HCL OneDB 2.0.1

Migrating and upgrading

Contents

Chapter 1. Migrating and upgrading

You can upgrade to the 2.0.0.0 version of HCL OneDB™ or migrate from other database servers to HCL OneDB™. Upgrading is an in-place migration method that uses your existing hardware and operating system software. Some changes to the HCL OneDB™ database server can affect upgrading from a previous release.

Upgrade tasks

To upgrade to HCL OneDB™ version 2.0.0.0:

1. Understand your migration path and plan your migration: Overview of HCL OneDB™ migration on page 3
2. Prepare for migration, including reviewing changes to HCL OneDB™ products since the release from which you are migrating: Preparing for migration on page 9
3. Do the migration tasks that are appropriate to your system: Migrating to the new version of HCL OneDB™ on page 20
4. Finish the migration process: Completing required post-migration tasks on page 25

HCL OneDB™ Migration Guide

The *HCL OneDB™ Migration Guide* describes how to move data manually between databases, servers, and computers.

These topics are intended for database server administrators or database administrators who are responsible for upgrading the database server or migrating data. These topics assume that you have the following background:

- A working knowledge of your computer, your operating system, and the utilities that your operating system provides
- Some experience with database server administration, operating-system administration, or network administration

Overview of HCL OneDB™ migration

Before you upgrade to the new version of HCL OneDB™, ensure that you understand the migration process and prerequisites.

The migration process

This overview of the migration process describes what you need to know to plan your migration and the resources that you can use to assist you.

Careful planning will ensure minimal impact on your business.

- Migration effort on page 3
- Migration skills on page 4
- Migration plans on page 4
- Types of migration on page 5
- Migration tools on page 5

Migration effort

Depending on your environment, the migration process can take a few hours or several weeks.

The migration effort is determined by many factors:

- Your current version of HCL Informix or HCL OneDB.
- The site architecture and configuration, before and after migration.
- The level of site customization, before and after migration.
- Integration of additional software products.
- To some extent, the size of the database.

Migration skills

Your HCL OneDB™ migration team needs database administration skills, system administration skills, and application programming skills.

The migration team needs:

- Database administration skills, to help migrate custom database extensions.
- System administration skills, to perform various system tasks. These tasks include operating system installation, configuration and maintenance and the installation and configuration of HCL OneDB™ and any additional software products.
- Application programming skills, to create and maintain scripts to evaluate and modify application programs.

If you prefer, highly-skilled HCL Services personnel and business partners are available to assist you in migrating your environment. Contact your HCL representative for further information.

Migration plans

Before you begin to migrate to a new version of the database server, you should plan for migration.

To plan your migration requirements, complete these tasks:

1. Inventory the existing HCL OneDB™ environment assets, such as machines, instances, databases, database customization, custom code, HCL software, and third-party software.
2. Itemize the requirements for the post-migrated environment. New requirements can include upgrading or adding hardware, using new features, or replacing custom-code with new built-in function.
3. Plan the migration activities. Typical activities include:
    - Performing a level-0 backup of the database.
    - Quiescing the database server and preventing connections to the database until migration completes.

      Important: Any connection attempts (for example, from cron jobs or monitoring scripts) to the database after you quiesce the database server and during migration will cause migration to fail.
    - Installing the new version of HCL OneDB™.
    - Migrating database data.
    - Migrating applications before using them with the new database server.

Depending on your environment, you might need to perform some of these activities more than once. You might not need to restore the level-0 backup; however, if you encounter problems you can always restore the backup of your current server.

Types of migration

There are two ways to migrate to HCL OneDB™ Version 2.0.0.0.

Upgrading (In-place migration)

Upgrading is a special case of migration that uses your existing hardware and operating system. You install a new or improved version of the product in a different location from your current version on the same machine. You can copy your configuration file and add new parameters. When you start the new HCL OneDB™ instance, the database data is automatically converted. For example, you can upgrade from Version 1.0.0.0 to Version 2.0.0.0.

Migrating from a non-OneDB / Informix database

The process of moving your data from another database management system (DBMS) such as Oracle or Sybase SQL Anywhere to HCL OneDB™ Version 2.0.0.0. This type of migration is especially useful if you are currently using various products. You can consolidate to take advantage of the HCL OneDB™ features and total cost of ownership.

If you have a high-availability cluster with one or more secondary database servers or if you use Enterprise Replication, you follow additional procedures to upgrade your servers.

Migration tools

- For in-place upgrades: You do not use any migration tools. Simply start the server by using the **oninit** utility. The data from the source database server is converted to the target database server automatically.

Upgrading HCL OneDB™ (in-place migration)

In-place migration upgrades HCL OneDB™ directly to the current version by installing the product in a new directory on the same computer, copying a few configuration files, and starting the new server to automatically convert your database data.

In-place migration uses your existing test and production hardware. The operating system on those computers must be supported by the new version of the server. Also, you must have enough space for the system database data conversion.

Upgrading consists of these steps:

1. Prepare your system. That includes closing all transactions, verifying the integrity of the data with the oncheck utility, and performing a level-0 backup. (If you are using Enterprise Replication or high-availability clusters, you must stop replication and perform required extra tasks.)

   If you remove in-place alter operations before you upgrade, you might speed up the upgrade process.
2. **UNIX:** Log in to the database server as the **root** user or, for a non-root installation, log in as the user who owns the database server.

3. **Windows:** Log in to the database server as a member of the Administrator's group or, for a non-root installation, log in as the user who owns the database server.
4. Install the new product on the computer.

   Important: The safest way to upgrade to a new version is to install the new version in another directory. To save space, you can select the product components that you want to install. After you test the database server instance with the similar configuration settings and the client connection information that you use for your current database server, you can remove the old version.

   If you do not have space on your computer for two versions, install the new version over the existing version. In this case, you cannot selectively install components; you must install the whole product to block objects from the previous version. Before you choose this approach, make sure that you have the original installation media for the old version because you cannot automatically revert to it.

   Note: Ensure that the desired edition is installed.

5. Copy the appropriate configuration files, such as the `onconfig` file and the `sqlhosts` file, to the new `$ONEDB_HOME/etc` directory. Set parameters that are new for the current release.
6. Set the **$ONEDB_HOME** environment variable to the directory of the new installation.
7. Start an instance of the new version of HCL OneDB™. The database data is automatically converted.
8. If you notice performance problems, run UPDATE STATISTICS and UPDATE STATISTICS FOR PROCEDURE operations.

This type of migration minimizes the risk of introducing errors. .

Hardware and operating system requirements

Ensure that you meet the operating system and hardware requirements for HCL OneDB™ 2.0.0.0.

Before you upgrade or migrate to the current version of HCL OneDB™, ensure that the system you choose meets the necessary operating system, hardware, disk, and memory requirements.

Your hardware must support the operating systems that HCL OneDB™ Version 2.0.0.0 supports, and it must provide enough disk space for the database server and all your other software applications.

Check the machine notes for information about the operating-system patches that you need for successful installation and operation of the database server. Follow any platform-specific instructions in the machine notes.

UNIX, Linux: You might have to change some of the kernel parameters for your UNIX™ or Linux™ operating system before you install HCL OneDB™ Version 2.0.0.0. Refer to the kernel-configuration instructions for your operating system.

Fix pack naming conventions

OneDB releases and fix packs contain version names that appear in the format `n1.n2.n3.n4`.

In this format:

- `n1` = major release number
- `n2,n3,n4` = minor fix pack level

For example, in Version 1.0.1.0 **1** is the major release number, **0.1.0** is the fix pack level.

Overview of moving data

If you are installing the new version of the database server on another computer or operating system (non-in-place migration), you can use one of several tools and utilities to move data from your current database server.

For example, suppose you migrated to the current version of HCL OneDB™ and created a few new databases, but decide to revert to the previous version. Before you revert, you can use one of the data-migration tools to save the data you added. After reverting, you can reload the data.

Before you move data, consider these issues:

- Changes in the configuration parameters and environment variables
- Amount of memory and dbspace space that is required
- Organization of the data
- Whether you want to change the database schema to accommodate more information, to provide for growth, or to enhance performance

For information about how to move data between database servers on different operating systems, also see Migrating database servers to a new operating system on page 30.

Prerequisites before moving data

Before you use any data migration utility, you must set your PATH, ONEDB_HOME, and ONEDB_SERVER environment variables.

For information about environment variables, see the *HCL OneDB™ Guide to SQL: Reference*.

Moving data between computers and dbspaces

You can move data between different computers, and you can import data from environments other than the HCL OneDB™ database server. Except when you use external tables, you must unload your data to ASCII files before you move the data to another computer.

If you are moving data into the HCL OneDB™ database server on another computer, you can use the **dbimport** and **dbload** utilities to load the data that you exported.

If you are moving data to an application that is not based on HCL OneDB™, you might need to use the UNLOAD statement because you can specify the delimiter that is used in the data files.

Importing data from a non-HCL OneDB™ source

The **dbimport** and **dbload** utilities can import data from any ASCII file that is properly formatted.

Most applications that produce data can export the data into files that have a suitable format for **dbimport**. If the format of the data is not suitable, use UNIX™, Linux™, or Windows™ utilities to reformat the data before you import it.

Moving data by using distributed SQL

If you want to move data with different binary pages and page sizes across platforms and you have expertise in using distributed SQL, you can use INSERT and SELECT SQL statements to transfer the data.

Important: Do not use INSERT and SELECT statements to move data if the database contains BLOB data types.

Prerequisites: A network connection must exist between database server instances.

To move data using INSERT and SELECT statements with fully qualified table names:

1. Capture the complete database schema from the source database server.
2. Alter the extent sizing and, if necessary, the lock modes on tables from page to row.
3. Create and verify the schema on the target database server.
4. Disable logging on both source and target servers where necessary.
5. Create and run the following scripts:
    a. Create and run separate scripts for:
        - Disabling select triggers on the source server
        - Disabling indexes, triggers and constraints for each table on the target database server.
    b. Create and run one script per table for the fully-qualified INSERT and SELECT statements.

        For example:

        ```
        INSERT INTO dbname@target:owner.table SELECT *
        FROM dbname@source:owner.table
        ```

        You can run the scripts in parallel. In addition, for larger tables, you can create multiple scripts that can partition the table to run in parallel.
    c. Create and run separate scripts for enabling indexes, triggers and constraints for each table
6. Run UPDATE STATISTICS on system catalog tables and stored procedures and functions on the target database server.
7. Adjust starting values for all tables that have serial columns on the target database server.
8. Turn on transaction logging on the source and target database servers.
9. Return the source and target database servers to multi-user mode.
10. Validate the data that was transferred to the target database server.

For information about INSERT and SELECT statements, refer to the *HCL OneDB™ Guide to SQL: Syntax*. For information on distributed transactions, refer to the *HCL OneDB™ Administrator's Guide* and the *HCL OneDB™ Administrator's Reference*.

Migration to OneDB 2.0.0.0

Preparing for migration to OneDB 2.0.0.0

Before you install the new version of HCL OneDB™, you must prepare the database server environment for migration by performing specified pre-migration tasks. If you are also migrating from 32-bit to 64-Bit database servers, you must perform additional tasks.

Preparing for migration

Preparing for migration includes gathering information about and backing up your data, so that you can reinstall the previous version of the server and restore your data if you have a migration problem. Preparing for migration is crucial for successful migration.

Before you begin

- If you use Enterprise Replication, you must first prepare your replication environment for migration. For more information, see Enterprise Replication and migration on page 19.

About this task

Review and complete all tasks that apply:

1. Reviewing changes in HCL OneDB product functionality on page 9.
2. Checking and configuring available space on page 10.
3. Configuring for recovery of restore point data in case an upgrade fails on page 11.
4. Renaming user-defined routines (UDRs) that have the following names: CHARINDEX() , LEFT(), RIGHT(), INSTR(), DEGREES(), RADIANS(), REVERSE(), SUBSTRING_INDEX(), LEN(), and SPACE().
   These names are reserved for built-in SQL string manipulation functions.
5. Adjusting settings:
   a. If you use UNICODE, ensure that the GL_USEGLU environment variable on the source server is set to the same value as the GL_USEGLU environment variable on the target server.
6. Saving copies of the current configuration files on page 12.
7. Saving a copy of the storage manager sm_versions file on page 13.
8. Closing all transactions and shutting down the source database server on page 14.
9. Initiating fast recovery to verify that no open transactions exist on page 15.
10. Verifying the integrity of the data on page 15.
11. Verifying that the database server is in quiescent mode on page 16.
12. Making a final backup of the source database server on page 16.
13. Verifying that the source database server is offline on page 17.

Reviewing changes in HCL OneDB™ product functionality

About this task

Changes in functionality in HCL OneDB™ 2.0.0.0 can potentially impact your applications, scripts, maintenance processes, and other aspects that are related to your database server environment.

Changes to functionality that was introduced before HCL OneDB™ 2.0.0.0 can also affect your plans.

Checking and configuring available space

Before you migrate to the new version of HCL OneDB™, you must make sure that you have enough available space for the new server, your data, and any other network and data tools that you use.

During migration, HCL OneDB™ drops and then recreates the **sysmaster** database. Depending on which version of HCL OneDB™ you migrate from, the **sysmaster** database in the current version can be significantly larger.

When you migrate to Version 2.0.0.0, you need the following space for building **sysmaster**, **sysutils**, and **sysadmin** databases:

- 21892 KB of logical-log spaces (or 10946 pages) for 2 K page platforms
- 26468 KB of logical-log spaces (or 6617 pages) for 4 K page platforms

During migration, a second database, the **sysadmin** database, is created in the **root** dbspace. As you work after migrating, the **sysadmin** database, could grow dramatically. You can move the **sysadmin** database to a different dbspace.

You might need to increase the physical log size to accommodate new features, and you might consider adding a new chunk.

If your migration fails because there is insufficient space in the partition header page, you must unload your data before you try to migrate again. Then you must manually load the data into the new version.

The root chunk should contain at least ten percent free space when converting to the new version of the server.

In some cases, even if the database server migration is successful, internal conversion of some databases might fail because of insufficient space for system catalog tables. For more information, see the release notes for this version of HCL OneDB™.

Add any additional free space to the system prior to the migration. If the dbspaces are nearly full, add space before you start the migration procedure. When you start the new version of HCL OneDB™ on the same root dbspace of the earlier database server, HCL OneDB™ automatically converts the **sysmaster** database and then each database individually.

For a successful conversion of each database, ensure that 2000 KB of free space per database is available in each dbspace where a database resides.

To ensure enough free space is available:

1. Calculate the amount of free space that each dbspace requires.

   In the following equation, *n* is the number of databases in the dbspace and *X* is the amount of free space they require:

   ```
   X kilobytes free space = 2000 kilobytes * n
   ```

The minimum number of databases is 2 (for the **sysmaster** and **sysadmin** databases).

2. Check the amount of free space in each dbspace to determine whether you need to add more space.

You can run SQL statements to determine the free space that each dbspace requires and the free space available. These statements return the free-space calculation in page-size units. The **free_space_req** column value is the free-space requirement, and the **free_space_avail** column value is the free space available.

The following SQL statement shows how to determine the free space that each dbspace requires:

```
DATABASE sysmaster;
SELECT partdbsnum(partnum) dbspace_num,
       trunc(count(*) * 2000) free_space_req
   FROM sysdatabases
GROUP BY 1
ORDER BY 1;
```

The following SQL statement queries the **syschunks** table and displays the free space available for each dbspace:

```
SELECT dbsnum dbspace_num, sum(nfree) free_space_avail
   FROM syschunks
GROUP BY 1
ORDER BY 1;
```

Important: If less free space is available than the dbspace requires, either move a table from the dbspace to another dbspace or add a chunk to the dbspace.

The dbspace estimates could be higher if you have an unusually large number of SPL routines or indexes in the database.

Configuring for recovery of restore point data in case an upgrade fails

By default, the CONVERSION_GUARD configuration parameter is enabled and a temporary directory is specified in the RESTORE_POINT_DIR configuration parameter. These configuration parameters specify information that OneDB can use if an upgrade fails. You can change the default values of these configuration parameters before beginning an upgrade.

Before you begin

Prerequisites: The directory specified in the RESTORE_POINT_DIR configuration parameter must be empty before the upgrade begins, but not when recovering from a failed update.

Important:

After a failed upgrade, do not empty the RESTORE_POINT_DIR directory before you attempt to run the onrestorept utility. The server must be offline after a failed upgrade.

About this task

You can change the value of the CONVERSION_GUARD configuration parameter or the directory for restore point files before beginning an upgrade. The default value for the CONVERSION_GUARD configuration parameter in the ONCONFIG file is (2), and the default directory where the server will store the restore point data is `$ONEDB_HOME/tmp`. You must change this information before beginning an upgrade. You cannot change it during an upgrade.

To change information:

1. If necessary for your environment, change the value of the CONVERSION_GUARD configuration parameter.

   When the CONVERSION_GUARD configuration parameter is set to `2` (the default value), the server will continue the upgrade even if an error related to capturing restore point data occurs, for example, because the server has insufficient space to store the restore point data.

   However, if the CONVERSION_GUARD configuration parameter is set to `2` and the upgrade to the new version of the server fails, you can use the **onrestorept** utility to restore your data.

   However, if you set the CONVERSION_GUARD configuration parameter to `2`, conversion guard operations fail (for example, because the server has insufficient space to store restore point data), and the upgrade to the new version fails, you cannot use the **onrestorept** utility to restore your data.
2. In the RESTORE_POINT_DIR configuration parameter, specify the complete path name for a directory that will store restore point files.

   The server will store restore point files in a subdirectory of the specified directory, with the server number as the subdirectory name.

What to do next

If the CONVERSION_GUARD configuration parameter is set to `1` and an upgrade fails, you can run the **onrestorept** utility to restore the OneDB instance back to its original state just before the start of the upgrade.

If the CONVERSION_GUARD configuration parameter is set to `1` and conversion guard operations fail (for example, because the server has insufficient space to store restore point data), the upgrade to the new version will also fail.

If any restore point files from a previous upgrade exist, you must remove them before you begin an upgrade.

Even if you enable the CONVERSION_GUARD configuration parameter, you should still make level 0 backup of your files in case you need to revert after a successful upgrade or in case a catastrophic error occurs and you cannot revert.

Saving copies of the current configuration files

Save copies of the configuration files that exist for each instance of your source database server. Keep the copies available in case you decide to use the files after migrating or you need to revert to the source database server.

Although you can use an old ONCONFIG configuration file with HCL OneDB™ Version 2.0.0.0, you should use the new Version 2.0.0.0 ONCONFIG file, or at least examine the file for new parameters. For information about Version 2.0.0.0 changes to the ONCONFIG file, see Configuration parameter changes by version.

Configuration files that you might have are listed in Table 1: Configuration files to save from the source database server on page 13.

Table 1. Configuration files to save from the source database server

UNIX™ or Linux™	Windows™
$INFORMIXDIR/etc/$ONCONFIG	**%INFORMIXDIR%\etc\%ONCONFIG%**
$INFORMIXDIR/etc/onconfig.std	**%INFORMIXDIR%\etc\onconfig.std**
$INFORMIXDIR/etc/oncfg*	**%INFORMIXDIR%\etc\oncfg***
$INFORMIXDIR/etc/sm_versions	**%INFORMIXDIR%\etc\sm_versions**
$INFORMIXDIR/aaodir/adtcfg	**%INFORMIXDIR%\aaodir\adtcfg.***
$INFORMIXDIR/dbssodir/adtmasks	**%INFORMIXDIR%\dbssodir\adtmasks.***
$INFORMIXDIR/etc/sqlhosts or **$INFORMIXSQLHOSTS**	**%INFORMIXDIR%\etc\sqlhosts** or **$INFORMIXSQLHOSTS**
$INFORMIXDIR/etc/tctermcap	
$INFORMIXDIR/etc/termcap	

If you use ON-Bar to back up your source database server and the logical logs, you must also save a copy of any important storage manager files and the following file:

UNIX™ or Linux™:

$INFORMIXDIR/etc/ixbar.servernum

Windows™:

%INFORMIXDIR%\etc\ixbar.servernum

The HCL OneDB™ Primary Storage Manager does not use the `sm_versions` file. If you plan to use the OneDB® Primary Storage Manager, you do not need the `sm_versions` file. However, if you use the Tivoli® Storage Manager or a third-party storage manager, you do need the `sm_versions` file.

Use directory as **OneDB_HOME** for the new database server, copy `sm_versions` to the new **$OneDB_HOME/etc**, or copy `sm_versions.std` to `sm_versions` in the new directory, and then edit the `sm_versions` file with appropriate values before starting the migration.

Saving a copy of the storage manager sm_versions file

Before you migrate to a later version of the database server, save a copy of your current **sm_versions** file, which should be in the **$INFORMIXDIR/etc** directory.

The HCL OneDB™ Primary Storage Manager does not use the **sm_versions** file. If you plan to use the OneDB® Primary Storage Manager, you do not need the **sm_versions** file. However, if you use the Tivoli® Storage Manager or a third-party storage manager, you do need the **sm_versions** file.

If you are using a different directory as **INFORMIXDIR** for the new database server, copy **sm_versions** to the new **$INFORMIXDIR/etc**, or copy **sm_versions.std** to **sm_versions** in the new directory, and then edit the **sm_versions** file with appropriate values before starting the migration.

For information about how to install and use the HCL OneDB™ Storage Manager, see the .

Preparing 12.10 BSON columns with DATE fields for upgrade

Before you upgrade from HCL OneDB™ 12.10 you must unload binary JSON (BSON) columns with DATE fields into JSON format so that you can load them into HCL OneDB™ 14.10

About this task

Perform the following steps on the 12.10 server.

1. Create an external table with a similar name as the original table and with JSON (instead of BSON) format for the date.
   For example, assume that the original table named **datetab** has a BSON column named i that has DATE fields in it. Use the following statement to create an empty, external table named **ext_datetab** that has a JSON column with DATE fields. The DATAFILES clause specifies the location and name of the delimited data file, which in this example is `disk:/tmp/dat.unl`.

```
create external table ext_datetab (j int, i json) using
        (datafiles ("disk:/tmp/dat.unl"),
        format "delimited");
```

2. Unload the data from the original table into the external table.
   For example:

```
insert into ext_datetab select j, i::json from datetab;
```

What to do next

Complete other pre-migration steps. After you upgrade to the new server, you must load the JSON columns with DATE fields from the external table into a new table in BSON format.

Closing all transactions and shutting down the source database server

Before migrating, terminate all database server processes and shut down your source database server. This lets users exit and shuts down the database server gracefully. If you have long running sessions, you must also shut those down.

Inform client users that migration time is typically five to ten minutes. However, if migration fails, you must restore from a level-0 backup, so ensure that you include this possibility when you estimate how long the server will be offline.

Before you migrate from the original source database server, make sure that no open transactions exist. Otherwise, fast recovery will fail when rolling back open transactions during the migration.

To let users exit and shut down the database server gracefully

1. Run the **onmode –sy** command to put the database server in quiescent mode.
2. Wait for all users to exit.
3. Run the **onmode –l** command to move to the next logical log.
4. Run the **onmode -c** to force a checkpoint.
5. Make a level-0 backup of the database server.

6. Run the **ontape -a** command after the level-0 backup is complete.
7. Run the **onmode –yuk** command to shut down the system.

If you need to perform an immediate shutdown of the database server, run these commands:

```
onmode -l
onmode -c
onmode -ky
```

Initiating fast recovery to verify that no open transactions exist

A shutdown procedure does not guarantee a rollback of all open transactions. To guarantee that the source database server has no open transactions, put the source database server in quiescent mode and initiate fast recovery.

Run the following command to enter quiescent mode and initiate a fast recovery:

```
oninit -s
```

UNIX/Linux Only

On UNIX™ or Linux™, the **oninit-s** command rolls forward all committed transactions and rolls back all incomplete transactions since the last checkpoint and then leaves a new checkpoint record in the log with no open transactions pending.

You must run the **oninit -s** command before you initialize the new version of HCL OneDB™. If any transactions remain when you try to initialize the new database server, the following error message appears when you try to initialize the new database server, and the server goes offline:

```
An open transaction was detected when the database server changed log versions.
Start the previous version of the database server in quiescent mode and then shut
down the server gracefully, before migrating to this version of the server.
```

For more information about fast recovery, see your *HCL OneDB™ Administrator's Guide*.

After you put the database server in quiescent mode and initiate fast recovery, issue the **onmode -yuk** command to shut down the database server. Then review the **online.log** file for any possible problems and fix them.

Only after proper shutdown can you bring the new database server (HCL OneDB™ Version 2.0.0.0) through the migration path. Any transaction that is open during the migration causes an execution failure in fast recovery.

Verifying the integrity of the data

After verifying that no open transactions exist, verify the integrity of your data by running the **oncheck** utility. You can also verify the integrity of the reserve pages, extents, system catalog tables, data, and indexes. If you find any problems with the data, fix the problems before you make a final backup of the source database server.

To obtain the database names, use the following statements with DB-Access:

```
DATABASE sysmaster;
SELECT name FROM sysdatabases;
```

Alternatively, to obtain the database names, run the **oncheck -cc** command without any arguments and filter the result to remove unwanted lines, as shown in this example:

```
oncheck -cc | grep "ting database"
```

Table 2: Commands for verifying the data integrity on page 16 lists the **oncheck** commands that verify the data integrity.

Table 2. Commands for verifying the data integrity

Action	oncheck Command
Check reserve pages	**oncheck -cr**
Check extents	**oncheck -ce**
Check system catalog tables	**oncheck -cc** ***database_name***
Check data	**oncheck -cD** ***database_name***
Check indexes	**oncheck -cI** ***database_name***

Verifying that the database server is in quiescent mode

Before you make a final backup, verify that your source database server is in quiescent mode.

Run the onstat - command to verify that the database server is in quiescent mode.

The first line of the onstat output shows the status of your source database server. If the server is in quiescent mode, the status line includes this information:

```
Quiescent -- Up
```

Making a final backup of the source database server

Use ON-Bar or **ontape** to make a level-0 backup of the source database server, including all storage spaces and all used logs. After you make a level-0 backup, also perform a complete backup of the logical log, including the current logical-log file.

Be sure to retain and properly label the tape volume that contains the backup.

Important: You must also make a final backup of each source database server instance that you plan to convert.

For ON-Bar, remove the **ixbar** file, if any, from the **$INFORMIXDIR%/etc** or **%INFORMIXDIR%\etc** directory after the final backup. Removing the **ixbar** file ensures that backups for the original source database server are not confused with backups about to be done for the new database server. Follow the instructions regarding expiration in your storage manager documentation.

For more information about making backups, see the *HCL OneDB™ Backup and Restore Guide*.

Verifying that the source database server is offline

Before you install the new database server, verify that the source database server is offline. You must do this because the new database server uses the same files.

You cannot install the new database server if any of the files that it uses are active.

You can also use the onstat utility to determine that shared memory was not initialized.

Pre-migration checklist of diagnostic information

Before you migrate to a newer version of HCL OneDB™, gather diagnostic information, especially if you have large, complex applications. This information will be useful to verify database server behavior after migration. This information will also be useful if you need help from HCL Software Support.

If you have problems, you or HCL Software Support can compare the information that you gather with information obtained after migration.

The following table contains a list of the diagnostic information that you can gather. You can print the checklist. Then, after you get the information specified in each row, check the second column of the row.

Table 3. Checklist of information to get before migrating

Information to Get Before Migrating	Done
Get the SQL query plans for all regularly used queries, especially complex queries, by using SET EXPLAIN ON.	
Run the **dbschema -d -hd** command for all critical tables. The output contains distribution information.	
Get **oncheck -pr** output that dumps all of the root reserved pages.	
Make a copy of the ONCONFIG configuration file. A copy of the ONCONFIG file is essential if you need to revert to an earlier version of the database server. In addition, a copy of this file is useful because **oncheck -pr** does not dump all of the configuration parameters.	
Prepare a list of all the environment variables that are set using the **env** command.	
During times of peak usage: • Obtain an **online.log** snippet, with some checkpoint durations in it • Run **onstat -aF, -g all**, and **-g stk all**.	

Table 3. Checklist of information to get before migrating (continued)

Information to Get Before Migrating	Done
During times of peak usage, run the following **onstat** commands repeatedly with the **-r repeat** option for a period of about three to five minutes: • **onstat -u**, to see the total number of **sqlexecs** used • **onstat -p**, for read and write cache rates, to detect deadlocks and the number of sequential scans • **onstat -g nta**, a consolidated output of **-g ntu**, **ntt**, **ntm** and **ntd** • **onstat -g nsc**, **-g nsd**, and **-g nss** for the status of shared memory connections • **onstat -P**, **-g tpf**, and **-g ppf** • **vmstat**, **iostat** and **sar**, for cpu utilization • **timex** of all queries that you regularly run	

Migrating from 32-bit to 64-bit database servers

If you are migrating from a 32-bit version of OneDB to a 64-bit version of OneDB or reverting from a 64-bit version of OneDB, you might need to follow additional steps to update certain internal tables.

These steps are documented in the platform-specific machine notes that are provided with your database server.

For 32- to 64-bit migrations, change SHMBASE and STACKSIZE according to the `onconfig.std` configuration file for the new version.

All UDRs and DataBlade® modules that were built in 32-bit mode must be recompiled in 64-bit mode because they will not work with the 64-bit database server. If you have any UDRs that were developed in 32-bit mode, make sure that proper size and alignment of the data structures are used to work correctly on a 64-bit computer after recompiling in 64-bit mode. For more information, refer to the machine notes.

Migrating from 32-bit to 64-bit with collection types that use the SMALLINT data type

If you are moving your database from a 32-bit computer to a 64-bit computer and your database contains collection types that use the SMALLINT data type, you must take extra steps to prevent memory corruption. Collection types are the ROW, LIST, SET, and MULTISET data types. This restriction applies if you are upgrading from an older version of HCL OneDB™ on 32-bit to the current version of HCL OneDB™ on 64-bit, or if you are moving from 32-bit to 64-bit on the current version of HCL OneDB™.

To migrate a database with SMALLINT collection types from 32-bit to 64-bit, use one of the following methods:

- Export and import the data.
    1. Export the data from the 32-bit computer.
    2. Import the data onto the 64-bit computer.
- Drop and recreate specific collection types and database objects.

1. Drop the collection types that use the SMALLINT data type and all other database objects that reference them (such as tables, columns, SPL routines, triggers, indexes, and so on).
2. Recreate the collections types and all the other necessary database objects.

Enterprise Replication and migration

You must coordinate the migration of all servers that are involved in data replication.

These topics describe the additional tasks that you must perform when migrating to and reverting from OneDB Version 2.0.0.0 if you are running Enterprise Replication.

Preparing to migrate with Enterprise Replication

If you use Enterprise Replication, you must do replication-related tasks to prepare for migration.

Before you begin

You must do all migration operations as user **informix**, unless otherwise noted.

Only a DBSA can run the cdr check queue command. With a non-root installation, the user who installs the server is the equivalent of the DBSA, unless the user delegates DBSA privileges to a different user.

To prepare for migration with Enterprise Replication:

1. Stop applications that are performing replicable transactions.
2. Make sure that the replication queues are empty.

   If you are migrating from HCL Informix 14.10. or later releases, run the following commands to check for queued messages and transactions:

   a. cdr check queue -q cntrlq *targetserver*
   b. cdr check queue -q sendq *targetserver*
   c. cdr check queue -q recvq *targetserver*
3. Make sure that the replication queues are empty by running the following commands:
   a. Run onstat -g grp to ensure that the Enterprise Replication grouper does not have any pending transactions. The grouper evaluates the log records, rebuilds the individual log records in to the original transaction, packages the transaction, and queues the transaction for transmission.
   b. Run onstat -g rqm to check for queued messages.
4. Shut down Enterprise Replication by running the cdr stop command.

Results

Now you can complete the steps in Preparing for migration on page 9 and, if necessary, in Migrating from 32-bit to 64-bit database servers on page 18.

Migrating with Enterprise Replication

If you use Enterprise Replication, you must complete replication-related tasks when you migrate to a new version of OneDB.

To migrate with Enterprise Replication:

1. Complete the steps in Preparing to migrate with Enterprise Replication on page 19.
2. Complete the steps in Preparing for migration on page 9.
3. Perform all migration operations as user **informix**.
4. Run cdr cleanstart command to start Enterprise Replication for the first time. This command forces Enterprise Replication to recreate its internal queue tables in new format.

Migrating to OneDB 2.0.0.0

When you migrate to HCL OneDB™ 2.0.0.0, you must complete required migration and post-migration tasks.

Migrating to the new version of HCL OneDB™

After you prepare your databases for migration, you can migrate to the new version of HCL OneDB™.

Before you begin

- Read the release notes and the machine notes for any new information.
- Complete the steps in Preparing for migration on page 9.

About this task

To upgrade a HCL OneDB™ non-root installation, you must run the installation program as the same user who installed the product being upgraded.

If you are migrating the database server from a version that does not support label-based access control, users who held the DBA privilege are automatically granted the SETSESSIONAUTH access privilege for PUBLIC during the migration process. For more information about SETSESSIONAUTH, see the *HCL OneDB™ Guide to SQL: Syntax*. For information about label-based access control, see the *HCL OneDB™ Security Guide*.

Important: Do not connect applications to a database server instance until migration has successfully completed.

Review and complete all tasks that apply:

1. Installing the new version of HCL OneDB on page 21.
2. Setting environment variables on page 22.
3. Customizing configuration files on page 22.
4. Adding Communications Support Modules.
5. Installing or upgrading any DataBlade modules on page 22.
6. Starting the new version of HCL OneDB on page 23.

Results

When the migration starts, the `online.log` displays the message `Conversion from version <version number> Started.`. The log continues to display start and end messages for all components. When the migration of all components is complete, the message `Conversion Completed Successfully` appears. For more information about this log, see Migration status messages on page 21.

What to do next

After migration, see Completing required post-migration tasks on page 25 for information about preparing the new server for use.

If the log indicates that migration failed, you can either:

- Install the old database server and restore your database from a level-0 backup.
- Run the onrestorept utility to back out of the upgrade and restore files to a consistent state without having to restore from a backup. You can run this utility only if you set the configuration parameters that enable the utility. See Restoring to a previous consistent state after a failed upgrade on page 24.

If the conversion of the High-Performance Loader **onpload** database failed, upgrade the **onpload** database. For more information, see Upgrading the High-Performance Loader onpload database on page 23.

Installing the new version of HCL OneDB™

Install and configure the new version of HCL OneDB™.

If possible, migrate on a database server dedicated to testing your migration before you migrate on your production database server.

Decide whether to upgrade on the same computer, known as an in-place migration, or an a different computer, known as a non-in-place migration and follow the appropriate process.

Important: Monitor the database server message log, `online.log`, for any error messages. If you see an error message, solve the problem before you continue the migration procedure.

Migration status messages

When the migration starts, the **online.log** displays the message "Conversion from version <*version number*> Started." The log continues to display start and end messages for all components.

When conversions of all components are complete, the message "Conversion Completed Successfully" displays. This message indicates that the migration process completed successfully, but it does not guarantee that each individual database was converted successfully. The message log might contain more information about the success or failure of the migration of each individual database. If migration of a particular database fails, then try to connect to the database to find out the exact cause of the failure.

At the end of the migration of each individual database, HCL OneDB™ runs a script to update some system catalog table entries. The message log includes messages that are related to this script. The success or failure of the script does not prevent the usage of a database.

For information about any messages in the message log, see the *HCL OneDB™ Administrator's Guide*.

Setting environment variables

After you install the current version of HCL OneDB™, verify that the **ONEDB_HOME**, **ONEDB_SERVER**, **ONCONFIG**, **PATH**, and **ONEDB_SQLHOST** (if used) environment variables are set to the correct values.

UNIX™ or Linux™: The client application looks for the **sqlhosts** file in the **etc** directory in the **ONEDB_HOME** directory. However, you can use the **ONEDB_SQLHOST** environment variable to change the location or name of the **sqlhosts** file.

The setting of the GL_USEGLU environment variable must match between the source and target server during migration.

Important: Before you start the Version 2.0.0.0 database server, you must set the DBONPLOAD environment variable to the name of the **pload** database if the name is not **onpload**, the default name.

For information about environment variables, see the *HCL OneDB™ Guide to SQL: Reference*.

Customizing configuration files

When you initialize the new version of HCL OneDB™, which contains a new `onconfig.std` file, use the same configuration that the old database server used. After you observe the performance of new version, you can examine the new file for new configuration parameters that you might want to use and can you start to use the new and changed configuration parameters.

Set the ALARMPROGRAM configuration parameter to either nothing or **no_log.sh** to prevent the generation of errors if the logical log fills during the migration. For more details, see Starting the new version of HCL OneDB on page 23. After the migration, change the value of ALARMPROGRAM to **log_full.sh**.

If the ALARMPROGRAM configuration parameter is set to the script **alarmprogram.sh**, set the value of BACKUPLOGS in **alarmprogram.sh** to `N`.

Important: To facilitate migration (and reversion), use the same values for your new database server for ROOTOFFSET , ROOTSIZE, and ROOTPATH that you used for the old database server. Also, keep the same size for physical logs and logical logs, including the same number of logical logs, and the same **sqlhosts** file.

If you use custom-code files with the High-Performance Loader, set the HPL_DYNAMIC_LIB_PATH configuration parameter in the **plconfig** file to the location of the shared library. For example, the location of the shared library might be **$INFORMIXDIR/lib/ipldd11a.*SOLIBSUFFIX***, where *SOLIBSUFFIX* is the shared-library suffix for your operating system.

For information about how to configure HCL OneDB™, see your *HCL OneDB™ Administrator's Guide*. For information about how to tune the configuration parameters, see the *HCL OneDB™ Performance Guide*.

Installing or upgrading any DataBlade® modules

After you install the new version of HCL OneDB™, you might need to install or upgrade any DataBlade® modules that you want to add to the database server.

Register the DataBlade® modules after you initialize the database server.

When you install HCL OneDB™, the built-in extensions, such as TimeSeries, are installed and registered automatically. You do not need to perform any actions to upgrade these DataBlade® modules, nor do you need to unload and load any data during migration.

Important: If the sysadmin database does not exist or the Scheduler is turned off, automatic registration does not occur. You must register each extension that you want to use by running the SYSBldPrepare() function.

Starting the new version of HCL OneDB™

After installing the new database server, start the server. Do not perform disk-space initialization, which overwrites whatever is on the disk space.

Prerequisite: If you installed HCL OneDB™ as user **root**, you must switch to user **informix** before starting the server.

Important: HCL OneDB™ writes to the logical logs with the transactions that result from creating the **sysmaster** database. If you run out of log space before the creation of the **sysmaster** database is complete, HCL OneDB™ stops and indicates that you must back up the logical logs. After you back up the logical logs, the database server can finish building the **sysmaster** database. You cannot use ON-Bar to back up the logical logs because the database has not been converted yet. If you have ALARMPROGRAM set to **log_full.sh** in the ONCONFIG configuration file, errors are generated as each log file fills during the migration. Set the value of ALARMPROGRAM to either nothing or **no_log.sh** so that these errors are not generated. If your logical log does fill up during the migration, you must back it up with **ontape**, the only backup tool you can use at this point. Issue the **ontape -a** command.

Start the new version of HCL OneDB™ for the first time by running **oninit** command on UNIX™ or by using the **Service** control application on Windows™.

As HCL OneDB™ starts for the first time, it modifies certain disk structures. This operation should extend the initialization process by only a minute or two. If your disks cannot accommodate the growth in disk structures, you will find a message in the message-log file that instructs you to run an oncheck command on a table. The oncheck utility will tell you that you need to rebuild an index. You must rebuild the index as instructed.

Upgrading the High-Performance Loader onpload database

If **onpload** conversion failed during database server migration, you can manually upgrade the **onpload** database.

Starting with Version 9.40.xC3, HCL OneDB™ has a new version of the **onpload** database with longer column lengths. The **onpload** database now requires slightly more disk space than it did before Version 9.40.xC3.

When you migrate to a new version of HCL OneDB™, you must also upgrade the **onpload** database.

To upgrade the onpload database:

1. If you are upgrading from a version of HCL OneDB™ that is prior to Version 9.40, run the **conploadlegacy.sh** script, as shown in this example:

   ```
   conploadlegacy.sh 7.31 9.40
   ```

2. If you are upgrading from a version of the database server that is prior to Version 9.40.xC3, you must also perform one of the following tasks:
   - Run the **conpload.sh** script, as shown in this example:

     ```
     conpload.sh 9.40 11.50
     ```

   - Set the new environment variable IFX_ONPLOAD_AUTO_UPGRADE to `1` for the upgrade to happen automatically the first time you start an HPL utility using the **ipload** or **onpladm** command, after you migrate to a new database server version. You cannot use the IFX_ONPLOAD_AUTO_UPGRADE environment variable with the **onpload** utility.

If you start an HPL utility before upgrading the **onpload** database, then you receive an error stating that the **onpload** database must be converted.

Starting with Version 9.40.xC3, the **ipload** utility does not support object names that contain more than 18 characters. The utility will continue to operate properly if legacy applications do not use long object names.

Restoring to a previous consistent state after a failed upgrade

If the CONVERSION_GUARD configuration parameter is enabled and an upgrade fails, you can run the onrestorept command to undo the changes that are made during the upgrade and restore OneDB to a consistent state.

Before you begin

The following prerequisites must be met:

- The directory that is specified by the RESTORE_POINT_DIR configuration parameter contains the files that were stored during the failed upgrade attempt.
- The server must be offline before you run the onrestorept -y command.

About this task

To restore the server to a previous consistent state after a failed upgrade:

Run the onrestorept -y command, which displays prompts while the command runs. If you do not specify -y, you must respond to every prompt that appears.

Important: Do not start the server until the onrestorept utility finishes running. Starting the server before the onrestorept utility finishes running can damage the server, requiring the server to be restored from a backup copy.

What to do next

After you restore the server to a consistent state, you can resume work in the source version of the server or find and fix the problem that caused the upgrade to fail.

Important: To start Enterprise Replication after the onrestorept utility restores the database server to a consistent state, you must use the cdr cleanstart command.

Before you attempt another upgrade, run the onrestorept -c command to remove the files in the directory that is specified in the RESTORE_POINT_DIR configuration parameter. After a successful upgrade, the server automatically deletes restore point files from that directory.

Completing required post-migration tasks

After you migrate, you must complete a series of post-migration tasks to prepare the new version of the server for use.

To complete post-migration tasks:

1. If you **do not** use OneDB® Primary Storage Manager: For ON-Bar, copy the sm_versions file on page 25.
2. Optionally update statistics on your tables after migrating on page 27.
3. Run UPDATE STATISTICS on some system catalog tables
4. Review client applications and registry keys on page 28.
5. Verify the integrity of migrated data on page 28.
6. Back up HCL OneDB after migrating to the new version on page 29.
7. Tune the new version for performance and adjust queries on page 29.
8. If you use specific features, you might have to perform additional post-migration tasks:
    - Migrating with Enterprise Replication on page 19
    - High-availability cluster migration
    - Register DataBlade modules on page 30

What to do next

Repeat the migration and post-migration procedures for each instance of HCL OneDB™ Version 2.0.0.0 that you plan to run on the computer.

Important: Do not connect applications to a database server instance until the migration has completed successfully. If a serious error occurs during the migration, you might need to revert to the previous version of the server, restore from a level-0 backup, and then correct the problem before restarting the migration tasks.

For ON-Bar, copy the sm_versions file

After migration, if you plan to use a storage manager other than the HCL OneDB™ Primary Storage Manager, copy your previous `sm_versions` file to use with ON-Bar. OneDB® Primary Storage Manager does not use the `sm_versions` file.After migration, copy your previous `sm_versions` file for the ON-Bar backup and restore system to run.

If you are using the same version of ISM, copy the same `sm_versions` file from your old database server to the new database server installation.

If you are using other storage managers, copy your previous `sm_versions` file from the old `$INFORMIXDIR/etc` directory to the new `$INFORMIXDIR/etc` directory.

If you are migrating from Version 7.31 or moving data from that version to the current version, unload the contents of the **sysutils:bar_version** table.

Finish preparing earlier versions of 12.10 databases for JSON compatibility

To make databases that were created with earlier versions of HCL OneDB™ 12.10 JSON compatible, you must complete some post-migration steps.

1. Prepare any databases that were created in 12.10.xC1 for JSON compatibility.

   If you upgraded to 12.10.xC4 or later fixpacks, skip this step because all databases are made JSON compatible during conversion.

   If you upgraded to 12.10.xC3 or 12.10.xC2, complete these steps:

   a. Run the appropriate script on the upgraded server as user **informix** or as a user with DBA privileges.
      - HCL OneDB™ 12.10.xC3: `convTovNoSQL1210.sql`
      - HCL OneDB™ 12.10.xC2: `convTovNoSQL1210X2.sql`

      Example

      For example, in HCL OneDB™ 12.10.xC3, to make the **db_name** database JSON compatible, you would run the following command as user **informix** or as a user with DBA privileges:

      UNIX™

      ```
      dbaccess -e db_name $INFORMIXDIR/etc/convTovNoSQL1210.sql
      ```

      Windows™

      ```
      dbaccess -e db_name %INFORMIXDIR%\etc\convTovNoSQL1210.sql
      ```

   b. Configure and start the wire listener.

2. If you used the wire listener in 12.10.xC2 or 12.10.xC3, after you upgrade to 12.10.xC4 or later fixpacks you must drop and recreate any indexes that you created on your collections. You do not need to drop and recreate the index that is automatically created on the _id field of a collection.
3. If you had binary JSON (BSON) DATE fields in your documents in 12.10.xC2, you must load the data from the external table that you created before you upgraded to a later 12.10 fix pack.
   On the upgraded server:

   a. Rename the table that contains the BSON column with DATE fields.
      For example, use the ALTER table statement to rename the table from **datetab** to **datetab_original**.

   b. Create a new table that has the original table name.

For example:

```
create table datetab(j int, i bson);
```

c. Load data into the new table from the external table that you created in your 12.10.xC2 server. During the load, convert the data from JSON format to BSON format.
For example:

```
insert into datetab select j, i::json::bson from ext_datetab;
```

d. Verify that the data loaded successfully.

e. Drop the original, renamed table (for example, **datetab_original**) after you are certain that the data loaded successfully into the new table (for example, **datetab**).

4. If you used the wire listener in 12.10.xC2 with a database that has any uppercase letters in its name, after you upgrade to a later fix pack you must update your applications to use only lowercase letters in the database name.

Optionally update statistics on your tables after migrating

Optionally run UPDATE STATISTICS on your tables and on UDRs that perform queries, if you have performance problems after migrating to the new version of HCL OneDB™.

An unqualified UPDATE STATISTICS statement includes no additional clauses:

```
UPDATE STATISTICS;
```

By default, an unqualified UPDATE STATISTICS statement updates the statistics in LOW mode for every permanent table in the database, including the system catalog tables.

You can run an UPDATE STATISTICS statement that includes only a FOR ROUTINE clause that specifies no routine name:

```
UPDATE STATISTICS FOR ROUTINE;
```

Running this statement reoptimizes the DML statement execution plans for every SPL routine in the database that operates on local tables.

Similarly, you can substitute the keyword FUNCTION for ROUTINE in the previous example to reoptimize execution plans only for SPL routines that return at least one value, or you can substitute the keyword FUNCTION for PROCEDURE to reoptimize execution plans only for SPL routines that return no value. In these cases, the database server does not update the statistics in the system catalog tables.

You do not need to run UPDATE STATISTICS statements on C or Java™ UDRs.

Update statistics on some system catalog tables after migrating

After migrating successfully to HCL OneDB™ Version 2.0.0.0, run UPDATE STATISTICS on some of the system catalog tables in your databases.

If you are migration from a Version 7.31 or 7.24 database server or moving data from that version to the current version, be sure to run UPDATE STATISTICS on the following system catalog tables in HCL OneDB™ Version 2.0.0.0:

sysblobs system catalog table
syscolauth system catalog table
syscolumns system catalog table
sysconstraints system catalog table
sysdefaults system catalog table
sysdistrib system catalog table
sysfragauth system catalog table
sysfragments system catalog table
sysindices system catalog table
sysobjstate system catalog table
sysopclstr system catalog table
sysprocauth system catalog table
sysproceduressysroleauth system catalog table
syssynonyms system catalog table
syssyntable system catalog table
systabauth system catalog table
systables system catalog table
systriggers system catalog table
sysusers system catalog table

Because you cannot migrate directly from a Version 7.31 or 7.24 database server to Version 2.0.0.0, use the dbexport and dbimport utilities or distributed SQL to move your data into the new database server.

Review client applications and registry keys

After you migrate a database server on the same operating system or move the database server to another compatible computer, review the client applications and `sqlhosts` file or registry-key connections. If necessary, recompile or modify client applications.

Verify that the client-application version you use is compatible with your database server version. If necessary, update the `sqlhosts` file or registry key for the client applications with the new database server information.

If you have a 64-bit ODBC application that was compiled and linked with a version of HCL OneDB™ Client Software Development Kit (Client SDK) that is prior to version 4.10, you must recompile the application after migrating. The SQLLEN and SQLULEN data types were changed to match the Microsoft™ 64-bit ODBC specification. Be sure to analyze any functions that take either of these data types to ensure that the correct type passes to the function. This step is crucial if the type is a pointer. Also note that in the ODBC specification, some parameters that were previously SQLINTEGER and SQLUINTEGER were changed to SQLLEN or SQLULEN.

For more information about interactions between client applications and different database servers, refer to a client manual.

Verify the integrity of migrated data

Open each database with DB-Access and use oncheck to verify that data was not corrupted during the migration process.

You can also verify the integrity of the reserve pages, extents, system catalog tables, data, indexes, and smart large objects, as Table 4: Commands for verifying the data integrity on page 29 shows.

Table 4. Commands for verifying the data integrity

Action	oncheck Command
Check reserve pages	oncheck -cr
Check extents	oncheck -ce
Check system catalog tables	oncheck -cc *database_name*
Check data	oncheck -cD *database_name*
Check indexes	oncheck -cI *database_name*
Check smart large objects	oncheck -cs *sbspace_name*
Check smart large objects plus extents	oncheck -cS *sbspace_name*

If the oncheck utility finds any problems, the utility prompts you to respond to corrective action that it can perform. If you respond `Yes` to the suggested corrective action, run the oncheck command again to make sure the problem has been fixed.

The oncheck utility cannot fix data that has become corrupt. If the oncheck utility is unable to fix a corruption problem, you might need to contact HCL Software Support before your proceed.

Important: If the value of the MAX_FILL_PAGES configuration parameter is 1, you must run oncheck -cD for all tables that include variable length data types (VARCHAR, NVARCHAR, and LVARCHAR), and take the suggested corrective action to avoid warnings about resetting the bitmap mode.

Back up HCL OneDB™ after migrating to the new version

Use a backup and restore tool (ON-Bar or **ontape**) to make a level-0 backup of the new database server. Do not overwrite the tapes that contain the final backup of the old database server.

For more information, see the *HCL OneDB™ Backup and Restore Guide*.

Important: Do not restore the backed up logical-log files from your old database server for your new database server.

Tune the new version for performance and adjust queries

After backing up the new server, you can tune the database server to maximize performance. If your queries are slower after the upgrade, there are steps you can take to adjust your configuration and queries.

If your queries are slower after an upgrade, find out what changed that affects your configuration and adjust your configuration and queries as necessary:

- Compare the default values in the new `onconfig.std` file to values in your previous installation, and make any necessary adjustments (see New configuration parameters).
- If you created sample queries for comparison, use them to analyze the performance differences between the old and new database servers and to determine if you need to adjust any configuration parameters or the layout of databases, tables, and chunks.
- If you changed your applications, check to see if the changes led to slower performance.
- If you changed your hardware, operating system, or network settings, determine if you need to adjust any related settings or environment variables.
- Make sure that you ran necessary update statistics after upgrading:

    Drop data distributions if necessary when upgrading on page

    Optionally update statistics on your tables after migrating on page 27

    Update statistics on some system catalog tables after migrating on page 27

Register DataBlade® modules

You must register any DataBlade® modules that you installed.

Registration is the process that makes the DataBlade® module code available to use in a particular database. For more information on how to use DataBlade® modules, see the *HCL OneDB™ DataBlade® Module Installation and Registration Guide*.

Migration of data between database servers

Migrating database servers to a new operating system

When you migrate to a new operating system, you must choose a tool for migrating your data, you might need to make some adjustments to your tables, and you must review environment-dependent configuration parameters and environment variables.

Choosing a tool for moving data before migrating between operating systems

If you are migrating between different operating systems, you must choose a method for exporting and importing data. The tool that you choose for exporting and importing data depends on how much data you plan to move.

All these methods deliver similar performance and enable you to modify the schema of the database. The tools that you can use include:

- The **dbexport** and **dbimport** utilities, which you can use to move an entire database
- The UNLOAD and LOAD statements, which move selected columns or tables (The LOAD statement does not change the data format.)
- The **dbload** utility, which you can use to change the data format
- The High-Performance Loader (HPL), which moves selected columns or tables or an entire database
- Enterprise Replication, which you can use to transfer data between HCL OneDB™ on one operating system and HCL OneDB™ on a second operating system.

Adjusting database tables for file-system variations

File system limitations vary between NFS and non-NFS file systems. You might need to break up large tables when you migrate to a new operating system.

For example, if you have a 3 GB table, but your operating system allows only 2 GB files, break up your table into separate files before you migrate. For more information, see your *HCL OneDB™ Administrator's Guide*.

The HCL OneDB™ storage space can reside on an NFS-mounted file system using regular operating-system files. For information about the NFS products you can use to NFS mount a storage space for the HCL OneDB™ database server, check product compatibility information.

Moving data to a database server on a different operating system

You can move data between HCL OneDB™ database servers on UNIX™ or Linux™ and Windows™.

About this task

To move data to a database server on a different operating system:

1. Save a copy of the current configuration files.
2. Use ON-Bar to make a final level-0 backup.
   For more information, refer to your *HCL OneDB™ Backup and Restore Guide*.
3. Choose one of the following sets of migration utilities to unload the databases:
   - **dbexport** and **dbimport**
   - UNLOAD, **dbschema**, and LOAD
   - UNLOAD, **dbschema**, and **dbload**
4. Bring the source database server offline.
5. Install and configure the target database server. If you are migrating to Windows™, also install the administration tools.
6. Bring the target database server online.
7. Use **dbimport**, LOAD, or **dbload**, or external tables to load the databases into the target database server, depending on which utility you used to export the databases.
8. Make an initial level-0 backup of the target database server.
9. Run UPDATE STATISTICS to update the information that the target database server uses to plan efficient queries.

Adapting your programs for a different operating system

When you change to a different operating system, you must review and, if necessary, adjust your environment-dependent configuration parameters and environment variables.

Certain database server configuration parameters and environment variables are environment-dependent.

For details, see the information about configuration parameters in the *HCL OneDB™ Administrator's Guide* and the *HCL OneDB™ Administrator's Reference* and the information about environment variables in the *HCL OneDB™ Administrator's Guide* and the *HCL OneDB™ Guide to SQL: Reference*.

Ensuring the successful creation of system databases

The first time the database server is brought online, the **sysmaster**, **sysutils**, **sysuser**, and **sysadmin** databases are built. After moving to a database server on a different operating system, check the message log to ensure that the **sysmaster** and **sysutils** databases have been created successfully before you allow users to access the database server.

After you ensure that client users can access data on the database server, the migration process is complete.

Next you might want to seek ways to obtain maximum performance. For details on topics related to performance, see your *HCL OneDB™ Performance Guide*.

Data migration utilities

External tables

External tables are a fast and versatile method for moving data between database servers. External tables are the fastest method for loading data into a RAW table with no indexes.

You run the CREATE EXTERNAL TABLE statement to define an external table that is not part of your database to unload data from your database. You run INSERT ... SELECT statements to load data from the external table into your database.

You can unload and load data in the internal HCL OneDB™ representation. All HCL OneDB™ data types are supported. You can define a value to be interpreted as a NULL when you load or unload data from an external table. You can specify the date and currency format and replace missing values with column defaults.

You define a named pipe to copy data from one HCL OneDB™ instance to another without writing the data to an intermediate file. You can monitor the I/O queues to determine whether you have enough FIFO virtual processors. If necessary, you can add more FIFO virtual processors to improve performance. You can specify to run high-speed transfers in parallel.

Rows that have conversion errors during a load are written to a reject file on the server that performs the conversion.

The dbexport and dbimport utilities

The dbexport and dbimport utilities import and export a database and its schema to disk or tape.

The dbexport utility unloads an entire database into text files and creates a schema file. You can unload the database and its schema file either to disk or tape. If you prefer, you can unload the schema file to disk and unload the data to tape. You can use the schema file with the dbimport utility to re-create the database schema in another HCL OneDB™ environment, and you can edit the schema file to modify the database that dbimport creates.

The dbimport utility creates a database and loads it with data from text files on tape or disk. The input files consist of a schema file that is used to re-create the database and data files that contain the database data. Normally, you generate the input files with the dbexport utility, but you can use any properly formatted input files.

Attention:

When you import a database, use the same environment variable settings and configuration parameter settings that were used when the database was created.

- If any environment variables or configuration parameters that affect fragmentation, constraints, triggers, or user-defined routines are set differently than they were when these database objects were created originally, the database that is created by the dbimport utility might not be an accurate reproduction of the original.
- Incompatible settings are likely to occur if you move data from an earlier version of the database server to a newer version. Over time, some configuration parameters or environment variables are deprecated, or their default values are changed. For example, assume that attached indexes were created by default in the original database. In the current version of the database server, detached indexes are created by default. If you want to maintain the original behavior, you can set the **DEFAULT_ATTACH** environment variable to 1 before you run the dbimport utility.

Also, the dbimport operation might fail when you attempt to import from a higher server version to a lower server version if the database schema changed between versions. For example, the **am_expr_pushdown** column was added to the **sysams** system catalog table in HCL OneDB™ 11.70. The dbimport operation will fail if you attempt to import a database from HCL OneDB™ 12.10 that contains the **am_expr_pushdown** column into a database from HCL OneDB™ 11.50 that is missing that column. In that case, you must review the messages in the `dbimport.out` file, which is in your current directory. After you address the issues that caused the dbimport operation to fail, run the dbimport command again.

Requirements or limitations apply in the following cases:

Compressed data

The dbexport utility uncompresses compressed data. You must recompress the data after you use the dbimport utility to import the data.

Date values

Use four-digit years for date values. The date context for an object includes the date that the object was created, the values of the **DBCENTURY** and **GL_DATE** environment variables, and some other environment variables. If the date context during import is not the same as when these objects were created, you might get explicit errors, you might not be able to find your data, or a check constraint might not work as expected. Some of these problems do not generate errors.

Tip: By default, the dbexport utility exports dates in four-digit years unless environment variables are set to values that would override that format.

High-availability clusters

You cannot use the dbexport utility on HDR secondary servers or shared disk (SD) secondary servers.

The dbexport utility is supported on a remote standalone (RS) secondary server only if the server is set to stop applying log files. Use the STOP_APPLY configuration parameter to stop application of log files.

The dbimport utility is supported on all updatable secondary servers.

Label-based access control (LBAC)

When you export data that is LBAC-protected, the data that is exported is limited to the data that your LBAC credentials allow you to read. If your LBAC credentials do not allow you to read a row, that row is not exported, but no error is returned. To export all the rows, you must be able to see all the rows.

NLSCASE mode

Whether the NLSCASE mode of your source database is SENSITIVE or INSENSITIVE, you can reduce the risk of case-sensitivity issues by always migrating to a target database that has the same NLSCASE mode as the source database. For tables that include columns with case-variant NCHAR and NVARCHAR data values (for example, 'IBM', 'ibm', 'Ibm'), you might encounter the following differences after migration:

- ORDER BY and sorting operations can produce a different ordering of qualifying rows in query results, compared to the result of the same query before migration.
- Unique indexes and referential constraints with which the data were compliant before the migration might have integrity violations in the new database, if any index or constraint key column contains case-variant forms of the same character string.
- Queries with predicates that apply conditional operators to NCHAR or NVARCHAR values might return different results from the same data after migration.

Nondefault database locales

If the database uses a nondefault locale and the **GL_DATETIME** environment variable has a nondefault setting, you must set the **USE_DTENV** environment variable to the value of `1` so that localized DATETIME values are processed correctly by the dbexport and dbimport utilities.

SELECT triggers on tables

You must disable SELECT triggers before you export a database with the dbexport utility. The dbexport utility runs SELECT statements during export. The SELECT statement triggers might modify the database content.

Virtual tables for the HCL OneDB™ MQ extension

The MQCreateVtiRead(), MQCreateVtiReceive(), and MCQCreateVtiWrite() functions create virtual tables, and map them to the appropriate message queue. When the dbexport utility unloads data, it removes the messages from WebSphere® MQ queues. Before you use the dbexport utility, drop any MQ virtual tables. After you load the database with the dbimport utility, you can create the tables in the target database by using the appropriate functions.

Syntax of the dbexport command

The dbexport command unloads a database into text files that you can later import into another database. The command also creates a schema file.

```
dbexport { [ {  | -c  | -d | -no-data-tablestablenames | -no-data-tables-accessmethodsaccessmethods  | -nw | -q |
<Destination Options>(explicit id )  | -ss | -si | -X } ] database | [ { -V | -version } ] }
```

Element	Purpose	Key Considerations
-c	Makes dbexport complete exporting unless a fatal error occurs	**References**: For details on this option, see dbexport errors on page 37.
-d	Makes dbexport export simple-large-object descriptors only, not simple-large-object data	**References**: For information about simple-large-object descriptors, see the .
-q	Suppresses the display of error messages, warnings, and generated SQL data-definition statements	None.
-ss	Generates database server-specific information for all tables in the specified database	**References**: For details on this option, see dbexport server-specific information on page 37.
-si	Excludes the generation of index storage clauses for non-fragmented tables The **-si** option is available only with the **-ss** option.	**References**: For details on this option, see dbexport server-specific information on page 37.
-X	Recognizes HEX binary data in character fields	None.
-no-data-tables	Prevents data from being exported for the specified tables. Only the definitions of the specified tables are exported.	Accepts a comma-separated list of names of tables for which data will not be exported. **Default behavior:** Only the definition of the **tsinstanceTable** table is exported, not the data. The data and definitions of all other tables are exported.
-no-data-tables-accessmethods	Prevents data from being unloaded using the specified access methods.	Accepts a comma-separated list of names of access methods. Tables using those access methods are not unloaded. **Default value:** ts_rts_vtam, ts_vtam Tables using ts_rts_vtam and ts_vtam access methods are not unloaded.

Element	Purpose	Key Considerations
-nw	Generates the SQL for creating a database without the specification of an owner	None.
-V	Displays the software version number and the serial number	None.
-version	Extends the **-V** option to display additional information about the build operating system, build number, and build date	None.
database	Specifies the name of the database that you want to export	**Additional information**: If your locale is set to use multibyte characters, you can use multibyte characters for the database name. **References:** If you want to use more than the simple name of the database, see Database Name on page .

You must have DBA privileges or log in as user **informix** to export a database.

Global Language Support: When the environment variables are set correctly, as described in the *HCL OneDB™ GLS User's Guide*, dbexport can handle foreign characters in data and export the data from GLS databases. For more information, refer to Database renaming on page 46.

You can set the IFX_UNLOAD_EILSEQ_MODE environment variable to enable dbexport to use character data that is invalid for the locale specified in the environment.

You can use delimited identifiers with the dbexport utility. The utility detects database objects that are keywords, mixed case, or have special characters, and the utility places double quotes around them.

In addition to the data files and the schema file, dbexport creates a file of messages named `dbexport.out` in the current directory. This file contains error messages, warnings, and a display of the SQL data definition statements that it generates. The same material is also written to standard output unless you specify the **-q** option.

During export, the database is locked in exclusive mode. If dbexport cannot obtain an exclusive lock, it displays a diagnostic message and exits.

Tip: The dbexport utility can create files larger than 2 GB. To support such large files, make sure your operating system file-size limits are set sufficiently high. For example, on UNIX™, set **ulimit** to unlimited.

Example

Example

The following command exports the table definitions but no data for all the tables in the customer database.

```
dbexport -no-data-tables -no-data-tables-accessmethods customer
```

Example

Example

The following command generates the schema and data for the `customer` database without the specification of an owner:

```
dbexport customer -nw
```

Termination of the dbexport utility

You can stop the **dbexport** utility at any time.

To cancel **dbexport**, press your `Interrupt` key.

The **dbexport** utility asks for confirmation before it terminates.

dbexport errors

The **dbexport -c** option tells **dbexport** to complete exporting unless a fatal error occurs.

Even if you use the **-c** option, **dbexport** interrupts processing if one of the following fatal errors occurs:

- **dbexport** is unable to open the specified tape.
- **dbexport** finds bad writes to the tape or disk.
- Invalid command parameters were used.
- **dbexport** cannot open the database or there is no system permission for doing so.
- A subdirectory with the name specified during invocation already exists

dbexport server-specific information

The **dbexport -ss** option generates server-specific information. This option specifies initial- and next-extent sizes, fragmentation information if the table is fragmented, the locking mode, the dbspace for a table, the blobspace for any simple large objects, and the dbspace for any smart large objects.

The **dbexport -si option**, which is available only with the **-ss** option, does not generate index storage clauses for non-fragmented tables.

dbexport destination options

The dbexport utility supports disk and tape destination options.

Destination options

“ [{ `-o`*directory* | `-t`*device*`-b`*blocksize* `-s`*tapesize* [`-f`*pathname*] }] ”

Element	Purpose	Key Considerations
-b *blocksize*	Specifies, in kilobytes, the block size of the tape device.	None.
-f *pathname*	Specifies the name of the path where you want the schema file stored, if you are storing the data files on tape.	The path name can be a complete path name or a file name. If only a file name is given, the file is stored in the current directory. If you do not use the -f option, the SQL source code is written to the tape.
-o *directory*	Specifies the directory on disk in which dbexport creates the `*database*.exp` directory. This directory holds the data files and the schema file that dbexport creates for the database.	The specified directory must exist.
-s *tapesize*	Specifies, in kilobytes, the amount of data that you can store on the tape.	To write to the end of the tape, set the value to `0`. If you do not specify `0`, the maximum size is 2 097 151 KB.
-t *device*	Specifies the path name of the tape device where you want the text files and, possibly, the schema file stored.	You cannot specify a remote tape device.

When you write to disk, dbexport creates a subdirectory, *database*`.exp`, in the directory that the -o option specifies. The dbexport utility creates a file with the `.unl` extension for each table in the database. The schema file is written to the file *database*`.sql`. The `.unl` and `.sql` files are in the *database*`.exp` directory.

If you do not specify a destination for the data and schema files, the subdirectory *database*`.exp` is placed in the current working directory.

When you write the data files to tape, you can use the -f option to store the schema file to disk. You are not required to name the schema file *database*`.sql`. You can give it any name.

UNIX/Linux Only

For database servers on UNIX™ or Linux™, the command is:

```
dbexport //finland/reports
```

The following command exports the database **stores_demo** to tape with a block size of 16 KB and a tape capacity of 24 000 KB. The command also writes the schema file to `/tmp/stores_demo.imp`.

```
dbexport -t /dev/rmt0 -b 16 -s 24000 -f /tmp/stores_demo.imp
   stores_demo
```

The following command exports the same **stores_demo** database to the directory named `/work/exports/stores_demo.exp`. The resulting schema file is `/work/exports/stores_demo.exp/stores_demo.sql`.

```
dbexport -o /work/exports stores_demo
```

Windows™ Only

For Windows™, the following command exports the database **stores_demo** to tape with a block size of 16 KB and a tape capacity of 24 000 KB. The schema file is written to `C:\temp\stores_demo.imp`.

```
dbexport -t \\.\TAPE2 -b 16 -s 24000 -f
   C:\temp\stores_demo.imp  stores_demo
```

The following command exports the same **stores_demo** database to the directory named `D:\work\exports\stores_demo.exp`. The resulting schema file is `D:\work\exports\stores_demo.exp\stores_demo.sql`.

```
dbexport -o D:\work\exports stores_demo
```

Exporting time series data in rolling window containers

The dbexport utility exports time series data except any data that is in the dormant window of rolling window containers.

About this task

The active window in rolling window containers is re-created after the time series data is loaded into a container.

The dormant window is not exported. To export the data from the dormant window, you must move the data into the active window.

To export time series data in the dormant window of a rolling window container:

1. If necessary, increase the size of the active window by running the TSContainerManage function. The size of the active window must be large enough to fit all the intervals in the dormant window can fit into the active window.
2. Move the intervals in the dormant window into the active window by running the TSContainerManage function.
3. Export the data by running the dbexport utility.

Contents of the schema file that dbexport creates

The dbexport utility creates a schema file. This file contains the SQL statements that you need to re-create the exported database.

You can edit the schema file to modify the schema of the database.

If you use the -ss option, the schema file contains server-specific information, such as initial- and next-extent sizes, fragmentation information, lock mode, the dbspace where each table resides, the blobspace where each simple-large-object column resides, and the dbspace for smart large objects. The following information is not retained:

- Logging mode of the database

  For information about logging modes, see the *HCL OneDB™ Guide to SQL: Reference*.

- The starting values of SERIAL columns
- The dormant window interval values for time series rolling window containers

The statements in the schema file that create tables, views, indexes, partition-fragmented tables and indexes, roles, and grant privileges do so with the name of the user who originally created the database. In this way, the original owner retains DBA privileges for the database and is the owner of all the tables, indexes, and views. In addition, the person who runs the dbimport command also has DBA privileges for the database.

The schema file that dbexport creates contains comments, which are enclosed in braces, with information about the number of rows, columns, and indexes in tables, and information about the unload files. The dbimport utility uses the information in these comments to load the database.

The number of rows must match in the unload file and the corresponding unload comment in the schema file. If you change the number of rows in the unload file but not the number of rows in the schema file, a mismatch occurs.

Attention: Do not delete any comments in the schema file, and do not change any existing comments or add any new comments. If you change or add comments, the dbimport utility might stop or produce unpredictable results.

If you delete rows from an unload file, update the comment in the schema file with the correct number of rows in the unload file. Then dbimport is successful.

Syntax of the dbimport command

The dbimport command imports previously exported data into another database.

```
dbimport [{ | -c | -D | -nv | -q | -X}] <Input-File Location>(explicit id ) <Create Options>(explicit id ) [{ -V | -version }]
database
```

Element	Purpose	Key Considerations
-c	Completes importing data even when certain nonfatal errors occur	For more information, see dbimport errors and warnings on page 42.
-D	Specifies a default extent size of 16 KB for the first and subsequent extents during the import operation, if the extent sizes are not specified in the CREATE TABLE statement.	This option is ignored if the extent sizes are specified in the CREATE TABLE statement. Default values help to ensure that enough space is available in the dbspace that is designated for the import operation. This option prevents the automatic calculation of extent sizes during the import operation, and is useful especially in the following situations:

Element	Purpose	Key Considerations
		• When importing tables that contain columns with large maximum row sizes, such as LVARCHAR columns. • When importing data after the dbexport command was run without the -ss option. The -ss option specifies server-specific information about extent sizes.
-nv	While the dbimport -nv command is running, tables with foreign-key constraints that ALTER TABLE ADD CONSTRAINT creates in enabled or filtering mode are not checked for violations, as if you had also specified NOVALIDATE	By bypassing the checking of referential constraints, this option can reduce migration time for very large tables that already conform to their foreign-key constraints. The NOVALIDATE mode does not persist after the ALTER TABLE ADD CONSTRAINT statement has completed.
-q	Suppresses the display of error messages, warnings, and generated SQL data-definition statements	None.
-V	Displays the software version number and the serial number	None.
-version	Extends the **-V** option to display additional information about the build operating system, build number, and build date	None.
-**X**	Recognizes HEX binary data in character fields	None.
database	Declares the name of the database to create	If you want to use more than the simple name of the database, see Database Name on page .

The dbimport utility can use files from the following location options:

- All input files are on disk.
- All input files are on tape.
- The schema file is on disk, and the data files are on tape.

Important: Do not put comments into your input file. Comments might cause unpredictable results when the dbimport utility reads them.

The dbimport utility supports the following tasks for an imported HCL OneDB™ database server:

- Specify the dbspace where the database will reside
- Create an ANSI-compliant database with unbuffered logging
- Create a database that supports explicit transactions (with buffered or unbuffered logging)
- Create an unlogged database
- Create a database with the NLS case-insensitive property for NCHAR and NVARCHAR strings.
- Process all ALTER TABLE ADD CONSTRAINT and SET CONSTRAINTS statements in the `.sql` file of the exported database that define enabled or filtering referential constraints so that any foreign-key constraints that are not specified as DISABLED are in ENABLED NOVALIDATE or in FILTERING NOVALIDATE mode.

Note: If you specify the -nv option, the `.sql` file of the exported database is not modified, but any foreign-key constraints that ALTER TABLE ADD CONSTRAINT or SET CONSTRAINTS statements enable are processed without checking each row of the table for violations. The `ENABLED`, or `FILTERING WITH ERROR`, or `FILTERING WITHOUT ERROR` constraint mode specifications are implemented instead as the `ENABLED NOVALIDATE`, or `FILTERING WITH ERROR NOVALIDATE` or `FILTERING WITHOUT ERROR NOVALIDATE` modes. After the foreign-key constraints have been enabled without checking for violations, their modes automatically revert to whatever the `.sql` file specified so that subsequent DML operations on the tables enforce referential integrity.

The user who runs the dbimport utility is granted the DBA privilege on the newly created database. The dbimport process locks each table as it is being loaded and unlocks the table when the loading is complete.

Global Language Support: When the GLS environment variables are set correctly, as the *HCL OneDB™ GLS User's Guide* describes, dbimport can import data into database server versions that support GLS.

Termination of the dbimport utility

You can stop the **dbimport** utility at any time.

To cancel the dbimport utility, press your `Interrupt` key .

The **dbimport** utility asks for confirmation before it terminates.

dbimport errors and warnings

The **dbimport -c** option tells the **dbimport** utility to complete exporting unless a fatal error occurs.

If you include the **-c** option in a **dbimport** command, **dbimport** ignores the following errors:

- A data row that contains too many columns
- Inability to put a lock on a table
- Inability to release a lock

Even if you use the **-c** option, **dbimport** interrupts processing if one of the following fatal errors occurs:

- Unable to open the tape device specified
- Bad writes to the tape or disk
- Invalid command parameters
- Cannot open database or no system permission
- Cannot convert the data

The **dbimport** utility creates a file of messages called **dbimport.out** in the current directory. This file contains any error messages and warnings that are related to **dbimport** processing. The same information is also written to the standard output unless you specify the **-q** option.

dbimport input-file location options

The input-file location specifies the location of the `*database*.exp` directory, which contains the files that the dbimport utility imports.

If you do not specify an input-file location, dbimport searches for data files in the directory `*database*.exp` under the current directory and for the schema file in `*database*.exp/*database*.sql`.

dbimport input-file location

" [{ `-i`*directory* | `-t`*device* [`-b`*blocksize*`-s` *tapesize*] [`-f`*pathname*] }] "

Element	Purpose	Key Considerations
-b *blocksize*	Specifies, in kilobytes, the block size of the tape device	If you are importing from tape, you must use the same block size that you used to export the database. If you do not use the -b option, the default block size is 1.
-f *pathname*	Specifies where dbimport can find the schema file to use as input to create the database when the data files are read from tape	If you use the -f option to export a database, you typically use the same path name that you specified in the dbexport command. If you specify only a file name, dbimport looks for the file in the `.exp` subdirectory of your current directory. If you do not use the -f option, the SQL source code is written to the tape.
-i *directory*	Specifies the complete path name on disk of the `*database*.exp` directory, which holds the input data files and schema file that dbimport uses to create and load the new database. The directory name must be the same as the database name.	This directory must be the same directory that you specified with the dbexport -o option. If you change the directory name, you also rename your database.

Element	Purpose	Key Considerations
-s *tapesize*	Specifies, in kilobytes, the amount of data that you can store on the tape	To read to the end of the tape, specify a tape size of 0. If you are importing from tape, you must use the same tape size that you used to export the database. The maximum size is 2 097 151 KB. If you do not use the -s option, the default value is 0 (read to the end of the tape).
-t *device*	Specifies the path name of the tape device that holds the input files	You cannot specify a remote tape device.

Examples showing input file location on UNIX™ or Linux™

To import the **stores_demo** database from a tape with a block size of 16 KB and a capacity of 24 000 KB, issue this command:

```
dbimport -c  -t /dev/rmt0 -b 16 -s 24000 -f
   /tmp/stores_demo.imp stores_demo
```

The schema file is read from `/tmp/stores_demo.imp`.

To import the **stores_demo** database from the `stores_demo.exp` directory under the `/work/exports` directory, issue this command:

```
dbimport -c -i /work/exports stores_demo
```

The schema file is assumed to be `/work/exports/stores_demo.exp/stores_demo.sql`.

Examples showing input file location on Windows™

To import the **stores_demo** database from a tape with a block size of 16 KB and a capacity of 24 000 KB, issue this command:

```
dbimport -c  -t \\.\TAPEDRIVE -b 16 -s 24000 -f
   C:\temp\stores_demo.imp stores_demo
```

The schema file is read from `C:\temp\stores_demo.imp`.

To import the **stores_demo** database from the `stores_demo.exp` directory under the `D:\work\exports` directory, issue this command:

```
dbimport -c -i D:\work\exports stores_demo
```

The schema file is assumed to be `D:\work\exports\stores_demo.exp\stores_demo.sql`.

dbimport create options

The **dbimport** utility supports options for creating a database, specifying a dbspace for that database, defining logging options, and optionally specifying ANSI/ISO-compliance or NLS case-insensitivity (or both) as properties of the database.

Create options

```
“[-ddbspace][-l{ [buffered] |[-ansi]}] [-ci] ”
```

Element	Purpose	Key Considerations
-ansi	Creates an ANSI/ISO-compliant database in which the ANSI/ISO rules for transaction logging are enabled. Otherwise, the database uses explicit transactions by default.	If you omit the **-ansi** option, the database uses explicit transactions. **Additional Information:** For more information about ANSI/ISO-compliant databases, see the *HCL OneDB™ Guide to SQL: Reference*.
-ci	Specifies the NLS case-insensitive property. Otherwise, the database is case-sensitive by default.	**Additional Information:** See the *HCL OneDB™ Guide to SQL: Syntax* and *HCL OneDB™ Guide to SQL: Reference* descriptions of the NLS case-insensitive property.
-d *dbspace*	Specifies the dbspace where the database is created. .	If this is omitted, the default location is the **root** dbspace
-l	Establishes unbuffered transaction logging for the imported database. If the **-l** flag is omitted, the database is unlogged,	**References:** For more information, see Database-logging mode on page 46.
-l buffered	Establishes buffered transaction logging for the imported database. If **-l** is included but **buffered** is omitted, the database uses unbuffered logging.	**References:** For more information, see Database-logging mode on page 46.

If you created a table or index fragment containing partitions in OneDB Version 10.00 or a later version of the HCL OneDB™ database server, you must use syntax containing the partition name when importing a database that contains multiple partitions within a single dbspace. See the *HCL OneDB™ Guide to SQL: Syntax* for syntax details.

Example showing dbimport create options (UNIX™ or Linux™)

To import the **stores_demo** database from the **/usr/informix/port/stores_demo.exp** directory, issue this command:

```
dbimport -c stores_demo -i /usr/informix/port -l -ansi
```

The new database is ANSI/ISO-compliant.

The next example similarly imports the **stores_demo** database from the **/usr/informix/port/stores_demo.exp** directory. The imported database uses buffered transaction logging and explicit transactions. The **-ci** flag specifies *case insensitivity* in queries and in other operations on columns and character strings of the NCHAR and NVARCHAR data types:

```
dbimport -c stores_demo -i /usr/informix/port -l buffered -ci
```

The **-ansi** and **-ci** options for database properties are not mutually exclusive. You can specify an ANSI/ISO-compliant database that is also NLS case-insensitive, as in the following example of the **dbimport** command:

```
dbimport -c stores_demo -i /usr/informix/port -l -ansi -ci
```

Example showing dbimport create options (Windows™)

To import the **stores_demo** database from the **C:\USER\informix\port\stores_demo.exp** directory, issue this command:

```
dbimport -c stores_demo -i C:\USER\informix\port -l -ansi
```

The imported database is ANSI/ISO-compliant and is case-sensitive for all built-in character data types.

Database-logging mode

Because the logging mode is not retained in the schema file, you can specify logging information when you use the **dbimport** utility to import a database.

You can specify any of the following logging options when you use **dbimport**:

- ANSI-compliant database with unbuffered logging
- Unbuffered logging
- Buffered logging
- No logging

For more information, see dbimport create options on page 45.

The **-l** options are equivalent to the logging clauses of the CREATE DATABASE statement, as follows:

- Omitting any of the **-l** options is equivalent to omitting the WITH LOG clause.
- The **-l** option is equivalent to the WITH LOG clause.
- The **-l buffered** option is equivalent to the WITH BUFFERED LOG.
- The **-l -ansi** option is equivalent to the WITH LOG MODE ANSI clause, and implies unbuffered logging.

Database renaming

The **dbimport** utility gives the new database the same name as the database that you exported. If you export a database to tape, you cannot change its name when you import it with **dbimport**. If you export a database to disk, you can change the database name.

You can use the RENAME DATABASE statement to change the database name.

Alternative ways to change the database name

The following examples show alternative ways to change the database name. In this example, assume that **dbexport** unloaded the database **stores_demo** into the directory **/work/exports/stores_demo.exp**. Thus, the data files (the **.unl** files) are stored in **/work/exports/stores_demo.exp**, and the schema file is **/work/exports/stores_demo.exp/stores_demo.sql**.

To change the database name to a new name on UNIX™ or Linux™:

1. Change the name of the **.exp** directory. That is, change **/work/exports/stores_demo.exp** to **/work/exports/*newname*.exp**.
2. Change the name of the schema file. That is, change **/work/exports/stores_demo.exp/stores_demo.sql** to **/work/exports/stores_demo.exp/*newname*.sql**. Do not change the names of the **.unl** files.
3. Import the database with the following command:

   ```
   dbimport -i /work/exports newname
   ```

To change the database name to a new name on Windows™:

In the following example, assume that **dbexport** unloaded the database **stores_demo** into the directory **D:\work\exports\stores_demo.exp**. Thus, the data files (the **.unl** files) are stored in **D:\work\exports\stores_demo.exp**, and the schema file is **D:\work\exports\stores_demo.exp\stores_demo.sql**.

1. Change the name of the **.exp** directory. That is, change **D:\work\exports\stores_demo.exp** to **D:\work\exports\*newname*.exp**.
2. Change the name of the schema file. That is, change **D:\work\exports\stores_demo.exp\stores_demo.sql** to **D:\work\exports\stores_demo.exp\*newname*.sql**. Do not change the names of the **.unl** files.
3. Import the database with the following command:

   ```
   dbimport -i D:\work\exports
   ```

Changing the database locale with dbimport

You can use the **dbimport** utility to change the locale of a database.

To change the locale of a database:

1. Set the **DB_LOCALE** environment variable to the name of the current database locale.
2. Run **dbexport** on the database.
3. Use the DROP DATABASE statement to drop the database that has the current locale name.
4. Set the **DB_LOCALE** environment variable to the desired database locale for the database.
5. Run **dbimport** to create a new database with the desired locale and import the data into this database.

Simple large objects

When the **dbimport**, **dbexport**, and DB-Access utilities process simple-large-object data, they create temporary files for that data in a temporary directory.

Before you export or import data from tables that contain simple large objects, you must have one of the following items:

- A **\tmp** directory on your currently active drive
- The **DBTEMP** environment variable set to point to a directory that is available for temporary storage of the simple large objects

Windows™ Only

Windows™ sets the **TMP** and **TEMP** environment variables in the command prompt sessions, by default. However, if the **TMP**, **TEMP**, and **DBTEMP** environment variables are not set, **dbimport** places the temporary files for the simple large objects in the **\tmp** directory.

Attention: If a table has a CLOB or BLOB in a column, you cannot use **dbexport** to export the table to a tape. If a table has a user-defined type in a column, using **dbexport** to export the table to a tape might yield unpredictable results, depending on the export function of the user-defined type. Exported CLOB sizes are stored in hex format in the unload file.

The dbload utility

The **dbload** utility loads data into databases or tables that HCL® OneDB® products created. It transfers data from one or more text files into one or more existing tables.

Prerequisites: If the database contains label-based access control (LBAC) objects, the **dbload** utility can load only those rows in which your security label dominates the column-security label or the row-security label. If the entire table is to be loaded, you must have the necessary LBAC credentials for writing all of the labeled rows and columns. For more information about LBAC objects, see the *HCL OneDB™ Security Guide* and the *HCL OneDB™ Guide to SQL: Syntax*.

You cannot use the **dbload** utility on secondary servers in high-availability clusters.

When you use the **dbload** utility, you can manipulate a data file that you are loading or access a database while it is loading. When possible, use the LOAD statement, which is faster than **dbload**.

The **dbload** utility gives you a great deal of flexibility, but it is not as fast as the other methods, and you must prepare a command file to control the input. You can use **dbload** with data in a variety of formats.

The **dbload** utility offers the following advantages over the LOAD statement:

- You can use **dbload** to load data from input files that were created with a variety of format arrangements. The **dbload** command file can accommodate data from entirely different database management systems.
- You can specify a starting point in the load by directing **dbload** to read but ignore *x* number of rows.
- You can specify a batch size so that after every *x* number of rows are inserted, the insert is committed.
- You can limit the number of bad rows read, beyond which **dbload** ends.

The cost of **dbload** flexibility is the time and effort spent creating the **dbload** command file, which is required for **dbload** operation. The input files are not specified as part of the **dbload** command line, and neither are the tables into which the data is inserted. This information is contained in the command file.

Syntax of the dbload command

The **dbload** command loads data into databases or tables.

```
dbload [ { { | -ddatabase | -ccommand file | -lerror log file } [ {  | -r  | -k | -eerrors [-p]  | -iignore rows |  -ncommit interval
| -x } ] [ -s ] | [ { -V | -version } ] } ]
```

Element	Purpose	Key Considerations
-c *command file*	Specifies the file name or path name of a **dbload** command file	**References:** For information about building the command file, see Command file for the dbload utility on page 52.
-d *database*	Specifies the name of the database to receive the data	**Additional Information**: To use more than the simple name of the database, see Database Name on page .
-e *errors*	Specifies the number of bad rows that **dbload** reads before terminating. The default value for *errors* is 10.	**References:** For more information, see Bad-row limit during a load operation on page 51.
-i *ignore rows*	Specifies the number of rows to ignore in the input file	**References:** For more information, see Rows to ignore during a load operation on page 51.
-k	Instructs **dbload** to lock the tables listed in the command file in exclusive mode during the load operation	**References:** For more information, see Table locking during a load operation on page 50. You cannot use the **-k** option with the **-r** option because the **-r** option specifies that no tables are locked during the load operation.
-l *error log file*	Specifies the file name or path name of an error log file	If you specify an existing file, its contents are overwritten. If you specify a file that does not exist, **dbload** creates the file. **Additional Information:** The error log file stores diagnostic information and any input file rows that **dbload** cannot insert into the database.
-n *commit interval*	Specifies the commit interval in number of rows The default interval is 100 rows.	**Additional Information:** If your database supports transactions, **dbload** commits a transaction after the specified number of new rows is read and inserted. A message appears after each commit. **References:** For information about transactions, see the *HCL OneDB™ Guide to SQL: Tutorial*.
-p	Prompts for instructions if the number of bad rows exceeds the limit	**References:** For more information, see Bad-row limit during a load operation on page 51.

Element	Purpose	Key Considerations
-r	Prevents **dbload** from locking the tables during a load, thus enabling other users to update data in the table during the load	**Additional Information:** For more information, see Table locking during a load operation on page 50. You cannot use the **-r** option with the **-k** option because the **-r** option specifies that the tables are not locked during the load operation while the **-k** option specifies that the tables are locked in exclusive mode.
-s	Checks the syntax of the statements in the command file without inserting data	**Additional Information:** The standard output displays the command file with any errors marked where they are found.
-V	Displays the software version number and the serial number	None.
-version	Extends the **-V** option to display additional information about the build operating system, build number, and build date	None.
-X	Recognizes HEX binary data in character fields	None.

Tip: If you specify part (but not all) of the required information, **dbload** prompts you for additional specifications. The database name, command file, and error log file are all required. If you are missing all three options, you receive an error message.

Example

dbload command example

The following command loads data into the **stores_demo** database in the **turku** directory on a database server called **finland**:

```
dbload -d //finland/turku/stores_demo -c commands -l errlog
```

Table locking during a load operation

The **dbload -k** option overrides the default table lock mode during the load operation. The **-k** option instructs **dbload** to lock the tables in exclusive mode rather than shared mode.

If you do not specify the **-k** option, the tables specified in the command file are locked in shared mode. When tables are locked in shared mode, the database server still must acquire exclusive row or page locks when it inserts rows into the table.

When you specify the **-k** option, the database server places an exclusive lock on the entire table. The **-k** option increases performance for large loads because the database server does not need to acquire exclusive locks on rows or pages as it inserts rows during the load operation.

If you do not specify the **-r** option, the tables specified in the command file are locked during loading so that other users cannot update data in the table. Table locking reduces the number of locks needed during the load but reduces concurrency. If you are planning to load a large number of rows, use table locking and load during nonpeak hours.

Rows to ignore during a load operation

The **dbload -i** option specifies the number of new-line characters in the input file that **dbload** ignores before **dbload** begins to process data.

This option is useful if your most recent **dbload** session ended prematurely.

For example, if **dbload** ends after it inserts 240 lines of input, you can begin to load again at line 241 if you set *number rows ignore* to `240`.

The **-i** option is also useful if header information in the input file precedes the data records.

Bad-row limit during a load operation

The **dbload -e** option lets you specify how many bad rows to allow before **dbload** terminates.

If you set *errors* to a positive integer, **dbload** terminates when it reads (*errors* + 1) bad rows. If you set *errors* to zero, **dbload** terminates when it reads the first bad row.

If **dbload** exceeds the bad-row limit and the **-p** option is specified, **dbload** prompts you for instructions before it terminates. The prompt asks whether you want to roll back or to commit all rows that were inserted since the last transaction.

If **dbload** exceeds the bad-row limit and the **-p** option is not specified, **dbload** commits all rows that were inserted since the last transaction.

Termination of the dbload utility

If you press your `Interrupt` key, **dbload** terminates and discards any new rows that were inserted but not yet committed to the database (if the database has transactions).

Name and object guidelines for the dbload utility

You must follow guidelines for specifying network names and handling simple large objects, indexes, and delimited identifiers when you use the **dbload** utility.

Table 5. Name and object guidelines for the dbload utility

Objects	Guideline
Network names	If you are on a network, include the database server name and directory path with the database name to specify a database on another database server.

Table 5. Name and object guidelines for the dbload utility (continued)

Objects	Guideline
Simple large objects	You can load simple large objects with the **dbload** utility as long as the simple large objects are in text files.
Indexes	The presence of indexes greatly affects the speed with which the **dbload** utility loads data. For best performance, drop any indexes on the tables that receive the data before you run **dbload**. You can create new indexes after **dbload** has finished.
Delimited identifiers	You can use delimited identifiers with the **dbload** utility. The utility detects database objects that are keywords, mixed case, or have special characters, and places double quotes around them. If your most recent **dbload** session ended prematurely, specify the starting line number in the command-line syntax to resume loading with the next record in the file.

Command file for the dbload utility

Before you use the **dbload** utility, you must create a command file that names the input data files and the tables that receive the data. The command file maps fields from one or more input files into columns of one or more tables within your database.

The command file contains only FILE and INSERT statements. Each FILE statement names an input data file. The FILE statement also defines the data fields from the input file that are inserted into the table. Each INSERT statement names a table to receive the data. The INSERT statement also defines how **dbload** places the data that is described in the FILE statement into the table columns.

Within the command file, the FILE statement can appear in these forms:

- Delimiter form
- Character-position form

The FILE statement has a size limit of 4,096 bytes.

Use the delimiter form of the FILE statement when every field in the input data row uses the same delimiter and every row ends with a new-line character. This format is typical of data rows with variable-length fields. You can also use the delimiter form of the FILE statement with fixed-length fields as long as the data rows meet the delimiter and new line requirements. The delimiter form of the FILE and INSERT statements is easier to use than the character-position form.

Use the character-position form of the FILE statement when you cannot rely on delimiters and you must identify the input data fields by character position within the input row. For example, use this form to indicate that the first input data field begins at character position `1` and continues until character position `20`. You can also use this form if you must translate a character string into a null value. For example, if your input data file uses a sequence of blanks to indicate a null value, you must use this form if you want to instruct **dbload** to substitute null at every occurrence of the blank-character string.

You can use both forms of the FILE statement in a single command file. For clarity, however, the two forms are described separately in sections that follow.

Delimiter form of the FILE and INSERT statements

The FILE and INSERT statements that define information for the **dbload** utility can appear in a delimiter form.

The following example of a **dbload** command file illustrates a simple delimiter form of the FILE and INSERT statements. The example is based on the **stores_demo** database. An UNLOAD statement created the three input data files, **stock.unl**, **customer.unl**, and **manufact.unl**.

```
FILE stock.unl DELIMITER '|' 6;
INSERT INTO stock;
FILE customer.unl DELIMITER '|' 10;
INSERT INTO customer;
FILE manufact.unl DELIMITER '|' 3;
INSERT INTO manufact;
```

To see the **.unl** input data files, refer to the directory **$ONEDB_HOME/demo/*prod_name*** (UNIX™ or Linux™) or **%ONEDB_HOME%\demo\*prod_name*** (Windows™).

Syntax for the delimiter form

The syntax for the delimiter form specifies the field delimiter, the input file, and the number of fields in each row of data.

The following diagram shows the syntax of the delimiter FILE statement.

FILE*filename***DELIMITER** ' *c* ' *nfields*

Element	Purpose	Key Considerations
c	Specifies the character as the field delimiter for the specific input file	If the delimiter specified by *c* appears as a literal character anywhere in the input file, the character must be preceded with a backslash (\) in the input file. For example, if the value of *c* is specified as a square bracket ([) , you must place a backslash before any literal square bracket that appears in the input file. Similarly, you must precede any backslash that appears in the input file with an additional backslash. You can specify any printable character, as defined by current locale, the tab character `TAB` (`CTRL-I`), or a blank space (ASCII 32) as the delimiter symbol. You cannot specify non-printable character, a hexadecimal character, or a backslash character.
filename	Specifies the input file	None.
nfields	Indicates the number of fields in each data row	None.

The **dbload** utility assigns the sequential names **f01**, **f02**, **f03**, and so on to fields in the input file. You cannot see these names, but if you refer to these fields to specify a value list in an associated INSERT statement, you must use the **f01**, **f02**, **f03** format. For details, refer to How to write a dbload command file in delimiter form on page 55.

Two consecutive delimiters define a null field. As a precaution, you can place a delimiter immediately before the new-line character that marks the end of each data row. If the last field of a data row has data, you must use a delimiter. If you omit this delimiter, an error results whenever the last field of a data row is not empty.

Inserted data types correspond to the explicit or default column list. If the data field width is different from its corresponding character column width, the data is made to fit. That is, inserted values are padded with blanks if the data is not wide enough for the column or truncated if the data is too wide for the column.

If the number of columns named is fewer than the number of columns in the table, **dbload** inserts the default value that was specified when the table was created for the unnamed columns. If no default value is specified, **dbload** attempts to insert a null value. If the attempt violates a not null restriction or a unique constraint, the insert fails, and an error message is returned.

If the INSERT statement omits the column names, the default INSERT specification is every column in the named table. If the INSERT statement omits the VALUES clause, the default INSERT specification is every field of the previous FILE statement.

An error results if the number of column names listed (or implied by default) does not match the number of values listed (or implied by default).

The syntax of **dbload** INSERT statements resembles INSERT statements in SQL, except that in **dbload**, INSERT statements cannot incorporate SELECT statements.

Do not use the CURRENT, TODAY, and USER keywords of the INSERT INTO statement in a **dbload** command file; they are not supported in the **dbload** command file. These keywords are supported in SQL only.

For example, the following **dbload** command is not supported:

```
FILE "testtbl2.unl" DELIMITER '|' 1;
            INSERT INTO testtbl
                 (testuser, testtime, testfield)
              VALUES
                 ('kae', CURRENT, f01);
```

Load the existing data first and then write an SQL query to insert or update the data with the current time, date, or user login. You could write the following SQL statement:

```
INSERT INTO testtbl
               (testuser, testtime, testfield)
            VALUES
               ('kae', CURRENT, f01);
```

The CURRENT keyword returns the system date and time. The TODAY keyword returns the system date. The USER keyword returns the user login name.

The following diagram shows the syntax of the **dbload** INSERT statement for delimiter form.

```
INSERT INTO [ owner . ] table [ ( column ) ] [ <VALUES clause> (explicit id) ] ;
```

Element	Purpose	Key Considerations
column	Specifies the column that receives the new data	None.
owner.	Specifies the user name of the table owner	None.
table	Specifies the table that receives the new data	None.

Users who run **dbload** with this command file must have the Insert privilege on the named table.

How to write a dbload command file in delimiter form

Command files must contain required elements, including delimiters.

The FILE statement in the following example describes the **stock.unl** data rows as composed of six fields separated by a vertical bar (|) as the delimiter.

```
FILE stock.unl DELIMITER '|' 6;
INSERT INTO stock;
```

The last field on each line may optionally have a delimiter after it. Two consecutive delimiters define a null field.

Compare the FILE statement with the data rows in the following example, which appear in the input file **stock.unl**.

```
1|SMT|baseball gloves|450.00|case|10 gloves/case
2|HRO|baseball|126.00|case|24/case
3|SHK|baseball bat|240.00|case|12/case
```

The example INSERT statement is minimal; it contains only the required elements. Because the column list is omitted, the INSERT statement implies that values are to be inserted into every column in the **stock** table in the sequence defined in the CREATE TABLE statement. Because the VALUES clause is omitted, the INSERT statement implies that the input values for every field are defined in the most recent FILE statement in the same order in the CREATE TABLE statement. This INSERT statement is valid because the **stock** table contains six columns, which correspond to the number of values that the FILE statement defines.

The following example shows the first data row that is inserted into **stock** from this INSERT statement.

Field	Column	Value
f01	stock_num	1
f02	manu_code	SMT
f03	description	baseball gloves
f04	unit_price	450.00
f05	unit	case
f06	unit_descr	10 gloves/case

The FILE and INSERT statement in the following example illustrates a more complex INSERT statement syntax:

```
FILE stock.unl DELIMITER '|' 6;
INSERT INTO new_stock (col1, col2, col3, col5, col6)
   VALUES (f01, f03, f02, f05, 'autographed');
```

In this example, the VALUES clause uses the field names that **dbload** assigns automatically. You must reference the automatically assigned field names with the letter **f** followed by a number: **f01**, **f02**, **f10**, **f100**, **f999**, **f1000**, and so on. All other formats are incorrect.

Tip: The first nine fields must include a zero: f01, f02, ..., f09.

The user changed the column names, the order of the data, and the meaning of **col6** in the new **stock** table. Because the fourth column in **new_stock** (**col4**) is not named in the column list, the new data row contains a null value in the **col4** position (assuming that the column permits null values). If no default is specified for **col4**, the inserted value is null.

The following table shows the first data row that is inserted into **new_stock** from this INSERT statement.

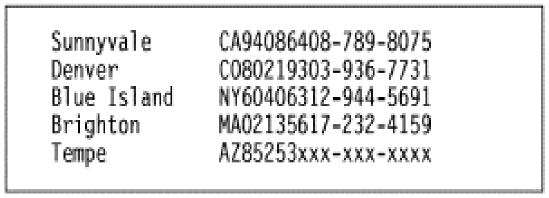
Column	Value
col1	1
col2	baseball gloves
col3	SMT
col4	null
col5	case
col6	autographed

Character-position form of the FILE and INSERT statements

The FILE and INSERT statements that define information for the **dbload** utility can appear in a character-position form.

The examples in this topic are based on an input data file, **cust_loc_data**, which contains the last four columns (**city**, **state**, **zipcode**, and **phone**) of the **customer** table. Fields in the input file are padded with blanks to create data rows in which the location of data fields and the number of characters are the same across all rows. The definitions for these fields are CHAR(15), CHAR(2), CHAR(5), and CHAR(12), respectively. Figure 1: A Sample Data File on page 56 displays the character positions and five example data rows from the **cust_loc_data** file.

Figure 1. A Sample Data File



```
Sunnyvale      CA94086408-789-8075
Denver         CO80219303-936-7731
Blue Island    NY60406312-944-5691
Brighton       MA02135617-232-4159
Tempe          AZ85253xxx-xxx-xxxx
```

The following example of a **dbload** command file illustrates the character-position form of the FILE and INSERT statements. The example includes two new tables, **cust_address** and **cust_sort**, to receive the data. For the purpose of this example, **cust_address** contains four columns, the second of which is omitted from the column list. The **cust_sort** table contains two columns.

```
FILE cust_loc_data
   (city 1-15,
    state 16-17,
    area_cd 23-25 NULL = 'xxx',
    phone 23-34 NULL = 'xxx-xxx-xxxx',
    zip 18-22,
    state_area 16-17 : 23-25);
INSERT INTO cust_address (col1, col3, col4)
   VALUES (city, state, zip);
INSERT INTO cust_sort
   VALUES (area_cd, zip);
```

Syntax for the character-position form

The syntax for the character-position form specifies information that includes the character position within a data row that starts a range of character positions and the character position that ends a range of character positions.

The following diagram shows the syntax of the character-position FILE statement.

FILE*filename*(*fieldn* [*start* [-*end*]] [NULL=' *null string* '])

Element	Purpose	Key Considerations
-end	Indicates the character position within a data row that ends a range of character positions	A hyphen must precede the *end* value.
fieldn	Assigns a name to the data field that you are defining with the range of character positions	None.
filename	Specifies the name of the input file	None.
null string	Specifies the data value for which **dbload** must substitute a null value	Must be a quoted string.
start	Indicates the character position within a data row that starts a range of character positions. If you specify *start* without *end*, it represents a single character.	None.

You can repeat the same character position in a data-field definition or in different fields.

The *null string* scope of reference is the data field for which you define it. You can define an explicit null string for each field that allows null entries.

Inserted data types correspond to the explicit or default column list. If the data-field width is different from its corresponding character column, inserted values are padded with blanks if the column is wider, or inserted values are truncated if the field is wider.

If the number of fields named is fewer than the number of columns in the table, **dbload** inserts the default value that is specified for the unnamed fields. If no default value is specified, **dbload** attempts to insert a null value. If the attempt violates a not-null restriction or a unique constraint, the insert fails, and an error message is returned.

If the INSERT statement omits the column names, the default INSERT specification is every column in the named table. If the INSERT statement omits the VALUES clause, the default INSERT specification is every field of the previous FILE statement.

An error results if the number of column names listed (or implied by default) does not match the number of values listed (or implied by default).

The syntax of **dbload** INSERT statements resembles INSERT statements in SQL, except that in **dbload**, INSERT statements cannot incorporate SELECT statements. The following diagram shows the syntax of the **dbload** INSERT statement for character-position form.

INSERT INTO [*owner* .] *table* [(*column*)] [<VALUES clause> (explicit id)] ;

Element	Purpose	Key Considerations
column	Specifies the column that receives the new data	None.
owner.	Specifies the user name of the table owner	None.
table	Specifies the table that receives the new data	None.

The syntax for character-position form is identical to the syntax for delimiter form.

The user who runs **dbload** with this command file must have the Insert privilege on the named table.

How to write a dbload command file in character-position form

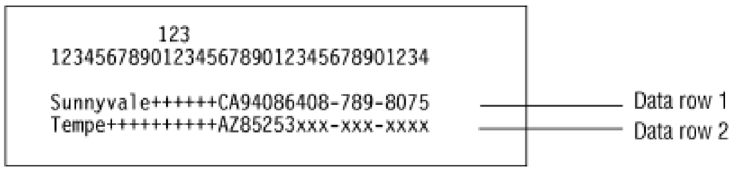
Command files must define data fields and use character positions to define the length of each field.

The FILE statement in the following example defines six data fields from the **cust_loc_data** table data rows.

```
FILE cust_loc_data
   (city 1-15,
    state 16-17,
    area_cd 23-25 NULL = 'xxx',
    phone 23-34 NULL = 'xxx-xxx-xxxx',
    zip 18-22,
    state_area 16-17 : 23-25);
INSERT INTO cust_address (col1, col3, col4)
   VALUES (city, state, zip);
```

The statement names the fields and uses character positions to define the length of each field. Compare the FILE statement in the preceding example with the data rows in the following figure.

Figure 2. A Sample Data File



```
          123
1234567890123456789012345678901234

Sunnyvale++++++CA94086408-789-8075
Tempe++++++++++AZ85253xxx-xxx-xxxx
```

The FILE statement defines the following data fields, which are derived from the data rows in the sample data file.

Column	Values from Data Row 1	Values from Data Row 2
city	Sunnyvale`++++++`	Tempe`++++++++++`
state	CA	AZ
area_cd	408	null
phone	408-789-8075	null
zip	94086	85253
state_area	CA408	AZxxx

The null strings that are defined for the **phone** and **area_cd** fields generate the null values in those columns, but they do not affect the values that are stored in the **state_area** column.

The INSERT statement uses the field names and values that are derived from the FILE statement as the value-list input. Consider the following INSERT statement:

```
INSERT INTO cust_address (col1, col3, col4)
   VALUES (city, state, zip);
```

The INSERT statement uses the data in the sample data file and the FILE statement to put the following information into the **cust_address** table.

Column	Values from Data Row 1	Values from Data Row 2
col1	Sunnyvale`++++++`	Tempe`++++++++++`
col2	null	null
col3	CA	AZ
col4	94086	85253

Because the second column (**col2**) in **cust_address** is not named, the new data row contains a null (assuming that the column permits nulls).

Consider the following INSERT statement:

```
INSERT INTO cust_sort
   VALUES (area_cd, zip);
```

This INSERT statement inserts the following data rows into the **cust_sort** table.

Column	Values from Data Row 1	Values from Data Row 2
col1	408	null
col2	94086	85253

Because no column list is provided, **dbload** reads the names of all the columns in **cust_sort** from the system catalog. (You cannot insert data into a temporary table because temporary tables are not entered into the system catalog.) Field names from the previous FILE statement specify the values to load into each column. You do not need one FILE statement for each INSERT statement.

Command file to load complex data types

You can create **dbload** command files that load columns containing complex data types into tables.

You can use **dbload** with the following data types:

- A BLOB or CLOB
- A SET inside a ROW type

The **dbload** utility does not work with the following data types:

- A CLOB or BLOB inside a ROW type
- A ROW type inside a SET

Important: All the load utilities (**dbexport**, **dbimport**, **dbload**, **onload**, **onunload**, and **onxfer**) rely on an export and import function. If you do not define this function when you write a user-defined data type, you cannot use these utilities.

Loading a new data type inside another data type can cause problems if the representation of the data contains handles. If a string represents the data, you must be able to load it.

You can use **dbload** with named row types, unnamed row types, sets, and lists.

Using the dbload utility with named row types

The procedure for using the **dbload** utility with named row types is somewhat different than the procedure for using **dbload** with other complex data types, because named row types are actually user-defined data types.

Suppose you have a table named **person** that contains one column with a named row type. Also suppose that the **person_t** named row type contains six fields: **name**, **address**, **city**, **state**, **zip**, and **bdate**.

The following syntax shows how to create the named row type and the table used in this example:

```
CREATE ROW TYPE person_t
   (
      name VARCHAR(30) NOT NULL,
      address VARCHAR(20),
      city VARCHAR(20),
      state CHAR(2),
      zip VARCHAR(9),
      bdate DATE
   );
CREATE TABLE person of TYPE person_t;
```

To load data for a named row type (or for any user-defined data type)

1. Use the UNLOAD statement to unload the table to an input file. In this example, the input file sees the named row type as six separate fields:

   ```
   Brown, James|13 First St.|San Francisco|CA|94070|01/04/1940|
   Karen Smith|1820 Elm Ave #100|Fremont|CA|94502|01/13/1983|
   ```

2. Use the **dbschema** utility to capture the schema of the table and the row type. You must use the **dbschema -u** option to pick up the named row type.

   ```
        dbschema -d stores_demo -u person_t > schema.sql
        dbschema -d stores_demo -t person >> schema.sql
   ```

3. Use DB-Access to re-create the **person** table in the new database.

   For detailed steps, see Use dbschema output as DB-Access input on page 79.

4. Create the **dbload** command file. This **dbload** command file inserts two rows into the **person** table in the new database.

   ```
   FILE person.unl DELIMITER '|' 6;
   INSERT INTO person;
   ```

   This **dbload** example shows how to insert new data rows into the **person** table. The number of rows in the INSERT statement and the **dbload** command file must match:

   ```
   FILE person.unl DELIMITER '|' 6;
      INSERT INTO person
      VALUES ('Jones, Richard', '95 East Ave.',
              'Philadelphia', 'PA',
      '19115',
      '03/15/97');
   ```

5. Run the **dbload** command:

   ```
   dbload -d newdb -c uds_command -l errlog
   ```

Tip: To find the number of fields in an unloaded table that contains a named row type, count the number of fields between each vertical bar (|) delimiter.

Using the dbload utility with unnamed row types

You can use the **dbload** utility with unnamed row types, which are created with the ROW constructor and define the type of a column or field.

In the following example, the **devtest** table contains two columns with unnamed row types, **s_name** and **s_address**. The **s_name** column contains three fields: **f_name**, **m_init**, and **l_name**. The **s_address** column contains four fields: **street**, **city**, **state**, and **zip**.

```
CREATE TABLE devtest
(
s_name ROW(f_name varchar(20), m_init char(1), l_name varchar(20)
not null),
```

```
s_address ROW(street varchar(20), city varchar(20), state char(20),
zip varchar(9)
);
```

The data from the **devtest** table is unloaded into the **devtest.unl** file. Each data row contains two delimited fields, one for each unnamed row type. The ROW constructor precedes each unnamed row type, as follows:

```
ROW('Jim','K','Johnson')|ROW('10 Grove St.','Eldorado','CA','94108')|
ROW('Maria','E','Martinez')|ROW('2387 West Wilton
Ave.','Hershey','PA','17033')|
```

This **dbload** example shows how to insert data that contains unnamed row types into the **devtest** table. Put double quotes around each unnamed row type or the insert will not work.

```
FILE devtest.unl DELIMITER '|' 2;
   INSERT INTO devtest (s_name, s_address)
   VALUES ("row('Stephen', 'M', 'Wu')",
      "row('1200 Grand Ave.', 'Richmond', 'OR', '97200')");
```

Using the dbload utility with collection data types

You can use the **dbload** utility with collection data types such as SET, LIST, and MULTISET.

SET data type example

The SET data type is an unordered collection type that stores unique elements. The number of elements in a SET data type can vary, but no nulls are allowed.

The following statement creates a table in which the **children** column is defined as a SET:

```
CREATE TABLE employee
   (
      name char(30),
      address char(40),
      children SET (varchar(30) NOT NULL)
   );
```

The data from the **employee** table is unloaded into the **employee.unl** file. Each data row contains four delimited fields. The first set contains three elements (**Karen**, **Lauren**, and **Andrea**), whereas the second set contains four elements. The SET constructor precedes each SET data row.

```
    Muriel|5555 SW Merry Sailing Dr.|02/06/1926|SET{'Karen','Lauren','Andrea'}|
    Larry|1234 Indian Lane|07/31/1927|SET{'Martha', 'Melissa','Craig','Larry'}|
```

This **dbload** example shows how to insert data that contains SET data types into the **employee** table in the new database. Put double quotes around each SET data type or the insert does not work.

```
FILE employee.unl DELIMITER '|' 4;
INSERT INTO employee
VALUES ('Marvin', '10734 Pardee', '06/17/27',
   "SET{'Joe', 'Ann'}");
```

LIST data type example

The LIST data type is a collection type that stores ordered, non-unique elements; that is, it allows duplicate element values.

The following statement creates a table in which the **month_sales** column is defined as a LIST:

```
CREATE TABLE sales_person
   (
      name CHAR(30),
      month_sales LIST(MONEY NOT NULL)
   );
```

The data from the **sales_person** table is unloaded into the **sales.unl** file. Each data row contains two delimited fields, as follows:

```
Jane Doe|LIST{'4.00','20.45','000.99'}|
Big Earner|LIST{'0000.00','00000.00','999.99'}|
```

This **dbload** example shows how to insert data that contains LIST data types into the **sales_person** table in the new database. Put double quotes around each LIST data type or the insert does not work.

```
FILE sales_person.unl DELIMITER '|' 2;
INSERT INTO sales_person
VALUES ('Jenny Chow', "{587900, 600000}");
```

You can load multisets in a similar manner.

The dbschema utility

The **dbschema** utility displays the SQL statements (the *schema*) that are necessary to replicate database objects.

You can also use the **dbschema** utility for the following purposes:

- To display the distributions that the UPDATE STATISTICS statement creates.
- To display the schema for the Information Schema views
- To display the schema for creating objects such as databases, tables, sequences, synonyms, storage spaces, chunks, logs, roles, and privileges
- To display the distribution information that is stored for one or more tables in the database
- To display information about user-defined data types and row types

After you obtain the schema of a database, you can redirect the **dbschema** output to a file that you can use with DB-Access.

The **dbschema** utility is supported on all updatable secondary servers.

The **dbschema** utility is also supported on read-only secondary servers. However, the **dbschema** utility displays a warning message when running on these servers.

Attention: Use of the **dbschema** utility can increment sequence objects in the database, creating gaps in the generated numbers that might not be expected in applications that require serialized integers.

Object modes and violation detection in dbschema output

The output from the **dbschema** utility shows object modes and supports violation detection.

The **dbschema** output shows:

- The names of not-null constraints after the not-null specifications.

  You can use the output of the utility as input to create another database. If the same names were not used for not-null constraints in both databases, problems could result.
- The object mode of objects that are in the disabled state. These objects can be constraints, triggers, or indexes.
- The object mode of objects that are in the filtering state. These objects can be constraints or unique indexes.
- The violations and diagnostics tables that are associated with a base table (if violations and diagnostics tables were started for the base table).

For more information about object modes and violation detection, see the SET, START VIOLATIONS TABLE, and STOP VIOLATIONS TABLE statements in the *HCL OneDB™ Guide to SQL: Syntax*.

Guidelines for using the dbschema utility

You can use delimited identifiers with the **dbschema** utility. The **dbschema** utility detects database objects that are keywords, mixed case, or that have special characters, and the utility places double quotation marks around those keywords.

Global Language Support: You must disable SELECT triggers and correctly set GLS environment variables before using the **dbschema** utility.

When the GLS environment variables are set correctly, as the *HCL OneDB™ GLS User's Guide* describes, the **dbschema** utility can handle foreign characters.

Syntax of the dbschema command

The **dbschema** command displays the SQL statements (the *schema*) that are necessary to replicate a specified database object. The command also shows the distributions that the UPDATE STATISTICS statement creates.

dbschema { <Table options> <Database options> { <UDT options> } | [{ **-V** | **-version** }] } <Storage, space, and log options> <No owner option>

UDT options

" { (explicit id) | [{ **-u all** }] | [{ **-ua** | **-ui** } *udt_name*] } "

Table options

" { [<Tables, Views, or Procedures>(explicit id)] [<Synonyms>(explicit id)] [<Privileges>(explicit id)] | **-hd** { **all** | <Table Name>(explicit id) } } "

" [**-r** { *role* | **all** } (explicit id)] "

Database options

" [**-ss**] "

" [**-seq** { *sequence* | **all** }] "

" **-d***database* "

" [**-w***password*] "

Storage space and log options

" [**-c** [**-ns**] *file_name*(explicit id)] "

No owner option

" [**-nw**] "

Element	Purpose	Additional Information
all	Directs **dbschema** to include all the tables or sequence objects in the database, or all the user-defined data types in the display of distributions.	None.

Element	Purpose	Additional Information
-c *file_name*	Generates commands to reproduce storage spaces, chunks, physical logs, and logical logs.	If you use the **-c** element without the **-ns** element, the database server generates SQL administration API commands. If you use the **-c** element and also use the **-ns** element, the database server generates **onspaces** or **onparams** commands.
-d *database*	Specifies the database to which the schema applies. The *database* can be on a remote database server.	To use more than the simple name of the database, see Database Name on page .
filename	Specifies the name of the file that contains the **dbschema** output.	If you omit a file name, **dbschema** sends the output to the screen. If you specify a file name, **dbschema** creates a file named *filename* to contain the **dbschema** output.
-hd	Displays the distribution as data values.	If you specify the ALL keyword for the table name, the distributions for all the tables in the database are displayed.

-it	Sets the isolation type for **dbschema** while **dbschema** queries catalog tables. Isolation types are:
	
	DR = Dirty Read
	CR = Committed Read
	CS = Cursor Stability
	CRU = Committed Read with RETAIN UPDATE LOCKS
	CSU = Cursor Stability with RETAIN UPDATE LOCKS
	DRU = Dirty Read with RETAIN UPDATE LOCKS
	LC = Committed Read, Last Committed
	RR = Repeatable Read	

-l	Sets the lock mode to wait *number of* seconds for **dbschema** while **dbschema** queries catalog tables.	None.
-ns	Generates **onspaces** or **onparams** utility commands to reproduce storage spaces, chunks, physical logs, and logical logs.	The **-c** element must precede the **-ns** element in your command.

Element	Purpose	Additional Information
-nw	Generates the SQL for creating an object without the specification of an owner.	The **-nw** element is also a **dbexport** command option.
-q	Suppresses the database version from the header.	This optional element precedes other elements.
-r	Generates information about the creation of roles.	For details, see Role creation on page 75.
-seq *sequence*	Generates the DDL statement to define the specified *sequence* object	None.
-ss	Generates server-specific information	This option is ignored if no table schema is generated.
-si	Excludes the generation of index storage clauses for non-fragmented tables	This option is available only with the **-ss** option.
-sl *length*	Specifies the maximum length, in bytes, of unformatted CREATE TABLE and ALTER TABLE statements.	None.
-u all	Prints the definitions of all user-defined data types, including all functions and casts defined over the types.	Specify **-u all** to include all the user-defined data types in the list of distributions.
-ua *udt_name*	Prints the definition of a user-defined data type, including functions and casts defined over an opaque or constructor type.	None.
-ui *udt_name*	Prints the definition of a user-defined data type, including type inheritance.	None.
-V	Displays the software version number and the serial number	None.
-version	Extends the **-V** option to display additional information about the build version, host, operating system, build number and date, and the GLS version.	None.
-w *password*	Specifies the database password, if you have one.	None.

You must be the DBA or have the Connect or Resource privilege for the database before you can run **dbschema** on it.

Example

Example

The following command generates the schema with all the tables or sequence objects in the `customer` database, but without the specification of an owner:

```
dbschema -d customer all -nw
```

Database schema creation

You can create the schema for an entire database or for a portion of the database.

Use the **dbschema** utility options to perform the following actions:

- Display CREATE SYNONYM statements by owner, for a specific table or for the entire database.
- Display the CREATE TABLE, CREATE VIEW, CREATE FUNCTION, or CREATE PROCEDURE statement for a specific table or for the entire database.
- Display all GRANT privilege statements that affect a specified user or that affect all users for a database or a specific table. The user can be either a user name or role name.
- Display user-defined and row data types with or without type inheritance.
- Display the CREATE SEQUENCE statement defining the specified *sequence* object, or defining all sequence objects in the database.

When you use **dbschema** and specify only the database name, it is equivalent to using **dbschema** with all its options (except for the **-hd** and **-ss** options). In addition, if Information Schema views were created for the database, this schema is shown. For example, the following two commands are equivalent:

```
dbschema -d stores_demo
dbschema -s all -p all -t all -f all -d stores_demo
```

SERIAL fields included in CREATE TABLE statements that **dbschema** displays do not specify a starting value. New SERIAL fields created with the schema file have a starting value of `1`, regardless of their starting value in the original database. If this value is not acceptable, you must modify the schema file.

Creating schemas for databases across a UNIX™ or Linux™ network

The **dbschema -d** option creates and displays the schema for databases on a UNIX™ or Linux™ network.

You can specify a database on any accessible database server.

The following command displays the schema for the **stores_demo** database on the **finland** database server on the UNIX™ or Linux™ system console:

```
dbschema -d //finland/stores_demo
```

Changing the owner of an object

You can edit **dbschema** output to change the owner of a new object.

The **dbschema** utility uses the *owner.object* convention when it generates any CREATE TABLE, CREATE INDEX, CREATE SYNONYM, CREATE VIEW, CREATE SEQUENCE, CREATE PROCEDURE, CREATE FUNCTION, or GRANT statement, and when it reproduces any unique, referential, or check constraint. As a result, if you use the **dbschema** output to create a new object (table, index, view, procedure, constraint, sequence, or synonym), the owner of the original object owns the new object. If you want to change the owner of the new object, you must edit the **dbschema** output before you run it as an SQL script.

You can use the output of **dbschema** to create a new function if you also specify the path name to a file in which compile-time warnings are stored. This path name is displayed in the **dbschema** output.

For more information about the CREATE TABLE, CREATE INDEX, CREATE SYNONYM, CREATE VIEW, CREATE SEQUENCE, CREATE PROCEDURE, CREATE FUNCTION, and GRANT statements, see the *HCL OneDB™ Guide to SQL: Syntax*.

dbschema server-specific information

The **dbschema -ss** option generates server-specific information. The **-ss** option always generates the lock mode, extent sizes, and the dbspace name if the dbspace name is different from the database dbspace. In addition, if tables are fragmented, the **-ss** option displays information about the fragmentation strategy.

When you specify the **dbschema -ss** option, the output also displays any GRANT FRAGMENT statements that are issued for a particular user or in the entire schema.

The **-si option**, which is available only with the **-ss** option, excludes the generation of index storage clauses for non-fragmented tables.

If the dbspace contains multiple partitions, dbspace partition names appear in the output.

For information about fragment-level authority, see the GRANT FRAGMENT and REVOKE FRAGMENT statements in the *HCL OneDB™ Guide to SQL: Syntax*.

User-defined and complex data types

The **dbschema -u** option displays the definitions of any user-defined and complex data types that the database contains. The suboption **i** adds the type inheritance to the information that the **dbschema -u** option displays.

The following command displays all the user-defined and complex data types for the **stork** database:

```
dbschema -d stork -u all
```

Output from **dbschema** that ran with the specified option `-u all` might appear as the following example shows:

```
create row type 'informix'.person_t
   (
   name varchar(30, 10) not null,
   address varchar(20, 10),
   city varchar(20, 10),
   state char(2),
   zip integer,
   bdate date
   );
create row type 'informix'.employee_t
   (
   salary integer,
   manager varchar(30, 10)
   ) under person_t;
```

The following command displays the user-defined and complex data types, as well as their type inheritance for the **person_t** type in the **stork** database:

```
dbschema -d stork -ui person_t
```

Output from **dbschema** that ran with the option `-ui person_t` might appear as the following example shows:

```
create row type 'informix'.person_t
   (
   name varchar(30, 10) not null,
   address varchar(20, 10),
   city varchar(20, 10),
   state char(2),
   zip integer,
   bdate date
   );
create row type 'informix'.employee_t
   (
   salary integer,
   manager varchar(30, 10)
   ) under person_t;
create row type 'informix'.sales_rep_t
   (
   rep_num integer,
   region_num integer,
   commission decimal(16),
   home_office boolean
   ) under employee_t;
```

Sequence creation

The dbschema -seq *sequence* command generates information about sequence creation.

The following syntax diagram fragment shows sequence creation.

```
-seq { sequence | all }
```

Element	Purpose	Key Considerations
-seq *sequence*	Displays the CREATE SEQUENCE statement defining ***sequence***	None.
-seq all	Displays all CREATE SEQUENCE statements for the database	None.

Running **dbschema** with option **-seq** sequitur might produce this output:

```
CREATE SEQUENCE sequitur INCREMENT 10 START 100 NOCACHE CYCLE
```

For more information about the CREATE SEQUENCE statement, see the *HCL OneDB™ Guide to SQL: Syntax*.

Synonym creation

The dbschema -s command generates information about synonym creation.

The following syntax diagram fragment shows the creation of synonyms.

Synonyms

“ `-s` ”

“ { *ownername* | `all` } ”

Element	Purpose	Key Considerations
-s *ownername*	Displays the CREATE SYNONYM statements owned by *ownername*	None.
-s all	Displays all CREATE SYNONYM statements for the database, table, or view specified	None.

Output from **dbschema** that ran with the specified option `-s alice` might appear as the following example shows:

```
CREATE SYNONYM 'alice'.cust FOR 'alice'.customer
```

For more information about the CREATE SYNONYM statement, see the *HCL OneDB™ Guide to SQL: Syntax*.

Table, view, or procedure creation

Several **dbschema** options generate information that shows the creation of tables, views, and procedures.

The following syntax diagram shows the creation of tables, views, and procedures.

Tables, Views, or Procedures:

“ [`-t` { *table* | *view* | `all` }] ”

“ [`-t` { *table* | *view* | { `p` | `f` } `all` | `all` }] ”

Element	Purpose	Key Considerations
-f all	Limits the SQL statement output to those statements that replicate all functions and procedures	None.
-f *function*	Limits the SQL statement output to only those statements that replicate the specified function	None.

Element	Purpose	Key Considerations
-f *procedure*	Limits the SQL statement output to only those statements that replicate the specified procedure	None.
-ff all	Limits the SQL statement output to those statements that replicate all functions	None.
-fp all	Limits the SQL statement output to those statements that replicate all procedures	None.
-t *table*	Limits the SQL statement output to only those statements that replicate the specified table	None.
-t *view*	Limits the SQL statement output to only those statements that replicate the specified view	None.
-t all	Includes in the SQL statement output all statements that replicate all tables and views	None.

For more information about the CREATE PROCEDURE and CREATE FUNCTION statements, see the *HCL OneDB™ Guide to SQL: Syntax*.

Table information

The dbschema -ss command retrieves information about fragmented tables, the lock mode, and extent sizes.

The following **dbschema** output shows the expressions specified for fragmented table.

```
{ TABLE “sallyc".t1 row size = 8 number of columns = 1 index size = 0 }
create table “sallyc?.t1
(
c1 integer
) fragment by expression
(c1 < 100 ) in db1 ,
((c1 >= 100 ) AND (c1 < 200 ) ) in db2 ,
remainder in db4
extent size 16 next size 16 lock mode page;
revoke all on “sallyc".t1 from “public";
```

The following **dbschema** output shows information about partitions in partition-fragmented tables.

```
DBSCHEMA Schema Utility grant dba to "sqlqa";
```

```
{ TABLE "sqlqa".t1 row size = 24 number of columns = 2 index size = 13 }
create table "sqlqa".t1
(
c1 integer,
c2 char(20)
)
fragment by expression
partition part_1 (c1 = 10 ) in dbs1 ,
partition part_2 (c1 = 20 ) in dbs1 ,
partition part_3 (c1 = 30 ) in dbs1 ,
partition part_4 (c1 = 40 ) in dbs1 ,
partition part_5 (c1 = 50 ) in dbs1
extent size 16  next size 16 lock mode page;
```

Storage space, chunk, and log creation

The dbschema -c command generates SQL administration API commands for reproducing storage spaces, chunks, logical logs, and physical logs. If you use the dbschema -c -ns command, the database server generates **onspaces** or **onparams** utility commands for reproducing storage spaces, chunks, physical logs, and logical logs.

For example:

- Run the following command to generate a file named `dbschema1.out` that contains the commands for reproducing the storage spaces, chunks, physical logs, and logical logs in SQL Admin API format:

  ```
  dbschema -c dbschema1.out
  ```

- Run the following command to generate a file named `dbschema2.out` that contains the commands for reproducing the storage spaces, chunks, physical logs, and logical logs in **onspaces** and **onparams** utility format:

  ```
  dbschema -c -ns dbschema2.out
  ```

Optionally, specify **-q** before you specify `-c` or `-c -ns` to suppress the database version when you run the command. For example, specify:

```
dbschema -q -c -ns dbschema3.out
```

Sample output for the creation of storage spaces, chunks, and logs

The output of the **dbschema -c** or **dbschema -c -ns** commands contain all of the SQL administration API or **onspaces** and **onparams** utility commands that you can use to reproduce storage spaces, chunks, and logs.

Example of output in SQL administration API format

```
# Dbspace 1 -- Chunk 1
EXECUTE FUNCTION TASK ('create dbspace', 'rootdbs',
 '/export/home/informix/data/rootdbs1150fc4', '200000',
 '0', '2', '500', '100')

# Dbspace 2 -- Chunk 2
EXECUTE FUNCTION TASK ('create dbspace', 'datadbs1',
 '/export/home/informix/data/datadbs1150fc4', '5000000',
 '0', '2', '100', '100')

# Dbspace 3 -- Chunk 3
EXECUTE FUNCTION TASK ('create dbspace', 'datadbs2',
 '/export/home/informix/data/datadbs2150fc4', '5000000',
 '0', '2', '100', '100')

# Dbspace 4 -- Chunk 4
EXECUTE FUNCTION TASK ('create dbspace', 'datadbs3',
 '/export/home/informix/data/datadbs3_1150fc4', '80000',
 '16', '8', '400', '400')
EXECUTE FUNCTION TASK ('start mirror', 'datadbs3',
 '/export/home/informix/data/datadbs3_1150fc4', '80000',
 '16', '/export/home/informix/data/mdatadbs3_1150fc4', '16')
```

```
# Dbspace 5 -- Chunk 5
EXECUTE FUNCTION TASK ('create tempdbspace', 'tempdbs',
 '/export/home/informix/data/tempdbs_1150fc4', '1000',
 '0', '2', '100', '100')

# Dbspace 6 -- Chunk 6
EXECUTE FUNCTION TASK ('create sbspace', 'sbspace',
 '/export/home/informix/data/sbspace_1150fc4',
 '1000', '0')

# Dbspace 6 -- Chunk 7
EXECUTE FUNCTION TASK ('add chunk', 'sbspace',
 '/export/home/informix/data/sbspace_1_1150fc4',
 '1000', '0')

# Dbspace 7 -- Chunk 8
EXECUTE FUNCTION TASK ('create blobspace', 'blobdbs',
 '/export/home/informix/data/blobdbs_1150fc4',
 '1000', '0', '4')

# External Space 1
EXECUTE FUNCTION TASK ('create extspace', 'extspace',
'/export/home/informix/data/extspac_1150fc4')

# Physical Log
EXECUTE FUNCTION TASK ('alter plog', 'rootdbs', '60000')

# Logical Log 1
EXECUTE FUNCTION TASK ('add log', 'rootdbs', '10000')
```

Example of output in onspaces and onparams utility format

```
# Dbspace 1 -- Chunk 1
onspaces -c -d rootdbs -k 2 -p
 /export/home/informix/data/rootdbs1150fc4
 -o 0 -s 200000 -en 500 -ef 100

# Dbspace 2 -- Chunk 2
onspaces -c -d datadbs1 -k 2 -p
 /export/home/informix/data/datadbs1150fc4
 -o 0 -s 5000000 -en 100 -ef 100

# Dbspace 3 -- Chunk 3
onspaces -c -d datadbs2 -k 2 -p
 /export/home/informix/data/datadbs2150fc4
 -o 0 -s 5000000 -en 100 -ef 100

 Dbspace 4 -- Chunk 4
onspaces -c -d datadbs3 -k 8
 -p /export/home/informix/data/datadbs3_1150fc4
 -o 16 -s 80000 -en 400 -ef 400
 -m /export/home/informix/data/mdatadbs3_1150fc4 16

# Dbspace 5 -- Chunk 5
onspaces -c -d tempdbs -k 2 -t -p
 /export/home/informix/data/tempdbs_1150fc4 -o 0 -s 1000
```

```
# Dbspace 6 -- Chunk 6
onspaces -c -S sbspace -p
 /export/home/informix/data/sbspace_1150fc4
 -o 0 -s 1000 -Ms 500

# Dbspace 7 -- Chunk 7
onspaces -c -b blobdbs -g 4 -p
 /export/home/informix/data/blobdbs_1150fc4 -o 0 -s 1000

# External Space 1
onspaces -c -x extspace -l
 /export/home/informix/data/extspac_1150fc4

# Logical Log 1
onparams -a -d rootdbs -s 10000
```

Role creation

The dbschema -r command generates information on the creation of roles.

The following syntax diagram shows the creation of roles.

Roles

“[-r { *role* | all }]”

Element	Purpose	Key Considerations
-r *role*	Displays the CREATE ROLE and GRANT statements that are needed to replicate and grant the specified role.	You cannot specify a list of users or roles with the **-r** option. You can specify either one role or all roles.
-r all	Displays all CREATE ROLE and GRANT statements that are needed to replicate and grant all roles.	None

The following **dbschema** command and output show that the role **calen** was created and was granted to **cathl**, **judith,** and **sallyc**:

```
sharky% dbschema -r calen -d stores_demo

DBSCHEMA Schema Utility
Software Serial Number RDS#N000000
create role calen;

grant calen to cathl with grant option;
grant calen to judith ;
grant calen to sallyc ;
```

Privileges

The dbschema -p command generates information on privileges.

The following syntax diagram fragment shows privileges information.

Privileges

```
“ -p ”

“ { user | all } ”
```

Element	Purpose	Key Considerations
-p *user*	Displays the GRANT statements that grant privileges to *user*, where *user* is a user name or role name. Specify only one user or role	You cannot specify a specific list of users with the **-p** option. You can specify either one user or role, or all users and roles.
-p all	Displays the GRANT statements for all users for the database, table, or view specified, or to all roles for the table specified	None.

The output also displays any GRANT FRAGMENT statements that are issued for a specified user or role or (with the **all** option) for the entire schema.

Granting privileges

You can generate **dbschema** information about the grantor of a GRANT statement.

In the **dbschema** output, the AS keyword indicates the grantor of a GRANT statement. The following example output indicates that **norma** issued the GRANT statement:

```
GRANT ALL ON 'tom'.customer TO 'claire' AS 'norma'
```

When the GRANT and AS keywords appear in the **dbschema** output, you might need to grant privileges before you run the **dbschema** output as an SQL script. Referring to the previous example output line, the following conditions must be true before you can run the statement as part of a script:

- User **norma** must have the Connect privilege to the database.
- User **norma** must have all privileges WITH GRANT OPTION for the table **tom.customer**.

For more information about the GRANT, GRANT FRAGMENT, and REVOKE FRAGMENT statements, see the *HCL OneDB™ Guide to SQL: Syntax*.

Displaying privilege information for a role

You can generate **dbschema** information about the privileges that were granted for a particular role.

A *role* is a classification with privileges on database objects granted to the role. The DBA can assign the privileges of a related work task, such as an engineer, to a role and then grant that role to users, instead of granting the same set of privileges to every user. After a role is created, the DBA can use the GRANT statement to grant the role to users or to other roles.

For example, issue the following **dbschema** command to display privileges that were granted for the **calen** role.

```
% dbschema -p calen -d stores_demo
```

An example of information the **dbschema** utility output is:

```
grant alter on table1 to 'calen'
```

Distribution information for tables in dbschema output

The dbschema -hd command with the name of the table retrieves the distribution information that is stored for a table in a database. If you specify the ALL keyword for the table name, the distributions for all the tables in the database are displayed.

During the **dbimport** operation, distribution information is created automatically for leading indexes on non-opaque columns. Run the UPDATE STATISTICS statement in MEDIUM or HIGH mode to create distribution information about tables that have the following types of indexes:

- Virtual Index Interface (VII) or function indexes
- Indexes on columns of user-defined data types
- Indexes on columns of built-in opaque data types (such as BOOLEAN or LVARCHAR)

Output from the **dbschema** utility shows distribution information if you used the SAMPLING SIZE keywords when UPDATE STATISTICS in MEDIUM or HIGH mode ran on the table.

For information about using the UPDATE STATISTICS statement, see the *HCL OneDB™ Guide to SQL: Syntax*.

The output of **dbschema** for distributions is provided in the following parts:

- Distribution description
- Distribution information
- Overflow information

Each section of **dbschema** output is explained in the following sections. As an example, the discussion uses the following distribution for the fictional table called **invoices**. This table contains 165 rows, including duplicates.

You can generate the output for this discussion with a call to **dbschema** that is similar to the following example:

```
dbschema -hd invoices -d pubs_stores_demo
```

Example of dbschema output showing distribution information

The **dbschema** output can show the data distributions that have been created for the specified table and the date when the UPDATE STATISTICS statement that generated the distributions ran.

The follow example of **dbschema** output shows distribution information.

```
Distribution for cathl.invoices.invoice_num

High Mode, 10.000000 Resolution

--- DISTRIBUTION ---

     (                         5)
  1: (  16,     7,            11)
  2: (  16,     6,            17)
  3: (  16,     8,            25)
  4: (  16,     8,            38)
  5: (  16,     7,            52)
  6: (  16,     8,            73)
  7: (  16,    12,            95)
  8: (  16,    12,           139)
  9: (  16,    11,           182)
 10: (  10,     5,           200)

--- OVERFLOW ---

  1: (   5,                 56)
  2: (   6,                 63)
}
```

Description of the distribution information in the example

The first part of the sample **dbschema** output describes which data distributions have been created for the specified table. The name of the table is stated in the following example:

```
Distribution for cathl.invoices.invoice_num
```

The output is for the **invoices** table, which is owned by user `cathl`. This data distribution describes the column **invoice_num**. If a table has distributions that are built on more than one column, **dbschema** lists the distributions for each column separately.

The **dbschema** output also shows the date when the UPDATE STATISTICS statement that generated the distributions ran. You can use this date to tell how outdated your distributions are.

The last line of the description portion of the output describes the mode (MEDIUM or HIGH) in which the distributions were created, and the resolution. If you create the distributions with medium mode, the confidence of the sample is also listed. For example, if the UPDATE STATISTICS statement runs in HIGH mode with a resolution of `10`, the last line appears as the following example shows:

```
High Mode, 10.000000 Resolution
```

Distribution information in dbschema output

The distribution information in **dbschema** output describes the bins that are created for the distribution, the range of values in the table and in each bin, and the number of distinct values in each bin.

Consider the following example:

```
     (                        5)
  1: (  16,     7,           11)
  2: (  16,     6,           17)
  3: (  16,     8,           25)
  4: (  16,     8,           38)
  5: (  16,     7,           52)
  6: (  16,     8,           73)
  7: (  16,    12,           95)
  8: (  16,    12,          139)
  9: (  16,    11,          182)
 10: (  10,     5,          200)
```

The first value in the rightmost column is the smallest value in this column. In this example, it is `5`.

The column on the left shows the bin number, in this case `1` through `10`. The first number in parentheses shows how many values are in the bin. For this table, 10 percent of the total number of rows (`165`) is rounded down to `16`. The first number is the same for all the bins except for the last. The last row might have a smaller value, indicating that it does not have as many row values. In this example, all the bins contain 16 rows except the last one, which contains 10.

The middle column within the parentheses indicates how many distinct values are contained in this bin. Thus, if there are 11 distinct values for a 16-value bin, it implies that one or more of those values are duplicated at least once.

The right column within the parentheses is the highest value in the bin. The highest value in the last bin is also the highest value in the table. For this example, the highest value in the last bin is `200`.

Overflow information in dbschema output

The last portion of the **dbschema** output shows values that have many duplicates.

The number of duplicates of indicated values must be greater than a critical amount that is determined as approximately 25 percent of the resolution times the number of rows. If left in the general distribution data, the duplicates would skew the distribution, so they are moved from the distribution to a separate list, as the following example shows:

```
--- OVERFLOW ---

   1: (   5,            56)
   2: (   6,            63)
```

For this example, the critical amount is `0.25 * 0.10 * 165`, or `4.125`. Therefore, any value that is duplicated five or more times is listed in the overflow section. Two values in this distribution are duplicated five or more times in the table: the value `56` is duplicated five times, and the value `63` is duplicated six times.

Use dbschema output as DB-Access input

You can use the **dbschema** utility to get the schema of a database and redirect the **dbschema** output to a file. Later, you can import the file into DB-Access and use DB-Access to re-create the schema in a new database.

Inserting a table into a dbschema output file

You can insert CREATE TABLE statements into the **dbschema** output file and use this output as DB-Access input.

The following example copies the CREATE TABLE statements for the customer table into the **dbschema** output file, **tab.sql**:

```
dbschema -d db -t customer > tab.sql
```

Remove the header information about **dbschema** from the output file, **tab.sql**, and then use DB-Access to re-create the table in another database, as follows:

```
dbaccess db1 tab.sql
```

Re-creating the schema of a database

You can use **dbschema** and DB-Access to save the schema from a database and then re-create the schema in another database. A **dbschema** output file can contain the statements for creating an entire database.

To save a database schema and re-create the database:

1. Use **dbschema** to save the schema to an output file, such as **db.sql**:

   ```
   dbschema -d db > db.sql
   ```

   You can also use the **-ss** option to generate server-specific information:

   ```
   dbschema -d db -ss > db.sql
   ```
2. Remove the header information about **dbschema**, if any, from the output file.
3. Add a CREATE DATABASE statement at the beginning of the output file or use DB-Access to create a new database.
4. Use DB-Access to re-create the schema in a new database:

   ```
   dbaccess - db.sql
   ```

   When you use **db.sql** to create a database on a different database server, confirm that dbspaces exist.

Results

The databases **db** and **testdb** differ in name but have the same schema.

Migrating existing keystores

When migrating to a new version of OneDB, it may be necessary to migrate the existing keystores.

Separate keystores are used for two different purposes:

- SSL/TLS database connections
- Encrypting data at rest (EAR)

OneDB uses OpenSSL as encryption library. The installation of an appropriate version of OpenSSL therefore is a general prerequisite for using OneDB. See the machine specific notes for details on the OpenSSL requirement.

Existing keystores for SSL/TLS database connections

OneDB supports PKCS#12 format keystores for use with the OpenSSL encryption library. If you have used SSL/TLS connections before installing OneDB, you will have keystores and stash files with the keystore password. This topic describes, how such keystores can be migrated for use with SSL/TLS connections of OneDB. The steps are applicable to keystores for OneDB database server as well as for OneDB database clients.

For more information on SSL/TLS keystores, see Secure sockets layer protocol.

When migration of a keystore file is necessary:

- If your keystore has the PKCS#12 standard format, then this keystore need not be migrated. PKCS#12 format keystore files usually have ".p12" as file name extension.
- If your keystore has the IBM GSKit (Global Security Kit) proprietary CMS format, then this keystore needs to be converted to the PKCS#12 standard format. Keystore files in the CMS format usually have ".kdb" as file name extension.

  To convert the keystore:

  As the CMS format is GSKit-specific, you need the GSKit command "gsk8capicmd" (or "gsk7capicmd") in order to convert the keystore. Use a command like:

  ```
      gsk8capicmd -keydb -convert -db KEYSTOREFILE.kdb -pw PASSWORD \
        -old_format cms -new_db KEYSTOREFILE.p12 -new_pw PASSWORD \
        -new_format pkcs12
  ```

If you used a password stash file for your keystore:

- Check if a password stash file exists for your keystore. Generally, the stash file is located in the same directory and has the same file name as the keystore file with the only difference being the file name extension. A password stash file created for use with GSKit usually has file name extension ".sth". A password stash file created for use with OneDB and OpenSSL has the file name extension ".stl".
- If, for your keystore you already have a password stash file with file name extension ".stl", then this should be sufficient.
- If you only have a "*.sth" file, or if you are in doubt, create a new password stash file using the utility onkstash. Use a command like the following:

  ```
  onkstash KEYSTOREFILE.p12 PASSWORD
  ```

If you do not know the password for your already existing keystore:

If you have a "*.sth" password stash file for your keystore, then you may no longer know the password and for all keystore access rely solely on this stash file. But you need to know the password in order to run the above onkstash command.

- If you need to convert your keystore from the CMS to the PKCS#12 format, then use the option "-stashed" instead of "-pw *PASSWORD*" in the conversion command. Run the command like:

  ```
  gsk8capicmd -keydb -convert -db KEYSTOREFILE.kdb -stashed \
        -old_format cms -new_db KEYSTOREFILE.p12 -new_pw PASSWORD \
  ```

```
      -new_format pkcs12
```

By specifying the new password for the converted keystore file with "-new_pw *PASSWORD*", the password for the converted PKCS#12 keystore is known and can then be used in the subsequent onkstash command.

- If you do not need to convert your keystore because it already has the PKCS#12 format, then you need to use the GSKit utility to change the password. Instead of requiring the old password, with option "-stashed" the GSKit utility can change the password using the stash file, so that you only need to specify the new password. Run a command like:

```
gsk8capicmd -keydb -changepw -db KEYSTOREFILE.p12 -stashed \
  -new_pw PASSWORD
```

Existing Keystores for Encrypting Data at Rest

OneDB server allows you to encrypt your data that is at rest. The term "data at rest" refers to data that resides on a persistent medium, like a disk or a backup medium. Data that is in main memory, on the other hand, is not considered being data at rest, even if it stays in memory for a long time.

For the encryption of data at rest, OneDB offers two different encryption methods, Storage Space Encryption on page for database data on disk, and Integrated Backup Encryption on page for data written to a backup medium. If you have used either of these two methods with a system from which you are migrating to OneDB, you may have to perform a conversion task for the keystores used with the encryption. This is because OneDB uses OpenSSL as encryption library and OpenSSL only supports the standard PKCS#12 format for keystores. In addition, OneDB uses its own format for keystore password stash files, and this format cannot be compatible with other formats, e.g. of stash files created with GSKit.

Follow these steps:

1. Check, what type of encryption at rest you are using, and what the associated keystore files are. In your server configuration file, check for the settings of parameters DISK_ENCRYPTION, BAR_ENCRYPTION and BAR_DECRYPTION. For parameters that are active, check the value for *keystore*. The *keystore* value contains the name of the keystore file. By default, i.e. when no absolute path name is specified, these keystore files are in the `$ONEDB_HOME/etc` directory.
2. The *keystore* value in the parameters only specifies the file name, without the file name extension. Find the keystore files in the file system and check their file name extension. Keystore files with file name extension ".p12" should be in the standard PKCS#12 format. Keystores in this format can be used with OneDB without conversion.
3. If you find a keystore file that has ".kdb" as file name extension, then this is a keystore in the GSKit proprietary CMS format. If you dont' find a corresponding file with the same name, but the ".p12" file name extension, then the CMS keystore file ("*.kdb") must be converted to a PKCS#12 format keystore. As the CMS format is GSKit proprietary, you need a GSKit utility like "gsk8capicmd" or "gsk7capicmd" to perform this conversion. Use a command like the following:

```
gsk8capicmd -keydb -convert -db KEYSTOREFILE.kdb -pw PASSWORD \
      -old_format cms -new_db KEYSTOREFILE.p12 -new_pw PASSWORD \
      -new_format pkcs12
```

With the "-pw" option, you have to provide the password for the CMS keystore file ("*.kdb") and with the "-new_pw" option, you have to provide a password for the new PKCS#12 keystore. The two passwords can be same.

    - If you do not know the password for the “*.kdb” keystore file, then check if you have a password stash file. The password stash file has same name as the keystore file, but file name extension will be “.sth”. If you have the “*.sth” file for your keystore file, then you can perform the conversion without specifying the password for the “*.kdb” file. The GSKit utility can instead use the stash file to retrieve the needed password. Use a command like the following:

      ```
      gsk8capicmd -keydb -convert -db KEYSTOREFILE.kdb -stashed \
           -old_format cms -new_db KEYSTOREFILE.p12 -new_pw PASSWORD \
           -new_format pkcs12
      ```

4. Check, if you have a password stash file for your PKCS#12 keystore (original “*.p12” file or converted from a “*.kdb” file). The password stash file has the same file name as the keystore file, but a different file name extension, either “.stl” or “.sth”.
    - A password stash file with file name extension “.stl” should be usable by OneDB as-is.
    - A password file with file name extension “.sth” was created with GSKit and is in a GSKit proprietary format. In this case, you have to create a new password stash file (“*.stl”) for use with OneDB. Use the onkstore utility with a command like the following:

      ```
      onkstore -stash -file KEYSTOREFILE
      ```

      The command prompts you for the password of the *KEYSTOREFILE*.p12 file. Alternatively, you can store the password in a file and provide the name of this file on the command line by adding the option “-pw <password file>” to the above command.
        - To run the “onkstore -stash” command, you need to know the password for the PKCS#12 keystore file. If you already had the “*.p12” file, but you do not know the password for it, then check if you have a corresponding “*.sth” password stash file. If you have such a “*.sth” file, then you first need to change the password of your existing PKCS#12 keystore to a new known password. You need to use a GSKit utility to change the password, as the GSKit utility can retrieve the needed password from the “*.sth” stash file. Use a command like the following:

          ```
          gsk8capicmd -keydb -changepw -db KEYSTOREFILE.p12 -stashed \
            -new_pw PASSWORD
          ```

          After changing the password, you know the new password for the “*.p12” keystore file and can run the above onkstore -stash command to create the new password stash file for OneDB.

Index