HVR Software to obtain the storage URL location and corresponding access key for the HVR Image for Azure.
2. Copy the VHD file to the desired Azure location using **AZCopy**. **AZCopy** is a commandline utility of Microsoft which can be downloaded from their site. Remember to copy it to a location in the desired Region (matching the SQL database or HDinsight location).
3. Use the "Create Image from VHD" in Azure to create a new image for the VHD file.
4. Repeat these steps for every region you want to use HVR in.

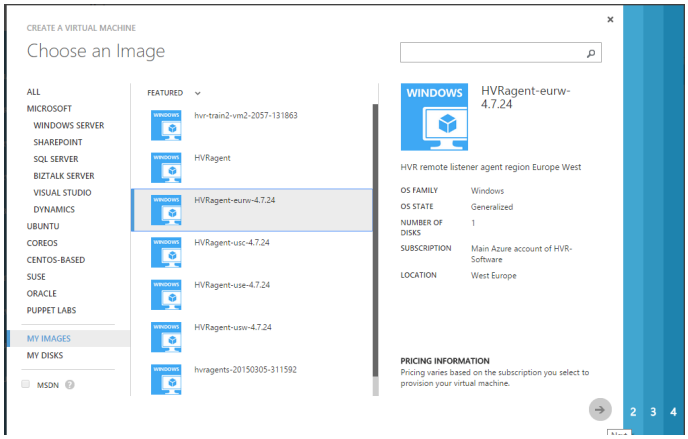
The image is now available in your private Image Gallery.

Using the HVR for Azure Image to creat a remote listener agent

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

1. In the Azure console, choose **+/Compute/Virtual Machine/From Gallery**. Select the **HVR Image for Azure** and make sure to pick the location where the database resides:





2. Create an HVR agent VM in the same Region as the database or HDInsight environment. A standard tier **A1 instance** VM is sufficient to run HVR's agent.
3. HVR uses port 4343 for incoming connections to the HVR listener agent(**endpoint**). In the creation process open this port for incoming traffic:
4. When the VM status is moved from 'provisioning" to "running", the agent is ready for use. Test the agent by entering its credentials in an (on-premises) hub:

3.4 Configuring Encrypted Network Connections

An HVR connection to a remote HVR location can be configured so that communication over the network is encrypted. If necessary, each HVR location in an HVR channel can be given its own private key and public certificate pair. Also the hub machine can be given its own private key and public certificate pair so that the locations can verify their incoming connection.
To allow the hub to verify the identity of each remote location, supply the location's public key and private certificate to the HVR slave process and then, 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, on the hub, supply the hub's public key and private certificate using parameter **LocationProperties /SslLocalCertificateKeyPair** and on the remote location point to a copy of the hub's public certificate in the XML HVR access file.
HVR can generate the private key and public certificate on the hub machine, or on the remote location.

Simple Configuration Steps

These steps set up a secure HVR network connection using the standard private key and public certificate files which are delivered with HVR in **$HVR_HOME/lib/cert**.

1. Configure a remote connection to an HVR location. The setup steps for this are described in either section New Installation on Unix or Linux or New Installation on Windows depending on which operating system is used.

2. Setup HVR on the remote HVR location to expect an encrypted connection. HVR on the remote machine consists of an HVR executable with option **–r** which is configured to listen on a specific port number. The command line for this process will be configured in one of the following ways, depending on the Operating System and how the existing remote connection is configured.

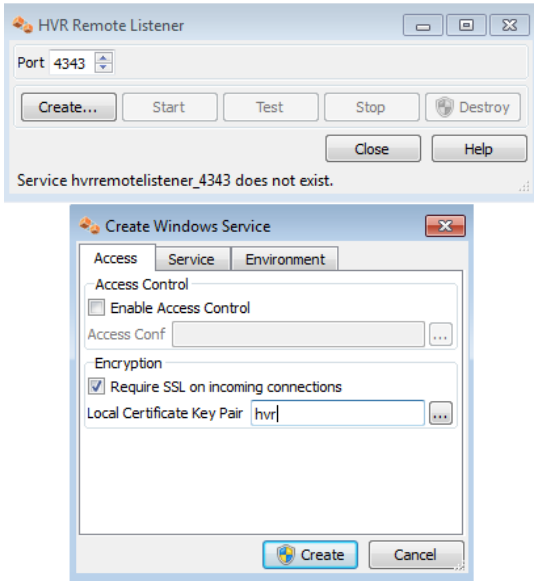
   On Unix, edit file **/etc/inetd.conf**:

   ```
   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 -Khvr
   ```

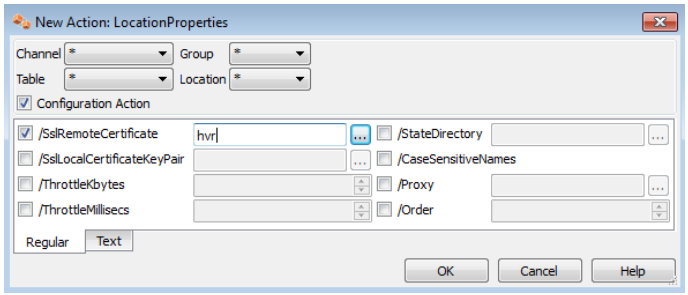
   On Linux, add the following line into file **/etc/xinetd.d/hvr**:

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

   On Windows, use the following dialog:



3. Register the public certificate file in the channel configuration using the action **LocationProperties /SslRemoteCertificate**.



4. Regenerate the relevant job scripts:

```
$ hvrload -oj -ldec02 hubdb mychn
```

Advanced Configuration Steps

These steps set up a secure HVR connection using a newly generated private key and public certificate:

1. Configure a remote connection to an HVR location. The setup steps for this are described in either section New Installation on Unix or Linux or New Installation on Windows depending on which operating system is used.

2. Generate a private key and public certificate pair. The command to do this is as follows:

```
$ hvrsslgen [-opts] nm subj
```

   This command generates two files, a file named *nm*.**priv_key** and a file containing the corresponding public certificate named *nm*.**pub_cert**. The second argument *subj* is written into the subject field of the X509 public certificate file. Both files are needed on the remote machine, only the public certificate file must be copied to the hub machine.

   Option **–a***bits* can be used to generate an asymmetric (RSA) key pair with length *bits*. The default is 1024.

   Option **–s***bits* is required to use a symmetric key with length *bits* during symmetric encryption. The default is 128.

3. Setup HVR on the remote HVR location to use the newly generated private key and public certificate. HVR on the remote machine consists of an HVR executable with option **–r** which is configured to listen on a specific port number. The option **–C** should be added to specify the public certificate and private key respectively. If these files are not located in **$HVR_HOME/lib/cert** then the absolute pathnames must be supplied.

4. Place the public certificate file on the hub machine, and register it in the channel configuration using the action **LocationProperties /SslRemoteCertificate**.

5. Regenerate the relevant job scripts:

```
$ hvrload -oj -ldec02 hubdb mychn
```

Encryption Implementation

When encryption is activated for an HVR location, every byte sent over the network (in either direction) is encrypted with the Transport Layer Security (TLSv1) protocol. An RSA public/private key pair is used for authentication and session startup. The public key is embedded in an X509 certificate, and the private key is encrypted using an internal password with a 3DES algorithm. After the session is initiated, encryption switches to a symmetric cipher using a RC5 algorithm. By default the keys used for this asymmetric negotiation are 1024 bits long and the symmetric key is 128 bits long, although longer key lengths can be specified when the public/private keys are being generated.

The hub database guards against third parties impersonating the remote HVR location (e.g. by spoofing) by comparing the SHA1 checksums of the certificate used to build the secure connection and its own copy of the certificate.

Public certificates are self–signed. HVR checks that the hub at the remote machines' copies of this certificate are identical so signing by a root authority is unnecessary.

For network encryption, HVR uses OpenSSL, which is developed by the OpenSSL Project (http://www.openssl.org).

Notes

HVR always encrypts passwords before sending them of the network, regardless of whether and encrypted configuration is configured. HVR will also encrypt all data for file replication if SFTP or FTPS protocol is used.

After generation, it is wise to restrict read access to the private key files using an Operating System command, e.g.:

```
$ chmod 600 nm.priv_key
```

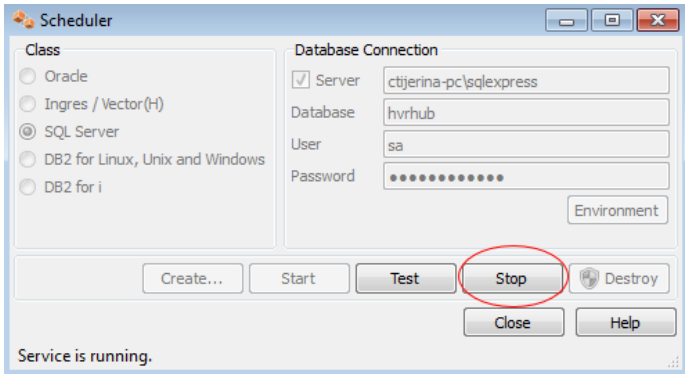
Unix & Linux

HVR's encryption will attempt to exploit the entropy (randomness) generation capability supplied by the **/dev/urandom** device. This is not absolutely necessary, but for optimal security it is recommended that this Operating System option is installed, especially on the machine used to generate public/private key pairs.

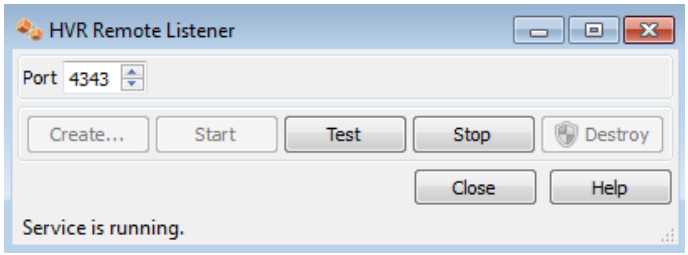
3.5 Installing an HVR Upgrade

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.
Normally all machines should 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 this is a hub machine, then stop the HVR Scheduler:



2. If the HVR Remote Listener service is running, it should 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 performing command **slibclean** as **root**. This step is not needed for other flavors of Unix or Linux.

5. Read and uncompress the distribution file.

   Unix & Linux

   Under Unix and Linux this is done by:

   ```
   $ cd $HVR_HOME
   $ umask 0
   $ gzip -dc /tmp/hvr-4.5.0-linux_glibc2.5-x64-64bit.tar.gz | tar xf -
   ```

   On Microsoft Windows this is performed using the installation wizard:

```
C:\> d:\hvr-4.5.0-windows-x64-64bit-setup.exe
```

6. 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
   ```

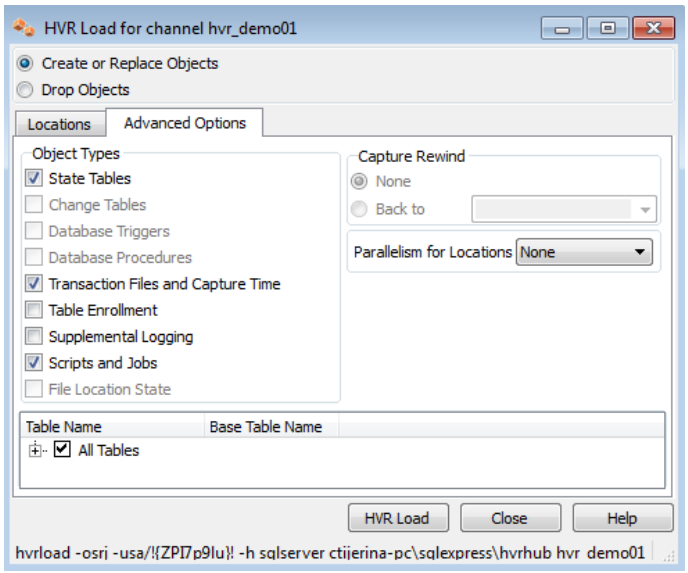
   This step is not needed if capture will be trigger–based or if capture will be from another machine.

   If HVR and **ingres** share the same Unix group, then the permissions can be tightened from 4755 to 4750.

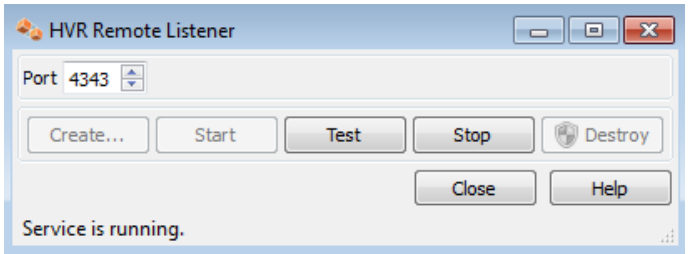
7. For certain bug fixes it may be necessary to regenerate the job scripts and enroll files for each channel. Note that this upgrade step is seldom needed; only if it is explicitly mentioned in the HVR Release Notes.

   This can be done on the command line as follows:

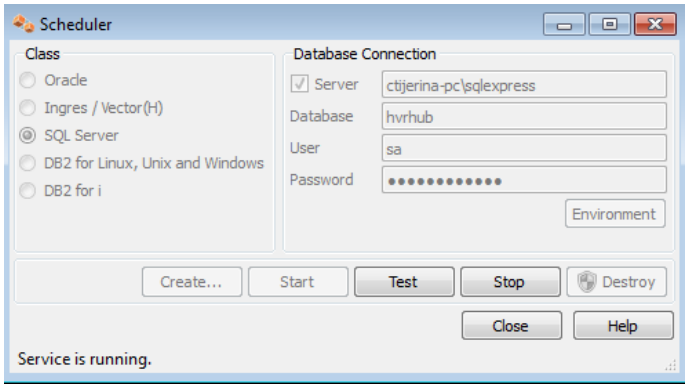
   ```
   $ hvrload -oej hubdb/password mychn
   ```



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



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


Scheduler
Class
Oracle
Ingres / Vector(H)
SQL Server
DB2 for Linux, Unix and Windows
DB2 for i
Database Connection
Server
ctijerina-pc\sqlexpress
Database
hvrhub
User
sa
Password
Environment
Create...
Start
Test
Stop
Destroy
Close
Help
Service is running.

4

After Installing HVR

After installing HVR the operator may need to make changes to the user profiles and to start HVR after a reboot. A special maintenance script can also be installed for housekeeping HVR's scheduler. There may also be changes necessary for backup and archiving of the Ingres or Oracle database.

4.1 Restarting HVR after Unix Reboot

Unix & Linux

Script **hvr_boot** starts and stops HVR server processes defined in file **/etc/hvrtab**. This ensures that these processes are restarted when a machine is rebooted. Each line of this file refers either to an HVR Scheduler (if the second field is a database name) or an HVR Remote Listener (if the second field is a port number).
On most Unix and Linux machines this script should be copied to the **init.d** or **rc.d** directory. On newer Solaris versions however the script must be registered in Solaris's System Management Facility (SMF).
This script should only be executed by **root**.
Note that use of **hvr_boot** to stop HVR server processes at system shutdown is optional; they will stop anyway!
For an Oracle RAC, script **hvr_boot** can be enrolled in the cluster with command **crs_profile**.

File Format

Each line of **/etc/hvrtab** contains four or more fields, separated by spaces. These fields have the following meaning: *username hub_or_port hvr_home hvr_config [options].*
The first field is the Unix username under which HVR runs. The second field is the name of the hub database (for an HVR Scheduler) or a TCP/IP port number (for a HVR Remote Listener). This second field can also have special value **hvrfailover** which invokes command **hvrfailover boot** at reboot.
The third and fourth fields are the values of **$HVR_HOME** and **$HVR_CONFIG**. Other fields are optional, and are passed to the HVR process as options.
Lines starting with a hash (#) are treated as comments.

Example

```
# This /etc/hvrtab file starts a Remote Listener and two HVR schedulers
# (one for an Oracle hub and one for an Ingres hub).
# Note use of command hvrcrypt to encrypt the Oracle password.
root hvrremotelistener_4343 /opt/hvr/hvr_home /opt/hvr/hvr_config
hvr user/!{DszmZY}! /opt/hvr/hvr_home /opt/hvr/hvr_config -EHVR_TMP=/tmp -EORACLE_HOME=/opt/
    oracle -ORACLE_SID=prod
hvr inghubdb /opt/hvr/hvr_home /opt/hvr/hvr_config -EHVR_TMP=/tmp -EII_SYSTEM=/opt/ingres -
    EHVR_PUBLIC_PORT=50001
```

Boot Dependencies

Starting the HVR Scheduler should be dependent on starting the DBMS first, otherwise the HVR Scheduler will fail immediately when it tries to connect to the hub database.
For non–Solaris machines which use an **init.d** or **rc.d** directory this is done with the start and stop sequence number in the boot filename. The start sequence of HVR must be bigger than the start sequence of the DBMS and the stop sequence of HVR used must be smaller than the stop sequence of the DBMS.
The INSTALLATION STEPS below use start sequence **97** and stop sequence **03** (except HP–UX which uses **997** and **003** because it expects three digits).
For SMF, the dependency can be defined by editing file **hvr_boot.xml** and replacing string **svc:/milestone/multi–user–server** with the name of the DBMS service, e.g. **svc:/application/database/oracle**.

Installation Steps

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 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 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, for 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;

```
$ 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/application/manifest/hvr_boot.xml
svc> quit
$ svcadm enable svc:/application/hvr_boot
$ svcs -a|grep hvr     # Check the service is running online 16:00:29 svc:/application/
    hvr_boot:default
```

4.2 Restarting HVR after Windows

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.
For example, suppose the scheduler service is called **hvrscheduler_hvr4** and the DBMS service is called **OracleServiceOHS**, then use the following command:

```
$ sc config hvrscheduler_hvr4 depend=OracleServiceOHS
```

To see the service name, use the following command:

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

4.3 Adapting a Channel for DDL Statements

HVR only captures inserts, updates and deletes (known as DML statements) as well as truncate table (modify to truncated) and replicates these to each integrate database. But DDL statements, such as create, drop and alter table, or bulk copy (copy from) are not captured by HVR. When DDL statements are used, the following points 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 adapt a channel:

1. 'Online Adapt'. This is the least disruptive, because during this procedure users can still make changes to all tables, but it requires more steps.
2. 'Offline Adapt'. This is the most disruptive, because users are not allowed to make changes to any of the replicated tables.

The following steps do not apply for bi–directional replication (changes travelling in both directions). For bi–directional replication, please contact HVR Technical Support for minimal–impact adapt steps.

Online Adapt Steps (Log-Based Capture)

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
$ hvrtrigger -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. 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 Select** 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 have **Only in Channel**.

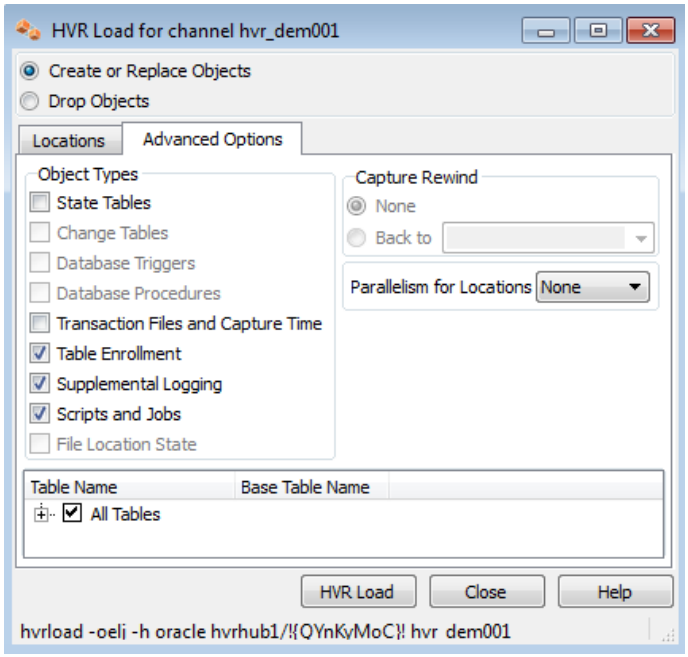
4. Use **Table Select** to the integrate locations to check that all the tables have value **Same** in the **Match** column.

5. Regenerate the enroll information (for all tables), the replication jobs & scripts and enable DBMS logging for the new tables in the capture database.

   This can also be done on the command line as follows:

```
$ hvrload -oelj hubdb chn
```

6. Use **hvrrefresh** 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. The 'Online Refresh' option **–q** is needed.



For the first target database **–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.

```
$ hvrrefresh -gb -qrw -rdb1 -ldb2 -tdm01_product -h oracle hvrhub/!{!b/Q.KqR}! hvr_demo01
```

For any extra target database use option **–qwo** because the capture job should not skip any changes, but the integrate jobs should apply changes which occurred during the refresh using resilience.

```
$ hvrrefresh -qwo -rdb1 -ldb2 -tdm01_product -h oracle hvrhub/!{!b/Q.KqR}! hvr_demo01
```

For an Ingres target database, HVR bulk refresh will sometimes disable journaling on affected tables. If **hvrrefresh** gave this warning, then it is necessary to perform **ckpdb +j** on each target database to re–enable journaling.

7. If any fail tables exists in the integrate databases (**/OnErrorSaveFailed**) for the tables which have had columns added or dropped, then these fail tables must be dropped.

```
$ hvrload -oc -lint1 -lint2 -ttbl1 -ttbl2 hubdb chn
```

8. Trigger 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 Adapt Steps

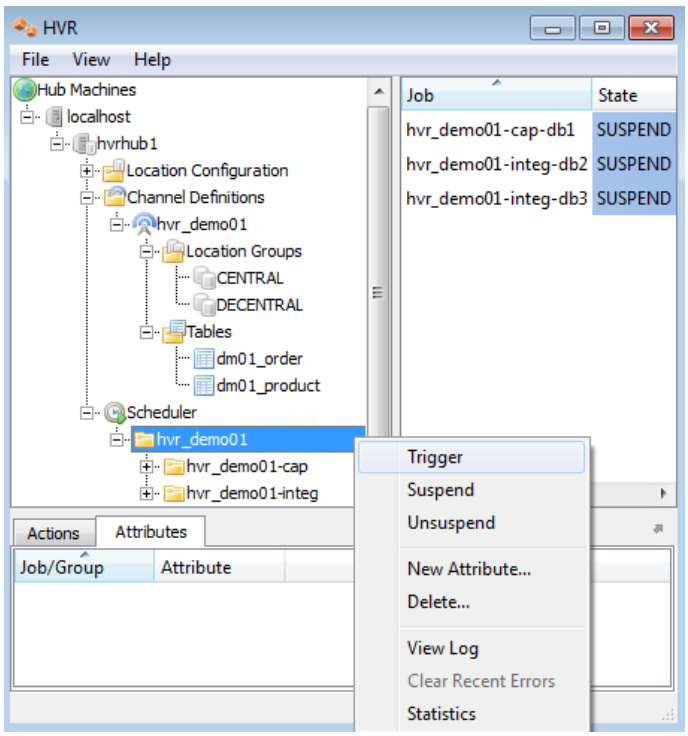
1. Start downtime. Make sure that users cannot make changes to any of the replicated tables.
2. Suspend the replication jobs

```
$ hvrsuspend hubdb chn
```

3. Unload the channel:

```
$ hvrload -d hubdb chn
```

4. Run the SQL script with the DDL statements against both the source and target databases.
5. Use the HVR GUI **Table Select** 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 Select** to the integrate locations to check that all the tables have value **Same** in the **Match** column.

7. Reload the channel:

```
$ hvrload hubdb chn
```

8. Use **hvrrefresh** to synchronize all the tables that are affected by the SQL script in step 4 (except for the tables that were only dropped). This includes tables that were only affected by DML statements in this script.

```
$ hvrrefresh -r src -ttbl1 -ttbl2 hubdb chn
```

The **–t** options can also just be omitted, which will cause all replicated tables to be refreshed.

For an Ingres target database, HVR bulk refresh will sometimes disable journaling on affected tables. If **hvrrefresh** gave this warning, then it is necessary to perform **ckpdb +j** on each target database to re–enable journaling.

9. Unsuspend the capture and integrate jobs:

```
$ 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.

4.4 Regular Maintenance and Monitoring for HVR

Periodically it is necessary to restart the HVR Scheduler, purge old logfiles 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**. See also section hvrmaint.
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.

4.5 Managing Oracle Archive/Redo Logfiles

Oracle

HVR can capture changes straight from Oracle's redo and archive files.
If log–based capture is defined for an Oracle database (action **DbCapture /LogBased**) 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. See section hvrlogrelease.

4.6 Checkpointing an Ingres Capture Database

Ingres

- 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.
  Under 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 **DbCapture /LogBased**) 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.

5

Command Reference

HVR can be configured and controlled either by using the GUI or by using the command line interface. This chapter contains a section describing each HVR command and its parameters. Most HVR commands take a hub database name as first argument. For more conventions see section Calling HVR on the Command Line.

Commands

Command	Description
hvr runtime engine	HVR runtime engine
hvradapt	Select base table definitions and compare with channel information
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
hvrcrypt	Encrypt passwords
hvrfailover	Failover between Business Continuity nodes using replication
hvrgui	HVR Graphical User Interface
hvrload	Load a replication channel
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
hvrrouterview	View or extract contents from internal router files
hvrscheduler	HVR Scheduler server
hvrstatistics	Extract statistics from HVR scheduler logfiles
hvrsuspend	Suspend (or un–suspend) jobs
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
hvrtrigger	Trigger jobs

5.1 Hvr runtime engine

Name

hvr – HVR runtime engine.

Synopsis

hvr [**–E***n*=*v*]... [**–t***x*] [*script* [*–scropts*] [*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 slave 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 section hvrproxy.

Options

–a*accessxml*
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 **inetd** service should be configured to start the HVR slave 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*
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	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 slaves 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 slave to service remote HVR connections. On Unix the **hvr** executable is invoked with this option by the **inetd** 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*	Internal slave co–process. HVR sometimes uses slave 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 operators can distinguish between slave 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 slave so it only accepts 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 **inetd** 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).
│ ├─ **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).
│ ├─ **sftp_id.pub_key**	Used if SFTP server wants to authenticate HVR client. New **sftp_id.*_key** files should be generated with command **ssh–keygen**, not **hvrsslgen**.
│ ├─ **sftp_id.priv_key**	same as pub_key
│ ├─ **ca–bundle.crt**	Used to override **ca–bundle.crt** during authentication to FTPS server *host*. These files must be created with command **hvrsslgen**.
│ └─ **ca–bundle.crt**	Used by HVR to authenticate FTPS server. 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 **/usr/share/ssl/certs/ca–bundle.crt** on Unix/Linux.
├─ **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 **hvrvalidpw_example**.
├─ **hvrvalidpw**	Optional script for password validation. If HVR detects this script, it invokes it for password validation instead of attempting this validation internally.
├─ **hvrvalidpw_example**	Sample custom password validation.
└─ **hvrscripthelp.html**	Description of HVR's internal script syntax and procedures.

HVR_CONFIG	
├─ **job**	HVR scripts generated by **hvrload**.
├─ **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.

5.2 Hvradapt

Name

hvradapt – Select base table definitions and compare with channel information

Synopsis

hvradapt [**-f***fname*] [**-i***ign*]... [**h***class*] –**l***loc* [**n***tbltmpl*]... [**u***user*] [**-R**] [**-r***tbls*]... [**-s***scope*] [**-S***scope*] [**-x**] –*hubdb chn*

Description

Command **hvradapt** will compare the base tables in a database with the table information for a channel. It will then either add, replace or delete table information (in catalogs hvr_table and hvr_column) so this information matches.
This is equivalent to the Select Table dialog in the HVR GUI.
If the adapt location contains a table which is not in the channel but is matched by the template it is added to the channel. If a table is in the channel but is not matched by the template, then it is deleted from the channel. And if a table is both matched by the template and included in the channel, but has the wrong column information in the channel, then this column information is updated. If no template is supplied (no –n option), then no tables are added or deleted; only existing column information is updated where necessary.

Options

–**f***fname*	Append modified or added HVR table names to file
–**i***ign*	Ignore data type (**d**) key (**k**) or extra column (**x**) differences
–**h***class*	Specify hub database. Valid values are **oracle**, **ingres**, **sqlserver** and **db2**
–**l***loc*	Specifies the adapt location, typically the channel's capture location
–**n***tbltmpl*	Specifies a table template for the channel. This template defines which base tables in the adapt location should be included in the channel. Multiple templates can be supplied; each can be either a base table name, a pattern (such as **mytbl***), or an absolute path to a file containing table names and/or patterns.
–**R**	Do not re-describe tables
–**r***tbls*	Re-describe only specific tables
–**s***scope*	Add **TableProperties /Schema** action to *scope*. Valid values for *scope* are all (**a**), group (**g**) or location (**l**). The default is all (**a**).
–**S***scope*	Add **ColumnProperties /Absent** to *scope* instead of updating the column information. Valid values for *scope* are group (**g**) or location (**l**). The default is not to add these **ColumnProperties /Absent** actions.
–**u***user[/pwd]*	Connect to hub database using DBMS account *user*. For some databases (e.g. SQL Server) a password must also be supplied.
–**x**	Check only mode. Do not apply any changes to the catalogs

Example

The following example demonstrates the use of a shell script to automatically enroll new or modified tables. First, a template file (e.g. **/tmp/adapt.tmpl**) is created with the contents:

```
order                   # Include tables order,
order_lines             # order_lines,
product                 # product,
product_details         # and product_details
schema1.*               # All tables in schema1
schema2.tbl1            # Table in non-default schema
history_*               # Pattern for dynamic tables
!tmp_*                  # Negative pattern to exclude tables
```

A template file entry can be a table name (e.g. `tab1` or `owner.tab2`) or a pattern (`tab*`, `*.*` or `!tmp*`). The special symbol '`+`' matches all tables already in the channel. Table names with special characters must be double quoted (`"a b"`). Empty lines and hash comments (`#Test`) are ignored.

Subsequently, the script below runs **hvradapt** to check for new or modified tables in location **loc1**. If **hvradapt** finds any, the script executes the necessary commands to enroll the tables in the channel **mychn** and perform a refresh.

```
#!/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

if test -f $F                                       # If file $F exists then new or changed
    tables were detected
then
    hvrsuspend $hub $chn-integ                      # Stop integrate jobs
    hvrload -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
    hvrtrigger -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
```

5.3 Hvrcatalogexport, hvrcatalogimport

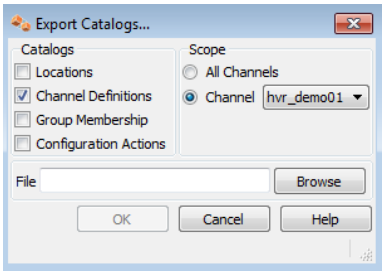
Name

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).
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. This can be an Oracle, Ingres, SQL Server or DB2 database depending on its form. See further section 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

–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** and **db2**. 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 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 (see also section 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

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hvr_catalog SYSTEM "http://www.hvr-software.com/dtd/1.0/hvr_catalog.dtd">
<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>
```

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.

5.4 Hvrcompare

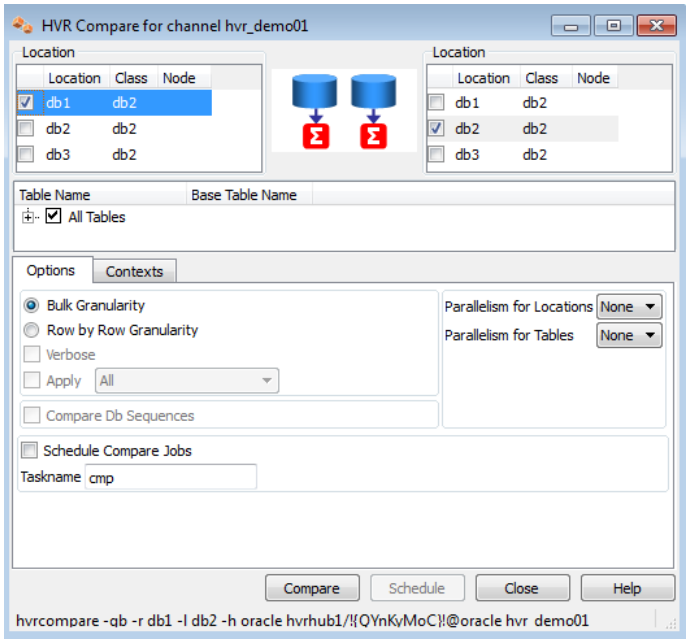
Name

hvrcompare – Compare data in tables.

Synopsis

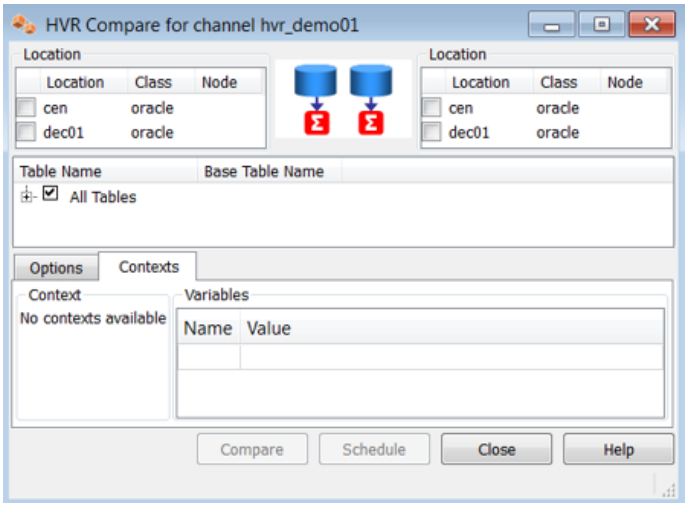
hvrcompare [*–options*] *hubdb chn*

Description



Command **hvrcompare** compares the data in 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; this can be an Oracle, Ingres, SQL Server or DB2 database depending on its form. See further section Calling HVR on the Command Line.
Table compare can be performed row–by–row or as a bulk operation, depending on which **–g** option is supplied. Bulk compare involves calculating the checksum for each object and reporting whether the replicated tables are identical.
Row–wise compare causes data to be piped from one location to the other location where each individual row is compared. This results in a list of a minimal number of inserts, updates or deletes needed to resynchronize the tables.

Options



–C*ctx*

Enable context *ctx*. 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 /RefreshCondition="{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**.
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

Remove (drop) scripts and scheduler jobs & job groups generated by previous **hvrcompare** command.

–g*x*

Granularity of compare operation in database locations. Valid values of *x* are:

b	Bulk compare using checksums. This is the default.
r	Row–wise compare of tables.

–h*class*

Specify hub database. Valid values are **oracle**,**ingres, sqlserver** and **db2**. See also section Calling HVR on the Command Line.

–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.
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 verbose option (**–v**) will not generate SQL for it. The **–m** option can only be used with row–wise granularity (option **–gr**).

–p*N*

Perform compare on different locations in parallel using *N* sub–processes. This cannot be used with option **–s**.

–P*M*

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.

–s

Schedule invocation of compare scripts using the HVR Scheduler. Without this option the default behavior is to perform the compare immediately. The jobs created by this option are named *chn*–**cmp**–*l1*–*l2*. These jobs are initially suspended. These jobs can be invoked using command **hvrtrigger** as in the following example:

```
$ hvrtrigger -u -w hubdb chn-cmp
```

The previous command unsuspends the jobsand instructs the scheduler to run them. Output from the jobs is copied to the **hvrtrigger** 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 on the command line (without HVR Scheduler) as follows:

```
$ hvrtrigger -i hubdb chn-cmp-loc1-loc2
```

–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 **/RefreshCondition** parameter contains string {**hvr_var**_*nm*}. This string is replaced with *val*.

Example

For bulk compare of table **order** in location **cen** and location **decen**:

```
$ hvrcompare -rcen -ldecen -torder hubdb/pwd sales
```

Notes

If **hvrcompare** is connecting between different DBMS types, then an ambiguity can occur because of certain datatype 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 **–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.

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.

5.5 Hvrcontrol

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.

HVR sends control files internally in these areas;

- Command **hvrtrigger** 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 **hvrtrigger –w** commands will terminate (otherwise they would keep hanging).
- Online refresh jobs (**hvrrefresh –q**) send **online_refr** 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.
- If **DbIntegrate /OnErrorBlockLoc** is defined, and if an integrate error occurs then the integrate job creates a **block_loc** control file for itself, so that all new changes from that location are written to the fail table. This **block_loc** control file can be removed manually or using command **hvrcontrol**.

Pending control files can be viewed using command **hvrrouterview** with option **–s**.

Options

–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 **–r** or **–s**.
–F	Affected jobs should finish at the end of the next replication cycle.
–h*class*	Specify hub database. Valid values are **oracle**,**ingres, sqlserver** and **db2**. See also section 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: *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 to **hvrcontrol**.
–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** 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** 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** 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**, –**r** or –**s**.
–u*user [/pwd]*	Connect to hub database using DBMS account *user*. For some databases (e.g. SQL Server) a password must also be supplied.
–w*where*	Where condition which must have form colname op val. 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 **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**, –**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 **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 **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**.

5.6 Hvrcrypt

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. It 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 **hvrload** 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

–h*class*	Specify hub database. Valid values are **oracle**,**ingres, sqlserver** and **db2**. See also 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=hvrhubaw
$ DBPWD=mypassword
$ 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 **hvrload**.

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 /SslCertificate**.

See Also

Parameter **/SslCertificate** in section LocationProperties.

5.7 Hvrfailover

Ingres
Unix & Linux

Name

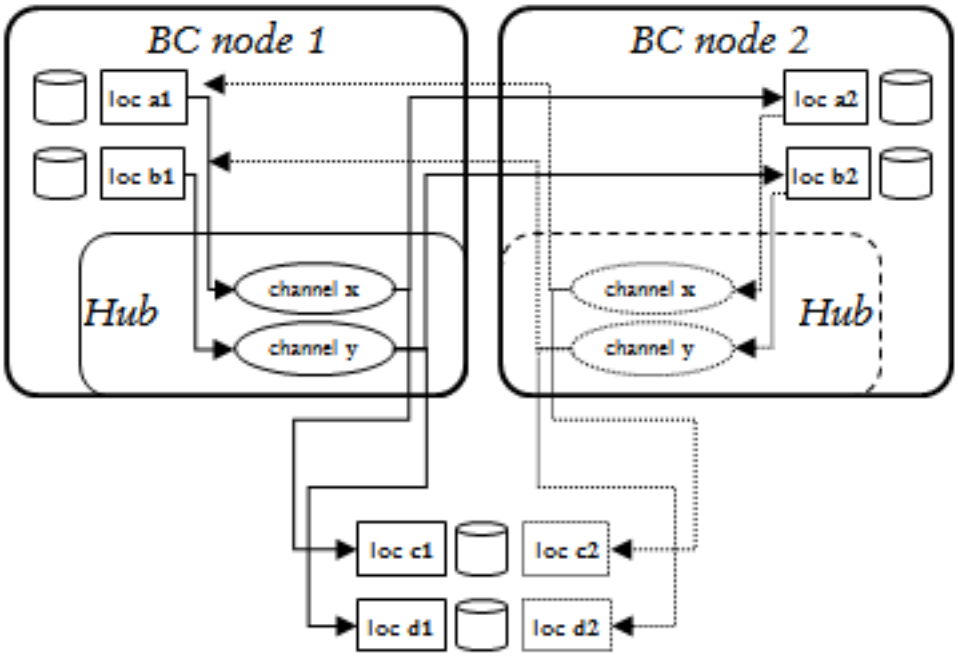
hvrfailover – Failover between Business Continuity nodes using replication

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 **hvrload**. 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;

–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.

Installation

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.

5.8 Hvrgui

Name

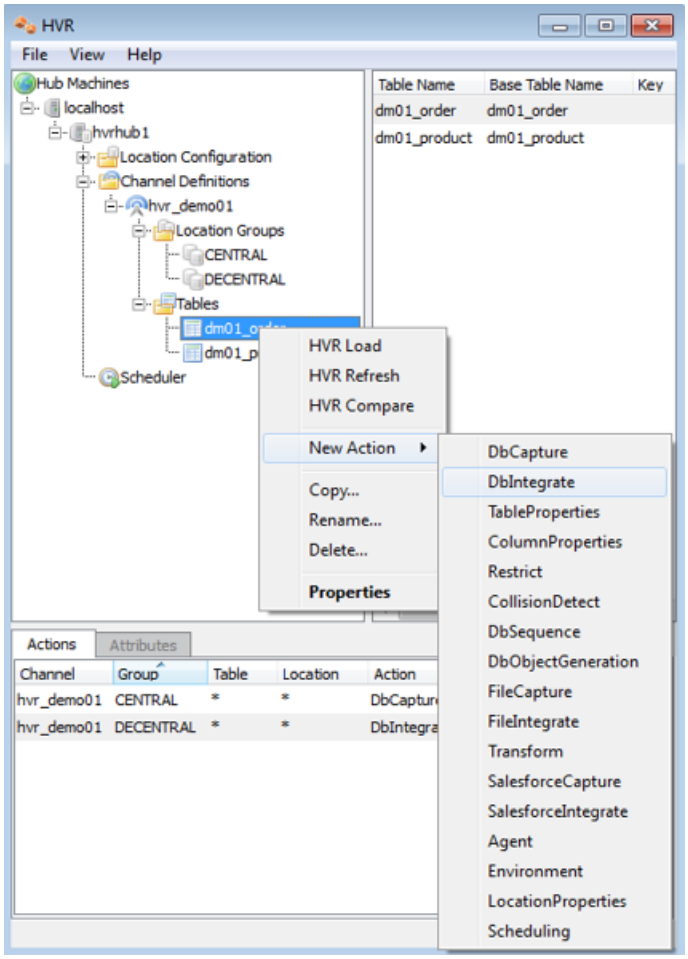
HVR GUI – HVR Graphical User Interface.

Synopsis

hvrgui

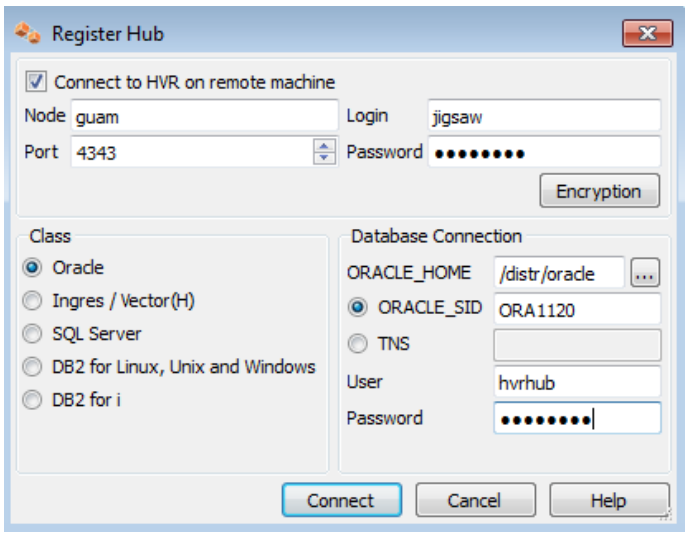
Description

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. At startup the **Register Hub** dialog window requires you to input data for connecting to the hub database.



The main window consists of four panes. The top–left pane contains a treeview with channel definitions and location configuration. The top–right pane displays details from the node selected in the treeview. The **Actions** pane lists the actions configured for the channel selected in the treeview. The bottom pane provides logfile information and error information.
After registering a hub, you can see folders for **Channel Definitions** and **Location Configuration**. A right–click 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 Load**, **Export**, **Import**
- Create an action for that object



Note

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.

5.9 Hvrload

Name

hvrload – Load a replication channel.

Synopsis

hvrload [*–options*] *hubdb chn*

Description

hvrload 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. This can be an Oracle, Ingres, SQL Server, or DB2 database depending on its form. See further section Calling HVR on the Command Line.

Options

–d
Drop objects only. If this option is not supplied, then **hvrload** 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 **hvrload –d**.

–h*class*
Specify hub database. Valid values are **oracle**,**ingres, sqlserver** and **db2**. See also section Calling HVR on the Command Line.

–i*time*
Capture rewind. Initialize channel to start capturing changes from a specific time in the past, rather than only changes made from the moment the **hvrload** command is run. Capture rewind is only supported for database log–based capture using parameter **/LogBased** (not for trigger–based capture) and for capture from file locations when parameter **/DeleteAfterCapture** is not defined.
Valid formats are *YYYY–MM–DD [HH:MM:SS]* in local time or **today** or **now**[[*+*|*–*]*SECS*] or an integer (seconds since **1970–01–01 00:00:00 UTC**).
For example, **–i "2010–07–29 10:30:21"** or **–i now–3600** (one hour ago).

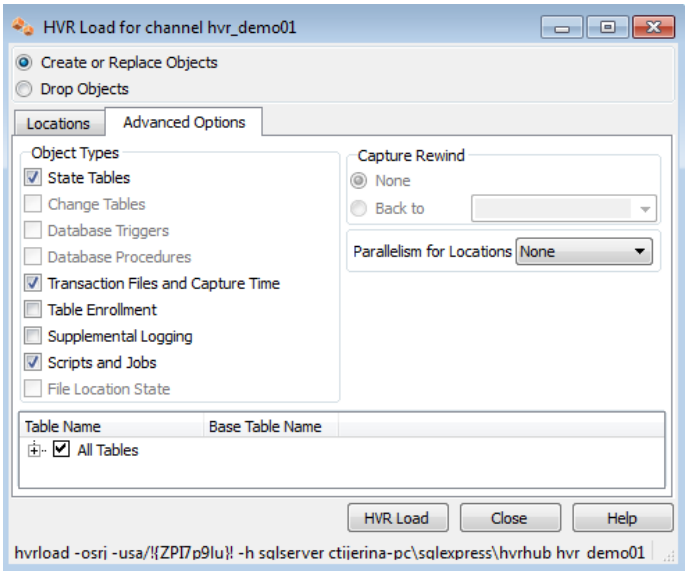
–l*x*
Only affect objects for locations specified by *x*. Values of *x* may be one of the following:

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 **hvrload**.

–o*S*

Operations limited to objects indicated by *S*.



Values of *S* may be on 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	Enroll information for tables with log–based capture.
l	Supplemental logging for Oracle 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 **hvrload** to effect all object types indicated. Not specifying a **–o** option implies all objects are affected (**–sctpreljf**).

–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**.

–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 **hvrload**.

–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

- **HVR_HOME**
  - **bin**
    - **hvrload**

HVR_CONFIG	
files	
***.logrelease**	Which log–based capture journals or archive files have been released by the capture job.
jnl	
hub	
chn	
YYYYMMDD	Journal files created by action **DbIntegrate /Journal**.
job	Directory containing generated job scripts. Some jobs use static scripts instead.
router	Directory containing replication state.
sqlgen	Directory containing temporary files used during generation of SQL.

See Also

Commands hvrproxy, hvrrouterview and hvrscheduler.

5.10 Hvrlogrelease

Name

hvrlogrelease – Manage DBMS logging files when not needed by log–based capture.

Synopsis

hvrlogrelease [*optfile*] [*–options*]

Description

hvrlogrelease can be used to manage DBMS logging files (Oracle archive files or Ingres journals) 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/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 **$HVR_CONFIG/files** and contain timestamps which allows **hvrlogrelease** to see which DBMS logging files could still be needed by HVR replication.
When **hvrlogrelease** is making private copies of the DBMS logging files (option **–purge** not defined) then it creates these files in a special directory which it creates inside the DBMS directory tree. The files are copied using a file system 'hard link' that avoids i/o overhead. For example, if HVR saw that the DBMS was writing archive logs to **$ORACLE_HOME/flash_recovery_area/***sid*/**archivelog** then **hvrlogrelease** would create a private directory **$ORACLE_HOME/flash_recovery_area/***sid*/**archivelog_hvr** and start creating and deleting hard links to the redo files in the original directory. The private directory used by HVR can be overridden by defining an **Environment** action to set **$HVR_LOG_RELEASE_DIR** before running **hvrload**.
Either option **–oracle_sid** or **–ingres_db** must be specified.
Note that **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.

Options

–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 **$HVR_CONFIG** should be defined with special options **–oracle_sid** and **–hvr_config** respectively.
–hvr_config=*dir*	Check for 'log release' files in this **$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**.
–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**.
–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
```

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**.

Examples

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"
```

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:

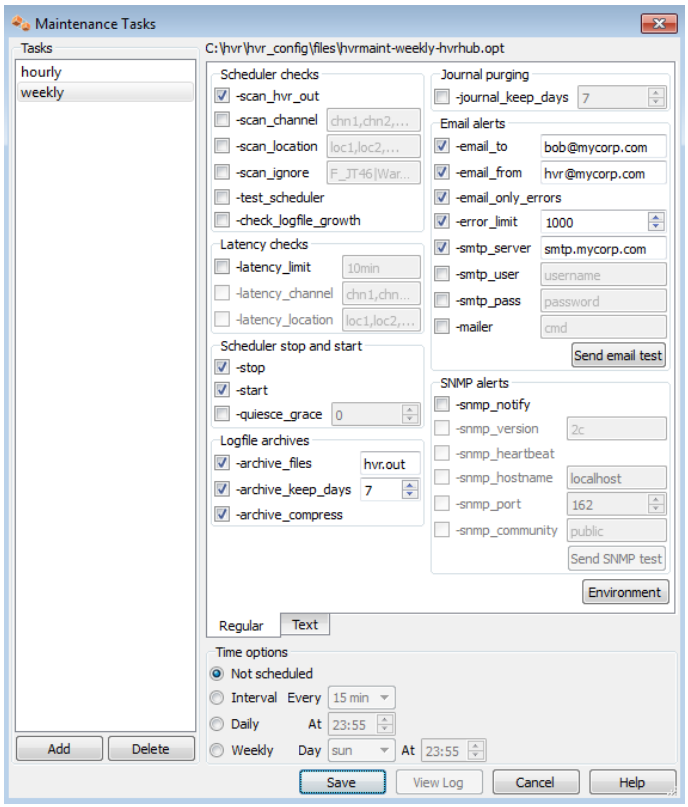
```
-env=HVR_HOME=/opt/hvr410/hvr_home
-hvr_config=/opt/hvr410/hvr_config
-ingres_db=mydb
-email_from=hvr@prod.mycorp.com
-email=bob@mycorp.com;jim@mycorp.com
-email_only_errors
```

Notes

Command **hvrlogrelease** must be scheduled to run under the DBMS's login, e.g. **oracle** or **ingres**. This is different from **hvrmaint** which must run under 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.

5.11 Hvrmaint

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 behavior of **hvrmaint** is controlled by its options. These can be supplied on its command line, or they can be bundled into a file *optfile*, which must be its first argument.

One way to use **hvrmaint** is to schedule it on its own with options **–stop** and **–start**, perhaps nightly or each weekend. These options instruct **hvrmaint** to restart the HVR Scheduler. Often other options would be supplied such as **–scan_hvr_out**, (scan log files for HVR errors) or **–archive_dir** (move old log files to archive directory). Option **–email** can be used to send an email to operator(s) that summarizes the status. When used in this way **hvrmaint** could be scheduled on Unix using **crontab**, and on Windows as a Windows Scheduled Task.

A second way to use **hvrmaint** is to run it frequently (perhaps every half hour) with options **–ping**, **–check_logfile_growth** and **–scan_hvr_out**. These options check that the HVR Scheduler is responding, that its logfile is growing and that no errors have been written. Running **hvrmaint** this way does not interrupt the HVR Scheduler. Option **–email_only_errors** can be supplied so that emails are only sent if a problem is found.

The last way to use **hvrmaint** is as part of a larger nightly or weekly batch script, perhaps a 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).

Options

–archive_files=*patt*	Move any files in directory **$HVR_CONFIG/log/***hub* with pattern *patt* to the archive directory. Files which are not matched by *patt* are deleted.
–archive_keep_days=*N*	Removed files in archive directory after *N* days. Requires option **–archive_files**. If this option is not specified, then archived files are kept indefinitely.
–archive_compress	Compress HVR Scheduler log files while moving them to the archive directory. This option is not supported on Windows.
–check_logfile_growth	Check that logfile **hvr.out** has grown in size since the last time **hvrmaint** was run. If this file has not grown than an error message will be written.
–env=*NAME=VALUE*	Set environment variable. This option can be repeated to set multiple variables such as **$HVR_HOME**, **$HVR_CONFIG**, **$HVR_TMP**, **$II_SYSTEM**, **$ORACLE_HOME** etc. . .
–email_to=*addr1[;addr2]*	Email output from **hvrmaint** output to *addr1* [and *addr2*]. Requires either option **–smtp_server** or option **–mailer**. Multiple email addresses can be specified, either using a single **–email_to** option with values separated by a semicolon or using multiple **–email_to** options.
–email_only_errors	Only send an email if **hvrmaint** encountered an error itself or detected an HVR error while scanning **hvr.out**.
–email_from=*addr*	Specify a sender address in email header.
–error_limit=*N*	Limit the number of HVR errors reported to *N*. This option prevents the generated emails becoming too large. Default is 1000.
–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**.
–journal_keep_days=*n*	Remove HVR journal files in directory **$HVR_CONFIG/jnl** after *n* days. These files are written by integrate jobs if parameter **DbIntegrate /Journal** is defined.
–latency_limit=*dur*	Check for replication latencies and consider jobs over the limit erroneous.
–latency_channel=*chn*	Only check latencies of jobs in specified channel(s).
-latency_location=*loc*	Only check latencies of jobs in specified location(s).
–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**.
–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**.
–quiesce_grace=*secs*	If jobs are still running when the HVR Scheduler must stop, allow *secs* seconds grace before killing them. The default is 60 seconds. This parameter is passed the HVR Scheduler using the **–q** option.
–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).
–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_ignore=*patt*	Ignore log records which match specified pattern (can be regular expression).
–scan_channel=*chn*	Only scan for general errors and errors in specified channel(s).
–scan_location=*loc*	Only scan for general errors and errors in specified locations(s)
-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 '1' or '2c' (default).
-snmp_heartbeat	Send a hvrMaintNotifySummary notification, even if there was nothing to report.
-snmp_hostname=*host*	SNMP agent hostname. Defaults to localhost.
-snmp_port=*port*	SNMP agent trap port. Defaults to port 162.
-snmp_community=*str*	Community string for SNMPv1/v2c transactions.
–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.
–start	Starts HVR Scheduler.
–stop	Stops HVR Scheduler.
–task_name=*task*	Task name is used for writing different offset files.

–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.

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.

Notes

Command **hvrmaint** cannot process log files containing more than 12 months of data.

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
-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_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
-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_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

su ingres -c /opt/ingres/utility/ingstop    # Stop DBMS
backup -f/dev/rmt/0m                # Perform system backup

su ingres -c /opt/ingres/utility/ingstart   # Restart DBMS

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
-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_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
-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_errors
-snmp_server=snmp.mycorp.com
```

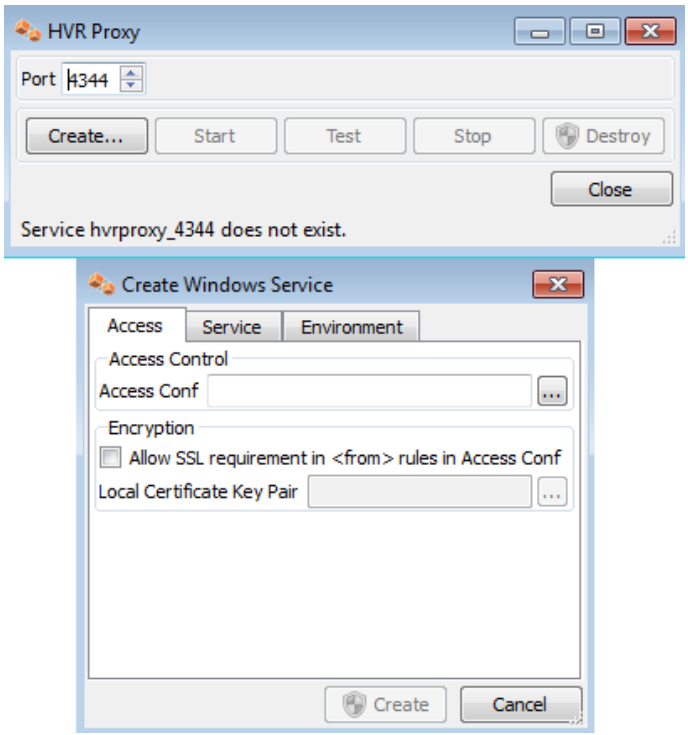
```
-output=c:\\opt\\hvr_config\\files\\hvrmaint.log
-scan_hvr_out
-stop
-archive_files=hvr.out              # Only archive log file hvr.out
-archive_keep_days=14               # Delete files after 2 weeks
-journal_keep_days=4
-start
```

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

2013-02-09 21:00:01 hvrmaint: Starting hvrmaint c:\tools\hvrmaint.opt -hub=hvr/ -stop -start
2013-02-09 21:10:21 hvrmaint: Stopping HVR Scheduler 4.4.4/5 (windows-x64-64bit).
2013-02-09 21:10:33 hvrmaint: Scanning d:\hvr_config\log\hvr\hvr.out (2013-02-08 21:00:03).
2013-02-09 21:11:13 hvrmaint: 7 errors (323 rows in fail tables) were detected during scan.
2013-02-09 21:12:33 hvrmaint: 3 capture jobs for 1 location did 606 cycles.
2013-02-09 21:12:59 hvrmaint: 6 integrate jobs for 2 locations did 400 cycles and integrated
    50 changes for 3 tables.
2013-02-09 21:13:53 hvrmaint: Archiving 9 log files to d:\hvr\archivelog\hvr_20050209.
2013-02-09 21:16:23 hvrmaint: Purging 0 archive directories older than 14 days.
2013-02-09 21:18:29 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 2013-02-09 19:43:52 and 2013-02-09 20:14:24
F_JJ106E_TIMEO_DB occurred 1 time at 2013-02-09 21:10:03

------------------------- Errors detected during scan--------------------------
2013-02-09 21:10:03: hvrscheduler: F_JJ106E_TIMEO_DB: Wait for response from database
 slave exceeded the 600 seconds defined by 'timeo_db' attribute. Waiting...
2013-02-09 19:43:52: 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 -----------------------
2013-02-09 21:19:01 hvrmaint: Sending e-mail to bob@mycorp.com; jim@mycorp.com
```

5.12 Hvrproxy

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 (**SeTcpPrivilege**). 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 an access configuration file 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**.

Options

–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). Operations **–as** and **–ah** can also be performed from the **Manage ▶ Services and Applications ▶ Services** window of Windows.

–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 *clsgrp* 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*n*=*v*
Set environment variable *n* to value *v* 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
```

Notes

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 log. See screen **Programs ▶ AdminitrativeTools ▶ EventViewer ▶ Log ▶ Application**.

Files

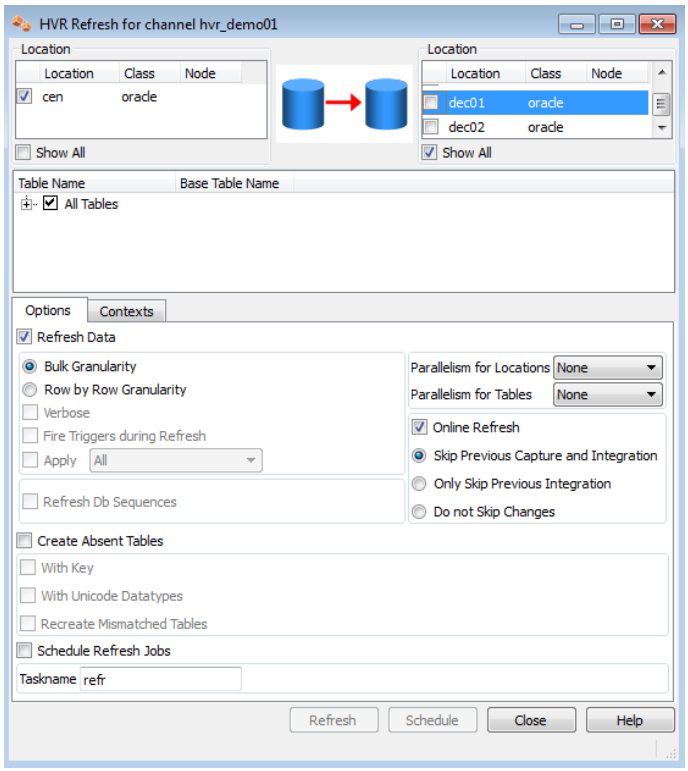
HVR_HOME	
├─ **bin**	
│ ├─ **hvr**	Executable for remote HVR service.
│ └─ **hvrproxy**	HVR Proxy executable.
└─ **lib**	
└─ **hvrproxy_example.xml**	Sample proxy access file.
HVR_CONFIG	
├─ **files**	
│ └─ **hvrproxy*port*.pid**	Process–id of daemon started with option **–d**.
└─ **log**	
└─ **hvrproxy**	
└─ **hvrproxy*port*.log**	Logfile for daemon started with **–d**.

See Also

Command hvr runtime engine.

5.13 Hvrrefresh

Name



hvrrefresh – Refresh the contents of tables in the channel.

Synopsis

hvrrefresh [*–options*] *hubdb chn*

Description

Command **hvrrefresh** performs the refresh of tables in a channel *chn* 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 first argument *hubdb* specifies the connection to the hub database; this can be an Oracle, Ingres, SQL Server or DB2 database depending on its form. See further section Calling HVR on the Command Line.

For database targets, the refresh can be performed row–by–row or as a bulk operation, depending on which **–g** option is supplied. 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. Row–wise refresh causes data to be piped from the read location to the write location whereby each individual row is compared. This results in a list of a minimal number of inserts, updates or deletes needed to resynchronize the tables. These SQL statements are then applied to the target database to affect the refresh.

An HVR channel can be defined purely for HVR Refresh, instead of being used for replication (capture and integrate jobs). In this case the channel must still be defined with actions **DbCapture** and **DbIntegrate**, even though HVR Load will never be called.

Options

–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 datatypes without any extra indexes, constraints, triggers or tables spaces. Value *S* can be one of the following:

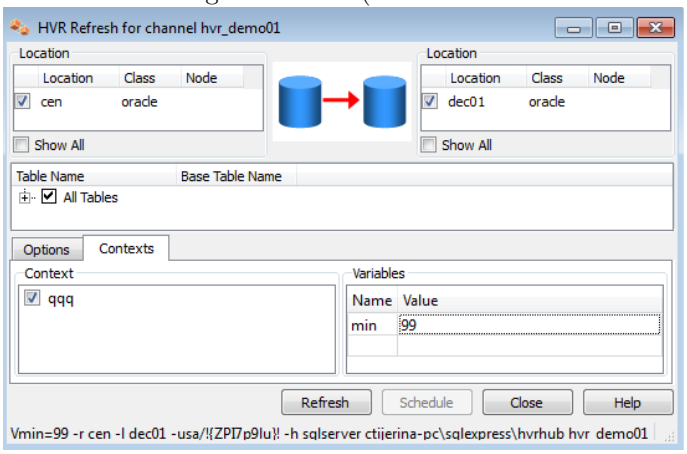
b	Create basic tables only (mandatory).
k	Create unique key or non–unique index. Default is no index. If the original table does not have a unique key, then a non–unique index is created not a unique key.
o	Only create tables, do refresh any data into them.
r	Recreate (drop and create) if the column names do not match.
u	Use Unicode datatypes such as **nvarchar** instead of **varchar**.

Several –c*S* 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. Parameter **DbObjectGeneration /RefreshCreateTableClause** can be used to add extra SQL to the **Create Table** statement which HVR will generate.

–C*ctx*

Enable context *ctx*. 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 /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. The default behavior is that database trigger/rule firing is disabled during refresh. Capture of the changes from **hvrrefresh** is also affected; normally HVR capture jobs will always ignore such changes whereas if option **–f** is supplied then they will only be ignored if the session name differs. The session name defaults to **hvr_integrate** and can be set on the capture side with action **DbCapture /IgnoreSession** and on the integrate side with **DbIntegrate /SessionName** (see parameter **/SessionName** in DbIntegrate). For Ingres databases this is avoided using statement **set no rules**. For Oracle and SQL Server databases this is avoided by disabling and re–enabling the triggers.

–g*x*

Granularity of refresh in database locations. Valid values of *x* are:

b	Bulk refresh using bulk data load. This is the default.
r	Row–wise refresh of tables.

–h*class*

Specify hub database. Valid values are **oracle**,**ingres, sqlserver** and **db2**. See also section Calling HVR on the Command Line.

–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 compared. Valid values of *mask* can be:

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 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**).

–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.

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.

–q*d*

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***ctx* 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** and 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 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 data will be read from location *loc* and written to the other location(s).

–s

Schedule invocation of refresh scripts using the HVR Scheduler. Without this option the default behavior is to perform the refresh immediately. The jobs created by this option are named *chn*–**refr**–*l1*–*l2*. These jobs are initially created in **SUSPEND** state. They can be invoked using command **hvrtrigger** as in the following example:

```
$ hvrtrigger -u -w hubdb chn-refr
```

The previous command unsuspends the jobs and instructs the scheduler to run them. Output from the jobs is copied to the **hvrtrigger** 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 on the command line (without HVR Scheduler) as follows:

```
$ hvrtrigger -i hubdb chn-refr-loc1-loc2
```

–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 must also be supplied.
–v	Verbose. This causes row–wise compare and 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*.

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
```

Notes

The effects of **hvrrefresh** can be customized by defining different actions in the channel. Possible actions include **DbIntegrate /DbProcDuringRefresh** (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 datatype 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.

Files

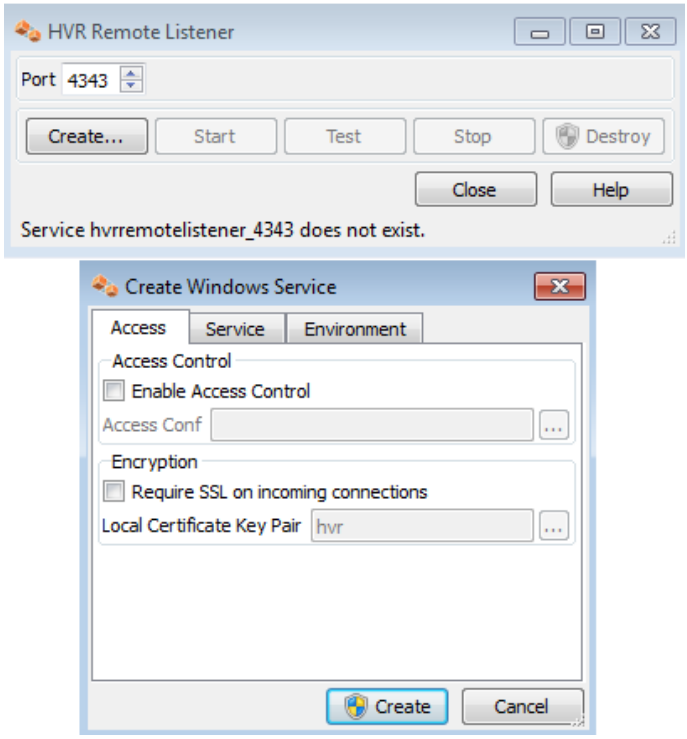
HVR_CONFIG	
└— **job**	Directory containing all scripts generated by **hvrrefresh**.
└— *hubdb*	
└— *chn*	
└— *chn*–**refr**–*loc1*–*loc2*	Script to refresh *loc1* with *loc2*.

See Also

Commands hvrcompare, hvrgui and hvrcrypt.

5.14 Hvrremotelistener

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 daemon **inetd**.
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 (**SeTcpPrivilege**). 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 an access configuration file 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:

```
<hvraccess>
  <allow>
     <from> <host>myhub</host> <ssl remote_cert="hub"/> </from>
  </allow>
</hvraccess>
```

Options

–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** (create and start) or **–ahd** (halt and destroy). Operations **–as** and **–ah** can also be performed from the **Manage ▶ Services and Applications ▶ Services** window of Windows.

–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 **inetd** service should be configured to start the HVR slave 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 *clsgrp* 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 **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

Stop **hvrremotelistener** daemon using the process–id 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 slaves 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**.

–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 OS and must be re–entered if passwords change.

–U*user*	Limits the HVR slave so it only accepts connections which are able to supply the password for account *user*. Multiple **–U** options can be supplied.

Notes

HVR Remote Listener is supported on Unix and Linux but it is more common on these machines to start remote HVR executables using the **inetd** process.
When running as a Windows service errors are written to the Windows event log. See screen **Programs ▶ AdminitrativeTools ▶ EventViewer ▶ Log ▶ Application**.

Custom HVR Password Validation

When **hvrremotelistener** 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**. This prevents an attempt to change the user. Also change **hvrremotelistener** configuration so that this service runs as an unprivileged 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**	Executable for remote HVR service.
│ └─ **hvrremotelistener**	HVR Remote Listener executable.
└─ **lib**	
├─ **hvrpasswd**	Password file employed by **hvrvalidpw_example**.
├─ **hvrvalidpw**	Optional script for password validation. If HVR detects this script, it invokes it for password validation instead of attempting this validation internally.
└─ **hvrvalidpw_example**	Sample custom password validation.
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**.

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
```

See Also

Command hvr runtime engine.

5.15 Hvrretryfailed

Name

hvrretryfail – 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 or DB2 database depending on its form. See further section Calling HVR on the Command Line.

Options

–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. Default is commit every 100 deletes. This is only allowed for database locations.
–h*class*	Specify hub database. Valid values are **oracle**,**ingres, sqlserver** and **db2**. See also section 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.

5.16 Hvrrouterview

Name

hvrrouterview – View or extract contents from internal router files.

Synopsis

hvrrouterview [*–restrict opts*] [*–xml opts*] [**–F**] *hubdb chn [txfile]...*
hvrrouterview –x*tgt* [*–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 (shown 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.pref**).
The first argument *hubdb* specifies the connection to the hub database. This can be an Oracle, Ingres, SQL Server or DB2 database depending on its form. See further section Calling HVR on the Command Line.

General Options

–F	Needed by hvrrouterview to recognize transaction file(s) created by a 'blob file' channel, which has no table information.
–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 must also be supplied.

Restrict Options

–b*time*	Only changes after capture time. Valid formats are *YYYY–MM–DD [HH:MM:SS]* in local time 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 **today** or **now**[[*+*|*–*]*SECS*] or an integer (seconds since **1970–01–01 00:00:00 UTC**). For example, **–e "2010–07–29 10:30:21"** or **–e now–3600** (changes more than one hour ago). 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/***hubdb*/*chn*. These files are kept if parameter **/Journal** is defined for **DbIntegrate** or **FileIntegrate**.
–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 colname op val. 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 **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

–d	Show column datatype information.
–h	Print data values in hexadecimal format.

Extract Options

–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**.

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**.

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**.

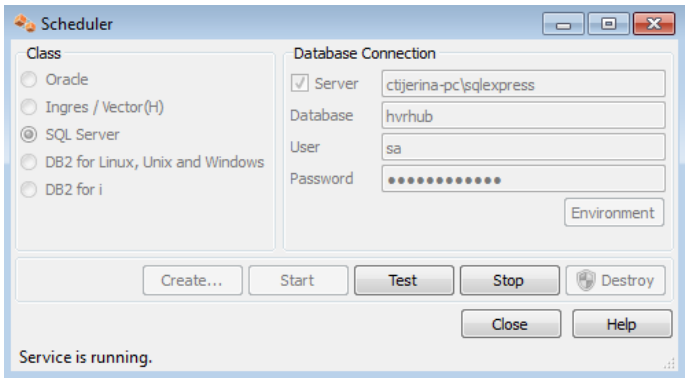
```
$ hvrrouterview -F -j -bnow-3600 -x/tmp hubdb chn
```

See Also

Command hvrload.

5.17 Hvrscheduler

Name



hvrscheduler – HVR Scheduler server

Synopsis

hvrscheduler [*–options*] *hubdb*

Description

The HVR Scheduler is a process which runs jobs defined in catalog **hvr_job**. This catalog is a table found in the hub database.
These jobs are generated by commands **hvrload**, **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 or DB2 database depending on its form. See further section Calling HVR on the Command Line.
On Unix the HVR Scheduler runs as a daemon, and on Microsoft Windows it runs as a system service. It can be started and stopped within the HVR GUI. Alternatively, on the Unix command line, it is started with command **hvrscheduler** *hubdb* (no options) and stopped using **hvrscheduler –k**. On the Windows command line the Scheduler system service is created with **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

–a*x*

Windows

Administration operations for the HVR Scheduler Microsoft Windows system service. Allowed values of *x* are:

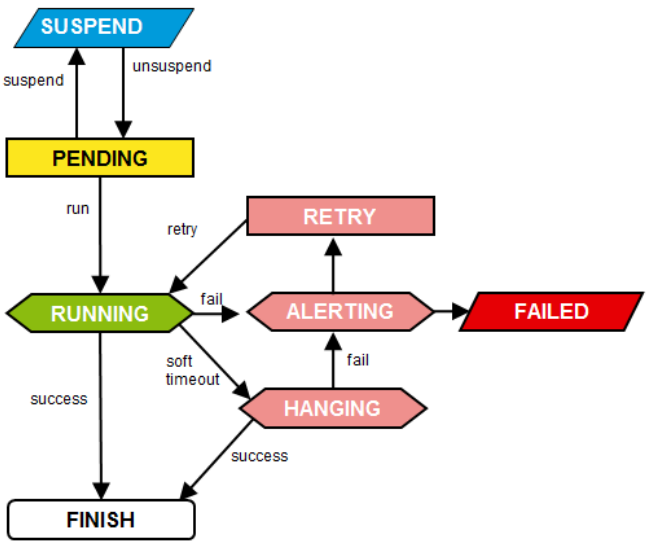
c	Create the HVR Scheduler service and configured it to start automatically at system reboot. The service will run under the default system unless **–P** option is given.
s	Start the HVR Scheduler service.
h	Halt (stop) the system service.
d	Destroy the system service.

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.
–h*class*	Specify hub database. Valid values are **oracle**,**ingres, sqlserver** and **db2**. 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	Kill old HVR Scheduler and any jobs which it may be running at that moment.
–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.
–s*lbl*	Internal slave co–process. 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	Timeout for connection to 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 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.

Environment Variables

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 '\', 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.

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.

Examples

Start a scheduler as a Unix daemon.

```
$ hvrscheduler hubdb
```

Create and start a Scheduler Windows Service.

```
c:\> hvrscheduler -acs hubdb
```

Notes

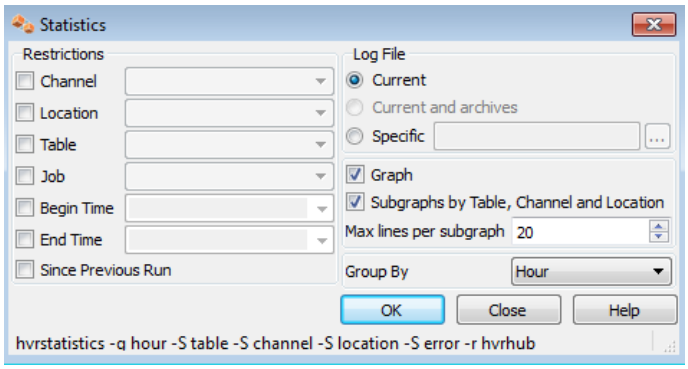
When starting the HVR Scheduler it is important that a database password is not exposed to other users. This can be encrypted using command **hvrcrypt**.

See Also

hvrcrypt, hvrsuspend, hvrtrigger.

5.18 Hvrstatistics

Name



hvrstatistics – Extract statistics from HVR scheduler logfiles

Synopsis

hvrstatistics [*–options*]... [*hubdb*]

Description

The first argument *hubdb* specifies the connection to the hub database. This can be an Oracle, Ingres, SQL Server or DB2 database depending on its form. See further section Calling HVR on the Command Line.

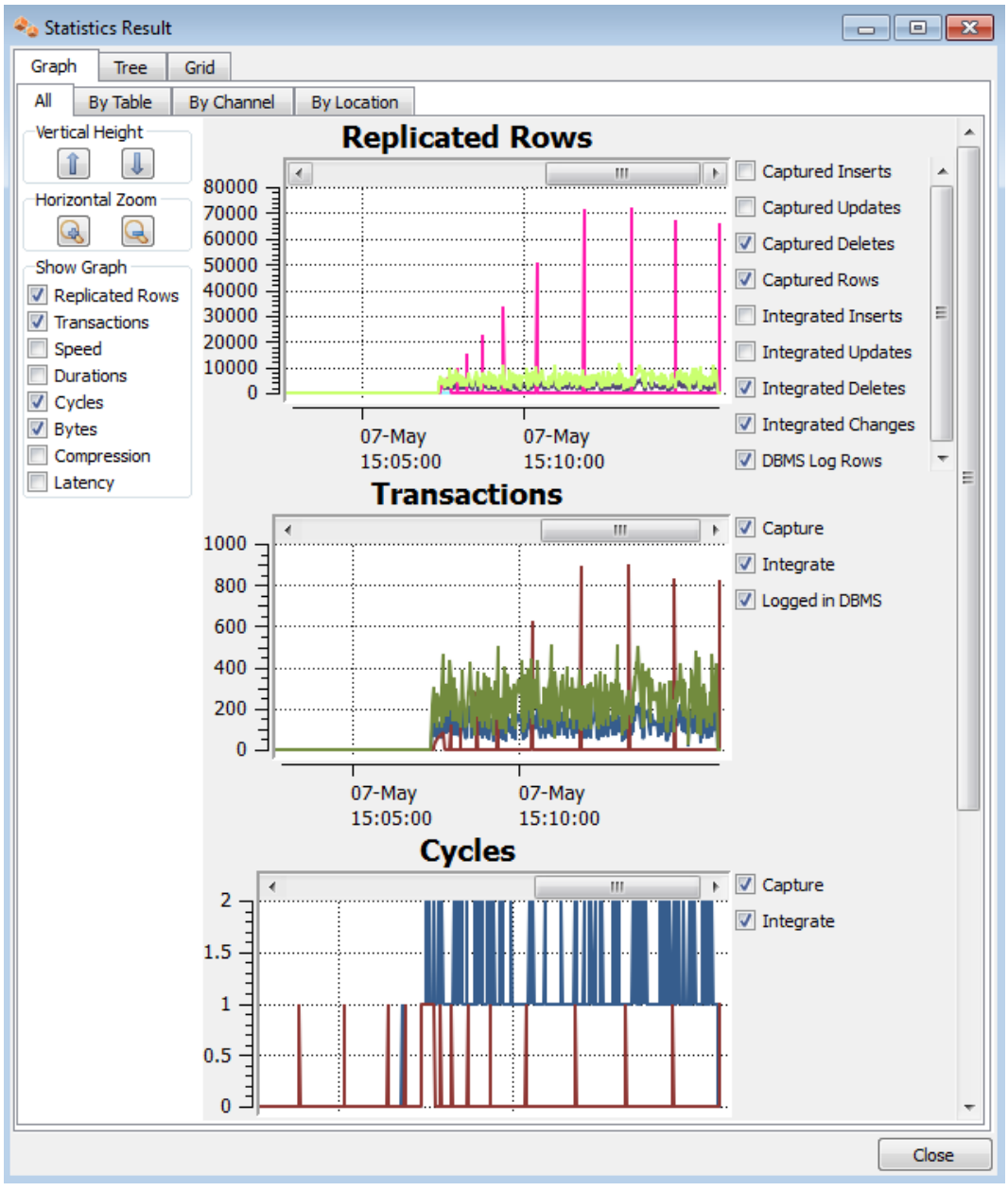
Options

–b*time*	Only lines after *time*. Argument must have format *YYYY–MM–DD HH:MI:SS*.
–c*chn*	Only parse output for specific channel. Alternatively, a specific channel can be omitted using form **–c!***chn*. This option can be specified multiple times.
–e*time*	Only lines until *time*. Argument must have format *YYYY–MM–DD HH:MI:SS*.
–f*file*	Parse scheduler log file *file*, instead of default **$HVR_CONFIG/log/***hubdb***/hvr.out**.
–g*col*	Summarize totals grouped by *col* which can be either **channel**, **location**, **job**, **table**, **year**, **month**, **day**, **hour**, **minute** or **second**. This option can be specified multiple times, which will subdivide the results by each column. Additionally, you can specify a reasonable subdivision of time-based columns, e.g.: `-s "10 minutes"` or `-s "6 hours"`
–i	Incremental. Only lines added since previous run of **hvrstatistics**. The position of the last run is stored in file **$HVR_CONFIG/files/hvrstatistics.offset**.
–I*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. 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.

	Summarize totals grouped by *col* which can be either **channel**, **location**, **job**, **table**, **year**, **month**, **day**, **hour**, **minute** or **second**.
–S *col*	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.

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
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
```

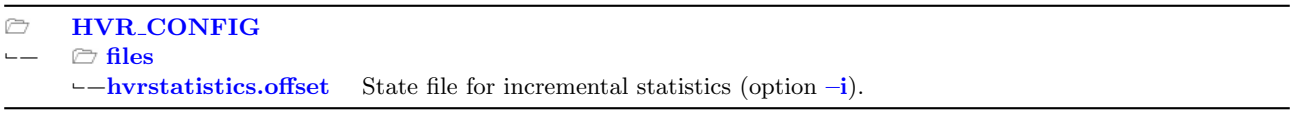
To create a CSV file with the same data use option –s as follows:

```
$ 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 transac-tions	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



Notes

Command **hvrstatistics** cannot process files containing more than 12 months of data.

5.19 Hvrsuspend

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. This can be an Oracle, Ingres, SQL Server or DB2 database depending on its form. See further section Calling HVR on the Command Line.
This command connects to the HVR Scheduler so the scheduler must be already running.

Options

–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
$ hvrload -oj -ld01 hubdb chn           # Regenerate script for location d01
$ hvrsuspend -u hubdb chn               # Reactivate replication
```

5.20 Hvrtestlistener, hvrtestlocation, hvrtestscheduler

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

–**C***pubcert*	Remote public certificate for testing encrypted SSL connection.
–**h***class*	Specify hub database. Valid values are **oracle**,**ingres, sqlserver** and **db2**. See also section Using HVR for Replication Transformations.
–**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.

5.21 Hvrtrigger

Name

hvrtrigger – Trigger jobs.

Synopsis

hvrtrigger [*–options*] *hubdb jobs...*

Description

Command **hvrtrigger** causes HVR jobs to be run. The jobs are either run via the HVR Scheduler or (if option **–i** is used) they are run directly by the **hvrtrigger** 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 name is specified, then **hvrtrigger** 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. This can be an Oracle, Ingres, SQL Server or DB2 database depending on its form. See further section Calling HVR on the Command Line.

Options

–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.
–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 **hvrtrigger** will wait indefinitely. This option cannot be used with option **–i**.
–u	Unsuspend. This option cannot be used with option **–t***N* or **–i**.
–w	Wait until all triggered jobs have finished running or have completed a full replication cycle. While **hvrtrigger** is waiting for a job, its output is carbon–copied to the **hvrtrigger** command's **stdout** and **stderr**. This option cannot be used with option **–i**.

Example

Run a capture job from the command line, without the HVR Scheduler;

```
$ hvrtrigger -i hubdb chn-cap-loc
```

Run all integrate jobs via the HVR Scheduler;

```
$ hvrtrigger hubdb chn-integ
```

Run all capture jobs and then all integrate jobs from the command line;

```
$ hvrtrigger -i hubdb chn
```

Run a refresh job from the command line;

```
$ hvrtrigger -i hubdb chn-refr-loc1-loc2
```

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.

Files

HVR_CONFIG		
└─ **log**		
└─ *hubdb*		
├─ **hvr.ctrl**	Audit log containing **hvrtrigger** actions.	
└─ **hvr.out**	Log of job output and errors.	

6

Action Reference

Action	Parameter	Value	Description
DbCapture	**/LogBased**		Capture changes directly from DBMS logging system
	/ClusterThread	*threadnum*	Only capture from specific logging thread of cluster
	/LogJournal	*sch.jnl*	Specify DB2-for-i journal
	/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
	/NoBeforeUpdate		Only capture the new values for updated rows
	/NoTruncate		Do not capture truncate table statements
	/IgnoreSessionName	*sess_name*	Ignore changes applied by session *sess_name*
	/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
	/Coalesce		Coalesce consecutive changes on the same row into a single change
	/AugmentIncomplete	*col_type*	Capture job must select for column values. Can be *NONE*, *LOB* or *ALL*.
	/SupplementalLogsPK	*action*	Mechanism to capture updates to Sql Server tables with primary key.
	/SupplementalLogsNoPK	*action*	Mechanism to capture updates to Sql Server tables without primary key.
DbIntegrate	**/DbProc**		Apply changes by calling integrate database procedures
	/OnErrorSaveFailed		Write failed row to fail table
	/OnErrorBlockLocation		On integrate error block changes from capture location
	/TxBundleSize	*int*	Bundle small transactions for improved performance
	/TxSplitLimit	*int*	Split very large transactions to limit resource usage
	/CycleByteLimit	*int*	Max amount of routed data (compressed) to process per integrate cycle
	/Burst		Resort changes, load into staging table and apply with set-wise SQL
	/BurstCommitFrequency	*freq*	Frequency of commits. Values *STATEMENT*, *TABLE* or *CYCLE*
	/Resilient		Resilient integrate for inserts, updates and deletes
	/ResilientInsert		Resilient inserts; convert duplicate inserts to update
	/ResilientUpdate		Resilient updates; convert lost updates to inserts
	/ResilientDelete		Resilient deletes; ignore lost deletes
	/ResilientDeleteCondition	*sql_expr*	Resilient delete only if *sql_expr* is true
	/ResilientWarning		Warning (not silent) if resilient conversion occurs
	/NoTriggerFiring		Enable/Disable triggering of database rules
	/SessionName	*sess_name*	Integrate changes with special session name *sess_name*
	/DbProcDuringRefresh		Apply changes during refresh using database procedures
	/Coalesce		Coalesce consecutive changes on the same row into a single change
	/Journal		Move processed transaction files to journal directory on hub
	/Delay	*N*	Delay integration of changes for *N* seconds
TableProperties	**/BaseName**	*tbl_name*	Name of table in database differs from name in catalogs
	/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
	/TrimWhiteSpace		Remove trailing whitespace from varchars
	/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
ColumnProperties	**/Name**	*col_name*	Name of column in **hvr_column** catalog
	/DatatypeMatch	*data_type*	Datatype used for matching instead of **/Name**

Action	Parameter	Value	Description
	/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
	/IntegrateExpression	*sql_expr*	SQL expression for column value when integrating
	/NoUpdate		Do not use default value during updates, only inserts
	/IgnoreDuringCompare		Ignore values in column during compare and refresh
	/Datatype	*data_type*	Datatype 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
	/Key		Add column to table's replication key
	/DistributionKey		Distribution key column
	/TimeKey		Convert all changes to inserts, using this column for time dimension
	/Context	*ctx*	Ignore action unless refresh/compare context *ctx* is enabled
Restrict	**/CaptureCondition**	*sql_expr*	Restrict during capture
	/IntegrateCondition	*sql_expr*	Restrict during integration
	/RefreshCondition	*sql_expr*	Restrict during refresh and 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
	/Context	*ctx*	Action only applies if Refresh/Compare context matches
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
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
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
FileCapture	**/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*
	/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
FileIntegrate	**/RenameExpression**	*expression*	Expression to name new files, containing brace substitutions
	/ErrorOnOverwrite		Error if a new file has same name as an existing file
	/MaxFileSize	*size*	Limit each XML file to *size* bytes
	/Journal		Move processed transaction files to journal directory on hub
	/Burst		Sort and coalesce changes to create a single file per table in a cycle
	/ConvertNewlinesTo	*OSTYPE*	Write files with *UNIX* or *DOS* style newlines
	/Verbose		Report name of each file integrated
	/OnErrorSaveFailed		If file move fails save contents, instead of giving fatal error
	/CycleByteLimit	*int*	Max amount of routed data (compressed) to process per integrate cycle
	/Context	*ctx*	Action only applies if Refresh/Compare context matches
Transform	**Row2Csv**		Write file as Comma Separated Value, instead of XML
	/Command	*path*	Path to script or executable performing custom transformation
	/XsltStylesheet	*path*	Stylesheet for performing XSLT transformation
	SoftDelete	*colname*	Convert deletes into updates of a boolean soft–delete column

Action	Parameter	Value	Description
	Xml2Csv		Convert XML to Comma Separated Value.
	Csv2Xml		Convert Comma Separated Value to XML.
	File2Column		Load the contents of a file into a column with LOB datatype
	Tokenize		Encrypt columns using a tokenization server
	SapAugment		Capture job selecting for de-clustering of multi-row SAP cluster tables
	SapXForm		Invoke SAP transformation for SAP pool and cluster tables
	/Builtin	*transform*	Select built-in transform
	/Order	*int*	Specify order of transformations
	/ArgumentName	*names*	Name(s) of parameter()s for XSLT stylesheet (space separated)
	/ArgumentValue	*vals*	Value(s) of parameter(s) for transform (space separated)
	/FileTags		Transform <file> and <bytes> tags instead of file contents
	/ExecOnHub		"Execute transform on hub instead of location's machine
	/Context	*context*	Action only applies if Refresh/Compare *context* matches
SalesforceCapture	**/BulkAPI**		Use Salesforce Bulk API (instead of the SOAP interface)
	/SerialMode		Force serial instead of parallel processing for Bulk API
SalesforceIntegrate	**/BulkAPI**		Use Salesforce Bulk API (instead of the SOAP interface)
	/SerialMode		Force serial mode instead of parallel processing for Bulk API
	/OnErrorSaveFailed		Write failed rows/files to fail directory
	/TableName	*apitab*	API name of table to upload attachments into
	/KeyName	*apikey*	API name of attachment table's key column
	/Journal		Move processed transaction files to journal directory on hub
Agent	**/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
Environment	**/Name**	*name*	Name of environment variable
	/Value	*value*	Value of environment variable
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
	/StateDirectory	*path*	Directory for file location state files. Defaults to <top>/_hvr_state
	/CaseSensitiveNames		DBMS table and columns names are treated case sensitive by HVR
	/Proxy	*proxy*	Proxy server URL for FTP, SFTP, WebDAV or Salesforce locations
	/Order	*N*	Specify order of hub->loc proxy chain
	/StagingDirectoryHvr	*URL*	Directory for bulk load staging files
	/StagingDirectoryDb	*URL*	Location for the bulk load staging files visible from the Database
	/StagingDirectoryCredentials	*redentials*	Credentials to be used for S3 authentication during RedShift bulk load
Scheduling	**/CaptureStartTimes**	*times*	Trigger capture job at specific times, rather than continuous cycling
	/CaptureOnceOnTrigger		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
	/IntegrateOnceOnTrigger		Integrate job runs for one cycle after trigger
	/RefreshStartTimes	*times*	Trigger refresh job at specific times
	/CompareStartTimes	*crono*	Trigger compare job at specific times

6.1 DbCapture

Description

Action **DbCapture** instructs HVR to capture changes to a database table. Various parameters are available to modify the functionality and performance of capture.

Parameters

Parameter	Argument	Description
/LogBased		Capture changes directly from the DBMS logging system. If this parameter is not defined then database triggers will be used for capture. Log based gives improved performance and low latency but limits which other **DbCapture** parameters can be used. Log based capture is required for table **truncate** statements to be captured such as **modify to truncated**. Otherwise these statements are not replicated. Configuring log based capture requires that journaling is enabled on Ingres tables and archiving is enabled for Oracle tables. If this parameter is defined for any table, then it affects all tables captured from that location.
/ClusterThread	*threadnum*	Only capture from specific logging thread of Oracle RAC cluster. By default a single capture job will be defined which pulls changes from all threads in an Oracle cluster. An alternative is to create multiple HVR locations for the same Oracle database and to define each with a separate **DbCapture /ClusterThread** parameter. This means the changes will no longer be captured in the logical order, but can increase parallelism.
/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.
/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. 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*	This parameter instructs HVR trigger based capture jobs to wait for a fixed interval before toggling and reselecting capture tables, instead of dynamically waiting 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. If this parameter is defined for any table, then it affects all tables captured from that location.

Parameter	Argument	Description
/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 datatypes. 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.
/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.
/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 **DbIntegrate /SessionName**. For more information, see section **SESSION NAMES AND RECAPTURING** below. If this parameter is defined for any table with log based capture, then it affects all tables captured from that location.
/IgnoreCondition	*sql_expr*	Ignore (do not capture) any changes that satisfy expression *sql_expr*. 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.
/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.

Parameter	Argument	Description
/HashBuckets Ingres	*int*	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.
/HashKey Ingres	*col_list*	Specify different key for capture table hashing. 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.
/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.
/AugmentIncomplete	*col_type*	During capture, HVR may receive partial values for certain column types. The **/AugmentIncomplete** option tells HVR to perform additional steps to retrieve the full value. Valid values for col_type are *NONE*, *LOB* or *ALL*. Default is *LOB* for SqlServer and *NONE* for others
/SupplementalLogsPK SQL Server	*action*	Specify what action should be performed to enable supplemental logging for tables with a primary key. Supplemental logging should be enabled to make log-based capture of updates possible. If supplemental logging is not enabled then log-based capture can only process inserts and deletes. Valid values • **ARTICLE**: Enable supplemental logging of updates by creating an SQL Server transactional replication article. This is fairly efficient, but it is only allowed if source table has a primary key. The disadvantage is that if users attempt some DDL such as DROP TABLE or TRUNCATE TABLE they will give an error message. This is the default setting for tables with a primary key. • **CDC_TAB**: Enable supplemental logging of updates by creating a Change Data Capture (CDC) instance for the source table. It allows users to perform DDL statements such as DROP TABLE without getting an error (but the HVR capture job may fail), but TRUNCATE TABLE will still give the user an error message. Creating a CDC instance causes I/O overhead (SQL Server jobs copy each change to a CDC table, which no-one uses). • **IGNORE_UPDATE**: Do not enable supplemental logging. Capture of update statements is not supported, so HVR will only capture inserts and deletes to such tables. All DDL can be performed; DROP TABLE and TRUNCATE TABLE are all allowed. The **ARTICLE** action is the default.

Parameter	Argument	Description
/SupplementalLogsNoPK SQL Server	*action*	Specify what action should be performed to enable supplemental logging for tables without primary key. Supplemental logging should be enabled to make log-based capture of updates possible. If supplemental logging is not enabled then log-based capture can only process inserts and deletes. Valid values • **CDC_TAB**: Enable supplemental logging of updates by creating a Change Data Capture (CDC) instance for the source table. The disadvantage is that if users attempt TRUNCATE TABLE it will give them an error message. Another disadvantage is that creating a CDC instance causes I/O overhead (SQL Server jobs copy each change to a CDC table, which no-one uses). • **IGNORE_UPDATE**: Do not enable supplemental logging. Capture of update statements is not supported, so HVR will only capture inserts and deletes to such tables. • **WARN**: This is similar to **IGNORE_UPDATE**, but also instructs the **HVR Load** command to produces a warning message if the channel includes a table without primary key. The default behavior is to create a CDC instance (similar to **CDC_TAB** option), but also produce a warning message telling the user about the drawbacks.

Session Names and Recapturing

Replication recapturing is when changes made by integration are captured again. Recapturing is controlled using session names. Depending on the situation recapturing can be useful or unwanted. The following are some examples:

Bi directional replication.

During bidirectional replication if integrated changes are captured again then they can boomerang back to the original capture database. This would form an infinite loop. For this reason capture triggers check the session name and avoid recapturing integration.

Cascade replication.

Cascade replication is when changes from one channel are captured again by a different channel and replicated onto a different group of databases. Recapturing is necessary for cascade replication. Recapturing can be configured using action **Integrate /SessionName** so that integration is not recognized by the cascade channel's capture triggers.

Batch work to purge old data.

Sometimes replication of large blocks of batch work is too expensive. It may be more efficient to repeat the same batch work on each replicated database. Capturing of the batch work can be disabled by setting the session name so that the capture triggers see the changes as belonging to an integrate session. 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.

Sometimes an application will have database triggers on the replicated tables. These triggers will have already been fired on the capture database so firing them again during HVR integration is unnecessary and can cause consistency problems. For Ingres databases this rule firing can be avoided with action **DbIntegrate /NoTriggerFiring**, but for other DBMS's the application triggers can be modified so that they have no affect during replication 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.

Oracle

- Oracle client information.

For Oracle, the integrate job always sets the session name by calling **set_client_info()**. This is recognized by trigger based capture but not by log based capture jobs (although log based channels still set this session name during integrate). End user sessions can therefore make changes without trigger based capture being activated by calling **dbms_application_info.set_client_info('hvr_integrate')**. An application database trigger can be prevented from firing on the integrate side by changing it to include the following clause: **where userenv('CLIENT_INFO') <> 'hvr_integrate'**.

Ingres

- Ingres role

For Ingres, the integrate job always connects to the target database using an Ingres role, which is the name of the session (e.g. **hvr_integrate**). This is recognized by trigger based capture but not by log based capture jobs (although log based channels still set this session name during integrate). End user sessions can therefore make changes without trigger based capture being activated by connecting to the database with **sql** option **–R**. An application database rule can be prevented from firing on the integrate side by changing it to include the following clause: **where dbmsinfo('role') != 'hvr_integrate'**.

SQL Server

- SQL Server application name

For SQL Server, the integrate job always connects to the target dataset using an application name which is the session name (e.g. **hvr_integrate**). This is recognized by trigger based capture but not by log based capture jobs (although log based channels still set this session name during 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 integrate side by changing it to include the following clause: **where app_name() <> 'hvr_integrate'**.

- Current user name

All capture jobs (both trigger based and log based) also recognize the current DBMS user name as the session name. End user sessions can therefore make changes without trigger based or log based capture being activated 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 integrate side by changing it to include the following clause: **where user <> 'hvr_integrate'**.

- Integrate state table

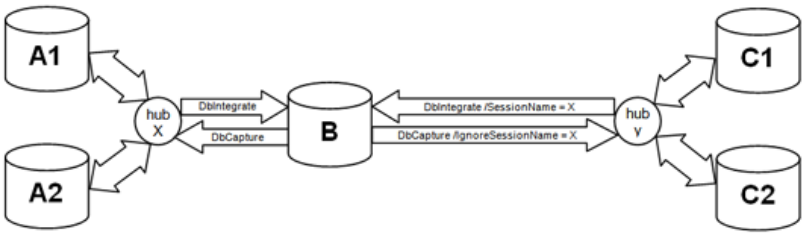
Each transaction performed by an integrate job also contains an update to an integrate state table (named **hvr_stin*** or **hvr_stis***) which also has a column **session_name** containing the session name. This allows log based capture jobs to recognize the session name defined by an integrate job, but this is ignored by trigger based capture jobs.

Example

Changes made to databases **A1** and **A2** must replicate via **hub X** to **B** and then cascade via **hub Y** to **C1** and **C2**. Changes made to **B** must replicate to **A2** and **A2** 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. This is solved by adding parameters **/SessionName** and **/IgnoreSessionName** to the channel in **hub Y**.

Dependencies

Parameter **/LogBased** cannot be used with **/QuickToggle**, **/KeyOnlyCaptureTable**, **/HashBuckets**, **/IgnoreCondition** or **/IgnoreUpdateCondition**.
Parameter **/HashKey** requires that **/HashBuckets** is also defined.



Examples

For parameter **/LogBased** see channel **hvr_demo50**, **hvr_demo51** and **hvr_demo70** in **$HVR_HOME/demo**.
For parameters **/HashBuckets** and **/HashKey** see channel **hvr_demo16** in **$HVR_HOME/demo**.

6.2 DbIntegrate

Description

Action **DbIntegrate** instructs HVR to integrate changes into a database table. Various parameters are available to tune the integration functionality and performance.

Parameters

Parameter	Argument	Description
/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 **hvrload** and called *tbl*__**ii**, *tbl*__**iu**, *tbl*__**id**. This parameter cannot be used on tables with long column datatypes.
/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.
/OnErrorBlockLocation		On integration error, block further changes from the current capture location by writing them into a 'fail table' *tbl*__**f** but continue integrating changes from the other capture locations. Changes written into the fail table can be retried afterwards using command **hvrretryfailed**. Integration of changes from blocked capture locations can be restarted with command **hvrcontrol –D –n block_loc**. If this parameter is defined for any table, then it affects all tables integrated to that location.
/TxBundleSize	*int*	Bundle small transactions together for improved performance. 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. The default transaction bundle size is 100. If this parameter is defined for any table, then it affects all tables integrated to that location.
/TxSplitLimit	*int*	Split very large transactions to limit resource usage. 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. The default is 0, which means transactions are never split. If this parameter is defined for any table, then it affects all tables integrated to that location.
/CycleByteLimit	*int*	Maximum amount of routed data (compressed) to process per 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**). The default is **10 Mb**. Value 0 means unlimited, so the integrate job will process all available work in a single cycle. If the supplied value is smaller than the size of the first transaction file in the **router** directory, then all the transactions in that file will be processed. Transactions in a transaction file will never be split between cycles or subcycles.

Parameter	Argument	Description
/Burst		Integrate changes into 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 trickle 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.
/BurstCommitFrequency	*freq*	Frequency for committing burst set wise SQL statements. Valid values • **CYCLE**: this means all changes for the integrate job cycle are committed in a single transaction • **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. The default frequency for burst is **CYCLE**.
/Resilient		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. Parameter **/Resilient** is equivalent to defining **/ResilientInsert**, **/ResilientUpdate** and **/ResilientDelete** individually.
/ResilientInsert		Resilient integration of inserts. If a row already exists then the insert is converted to an update. Resilient insert is also implied by parameter **/Resilient**.
/ResilientUpdate		Resilient integration of updates. An update of a nonexistent row is converted to an insert. Resilient update is also implied by parameter **/Resilient**.
/ResilientDelete		Resilient integration of deletes. A delete of a nonexistent row is discarded. Resilient delete is also implied by parameter **/Resilient**.
/ResilientDeleteCondition	*sql_expr*	Resilient integration of deletes only if *sql_expr* is TRUE. The SQL expression can contain substitutions: • Pattern **{***col_name***}** is replaced with the actual value from that column of the row. The name used should be the column's 'HVR column name' and not the 'base name'. • **{hvr_cap_loc}** is replaced with the location where the capture was changed. • **{hvr_cap_tstamp}** is replaced with the moment that the change occurred. • **{hvr_cap_user}** is replaced with the name of the user which made the change. Conditional delete resilience can be used on a detail table if a replicated delete sometimes fails because of a previous delete to a master table, which caused a cascade delete. For this example add the following condition to the detail table; **not exists (select 1 from MASTER where key1={key1} and key2={key2})**
/ResilientWarning		Write warning message for each resilient conversion. Normally conversion (for example a lost update being converted to an insert) occurs silently.

Parameter	Argument	Description
/NoTriggerFiring Ingres		Integrate changes without firing database rules using Ingres SQL statement **set norules**. This can be used to prevent replication rules, which have already been fired on the capture database from firing again during integration. Another way to do the same is to change the replication rules to recognize the integrate session name (see parameter **/SessionName**). This statement is only allowed if integration connects the integrate database as the user which owns the table. During refresh database rules can be disabled using **hvrrefresh** with option **–f**. If this parameter is defined for any table, then it affects all tables integrated to that location.
/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 action **DbCapture**. If this parameter is defined for any table, then it affects all tables integrated to that location.
/DbProcDuringRefresh		Apply changes during refresh using database procedures. This allows row wise refresh to select from a view on its 'write' side before applying changes to a real table using a database procedure, which can be supplied using action **DbObjectGeneration**. Commands **hvrcompare** and **hvrrefresh** can also have views on their 'read' side instead of regular tables.
/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**.
/Journal		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	*secs*	Delay integration of changes for *N* seconds. For example if **/Delay=900** is defined, then 15 minutes after each change is captured it will be integrated into the target database.

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 **DbCapture /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 datatype 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')**. This is described in more detail in SESSIONS AND RECAPTURING in section DbCapture.
For Ingres target databases, database rule firing can be prevented by specifying **DbIntegrate /NoTriggerFiring** or with **hvrrefresh** option **–f**.

Dependencies

Parameters **/OnErrorSaveFailed** and **/OnErrorBlockedLocation** cannot be used together.
Parameters **/ResilientDelete** and **/ResilientDeleteCondition** cannot be used together.

Parameter **/ResilientWarning** requires that one of the other resilient parameters is defined.
Parameter **/DbProc** cannot be used on tables with long datatypes, for example **long varchar** or **blob**.

Example

For parameter **/Journal** see channel **hvr_demo20 in $HVR_HOME/demo**.

6.3 TableProperties

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

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).
/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 datatype or if a conversion is impossible because of a datatype 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 datatype. The default value used for date is **0001 01 01**.
/TrimWhiteSpace		Remove trailing whitespace from varchars.
/TrimTime	*policy*	Trim time when converting from Oracle and SqlServer date. Value *policy* can be **YES** (always trim time component off), **NO** (never trim) or **MIDNIGHT** (only trim if value has time **00:00:00**. Default is **NO**.
/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 must have form DD/MM/YYYY.

6.4 ColumnProperties

Description

Action **ColumnProperties** defines properties of a column. This column is matched either by specifying parameter **/Name** or using parameter **/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

Parameter	Argument	Description
/Name	*col_name*	Name of column in **hvr_column** catalog.
/DatatypeMatch	*data_type*	Datatype used for matching instead of **/Name**. Value of datatype in **hvr_column** catalog. This parameter can be used to associate a **ColumnProperties** action with all columns which match this datatype.
/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).
/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.).
/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.
/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**'. The following substitutions are allowed: • Pattern **{***col_name***}** is replaced with the actual value from that column of the row. The name used should be the column's 'base name' and not the 'HVR column name'. • **{hvr_cap_loc}** is replaced with the location name. • **{hvr_cap_tstamp}** is replaced with the moment that the change occurred. Note that the time is only in seconds, which does not give enough granularity to distinguish or sort individual changes. • **{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* to command **hvrrefresh** or **hvrcompare**. For many databases (e.g. Oracle and SQL Server) a subselect can be supplied, for example (**select descrip from lookup where id={id}**).

Parameter	Argument	Description
/IntegrateExpression	*sql_expr*	Expression for column value when integrating changes or loading data into a target table. HVR may evaluate itself of 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**. Possible expressions include **null**, **5** or '**hello**'. The following substitutions are allowed: • Pattern **{***col_name***}** is replaced with the actual value from that column of the row. The name used should be the column's 'HVR column name', and not the table's 'base name'. • **{hvr_cap_loc}** is replaced with the name of the location where the change occurred. • **{hvr_cap_tstamp}** is replaced with the moment that the change occurred. • **{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* to command **hvrrefresh** or **hvrcompare**. • **{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_integ_tstamp}** is replaced with the time that the change is integrated. Note that the time is often only in seconds, which does not give enough granularity to distinguish or sort individual changes. • **{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. This can be useful for **ColumnProperties /TimeKey**. • **{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 zero indicates that commit information is missing for that change. • **{{hvr_key_names** *sep***}}** is replaced with the values of table's key columns, concatenated together with separator *sep*. For many databases (e.g. Oracle and SQL Server) a subselect can be supplied, for example (**select descrip from lookup where id={id}**).
/NoUpdate		The value for this column should never be changed by HVR when integrating an **update statement**. This means for example that an expression supplied using **/IntegrateExpression** will only be evaluated during an insert statement. By default HVR integration will only change the column's value if the captured update also changed that value or parameters **/IgnoreDuringCompare** or **DbCapture /NoBeforeUpdate**, or **DbIntegrate /DbProc** are defined. Otherwise the column's value is preserved. This parameter cannot be used with **/Extra**.
/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.
/Datatype	*data_type*	Datatype in database if this differs from **hvr_column** catalog
/Length	*int*	String length in database if this differs from value defined in **hvr_column** catalog.
/Precision	*int*	Integer precision in database if this differs from value defined in **hvr_column** catalog.
/Scale	*int*	Integer scale in database if this differs from value defined in **hvr_column** catalog.
/Nullable		Nullability in database if this differs from value defined in **hvr_column** catalog.
/Identity SQL Server		Column has SQL Server identity attribute. Only effective when using integrate database procedures.
/Key		Add column to table's replication key.

Parameter	Argument	Description
/DistributionKey		Distribution key column. The distribution key is used for parallelizing changes within a table. It also controls the **distributed by** clause for a create table in distributed databases such as Teradata, Paraccel and Greenplum.
/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**, which will populate a value.
/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 **ColumnProperties /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 datatypes, or **0** for numeric datatypes, 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.

Dependencies

Parameters **/BaseName**, **/Extra** and **/Absent** cannot be used together.
Parameters **/Extra** and **/Absent** cannot be used on columns which are part of the replication key. They cannot both be defined in a given database on the same column, nor can either be combined on a column with parameter **/BaseName**.
Parameter **/Length**, **/Precision**, **/Scale**, **/Nullable** and **/Identity** can only be used if parameter **/Datatype** is defined.

Examples

For parameter **/IgnoreDuringCompare** see channel **hvr_demo07** in **$HVR_HOME/demo**.
See also channel **hvr_demo24** and **hvr_demo26** in **$HVR_HOME/demo**.

6.5 Restrict

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

Parameter	Argument	Description
/CaptureCondition	*sql_expr*	Only rows where this condition is TRUE are captured. The SQL expression on a database location can contain these substitutions: • Pattern **{***col_name***}** is replaced with the actual value from that column of the row. The name used should be the column's 'base name' and not the 'HVR column name'. • **{hvr_cap_loc}** is replaced with the location name. • **{hvr_cap_tstamp}** is replaced with the moment that the change occurred. • **{hvr_cap_user}** is replaced with the name of the user which 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 **DbCapture /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 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 this condition is TRUE are integrated. The SQL expression can contain substitutions: • Pattern **{***col_name***}** is replaced with the actual value from that column of the row. The name used should be the column's 'HVR column name' and not the 'base name'. • **{hvr_cap_loc}** is replaced with the location where the capture was changed. • **{hvr_cap_tstamp}** is replaced with the moment that the change occurred. • **{hvr_cap_user}** is replaced with the name of the user which 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.

Parameter	Argument	Description
/RefreshCondition	*sql_expr*	Only rows where this condition evaluates as TRUE are refreshed or compared. For compare this parameter controls which rows are selected for comparison (it can be defined on both databases or just on one). For refresh it can also be defined either 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 **truncate** *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 can contain substitutions: • Pattern **{***col_name***}** is replaced with the actual value from that column of the row. The name used should be the column's 'base name' and not the 'HVR column name'. • Pattern **{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**. • Pattern **{hvr_local_loc}** is replaced with the current location name. • Pattern **{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.
/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 **DbCapture**. 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 with **DbCapture /LogBased**. A possible alternative for log–based capture channels is **Restrict /AddressTo** and **Restrict /AddressSubscribe**.

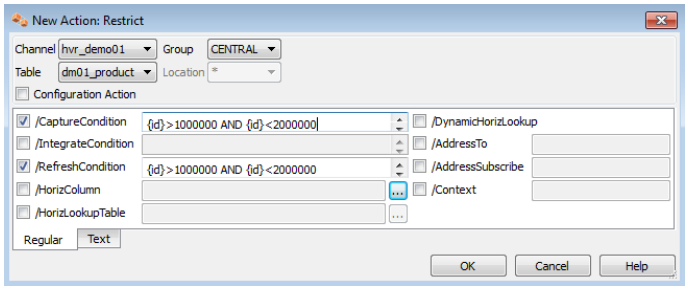
Parameter	Argument	Description
/DynamicHorizLookup		Dynamic replication of changes to lookup table. Normally only changes to the horizontally partitioned table are replicated. This parameter causes changes to the lookup table to also trigger capture. This is done by creating extra rules/triggers that 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 with **DbCapture /LogBased**. A possible alternative for log–based capture channels is **Restrict /AddressTo** and **Restrict /AddressSubscribe**.
/AddressTo	*addr*	Captured changes should only be sent to integrate locations that match integrate address *addr*. The 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** or **Alias7**. • A pattern to match one of the above, such as **dec***. • A column name enclosed in braces, such as **{mycol}**. The contents of this column will be used as an integrate address. This is similar to parameter **/HorizColumn**. • A list of the above, separated by a semicolon, colon or comma, such as **cen,{col3}**. This parameter should be defined with **DbCapture** or **FileCapture**. This parameter does not do 'update conversion'.
/AddressSubscribe	*addr*	This integrate location should be sent a copy of any changes that match integrate address *addr*. The address can be one of the following: • A different integrate location name, such as **dec01**. • A location group name containing other integrate locations, such as **DECEN**. • A pattern to match one of the above, such as **dec***. • An alias to match an integrate address defined with **/AddressTo** or **/HorizColumn** or matched by **{hvr_address}** in **FileCapture /Pattern**. An alias can contain numbers, letters and underscores, for example **22** or **Alias7**. • A list of the above, separated by a semicolon, colon or comma, such as **dec*,CEN**. This parameter should be defined with **DbIntegrate** or **FileIntegrate**.
/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 **Restrict /RefreshCondition="{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.

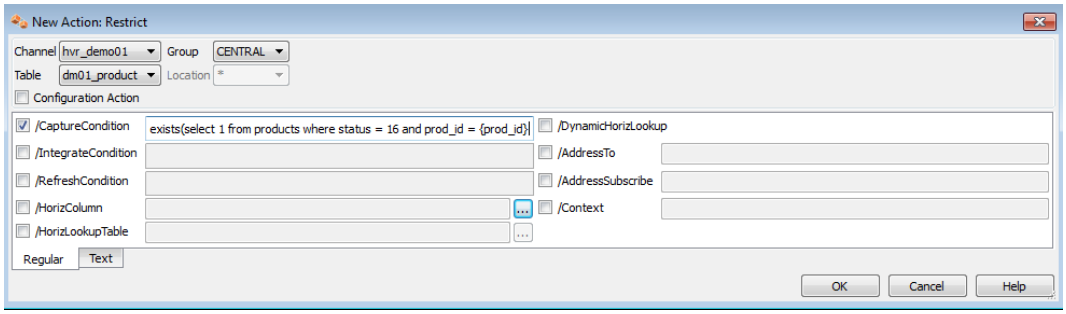
Horizontal Partitioning

Horizontal partitioning means that different parts of a table should be replicated into different directions. Logic is added inside capture to calculate the destination address for each change, based on the row's column values. The destination is put in a special column of the capture table named **hvr_address**. Normally during routing each capture change is sent to all other locations which have a **DbIntegrate** action defined for that row, but this **hvr_address** column overrides this. The change is sent instead to only the destinations specified.
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 **DbIntegrate** defined. It can also contain a comma separated list of the above.

Examples

To replicate only rows of table **product** 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





See **hvr_demo26** in **$HVR_HOME/demo**.

6.6 CollisionDetect

Description

Action **CollisionDetect** causes HVR's integration to detect 'collisions', i.e. when a target row has been changed after the change that is being applied was captured. Collisions can happen during bi–directional replication; 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 danger that the change to A will we applied to B while the change to B is on its way to A. Collisions can also happen without bi–directional replication; if changes are made first to A and then to B, then the change to B could reach database C before the change from A reaches C. Undetected collisions can lead to inconsistencies; the replicated database will become more different as time passes.
The default behavior for **CollisionDetect** is automatic resolution using a simple rule; the most recent change is kept and the older changes discarded. The timestamps used have a granularity of one second; if the changes occur in the same second then one arbitrary location (the one whose name sorts first) will 'win'. Parameters are available to change this automatic resolution rule and to tune performance.
Collision detection requires that HVR maintains extra timestamp information for each tuple, unless parameter **/TimestampColumn** is defined. This information is held in a special history table (named *tbl_*__**h**). This table is created and maintained by both capture and integration for each replicated table. The old rows in this history table are periodically purged using timestamp information from the integrate receive timestamp table (see also section Integrate Receive Timestamp Table).

Parameters

Parameter	Argument	Description
/TreatCollisionAsError		Treat a collision as an error instead of performing automatic resolution using the 'first wins' rule. If **DbIntegrate /OnErrorSaveFailed** is defined then the collision will be written to the fail table and the integration of other changes will continue. If **DbIntegrate /OnErrorBlockLocation** is defined then the location from which this change originated will be blocked. If neither of these are 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 are more than one database where changes can occur then if a change occurs in the same second, then 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. If this parameter is defined with parameter **/Context** then it can be used for refresh only without enabling collision detection during replication.
/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 **CollisionDetect /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 or DB2 database depending on its form. See further section Calling HVR on the Command Line. The

following options are allowed:

–a	Purge all history, not just changes older than receive stamps.
–f*freq*	Commit frequency. Default is commit 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.

Examples

See channels **hvr_demo11** and **hvr_demo23** in **$HVR_HOME/demo**.

6.7 DbSequence

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 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.

Parameters

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.

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 (see SESSION NAMES AND RECAPTURING in section DbCapture).

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.

Notes

Capture of database sequences requires log–based capture (**DbCapture /LogBased**).
This action is not supported for SQL Server databases, because SQL Server does not have DBMS sequences.
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.

6.8 DbObjectGeneration

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 **/NoCapture*** 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

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 **hvrload** can be used to generate the initial contents for this file.
/IncludeSqlDirectory	*dir*	Search directory *dir* for include SQL file.
/CaptureTableCreateClause	*sql_expr*	Clause for 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. 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.

Injecting SQL Include Files

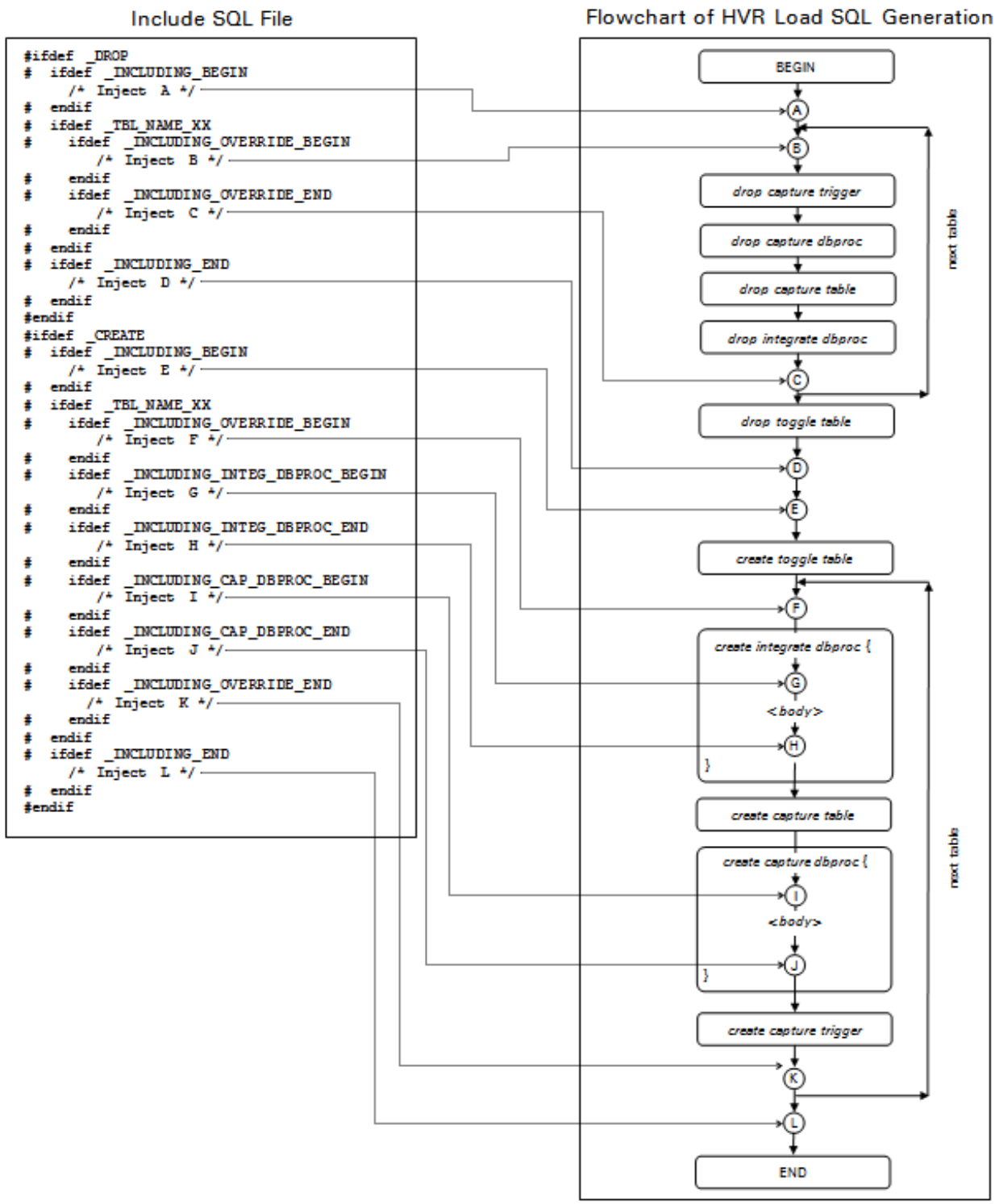
Parameter **/IncludeSqlFile** can be used to inject special logic inside standard SQL which is generated by **hvrload**. The SQL that HVR would normally generate can be seen with **hvrload** option **–S**. Conditions (using **#ifdef** syntax lent from the C preprocessor), control where abouts this SQL 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 (**/LogBased** not defined)
- Integrate database procedures are only defined if **DbIntegrate/DbProc** is defined.
- The **_CREATE** section is omitted if **hvrload** option **–d** is defined without **–c**.
- Sections for specific tables are omitted if **hvrload** option **–t** is specified for different tables.
- Database procedures are only generated if **hvrload** 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 **hvrload** 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 **hvrload** is creating database objects.
_DB_CAPTURE	Defined if action **DbCapture** is defined on this location.
_DB_INTEGRATE	Defined if action **DbIntegrate** is defined on this location.
_DBPROC_COL_NAMES	Contains the list of columns in the base table, separated by commas.

Macro	Description
_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 **hvrload** is dropping database objects.
_FLAG_OC	Defined when **hvrload** option **–oc** or no **–o** option is supplied.
_FLAG_OP	Defined when **hvrload** option **–op** or no **–o** option is supplied.
_FLAG_OS	Defined when **hvrload** option **–os** or no **–o** option is supplied.
_FLAG_OT	Defined when **hvrload** 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 **hvrload** is including the SQL file at the beginning of its SQL.
_INCLUDING_END	Defined when **hvrload** is including the SQL file at the end of its SQL.
_INCLUDING_CAP_DBPROC_BEGIN	Defined when **hvrload** is including the SQL file at the beginning of each capture database procedure.
_INCLUDING_CAP_DBPROC_DECLARE	Defined when **hvrload** is including the SQL file for the declare block of each capture database procedure.
_INCLUDING_CAP_DBPROC_END	Defined when **hvrload** is including the SQL file at the end of each capture database procedure.
_INCLUDING_INTEG_DBPROC_BEGIN	Defined when **hvrload** is including the SQL file at the beginning of each integrate database procedure.
_INCLUDING_INTEG_DBPROC_DECLARE	Defined when **hvrload** is including the SQL file for the declare block of each integrate database procedure.
_INCLUDING_INTEG_DBPROC_END	Defined when **hvrload** is including the SQL file at the end of each integrate database procedure.
_INCLUDING_OVERRIDE_BEGIN	Defined as **hvrload** 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 **hvrload** 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.

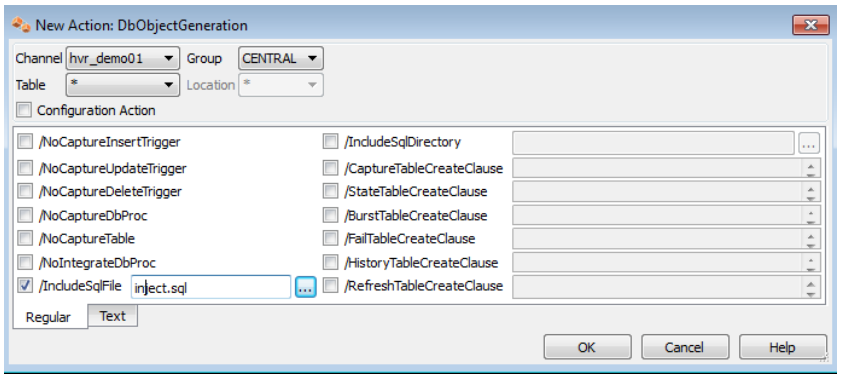


Examples

The following uses action **DbObjectGeneration** to inject some special logic (contained in file **inject.sql**) into the integrate database procedure for table **scenario2**. 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 **DbIntegrate** so that integrate database procedures are generated.

```
#if defined _INCLUDING_INTEG_DBPROC_BEGIN && \
   defined _TBL_NAME_SCENARIO2 && \
   _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 scenario2 where id =  :id;
        return;
endif;
#endif
```



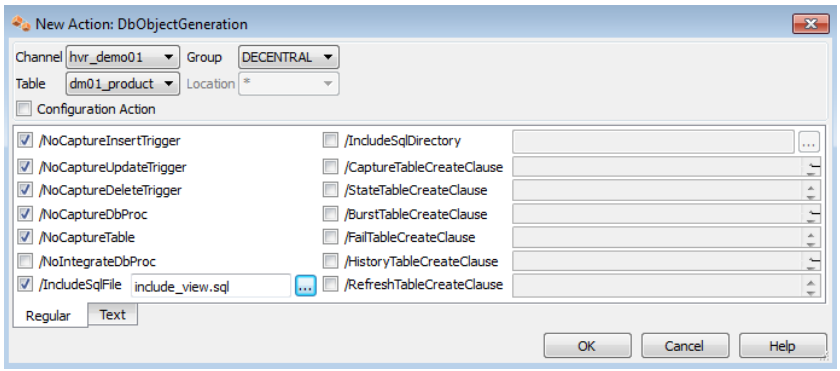
The following example replicates updates to column **balance** of table **account** as differences, instead of as absolute values. The channel should contain the following actions: **DbCapture** (not log–based), **DbIntegrate /DbProc** (at least for this table) and **DbObjectGeneration /IncludeSqlFile=***thisfile*.

```
#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
```

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 **DbCapture** 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
```



Notes:

- If long datatypes 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 **select** with an outer–join to the base table (which could also be a view).
- Commands **hvrcompare** and **hvrrefresh** can also have views on the 'read' side instead of regular tables. If **DbIntegrate /DbProcDuringRefresh** is defined then row–wise refresh can also select from a view on the 'write' side before applying changes using a database procedure.

6.9 FileCapture

Description

Action **FileCapture** instructs HVR to capture files from a file location's directory.
If a channel is defined with table information then it can replicate both database and file location. In this case any files captured from the file location are parsed as change records in XML format. The XML schema used by HVR can be found in **$HVR_HOME/lib/hvr.dtd**. A **Transform** action can be used to map a different format (e.g. CSV) into this XML.
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 **FileCapture** and **FileIntegrate** (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.
Bi directional replication (replication in both directions with changes happening in both file locations) is not currently supported.
File deletion is not currently captured by HVR.
If **FileCapture** is defined 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

Parameter	Argument	Description
/DeleteAfterCapture		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.

Parameter	Argument	Description
/Pattern	*pattern*	Only capture files whose names match pattern. The default pattern is '**/*' which means search all subdirectories and match all files. Possible patterns are: • '***.c**' – Wildcard, for files ending with **.c**. A single asterisk matches all or part of a filename or subdirectory name. • '****/*txt**' – Recursive Subdirectory Wildcard, to walk through the directory tree, matching files ending with **txt**. A double asterisk matches zero, one or more subdirectories but never matches a filename or part of a subdirectory 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. • A 'named pattern' such as **{office}.txt**. The value inside the braces must be an identifier. The 'named pattern' matches the same as a wildcard (*) but also sets a property that can be used for a 'named substitution' (see parameter **/RenameExpression** of action **FileIntegrate**). For example, 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**. If a file is matched with 'named pattern' **{hvr_address}**, then it is only replicated to integrate locations specified by the matching part of the filename. Locations can be specified as follows: – 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 **Restrict /AddressSubscribe**, for example **22** or **Alias7**. – A list of the above, separated by a semicolon, colon or comma, such as **cen,dec01**. 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	*pattern*	Ignore files whose names match *pattern*. For example, to ignore all files underneath subdirectory **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	*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 encodings.
/AccessDelay	*secs*	Delay read for *secs* seconds to ensure writing is complete. HVR will ignore this file until its last create or modify timestamp is > *secs* seconds old.
/UseDirectoryTime		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 subdirectory 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

It is often better to avoid HVR capturing files until is completely 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 subdirectory and then moving the whole subdirectory into the file location's top directory together.

XML Format

The following is an example of an XML file containing changes which can be replicated to 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="hvr_op">0</column>
            <column name="prod_id">2</column>
            <column name="prod_price" is_null="true"/>
            <column name="prod_descrip" format="hex"/>ffff<column>
        </row>
   </table>
</hvr>
```

Notes

- 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**).

6.10 FileIntegrate

Description

Action **FileIntegrate** instructs HVR to integrate files into a file location's directory.
If a channel is defined with table information then it can replicate both database and file location. In this case any changes replicated to the file location are integrated as records in XML format. The XML schema used by HVR can be found in **$HVR_HOME/lib/hvr.dtd**. This XML data can be mapped to another format (e.g. CSV) using a **Transform** action.
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 **FileCapture** and **FileIntegrate** (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.

Parameters

Parameter	Argument	Description
/RenameExpression	*expression*	Expression to name new files. A rename expression can contain constant strings mixed with substitutions. Possible substitutions include: • **{hvr_cap_loc}** is replaced with the name of the capture location. • **{hvr_cap_tstamp}** or **{hvr_cap_tstamp** *spec***}** is replaced with the capture time. The time is in GMT. The default format is *YYYYMMDDhhmmss*, but this can be overwritten with parameter *spec* (see paragraph TIMESTAMP FORMAT SPECIFIER below). • **{hvr_integ_tstamp}** or **{hvr_integ_tstamp** *spec***}** is replaced with the time that the file was integrated. The time is in GMT. The time is in GMT. The default format is *YYYYMMDDhhmmssSSS*, but this can be overwritten with parameter *spec* (see section TIMESTAMP FORMAT SPECIFIER below). • If the file was captured from a file location, then substitutions **{hvr_cap_subdirs}** and **{hvr_cap_filename}** are also allowed. • **{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**. • If the file was captured with a 'named pattern' (see parameter **DbCapture /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**. • **{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_schema}** is replaced with the schema name of the table. This is only allowed if the channel is defined with the tables. • **{***colname***}** is replaced with the current value of column *colname*. This must be a column of the current table. If no rename expression is defined, then new files are named **{hvr_cap_subdirs}/{hvr_cap_filename}** if they were captured from another file location or **{hvr_integ_tstamp}.xml** if they are for database changes and channel is defined with the tables.
/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.

Parameter	Argument	Description
/MaxFileSize	*X*	Limit each XML file to no more than *X* bytes. This parameter cannot be used for 'blob' file channels which contain no table information and only replicated files as 'blobs'. When this size is reached, 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.
/Journal		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** with option **–F**. Old journal files can be purged with command **hvrmaint –journal_keep_days=***N*. 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.
/Burst		Integrate changes into target file(s) using Burst algorithm. All changes for the cycle are first sorted and coalesced, so that only a single change remains for each row to the source table (see parameter **/Coalesce**). These changes are then written into target file(s). The order in which the changes are written is completely different from the order in which they occurred on the capture machine.
/ConvertNewlinesTo	*OSTYPE*	Write files with UNIX or DOS style newlines Value *OSTYPE* must be **UNIX** or **DOS**. Value **Unix** means any carriage–return value (hex **0x0d**) bytes are removed. Value **DOS** means that each linefeed is prefixed with a carriage–return (hex **0x0a** becomes **0x0d0a**) unless it already has a carriage–return prefix. New line conversion will work with text files of most encodings, including ASCII and UTF–8, but is not supported with UTF–16 (on Windows a 'Unicode' file is UTF–16) or binary files.
/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.
/OnErrorSaveFailed		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.
/CycleByteLimit	*int*	Maximum amount of routed data (compressed) to process per 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). The default is **10 Mb**. Value 0 means unlimited, so the integrate job will process all available work in a single cycle. If the supplied value is smaller than the size of the first transaction file in the **router** directory, then all the transactions in that file will be processed. Transactions in a transaction file will never be split between cycles or sub-cycles.
/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 **FileIntegrate /RenameExpression /Context=qqq** is defined, then the file will only be renamed if context **qqq** is enabled (option **-Cqqq**).

XML Format

The following is an 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 of HVR's XML

The following 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 **DbCapture**, **FileIntegrate** and **ColumnProperty /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
   values (1, 'hello',
   '746f206265206f72206e6f7420746f2062652c0074686174206973207320746865');
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 **ColumnProperty /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>
     <column name="a">1</column>      <-- Note: Hvr_op=1 means insert -->
     <column name="b">hello</column>
     <column name="c" format="hex">        <-- Note: Binary shown in hex       -->
                                             <-- Note: Text after hash is comment -->
       746f 2062 6520 6f72 206e 6f74 2074 6f20 # to be or not to
       6265 2c00 7468 6174 2069 7320 7468 65   # be,.that is the
     </column>
   </row>
   <row>
     <column name="hvr_op_val">1</column>
     <column name="a">2</column>
```

```
      <column name="b">&lt;world&gt;</column> <-- Note: Standard XML escapes used -->
      <column name="c" format="hex">
        7175 6573 7469 6f6e           # question
      </column>
    </row>
  </table>                  <-- Note: Table tag switches current table -->
  <table name="mytab">
    <row>
      <column name="hvr_op_val">1</column>
      <column name="aa">33</column>
      <column name="bb">2012-09-17 17:32:27</column> <-- Note: HVRs own date format -->
    </row>
  </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">&lt;world&gt;</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 -->
```

Timestamp Format Specifier

Substitutions **{hvr_cap_tstamp** *spec*} and**{hvr_integ_tstamp** *spec*} allow the format of the timestamp to be specified with sequences from the **strftime(3)** subroutine in **C**. Below are the most useful sequences:

Param	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

Param	Description	Example
%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
%y	Year without century.	14
%Y	Year including century.	2014

The default format for **{hvr_integ_tstamp}** is YYYYMMDDhhmmssSSS. This is equivalent to **%Y%m%D%H%M%S** plus milliseconds, which cannot be specified with any **%** format. This means that if a format is supplied with **{hvr_integ_tstamp** *spec*}, then HVR will throttle file integration to no faster than one file per second.
Time formats can only be specified for **{hvr_cap_tstamp}** and **{hvr_integ_tstamp}** in **FileIntegrate /RenameExpression**, not for other HVR parameters such as **/IntegrateExpression** or **/CaptureCondition**.

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.

6.11 Transform

Description

Action **Transform** defines a data mapping that is applied inside the HVR pipeline. Multiple transforms can be defined, both on the capture and integrate side. A transform can either be a command (a script or an executable), or it can be written in XSLT (Extensible Stylesheet Language Transformation), or it can be a transform which is built into HVR (parameter **/Builtin**).
Action **Transform** can only be used for file locations or database locations.
If a channel also contains database locations and table information, then HVR capture expects to read HVR's XML, and HVR's integrate will output HVR's XML. The **Transform** action can be defined to capture from a file location to convert input files into HVR's XML or to integrate into a file location to output files in a different format.
If the channel is not defined with table information (a 'blob' file channel), then neither capture nor integrate transformation pipelines need to conform to HVR's XML.
When a **Transform** action is defined for a file location then it will be called once for each file, unless parameter **/FileTags** is defined.
If a **Transform** action is defined on a database location it 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).
A transform action defined on a database location should be able to be called by HVR in different 'modes'. The transform mode is given in environment variable **$HVR_TRANSFORM_MODE**. For example, on the target side the mode will normally be **integ**, but during row–wise refresh and compare the transform will also be called in mode **refr_read** or **cmp_read**. The transform should work in reverse during these modes.

Parameters

Parameter	Argument	Description
Row2Csv		Write file as Comma Separated Value, instead of XML. Field and line separators and encoding can be defined with option in **/ArgumentValue**.
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**.
/XsltStyleSheet	*sheet*	Pathname of XSLT stylesheet. XSLT is a declarative, XML–based language for transforming XML documents. The value should be an absolute pathname on the hub machine, even if the transform if defined for a remote location.
SoftDelete		Convert deletes into updates of a boolean soft–delete column.
Xml2Csv		Convert XML to Comma Separated Value.
Csv2Xml		Convert Comma Separated Value to XML.
File2Column		Load the contents of a file into a column with LOB datatype.
Tokenize		Encrypt columns using a tokenization server.
SapAugment		Capture job selecting for de-clustering of multi-row SAP cluster tables.
SapXForm		Invoke SAP transformation for SAP pool and cluster tables. This feature requires SAP decluster engine, which is not included in regular HVR distribution. Also requires **/ExecOnHub**.
/Builtin	*type*	Use built–in HVR Transform. Valid values are: • **SoftDelete**: convert deletes into updates of a boolean soft–delete column • **Xml2Csv**: convert XML to Comma Separated Value • **Csv2Xml**: convert Comma Separated Value to XML • **File2Column**: load the contents of a file into a column with LOB datatype • **Tokenize**: encrypt columns using a tokenization server.
/Order	*int*	Specify order of transformations. This allows a pipeline of transforms to be defined within a job. Both command and XSLT transforms can be interleaved. If more than one transform is defined on the same location without this parameter, then the execution order is undefined.

Parameter	Argument	Description
/ArgumentName	*names*	Parameter name for each argument value. This is needed to pass parameters to the XSLT stylesheet, which can then get their values using expression **<xsl:param name="***name***"/>**. Multiple argument names should be supplied by separating each name with a space.
/ArgumentValue	*vals*	Values for transform parameters. Multiple arguments can be supplied by separating each value with a space. A space inside an argument value should be quoted (e.g. **"hello" "world 2"**). Quoted arguments can also contain these escaped values: **\"** **** **\n** **\t** and **\r**. Argument values can be defined for command or XSLT transforms. For XSLT transforms there must be a **/ArgumentName** element for each **/ArgumentValue** element.
/FileTags		Transform *file* and *bytes* tags instead of file contents.
/ExecOnHub		Execute transform on hub instead of location's machine
/Context	*context*	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 **Transform /BuiltIn=SoftDelete /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 Transforms

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 with **/ArgumentValue**.

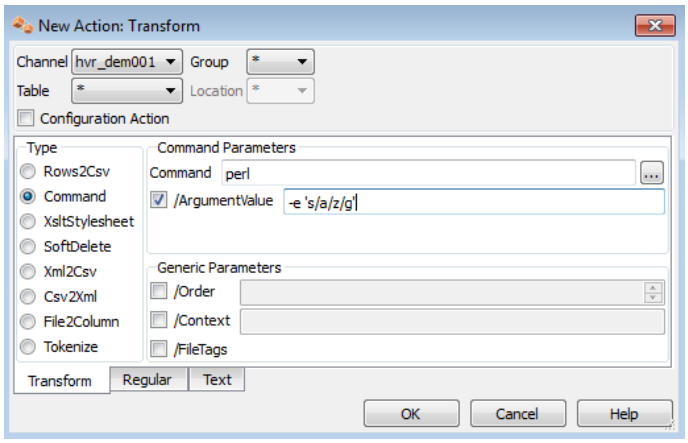
A transform inherits the environment from its parent process. On the hub, the parent of the transform's parent process is the HVR Scheduler. On a remote Unix machine, it is the **inetd** daemon. On a remote Windows machine it is the HVR Remote Listener service. Differences with the environment process 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 **$HVR_VAR_***XXX* are defined for each context variable supplied to HVR Refresh or Compare (option **–V***xxx*=*val*).
- For database locations, environment variable **$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 **$HVR_LOC_STATEDIR** are set to the file location's top and state directory respectively.
- If the transform is an integrate job for a 'blob' file channel without table information, or is a capture job, then environment variables **$HVR_CAP_LOC**, **$HVR_CAP_TSTAMP**, **$HVR_CAP_FILENAME** and **$HVR_CAP_SUBDIRS** are set with details about the current file.
- If the transform is defined on a file location, then variable **$HVR_FILE_PROPERTIES** contains a colon–separated *name*=*value* list of other file properties. This includes values set by 'named patterns' (see **FileCapture /Pattern**).
- Any variable defined by action **Environment** is also set in the transform'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 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.

The output of the last transform in the capture job must be HVR's own XML, and the input of the first transform in the integrate job will be HVR's own XML. This XML is described in section FileIntegrate.

Command Tranform Examples



A simple example is **Transform /Command=perl /ArgumentValue="–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. Transform **hvrcsv2xml.pl** transforms CSV files (Comma Separated Values) to HVR's XML.
Transform **hvrxml2csv.pl** transforms HVR's XML back to CSV format. And **hvrfile2column.pl** transforms the contents of a file into a HVR compatible XML file; the output is a single record/row.

XLST Transforms

XSLT (Extensible Stylesheet Language Transformation) is a declarative, XML–based language for transforming XML documents. HVR's implementation conforms to XSLT version 1.0, so developers are referred to either an XSLT 1.0 tutorial on the internet, or to a standard reference book.
Arguments for an XSLT transform can be passed by defining parameters **/ArgumentName** and **/ArgumentValue**. An additional XSLT parameter named **hvr_mode** is also passed, either with value **cap_transform** or **integ_transform**.
The input to an XSLT stylesheet must be valid XML. The output of the last transform in the capture job must be HVR's own XML, and the input of the first transform in the integrate job will be HVR's own XML. This XML is described in section FileIntegrate.
Command**hvrxslt** transforms the contents of an input file using the mapping implied by an XSLT stylesheet. This can be used to test a transform defined with**/XsltStylesheet**.

```
$ hvrxslt [-anm =val]... sheet ifile > ofile
```

Option *–anm=val* can be used to pass a parameter to the XSLT. This can get the value using expression **<xsl:param name="***nm***"/>**. Note that if the parameter value is a string (as opposed to a node identifier) then the string must be quoted for XSL. For example the following command passes string **world** to parameter **hello** using single quotes for XSL escaping and double quotes for UNIX escaping; **hvrxslt "–ahello='world"'**.

XSLT Transform Example

Stylesheet **$HVR_HOME/demo/xslt/hvr2csv.xsl** (shown below) converts from HVR's XML schema into a simple comma–separated file format. It accepts two optional parameters named **field_separator** and **line_separator** which default to a comma and newline respectively.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="text"/>
    <xsl:param name="field_separator" select="','"/>
    <xsl:param name="line_separator" select="'&#x0A;'"/> <!-- newline -->
    <xsl:template match="/">
        <xsl:for-each select="hvr/table/row"> <!-- For each row -->
            <xsl:for-each select="column">    <!-- For each column -->
                <xsl:value-of select="."/>
                <xsl:if test="position()  != last()">
                    <xsl:value-of select="$field_separator"/>
                </xsl:if>
            </xsl:for-each>
            <xsl:value-of select="$line_separator"/>
        </xsl:for-each>
    </xsl:template>
</xsl:stylesheet>
```

Suppose the above stylesheet was applied to an XML file **test.xml** containing the following;

```
<?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">iPod</column>
        </row>
    </table>
</hvr>
```

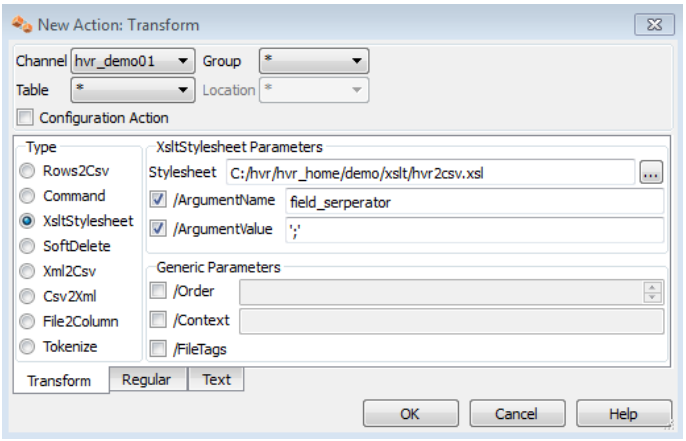
Stylesheet **hvr2csv.xsl** would transform the above file into the following output;

```
1,30,DVD
2,300,iPod
```

To use a semicolon as field separator (instead of a comma), use the following command;

```
$ hvrxslt "-afield_separator=';'" $HVR_HOME/demo/xslt/hvr2csv.xsl test.xml
```

The above XSLT stylesheet is included in the channel using the following **Transform** action.



Built-in Transform Options

Some transforms defined with parameter **/Builtin** allow options in the **/ArgumentValue** parameter. Options for **/Builtin=Xml2Csv**

–c *encoding* Encoding of output file. Default is UTF–8.

–e *escapechar*	Escape character for the quote character inside quoted fields.
–F	Enable formal parsing.
–f *fieldsep*	Character that delimits columns. Default: comma (,).
–h	Generate line with column headers.
–l *linesep*	Line separator. The default is **\n** on Unix and **\r\n** on Windows.
–q *quotechar*	Character to quote values that contain field separator. Default: quote (\").
–s *style*	Quoting style. Must be one of the following: **none** = no quoting. **escape** = ” and \". **double** = ” and “”.
–z *term*	Termination line printed before end.

Options for **/Builtin=Csv2Xml**

–c *encoding*	Encoding of output file. Default is UTF–8.
–e *escapechar*	Escape character for the quote character inside quoted fields.
–F	Enable formal parsing.
–f *fieldsep*	Character that delimits columns. Default: comma (,).
–h	Generate line with column headers.
–l *linesep*	Line separator. The default is **\n** on Unix and **\r\n** on Windows.
–n	Convert empty values to NULL if that value is nullable.
–q *quotechar*	Character to quote values that contain field separator. The default is a quote (\").
–s *style*	Quoting style. Must be one of the following: **none** = no quoting. **escape** = ” and \". The quote and escape can be overwritten with options **–e** and **–q**. **hvr3** = backward compatibility with HVR version 3.
–t *table*	Required if specified on more than one table.
–z *term*	Termination line printed before end.

Options for **/Builtin=File2Column**

–c *col=value*	Specify column values. Use **{hvr_file_contents}** as value for file input data. Other common patterns such as **{hvr_cap_filename}** available.
–t *table*	Required if specified on more than one table.

Options for **/Builtin=Tokenize**

–e *endpoint*	URL of tokenization server.
–s *tokenization_schema*	Schema name in server.
–t *table*	Table for following **–c** parameters.
–c *column*	Name of column to tokenize

Notes

If a **Transform** action is defined on a specific table, then it affects the entire job including data from other tables for that location.
Action **FileIntegrate /MaxFileSize** can be used with an XSLT transform on an integrate location to limit the amount of memory used at runtime (the XSLT engine reads the entire file into memory).
The transform command can only change the contents of the file, not its file name (use **FileIntegrate /RenameExpression** for this).

6.12 SalesforceCapture

Description

Rows can be read from Salesforce locations using action **SalesforceCapture** 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 be scheduled to run continuously. Instead it can be run manually or periodically with action **Scheduling /CaptureStartTimes**.
A channel with **SalesforceCapture** 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.

Parameters

Parameter	Argument	Description
/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.
/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.

Notes

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.
A capture restriction can be defined on a Salesforce location in SOQL using action **Restrict /CaptureCondition**.

6.13 SalesforceIntegrate

Description

A channel containing database locations can write rows into Salesforce locations using action **SalesforceIntegrate**. Attachments can also be written into a Salesforce location by defining a 'blob' file channel with **FileCapture** and **SalesforceIntegrate**.
If a channel with database capture 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

Parameter	Argument	Description
/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.
/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.
/OnErrorSaveFailed		Write failed rows/files to fail directory. The failed rows are saved within **$HVR_CONFIG/work/***hub*/*chn*/*loc*/**sf** and an error message is written, which describes how the rows can be retried.
/TableName	*userarg*	API name of Salesforce table into which attachments should be uploaded. See paragraph SALES FORCE ATTACHMENT INTEGRATION below.
/KeyName	*userarg*	API name of key in Salesforce table for uploading attachments. See paragraph INTEGRATING ATTACHMENTS below.
/Journal		Move processed transaction files to journal directory **$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.

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 **SalesforceIntegrate /TableName** or using 'named pattern' **{hvr_sf_tbl_name}** in the **FileCapture /Pattern** parameter. Likewise, the key column can be defined either with **SalesforceIntegrate /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 **FileCapture /Pattern="{hvr_sf_key_value}.jpg"** should be used with action **SaleforceIntegrate /TableName="Employee" /KeyName="EmpId"**.

Notes

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.

6.14 Agent

Description

An agent is a block of user–supplied logic which is executed by HVR during replication. An agent can be an Operating System command or a database procedure. Each time HVR executes an agent it passes parameters to indicate what stage the job has reached (e.g. start of capture, end of integration etc.).

Parameters

Parameter	Argument	Description
/Command	*Path*	Name of agent 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**. 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**.
/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 invocation by the HVR job.
/UserArgument	*Userarg*	Pass extra argument *userarg* to each agent execution.
/ExecOnHub		Execute agent on hub machine instead of location's machine.
/Order	*Int*	Specify order of agent execution.
/Path	*Dir*	Search directory *dir* for agent.

Agent Arguments

If an agent 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 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 was called. Agents are not called during HVR Compare.
Modes **cap_end** and **integ_end** are passed information about whether data was actually replicated. Command agents can use **$HVR_TBL_NAMES** or **$HVR_FILE_NAMES** and database procedure agents 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**.
Agents specified with **/Command** are called as follows:

```
agent mode chn_name loc_name userarg
```

Database procedure agents (specified with modifier **/DbProc**) are called as follows. The last parameter (**hvr_changed_tables**) specifies the number of tables that were changed.

Ingres

```
execute procedure agent (hvr_agent_mode='integ_end', hvr_chn_name='chn', hvr_loc_name='xx',
    hvr_agent_arg='hello', hvr_changed_tables=N);
```

Oracle

```
agent (hvr_agent_mode$=>'integ_end', hvr_chn_name$=>'chn', hvr_loc_name$=>'xx', hvr_agent_arg$
    =>'hello', hvr_changed_tables$=N);
```

SQL Server

```
execute agent @hvr_agent_mode='integ_end', @hvr_chn_name='chn', @hvr_loc_name='xx',
    @hvr_agent_arg='hello', @hvr_changed_tables=N;
```

Agent Environment

An agent inherits the environment of its parent process. On the hub the parent of the agent'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. If an extremely large number of files are replicated then this value could be abbreviated and suffixed with "**...**".
- Any variable defined by action **Environment** is also set in the agent's environment.
- The current working directory for local file locations (not FTP, SFTP or SharePoint/WebDAV) 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 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.

Dependencies

Parameter **/DbProc** cannot be used with parameters **/Command** or **/Path**.

Example

```
#!perl

# Exit codes: 0=success, 1=error, 2=ignore_mode

if($ARGV[0] eq "integ_end") {
    print "Hello World\n";
    exit 0;
}
else {
    exit 2;
}
```

6.15 Environment

Description

This action 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 slave 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

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 **0** 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	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**).
HVR_SORT_BYTE_LIMIT	Amount of memory to use before sorting large data volumes in temporary files. The default limit is **67108864** (64Mb).
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 **1038336** (1Mb).

6.16 LocationProperties

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

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).
/ThrottleMillisecs	*int*	Restrict network bandwidth usage by sleeping *int* millisecs between packets. See **/ThrottleKbytes** for setting the package size.
/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.

Parameter	Argument	Description
/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**.
/Proxy	*url*	URL of proxy server. 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. Or, 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.
/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.
/StagingDirectoryHvr	*URL*	Directory for bulk load staging files. For Greenplum this should be a local directory on the machine where HVR connects to Greenplum. For Redshift it should be an S3 location.
/StagingDirectoryDb	*URL*	Location for the bulk load staging files visible from the Database. Should point to same files as **/StagingDirectoryHvr**. For Greenplum this should either be a local directory on the Greenplum headnode, or a it should be a URL which points to **/StagingDirectoryHvr**, for example a path starting with **gpfdist:**. For Redshift it should be the same S3 location that is used for **StagingDirectoryHvr**.
/StagingDirectoryCredentials	*credentials*	Credentials to be used for S3 authentication during RedShift bulk load. • Use credentials. You can retrieve these from the amazon console: **aws_access_key_id=**<*access-key-id*>**;aws_secret_access_key=**<*secret-access-key*> • Use temporary credentials from the role of the current ec2 node: **role=**<*role*>

Dependencies

If one of the parameters **/ThrottleMilliSecs** or **/ThrottleKbytes** is defined, then the other must also be defined.

6.17 Scheduling

Description

Action **Scheduling** controls how the replication jobs generated by **hvrload** 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 **DbCapture /ToggleFrequency**.
If this action is defined on a specific table, then it affects the entire job including data from other tables for that location.

Parameters

Parameter	Argument	Description
/CaptureStartTimes	*times*	Defines that the capture jobs should be triggered at the given times, rather than cycling continuously. For the format of *times* see START TIMES below.
/CaptureOnceOnTrigger		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 **hvrtrigger**.
/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 START TIMES below.
/IntegrateOnceOnTrigger		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 **hvrtrigger**.
/RefreshStartTimes	*times*	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 START TIMES below.
/CompareStartTimes	*crono*	Defines that the compare jobs should be triggered at the given times. By default they must be triggered manually. For the format of *times* see START TIMES below.

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.

Example

/CaptureStartTimes="0 * * * 1–5" specifies that capture jobs should be triggered at the start of each hour from Monday to Friday.

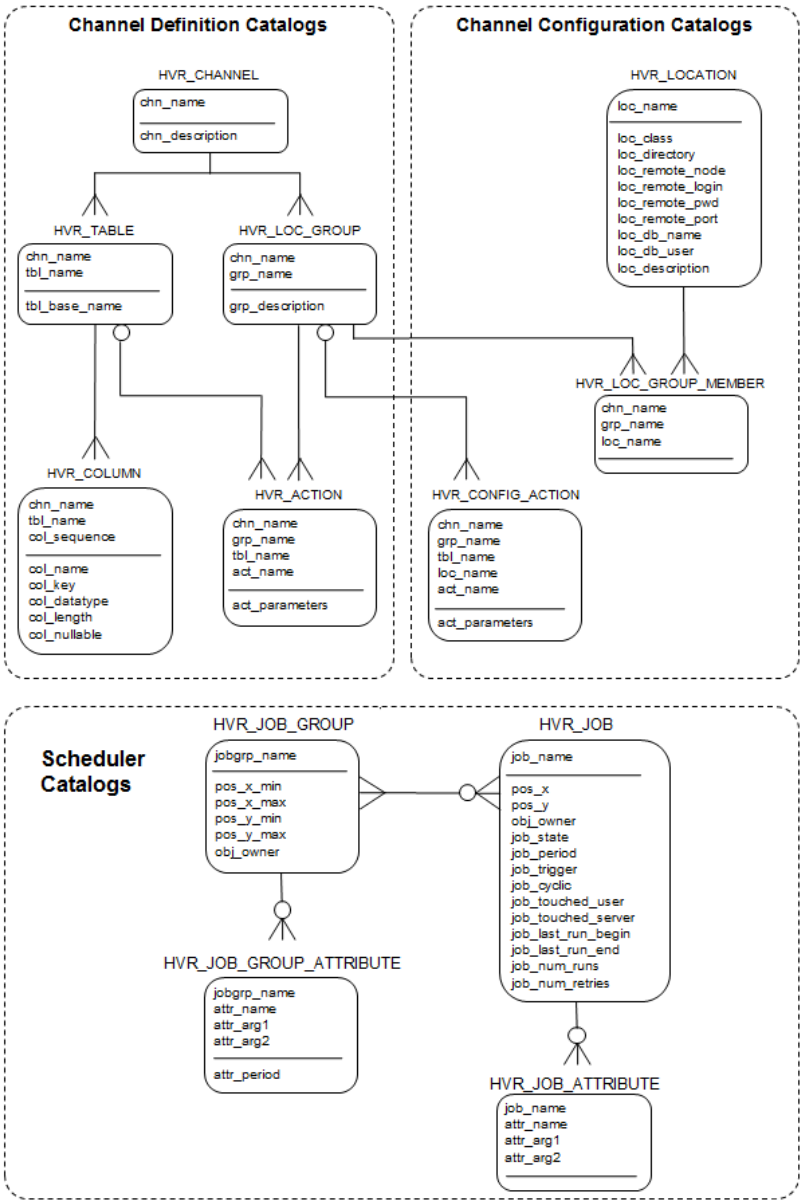
7

Objects Used Internally by HVR

This chapter describes the database objects HVR uses inside its hub database and also the objects it creates inside replicated databases.

7.1 Catalog Tables

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.



HVR Channel

Column	Datatype	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.

Column	Datatype	Optional?	Description
chn_description	String 200 characters	Yes	Description of channel.

HVR_TABLE

Column	Datatype	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	Datatype	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	number	Yes	Is column part of the unique replication key? Value are 0 (column not in key) or 1 (column is in key). Key information is needed to replicate updates and deletes. The replication key does not have to match a physical unique index in the replicated table. If necessary, uniqueness can be achieved by defining all columns as part of the replication key. If this is still not a unique key then action **TableProperties /DuplicateRows** must be defined.
col_datatype	string 38 characters	No	Datatype of column. Any database type can be used here, i.e. **varchar**, **varchar2**, **char**, **integer**, **integer4**, **number** or **date**.
col_length	string 8 characters	Yes	The meaning of this column depends on the datatype: String datatypes such as **binary**, **byte**, **c**, **char**, **text**, **raw**, **varchar**, **varchar2**: Maximum length of string. Datatypes **number** and **decimal**: 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. Other datatypes: Unused.
col_nullable	number	No	Is column datatype nullable? Values are 0 (not nullable) or 1 (nullable).

HVR_LOC_GROUP

Column	Datatype	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	Datatype	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	Datatype	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. **teradata** Teradata database. **paraccel** Paraccel database. **greenplum** Greenplum database. **file** File location, including FTP, SFTP and WebDAV/SharePoint. **salesforce** Salesforce.com connection.

Column	Datatype	Optional?	Description
loc_directory	string 200 characters	Yes	The meaning of this column depends on the contents of **loc_class**. **oracle**: The value of **$ORACLE_HOME**; it defaults to the setting on the hub machine. **ingres**: The value of **$II_SYSTEM**; it defaults to the setting on the hub machine. **sqlserver**: Unused. **db2**: Unused. **teradata**: Unused. **paraccel**: Unused. **greenplum**: Unused. **file**: Top directory of file location. For a non–local directory it has the form of a URL: **ftp**[**s**]**://**[*login*[**:***pwd*]**@**]*host*[**:***port*]**//***path* or **sftp://**[*login*[**:***pwd*]**@**]*host*[**:***port*]**//***path* or **webdav**[**s**]**://**[*login*[**:***pwd*]**@**]*host*[**:***port*]**/***path*. Note that HVR requires two slashes (//) before the FTP or SFTP directory path; this means a path relative to the FTP/SFTP server's root, not relative to its home directory. Explicit SSL or TLS encryption is indicated using symbol **~S** or **~T** respectively, e.g. **ftps://***login***:***pwd***@***host***~S//***path*. **salesforce**: The URL of the Salesforce endpoint. Pathnames can follow Unix or Windows conventions.
loc_remote_node	string 80 characters	Yes	Network name or IP address of the machine on which remote location resides. Only necessary for HVR remote connections.
loc_remote_login	string 24 characters	Yes	Login name under which HVR slave process will run on remote machine. Only necessary for remote HVR connections.
loc_remote_pwd	string 128 characters	Yes	Password for login name on remote machine. Only necessary for remote HVR connections. This column can be encrypted using command **hvrcryptdb**.
loc_remote_port	number	Yes	TCP/IP port number for remote HVR connection. On Unix the **inetd** daemon must be configured to listen on this port. On Windows the HVR Remote Listener Service listens on this port itself. Only necessary for remote HVR connections.
loc_db_name	string 128 characters	Yes	The meaning of this column depends on the value of **loc_class**. **ingres**: The Ingres database name. May contain an Ingres vnode (e.g. *vnode::db*), although a remote HVR connection should be more efficient over a network than Ingres/Net. **oracle**: The value of **$ORACLE_SID**; it defaults to the value on the hub machine. **sqlserver**: Value of the form *dbname* or *inst**dbname* where *dbname* is the name of the Microsoft SQL Server database and *inst* is the name of the SQL Server instance. May contain a node name (e.g. *node**inst**db*) although a remote HVR connection should be more efficient over a network than a SQL Server client/server connection. **file**: Unused. **salesforce**: Path of the Salesforce dataloader JAR file.

Column	Datatype	Optional?	Description
loc_db_user	string 128 characters	Yes	The meaning of this column depends on the value of **loc_class**. Passwords in this column can be encrypted using command **hvrcryptdb**. **ingres**: The DBA (owner) of the database. If specified, HVR connects to the location using Ingres **–u** flag and this value. **oracle**: Value of the form the [*user*]/[*pwd*], where *user* is the name of the Oracle user and *pwd* is the user's password. Both *user* and *pwd* are optional. If this column is empty, it defaults to the value used to connect to the hub database. May contain a TNS aliases (e.g. *user/pwd@node*) although a remote HVR connection should be more efficient over a network than SQL*Net. **sqlserver**: Value of the form [*user*]/[*pwd*] where *user* is the name of the Microsoft SQL Server user and *pwd* is the user's password. This value can be encrypted using command **hvrcryptdb**. **file**: Unused. **salesforce**: Value of the form [*user*]/[*pwd*] where *user* is the name of the Salesforce user and *pwd* is the user's password. This value can be encrypted using command **hvrcryptdb**.
loc_description	string 200 characters	Yes	Description of location.

HVR_LOC_GROUP_MEMBER

Column	Datatype	Optional?	Description
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	Datatype	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.

Column	Datatype	Optional?	Description
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_JOB

Column	Datatype	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 **hvrload** (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	Datatype	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	Datatype	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	Datatype	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 (*).

7.2 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**	Capture tables (trigger–based capture only).
*tbl*_ **_c1**	
*tbl*_ **_ci**	Capture database rules or triggers.
*tbl*_ **_cd**	
*tbl*_ **_cu**	
*tbl*_ **_c**	Capture database procedures (trigger–based capture only).
*tbl*_ **_c0**	
*tbl*_ **_c1**	
*tbl*_ **_l**	Database procedures, rules and triggers for capture of dynamic lookup table. Created for action **Restrict /DynamicHorizLookup**.
*tbl*_ **_li**	
*tbl*_ **_ld**	
*tbl*_ **_lu**	
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**	Integrate database procedures. Created for action **DbIntegrate /DbProc**.
*tbl*_ **_id**	
*tbl*_ **_iu**	
*tbl*_ **_b**	Burst tables. Used for staging if **DbIntegrate /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.
hvr_stbu*chn_loc*	Integrate burst state table. Created if action **DbIntegrate /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).

Notes

- 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.

7.3 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:

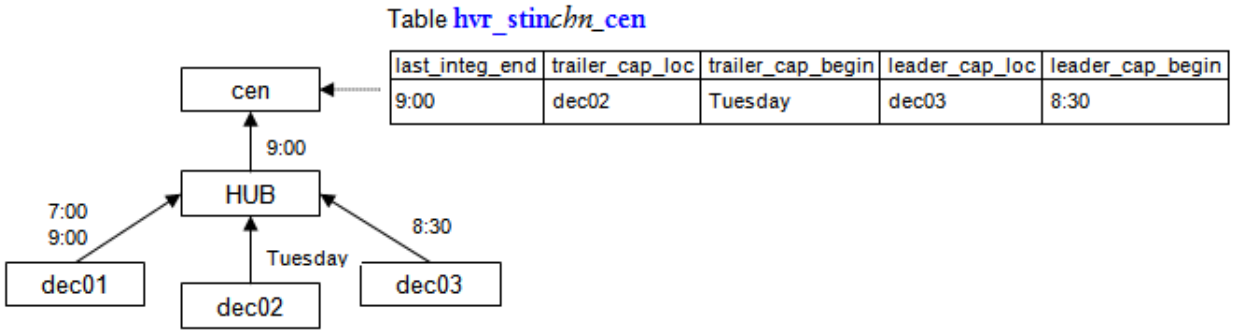
Column	Datatype	Description
hvr_seq	**Float** or **byte10** on Ingres, **numeric** on Oracle and **timestamp** on SQL Server.	Sequence in which capture triggers were fired. Operations are replayed on integrate databases in the same order, which is important for consistency.
hvr_tx_id	string	Transaction 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** statement.
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) or **5** (truncate table). 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 **DbCapture /NoBeforeUpdate**. During an online 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**.
hvr_cap_loc	string	Name of location on which the change was captured.
hvr_cap_tstamp	date, or integer if action **CollisionDetect /IntegerTimestamp** is defined	Time stamp of capture change, for collision detection.
hvr_cap_user	string	Name of user who performed captured change.
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.

7.4 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 Jan 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.

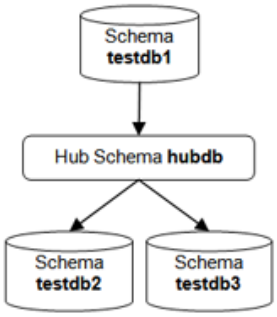


Appendices

A

Quick Start for HVR on Oracle

This appendix shows how to set up an HVR channel (called **hvr_demo01**) to replicate between Oracle databases. In real life HVR would usually replicate between Oracle instances on different machines. But for simplicity, in this example HVR will replicate between three schemas inside a single Oracle instance on the hub machine. The steps actually start by creating new users and tables for HVR to replicate between.



Create Test Schemas and Tables

Generally when getting started with HVR a source schema with tables and data already exists. If so then this step can be skipped.
This Quickstart uses two empty tables named **dm01_product** and **dm01_order**. In an existing Oracle database, create a test schema and create the tables using the following commands.

```
$ sqlplus system/manager
SQL> create user testdb1 identified by hvr
 2  default tablespace users
 3  temporary tablespace temp
 4  quota unlimited on users;

SQL> grant create session to testdb1;
SQL> grant create table to testdb1;
SQL> grant create sequence to testdb1;
SQL> grant create procedure to testdb1;
SQL> grant create trigger to testdb1;
SQL> grant create view to testdb1;
SQL> grant execute any procedure to testdb1;
```

Create the test tables.

```
$ cd $HVR_HOME/demo/hvr_demo01/base/oracle
$ sqlplus testdb1/hvr < hvr_demo01.cre
$ sqlplus testdb1/hvr < hvr_demo01.mod
```

For the target, create two test schemas, again each containing two empty tables named **dm01_product** and **dm01_order**.

```
$ sqlplus system/manager
 SQL> create user testdb2 identified by hvr
 2  default tablespace users
 3  temporary tablespace temp
 4  quota unlimited on users;

SQL> create user testdb3 identified by hvr
```

```
 2  default tablespace users
 3  temporary tablespace temp
 4  quota unlimited on users;
SQL> grant create session to testdb2;
SQL> grant create table to testdb2;
SQL> grant create sequence to testdb2;
SQL> grant create procedure to testdb2;
SQL> grant create trigger to testdb2;
SQL> grant create view to testdb2;
SQL> grant execute any procedure to testdb2;
```

Give the same grants to **testdb3**.

You can either create the tables using HVRs .cre and .mod scripts like above or let HVR create them during initial loading (**HVR Refresh** with **Create Absent Tables**).

```
Create the test tables using HVRs scripts:.

$ cd $HVR_HOME/demo/hvr_demo01/base/oracle
$ sqlplus testdb2/hvr < hvr_demo01.cre
$ sqlplus testdb2/hvr < hvr_demo01.mod
$ sqlplus testdb3/hvr < hvr_demo01.cre
$ sqlplus testdb3/hvr < hvr_demo01.mod
```

Install HVR

First read section Introduction which explains the HVR's terminology and architecture. In particular this explains the importance of a hub database.
Then install the HVR software on the hub machine by following the installation steps in section New Installation on Unix or Linux or New Installation on Windows. If the hub machine is a Unix machine then HVR can either be installed on a Windows PC (so the HVR GUI can run on the PC and connect to the Unix hub machine) or the HVR GUI can be run on the Unix hub machine and connect back to an X server running on a PC.
This Quickstart assumes the Oracle Database on the hub server is also the source database. Most real-time integration scenarios use log-based capture (parameter **/LogBased**). To enable log-base capture, configure the following:

- The user name that HVR uses must be in Oracle's group. On Unix and Linux this can be done by adding the user name used by HVR to the line in **/etc/group** that begins with **dba**. On Windows right–click **My Computer** and select **Manage ▶ Local Users and Groups ▶ Groups ▶ ora_dba ▶ Add to Group ▶ Add**.
- The Oracle instance should have archiving 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 query **select destination, status from v$archive_dest.** By default, this will return values **USE_DB_RECOVERY_FILE_DEST, VALID**, which is inside the flashback recovery area. Alternatively, an archive destination can be defined with the following statement: **alter system set log_archive_dest_1='location=/disk1/arc'** and then restart the instance.

Create the Hub Database

Create the hub database, in which the HVR GUI will store the channel definition. This is actually another user/schema in the Oracle instance.

```
$ sqlplus system/manager
SQL> create user hvrhub identified by hvr
 2  default tablespace users
 3  temporary tablespace temp
 4  quota unlimited on users;

SQL> grant create session to hvrhub;
SQL> grant create table to hvrhub;
SQL> grant create sequence to hvrhub;
SQL> grant create procedure to hvrhub;
SQL> grant create trigger to hvrhub;
SQL> grant create view to hvrhub;
SQL> grant execute any procedure to hvrhub;
```

DBMS Alert Grant

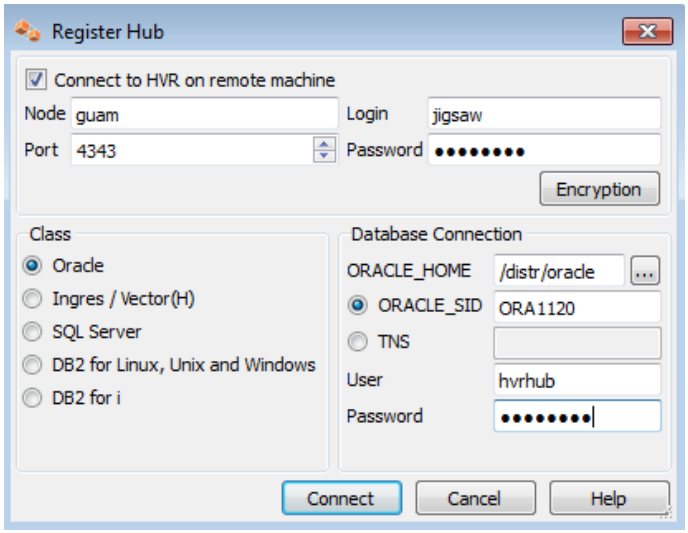
If trigger–based capture will be used (when **/LogBased** is not defined), then execution rights for **dbms_alert** must also be granted to the schema that is the source location of replication (i.e. **testdb1**). Login as **oracle** and perform the following extra statements:

```
$ sqlplus
Enter user-name: / as sysdba
SQL> grant execute on dbms_alert to testdb1;
SQL> exit;
```

Connect To Hub Database

Start the HVR GUI on a PC by clicking on the HVR GUI icon (this is created by the HVR Installer for Windows) or by running **hvrgui** on Linux.

First, Register the hub database: right–click on **hub machines** ▶ **Register hub**. Enter connection details.

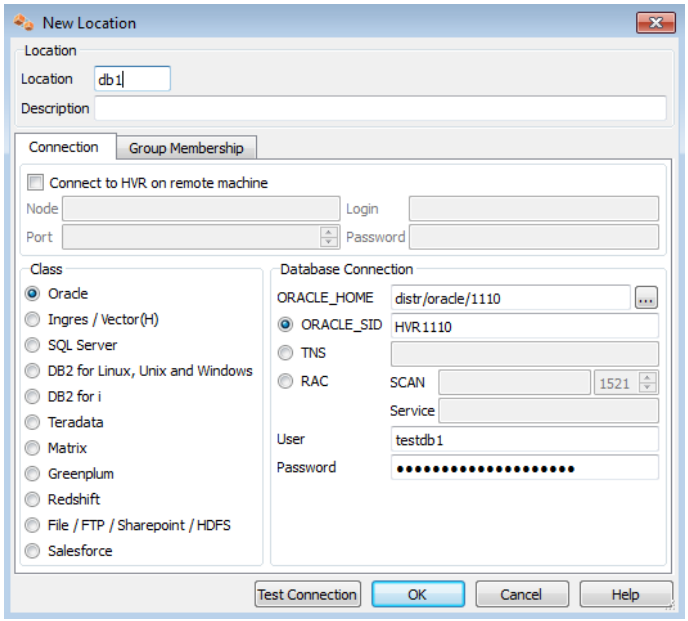


In this example the hub is a machine called **guam**, where an INET daemon is listening on port 4343. See section New Installation on Unix or Linux for how to configure this.

For a new hub database a dialog will prompt **Do you wish to create the catalogs?**; answer **Yes**.

Create Oracle Locations

Next create three locations (one for each test database) using right–click on **Location Configuration** ▶ **New Location**.



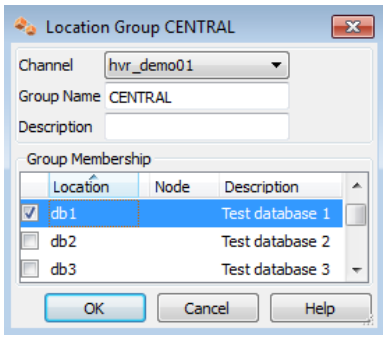
In this example there is no need to check **Connect to HVR on remote machine** because **testdb1** is on the same machine as the hub.
Ignore the **Group Membership** tab for now.
Make locations for **testdb2** and **testdb3** too.

Create a Channel

The next step is to create a channel. For a relational database the channel represents a group of tables that is captured as unit. Create a channel using right–click on **Channel Definitions** ▶ **New Channel**.
Choose any name you like.

Create Location Groups

The channel needs two location groups. Under the new channel: right–click on **Location Groups** ▶ **New Group**. Enter a group name (for instance **CENTRAL**).



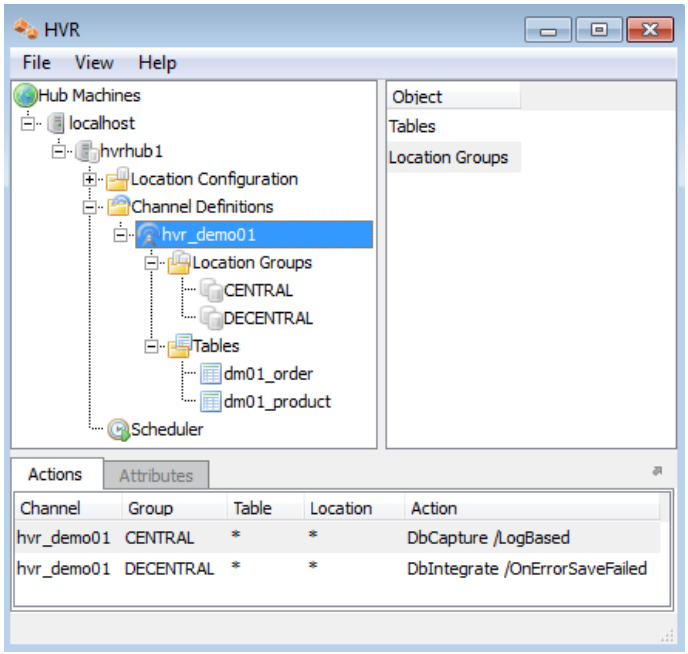
Add location **db1** as a member of this group by checking the box for **db1**.
Then create a second location group, called **DECENTRAL** that has members **db2** and **db3**.
The new channel also needs a list of tables to replicate. This can be done as follows; right–click on **Tables** ▶ **Table Select**.

- Choose the first of the three locations ▶ **Connect**.
- In the **Table Selection** window, click on both tables and click **Add**.
- In new dialog **HVR Table Name** click **OK**.
- Close the **Table Selection** window.
- Perform table select again on one of the other locations and confirm that all tables to be replicated have value **Same** in column **Match**.

Define Actions

The new channel needs two actions to indicate the direction of replication.

- Right–click on group **CENTRAL ▶ New Action ▶ DbCapture**. Check **/LogBased** so that the channel will detect changes using the log file, instead of using database triggers.
- Right–click on **Group DECENTRAL ▶ New Action ▶ DbIntegrate**. Check **/OnErrorSaveFailed**, this affects how replication errors are handled.

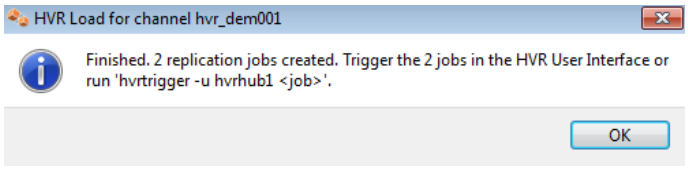


Note that the Actions pane only displays actions related to the objects selected in the left–hand pane. So click on channel **hvr_demo01** to see both actions.

Enable Replication with HVR Load

Now that the channel definition is complete, create the runtime replication system.

Right–click on channel **hvr_demo01 ▶ HVR Load**. Choose **Create or Replace Objects** and click **HVR Load**.

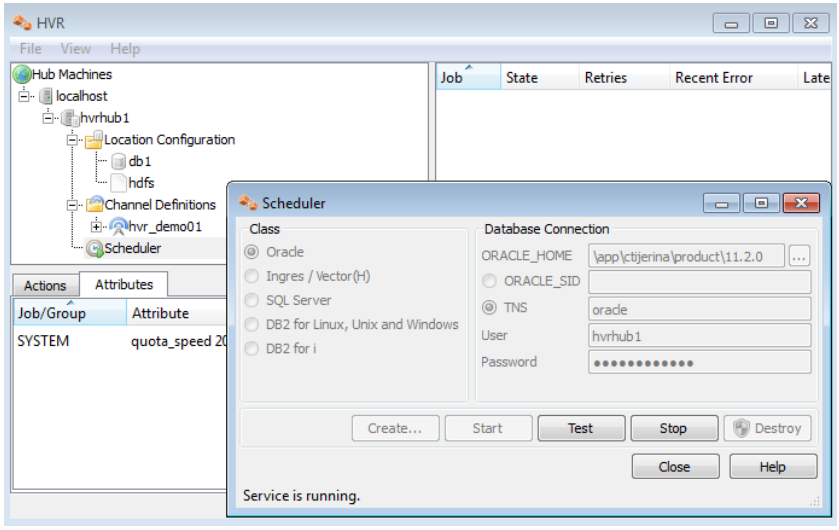


From the moment that HVR Load is done, all changes to database **testdb1** will be captured by HVR when its capture job looks inside the logging.

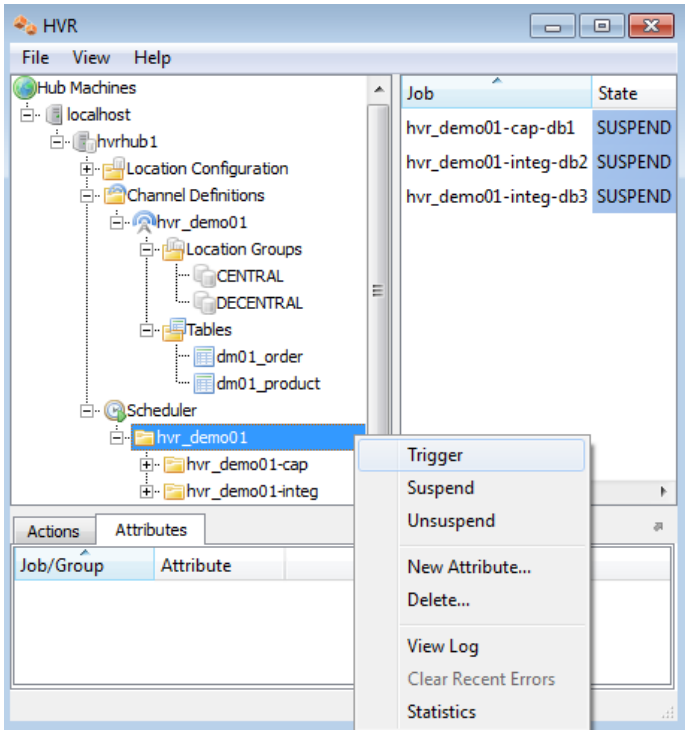
HVR Load also creates three replication jobs, which can be seen under the Scheduler node in the GUI.

Start Scheduling of Replication Jobs

Start the Scheduler on the hub machine by clicking in the HVR GUI on the **Scheduler** node of the hub database.



Next, instruct the HVR Scheduler to trigger the replication jobs.



The replication jobs inside the Scheduler each execute a script under **$HVR_CONFIG/job/hvrhub/hvr_demo01** that has the same name as the job. So job **hvr_demo01–cap–db1** detects changes on database **testdb1** and stores these as transactions files on the hub machine.

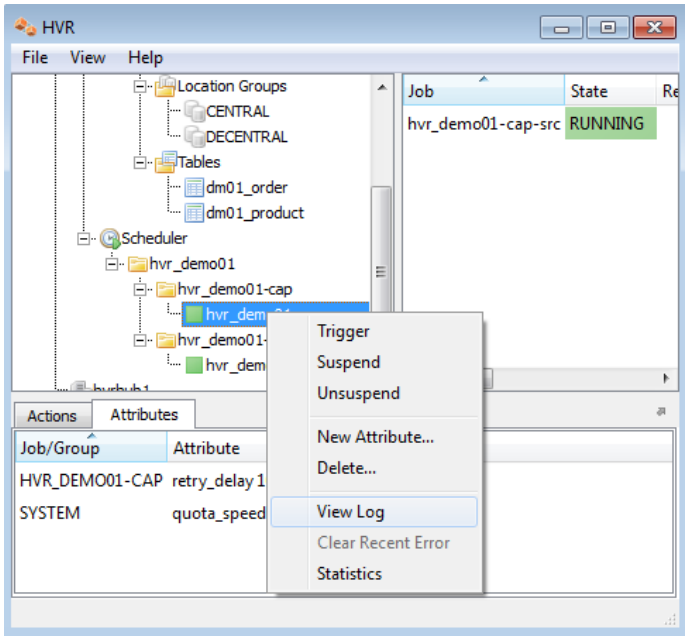
The other two jobs (**hvr_demo01–integ–db2** and **hvr_demo01–integ–db3**) pick up these transaction files and perform inserts, updates and deletes on the two target databases.

Test Replication

To test replication, make a change in **testdb1**:

```
testdb1/hvr
SQL> insert into dm01_product values (1, 19.99, 'DVD');
SQL> commit;
```

In the HVR log file you can see the output of the jobs by clicking on **View Log**. This log file can be found in **$HVR_CONFIG/log/*hubdb*/hvr_demo01–cap–db1**.



The job output looks like this:

```
hvr_demo01-cap-db1: Scanned 1 transaction containing 1 row (1 ins) for 1 table.
hvr_demo01-cap-db1: Routed 215 bytes (compression=40.6%) from 'db1' into 2 locations.
hvr_demo01-cap-db1: Capture cycle 3.
hvr_demo01-integ-db2: Integrate cycle 2 for 1 transaction file (215 bytes).
hvr_demo01-integ-db2: Integrated 1 change from 'dm01_product' (1 ins).
hvr_demo01-integ-db2: Integrate used 1 transaction and took 0.004 seconds.
hvr_demo01-integ-db3: Integrate cycle 2 for 1 transaction file (215 bytes).
hvr_demo01-integ-db3: Integrated 1 change from 'dm01_product' (1 ins).
hvr_demo01-integ-db3: Integrate used 1 transaction and took 0.013 seconds.
hvr_demo01-integ-db3: Waiting...
```

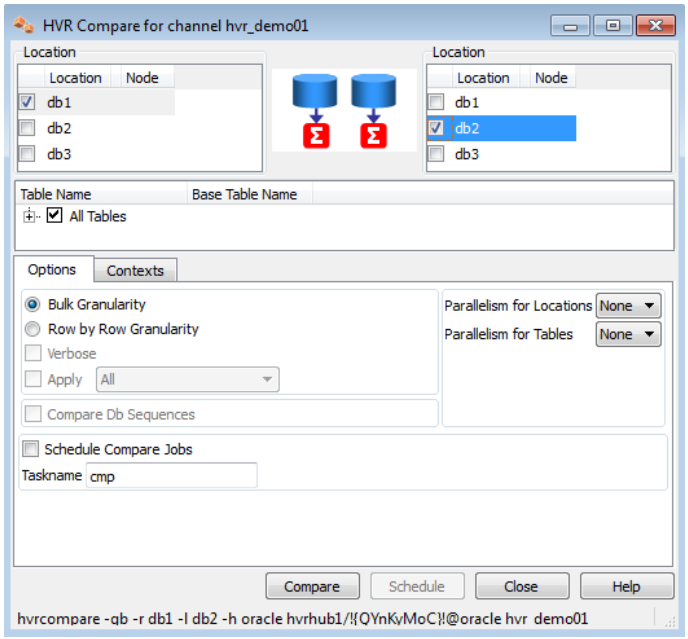
This indicates that the jobs replicated the original change to **testdb2** and **testdb3**. A query on **testdb2** confirms this:

```
testdb2/hvr
SQL> select * from dm01_product;
```

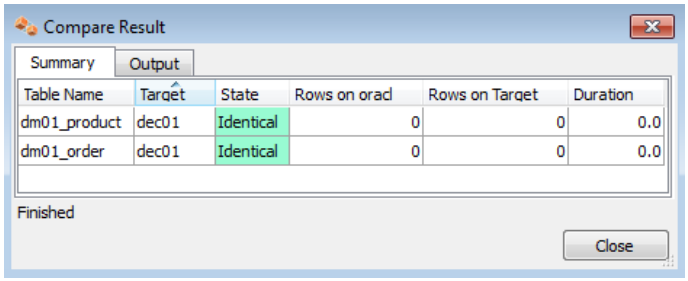
prod_id	prod_price	prod_descrip
1	19.99	DVD

HVR Compare and Refresh

HVR Compare checks whether two locations have identical rows, and HVR Refresh copies the content of one location to the second location. In the HVR GUI, right–click on a channel ▶ **HVR Compare** (or **HVR Refresh**). Choose two locations by clicking on the **Select** buttons.



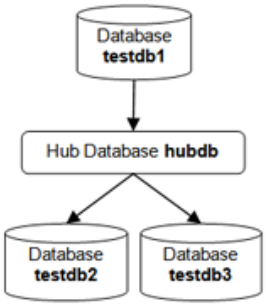
The outcome of the comparison is displayed below;



B

Quick Start for HVR on Ingres

This appendix shows how to set up an HVR channel (called **hvr_demo01**) to replicate between Ingres databases. The steps actually start by creating new databases and tables for HVR to replicate between. In real life these databases would already exist and be filled with the user tables, but for simplicity everything is created from scratch. Likewise these databases would normally be on different machines, but again for simplicity everything is just kept on the hub machine.



Create Test Databases and Tables

Create three test databases, each containing two empty tables named **dm01_product** and **dm01_order**. If replication is configured between existing databases and tables then this step should be skipped.

```
$ createdb testdb1
$ createdb testdb2
$ createdb testdb3

$ cd $HVR_HOME/demo/hvr_demo01/base/ingres
$ sql testdb1 < hvr_demo01.cre
$ sql testdb1 < hvr_demo01.mod
$ sql testdb2 < hvr_demo01.cre
$ sql testdb2 < hvr_demo01.mod
$ sql testdb3 < hvr_demo01.cre
$ sql testdb3 < hvr_demo01.mod

$ ckpdb +j testdb1
```

The last command (to checkpoint the source database) is needed so that if HVR log–based capture cannot find a change anymore in the Ingres log file it can always look into the Ingres journal files.

Install HVR

First read section Introduction which explains the HVR's terminology and architecture. In particular this explains the importance of a hub database.
Then install the HVR software on the hub machine by following the installation steps in section New Installation on Unix or Linux or New Installation on Windows. If the hub machine is a Unix machine then HVR can either be installed on a Windows PC (so the HVR GUI can run on the PC and connect to the Unix hub machine) or the HVR GUI can be run on the Unix hub machine and connect back to an X server running on a PC.

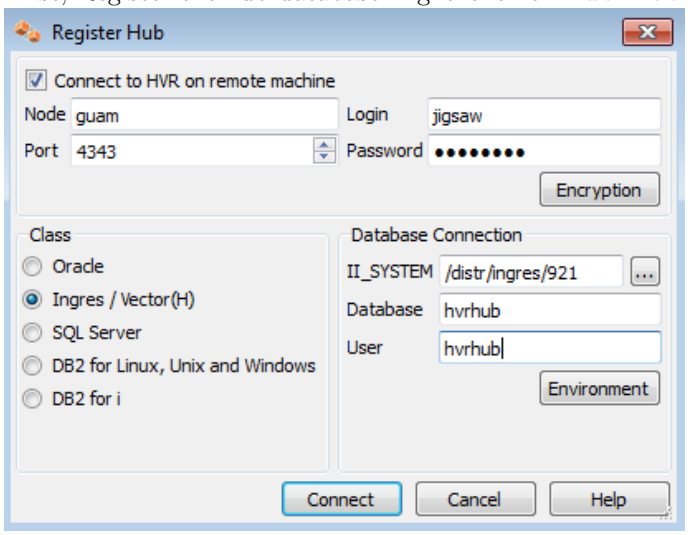
Create the Hub Database

Create the hub database, in which the HVR GUI will store the channel definition.

```
$ createdb hvrhub
```

Connect to Hub Database

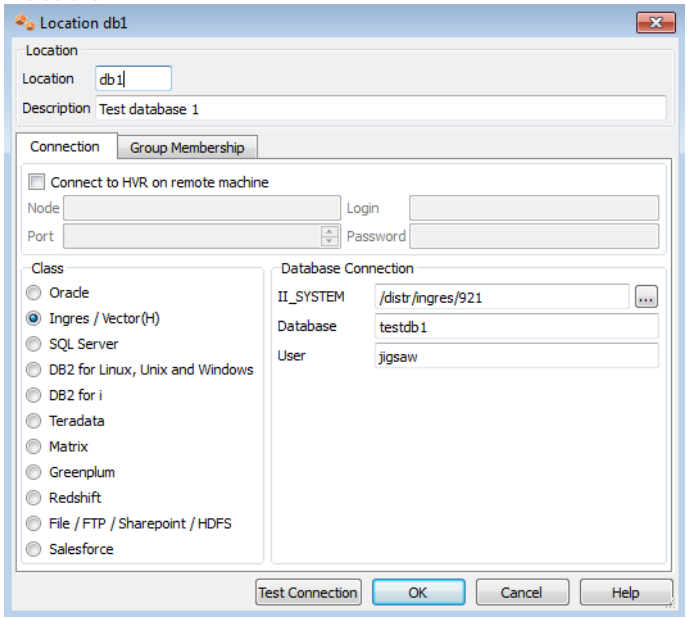
Start the HVR GUI on a PC by clicking on the HVR GUI icon (this is created by the HVR Installer for Windows) or by running **hvrgui** on Linux.
First, Register the hub database: right–click on **hub machines** ▶ **Register hub**. Enter connection details.



In this example the hub is a machine called **guam**, where an INET daemon is listening on port 4343. See section New Installation on Unix or Linux for how to configure this.
For a new hub database a dialog will prompt **Do you wish to create the catalogs?**; answer **Yes**.

Create Ingres Locations

Next create three locations (one for each test database) using right–click on **Location Configuration** ▶ **New Location**.

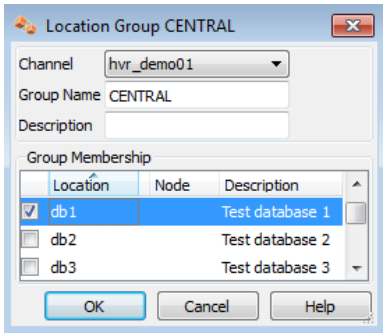


In this example there is no need to check **Connect to HVR on remote machine** because **testdb1** is on the same machine as the hub.
Ignore the **Group Membership** tab for now.
Make locations for **testdb2** and **testdb3** too.

Now define a channel using **Channel Definitions ▶ New Channel**.

Create Location Groups

The channel needs two location groups. Under the new channel: right–click on **Location Groups ▶ New Group**. Enter a group name (for instance **CENTRAL**).



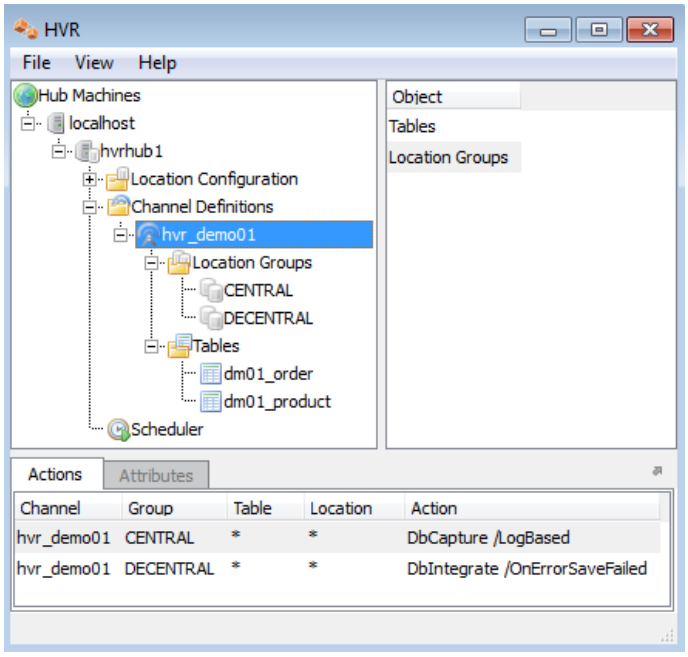
Add location **db1** as a member of this group by checking the box for **db1**.
Then create a second location group, called **DECENTRAL** that has members **db2** and **db3**.
The new channel also needs a list of tables to replicate. This can be done as follows; right–click on **Tables ▶ Table Select**.

- Choose the first of the three locations ▶ **Connect**.
- In the **Table Selection** window, click on both tables and click **Add**.
- In new dialog **HVR Table Name** click **OK**.
- Close the **Table Selection** window.
- Perform table select again on one of the other locations and confirm that all tables to be replicated have value **Same** in column **Match**.

Define Actions

The new channel needs two actions to indicate the direction of replication.

- Right–click on group **CENTRAL ▶ New Action ▶ DbCapture**. Check **/LogBased** so that the channel will detect changes using the log file, instead of using database triggers.
- Right–click on **Group DECENTRAL ▶ New Action ▶ DbIntegrate**. Check **/OnErrorSaveFailed**, this affects how replication errors are handled.

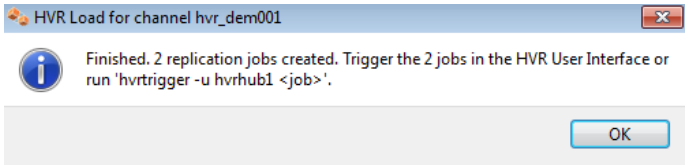


Note that the Actions pane only displays actions related to the objects selected in the left–hand pane. So click on channel **hvr_demo01** to see both actions.

Enable Replication with HVR Load

Now that the channel definition is complete, create the runtime replication system.

Right–click on channel **hvr_demo01** ▶ **HVR Load**. Choose **Create or Replace Objects** and click **HVR Load**.

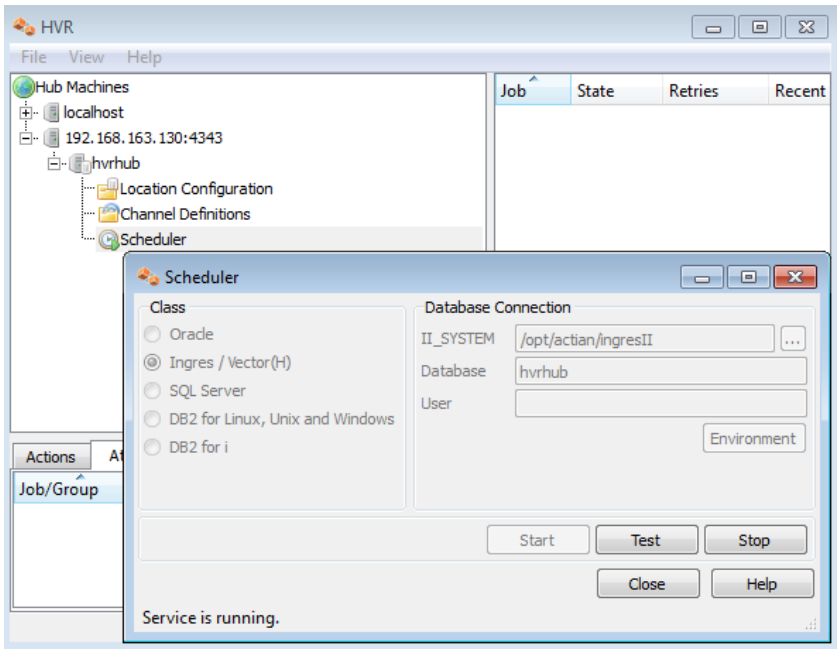


From the moment that HVR Load is done, all changes to database **testdb1** will be captured by HVR when its capture job looks inside the logging.

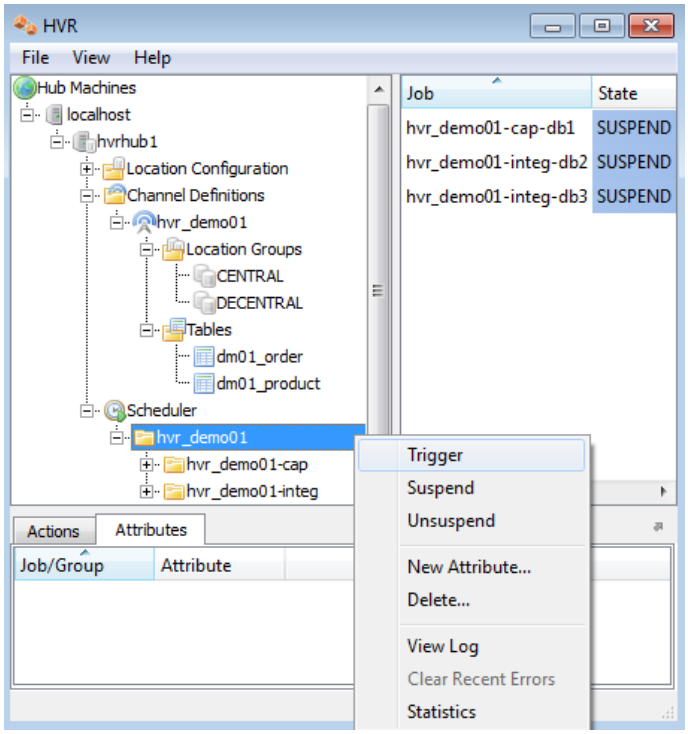
HVR Load also creates three replication jobs, which can be seen under the Scheduler node in the GUI.

Start Scheduling of Replication Jobs

Start the Scheduler on the hub machine by clicking in the HVR GUI on the **Scheduler** node of the hub database.



Next, instruct the HVR Scheduler to trigger the replication jobs. The replication jobs inside the Scheduler each execute a script under **$HVR_CONFIG/job/hvrhub/hvr_demo01** that has the same name as the job. So job **hvr_demo01–cap–db1** detects changes on database **testdb1** and stores these as transactions files on the hub machine.



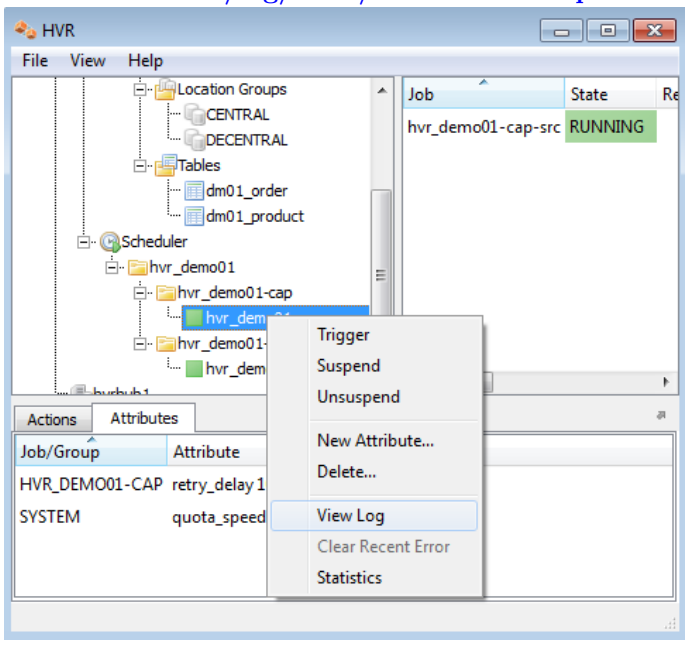
The other two jobs (**hvr_demo01–integ–db2** and **hvr_demo01–integ–db3**) pick up these transaction files and perform inserts, updates and deletes on the two target database.

Test Replication

To test replication, make a change in **testdb1**:

```
testdb1/hvr
SQL> insert into dm01_product values (1, 19.99, 'DVD');
SQL> commit;
```

In the HVR log file you can see the output of the jobs by clicking on **View Log**. This log file can be found in **$HVR_CONFIG/log/*hubdb*/hvr_demo01–cap–db1**.



The job output looks like this:

```
hvr_demo01-cap-db1: Scanned 1 transaction containing 1 row (1 ins) for 1 table.
hvr_demo01-cap-db1: Routed 215 bytes (compression=40.6%) from 'db1' into 2 locations.
hvr_demo01-cap-db1: Capture cycle 3.
```

```
hvr_demo01-integ-db2: Integrate cycle 2 for 1 transaction file (215 bytes).
hvr_demo01-integ-db2: Integrated 1 change from 'dm01_product' (1 ins).
hvr_demo01-integ-db2: Integrate used 1 transaction and took 0.004 seconds.
hvr_demo01-integ-db3: Integrate cycle 2 for 1 transaction file (215 bytes).
hvr_demo01-integ-db3: Integrated 1 change from 'dm01_product' (1 ins).
hvr_demo01-integ-db3: Integrate used 1 transaction and took 0.013 seconds.
hvr_demo01-integ-db3: Waiting...
```

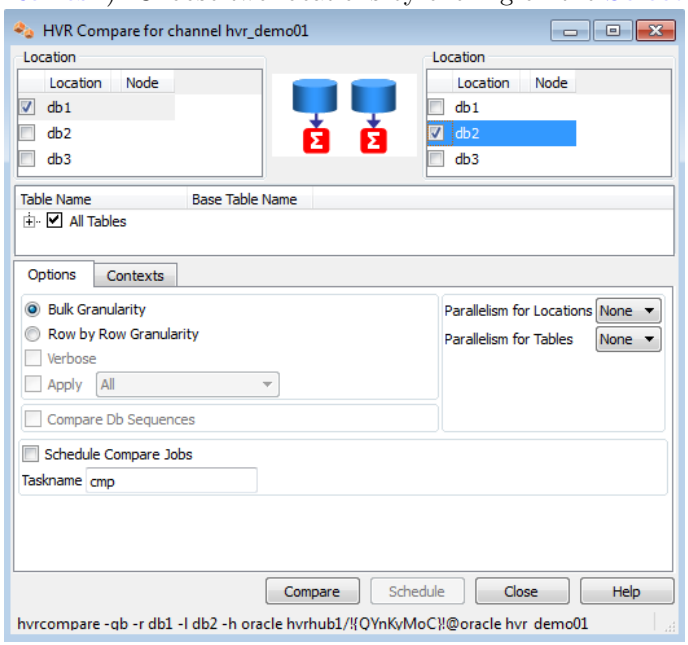
This indicates that the jobs replicated the original change to **testdb2** and **testdb3**. A query on **testdb2** confirms this:

```
testdb2/hvr
SQL> select * from dm01_product;
```

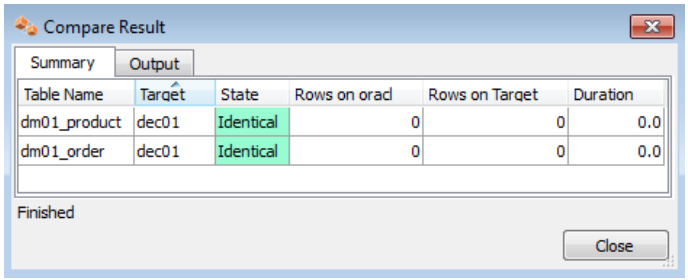
prod_id	prod_price	prod_descrip
1	19.99	DVD

HVR Compare and Refresh

HVR Compare checks whether two locations have identical rows, and HVR Refresh copies the content of one location to the second location. In the HVR GUI, right–click on a channel ▶ **HVR Compare** (or **HVR Refresh**). Choose two locations by clicking on the **Select** buttons.



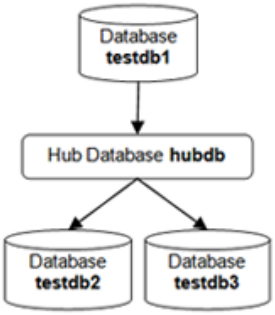
The outcome of the comparison is displayed below;



C

Quick Start for HVR on SQL Server

This appendix shows how to set up an HVR channel (called **hvr_demo01**) to replicate between SQL Server databases. The steps actually start by creating new databases and tables for HVR to replicate between. In real life these databases would already exist and be filled with the user tables, but for simplicity everything is created from scratch. Likewise these databases would normally be on different machines, but again for simplicity everything is just kept on the hub machine.



Create Test Databases and Tables

Generally when getting started with HVR a source schema with tables and data already exists. If so then this step can be skipped.
This Quickstart uses two empty tables named dm01_product and dm01_order. In an existing SQL server database, create a test schema and create the tables using the following commands.

In SQL Server Management Studio, create database **testdb1**. Next, create the test tables;

```
C:\> cd %HVR_HOME%\demo\hvr_demo01\base\sqlserver
C:\> osql -U hvr -P hvr -d testdb1 < hvr_demo01.cre
C:\> osql -U hvr -P hvr -d testdb1 < hvr_demo01.mod
```

For the target, create two test databases, each containing two empty tables named **dm01_product** and **dm01_order**.
In SQL Server Management Studio, create databases **testdb2** and **testdb3**. You can either create the tables using HVRs scripts or let HVR create them during initial loading (**HVR Refresh** with **Create Absent Tables**).
Create the test tables using HVRs script:

```
C:\> cd %HVR_HOME%\demo\hvr_demo01\base\sqlserver
C:\> osql -U hvr -P hvr -d testdb2 < hvr_demo01.cre
C:\> osql -U hvr -P hvr -d testdb2 < hvr_demo01.mod
C:\> osql -U hvr -P hvr -d testdb3 < hvr_demo01.cre
C:\> osql -U hvr -P hvr -d testdb3 < hvr_demo01.mod
```

Install HVR

First read section Introduction which explains the HVR's terminology and architecture. In particular this explains the importance of a hub database.

Then install the HVR software on the hub machine by performing the steps in section New Installation on Windows.

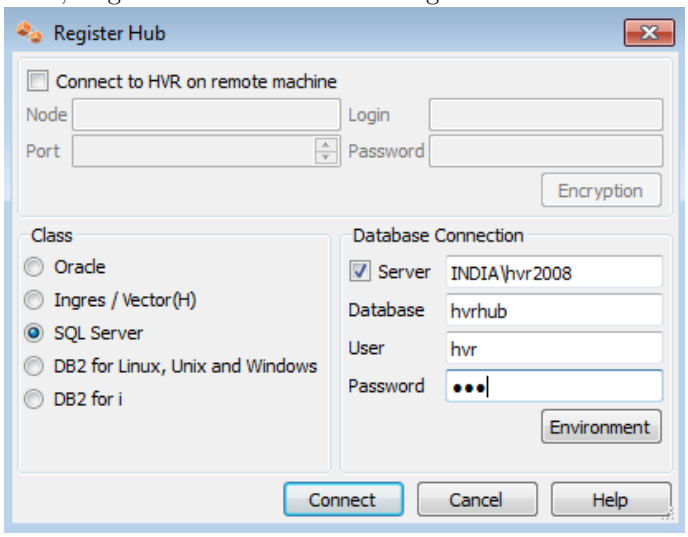
Follow the steps in Requirements for Microsoft SQL Server for enabling (log-based) capture on SQL Server

Create the Hub Database

In SQL Server Management Studio, create the hub database (e.g. **hvrhub**) to store the channel definition.

Connect to Hub Database

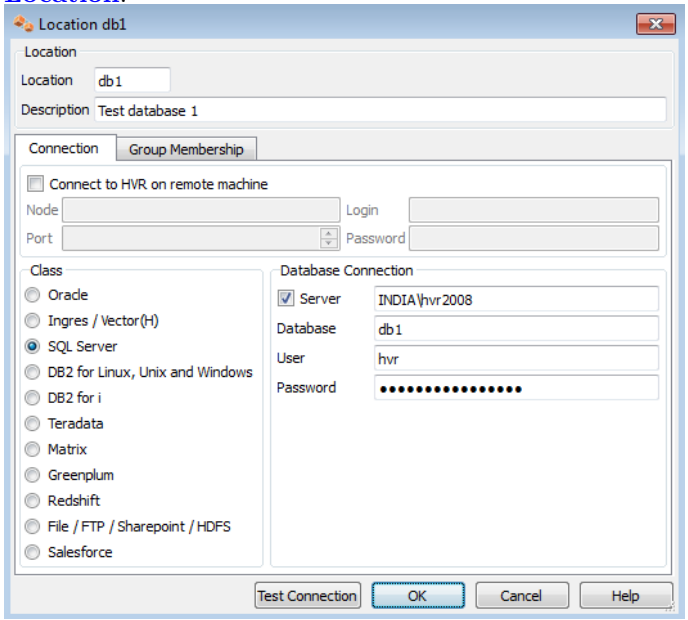
Start the HVR GUI on the hub machine by clicking on the HVR GUI icon.
First, Register the hub database: Right–click on **hub machines** ▶ **Register hub**. Enter connection details.



For a new hub database a dialog will prompt: **Do you wish to create the catalogs?** Answer **Yes**.

Create SQL Server Locations

Next create three locations (one for each test database) using right–click on **Location Configuration** ▶ **New Location**.

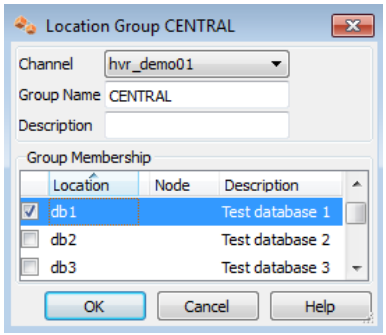


In this example there is no need to check **Connect to HVR on remote machine** because **testdb1** is on the same machine as the hub.
Ignore the **Group Membership** tab for now.
Make locations for **testdb2** and **testdb3** too.

Now define a channel using **Channel Definitions ▶ New Channel**.

Create Location Groups

The channel needs two location groups. Under the new channel: right–click on **Location Groups ▶ New Group**. Enter a group name (for instance **CENTRAL**).



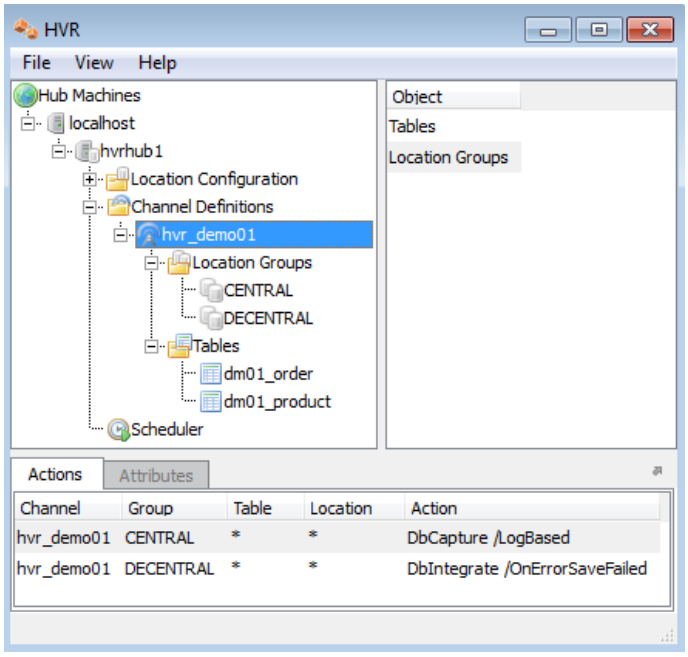
Add location **db1** as a member of this group by checking the box for **db1**.
Then create a second location group, called **DECENTRAL** that has members **db2** and **db3**.
The new channel also needs a list of tables to replicate. This can be done as follows; right–click on **Tables ▶ Table Select**.

- Choose the first of the three locations ▶ **Connect**.
- In the **Table Selection** window, click on both tables and click **Add**.
- In new dialog **HVR Table Name** click **OK**.
- Close the **Table Selection** window.
- Perform table select again on one of the other locations and confirm that all tables to be replicated have value **Same** in column **Match**.

Define Actions

The new channel needs two actions to indicate the direction of replication.

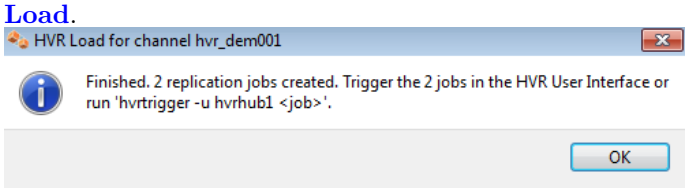
- Right–click on group **CENTRAL ▶ New Action ▶ DbCapture**. Check **/LogBased** so that the channel will detect changes using the log file, instead of using database triggers.
- Right–click on **Group DECENTRAL ▶ New Action ▶ DbIntegrate**. Check **/OnErrorSaveFailed**, this affects how replication errors are handled.



Note that the Actions pane only displays actions related to the objects selected in the left–hand pane. So click on channel **hvr_demo01** to see both actions.

Enable Replication with HVR Load

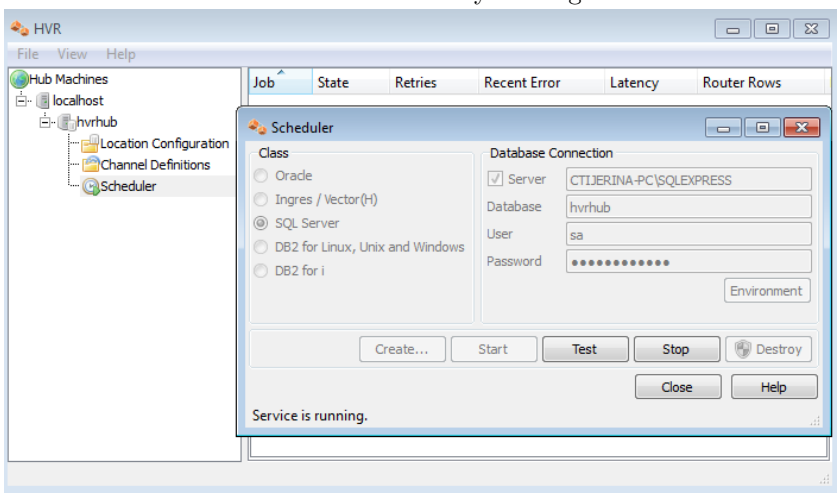
Now that the channel definition is complete, create the runtime replication system.
Right–click on channel **hvr_demo01** ▶ **HVR Load**. Choose **Create or Replace Objects** and click **HVR Load**.



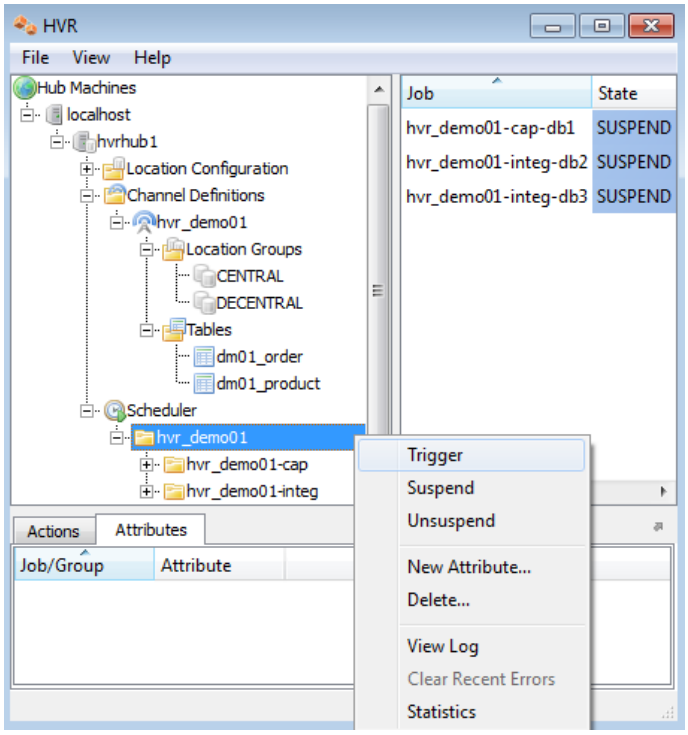
From the moment that HVR Load is done, all changes to database **testdb1** will be captured by HVR when its capture job looks inside the logging.
HVR Load also creates three replication jobs, which can be seen under the Scheduler node in the GUI.

Start Scheduling of Replication Jobs

Start the Scheduler on the hub machine by clicking in the HVR GUI on the **Scheduler** node of the hub database.



Next, instruct the HVR Scheduler to trigger the replication jobs.



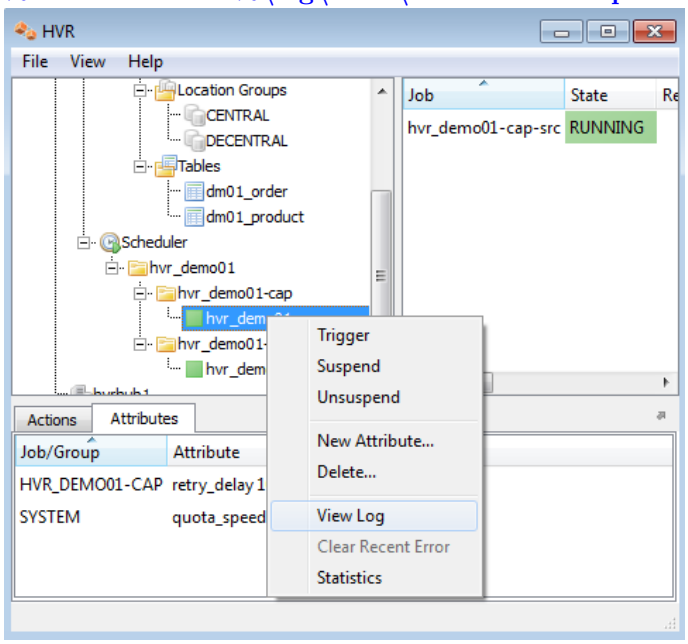
The replication jobs inside the Scheduler each execute a script under **$HVR_CONFIG/job/hvrhub/hvr_demo01** that has the same name as the job. So job **hvr_demo01–cap–db1** detects changes on database **testdb1** and stores these as transactions files on the hub machine. The other two jobs (**hvr_demo01–integ–db2** and **hvr_demo01–integ–db3**) pick up these transaction files and perform inserts, updates and deletes on the two target databases.

Test Replication

To test replication, make a change in **testdb1**:

```
SQL> insert into dm01_product values (1, 19.99, 'DVD');
```

In the HVR log file you can see the output of the jobs by clicking on **View Log**. This log file can be found in **%HVR_CONFIG%\log***hubdb***\hvr_demo01–cap–db1**.



The job output looks like this:

```
C:\> notepad %HVR_CONFIG%\log\hvrhub\hvr.out

hvr_demo01-cap-db1: Capture cycle 1.
hvr_demo01-cap-db1: Selected 1 row from 'dm01_product__c0' (201 wide).
hvr_demo01-cap-db1: Routed 212 bytes (compression=42.7%) from 'db1' into \
2 locations.
hvr_demo01-cap-db1: Finished. (elapsed=1.20s)
hvr_demo01-integ-db2: Integrate cycle 1 for 1 transaction file (212 bytes).
hvr_demo01-integ-db2: Integrated 1 change from 'dm01_product' (1 ins).
hvr_demo01-integ-db2: Integrate used 1 transaction and took 0.017 seconds.
hvr_demo01-integ-db2: Finished. (elapsed=0.14s)
hvr_demo01-integ-db3: Integrate cycle 1 for 1 transaction file (212 bytes).
hvr_demo01-integ-db3: Integrated 1 change from 'dm01_product' (1 ins).
hvr_demo01-integ-db3: Integrate used 1 transaction and took 0.02 seconds.
hvr_demo01-integ-db3: Finished. (elapsed=0.15s)
```

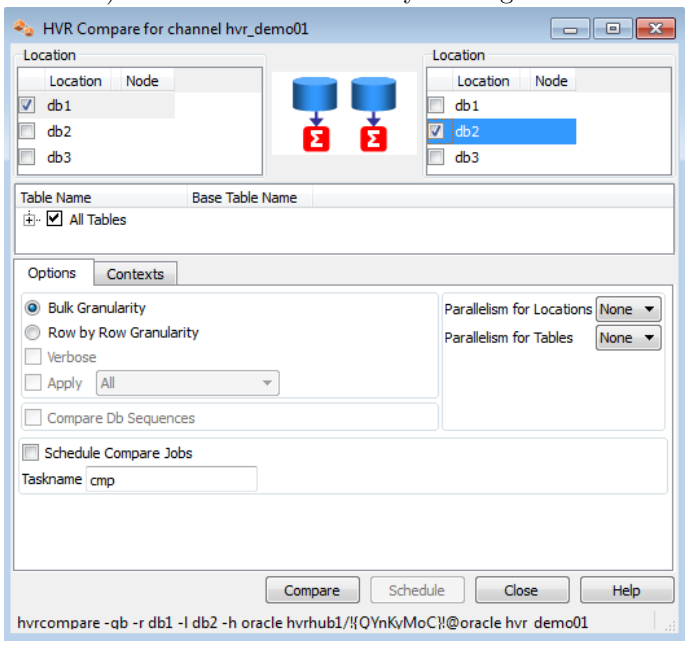
This indicates that the jobs replicated the original change to **testdb2** and **testdb3**. A query on **testdb2** confirms this:

```
SQL> select * from dm01_product;
```

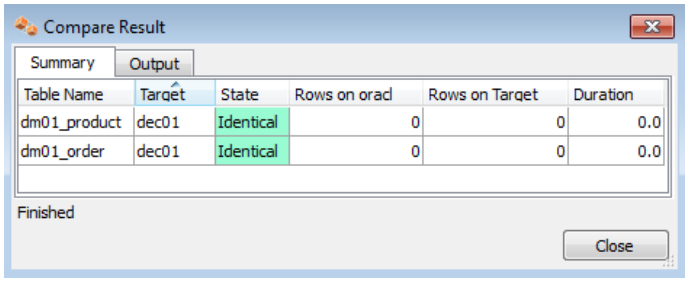
prod_id	prod_price	prod_descrip
1	19.99	DVD

HVR Compare and Refresh

HVR Compare checks whether two locations have identical rows, and HVR Refresh copies the content of one location to the second location. In the HVR GUI, right–click on a channel ▶ **HVR Compare** (or **HVR Refresh**). Choose two locations by clicking on the **Select** buttons.



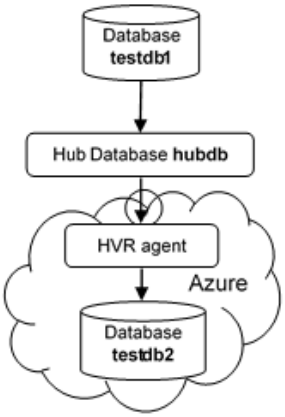
The outcome of the comparison is displayed below;



D

Quick Start for HVR into Azure SQL

This appendix shows how to set up an HVR channel (called **hvr_demo01**) to replicate between a local SQL Server database and an Azure SQL server database residing in the Azure cloud. To create the Azure components, the **HVR for Azure Image** will be used. The steps actually start by creating new databases and tables for HVR to replicate between. In real life these databases would already exist and be filled with the user tables, but for simplicity everything is created from scratch. Also, for simplicity, we will assume the source database resides on the hub as well and SQL server has already been installed there.



Create Test Databases and Tables

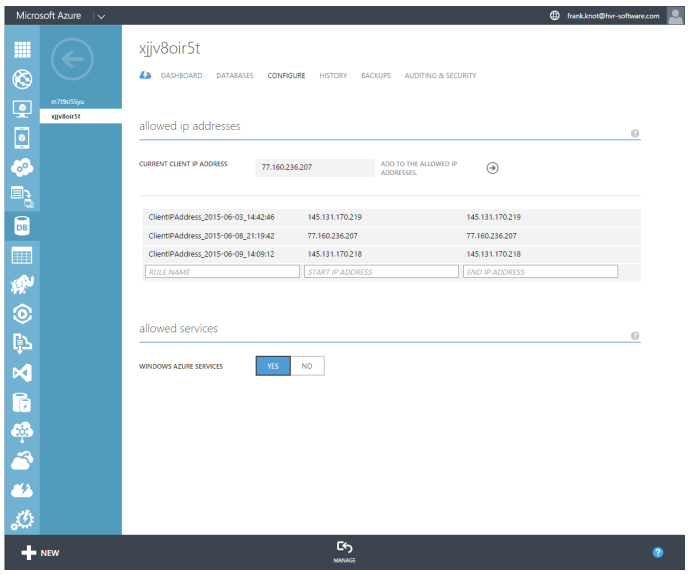
Generally when getting started with HVR a source schema with tables and data already exists. If so then this step can be skipped.

This Quickstart uses two empty tables named dm01_product and dm01_order. In an existing SQL server database, create a test schema and create the tables using the following commands.

In SQL Server Management Studio, create database **testdb1**. Next, create the test tables;

```
C:\> cd %HVR_HOME%\demo\hvr_demo01\base\sqlserver
C:\> osql -U hvr -P hvr -d testdb1 < hvr_demo01.cre
C:\> osql -U hvr -P hvr -d testdb1 < hvr_demo01.mod
```

In the Azure portal, create an SQL database using **New -> Data Service -> SQL Database -> Quick_Create** called **testdb2**. Modify the database server configuration by enabling access from Azure services:



Create the tables either using HVR's script as shown earlier or use HVR to create the tables during the initial load (**HVR Refresh** with **Create Absent Tables**). Check **With key** as Azure SQL requires tables to have a primary key.

Install HVR on-premise

First read section Introduction which explains the HVR's terminology and architecture. In particular this explains the importance of a hub database.

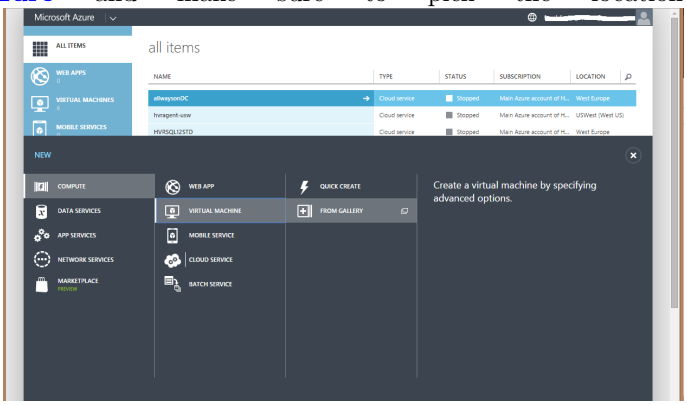
Then install the HVR software on the hub machine by performing the steps in section New Installation on Windows.

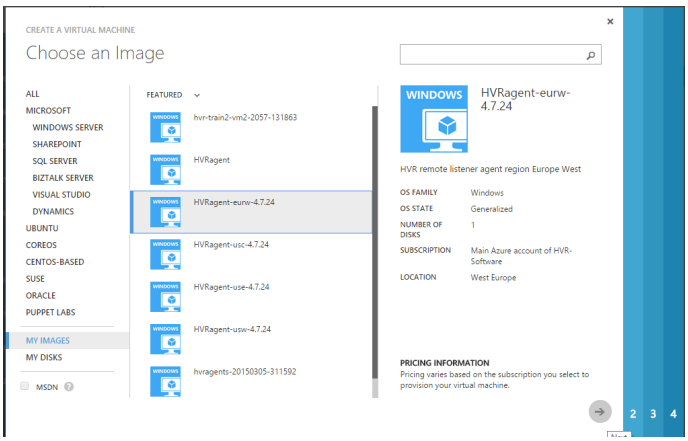
Follow the steps in Requirements for Microsoft SQL Server for enabling (log-based) capture on SQL Server

Install HVR remote listener agent on Azure VM

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

1. In the Azure console, choose **+/Compute/Virtual Machine/From Gallery**. Select the **HVR Image for Azure** and make sure to pick the location where the database resides:





2. Create an HVR agent VM in the same Region as the database or HDInsight environment. A standard tier **A1 instance** VM is sufficient to run HVR's agent.
3. HVR uses port 4343 for incoming connections to the HVR listener agent(**endpoint**). In the creation process open this port for incoming traffic:
4. When the VM status is moved from 'provisioning" to "running", the agent is ready for use. Test the agent by entering its credentials in an (on-premises) hub:

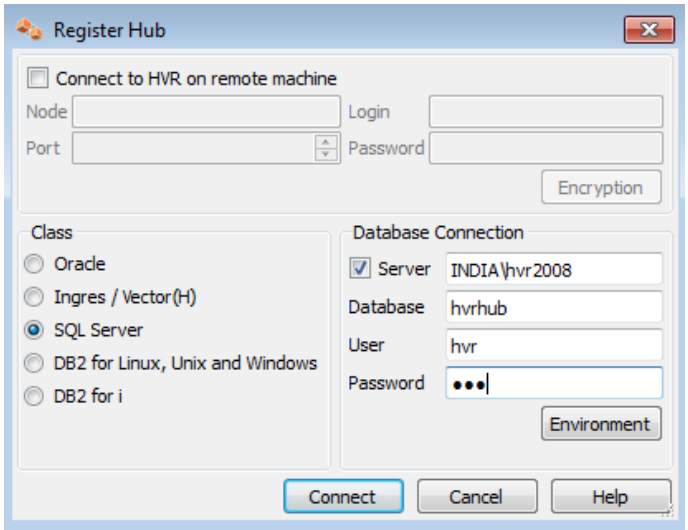
If you don't have the **HVR for Azure Image** available, your can obtain it by following the steps in New Installation of HVR Image for Azure get Image or do the install manually: Install HVR Agent on Azure VM

Create the Hub Database

In SQL Server Management Studio, create the hub database (e.g. **hvrhub**) to store the channel definition.

Connect to Hub Database

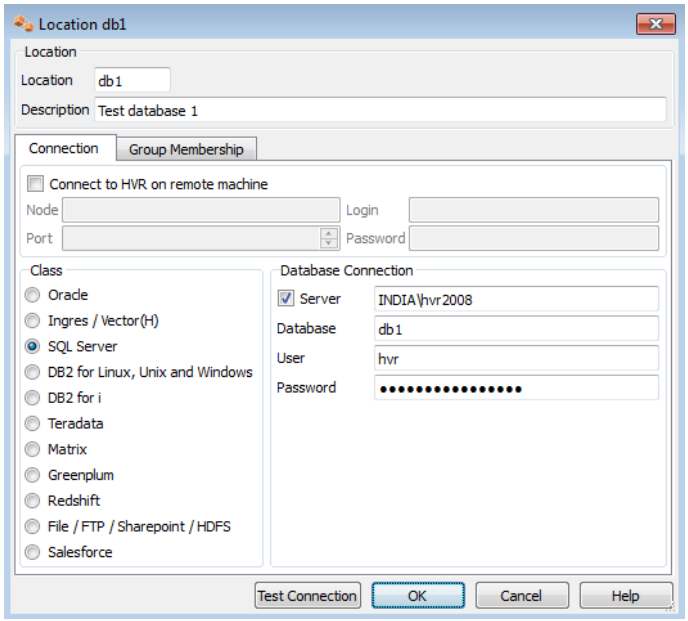
Start the HVR GUI on the hub machine by clicking on the HVR GUI icon.
First, Register the hub database: Right–click on **hub machines** ▶ **Register hub**. Enter connection details.



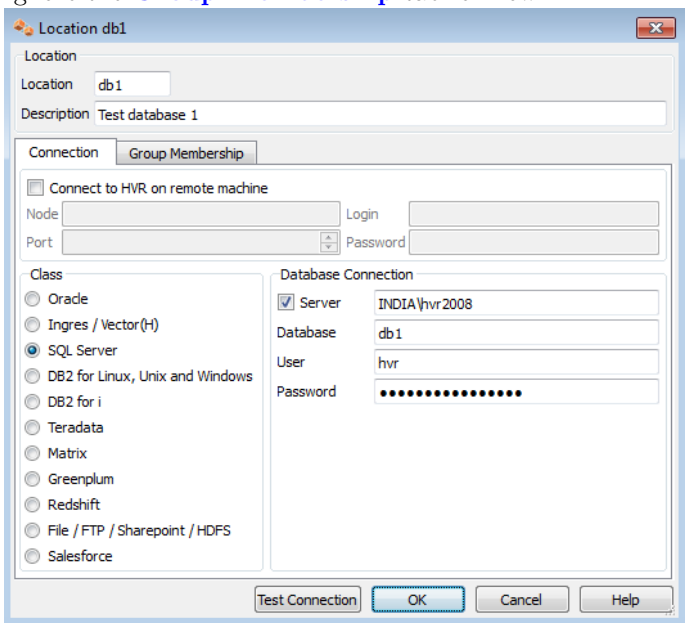
For a new hub database a dialog will prompt: **Do you wish to create the catalogs?** Answer **Yes**.

Create SQL Server Locations

Next create two locations (one for each database) using right–click on **Location Configuration** ▶ **New Location**.



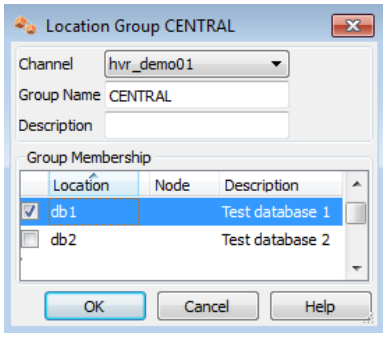
For the source database location there is no need to check **Connect to HVR on remote machine** because **testdb1** is on the same machine as the hub.
Ignore the **Group Membership** tab for now.



For the Azure database location, check the option **Connect to HVR on remote machine**. Enter the connection details of the VM. For SQL Server authentication Enter the Azure SQL database credentials at the **Database Connection** or use Windows Authentication by leaving the database credentials blank.

Create Location Groups

The channel needs two location groups. Under the new channel: right–click on **Location Groups** ▶ **New Group**. Enter a group name (for instance **CENTRAL**).



Add location **db1** as a member of this group by checking the box for **db1**.

Then create a second location group, called **DECENTRAL** that has member **db2**.

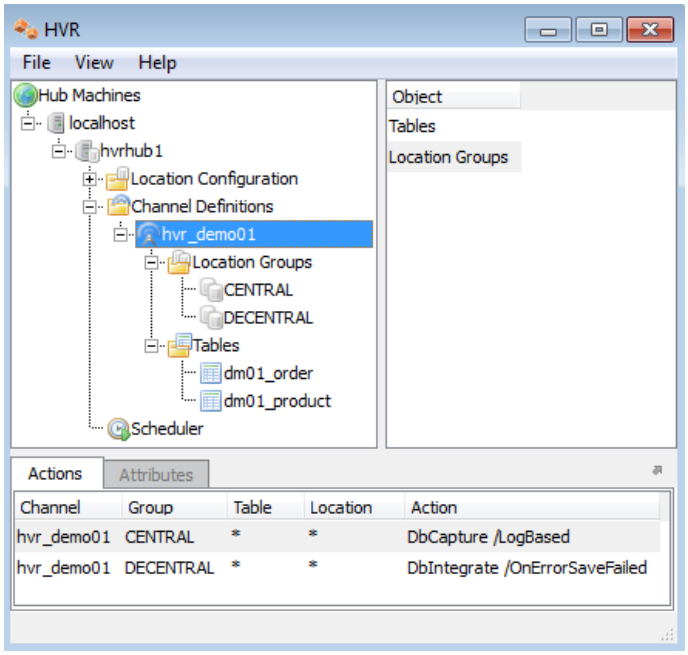
The new channel also needs a list of tables to replicate. This can be done as follows; right–click on **Tables** ▶ **Table Select**.

- Choose the first location ▶ **Connect**.
- In the **Table Selection** window, click on both tables and click **Add**.
- In new dialog **HVR Table Name** click **OK**.
- Close the **Table Selection** window.
- Perform table select again on one of the other locations and confirm that all tables to be replicated have value **Same** in column **Match**.

Define Actions

The new channel needs two actions to indicate the direction of replication.

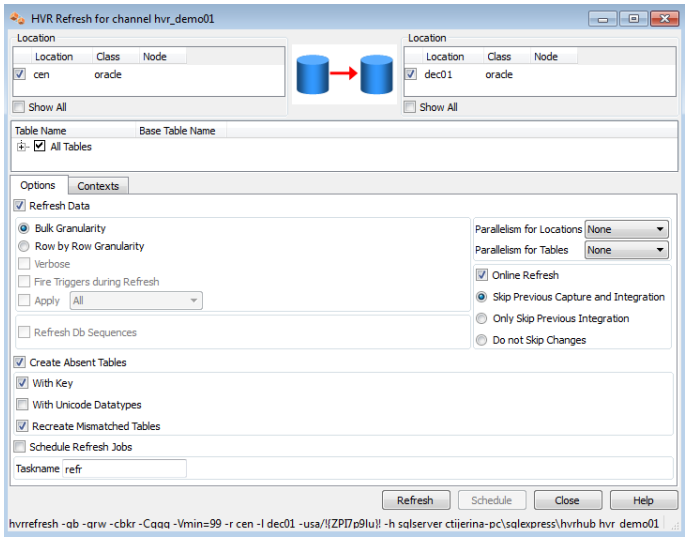
- Right–click on group **CENTRAL** ▶ **New Action** ▶ **DbCapture**. Check **/LogBased** so that the channel will detect changes using the log file, instead of using database triggers.
- Right–click on **Group DECENTRAL** ▶ **New Action** ▶ **DbIntegrate**. Check **/OnErrorSaveFailed**, this affects how replication errors are handled.



Note that the Actions pane only displays actions related to the objects selected in the left–hand pane. So click on channel **hvr_demo01** to see both actions.

Perform Initial Loading and Table Creation

HVR Refresh copies the data from one location to another location and optionally creates missing or mismatched tables and keys. In the HVR GUI, right-click on the channel and select **HVR Refresh**

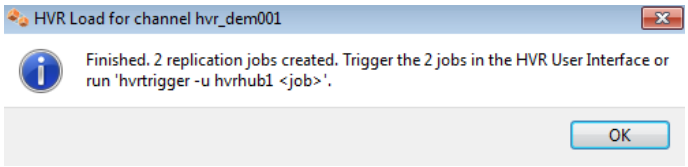


For the source select location **db1** and for target select location check **db2**. Check the options **Create Absent Tables, With Key, Recreate Mismatched Tables** and click **Refresh**.

Enable Replication with HVR Load

Now that the channel definition is complete, create the runtime replication system.

Right–click on channel **hvr_demo01** ▶ **HVR Load**. Choose **Create or Replace Objects** and click **HVR Load**.

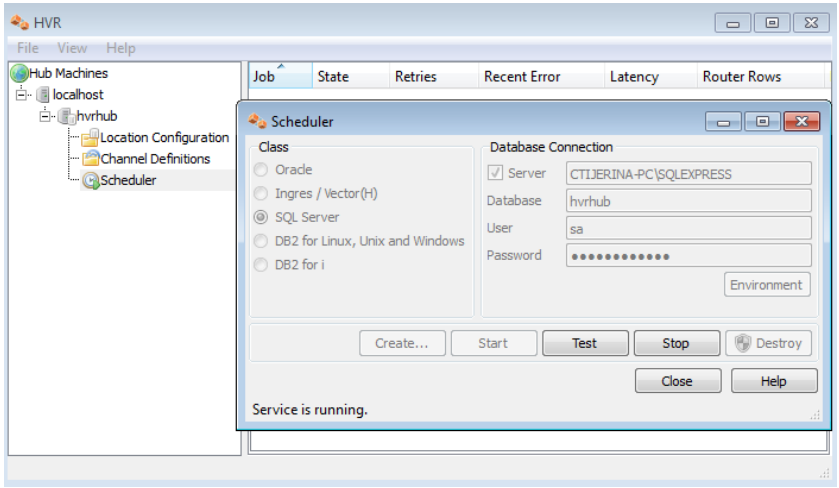


From the moment that HVR Load is done, all changes to database **testdb1** will be captured by HVR when its capture job looks inside the logging.

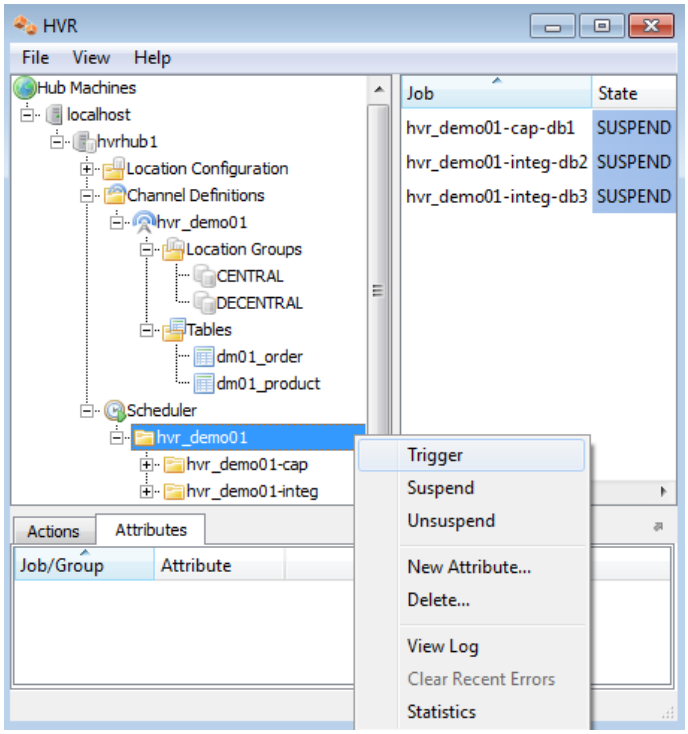
HVR Load also creates three replication jobs, which can be seen under the Scheduler node in the GUI.

Start Scheduling of Replication Jobs

Start the Scheduler on the hub machine by clicking in the HVR GUI on the **Scheduler** node of the hub database.



Next, instruct the HVR Scheduler to trigger the replication jobs.



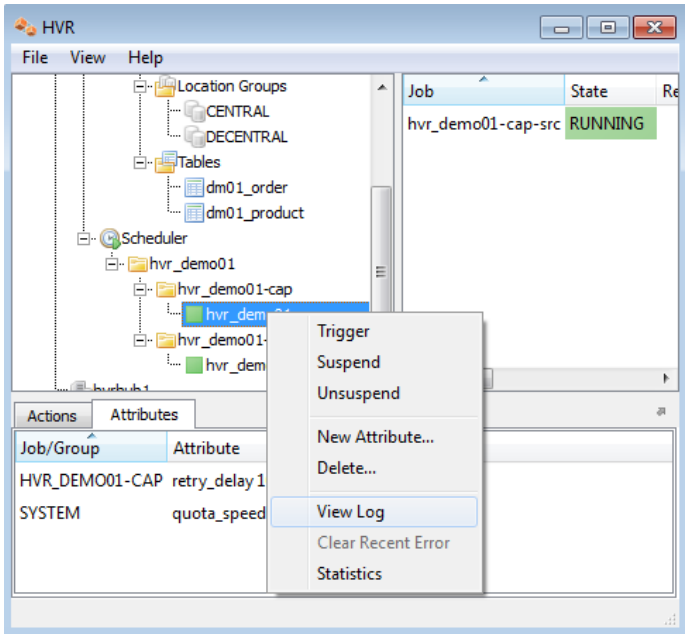
The replication jobs inside the Scheduler each execute a script under **$HVR_CONFIG/job/hvrhub/hvr_demo01** that has the same name as the job. So job **hvr_demo01–cap–db1** detects changes on database **testdb1** and stores these as transactions files on the hub machine. The other job (**hvr_demo01–integ–db2** picks up these transaction files and performs inserts, updates and deletes on the Azure target database.

Test Replication

To test replication, make a change in **testdb1**:

```
$ sqlplus testdb1/hvr
SQL> insert into dm01_product values (1, 19.99, 'DVD');
SQL> commit;
```

View the output of the jobs using a right mouse click on the job and select **View Log**. This log file is stored in **%HVR_CONFIG%\log***hubdb***\hvr_demo01–cap–db1**.



Here is sample job output:

```
C:\> notepad %HVR_CONFIG%\log\hvrhub\hvr.out
```

```
hvr_demo01-cap-db1: Capture cycle 1.
hvr_demo01-cap-db1: Selected 1 row from 'dm01_product__c0' (201 wide).
hvr_demo01-cap-db1: Routed 212 bytes (compression=42.7%) from 'db1' into \
1 location.
hvr_demo01-cap-db1: Finished. (elapsed=1.20s)
hvr_demo01-integ-db2: Integrate cycle 1 for 1 transaction file (212 bytes).
hvr_demo01-integ-db2: Integrated 1 change from 'dm01_product' (1 ins).
hvr_demo01-integ-db2: Integrate used 1 transaction and took 0.017 seconds.
hvr_demo01-integ-db2: Finished. (elapsed=0.14s)
```

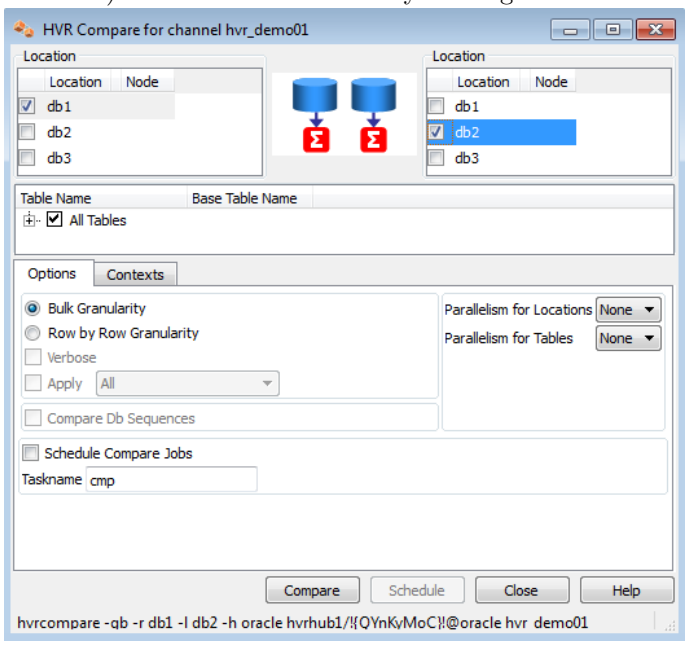
This log indicates that the jobs replicated the original change to **testdb2**. Run a query on **testdb2** to confirm:

```
$ sqlplus testdb2/hvr
SQL> select * from dm01_product;
```

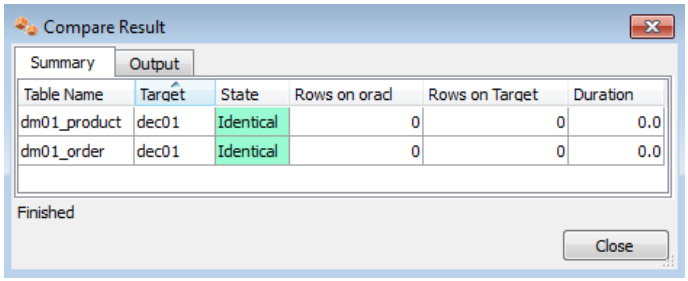
prod_id	prod_price	prod_descrip
1	19.99	DVD

HVR Compare and Refresh

HVR Compare checks whether two locations have identical rows, and HVR Refresh copies the content of one location to the second location. In the HVR GUI, right–click on a channel ▶ **HVR Compare** (or **HVR Refresh**). Choose two locations by clicking on the **Select** buttons.



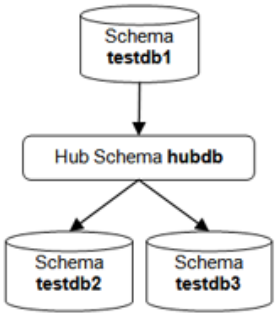
The outcome of the comparison is displayed below;



E

Quick Start for HVR on DB2

This appendix shows how to set up an HVR channel (called **hvr_demo01**) to replicate between DB2 databases. In real life HVR would usually replicate between databases on different machines. But for simplicity, in this example HVR will replicate between three DB2 databases on the hub machine. The steps start by creating new databases and tables for HVR to replicate between.



Create Test Databases and Tables

Create three test databases, each containing two empty tables named **dm01_product** and **dm01_order**. If replication is configured between existing databases and tables then this step should be skipped.

```
$ db2 create database testdb1
$ db2 create database testdb2
$ db2 create database testdb3
```

Create the test tables.

```
$ cd $HVR_HOME/demo/hvr_demo01/base/db2
$ db2 connect testdb1
$ db2 '-td;;' <hvr_demo01.cre
$ db2 '-td;;' <hvr_demo01.mod
$ db2 connect reset
$ db2 connect testdb2
$ db2 '-td;;' <hvr_demo01.cre
$ db2 '-td;;' <hvr_demo01.mod
$ db2 connect reset
$ db2 connect testdb3
$ db2 '-td;;' <hvr_demo01.cre
$ db2 '-td;;' <hvr_demo01.mod
$ db2 connect reset
$ db2 update db cfg for testdb1 using logretain on
$ db2 backup db testdb1
```

Install HVR

First read section Introduction which explains the HVR's terminology and architecture. In particular this explains the importance of a hub database.
Then install the HVR software on the hub machine by following the installation steps in section New Installation on Unix or Linux or New Installation on Windows. If the hub machine is a Unix machine then HVR can either be installed on a Windows PC (so the HVR GUI can run on the PC and connect to the Unix hub

machine) or the HVR GUI can be run on the Unix hub machine and connect back to an X server running on a PC.

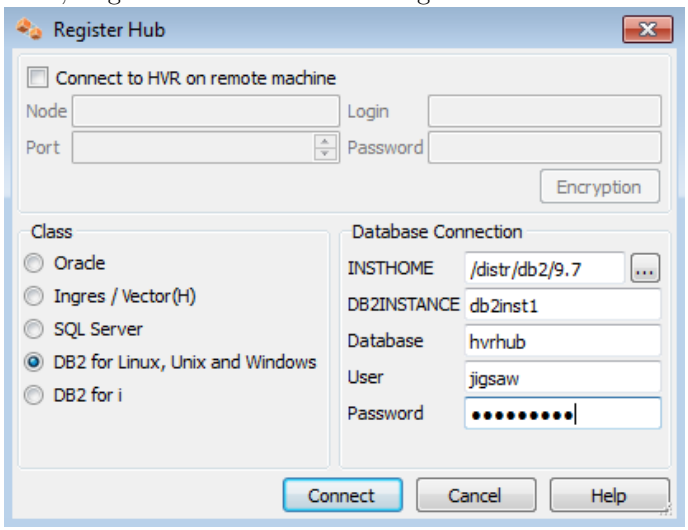
Create the Hub Database

Create the hub database, in which the HVR GUI will store the channel definition.

```
$ db2 create database hvrhub
```

Connect to Hub Database

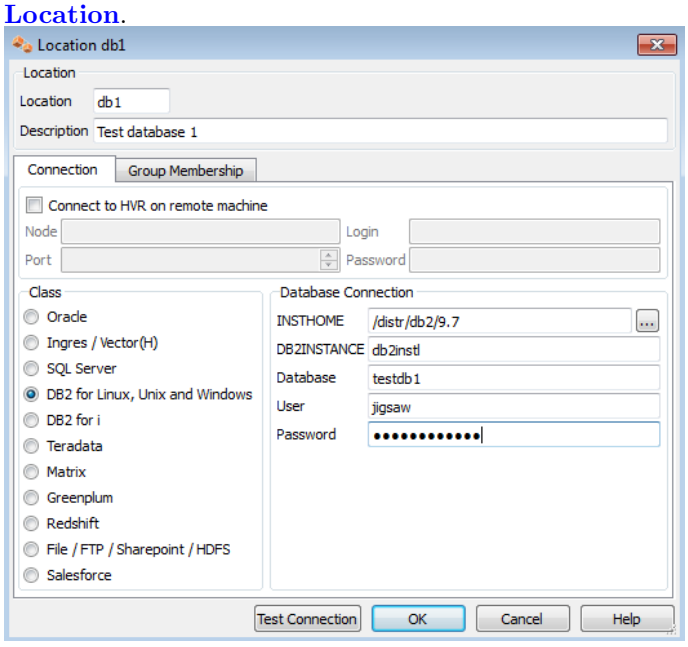
Start the HVR GUI on a PC by clicking on the HVR GUI icon (this is created by the HVR Installer for Windows) or by running **hvrgui** on Linux.
First, Register the hub database: right–click on **hub machines** ▶ **Register hub**.



Enter connection details and click **Connect**. For a new hub database a dialog will prompt **Do you wish to create the catalogs?**; answer **Yes**.

Create DB2 Locations

Next create three locations (one for each test database) using right–click on **Location Configuration** ▶ **New Location**.

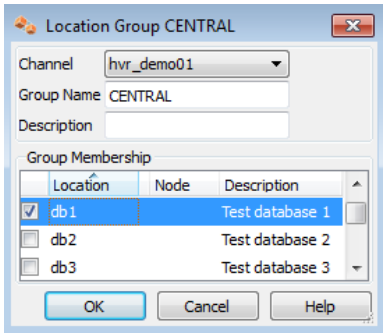


In this example there is no need to check **Connect to HVR on remote machine** because **testdb1** is on the same machine as the hub.

Ignore the **Group Membership** tab for now.
Make locations for **testdb2** and **testdb3** too.
Now define a channel using **Channel Definitions ▶ New Channel**.

Create Location Groups

The channel needs two location groups. Under the new channel: right–click on **Location Groups ▶ New Group**. Enter a group name (for instance **CENTRAL**).



Add location **db1** as a member of this group by checking the box for **db1**.

Then create a second location group, called **DECENTRAL** that has members **db2** and **db3**.

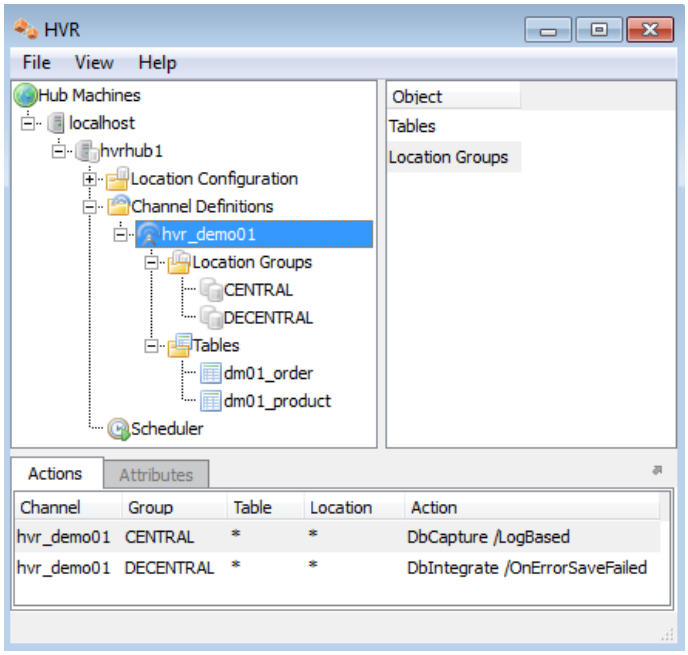
The new channel also needs a list of tables to replicate. This can be done as follows; right–click on **Tables ▶ Table Select**.

- Choose the first of the three locations ▶ **Connect**.
- In the **Table Selection** window, click on both tables and click **Add**.
- In new dialog **HVR Table Name** click **OK**.
- Close the **Table Selection** window.
- Perform table select again on one of the other locations and confirm that all tables to be replicated have value **Same** in column **Match**.

Define Actions

The new channel needs two actions to indicate the direction of replication.

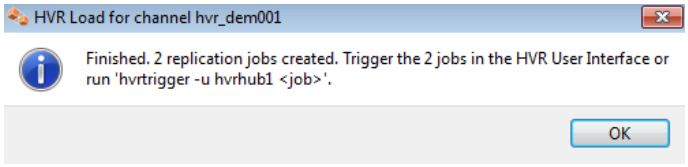
- Right–click on group **CENTRAL ▶ New Action ▶ DbCapture**. Check **/LogBased** so that the channel will detect changes using the log file, instead of using database triggers.
- Right–click on **Group DECENTRAL ▶ New Action ▶ DbIntegrate**. Check **/OnErrorSaveFailed**, this affects how replication errors are handled.



Note that the Actions pane only displays actions related to the objects selected in the left–hand pane. So click on channel **hvr_demo01** to see both actions.

Enable Replication with HVR Load

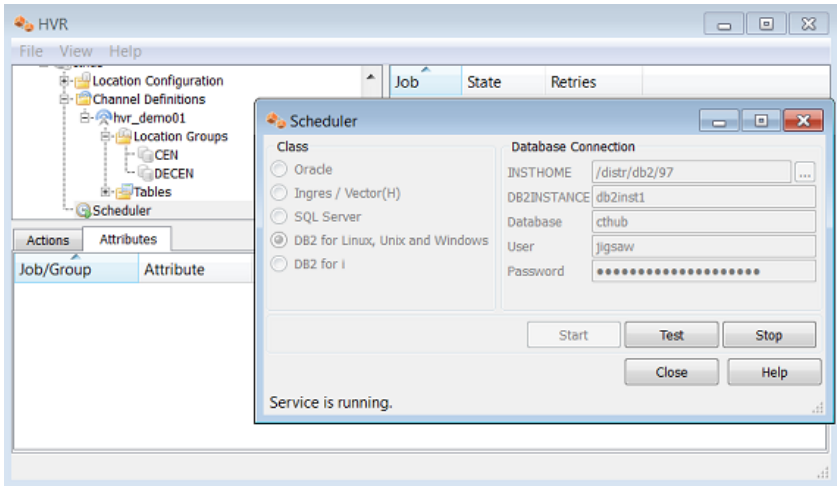
Now that the channel definition is complete, create the runtime replication system.
Right–click on channel **hvr_demo01** ▶ **HVR Load**. Choose **Create or Replace Objects** and click **HVR Load**.



From the moment that HVR Load is done, all changes to database **testdb1** will be captured by HVR when its capture job looks inside the logging.
HVR Load also creates three replication jobs, which can be seen under the Scheduler node in the GUI.

Start Scheduling of Replication Jobs

Start the Scheduler on the hub machine by clicking in the HVR GUI on the **Scheduler** node of the hub database.



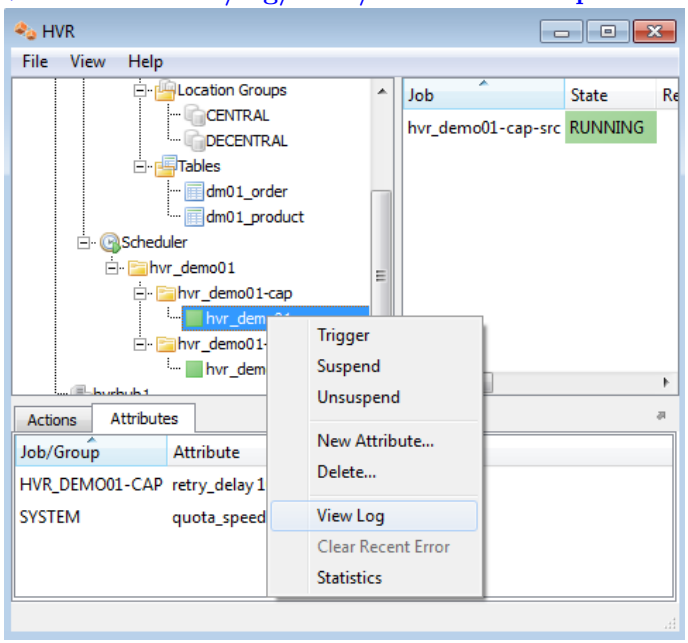
Next, instruct the HVR Scheduler to trigger the replication jobs.
The replication jobs inside the Scheduler each execute a script under **$HVR_CONFIG/job/hvrhub/hvr_demo01** that has the same name as the job. So job **hvr_demo01–cap–db1** detects changes on database **testdb1** and stores these as transactions files on the hub machine.
The other two jobs (**hvr_demo01–integ–db2** and **hvr_demo01–integ–db3**) pick up these transaction files and perform inserts, updates and deletes on the two target databases.

Test Replication

To test replication, make a change in **testdb1**:

```
testdb1/hvr
SQL> insert into dm01_product values (1, 19.99, 'DVD');
SQL> commit;
```

In the HVR log file you can see the output of the jobs by clicking on **View Log**. This log file can be found in **$HVR_CONFIG/log/*hubdb*/hvr_demo01–cap–db1**.



The job output looks like this:

```
hvr_demo01-cap-db1: Scanned 1 transaction containing 1 row (1 ins) for 1 table.
hvr_demo01-cap-db1: Routed 215 bytes (compression=40.6%) from 'db1' into 2 locations.
hvr_demo01-cap-db1: Capture cycle 3.
hvr_demo01-integ-db2: Integrate cycle 2 for 1 transaction file (215 bytes).
hvr_demo01-integ-db2: Integrated 1 change from 'dm01_product' (1 ins).
hvr_demo01-integ-db2: Integrate used 1 transaction and took 0.004 seconds.
hvr_demo01-integ-db3: Integrate cycle 2 for 1 transaction file (215 bytes).
hvr_demo01-integ-db3: Integrated 1 change from 'dm01_product' (1 ins).
hvr_demo01-integ-db3: Integrate used 1 transaction and took 0.013 seconds.
hvr_demo01-integ-db3: Waiting...
```

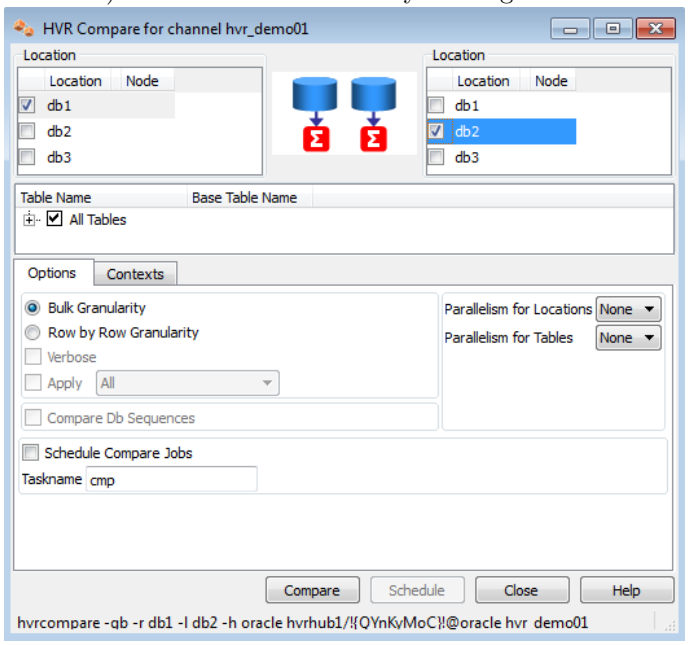
This indicates that the jobs replicated the original change to **testdb2** and **testdb3**. A query on **testdb2** confirms this:

```
testdb2/hvr
SQL> select * from dm01_product;
```

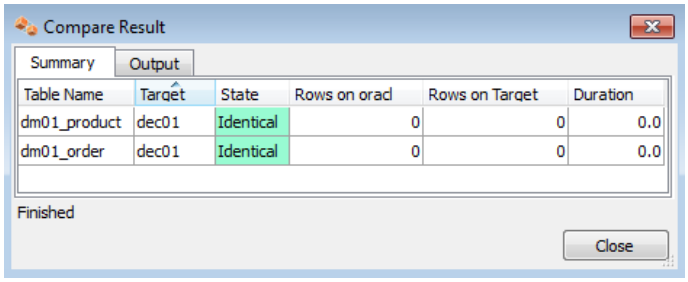
prod_id	prod_price	prod_descrip
1	19.99	DVD

HVR Compare and Refresh

HVR Compare checks whether two locations have identical rows, and HVR Refresh copies the content of one location to the second location. In the HVR GUI, right–click on a channel ▶ **HVR Compare** (or **HVR Refresh**). Choose two locations by clicking on the **Select** buttons.

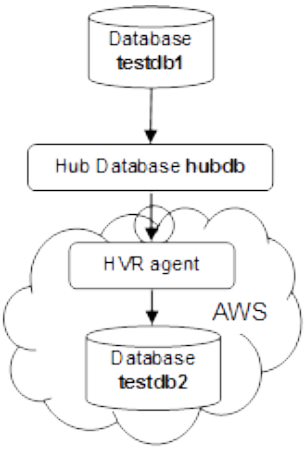


The outcome of the comparison is displayed below;



Quick Start for HVR into Redshift

This appendix shows how to set up an HVR channel (called **hvr_demo01**) to replicate between a local Oracle database database and an Amazon Redshift MPP database residing in the Amazon cloud (AWS). To connect efficiently to AWS, an HVR agent installation on an EC2 Linux VM is used. The steps actually start by creating new databases and tables for HVR to replicate between. In real life these databases would already exist and be filled with the user tables, but for simplicity everything is created from scratch. Also, for simplicity, we will assume the source database resides on the hub as well and Oracle has already been installed there.



Create Test Databases and Tables

Generally when getting started with HVR a source schema with tables and data already exists. If so then this step can be skipped.
This Quickstart uses two empty tables named **dm01_product** and **dm01_order**. In an existing Oracle database, create a test schema and create the tables using the following commands.

```
$ sqlplus system/manager
SQL> create user testdb1 identified by hvr
 2  default tablespace users
 3  temporary tablespace temp
 4  quota unlimited on users;

SQL> grant create session to testdb1;
SQL> grant create table to testdb1;
SQL> grant create sequence to testdb1;
SQL> grant create procedure to testdb1;
SQL> grant create trigger to testdb1;
SQL> grant create view to testdb1;
SQL> grant execute any procedure to testdb1;
```

Create the test tables.

```
$ cd $HVR_HOME/demo/hvr_demo01/base/oracle
$ sqlplus testdb1/hvr < hvr_demo01.cre
$ sqlplus testdb1/hvr < hvr_demo01.mod
```

In the AWS portal, create a Redshift database using **AWS -> Redshift Cluster -> Launch** with cluster identifier **testdb2**. If you don't have a **VPC** and **security group** yet, you will create them during this step. You can use the defaults to create a minimal cluster or choose a multi node cluster. After creation, you can either create the tables in the cluster using the above .cre and .mod scripts through the **SQL Workbench** console or let HVR create them during initial loading (**HVR Refresh** with **Create Absent Tables**).

Install HVR on-premise

First read section Introduction which explains the HVR's terminology and architecture. In particular this explains the importance of a hub database.
Then install the HVR software on the hub machine by following the installation steps in section New Installation on Unix or Linux or New Installation on Windows. If the hub machine is a Unix machine then HVR can either be installed on a Windows PC (so the HVR GUI can run on the PC and connect to the Unix hub machine) or the HVR GUI can be run on the Unix hub machine and connect back to an X server running on a PC.
This Quickstart assumes the Oracle Database on the hub server is also the source database. Most real-time integration scenarios use log-based capture (parameter **/LogBased**). To enable log-base capture, configure the following:

- The user name that HVR uses must be in Oracle's group. On Unix and Linux this can be done by adding the user name used by HVR to the line in **/etc/group** that begins with **dba**. On Windows right–click **My Computer** and select **Manage ▶ Local Users and Groups ▶ Groups ▶ ora_dba ▶ Add to Group ▶ Add**.
- The Oracle instance should have archiving 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 query **select destination, status from v$archive_dest.**
By default, this will return values **USE_DB_RECOVERY_FILE_DEST, VALID**, which is inside the flashback recovery area. Alternatively, an archive destination can be defined with the following statement: **alter system set log_archive_dest_1='location=/disk1/arc'** and then restart the instance.

Install HVR agent on Amazon Linux VM

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 New Installation 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.

Create the Hub Database

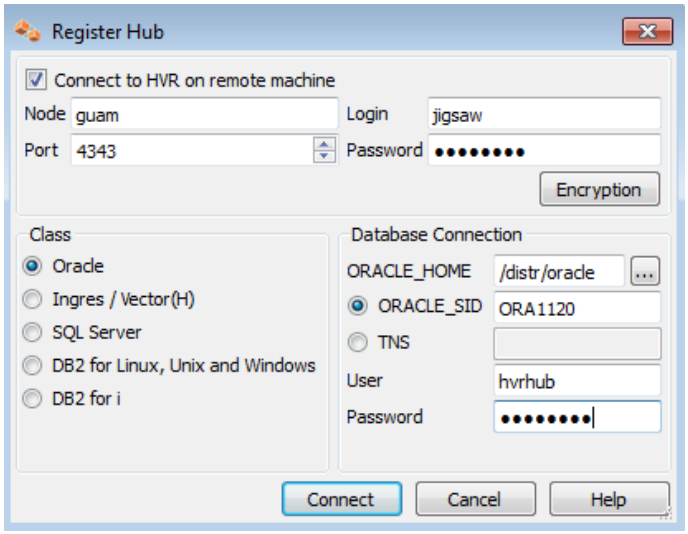
Create the hub database, in which the HVR GUI will store the channel definition. This is actually another user/schema in the Oracle instance.

```
$ sqlplus system/manager
SQL> create user hvrhub identified by hvr
 2  default tablespace users
 3  temporary tablespace temp
 4  quota unlimited on users;

SQL> grant create session to hvrhub;
SQL> grant create table to hvrhub;
SQL> grant create sequence to hvrhub;
SQL> grant create procedure to hvrhub;
SQL> grant create trigger to hvrhub;
SQL> grant create view to hvrhub;
SQL> grant execute any procedure to hvrhub;
```

Connect To Hub Database

Start the HVR GUI on a PC by clicking on the HVR GUI icon (this is created by the HVR Installer for Windows) or by running **hvrgui** on Linux.

First, Register the hub database: right–click on **hub machines** ▶ **Register hub**. Enter connection details.

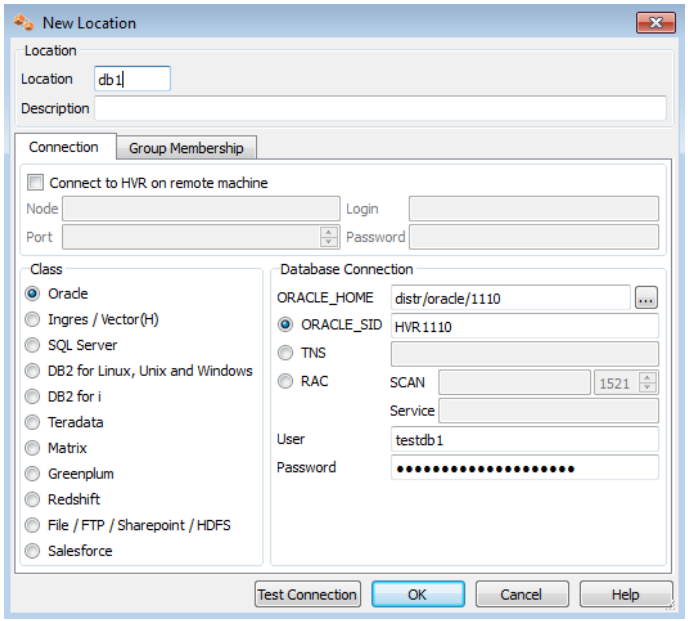


In this example the hub is a machine called **guam**, where an INET daemon is listening on port 4343. See section New Installation on Unix or Linux for how to configure this.

For a new hub database a dialog will prompt **Do you wish to create the catalogs?**; answer **Yes**.

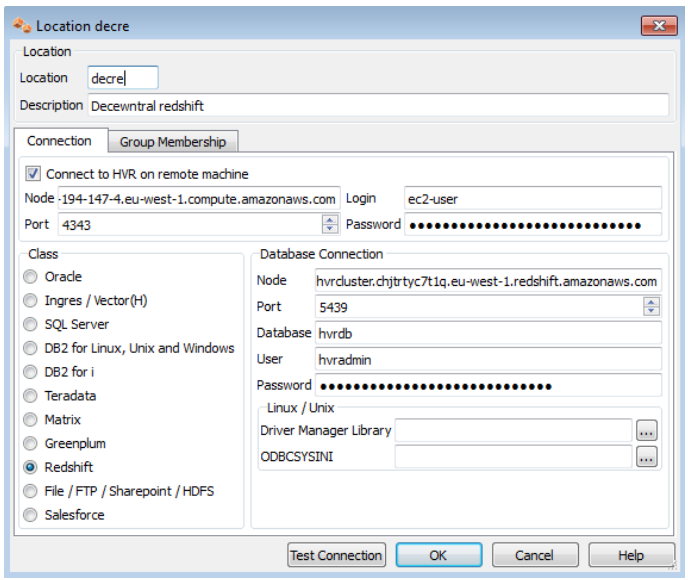
Create Oracle and Redshift Locations

Next create the source locations using right–click on **Location Configuration** ▶ **New Location**.



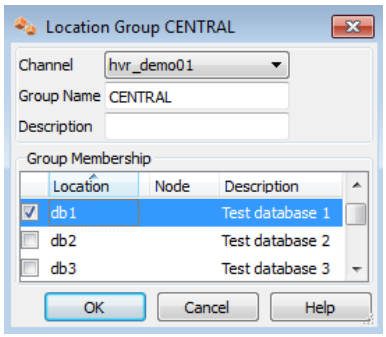
In this example there is no need to check **Connect to HVR on remote machine** because **testdb1** is on the same machine as the hub.
Ignore the **Group Membership** tab for now.

Create another location for the Redshift database **testdb2** too. Now do tick **Connect to HVR on remote machine**, because HVR needs to connect to the HVR remote listener agent installed before. Fill in the details of the VM running the remote listener there. Choose Redshift as database and fill in the Redshift Connection info.



Create Location Groups

The channel needs two location groups. Under the new channel: right–click on **Location Groups** ▶ **New Group**. Enter a group name (for instance **CENTRAL**).



Add location **db1** as a member of this group by checking the box for **db1**.

Then create a second location group, called **DECENTRAL** that has members **db2** and **db3**.

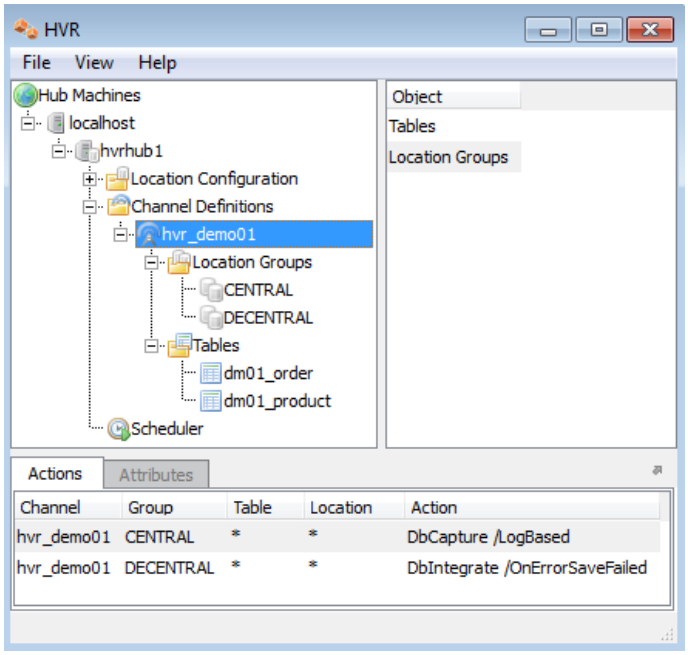
The new channel also needs a list of tables to replicate. This can be done as follows; right–click on **Tables** ▶ **Table Select**.

- Choose the first of the three locations ▶ **Connect**.
- In the **Table Selection** window, click on both tables and click **Add**.
- In new dialog **HVR Table Name** click **OK**.
- Close the **Table Selection** window.
- Perform table select again on one of the other locations and confirm that all tables to be replicated have value **Same** in column **Match**.

Define Actions

The new channel needs two actions to indicate the direction of replication.

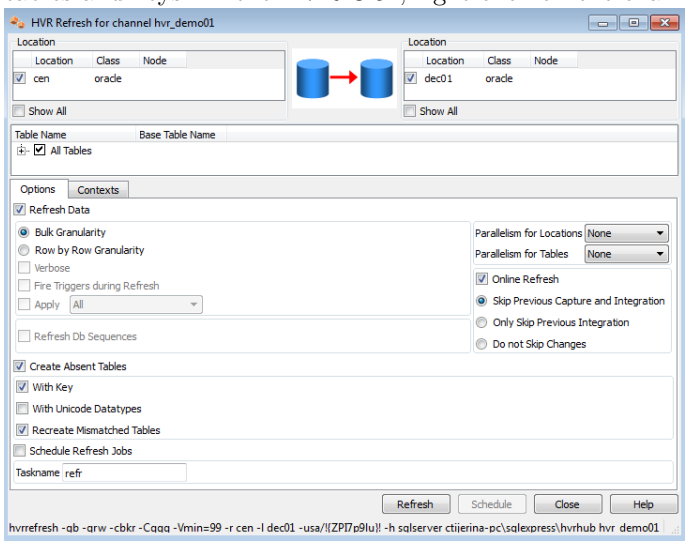
- Right–click on group **CENTRAL** ▶ **New Action** ▶ **DbCapture**. Check **/LogBased** so that the channel will detect changes using the log file, instead of using database triggers.
- Right–click on **Group DECENTRAL** ▶ **New Action** ▶ **DbIntegrate**. Check **/OnErrorSaveFailed**, this affects how replication errors are handled.



Note that the Actions pane only displays actions related to the objects selected in the left–hand pane. So click on channel **hvr_demo01** to see both actions.

Perform Initial Loading and Table Creation

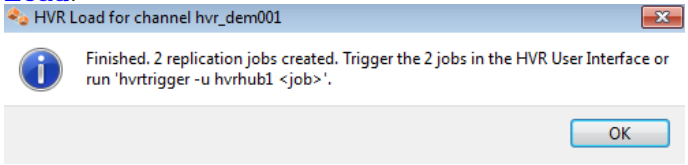
HVR Refresh copies the data from one location to another location and optionally creates missing or mismatched tables and keys. In the HVR GUI, right-click on the channel and select **HVR Refresh**



For the source select location **db1** and for target select location check **db2**. Check the options **Create Absent Tables, With Key, Recreate Mismatched Tables** and click **Refresh**.

Enable Replication with HVR Load

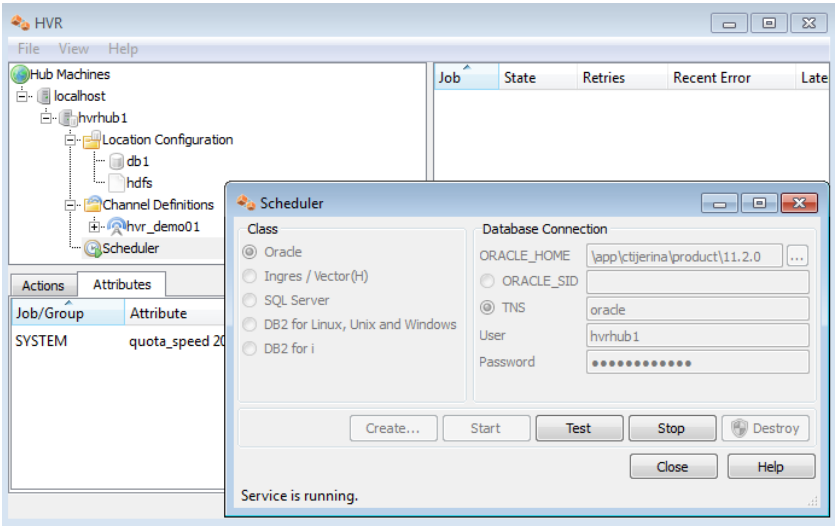
Now that the channel definition is complete, create the runtime replication system.
Right–click on channel **hvr_demo01 ▶ HVR Load**. Choose **Create or Replace Objects** and click **HVR Load**.



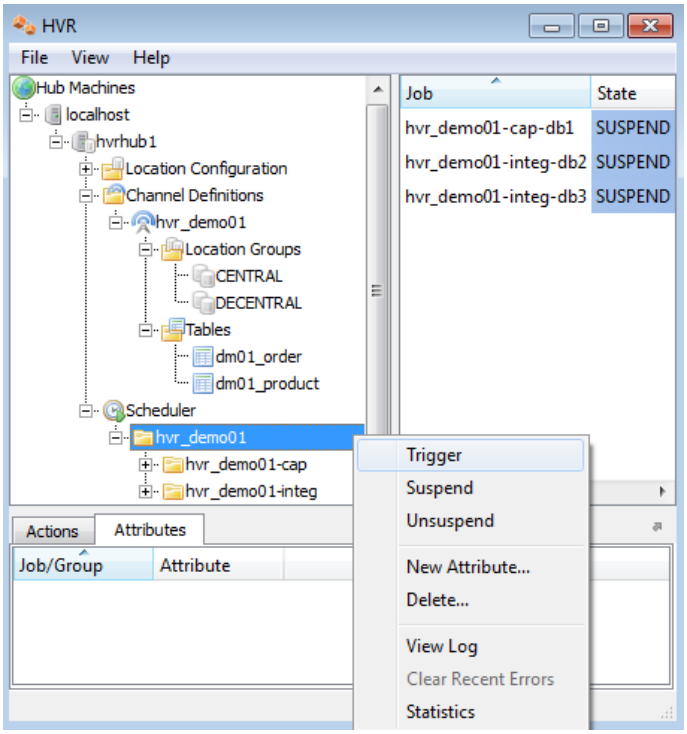
From the moment that HVR Load is done, all changes to database **testdb1** will be captured by HVR when its capture job looks inside the logging.
HVR Load also creates three replication jobs, which can be seen under the Scheduler node in the GUI.

Start Scheduling of Replication Jobs

Start the Scheduler on the hub machine by clicking in the HVR GUI on the **Scheduler** node of the hub database.



Next, instruct the HVR Scheduler to trigger the replication jobs.



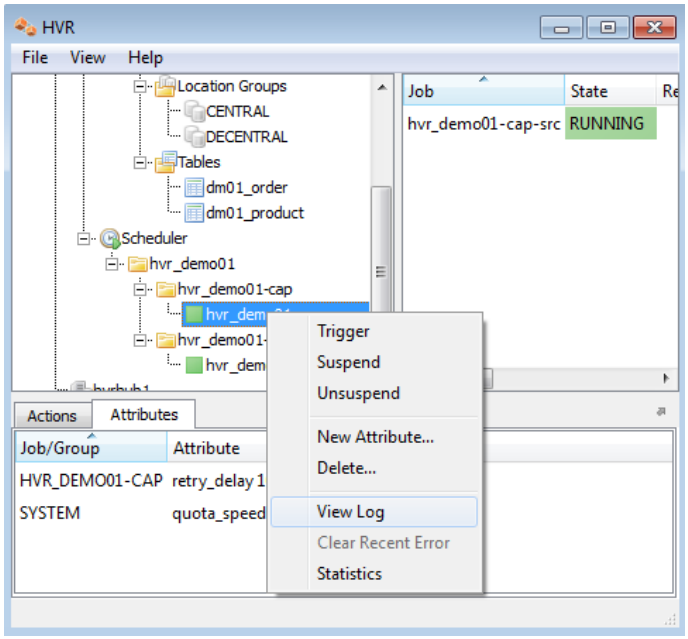
The replication jobs inside the Scheduler each execute a script under **$HVR_CONFIG/job/hvrhub/hvr_demo01** that has the same name as the job. So job **hvr_demo01–cap–db1** detects changes on database **testdb1** and stores these as transactions files on the hub machine. The other two jobs (**hvr_demo01–integ–db2** and **hvr_demo01–integ–db3**) pick up these transaction files and perform inserts, updates and deletes on the two target databases.

Test Replication

To test replication, make a change in **testdb1**:

```
testdb1/hvr
SQL> insert into dm01_product values (1, 19.99, 'DVD');
SQL> commit;
```

In the HVR log file you can see the output of the jobs by clicking on **View Log**. This log file can be found in **$HVR_CONFIG/log/*hubdb*/hvr_demo01–cap–db1**.



The job output looks like this:

```
hvr_demo01-cap-db1: Scanned 1 transaction containing 1 row (1 ins) for 1 table.
hvr_demo01-cap-db1: Routed 215 bytes (compression=40.6%) from 'db1' into 1 location.
hvr_demo01-cap-db1: Capture cycle 1.
hvr_demo01-integ-db2: Integrate cycle 2 for 1 transaction file (215 bytes).
hvr_demo01-integ-db2: Integrated 1 change from 'dm01_product' (1 ins).
hvr_demo01-integ-db2: Integrate used 1 transaction and took 0.004 seconds.
hvr_demo01-integ-db2: Waiting...
```

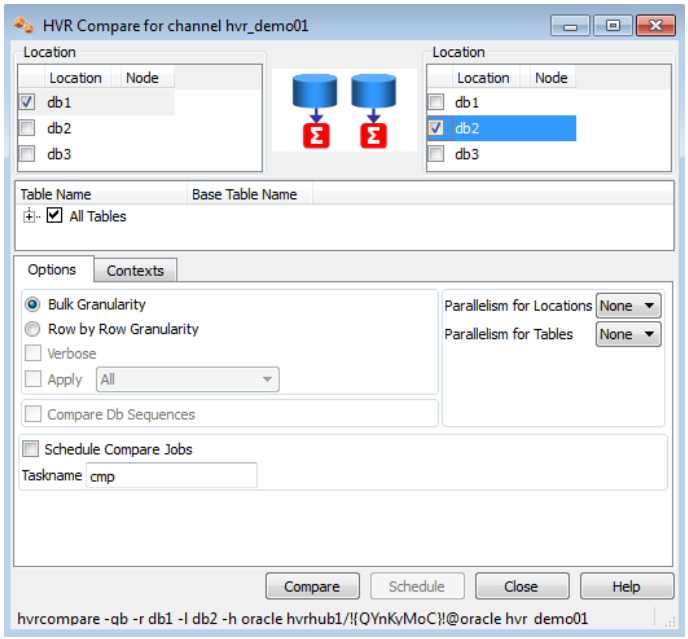
This indicates that the jobs replicated the original change to **testdb2**. A query on **testdb2** confirms this:

```
SQL> select * from dm01_product;
```

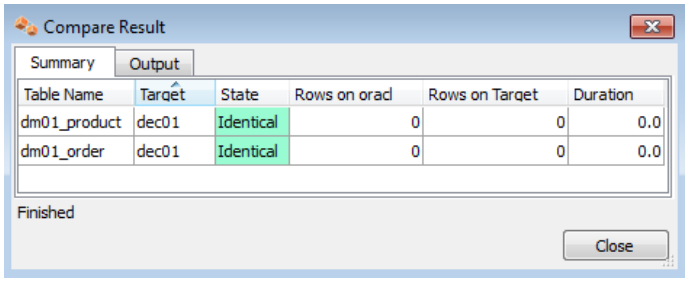
prod_id	prod_price	prod_descrip
1	19.99	DVD

HVR Compare and Refresh

HVR Compare checks whether two locations have identical rows, and HVR Refresh copies the content of one location to the second location. In the HVR GUI, right–click on a channel ▶ **HVR Compare** (or **HVR Refresh**). Choose two locations by clicking on the **Select** buttons.



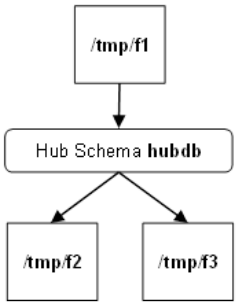
The outcome of the comparison is displayed below;



G

Quick Start for File Replication

This appendix shows how to set up an HVR channel (called **hvr_demo31**) to replicate files between directories. In real life, HVR would usually replicate between directories on different machines. Some of these directories would be reached via FTP, SFTP or SharePoint/WebDAV. But for simplicity, in this example HVR will replicate between three directories all created on the hub machine; files are captured from subdirectory **/tmp/f1** and copied to **/tmp/f2** and **/tmp/f3**. The channel is a 'blob' file channel, which means it has no table information and simply treats each file as a sequence of bytes without understanding their file format.



Create File Locations

Create three directories to test replication:

```
$ mkdir /tmp/f1
$ mkdir /tmp/f2
$ mkdir /tmp/f3
```

Install HVR

First read section Introduction which explains the HVR's terminology and architecture. In particular this explains the importance of a hub database.
Then install the HVR software on the hub machine by following the installation steps in section New Installation on Unix or Linux or New Installation on Windows. If the hub machine is a Unix machine then HVR can either be installed on a Windows PC (so the HVR GUI can run on the PC and connect to the Unix hub machine) or the HVR GUI can be run on the Unix hub machine and connect back to an X server running on a PC.

Create the Hub Database

Create the hub database, in which the HVR GUI will store the channel definition. This can be an Ingres database, Oracle schema or a SQL Server database. The steps to create an Oracle hub database schema are as follows:

```
$ sqlplus system/manager
SQL> create user hvrhub identified by hvr
 2  default tablespace users
 3  temporary tablespace temp
```

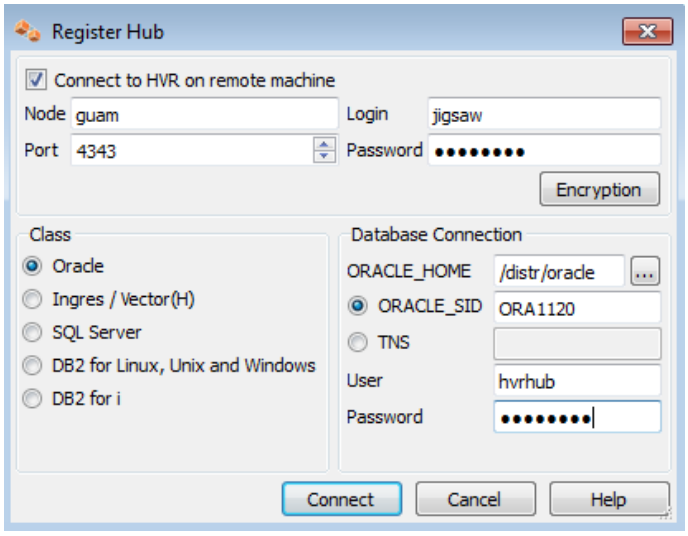
```
 4  quota unlimited on users;

SQL> grant create session to hvrhub;
SQL> grant create table to hvrhub;
SQL> grant create sequence to hvrhub;
SQL> grant create procedure to hvrhub;
SQL> grant create trigger to hvrhub;
SQL> grant create view to hvrhub;
SQL> grant execute any procedure to hvrhub;

$ sqlplus
Enter user-name: / as sysdba
SQL> grant execute on dbms_alert to hvrhub;
SQL> exit;
```

See also Create the Hub Database in Ingres, Create the Hub Database in SQL Server

Connect to Hub Database

Start the HVR GUI on a PC by clicking on the HVR GUI icon (this is created by the HVR Installer for Windows) or by running **hvrgui** on Linux.

First, Register the hub database: right–click on **hub machines** ▶ **Register hub**. Enter connection details.



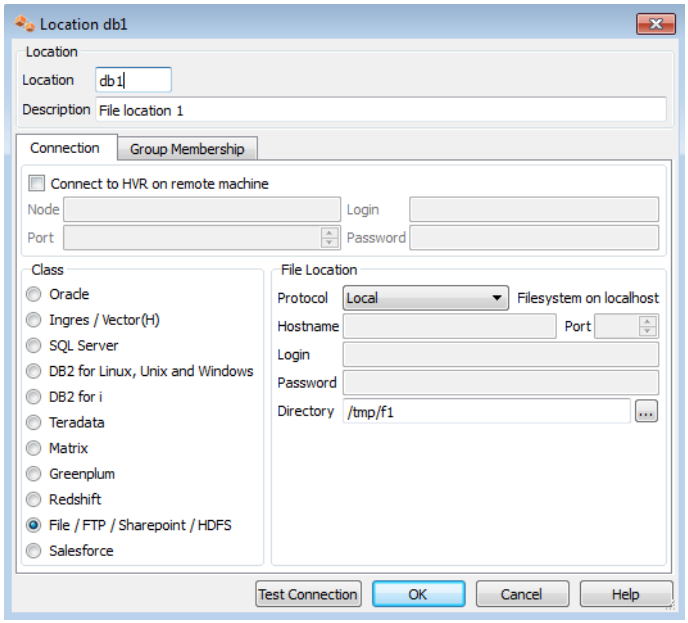
In this example the hub is a machine called **guam**, where an INET daemon is listening on port 4343. See section New Installation on Unix or Linux for how to configure this.

For a new hub database a dialog will prompt **Do you wish to create the catalogs?**; answer **Yes**.

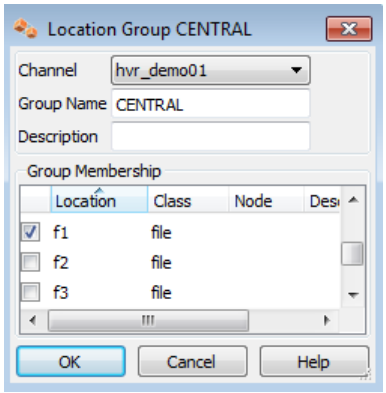
See also: Connect to Hub Database (Ingres), Connect to Hub Database in SQL Server

Create Channel and Location groups

Next create three file locations (one for each replicated directory) using right–click on **Location Configuration** ▶ **New Location**.



In this example there is no need to check **Connect to HVR on remote machine** because **/tmp/f1** is on the same machine as the hub.
Ignore the **Group Membership** tab for now.
Make locations for **/tmp/f2** and **/tmp/f3** as well.
Now define a channel using **Channel Definitions ▶ New Channel**.

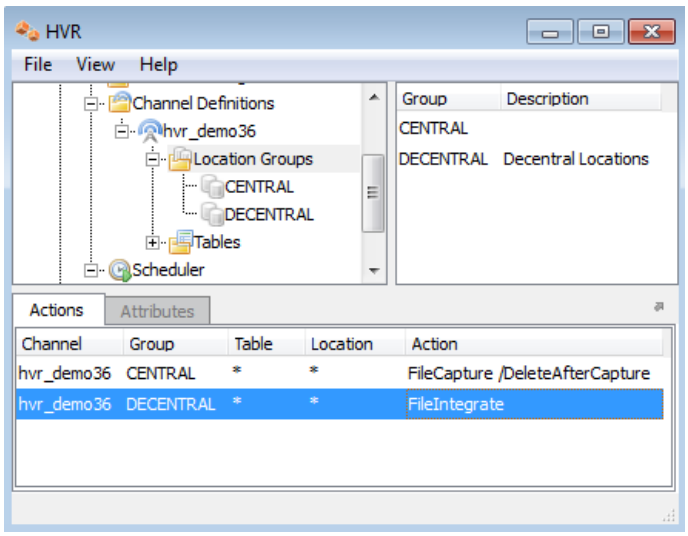


The channel needs two location groups. Under the new channel: right–click on **Location Groups ▶ New Group**. Enter a group name (for instance **CENTRAL**).
Add location **f1** as a member of this group by checking the box for **f1**.
Then create a second location group, called **DECENTRAL** that has members **f2** and **f3**.
This is a 'blob' file channel so it has no table layout information.

Define Actions

The new channel needs two actions to indicate the direction of replication.

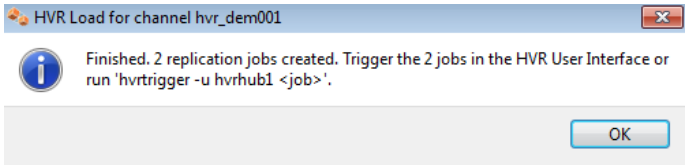
- Right–click on group **CENTRAL ▶ New Action ▶ FileCapture**. If parameter **/DeleteAfterCapture** is checked, then HVR will remove files from the source directory after they are captured. Otherwise, the contents of the directory (and its subdirectories) will be copied and changes will be detected by monitoring the files' timestamps.
- Right–click on **Group DECENTRAL ▶ New Action ▶ FileIntegrate**.



Note that the **Actions** pane only displays actions related the objects selected in the left hand pane. So click on channel **hvr_demo31** to see both actions.

Enable Replication with HVR Load

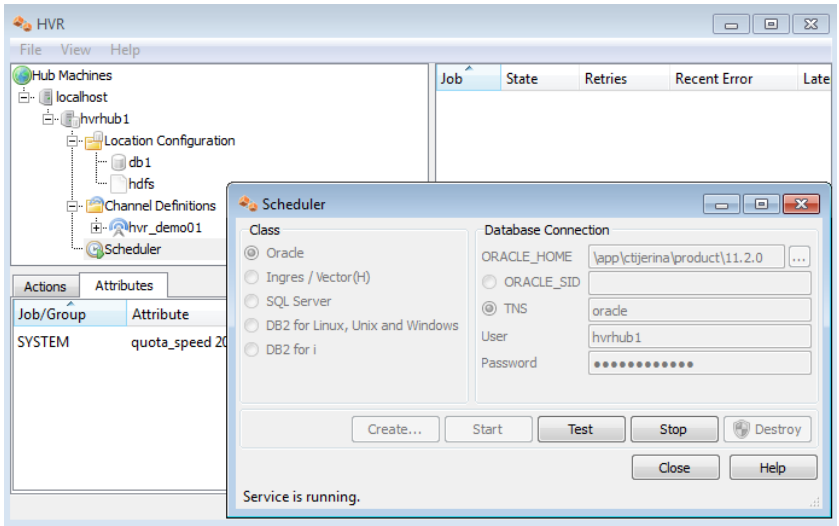
Now that the channel definition is complete, create the runtime replication system.
Right–click on channel **hvr_demo31** ▶ **HVR Load**. Choose **Create or Replace Objects** and click **HVR Load**.



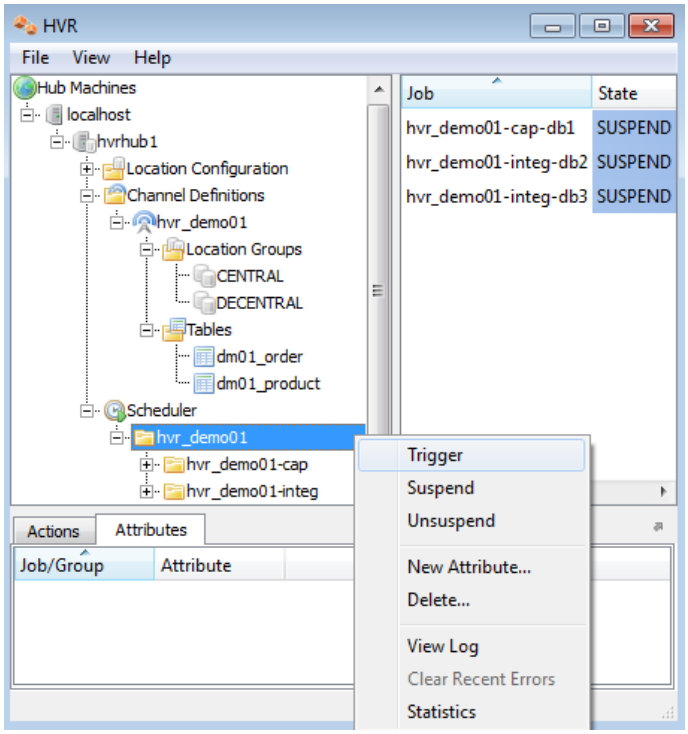
From the moment that HVR Load is done, all changes in directory **/tmp/f1** will be captured by HVR.
HVR Load also creates three replication jobs, which can be seen under the Scheduling node in the HVR GUI.

Start Scheduling of Replication Jobs

Start the Scheduler on the hub machine by clicking in the HVR GUI on the **Scheduler** node of the hub database.



Next, instruct the HVR Scheduler to trigger the replication jobs.



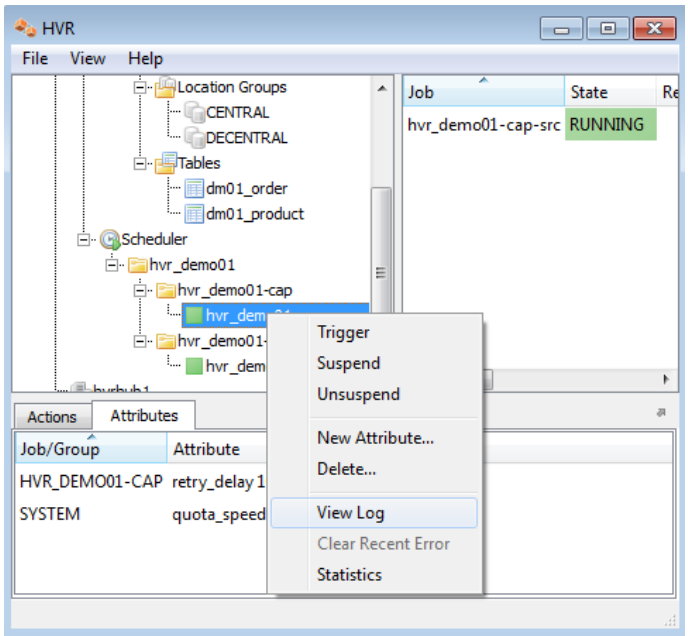
The replication jobs inside the Scheduler each execute a script under **$HVR_CONFIG/job/hvrhub/hvr_demo31** that has the same name as the job. So job **hvr_demo31–cap–f1** detects changes in directory **/tmp/f1** and stores these as transactions files on the hub machine. The other two jobs (**hvr_demo31–integ–f2** and **hvr_demo31–integ–f3**) pick up these transaction files, and copy new or modified files to the two target directories.

Test Replication

To test replication, copy any file into directory **/tmp/f1**:

```
$ echo hello > /tmp/f1/world
```

In the HVR log file you can see the output of the jobs by clicking on **View Log**. This log file can be found in **$HVR_CONFIG/log/*hubdb*/hvr_demo01–cap–db1**.



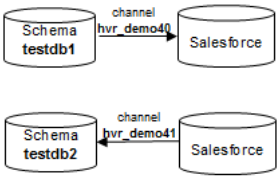
The job output looks like this:

```
hvr_demo31-integ-f3: Waiting...
hvr_demo31-cap-f1: Capture cycle 1 for 2 files (37 bytes).
hvr_demo31-cap-f1: Routed 150 bytes (compression=19.8%) from 'f1' into 2 locations.
hvr_demo31-integ-f2: Integrate cycle 2 for 1 transaction file (150 bytes).
hvr_demo31-cap-f1: Waiting...
hvr_demo31-integ-f2: Moved 2 files to 'C:\tmp\f2'.
hvr_demo31-integ-f2: Waiting...
hvr_demo31-integ-f3: Integrate cycle 2 for 1 transaction file (150 bytes).
hvr_demo31-integ-f3: Moved 2 files to 'C:\tmp\f3'.
hvr_demo31-integ-f3: Waiting...
```

This indicates that the new file has been replicated to directories **/tmp/f2** and **/tmp/f3**. Look in these directories to confirm this.

Quick Start for HVR on Salesforce

This appendix shows how to set up two HVR channels, one for database to Salesforce (called **hvr_demo40**) and one for Salesforce to Database (called **hvr_demo41**). For simplicity, the database will be located on the same machine as the hub database.

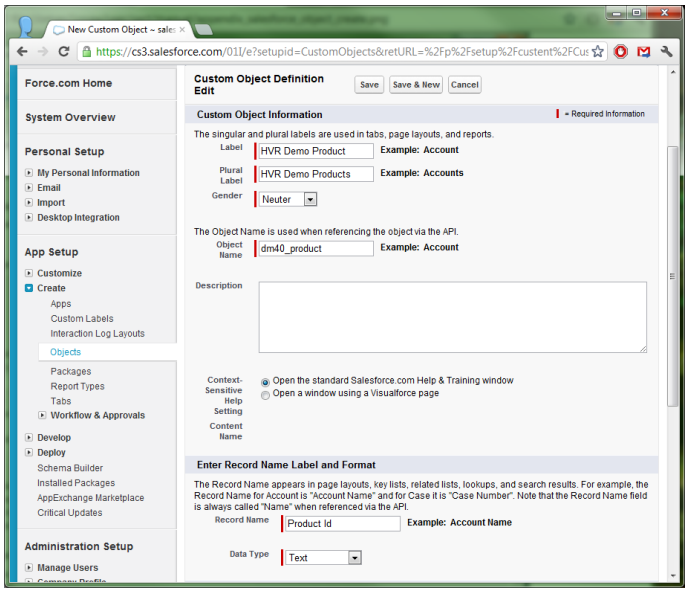


Create Test Schemas, Tables and Objects

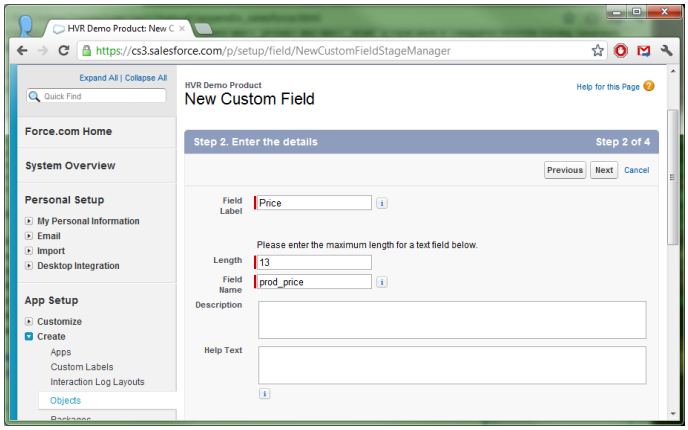
Create two test schemas, one for integration into Salesforce and the other for capture from Salesforce. Both of these schemas should contain two empty tables named **dm01_product** and **dm01_order**. If replication is configured between existing databases and tables then this step should be skipped.
First, create two schemas **testdb1** and **testdb2**. See the other Quick Start appendices for HVR on a specific database for instruction. For the test of this appendix, Oracle databases are assumed.
Create the test tables.

```
$ cd $HVR_HOME/demo/hvr_demo40/base/oracle
$ sqlplus testdb1/hvr < hvr_demo40.cre
$ sqlplus testdb1/hvr < hvr_demo40.mod
$ sqlplus testdb2/hvr < hvr_demo40.cre
$ sqlplus testdb2/hvr < hvr_demo40.mod
```

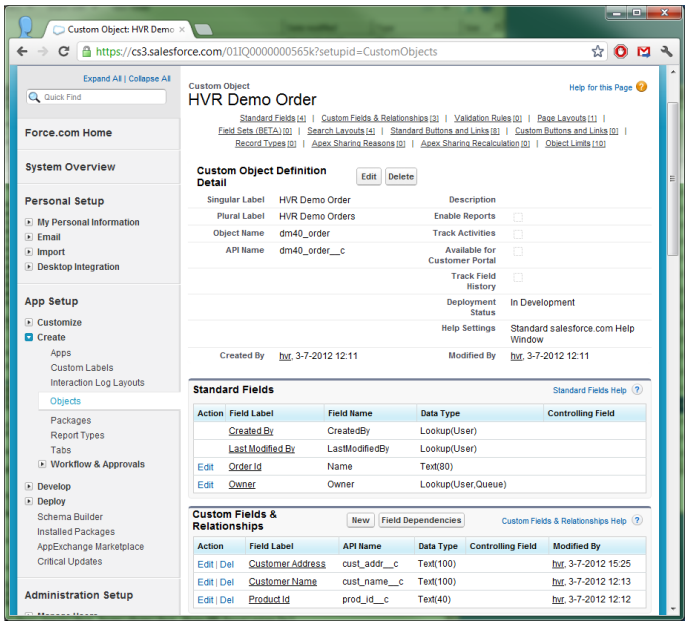
Now create the Salesforce objects:



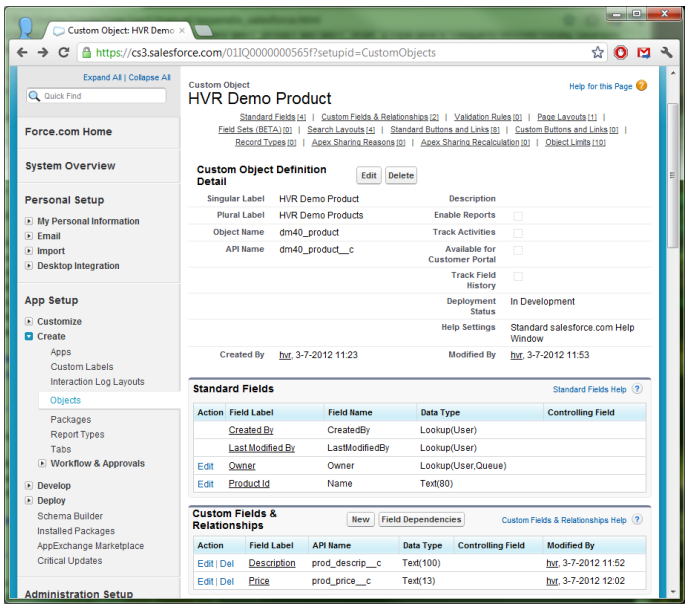
1. Log into Salesforce
2. Select menu item **Setup**
3. Select section **App Setup** ▶ **Create** ▶ **Objects**
4. Select button **New Custom Object**
5. Fill in object details
6. Select button **Custom Fields & Relationships** ▶ **New**
7. Fill in column details; use Text fields for this demo.



Create object **dm40_order** with custom fields **prod_id** and **cust_name**.



Also create object **dm40_product** with custom fields **prod_price** and **prod_descrip**.



Install Salesforce Data Loader

1. Log into www.salesforce.com
2. Select menu item **Setup**
3. Select section **Administrative Setup ▶ Data Management ▶ Data Loader**
4. Download the **ApexDataLoader.exe** and install.

Install HVR

First read section Introduction which explains the HVR's terminology and architecture. In particular this explains the importance of a hub database.
Then install the HVR software on the hub machine by following the installation steps in section New Installation on Unix or Linux or New Installation on Windows. If the hub machine is a Unix machine then HVR can either be installed on a Windows PC (so the HVR GUI can run on the PC and connect to the Unix hub machine) or the HVR GUI can be run on the Unix hub machine and connect back to an X server running on a PC.

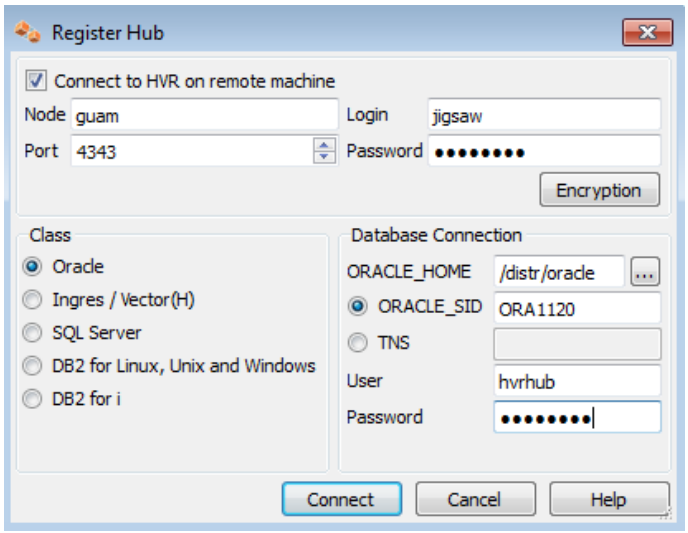
Create the Hub Database

Create the hub database, in which the HVR GUI will store the channel definition. This can be an Oracle schema, an Ingres database, a SQL Server database or a DB2 database. The steps to create an Oracle hub database schema are as follows:

```
$ sqlplus system/manager
SQL> create user hvrhub identified by hvr
2 default tablespace users
3 temporary tablespace temp
4 quota unlimited on users;
SQL> grant create session to hvrhub;
SQL> grant create table to hvrhub;
SQL> grant create sequence to hvrhub;
SQL> grant create procedure to hvrhub;
SQL> grant create trigger to hvrhub;
SQL> grant create view to hvrhub;
SQL> grant execute any procedure to hvrhub;
$ sqlplus
Enter user-name: / as sysdba
SQL> grant execute on dbms_alert to hvrhub;
SQL> exit;
```

Connect to Hub Database

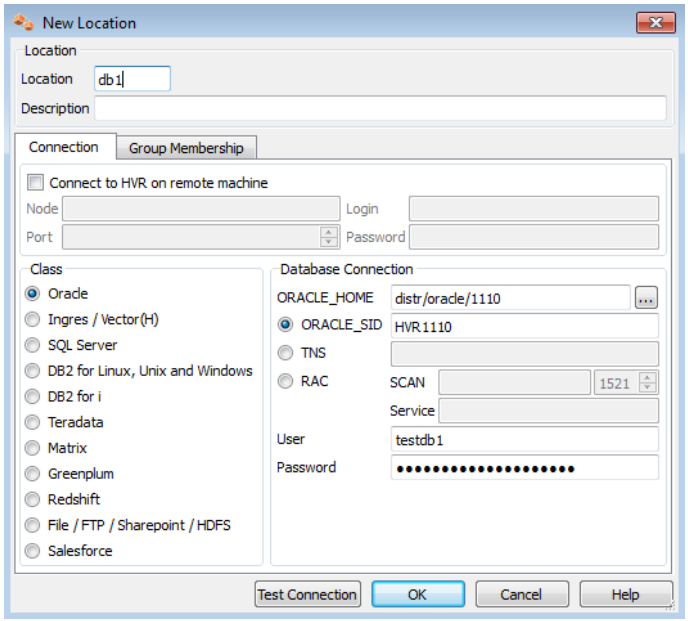
Start the HVR GUI on a PC by clicking on the HVR GUI icon (this is created by the HVR Installer for Windows) or by running **hvrgui** on Linux.
First, Register the hub database: right–click on **hub machines** ▶ **Register hub**. Enter connection details.



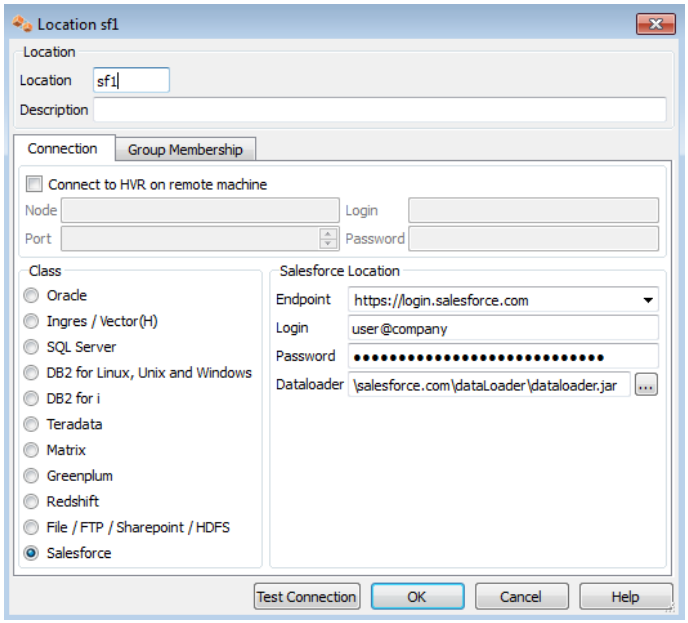
In this example the hub is a machine called **guam**, where an INET daemon is listening on port 4343. See section New Installation on Unix or Linux for how to configure this.
For a new hub database a dialog will prompt **Do you wish to create the catalogs?**; answer **Yes**.

Create Oracle Locations

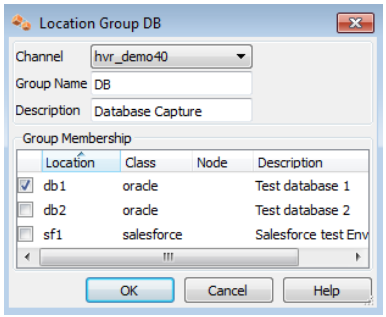
Next create two Oracle locations (one for each channel) using right–click on **Location Configuration** ▶ **New Location**. Ignore the **Group Membership** tab for now.



Create a Salesforce location using right–click on Location **Configuration** ▶ **New Location**.



The **Dataloader** field must point to the **.jar** file in the Data Loader installation directory. The default installation on windows is **C:\Program Files (x86)\salesforce.com\Data Loader**, and the name of the **.jar** file depends on the exact version.
Now define a channel using **Channel Definitions ▶ New Channel**.



The channel needs two location groups. Under the new channel: right–click on **Location Groups ▶ New Group**. Enter a group name, for example **DB**.
Add location **db1** as a member of this group by checking the box for **db1**.
Then create a second location group, called **SALESFORCE** that has member **sf1**.
The new channel also needs a list of tables to replicate. This can be done as follows; right–click on **Tables ▶ Table Select**.

1. Choose the **db1** location **▶ Connect**.
2. In the **Table Selection** window, click on both tables and click **Add**.
3. In new dialog **HVR Table Name** click **OK**.
4. Close the **Table Selection** window.

Define Actions

Finally the new channel needs two actions to indicate the direction of replication, and actions to map database table and column names to Salesforce Object and Field API names.
First we will specify the replication direction:

1. Right–click on group **DB ▶ New Action ▶ DbCapture**. Check **/LogBased** so that the channel will detect changes using the Oracle log file, instead of using Oracle database triggers.

Right–click on group **SALESFORCE ▶ New Action ▶ SalesforceIntegrate**. Check **/OnErrorSaveFailed**, this affects how replication errors are handled.
Note that the **Actions** pane only displays actions related to the objects selected in the left–hand pane. So

click on channel **hvr_demo40** to see both actions.
Next we will map the table and column names.

1. Right–click on group **SALESFORCE ▶ New Action ▶ TableProperties**. Select Table **dm40_order**. Check **/BaseName** and fill in the Salesforce API Name **dm40_order__c**. This name is case–sensitive.
2. Right–click on group **SALESFORCE ▶ New Action ▶ ColumnProperties**. Select Table **dm40_order**. Check **/Name**, click the **[...]** button and select **column cust_addr**. Check **/BaseName** and fill in the Salesfore API Name **cust_addr__c**. This name is case–sensitive.
3. Repeat this for the other tables and columns in the channel to the following tables.

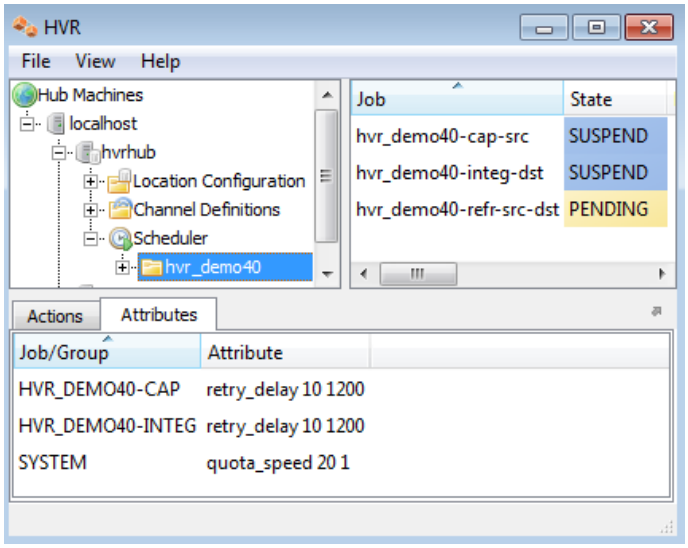
dm40_order	dm40_order__c
cust_addr	cust_addr__c
cust_name	cust_name__c
ord_id	Name
prod_id	prod_id__c

dm40_product	dm40_product__c
prod_descrip	prod_descrip__c
prod_id	Name
prod_price	prod_price__c

Note that action **SalesforceIntegrate** only integrates insert and update statements, not deletes.

Enable Replication with HVR Load

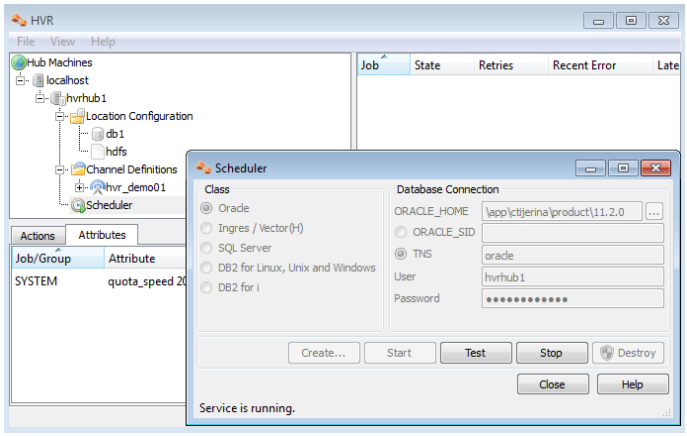
Now that the channel definition is complete, create the runtime replication system.
Right–click on channel **hvr_demo40 ▶ HVR Load**. Choose **Create or Replace Objects** and click **HVR Load**.



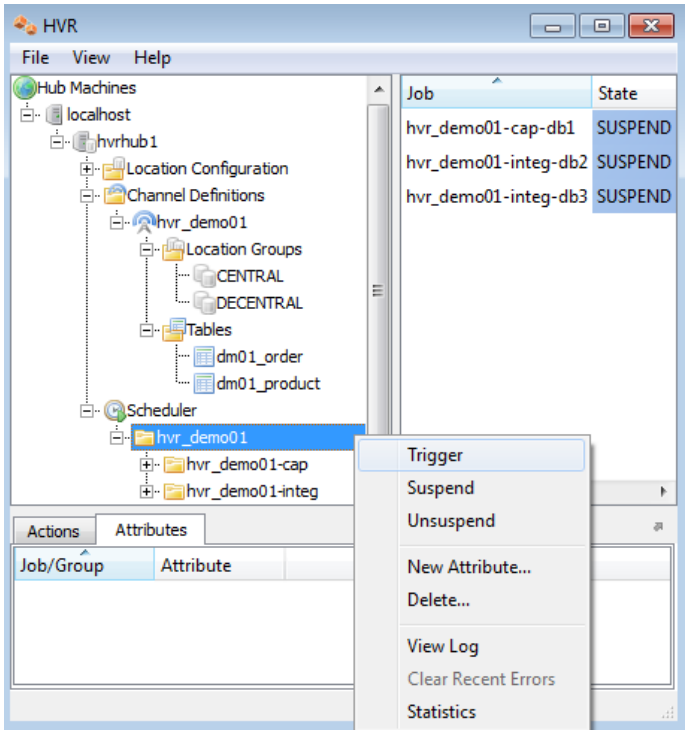
From the moment that HVR Load is done, all changes to database **testdb1** will be captured by HVR when its capture job looks inside the Oracle logging.
HVR Load also creates two replication jobs, which can be seen by looking at the **Scheduler** node of the hub database.

Start Scheduling of Replication Jobs

Start the Scheduler on the hub machine by clicking in the HVR GUI on the **Scheduler** node of the hub database.



Next, instruct the HVR Scheduler to trigger the replication jobs.

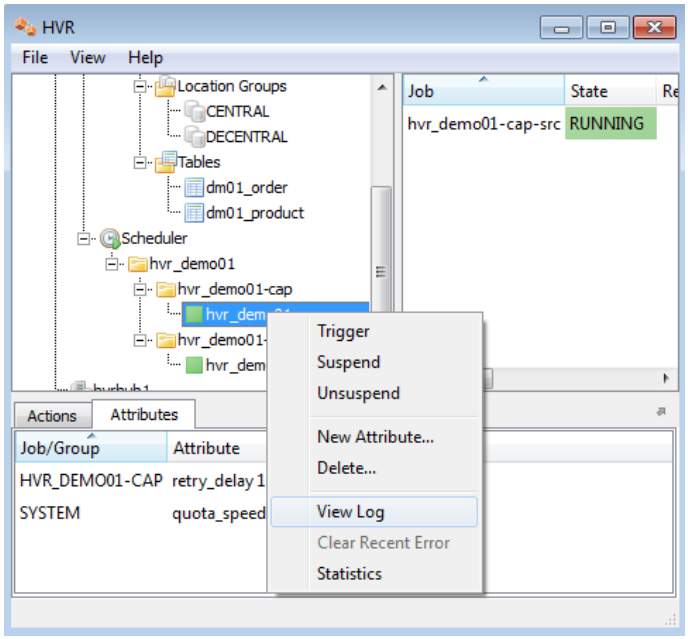


Test Replication

To test replication, make a change in **testdb1**:

```
$ sqlplus testdb1/hvr
SQL> insert into dm01_product values (1, 19.99, 'DVD');
SQL> commit;
```

In the HVR log file you can see the output of the jobs by clicking on **View Log**. This log file can be found in **$HVR_CONFIG/log/*hubdb*/hvr_demo01–cap–db1**.



The job output looks like this:

```
hvr_demo40-cap-db1: Capture cycle 1.
hvr_demo40-cap-db1: Scanned 1 transaction (188720 bytes) from 4 \
seconds ago containing 1 row (1 ins) for 1 table from file sequence 736 in \
0.52 seconds.
hvr_demo40-cap-db1: Routed 215 bytes (compression=50.0%) from 'db1' \
into 1 location.
hvr_demo40-cap-db1: Finished. (elapsed=1.08s)
hvr_demo40-integ-sf1: Integrate cycle 1 for 1 transaction file (215 \
bytes).
hvr_demo40-integ-sf1: Moved 1 file to \
'd:\data\dev\jggin\hvr_config\work\hvrhub\hvr_demo40\sf1\sf'.
hvr_demo40-integ-sf1[agent]: Processing 1 file...
hvr_demo40-integ-sf1[agent]: Uploading 1 rows for 'dm40_product' from 'sf-20120703152541860-
    dm40_product.csv'
hvr_demo40-integ-sf1[agent]: Uploaded 1 record from 'sf-20120703152541860-dm40_product.csv'
hvr_demo40-integ-sf1: Finished. (elapsed=21.71s)
hvr_demo40-cap_integ_all[eof]: Finished. (elapsed=22.84s)
```

This indicates that the jobs replicated the original change to Salesforce. You can verify the replication by looking at the Object data in Salesforce.

Create the Second Channel and Define Actions

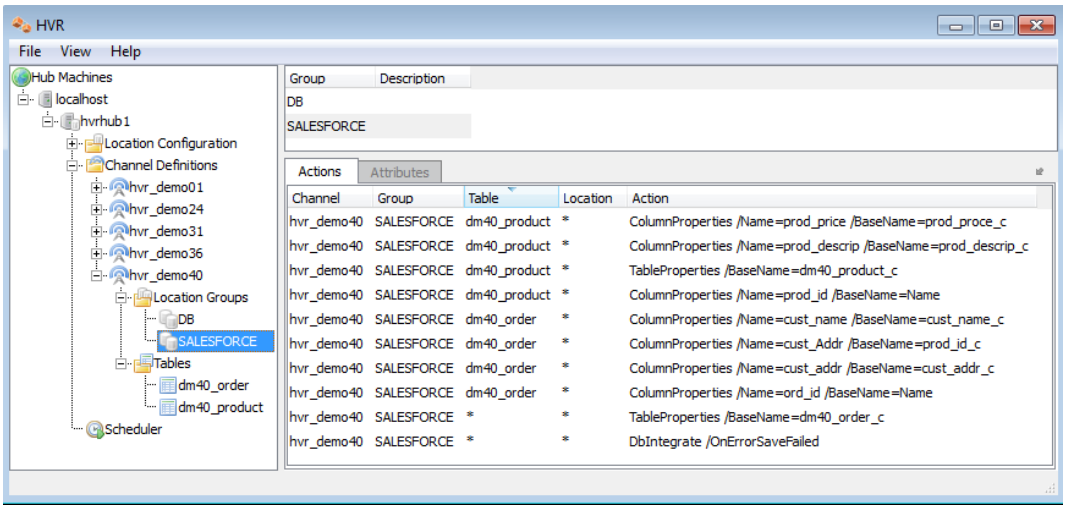
To retrieve data from Salesforce, we will construct a second channel. Follow the steps above to create a channel **hvr_demo41** with two groups; group **SALESFORCE** with group member location **sf1**, and group **DB** with group member locations **db2**.
First we will specify the replication direction:

1. Right–click on group **SALESFORCE** ▶ **New Action** ▶ **SalesforceCapture** Check **/UseBulkApi**, this will use the Salesforce Bulk API for efficient transfer of large datasets.
2. Right–click on group **DB** ▶ **New Action** ▶ **DbIntegrate**. Check **/ResilientInsert** to convert extra inserts to updates. Check **/OnErrorSaveFailed**, this affects how replication errors are handled.
   Note that HVR action **SalesforceCapture** reads all rows from the Salesforce source table instead of only capturing changes.

   Next we will map the table and column names, exactly in the same way we did with the channel **hvr_demo40** channel.

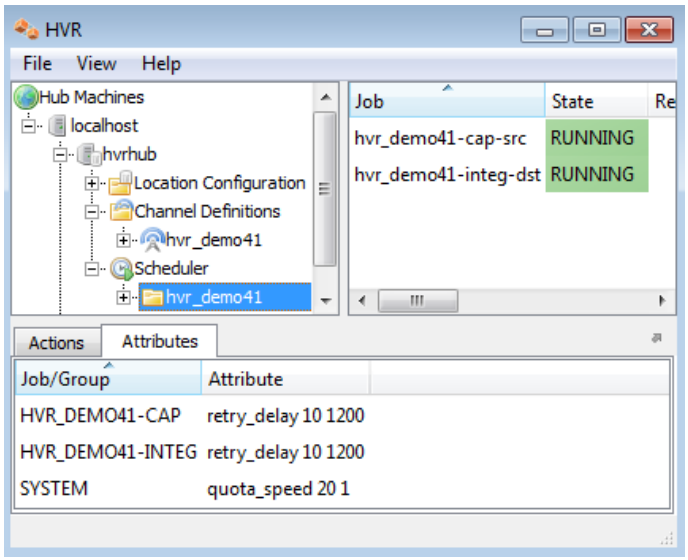
   Finally, we will specify scheduling actions. Since **SalesforceCapture** does not capture changes, all rows will be captured each capture cycle. We will specify a **Scheduling** action so the job will only run once a day.

3. Right–click on group **SALESFORCE ▶ New Action ▶ Scheduling Check /CaptureStartTimes** and click the [...] button. Select an appropriate start frequency, for example: Minute: 0, Hour: 0, and leave the other columns checked. This will result in a pattern of '0 0 * * *' which will schedule the job daily at midnight.



Enable Replication with HVR Load

Right–click on channel **hvr_demo41 ▶ HVR** Load, and load the channel.
Since the scheduler is already running, it does not need to be started again.
The job **hvr_demo41–cap–sf1** will remain **PENDING** until the scheduled start time.



Test Replication

To test the **hvr_demo41** channel, we will force a single cycle of the **PENDING** capture job by using **hvrtrigger**. It will capture all the data available in Salesforce, which is the data we inserted earlier with the **hvr_demo40** test.
In the HVR log file you can see the output of the jobs:

```
2013-07-04 10:34:41: hvrscheduler: localhost/9: Updated job hvr_demo41-cap-sf1 \
 with [Trigger=2].
2013-07-04 10:34:41: hvr_demo41-cap-sf1: Received 1 trigger control.
2013-07-04 10:34:56: hvr_demo41-cap-sf1[agent]: Processing 2 tables...
2013-07-04 10:35:16: hvr_demo41-cap-sf1[agent]: Downloaded 1 rows for 'dm40_product' to 'sf
    -20120704083457-dm40_product.csv'
2013-07-04 10:35:17: hvr_demo41-cap-sf1: Capture cycle 1 for 2 files (112 \
 bytes).
2013-07-04 10:35:18: hvr_demo41-cap-sf1: Routed 214 bytes \
```

```
 (compression=50.2%) from 'sf1' into 1 location.
2013-07-04 10:35:18: hvr_demo41-cap-sf1: Moved 2 files from location \
 'sf1'.
2013-07-04 10:35:18: hvr_demo41-cap-sf1: Processed 1 expired 'trigger' \
 control file.
2013-07-04 10:35:18: hvr_demo41-cap-sf1: Finished. (elapsed=37.26s)
2013-07-04 10:35:18: hvr_demo41-integ-db2: Integrate cycle 2 for 1 \
 transaction file (214 bytes).
2013-07-04 10:35:18: hvr_demo41-integ-db2: Integrated 1 change from \
 'dm40_product' (1 ins).
2013-07-04 10:35:18: hvr_demo41-integ-db2: Integrate used 1 transaction for 1 \
 individual row and took 0.14 seconds.
2013-07-04 10:35:18: hvr_demo41-integ-db2: Waiting...
```

You can verify the replication by looking at the target database:

```
$ sqlplus testdb2/hvr
SQL> select * from dm40_product;
  PROD_ID PROD_PRICE PROD_DESCRIP
---------- ---------- ----------------------------------------------------------------
         1      19.99 DVD
```

I

Quick Start for HVR Oracle into HDFS

This appendix shows how to set up an HVR channel (called **hvr_demo01**) to replicate from an Oracle Database into the Hadoop Distributed File System (HDFS). In this example HVR will replicate from a schema inside a single Oracle instance on the hub machine into HDFS. The steps below start by creating new users and tables for HVR to replicate between. In a real live situation tables and data likely already exist.

Create Test Schema and Tables

Generally when getting started with HVR a source schema with tables and data already exists. If so then this step can be skipped.
This Quickstart uses two empty tables named **dm01_product** and **dm01_order**. In an existing Oracle database, create a test schema and create the tables using the following commands.

```
$ sqlplus system/manager
SQL> create user testdb1 identified by hvr
 2  default tablespace users
 3  temporary tablespace temp
 4  quota unlimited on users;

SQL> grant create session to testdb1;
SQL> grant create table to testdb1;
SQL> grant create sequence to testdb1;
SQL> grant create procedure to testdb1;
SQL> grant create trigger to testdb1;
SQL> grant create view to testdb1;
SQL> grant execute any procedure to testdb1;
```

Create the test tables.

```
$ cd $HVR_HOME/demo/hvr_demo01/base/oracle
$ sqlplus testdb1/hvr < hvr_demo01.cre
$ sqlplus testdb1/hvr < hvr_demo01.mod
```

Install HVR on the hub

First read section Introduction which explains the HVR's terminology and architecture. In particular this explains the importance of a hub database.
Then install the HVR software on the hub machine by following the installation steps in section New Installation on Unix or Linux or New Installation on Windows. If the hub machine is a Unix machine then HVR can either be installed on a Windows PC (so the HVR GUI can run on the PC and connect to the Unix hub machine) or the HVR GUI can be run on the Unix hub machine and connect back to an X server running on a PC.
This Quickstart assumes the Oracle Database on the hub server is also the source database. Most real-time integration scenarios use log-based capture (parameter **/LogBased**). To enable log-base capture, configure

the following:

- The user name that HVR uses must be in Oracle's group. On Unix and Linux this can be done by adding the user name used by HVR to the line in **/etc/group** that begins with **dba**. On Windows right–click **My Computer** and select **Manage ▶ Local Users and Groups ▶ Groups ▶ ora_dba ▶ Add to Group ▶ Add**.
- The Oracle instance should have archiving 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 query **select destination, status from v$archive_dest.** By default, this will return values **USE_DB_RECOVERY_FILE_DEST, VALID**, which is inside the flashback recovery area. Alternatively, an archive destination can be defined with the following statement: **alter system set log_archive_dest_1='location=/disk1/arc'** and then restart the instance.

Install HVR to connect to Hadoop

In order to connect to Hadoop as a target HVR must be running on a Linux environment. This can be the HVR hub running on Linux, or HVR can be running as an agent on a Linux server to connect into HDFS, optionally on the Hadoop namenode. The HDFS interface requires a Java 7 Runtime Environment and Hadoop connectivity libraries.

Install the HVR software on the Linux machine by following the installation steps in section New Installation on Unix or Linux. For the Java Runtime environment and the Hadoop connectivity libraries there are two options.

- Install the HVR extras package. Download the package and install using the following command:

```
$ cd $HVR_HOME
$ tar -zxvf /tmp/hvr-4.6.2_13-linux_glibc2.5-x64-64bit-extras.tar.gz
```

This command creates the directory **$HVR_HOME/extras**.

- If Java 7 has been installed on the Linux environment and Hadoop Connectivity libraries are available then set the JAVA_HOME to reference the JRE/JDK home directory, and set the HADOOP_HOME to reference the Hadoop installation.

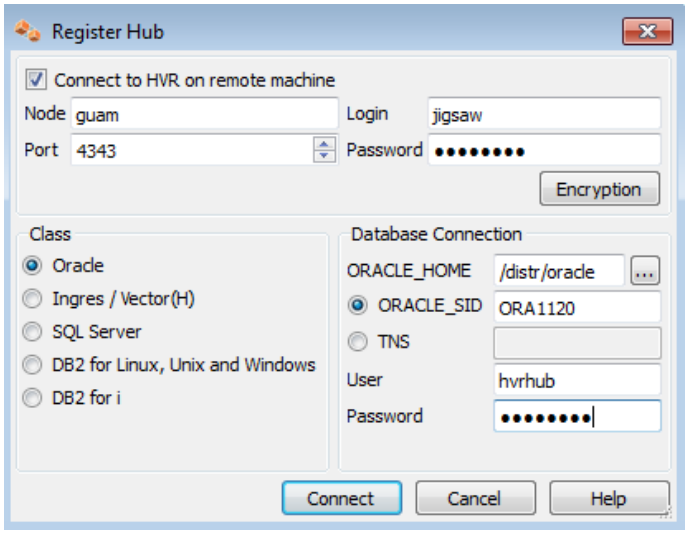
Create the Hub Database

Create the hub database, in which the HVR GUI will store the channel definition. This is actually another user/schema in the Oracle instance.

```
$ sqlplus system/manager
SQL> create user hvrhub identified by hvr
 2  default tablespace users
 3  temporary tablespace temp
 4  quota unlimited on users;

SQL> grant create session to hvrhub;
SQL> grant create table to hvrhub;
SQL> grant create sequence to hvrhub;
SQL> grant create procedure to hvrhub;
SQL> grant create trigger to hvrhub;
SQL> grant create view to hvrhub;
SQL> grant execute any procedure to hvrhub;
```

Connect To Hub Database

Start the HVR GUI on a PC by clicking on the HVR GUI icon (this is created by the HVR Installer for Windows) or by running **hvrgui** on Linux.
First, Register the hub database: right–click on **hub machines ▶ Register hub**. Enter connection details.

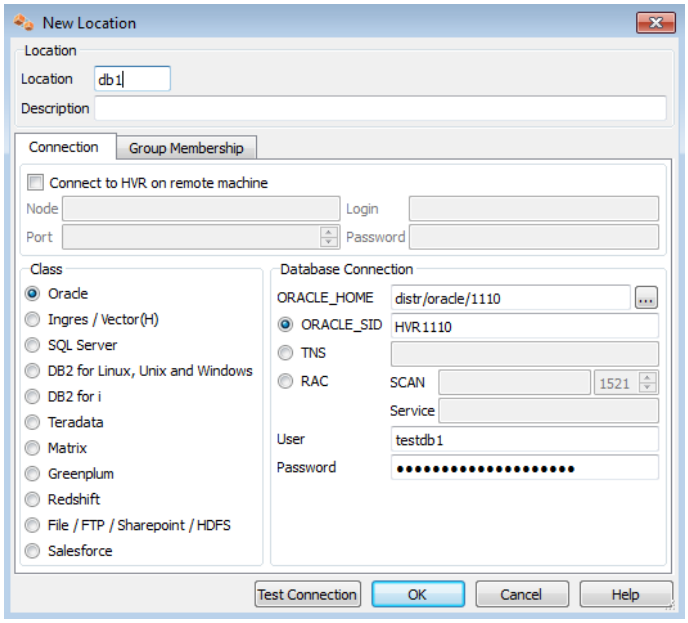


In this example the hub is a machine called **guam**, where an INET daemon is listening on port 4343. See section New Installation on Unix or Linux for how to configure this.

For a new hub database a dialog will prompt **Do you wish to create the catalogs?**; answer **Yes**.

Create Oracle Location

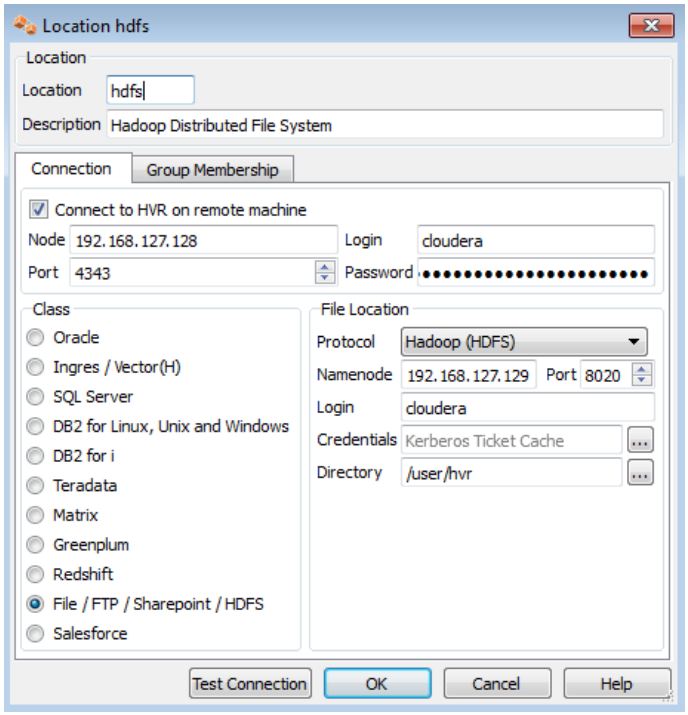
Next create a location for the Oracle source database using right–click on **Location Configuration ▸ New Location**.



In this example there is no need to check **Connect to HVR on remote machine** because **testdb1** is on the same machine as the hub.
Ignore the **Group Membership** tab for now.

Create HDFS Location

Create a location for HDFS using right–click on **Location Configuration ▸ New Location**.



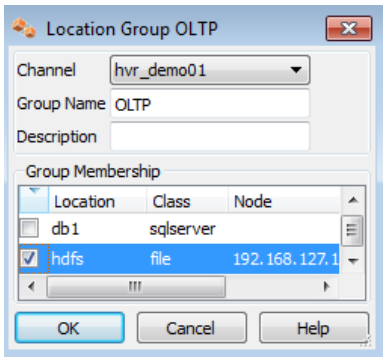
In this example **Connect to HVR on remote machine** is checked assuming this Linux environment is not the hub machine. If it is then it must not be checked.
Ignore the **Group Membership** tab for now.

Create a Channel

The next step is to create a channel. For a relational database the channel represents a group of tables that is captured as unit. Create a channel using right–click on **Channel Definitions** ▶ **New Channel**.
Choose any name you like.

Create Location Groups

The channel needs two location groups. Under the new channel: right–click on **Location Groups** ▶ **New Group**. Enter a group name (for instance **OLTP**).



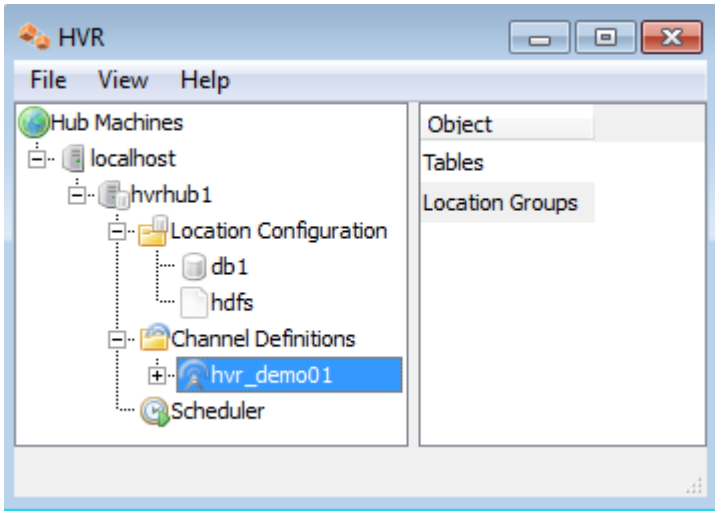
Add location **db1** as a member of this group by checking the box for **db1**.
Then create a second location group, called **DATALAKE** that has member **hdfs**.
The new channel also needs a list of tables to replicate. This can be done as follows; right–click on **Tables** ▶ **Table Select**.

- Choose the database location ▶ **Connect**.
- In the **Table Selection** window, click on both tables and click **Add**.
- In new dialog **HVR Table Name** click **OK**.
- Close the **Table Selection** window.

Define Actions

The new channel needs actions to define the replication.

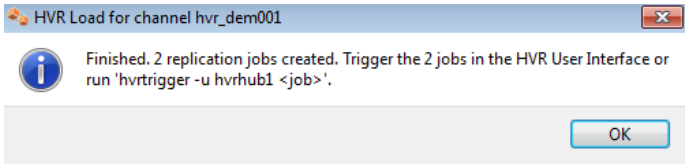
- Right–click on group **OLTP ▶ New Action ▶ DbCapture**. Check **/LogBased** so that the channel will detect changes using the log file, instead of using database triggers.
- Right–click on **Group HDFS ▶ New Action ▶ Transform**. Select Type **/Xml2Csv**, and optionally specify arguments to change the format of the output file.
- Right–click on **Group HDFS ▶ New Action ▶ FileIntegrate**. Check **/RenameExpression**, and put in **/{hvr_tbl_name}_{hvr_integ_tstamp}.csv** to create new files for every integrate cycle. Also add**ColumnProperty /Name=hvr_op_val /Extra /IntegrateExpression={hvr_op} /TimeKey** to include the operation type per record (0 for delete, 1 for insert, 2 for update). For a very busy system check the option **/Burst** to combine multiple transactions into one (larger) file per table.
- Right–click on **Group HDFS ▶ New Action ▶ Scheduling**. Check **/IntegrateStartTimes**, and select from the calendar. For example for a 10 minute refresh interval starting at the top of the hour check multiples of 10 for resulting **RefreshStartTimes /0,10,20,30,40,50 * * * *.**



Note that the Actions pane only displays actions related to the objects selected in the left–hand pane. So click on channel **hvr_demo01** to see both actions.

Enable Replication with HVR Load

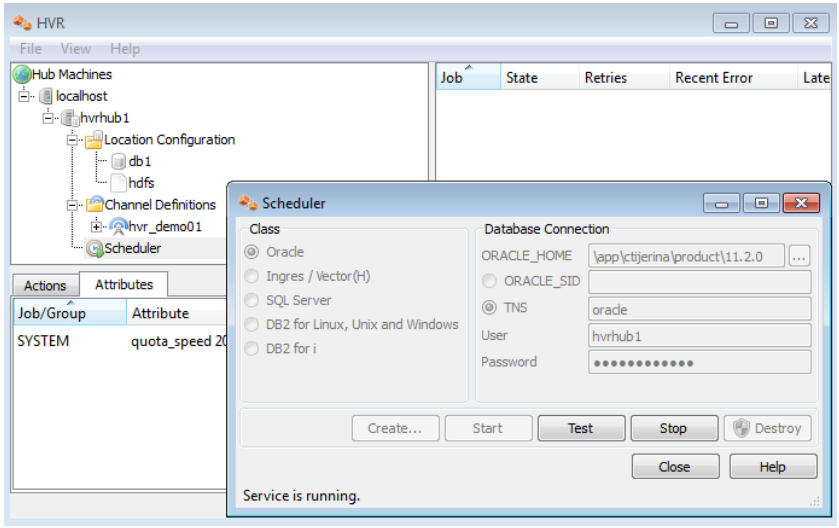
Now that the channel definition is complete, create the runtime replication system.
Right–click on channel **hvr_demo01 ▶ HVR Load**. Choose **Create or Replace Objects** and click **HVR Load**.



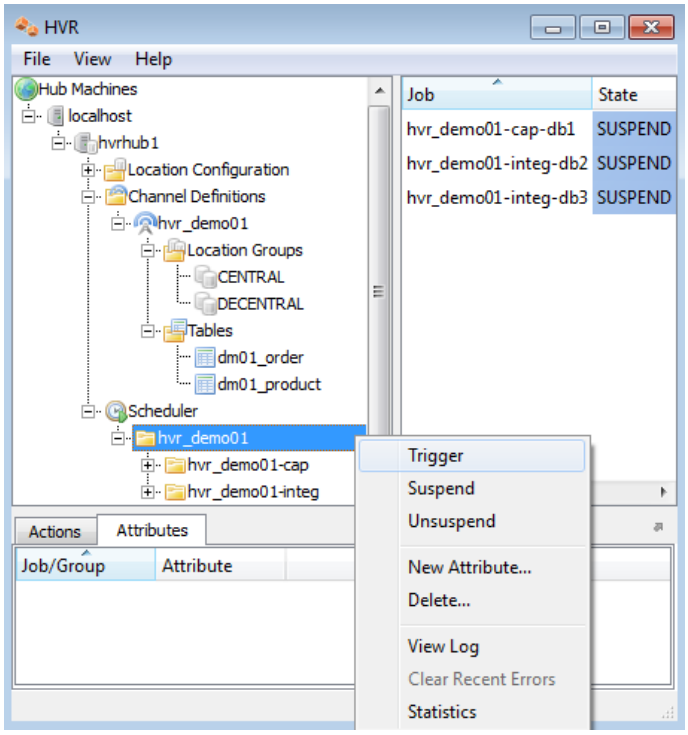
From the moment that HVR Load is done, all new transactions that start on the database **testdb1** will be captured by HVR when its capture job looks inside the transaction logs.
HVR Load also creates two replication jobs, which can be seen under the Scheduler node in the GUI.

Start Scheduling of Capture Job

Start the Scheduler on the hub machine by clicking in the HVR GUI on the **Scheduler** node of the hub database.



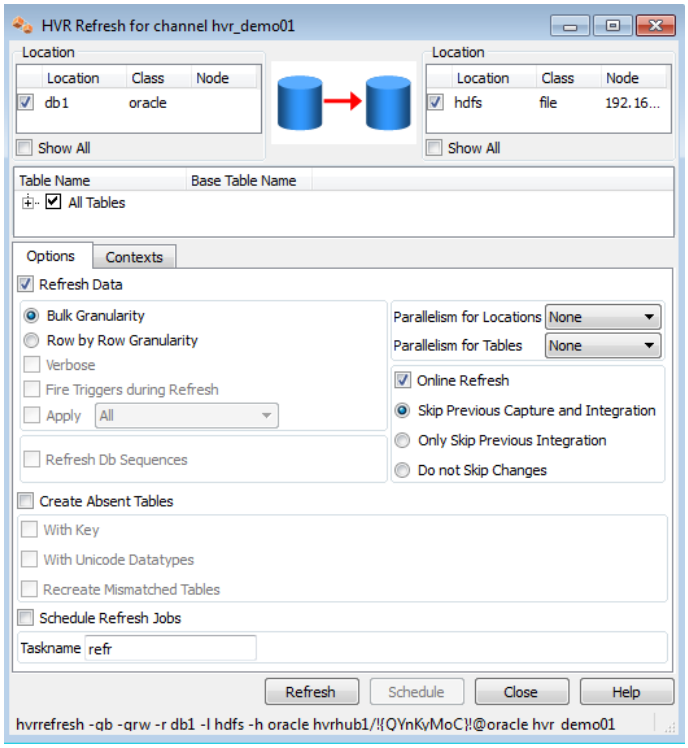
Next, instruct the HVR Scheduler to trigger the capture job to keep the capture job current with transaction log generation.



The capture job inside the Scheduler executes a script under **$HVR_CONFIG/job/hvrhub/hvr_demo01** that has the same name as the job. So job **hvr_demo01–cap–db1** detects changes on database **testdb1** and stores these as transactions files on the hub machine.

Perform Initial Load

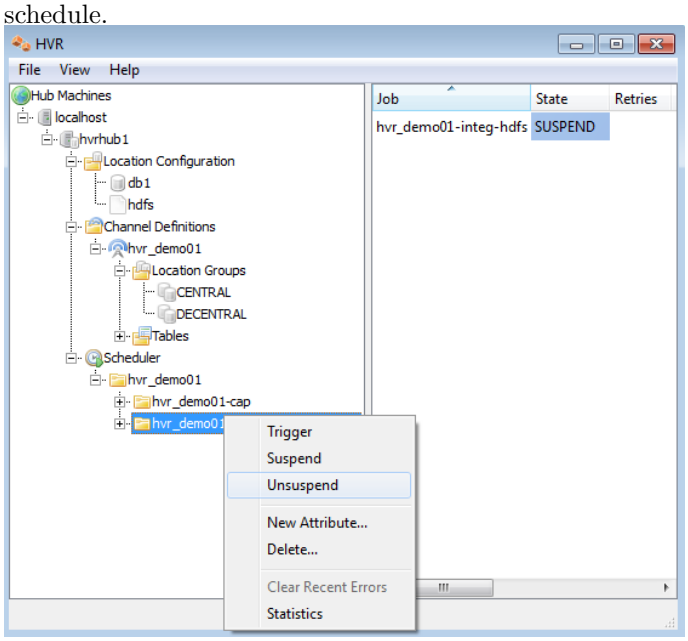
Perform the initial load from the OLTP database into HDFS using **HVR Refresh**. Right–click on the channel **hvr_demo01 ▶ HVR Refresh**.



Make sure the option for **Online Refresh** is checked and select **Skip Previous Capture and Integration**. Optionally set **Parallelism for Tables**.
Run the Refresh.

Start Scheduling of Capture Job

Instruct the HVR Scheduler to Unsuspend the integrate job to push changes into HDFS according to the defined schedule.



The integrate job inside the Scheduler executes a script under **$HVR_CONFIG/job/hvrhub/hvr_demo01** that has the same name as the job. So job **hvr_demo01–integ-hdfs** picks up transaction files on the hub on a defined schedule and creates files on HDFS containing these changes.
A scheduled job that is not running is in a **PENDING** state.

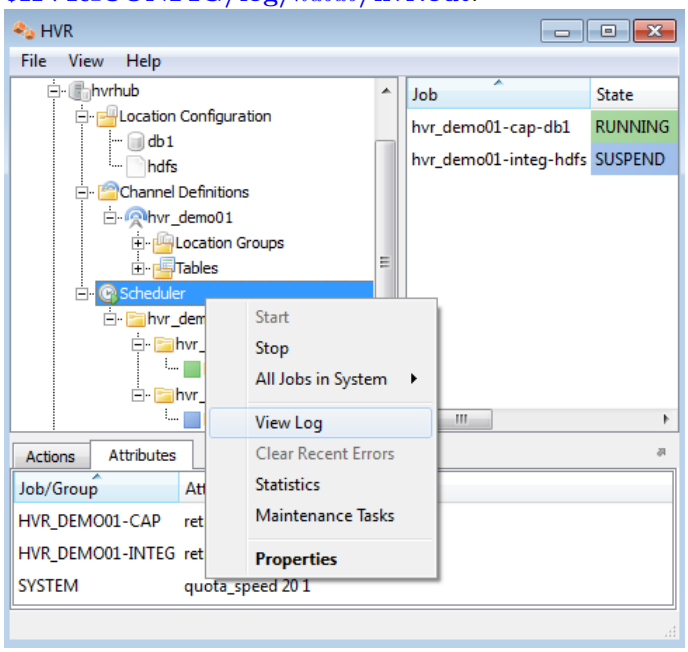
Test Replication

To test replication, make a change in **testdb1**:

```
testdb1/hvr
```

```
SQL> insert into dm01_product values (1, 19.99, 'DVD');
SQL> commit;
```

Next, instruct the HVR Scheduler to Trigger the integrate job rather than wait for the next scheduled run. In the HVR log file you can see the output of the jobs by clicking on **View Log**. This log file can be found in **$HVR_CONFIG/log/*hubdb*/hvr.out**.



The job output looks like this:

```
hvr_demo01-cap-db1: Scanned 1 transaction containing 1 row (1 ins) for 1 table.
hvr_demo01-cap-db1: Routed 215 bytes (compression=40.6%) from 'db1' into 2 locations.
hvr_demo01-cap-db1: Capture cycle 3.
hvr_demo01-integ-hdfs: Integrate cycle 1 for 1 transaction file (215 bytes).
hvr_demo01-integ-hdfs: Moved 1 file to 'hdfs://cloudera@192.168.127.128/user/hvr'.
hvr_demo01-integ-hdfs: Processed 1 expired 'trigger' control file.
hvr_demo01-integ-hdfs: Finished. (elapsed=4.04s)
```

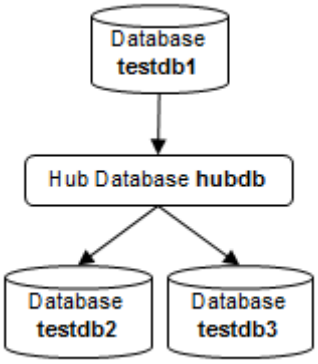
This indicates that the jobs replicated the original change to **HDFS**. A query on **HDFS** confirms this:

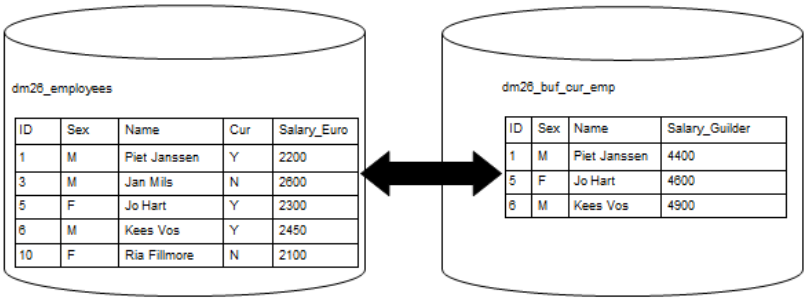
```
[cloudera@quickstart ~]$ hadoop fs -ls /user/hvr
Found 2 items
drwxr-xr-x   - cloudera supergroup          0 2015-02-06 10:38 /user/hvr/_hvr_state
-rw-r--r--   3 cloudera supergroup         12 2015-02-06 10:38 /user/hvr/
    dm01_product_20150206183858632.csv
[cloudera@quickstart ~]$ hadoop fs -cat /user/hvr/dm01_product_20150206183858632.csv

1,19.99,DVD
[cloudera@quickstart ~]$
```

J

Example Replication between Different Table Layouts

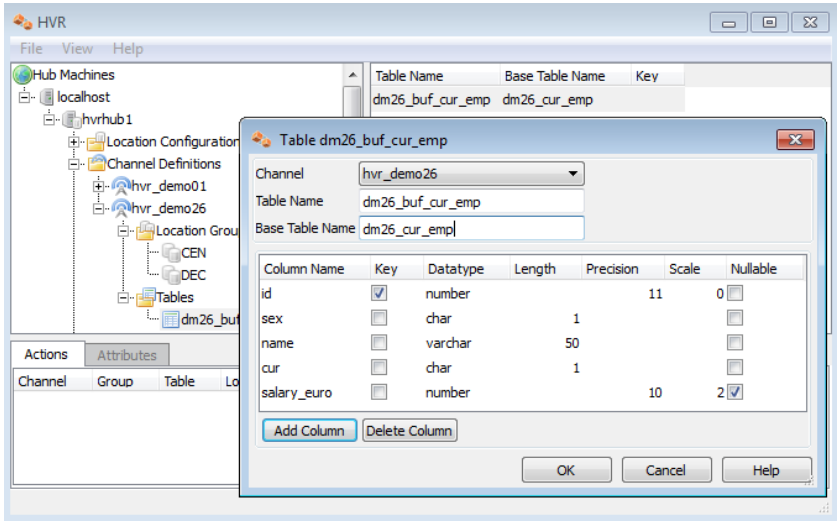
The employee table in a company's central database is called **dm26_employees** and contains both current employees and employees who have left the company; this is indicated with an extra column called **cur**. In the de–central database the employee table is called **dm26_buf_cur_emp** and only contains current employees. This version of table also contains an extra column for the employee's salary in the local currency.





The HVR channel is defined as follows: Action **DbCapture**, **DbIntegrate** and **CollisionDetect** are set on both sides for bi–directional replication. A **Restrict** action is defined on the central database to filter rows. The decentral database has a **TableProperties** action and two **ColumnProperties** actions which rename the table, hide one column and recalculate another.

The following is a screenshot of the channel definition to replicate between the tables.

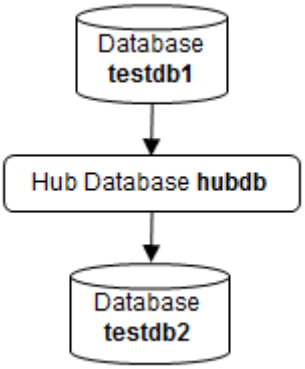


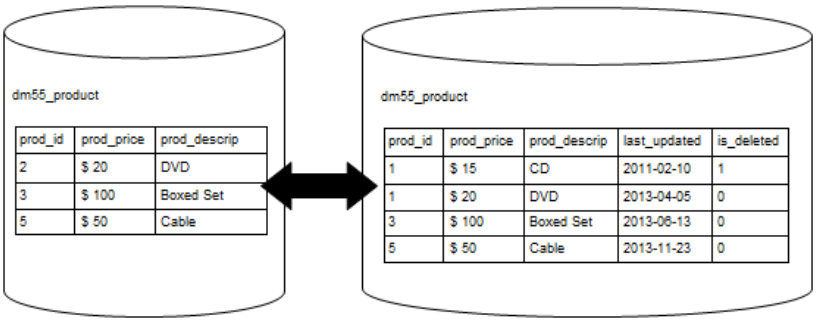
Data can be changed on either database and is replicated to the other database. Command **hvrrefresh** and **hvrcompare** can also be used for this channel; they will also add extra columns and filter out restricted rows as required. This channel can be found in directory **$HVR_HOME/demo/hvr_demo26**.

K

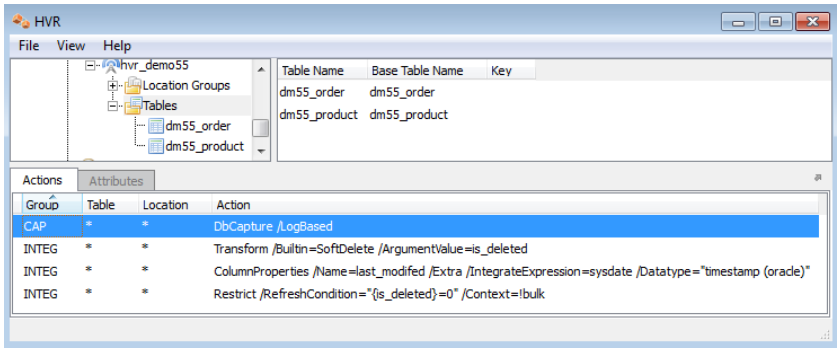
Example Replication into Data Warehouse

The tables of database **testdb1** must be replicated into a Data Warehouse staging tables. From these staging tables ETL processing will incrementally update renormalized tables. To assist this ETL step, HVR will add a timestamp column named **last_updated** and also a 'soft delete' column named **is_deleted** to each integrate table. Soft deletes means that HVR will convert deletes in the capture database into updates of the **is_deleted** column in the integrate database.





The HVR channel is defined as follows: Action **DbCapture** is defined on the source database with parameter **/LogBased**. On the target side action **DbIntegrate** is defined with parameter **/Burst**. This **/Burst** parameter boosts performance when delivering data into a data warehouse. The timestamp column is defined using action **ColumnProperties /Name=last_updated /Extra /Datatype=datetime /IntegrateExpression=sysdate**. The soft delete column is defined with action **Transform /Builtin=SoftDelete /ArgumentValue=is_deleted**. Compare and refreshes should typically ignore soft–deleted rows so an extra action is defined: **Restrict /RefreshCondition="is_deleted=0" /Context="!bulk"**.



The above actions are sufficient both for replication and compare and refresh of the source database with the data warehouse. HVR Refresh can also be used to create the data warehouse tables. The extra columns **last_updated** and **is_deleted** do not have to be included in HVR's information about each table, they are already implied by the **ColumnProperties** and **Transform** actions. This channel can be found in directory **$HVR_HOME/demo/hvr_demo55**.