opentext™

OpenText™ Documentum™ Content Management

High-Volume Server Development Guide

Understand the High-Volume Server features such as batch operations, currency scoping, data partitioning, and lightweight SysObjects.

EDCCS250400-DGD-EN-01

OpenText™ Documentum™ Content Management
High-Volume Server Development Guide
EDCCS250400-DGD-EN-01
Rev.: 2025-Oct-17

This documentation has been created for OpenText™ Documentum™ Content Management CE 25.4.
It is also valid for subsequent software releases unless OpenText has made newer documentation available with the product, on an OpenText website, or by any other means.

Open Text Corporation

275 Frank Tompa Drive, Waterloo, Ontario, Canada, N2L 0A1

Tel: +1-519-888-7111
Toll Free Canada/USA: 1-800-499-6544 International: +800-4996-5440
Fax: +1-519-888-0677
Support: https://support.opentext.com
For more information, visit https://www.opentext.com

Table of Contents

Chapter 1

Overview

This guide provides an overview and describes the fundamental features and behaviors of High-Volume Server. This guide also discusses the basic features of High-Volume Server in detail. This guide is intended for system and repository administrators, application programmers, and any other user who wishes to obtain an understanding of the services and behavior of High-Volume Server. The guide assumes that the reader has an understanding of OpenText™ Documentum™ Content Management Server, relational databases, object-oriented programming in Java, and Structured Query Language (SQL).

High-Volume Server is an extension of OpenText Documentum Content Management (CM) Server that supports features implemented to solve common problems with large content stores. The broad areas of enhancements that make up High-Volume Server are batch operations, currency scoping, database partitioning, and lightweight SysObjects.

1.1 Overview of batch operations and currency scoping

Inserting a single object into a repository is a resource-intensive event. There are many applications where many similar objects must be added to a repository. Batch operations can reduce the overhead by grouping these events into a batch and processing them all together. Documentum CM Server supports batch mode, such as the transaction mode, that allows you to submit new objects in a batch. Additionally, transaction mode is extended to support nested transactions.

An additional performance enhancement introduced in Documentum CM Server is setting the scoping for object and query currency checking in applications. This feature allows a developer to have finer control over when and whether to check the currency of objects and queries. In cases where the developer knows that the object or query must still be current, redundant currency checks can be eliminated.

1.2 Overview of data partitioning

Documentum CM Server supports splitting the repository across multiple partitions when used with some of the underlying relational databases that also support partitioning. This feature is useful to move older, less-frequently accessed data to slower, less expensive storage. It can also restrict queries to the subset of a repository contained on a particular partition (if known). It can also swap large amounts of data from offline tables into online partitions, greatly increasing ingestion speed. Finally, entire partitions can be deleted in one operation, providing an efficient bulk-deletion mechanism.

1.3 Overview of lightweight SysObjects

Lightweight SysObjects are part of an object model enhancement introduced to share system managed metadata among objects which only hold application-specific data. For example, policies for security, retention, and storage are stored in a shareable system object that is shared among all the lightweight objects. The system managed metadata is stored only once and it significantly reduces the disk storage requirements and improves the ingestion performance. However, to display and use lightweight SysObjects, an application must be aware of them.

Chapter 2

Batch operations

Documentum CM Server includes a batch mode that can increase the server performance when creating objects. Batch mode delays adding data to the relational database management system (RDBMS) until the batch is flushed or committed, instead of adding it for each operation.

2.1 Batch operations

Batch operations can increase performance for some applications that add groups of objects. When you add a repository object, there is a certain amount of overhead associated with adding the data in the database tables representing the new object. Using batch mode in Documentum CM Server, multiple items are not added to the RDBMS server until the batch is flushed. This means that the database set up and row locking and unlocking apply to all the objects in the batch as a whole. You gain performance improvements by handling these operations in a batch. You also gain a transactional benefit. Until the batch is committed or closed, the batch can be aborted and the changes rolled back to any previous commit. Every batch represents a transaction. When you open a batch, a transaction implicitly begins.

To support this mode, OpenText™ Documentum™ Content Management Foundation Java API has been enhanced with the addition of a batch manager to control batch operations. Methods defined for the batch manager are: open, flush, commit, abort, and close batches.

When a batch is opened, Documentum CM Server caches operations that modify the RDBMS until the batch is flushed. When flushed, the operations are performed in the database at that time. The batch continues to cache operations until the next flush. Calls to commit flush the cache, and commit the outstanding operations in the database. Calls to abort roll back the operations in the database to the last commit. Calls to close flush the cache, commit the outstanding operations in the database, and close the batch.

When you open a batch, you can choose to open a simple batch, or open a batch with groups. If an error occurs when the first type of batch is committed, none of the actions in the batch commit to the database. All of those items roll back out of the database and Documentum CM Server. If you open a batch with groups, if there is an error for an item in a group, only the members of that group roll back. The other groups without errors commit. Additionally, for a batch opened with groups, you can retrieve information about which actions failed and caused the rollback. You control whether to use groups by choosing the corresponding open batch call, `openBatchGroup`. The methods defined for the batch manager that work with groups are: `openBatchGroup`, `newGroup`, `getGroups`, and `getFailedGroups`. The `IDfBatchGroup` interface is used to examine batch groups.

An extension to transactions has been added that supports nested transactions. The earlier transaction mechanism did not allow nested transactions. Since the batch mode works similar to a transaction (in that it can be rolled back before it is committed), it was necessary to build in a transaction method that allowed nested transactions. A batch of operations can contain transactions, since the transaction used by the batch manager allows for nested transactions.

2.1.1 OpenText Documentum Content Management (CM) Foundation Java API components for batch operations

For batch operations, there is an interface for batch operations, for batch groups, and nested transactions:

- `IDfBatchManager`
- `IDfBatchGroup`
- `IDfLocalTransaction`

The following interfaces have been modified:

- `IDfSession`
- `IDfQuery`
- `IDfAuditTrailManager`
- `IDfEventManager`

2.1.1.1 IDfBatchManager

This interface provides control of the batch operation. A batch cannot be nested, so only one batch can be open at a time. Typically, you would open a batch, perform many operations, and close the batch. Other methods, flush and commit, provide more control of operations during the batch. The group methods allow you to examine any failed transactions, and allow contain any batch failures to that particular group. The batch manager supports the following methods:

- void `openBatch`(int size, boolean flushBatchOnQuery, boolean oneAuditPerBatch, boolean oneEventPerBatch, Properties options) throws DfException. This method opens a batch and sets the batch attributes.

  This method uses the following parameters:

  – size (batch size) is a hint to the server cache for the number of objects to create in the batch before flushing it. Since each type is related to several tables and repeating attributes require multiple rows, the mapping between the batch size and the number of objects created before flushing is only approximate. The server uses the batch size to determine the size of the cache to hold the tables for the objects.

  – `flushBatchOnQuery` sets whether the server must flush the batched tables when a query is issued during the batch. This flag can be overridden by an individual query by using the `IDfQuery.setFlushBatchOnQuery` method.

  - `oneAuditPerBatch` sets whether the server generates one audit trail for the batch or one audit trail per object insertion. The server will only honor this flag if the caller has audit configuration privileges.
  - `oneEventPerFlush` sets whether the server generates one event for the batch or one event per object insertion. Use this to reduce the number of events generated during object creation.
  - options contains a set of hints to the server. Currently, there is only one, `driver=bcp`. The server will ignore the options it does not support.

- void `openBatch`() throws DfException. This method is a special case of the earlier one. It opens a batch with the default batch size, with `flushBatchOnQuery` set to `true`, `oneAuditPerBatch` flag set to `false`, and `oneEventPerBatch` flag set to `false`. The default batch size can be set in `dfc.properties` by using the `dfc.batchmanager.max_batch_size` parameter.
- void `openBatchGroup`(int size, boolean flushBatchOnQuery, boolean oneAuditPerBatch, boolean oneEventPerBatch, String ignoreInsertError) throws DfException. Opens a batch with groups. When a batch is opened with this method, errors caused by data (for example, constraint violations), are kept within the batch manager. All the data from a group that contains that bad data will be removed; only groups without errors in the data will be committed into repository. The client will get an error report about the failures only after the batch commit. Those errors will not interrupt the client operations. Other failures will still be reported back to client synchronously. Batch manager also starts to combine all object creation and row inserts into a collection. Any failure in the collection will cause all data in the collection to be removed from the repository. Most of the parameters are described in the `openBatch` method, except for:
  - `ignoreInsertError` is a comma separated list of registered tables. Insert errors from these tables will be ignored, but errors from tables not in the list will roll back the batch transaction.
- void `newGroup`() throws DfException. Informs the batch manager to close the current group and open a new group. Nothing happens if the batch was not opened with a group.
- List`<IDfBatchGroup> getGroups`(). Gets a list of all the groups in the current transaction. The `commitBatch` and `closeBatch` methods clear the group information.
- List`<IDfBatchGroup> getFailedGroups`(). Gets a list of the groups that have errors and were removed from the batch. This method returns null if there are no errors. You can retrieve attributes of the failed objects, but because the objects have been removed from the repository, all save operations will fail. You should treat these objects as read-only. The result of this call is only valid after `commitBatch()`. Any insert after the `commitBatch()` call will reset the error from the previous `commitBatch()`. You should call the method immediately after `commitBatch()`.
- void `flushBatch`() throws DfException. This method flushes the batch's data in the server cache to the database. This clears the cache, but until the batch is

committed, the data is not available to other sessions. This method does not affect grouping.

- void `commitBatch`() throws DfException. This method flushes the batch's data in the server cache to the database and then commits the transaction. Essentially, this is equivalent to issuing a `closeBatch()` call followed by an `openBatch()` call. If there is a group, the current group will be closed and a new one will be opened.
- void `abortBatch`() throws DfException. This method clears up any outstanding entries in the batch, rolls back the transaction to the status right after the previous `commitBatch`, and closes the batch.
- void `closeBatch`() throws DfException. This method flushes the batch's data in the server cache to the database, commits the transaction, and then closes the batch.
- boolean `isOneAuditPerBatch`(). This method indicates whether there is one audit trail generated for the batch per batch size.
- boolean `isOneEventPerBatch`(). This method indicates whether there is only event notification (queue item) generated for the batch per batch size.
- boolean `isBatchActive`(). This method indicates whether the current session or session manager has turned on the batch mode, in other words, if there is an active batch.
- boolean `isFlushBatchOnQuery`(). This method indicates whether the server needs to flush the batched tables if a query is issued during the batch.
- int `getMaxBatchSize`(). This method returns the batch size.

If either of the `openBatch` methods is called while `isBatchActive` is `true`, `the DFC_BATCHMANAGER_BATCH_IN_PROGRESS` exception will be thrown.

2.1.1.2 IDfBatchGroup

Use the following methods to examine items in the group. If the group was retrieved by using the getFailedGroups method, you should treat the objects as read-only. If the batch is committed or closed, the group information is cleared.

- Collection<IPersistentObject> *getObjects*(). Returns the newly created objects in a group.
- List<String> *getInsertStmt*(). Returns the insert statements for registered tables.
- List<IPersistentObject> *getFailedObjects()*. Returns the error messages for the failed objects.

2.1.1.3 IDfLocalTransaction

An IDfLocalTransaction object is the handle to a nestable transaction. It is used with methods from IDfSession.

2.1.1.4 IDfSession

For batch operations, there are new methods added to the IDfSession interface. Unlike existing transaction methods, the new transaction methods can be nested. IDfSession supports the following new methods:

- IDfLocalTransaction *beginTransEx*() throws DfException. This method begins a transaction and returns a IDfLocalTransaction object. This method supports nested transactions, and internally uses a transaction stack to keep track of the current transactions.
- void *abortTransEx*(IDfLocalTransaction tx) throws DfException. This method checks if the transaction is on the stack. If so, the transaction is rolled back (aborted) and if there is any open batch or transaction above the transaction on the transaction stack, that batch is also aborted. Otherwise, if the transaction is not on the stack, a DFC_WRONG_TRANSACTION_SCOPE exception will be thrown.
- void *commitTransEx*(IDfLocalTransaction tx) throws DfException. This method checks if the transaction is on top of the stack. If so, then the transaction is committed; if not, then a DFC_WRONG_TRANSACTION_SCOPE exception will be thrown.
- IDfBatchManager *getBatchManager*(). This method will return the IDfBatchManager object of the session.

2.1.1.5 IDfQuery

The method added to IDfQuery allows a client to override the flag set by the IDfBatchManager on a per query basis. IDfQuery supports the void *setFlushBatchOnQuery*(boolean flushBatch) method:

This method specifies whether this query will cause the open batch to flush. Since the batch caches the inserted data, the batch must flush the data so that query can return the changes made in the batch. When set to true, the batch will flush the data whenever there is a query on the type or table that is in the batch. If the caller knows that a query result is not in the data in the batch (or does not care about the data currently in the batch), set the flag to false to avoid an unnecessary batch flush. This method allows clients to override the flushBatchOnQuery flag set in the IDfBatchManager on a per query basis. If not set, the query will use the flag set in the batch manager.

2.1.1.6 IDfAuditTrailManager

The following new method is added to support batch auditing:

List<IDfPair<IDfId,String>> *parseBatchedIds*(IDfId audittrailId) throws DfException. This method will parse the ID list value in an audit trail specified by the audittrailId and return an IDfList with the list of pairs <IDfId, String> for each object ID and the operation applied to it.

2.1.1.7 IDfEventManager

This new interface and method is added to support event notification to be used by the full-text engine. During batch insert, multiple events will be written into one queue item if oneEventPerBatch is specified.

List<IDfPair<IDfId,String>> *parseBatchedIds*(IDfId eventId) throws DfException. This method will parse the ID list value in an event specified by the eventId and return an IDfList with the list of pairs <IDfId, String> for each object ID and the operation applied to it.

2.1.2 Batch example

The example is a basic Foundation Java API program that uses a loop to create some objects. The loop is surrounded by calls to openBatch() and closeBatch(). Just using this simple enhancement to a client can provide modest performance gains. The batch code is relative easy to add to current applications. The following is a snippet from the example program that sets up, opens, and closes the batch:

```
import com.documentum.fc.client.IDfBatchManager;
...
            bMgr = session.getBatchManager();
            bMgr.openBatch();
            // Batched client operations
            ...
            bMgr.closeBatch();
```

The following is a simple standalone example program demonstrating batching. Be sure to have a copy of `dfc.properties` in the classpath for the program:

```
/*
 * StdBatchingExample.java
 * This example assumes that you have a standard type, "test_payment_check"
 * already created.  One way to create this type is to use the following
 * DQL statement:
 CREATE TYPE "test_payment_check" (
 account integer,
 check_number integer,
 transaction_date date,
 amount float,
 bank_code integer,
 routing integer
 ) WITH SUPERTYPE "dm_document" PUBLISH
 *
 * This example also assumes that there is a check image GIF file in the
 * working directory (use any GIF to stand in for a check image), and that
 * the repository name, username, and password are passed to the program
 * on the command line.
 */
import com.documentum.com.DfClientX;
import com.documentum.com.IDfClientX;
import com.documentum.fc.client.IDfClient;
import com.documentum.fc.client.IDfSession;
import com.documentum.fc.client.IDfSessionManager;
import com.documentum.fc.client.IDfBatchManager;
import com.documentum.fc.client.IDfSysObject;
import com.documentum.fc.common.*;
public class StdBatchingExample {
    public void begin(String username, String password, String docbaseName) {
        IDfClientX clientx;
        IDfClient  client;
        IDfSession session = null;
        IDfSessionManager sMgr = null;
        IDfBatchManager bMgr = null;
        IDfSysObject sysObj;
        try {
            //create a Client object
            clientx = new DfClientX();
            client = clientx.getLocalClient();

            //create a Session Manager object
            sMgr = client.newSessionManager();

            //create an IDfLoginInfo object
            IDfLoginInfo loginInfoObj = clientx.getLoginInfo();
            loginInfoObj.setUser(username);
            loginInfoObj.setPassword(password);

            //bind the Session Manager to the login info
            sMgr.setIdentity(docbaseName, loginInfoObj);
            session = sMgr.getSession(docbaseName);

            // get the Batch Manager for this session
            bMgr = session.getBatchManager();

            //Open a batch to create the objects
            bMgr.openBatch();

            for (int i=0; i<10; i++) {

                IDfTime currDate = new DfTime();
                sysObj = (IDfSysObject)session.newObject("test_payment_check");
                sysObj.setObjectName("test_payment_check_" + i);
                sysObj.setStorageType("filestore_01");
                sysObj.setContentType("gif");
                sysObj.setFile("check.gif");

                // set individual attributes
                sysObj.setInt("check_number",i);
                sysObj.setTime("transaction_date", currDate);
                sysObj.setInt("bank_code", 1);
                sysObj.setInt("routing", 231381116);
                sysObj.setInt("account", 1000000);
                sysObj.setDouble("amount", 25.00);

                sysObj.save();
            }
            // Close Batch and commit objects to repository
            bMgr.closeBatch();
        }
        catch(Throwable e) {
            if(e instanceof DfException) {
                System.out.println("DFC Exception:");
                String s = ((DfException)e).getStackTraceAsString();
                System.out.println(s);}
                    else {
                System.out.println("Non-DFC Exception");
                e.printStackTrace();
            }
        }
        finally {
                if (session != null)
                sMgr.release(session);
        }
    }
    public static void main(String[] args)
    throws java.lang.InterruptedException {
        String docbase = args[0];
        String user = args[1];
        String password = args[2];
        new StdBatchingExample().begin(user, password, docbase);
    }
}
```

Chapter 3

Currency scoping

Documentum CM Server currency scoping is a feature that controls object currency checking by using a scope manager. When objects are fetched or queries are run, they are saved temporarily for use by the client. How long a object or query can be used until it has to be refetched or rerun is managed by currency checking.

3.1 Currency scoping

Another Documentum CM Server feature is control over object currency checking by using a scope manager. When objects are fetched or queries are run, they are saved temporarily for use by the client. How long a object or query can be used until it has to be refetched or rerun is managed by currency checking. Objects and queries that do not change for the duration of a transaction, batch, or section of a client code will not be refetched, and are good candidates to benefit from currency scoping. The scope manager controls the checking of object currency, so that objects are checked once during the currently defined scope. The assumption is that the object or query will not change during the scope. This reduces the remote procedure call (RPC) communication traffic between Foundation Java API and Documentum CM Server and reduces redundant policy checks on objects.

3.1.1 Foundation Java API components for currency scoping

Currency checking is used to tell whether a cached object should be retrieved again from the repository, or if a query should be run again against the repository. In many cases, the developer knows that an object or query will not become stale (or won't care) during a particular series of operations, so setting a scope for currency checking (currency scoping) can reduce unnecessary retrievals from the repository. Currency scoping affects the IDfSession.getObjectWithCaching() method when you specify IDfSession.CURRENCY_CHECK_ONCE_PER_SCOPE as the value of currencyCheckValue, and the IDfQuery.execute() method when you specify IDfSession.CURRENCY_CHECK_ONCE_PER_SCOPE using the IDfQuery.setCurrencyCheckValue() method.

For currency scoping, there are new interfaces for controlling scope:

- IDfScopeManager
- IDfSessionScopeManager
- IDfScope

Methods to access the scope manager have been added to the following interfaces:

- IDfSession

- IDfSessionManager

A new enumerated type has also been added:

```
Enum ScopeBoundary
   {
     OPERATION,
     TRANSACTION,
     BATCH,
     CUSTOM
   }
```

IDfScopeManager and its methods control the currency scoping for each of the sessions managed by IDfSessionManager. Each session manager has one scope manager. The IDfScopeManager sets the scope boundaries for all the sessions of that session manager. Within a session, the IDfSessionScopeManager can set custom scope boundaries for that individual session.

Initially, no scope boundaries are set. After you set a scope boundary, a scope is pushed onto the scope stack when program execution crosses that boundary. For example, if you set TRANSACTION and BATCH scope boundaries, then scopes are pushed onto the stack whenever a transaction or a batch is entered. For query or object fetches set for CURRENCY_CHECK_ONCE_PER_SCOPE, the time is compared with that of the current scope's timestamp on the stack. If the query or fetch occurred since the current scope's timestamp, then the cached version is used. When the current scope boundary is exited, the scope is popped from the scope stack.

There are four possible scope boundaries, OPERATION, TRANSACTION, BATCH, and CUSTOM. The CUSTOM boundaries are set by using the beginScope() and endScope() methods in either the IDfScopeManager or the IDfSessionScopeManager interface.

It is possible for one currency scope to be opened inside of another. For example, if BATCH and TRANSACTION boundaries are set, and you begin a batch and then begin a transaction there is a TRANSACTION scope in the BATCH scope. You can control how the scope timestamp is set when a scope is opened inside another, either setting a new timestamp or using the timestamp of the enclosing scope. If you set the bNested parameter to true, a new timestamp is set for the scope; if you set bNested to false, the timestamp of the enclosing scope is used for the new scope.

Currency scoping can also be enabled on the server side. Similar to the client-side scoping, server-side scoping can improve performance, especially for queries and objects that will remain current while the client application is executing. Many server-side operations have been optimized to be able to take advantage of server-side currency scoping. As a developer, your only control is to enable or disable server-side scoping. However, different sessions created with the session manager can have different values. After changing the value of the enable parameter, subsequent sessions will have the new value. In most cases, you should enable server side scoping to improve performance.

3.1.1.1 IDfScopeManager

IDfScopeManager declares the public methods mentioned in this section. Each session manager will have one instance of IDfScopeManager. Scopes defined by the scope manager are effectively templates to create scopes on sessions that belong to the session manager of a scope manager instance.

- void *setDefaultScope*(ScopeBoundary boundary, boolean bNested) throws DfException

  Sets a scope boundary and defines whether to set a new timestamp when the boundary is crossed. The choices for scope boundary are OPERATION, TRANSACTION, or BATCH:

  – OPERATION—The boundary is the execution of an IDfOperation method.

  – BATCH—The boundary is the batch transaction existing between calls to openBatch and to either commitBatch, closeBatch, or abortBatch.

  – TRANSACTION—The boundary is entered during a call to beginTrans or beginTransEx, or can also be entered when a session is created through the session manager (if a transaction was started in the session manager with beginTransaction and the session is being created on the same thread that called beginTransaction). The TRANSACTION boundary begins when the session enters a transaction and ends when the transaction is committed or aborted.

  If bNested is false, the timestamp of the scope is inherited from the outer parent scope. Otherwise, the timestamp is the current time. Nested custom scopes are not supported by the scope manager (but are supported by the session scope manager). You cannot reset a scope if there are sessions that have the corresponding scope active. For example, if the TRANSACTION scope boundary is set when a session begins a transaction, that session will have an active scope based on that default scope. Until the session aborts or commits the transaction, thereby ending the scope, the default scope of TRANSACTION cannot be changed. This applies to all sessions of the session manager.

- IDfScope *getDefaultScope*(ScopeBoundary boundary)

  Returns the default scope template for the specified boundary.

- IDfScope *clearDefaultScope*(ScopeBoundary boundary)

  Removes the scope boundary. A new scope will not be entered when program execution crosses the boundary. This undoes the effect of the setDefaultScope() method. You cannot remove a scope boundary when there are sessions that have the corresponding scope active.

- IDfScope *beginScope*(boolean bNested) throws DfException

  Creates an explicit CUSTOM scope boundary to be applied to future sessions created through the session manager. If bNested is false, the timestamp of the scope is inherited from the outer parent scope. Otherwise, the timestamp is the current time. The scope manager does not support nested custom scopes; an exception is thrown if a custom scope already exists (the session scope manager does support nested scopes). If server scoping is enabled, and bNested is TRUE, the CUSTOM scope is pushed to the server.

- void *endScope*() throws DfException

  Close the current custom scope (as returned by the getCurrentScope() method). For each session whose current scope is this CUSTOM scope being closed, a call is made to popScope on that session. If any session has this custom scope on its stack but it's not the current scope, an exception is thrown and no action is taken on any sessions.

- boolean *isScopeActive*()

  Returns true if an explicit CUSTOM scope has been opened with the beginScope method.

- void *enableServerScoping*(boolean enable) throws DfException

  Enable or disable server side scoping based on the value of the parameter. The value is propagated to each session scope manager as sessions are created through the session manager.

- boolean *isServerScopingEnabled*()

  Returns TRUE if server side scoping is enabled.

- IDfScope *getCurrentScope*()

  This method returns the most recently created scope.

- int *getScopeCount*()

  Returns the number of elements on the scope stack.

- void *clearScopes*() throws DfException

  For each custom scope defined by this scope manager, a call is made to pop the scope off the stack for each session which has that scope as its current scope. If a TRANSACTION scope is defined by this scope manager (as a result of having opened a transaction with beginTransaction), no action is taken and an exception is thrown.

3.1.1.2 IDfSessionScopeManager

IDfSessionScopeManager declares methods that an application will use to access scopes from sessions. Each session has an instance of IDfSessionScopeManager.

- IDfScope *beginScope*(boolean bNested) throws DfException

  This method will create an explicit custom scope and push it onto the stack. This scope is local to the session and is closed by calling endScope. If bNested is false, the timestamp of the scope is inherited from the outer parent scope. Otherwise, the timestamp is the current time. Nested custom scopes are supported by the session scope manager (but not the scope manager).

- void *endScope*() throws DfException

  Pops the current custom scope off the stack. An exception is thrown if the current scope is not a custom scope or if the current scope is an explicit custom scope, or a TRANSACTION, BATCH, or OPERATION scope defined by the scope manager on the session manager.

- boolean *isScopeActive*()

  Returns true if an explicit CUSTOM scope has been opened with the beginScope method.

- void *enableServerScoping*(boolean enable) throws DfException

  Enable or disable server side scoping based on the value of the parameter. This value is applied to all new scopes created on this session. An exception is thrown if the server does not support scoping.

- boolean *isServerScopingEnabled*()

  Returns TRUE if server side scoping is enabled.

- IDfScope *getCurrentScope*()

  Returns the top element of the scope stack or NULL if the stack is empty.

- int *getScopeCount*()

  Returns the number of elements on the scope stack.

- void *clearScopes*() throws DfException

  Pops all the scopes on the scope stack. If any scopes on the stack were defined by the scope manager on the session manager, an exception is thrown and no action is taken.

3.1.1.3 IDfScope

The member methods in the Scope class are listed in this section. After you set a scope boundary, a scope is pushed onto the scope stack whenever program execution crosses that boundary. When the current scope boundary is exited, the scope is popped from the scope stack. The scope object contains the ScopeBoundary, the timestamp, and the bNested values for that scope.

- ScopeBoundary *getBoundary*()

  Returns the current operation's scope boundary value. The values of the enum ScopeBoundary are: OPERATION, TRANSACTION, BATCH, and CUSTOM.

- long *getTimeStamp*()

  Returns the current timestamp.

- boolean *isCurrent*(long timeStamp)

  Returns whether the timestamp passed in, typically representing the last checked timestamp of an object or a query, is current.

- boolean *isNested*()

  Returns whether the scope is nested or not. If isNested() returns TRUE, the current scope used the then current time for its timestamp; if isNested() returns FALSE, the current scope took its timestamp from the enclosing scope.

3.1.1.4 IDfSessionManager

Use the IDfScopeManager *getScopeManager*() method to retrieve the IDfScopeManager:

This method returns the IDfScopeManager for the scopes for sessions controlled by IDfSessionManager.

3.1.1.5 IDfSession

Use the IDfSessionScopeManager *getSessionScopeManager*() method to retrieve the IDfSessionScopeManager:

This method returns the IDfSessionScopeManager for the current session.

3.1.2 Currency scoping example

The example is a basic Foundation Java API program that repeatedly calls a method that queries the repository and creates an object. The query is one that is not likely to change during the course of a transaction or batch, so it only needs to be retrieved once at the beginning of a scope. The program first opens a session, transaction, and batch and creates one object so that the session setup overhead does not distort the non-scoping to scoping comparisons. Then the program creates one hundred objects using six scenarios: transaction without scoping, transaction with scoping, batch without scoping, batch with scoping, no batch or transaction without scoping, and no batch or transaction with scoping (custom scope).

The code to set the scope to a transaction is:

```
IDfSessionManager sMgr = null;
IDfLocalTransaction transaction;
...
    sMgr.getScopeManager().setDefaultScope(ScopeBoundary.TRANSACTION,true);
    transaction = session.beginTransEx();
    //client transaction operations
...
    session.commitTransEx(transaction);
    sMgr.getScopeManager().clearDefaultScope(ScopeBoundary.TRANSACTION);
```

In the example, the scope boundary is cleared after the transaction, but could be left in place. If not cleared, a new scope would be created if another transaction were entered later.

Similarly, to set the scope boundary to batch, use the following code:

```
IDfBatchManager bMgr = null;
...
   sMgr.getScopeManager().setDefaultScope(ScopeBoundary.BATCH, true);
   bMgr.openBatch();
   //client batch operations
...
   bMgr.closeBatch();
   sMgr.getScopeManager().clearDefaultScope(ScopeBoundary.BATCH);
```

As in the previous code snippet, the scope boundary is cleared after the batch, but could be left in place. If not cleared, a new scope would be created if another batch were entered later.

Finally, to set a custom scope boundary, use the following code:

```
session = sMgr.getSession(docbaseName);
...
   session.getSessionScopeManager().beginScope(true);
   //client operations
...
   session.getSessionScopeManager().endScope();
```

Notice that the custom scope on the current session is set using the Session Scope Manager. If the Scope Manager is used to set a custom scope, the scope begins when a new session is created by the Session Manager associated with the Scope Manager.

The following is a simple stand-alone example program demonstrating currency scoping and make sure that you have a copy of `dfc.properties` in the classpath for the program:

```
/*
 * StdScopingExample.java
 *
 *
 * This example assumes that you have a standard type "test_payment_check"
 * already created.  One way to create this type is to use the following
 * DQL statement:
 CREATE TYPE "test_payment_check" (
 account integer,
 check_number integer,
 transaction_date date,
 amount float,
 bank_code integer,
 routing integer
 ) WITH SUPERTYPE "dm_document" PUBLISH
 *
 * This example also assumes that the repository name, username, and
 * password are passed to the program on the command line.
 */

import com.documentum.com.DfClientX;
import com.documentum.com.IDfClientX;
import com.documentum.fc.common.*;
import com.documentum.fc.client.*;


public class StdScopingExample {

    public void begin(String username, String password, String docbaseName) {

        IDfClientX clientx;
        IDfClient  client;
        IDfSession session = null;
        IDfSessionManager sMgr = null;
        IDfBatchManager bMgr = null;

        IDfLocalTransaction transaction;
        int rpcCountAfter, rpcCountBefore;

        try {
            //create a Client object
            clientx = new DfClientX();
            client = clientx.getLocalClient();

            //create a Session Manager object
            sMgr = client.newSessionManager();

            //create an IDfLoginInfo object
            IDfLoginInfo loginInfoObj = clientx.getLoginInfo();
            loginInfoObj.setUser(username);
            loginInfoObj.setPassword(password);
            //bind the Session Manager to the login info
            sMgr.setIdentity(docbaseName, loginInfoObj);
            session = sMgr.getSession(docbaseName);

            // get the Batch Manager for this session
            bMgr = session.getBatchManager();

/* This first batch, transaction, and call to createObject() performs a number
 * of RPC calls to setup the session, pull over the data dictionary, get
 * batch and transaction information, and other kinds of initial setup. This
 * setup overhead is a one-time hit for this session, and is done here so the
 * comparison of the scoping techniques doesn't include this overhead.
 */
            bMgr.openBatch();
            transaction = session.beginTransEx();
            for ( int i = 0 ; i < 1 ; i++) {
                createObject( session );
            }
            session.commitTransEx(transaction);
            bMgr.closeBatch();

/* The following section of code calls the createObject() method
 * from within a transaction, but does not use currency scoping.
 * The query in that method gets repeated unnecessarily.
 */
            rpcCountBefore =
                    session.getConnectionConfig().getInt("network_requests");
            transaction = session.beginTransEx();
            for (int i=0 ; i < 100; i++) {
                createObject( session );
            }
            session.commitTransEx(transaction);
            rpcCountAfter =
                    session.getConnectionConfig().getInt("network_requests");
            System.out.println(
                    "RPC calls using a transaction with no currency scoping: "
                    + (rpcCountAfter-rpcCountBefore));

/* This section of code is identical the the previous one, except for the
 * addition of transaction as the currency scope, so that the query in the
 * method is only checked once during that scope.
 */
            sMgr.getScopeManager().setDefaultScope(ScopeBoundary.TRANSACTION,
                    true);
            rpcCountBefore =
                    session.getConnectionConfig().getInt("network_requests");
            transaction = session.beginTransEx();
            for ( int i=0 ; i < 100 ;  i++) {
                createObject( session );
            }
            session.commitTransEx(transaction);
            rpcCountAfter =
                    session.getConnectionConfig().getInt("network_requests");
            System.out.println(
                 "RPC calls with currency scoping boundary set to TRANSACTION: "
                    + (rpcCountAfter-rpcCountBefore));
            sMgr.getScopeManager().clearDefaultScope(ScopeBoundary.TRANSACTION);


/* The following section of code calls the createObject() method
 * from within a batch, but does not use currency scoping.
 * The query in that method gets repeated unnecessarily.
 */
            rpcCountBefore =
                    session.getConnectionConfig().getInt("network_requests");
            //Open a batch to create the objects
            bMgr.openBatch();

            for ( int i=0 ; i < 100 ;  i++) {
                createObject( session );
            }

            // Close Batch and commit objects to repository
            bMgr.closeBatch();

            rpcCountAfter =
                    session.getConnectionConfig().getInt("network_requests");
            System.out.println(
                    "RPC calls using a batch with no currency scoping: "
                    + (rpcCountAfter-rpcCountBefore));

/* This section of code is identical the the previous one, except for the
 * addition of batch as the currency scope, so that the query in the
 * method is only checked once during that scope.
 */
            sMgr.getScopeManager().setDefaultScope(ScopeBoundary.BATCH, true);
            rpcCountBefore =
                    session.getConnectionConfig().getInt("network_requests");

            //Open a batch to create the objects
            bMgr.openBatch();
            for ( int i=0 ; i < 100 ;  i++) {
                createObject( session );
            }

            // Close Batch and commit objects to repository
            bMgr.closeBatch();

            rpcCountAfter =
                    session.getConnectionConfig().getInt("network_requests");
            System.out.println(
                 "RPC calls with currency scoping boundary set to BATCH: "
                    + (rpcCountAfter-rpcCountBefore));
            sMgr.getScopeManager().clearDefaultScope(ScopeBoundary.BATCH);

 /* The following section of code calls the createObject() method
 * without any batch or transaction, but does not use currency scoping.
 * The query in that method gets repeated unnecessarily.
 */

            rpcCountBefore =
                    session.getConnectionConfig().getInt("network_requests");

            for ( int i=0 ; i < 100 ;  i++) {
                createObject( session );
            }

            rpcCountAfter =
                    session.getConnectionConfig().getInt("network_requests");
            System.out.println("RPC calls with no currency scoping: "
                    + (rpcCountAfter-rpcCountBefore));

/* This section of code is identical the the previous one, except for the
 * addition of a custom currency scope, so that the query in the
 * method is only checked once during that scope.
 * Note that the call is to the Session Scope Manager in this section. A call
 * to beginScope() by the Session Scope Manager begins the currency scope in the
 * current session, but a call to beginScope() by the Scope Manager begins
 * currency scoping in future sessions created by the Session Manager.
 */

            rpcCountBefore =
                    session.getConnectionConfig().getInt("network_requests");
            session.getSessionScopeManager().beginScope(true);
            for ( int i=0 ; i < 100 ;  i++) {
                createObject( session );
            }
            session.getSessionScopeManager().endScope();
            rpcCountAfter =
                    session.getConnectionConfig().getInt("network_requests");
            System.out.println("RPC calls with a custom currency scoping: "
                    + (rpcCountAfter-rpcCountBefore));
        }
        catch(Throwable e) {
            if(e instanceof DfException) {
                System.out.println("DFC Exception:");
                String s = ((DfException)e).getStackTraceAsString();
                System.out.println(s);
            } else {
                System.out.println("Non-DFC Exception");
                e.printStackTrace();
            }
        }
        finally {
                if (session != null)
                sMgr.release(session);
        }
    }

/* This method is called repeatedly by this sample program. It queries the
 * repository and retrieves an object, then creates a single new object.
 */
    public void createObject( IDfSession session ) throws DfException {
        IDfFolder folder = null;
        IDfSysObject sysObj;
        IDfTime currDate = new DfTime();

 /* The section that follows queries for the object ID of the /System folder,
  * and retrieves the object. Since this is unlikely to change during a session,
  * there is a performance improvement if currency scoping is enabled.
  * For this example code, currency scoping is enabled by the calling code,
  * rather than here, demonstrating how the code behavior depends on the
  * currency scoping set by the calling code.
  */
        IDfQuery query = new DfQuery();
        query.setCurrencyCheckValue(IDfSession.CURRENCY_CHECK_ONCE_PER_SCOPE);
        query.setDQL("select r_object_id from dm_folder " +
                "where any r_folder_path = '/System'");
        IDfCollection collection =
                query.execute(session, DfQuery.DF_CACHE_QUERY);
        if (collection.next()) {
            IDfId folderId = collection.getId("r_object_id");
            folder = (IDfFolder) session.getObjectWithCaching( folderId, null,
                  null, IDfSession.CURRENCY_CHECK_ONCE_PER_SCOPE, false, false);
        }
        sysObj = (IDfSysObject)session.newObject("test_payment_check");
        sysObj.save();
    }
    public static void main(String[] args)
    throws java.lang.InterruptedException {
        String docbase = args[0];
        String user = args[1];
        String password = args[2];
        new StdScopingExample().begin(user, password, docbase);
    }
}
```

Chapter 4

Data partitioning

Data partitioning allows Documentum CM Server to use, and to be aware of, the partitioning features of the underlying database to store object metadata in partitions in that database. Partitioning the repository can allow optimized queries (if the partition is known) and storage of data on different physical devices based on storage and access requirements. It also allows a technique to improve loading of objects by first loading into a schema-equivalent offline table and then exchanging the offline table into the repository.

Data partitioning is only supported on Oracle and SQL Server installations currently. Data partitioning must be enabled on the underlying RDBMS for Documentum CM Server to use this feature.

Before Documentum CM Server supported partitioning, some installations used hash partitioning in the underlying RDBMS to improve performance. The hash was based on the object ID, so typical database access was spread across the partitions. A partitioning aware repository, in contrast, uses range partitioning for the tables of partitioned types. A new internal integer column i_partition is added to the tables of types that are partitionable. That i_partition column is used as the partition key when the database tables are created. The value of i_partition will be exposed as an attribute in the base type of objects, so Documentum CM Server can read and modify the partition where an object's metadata is stored, can write queries to target specific partitions, and can group related objects into the same partition.

Subtypes are partitionable if their supertypes are. If a supertype is partitioned, then all subtypes created from it are also partitionable. If a supertype is not partitioned, a subtype cannot be partitioned either.

Note: For release 6.5, Documentum CM Server used a different model of partitioning, did not use a scheme object, and used a different administrative method to handle partitioning. The *OpenText Documentum High-Volume Server 6.5 Development Guide* provides information about the earlier model. Also, see "Upgrading from release 6.5" on page 30, if you are upgrading a partitioned repository from release 6.5.

To use data partitioning, you must use the PARTITION_OPERATION administrative method to create (or modify) the partitioning scheme object for your application. This object contains the partition names, the associated i_partition ranges, and the tablespace (or filegroup) where the database stores the tables. The object also has attributes that lists the types and registered tables associated with the partition scheme. A type or registered table can only be in one partition scheme, and cannot be moved to another scheme. You cannot directly modify any of the properties of a dm_partition_scheme object, except for the object_name property. All other modifications are done through the PARTITION_OPERATION method.

This method generates an SQL script to run against the database that creates the partitions and specifies the ranges of i_partition values for objects that will reside in those partitions.

The following intrinsic types are partitioned in a newly-created repository if partitioning was selected during creation:

- dm_acl
- dm_sysobject
- dmr_containment
- dm_assembly
- dm_relation
- dm_relation_type
- dmi_otherfile
- dmi_replica_record
- dmr_content
- dmi_subcontent
- dmi_queue_item

Use the PARTITION_OPERATION administrative method to create the partitioning scheme and to generate the partitioning script needed to create the partitions.

To partition the repository:

1. Add and set `enable_database_partition=T` in `server.ini`.
2. Restart the Documentum CM Server.
3. Run the PARTITION_OPERATION without passing the `type_name` parameter so that the repository is marked for partition.

   For example:

   ```
   EXECUTE partition_operation WITH operation='db_partition',
   partition_scheme='de2_sch'
   ```

   object ID of the partitioning script is generated.
4. Copy the object ID in the EXECUTE command to get the location of the partitioning script on the Documentum CM Server.

   For example:

   ```
   EXECUTE get_file_url FOR '0855706080007c19' WITH "format"='text'
   ```
5. Stop the Documentum CM Server.
6. Run the script on the database and verify the output of the sql-script.
7. Start the repository.

The repository is now partitioned.

Note: When a repository is partitioned, by default all the indexes created on the type tables are local indexes. In some cases, for performance reasons, you may want to create global indexes on some of the type tables. However, if you create global indexes for some tables, those tables cannot be used in partition exchange. For information on how to create indexes, see the MAKE_INDEX administrative method in the *OpenText Documentum Content Management - Server DQL Reference Guide (EDCCS250400-DRD)*.

4.1 Using data partitioning

When a repository is partitioned, a range of i_partition values is assigned to each database partition. In effect, the i_partition values are used as a partition key, so an object is stored in a partition based on its i_partition value. For example, imagine that i_partition values from 0 to 9 are assigned to partition P0, values from 10 to 19 are assigned to P1, and values from 20 to 29 are assigned to P2. If an object is created with an i_partition value of 15, it will be stored in P1. If an application changes the i_partition value, the object will be moved to the appropriate partition. If an object is created without explicitly setting the i_partition value, it is stored in the default partition; the i_partition value is set to '0', and the object is stored in P0.

When a new repository is created with partitioning enabled, only a single partition exists. All partitionable objects are in this partition, until the partition scheme is modified. When a partition is added, a new range of i_partition values are assigned to the new partition. Objects are placed in a partition when its i_partition value is in the range assigned to that partition.

When you query a partitioned repository, use partition pruning to increase the query efficiency, where possible. For example, if you retrieve the i_partition value as well as the r_object_id value in your queries, subsequent queries against the objects returned will be more efficient if you use the i_partition value to search only the specified partition.

By default, ACL objects are restricted to be stored in a single partition.

There are some exceptions to the general rule that related objects are partition aligned. Relation objects will be moved if the relationship is defined between objects in the same partition, but if the objects are in different partitions, the relationship object will be stored in the with i_partition value of zero (the default partition). A content object cannot be shared among multiple SysObjects unless all the parents have the same partition designation. Renditions will inherit the partition designation of the parent SysObject and the storage for the content of such rendition objects will be derived from the partition definition. When an ACL object is shared by multiple sysobjects, the ACL object will be forced to stay in default partition. If an ACL object is created internally for a single sysobject, it will be stored in the same partition as the sysobject.

A lightweight object can be in a different partition than its shareable parent, but a lightweight object will be placed in the partition of its parent if the i_partition value is not explicitly set. If the i_partition value is changed for either the shared parent or the lightweight object, the change will not automatically propagate either way.

If a lightweight object is materialized, a private parent is created from the shared parent. This private parent is created in the same partition as the lightweight object.

It's the application's or the object/type customization model's responsibility to be able to align components of a VDM (virtual document) in a partition. Documentum CM Server will not enforce data alignment for the components of a VDM.

4.2 Using DQL with data partitioning

DQL has been enhanced to enable and support data partitioning.

4.2.1 Partition pruning in DQL statements

In a database system, partition pruning allows the database server to eliminate certain partitions when performing a search based on the search condition – typically a WHERE clause. For example, partitions used in partition pruning contain rows whose partition key columns fall into discrete values. When a query predicate is based on those partitioning keys, the database server can quickly ascertain whether rows in a particular partition can satisfy the query. This behavior is called partition pruning, or partition elimination. It can save considerable time and resources during query execution.

Since partition pruning can improve query performance, we recommend that you retrieve the i_partition value as well as the r_object_id value when you retrieve objects. Any later point queries against those objects will be much more efficient, as they will run only against the specified partition.

The SELECT, UPDATE OBJECT, and DELETE OBJECT statements support partition pruning.

The SELECT statement partition pruning clauses are shown in bold:

```
SELECT … FROM type [WITHIN PARTITION ( {partition_id [,…] } )], … ,
type [WITHIN PARTITION ( { partition_id [,…] } ] WHERE…
```

The UPDATE OBJECT statement partition pruning clauses are shown in bold:

```
UPDATE type [WITHIN PARTITION ( { partition_id[,…] } )] OBJECT … WHERE …
```

And the DELETE OBJECT statement partition pruning clauses are shown in bold:

```
DELETE type [WITHIN PARTITION ( { partition_id  [,…] } )] OBJECT … WHERE …
```

For example, we have the following example DQL query:

```
SELECT object_name, authors

FROM dm_sysobject WITHIN PARTITION ( 1, 10)

WHERE object_name LIKE ‘test%’
```

4.2.2 Creating data partitions

You can create data partitions using the DQL administrative method PARTITION_OPERATION. First, you use the method to create a partition scheme object that describes the partition. Then you use the method to generate an SQL script to run against the underlying database in your repository. When you run the script, the database tables are modified to reflect the partitioning scheme in the object you created. You can either run the DQL command from within Documentum Administrator, or using IDQL.

After you have generated the script, you can review it before you apply it to your database.

Since the partitioning object and the database partitioning do not match until the script is run, the is_validated property of the partition scheme object is set to FALSE until the script is successfully run.

4.3 Foundation Java API interface for data partitioning

Public methods have been added to the IDfPersistentObject and IDfSession classes to support data partitioning.

4.3.1 IDfPersistentObject

It is the application's responsibility to properly set the value for the i_partition attribute when a new object is created for a partitionable type. It is also the application's responsibility to ensure partition alignment when creating unrelated objects, if they should be aligned. For example, if two objects share content, they must be in the same partition.

To enable applications to set or retrieve the i_partition value, two new public APIs setPartition() and getPartition() are added to IDfPersistentObject Foundation Java API interface. The methods are:

- void *setPartition*(integer partition) throws DfException

  Sets the i_partition value on the object.

- int *getPartition*(integer partition) throws DfException

  Retrieves the i_partition value from the object.

If an application creates an object and does not specify its i_partition value, that value is automatically set to 0, so the object will be assigned to the first partition. If an application assigns an i_partition value greater than the largest currently defined partition range, the object will be assigned to the last partition. In normal use, the last partition should not contain any objects.

4.3.2 IDfSession

This method is similar to the public method getObject(): IDfPersistentObject *getObjectWithOptions*(IDfID objectID, IDfGetObjectOptions objectOptions) throws DfException

Use the objectOptions parameter to set the partition ID as a hint to fetch the object. With that hint, Documentum CM Server will fetch the object more efficiently.

4.4 Upgrading from release 6.5

If you have a partitioned repository from release 6.5 (partitioning was not available in earlier releases), the upgrade process will create dm_partition_scheme objects for each partitioned root type and registered table in your database. If the dmi_object_type table is partitioned, and it's partitioning scheme matches the scheme of a type, it will also be added to that types's scheme. Because of the change in the partitioning model from 6.5 to 6.6, you may have the table dmi_object_type in more than one partition scheme. This is the only case where a type or table can be in more than one partition scheme.

Starting with the 6.6 release, the idea of a last partition, one that contains objects with i_partition values greater than the largest defined value, is no longer supported.

The GENERATE_PARTITION_SCHEME_SQL method is deprecated. In the 6.6 release, this method was refactored to use the PARTITION_OPERATION method for backward compatibility.

4.5 Partition switch in and switch out

You can exchange offline tables in the database with the online tables used by Documentum CM Server (partition switch in), and exchange the online tables for offline tables (switch out). Databases that support this partition exchange have native methods that allow you to quickly load offline tables. When the offline tables are ready, they can be swapped into the repository, ingesting large quantities of data with minimal disruption to normal operations. Or, existing data can be quickly swapped out of the repository.

Typically, an application of partition exchange is to load many objects of a few types into the repository. The objects will typically have content associated with them, and each of the objects and content will be very similar. A use case for this is to move large amounts of existing data from a legacy system into Documentum CM Server.

For a specific example of using this feature, we assume that a legacy application tracks payments made with checks. The application data will be loaded into objects that each have an image of the check as content, and the check transaction information as metadata. The check images in an application will typically all be of the same file type, and the transaction records reside in a small number of types.

There is a restriction on what kinds of objects can be ingested into the server using partition switch in, or removed by switch out. The restrictions are as follows:

- the shared parent SysObjects already exist in the repository
- lightweight SysObjects only
- content from external stores only
- no private ACLs (the object ACL is associated with the shared parent)

In principle, other kinds of objects could be ingested, but this is not supported. For assistance with ingestion outside of these restrictions, contact OpenText Global Technical Services.

Since the objects are swapped into the repository in already populated tables, there are no audit records created by the repository for the ingestion of the objects. This method of ingestion will not execute any TBOs on the types being ingested.

As an example of using partition exchange to load a large number of objects with contents, consider the preceding application. For our example, each check is imaged, and the image is stored in an external content store. The payment metadata is stored in two types, one with the bank information, and one with the specific check information. In this example, we'll create text files to contain the object metadata, load the date into offline tables in the database, and then switch in the offline tables into an active repository partition to load the objects into the repository. We'll store the bank information in a shareable parent object and the check information in a lightweight child object. Lightweight objects and shareable parents are discussed in “Lightweight SysObjects“ on page 49.

This example is typical of many installations that have large amounts of legacy data created by another application. This example shows how that data can be moved into Documentum CM Server.

Note: In order to exchange partitions, the table indexes for types involved in the partition swap must be local indexes. By default, a newly partitioned repository creates local indexes. If you have created global indexes on those tables, the indexes must be rebuilt as local indexes before you can exchange partitions.

In this example, we will use the following conditions:

- Oracle database

  This technique can also be used with a SQL Server installation.
- An external file store that already holds the content

  Typically created by another system and stored at the external store location, the external store represents legacy data already in another system that you want to ingest into Documentum CM Server.
- No subcontent

  Since we are using an external filestore, the use of subcontent is not supported.

- No ACLs

  Again, this restriction simplifies the example, but typically the ACLs will apply to the parent objects.

- One shareable parent object (test_payment_bank) for all of the lightweight children (test_payment_check)

  We will create the parent object using Foundation Java API, but create all the child objects using text files loaded into the database and then exchanged for the partition.

The restrictions for this example reduces the number of type hierarchies that we will partition, and the offline tables that we will switch into the repository. We will partition the dm_sysobject type hierarchy, for the lightweight system objects that we are ingesting. We will also partition the dmr_content type. There is an additional table that we will partition, dmi_object_type.

The dmi_object_type table keeps track of the type of many of the objects in the repository, especially for types that share the same tag use. This table helps Documentum CM Server to correctly retrieve object metadata. The dmi_object_type table does not represent a type, and is not documented in *OpenText Documentum Content Management - Server System Object Reference Guide (EDCCS250400-ORD)*, since you must not manipulate object types by using this table. However, when you ingest objects using a partition exchange, you must inform the repository of the type of each object that is ingested. This is the only way that you should modify this table. Any other uses are not supported.

The offline tables that we will switch into the repository are:

- dmi_object_type
- dmr_content_s
- dmr_content_r
- test_payment_check_s
- test_payment_check_r

The dmi_object_type table contains the object type and r_object_id of each object ingested. The two dmr_content tables contain the metadata for the content objects. The two test_payment_check tables contain the metadata for the ingested check information. If we were trying in ingest standard SysObjects, we would have to supply the metadata for all the supertypes on the test_payment_check object. Since this is a lightweight type, that information is already contained in the parent object, so we do not have to generate that outside of the repository.

For this example, follow these steps:

1. “Create the external filestore” on page 33
2. “Create the object types in the repository” on page 33
3. “Create the empty partition to exchange with the offline tables” on page 33

4. “Create the empty offline tables” on page 35
5. “Create the data to load” on page 37
6. “Load the data into the offline tables” on page 42
7. “Create the indexes” on page 44
8. “Switch the data into the partition” on page 46
9. “Verify that the data has been loaded” on page 47

4.5.1 Create the external filestore

Create an external filestore named filestore_ex on the server machine. The example uses the storage path c:\ExternalStore. The *OpenText Documentum Content Management - Administrator User Guide (EDCAC250400-UGD)* contains more information about creating an external store.

4.5.2 Create the object types in the repository

Create the shareable parent type with the following DQL command:

```
CREATE SHAREABLE TYPE test_payment_bank (
bank_code integer,
routing integer,
branch integer repeating
) WITH SUPERTYPE dm_sysobject PUBLISH
```

Then create the lightweight type with the following DQL command:

```
CREATE LIGHTWEIGHT TYPE test_payment_check (
account integer,
check_Number integer,
transaction_date date,
amount float,
deductions float repeating
) SHARES test_payment_bank PUBLISH
```

4.5.3 Create the empty partition to exchange with the offline tables

The steps to take depend on whether or not you enabled partitioning when you created the repository.

4.5.3.1 Originally non-partition-enabled repository

1. Run the following command in DQL:

   ```
   EXECUTE partition_operation WITH "operation"='create_scheme',
   "partition_scheme"='<name of the scheme>',
   "partition_name"='<name of the first partition, for example P1>',
   "range"=<specify the range, for example 100>,
   "tablespace"='dm_<repository name>_DOCBASE',
   "partition_name"='<name of the second partition, for example P2>',
   "range"=<specify the range, for example 200>,
   "tablespace"='dm_<repository name>_DOCBASE',
   "partition_name"='<name of the nth partition>',"range"=<specify the range>,
   "tablespace"='DM_<repository name>_DOCBASE'
   ```

   The result of this command is the object ID of the dm_partition_scheme object that you have just created.

2. Now that the partition scheme is created, generate the SQL script to run against the database. In this example, you run the db_partition operation without specifying any tables or types. Run the following command in DQL:

   ```
   EXECUTE partition_operation WITH "operation"='db_partition',
   "partition_scheme"='<name of the scheme>'
   ```

3. Run the following command in DQL:

   ```
   EXECUTE get_file_url FOR
   <object id generated from step 2>
   WITH “format”='text'
   ```

4. Run the following command in IAPI:

   ```
   getpath,c,<session id>,
   <object id generated from step 2>
   ```

   This generates the path.

5. Stop the Documentum CM Server. Provide the access permissions to the *system* database using the following commands:

   ```
   alter user <repository_name> quota unlimited on system;
   grant create session to <repository_name>;
   grant connect to <repository_name>;
   grant resource to <repository_name>;
   grant create any table to <repository_name>;
   grant Create Any Table to system;

   grant dba to <repository_name>;
   grant dba to system;
   ```

6. In the command prompt, run the following command:

   ```
   C:\Windows\System32>osql -Usa -Pmanager -SConfig13903VMO -n -i
   <path generated from step 4>

   sqlplus -S sys/password@orcl as sysdba @
   /export/home/d71rc2/dctm/data/<repository_name</content_storage_01
   /00030841/80/00/2d/43.txt
   ```

7. Start the Documentum CM Server.

4.5.3.2 Originally partition-enabled repository

If you created the repository with partitioning enabled, then you will add the additional partitions needed for this example slightly differently. Since dm_sysobject and dmr_content are already partitioned and have their own partition schemes, we need to modify both of those schemes for this example. First, you will add partitions to dm_sysobject_sch and dmr_content_sch. Second, you will add the non-partitioned dmi_object_type table to the dm_sysobject_sch. The first steps are done online by Documentum CM Server, the second step requires you to manually run the script against the database.

First, use the following DQL command to add partitions to the dm_sysobject_sch:

```
EXECUTE partition_operation WITH "operation"='add_partition',
"partition_scheme"='dm_sysobject_sch',
"partition_name"='P1',"range"=100,"tablespace"='dm_techpubsglobal_docbase',
"partition_name"='P2',"range"=200,"tablespace"='dm_techpubsglobal_docbase'
```

This command runs the partitioning script on the database for you, so you do not run it manually. Now, use the following command to add the same partitions to the scheme for dmr_content:

```
EXECUTE partition_operation WITH "operation"='add_partition',
"partition_scheme"='dmr_content_sch',
"partition_name"='P1',"range"=100,"tablespace"='dm_techpubsglobal_docbase',
"partition_name"='P2',"range"=200,"tablespace"='dm_techpubsglobal_docbase'
```

Now, partition the dmi_object_type table by using the following command:

```
EXECUTE partition_operation with "operation"='db_partition',
"partition_scheme"='dm_sysobject_sch',
"table_name"='dmi_object_type',"owner_name"='TechPubsGlobal'
```

Retrieve the partitioning script, stop the server, run the script on the database, and restart the server. You should now have a partitioned repository. The types and table needed for this example are now represented by two partition scheme objects, dm_sysobject_sch and dmr_content_sch.

4.5.4 Create the empty offline tables

There are five tables that need to be populated for this example. They are:

- DMI_OBJECT_TYPE
- DMR_CONTENT_R
- DMR_CONTENT_S
- TEST_PAYMENT_CHECK_R
- TEST_PAYMENT_CHECK_S

The two DMR_CONTENT tables keep track of what objects have which pieces of content.

The two TEST_PAYMENT_CHECK tables store the check data. If we correctly fill up these tables and load them in, we can then exchange them for an active partition in the repository.

The offline tables in the database and the tables in the active partition must be named differently. The offline table that corresponds to an active table appends a suffix to the table name. If you specify the suffix as 'x', the offline table that corresponds to the active table DMI_OBJECT_TYPE is named DMI_OBJECT_TYPEX, and the offline table for TEST_PAYMENT_CHECK_R is named TEST_PAYMENT_CHECK_RX.

The schema of the offline tables must exactly match the schema of the active tables in the repository. It is possible to carefully create those tables, but the switch_out operation provides a simple way to create those tables from an empty partition. When you switch_out the empty partition, you create all the empty tables needed, with all the correct columns. Using switch_out creates more tables than we need for this example, but it is much easier and less error-prone to let Documentum CM Server do this.

The process is slightly different for a repository that was originally partition-enabled and for one that you enabled after creation. The originally partition-enabled repository has two partition schemes that you work with, dm_sysobject_sch and dmr_content_sch, and the originally non-partitioned repository uses one scheme, example_sch.

4.5.4.1 Originally non-partition-enabled

To create the offline tables in an originally non-partition-enabled repository for our example, use the following command with example_sch:

```
EXECUTE partition_operation WITH "operation"='switch_out',
"partition_scheme"='example_sch',
"temp_table_suffix"='x',"partition_name"='P2',"execute_now"=TRUE
```

4.5.4.2 Originally partition-enabled

To create the offline tables in an originally partition-enabled repository for our example, use the following command with dm_sysobject_sch:

```
EXECUTE partition_operation WITH "operation"='switch_out',
"partition_scheme"='dm_sysobject_sch',
"temp_table_suffix"='x',"partition_name"='P2',"execute_now"=TRUE
```

and use the following command with dmr_content_sch:

```
EXECUTE partition_operation WITH "operation"='switch_out',
"partition_scheme"='dmr_content_sch',
"temp_table_suffix"='x',"partition_name"='P2',"execute_now"=TRUE
```

4.5.5 Create the data to load

Each of the empty offline tables will have to be loaded with data by means of the sqlloader utility. Some kind of application will normally need to be written to provide the data to populate each field of the tables with the correct data. The application must also keep track of the relationships between the data in the tables.

For this example, a simplified Java program is provided that will create the necessary tables. It creates the shared parent object, and then creates text files corresponding to the five tables that will be loaded into the database.

The sample is in “Sample program for generating metadata for partition exchange“ on page 93.

To determine what kind of values to use to fill the offline tables, you can create test objects in the repository and examine the property values that the repository creates, and examine the property descriptions in *OpenText Documentum Content Management - Server System Object Reference Guide (EDCCS250400-ORD)*. However, there are a few values that we recommend that you set somewhat differently from how they are set by the repository. The following tables list and describe the values loaded into the tables for our example.

The DMI_OBJECT_TABLE keeps track of the type of each object.

Table 4-1: DMI_OBJECT_TYPE table

Property	Value	Comments
R_OBJECT_ID	ID of child object	Use NEXT_ID_LIST to reserve IDs.
I_TYPE	child object type	Retrieve the integer representing the child type in your repository, and use that value.
I_PARTITION	partition value	Supply the partition number for the object.

The two DMR_CONTENT tables, DMR_CONTENT_R and DMR_CONTENT_S, keep track of the location of each content file, it's format, and links the content to an object.

Table 4-2: DMR_CONTENT_R table

Property	Value	Comments
R_OBJECT_ID	ID of content object	Use NEXT_ID_LIST to reserve IDs.
I_POSITION	-1	

Property	Value	Comments
I_PARTITION	partition value	Supply the partition number for the object.
PARENT_ID	ID of child object	In this usage, the lightweight child object is the parent of the content object.
PAGE	0	
FULLTEXT_INDEX	empty	
INDEX_PENDING	empty	
UPDATE_COUNT	empty	
I_INDEX_FORMAT	empty	
I_FORMAT	R_OBJECT_ID of the TIFF format	Retrieve the ID of this file format from your repository, and use that value. See the getFormatID method in the example for details.
I_RENDITION	0	
I_PX	0	
I_PY	0	
I_PZ	0	
I_ENCODING	empty	
I_FULL_FORMAT	tiff	Text name of the format.
INDEX_PARENTS	empty	
INDEX_SET_TIMES	empty	
INDEX_PAGES	empty	
INDEX_SUBTYPES	empty	
INDEX_OPERATIONS	empty	
INDEX_FORMATS	empty	
CONTENT_ATTR_NAME	empty	
CONTENT_ATTR_VALUE	empty	
PAGE_MODIFIER	empty	
CONTENT_ATTR_NUM_VALUE	empty	
CONTENT_ATTR_DATE_VALUE	empty	
CONTENT_ATTR_DATA_TYPE	empty	

Table 4-3: DMR_CONTENT_S table

Property	Value	Comments
R_OBJECT_ID	ID of content object	Same as DMR_CONTENT_R value.
RENDITION	0	
PARENT_COUNT	1	In this example, we pretend that there is only one lightweight SysObject using this content file, but you could easily modify the example to generate a new content file for each object.
STORAGE_ID	R_OBJECT_ID of the filestore	See the getFileStoreID method in the example for details.
DATA_TICKET	0	For external filestores, the data ticket value is 0.
OTHER_TICKET	0	
CONTENT_SIZE	file size	The size of the TIFF image file. The repository sets this value differently for external filestores, but we recommend that you explicitly set this property value to the file size of the content file.
FULL_FORMAT	tiff	Text name of the format of the content. The example uses TIFF, but you would set this to your content type.
FORMAT	R_OBJECT_ID of the TIFF format	Retrieve the ID of this file format from your repository, and use that value. See the getFormatID method in the example for details.
RESOLUTION	0	
X_RANGE	0	
Y_RANGE	0	
Z_RANGE	0	
ENCODING	empty	In the example, we use a double-quoted blank space for this value.
LOSS	0	

Property	Value	Comments
TRANSFORM_PATH	empty	In the example, we use a double-quoted blank space for this value.
SET_CLIENT	hostname	
SET_FILE	content location	
SET_TIME	time	
IS_OFFLINE	0	
I_CONTENTS	empty	In the example, we use a double-quoted blank space for this value.
I_OTHER_CONTENTS	empty	In the example, we use a double-quoted blank space for this value.
IS_ARCHIVED	0	
INDEX_FORMAT	0000000000000000	
INDEX_PARENT	0000000000000000	
FULL_CONTENT_SIZE	file size	Set to same value as CONTENT_SIZE.
R_CONTENT_HASH	empty	In the example, we use a double-quoted blank space for this value.
I_PARKED_STATE	0	
OTHER_FILE_SIZE	0	
I_PARTITION	partition value	Supply the partition number for the object.
I_IS_REPLICA	0	
I_VSTAMP	0	

The TEST_PAYMENT_CHECK tables contain the metadata for the lightweight SysObjects to be ingested. The properties defined for the type are populated here, as well as some table values intended for the system. The link from each lightweight SysObject to its sharing parent is set with the I_SHARING_PARENT property value. Repeating properties require an index position value, stored in the I_POSITION property. The index is a negative integer.

Table 4-4: TEST_PAYMENT_CHECK_R table

Property	Value	Comments
R_OBJECT_ID	ID of child object	Same as DMI_OBJECT_TYPE.

Property	Value	Comments
I_POSITION	position of repeating attribute	Use negative integers, beginning with -1 for the first item, -2 for the second item, and so on, as the position for repeating attributes.
I_PARTITION	partition value	Supply the partition number for the object.
DEDUCTIONS		The sample program supplies two deduction values, 10.00 and 20.00.

Table 4-5: TEST_PAYMENT_CHECK_S table

Property	Value	Comments
R_OBJECT_ID	ID of child object	Same as DMI_OBJECT_TYPE.
ACCOUNT		
CHECK_NUMBER		
TRANSACTION_DATE		
AMOUNT		
I_PARTITION	partition value	Supply the partition number for the object.
OBJECT_NAME		
I_SHARING_PARENT	parent object ID	This is the ID of the shared parent object.
R_PAGE_CNT	1	
I_VSTAMP	0	

4.5.5.1 NEXT_ID_LIST

This apply method reserves a list of object IDs from the repository. Use this method to reserve the object IDs for the objects you plan to ingest using partition exchange. Each ingested object requires a unique object ID, and this method allows you to reserve a set of IDs. After the reservation, the IDs will not be used by the repository, so you will not ingest an object ID in use by another object.

Since the IDs will not otherwise be used by the repository, you will want to carefully consider how many to reserve for each call. Reserve enough to reduce the number of ID requests, but not so many as to use up IDs if the process fails, or that there are leftover IDs when the objects are created. The equivalent API command is:

```
apply,c,NULL,NEXT_ID_LIST,TAG,I,08,HOW_MANY,I,100
```

The following is a short Java code snippet demonstrating its use. Supply the session, object tag, and number of IDs requested to this method, and it will return a list of object IDs with the proper ID tag:

```
private static IDfCollection myGetNewObjectIds(IDfSession session, String tag,
        String number) throws Throwable
{
    IDfList args = new DfList();
    IDfList types = new DfList();
    IDfList values = new DfList();

    args.appendString("TAG");
    types.appendString("I");
    values.appendString(tag);

    args.appendString("HOW_MANY");
    types.appendString("I");
    values.appendString(number);

    return session.apply("NULL", "NEXT_ID_LIST", args, types, values);
}
```

The sample program in "Sample program for generating metadata for partition exchange" on page 93 demonstrates using this method to retrieve object IDs for the new objects.

4.5.6 Load the data into the offline tables

Now that the data files have been created, it is time to load the data into the offline database tables. We will use the sqlloader to accomplish this.

An sqlloader command line looks as follows:

```
sqlldr userid=username/password control=dmiObjectType.ctl log=dmiObjectType.log
 DIRECT=TRUE
```

where the userid parameter is the username/password for the owner of Documentum CM Server, log is the log file for the command, and setting DIRECT=TRUE bypasses Oracle's transaction control, greatly increasing the ingestion rate. The control file contains the detailed commands to load the data. There are five sqlloader command lines and five control files. You will need to edit the loader command files and replace Exchange/Exchange with the username/ password of Documentum CM Server.

Note: If you experience difficulties with the following information, please consult the Oracle documentation or Oracle support for assistance.

These are the five loading command files:

- *load_dmi_object.cmd* contains the following sqlloader commands:

  ```
  now
  sqlldr userid=username/password control=dmiObjectType.ctl log=dmiObjectType.log
   DIRECT=TRUE
  now
  ```

- *load_dmr_content_r.cmd* contains the following sqlloader commands:

```
now
sqlldr userid=username/password control=dmrContentR.ctl log=dmrContentR.log
 DIRECT=TRUE
now
```

- *load_dmr_content_s.cmd* contains the following sqlloader commands:

```
now
sqlldr userid=username/password control=dmrContentS.ctl log=dmrContentS.log
 DIRECT=TRUE
now
```

- *load_test_payment_check_r.cmd* contains the following sqlloader commands:

```
now
sqlldr userid=username/password control=testPaymentCheckR.ctl
log=testPaymentCheckR.log
 DIRECT=TRUE
now
```

- *load_test_payment_check_s.cmd* contains the following sqlloader commands:

```
now
sqlldr userid=username/password control=testPaymentCheckS.ctl
log=testPaymentCheckS.log
 DIRECT=TRUE
now
```

These are the five control files:

- *dmiObjectType.ctl* contains the following commands:

```
load data
 infile 'dmiObjectType.txt'
 into table DMI_OBJECT_TYPEx
 fields terminated by "," optionally enclosed by '"'
 (R_OBJECT_ID, I_TYPE, I_PARTITION)
```

- *dmrContentR.ctl* contains the following commands:

```
load data
 infile 'dmrContentR.txt'
 into table DMR_CONTENT_Rx
 fields terminated by "," optionally enclosed by '"'
 (R_OBJECT_ID, I_POSITION, I_PARTITION, PARENT_ID, PAGE, FULLTEXT_INDEX,
  INDEX_PENDING, UPDATE_COUNT, I_INDEX_FORMAT, I_FORMAT, I_RENDITION,
  I_PX, I_PY, I_PZ, I_ENCODING, I_FULL_FORMAT, INDEX_PARENTS, INDEX_SET_TIMES,
  INDEX_PAGES, INDEX_SUBTYPES, INDEX_OPERATIONS, INDEX_FORMATS,
  CONTENT_ATTR_NAME, CONTENT_ATTR_VALUE, PAGE_MODIFIER, CONTENT_ATTR_NUM_VALUE,
  CONTENT_ATTR_DATE_VALUE, CONTENT_ATTR_DATA_TYPE)
```

- *dmrContentS.ctl* contains the following commands:

```
load data
 infile 'dmrContentS.txt'
 into table DMR_CONTENT_Sx
 fields terminated by "," optionally enclosed by '"'
 (R_OBJECT_ID, RENDITION, PARENT_COUNT, STORAGE_ID, DATA_TICKET, OTHER_TICKET,
  CONTENT_SIZE, FULL_FORMAT, FORMAT, RESOLUTION, X_RANGE, Y_RANGE, Z_RANGE,
  ENCODING, LOSS, TRANSFORM_PATH, SET_CLIENT, SET_FILE, SET_TIME, IS_OFFLINE,
  I_CONTENTS, I_OTHER_CONTENTS, IS_ARCHIVED, INDEX_FORMAT, INDEX_PARENT,
  FULL_CONTENT_SIZE, R_CONTENT_HASH, I_PARKED_STATE, OTHER_FILE_SIZE, I_PARTITION,
  I_IS_REPLICA, I_VSTAMP)
```

- *testPaymentCheckR.ctl* contains the following commands:

```
load data
 infile 'testPaymentCheckR.txt'
 into table TEST_PAYMENT_CHECK_Rx
 fields terminated by "," optionally enclosed by '"'
 (R_OBJECT_ID, I_POSITION, I_PARTITION, DEDUCTIONS)
```

- *testPaymentCheckS.ctl* contains the following commands:

```
load data
 infile 'testPaymentCheckS.txt'
 into table TEST_PAYMENT_CHECK_Sx
 fields terminated by "," optionally enclosed by '"'
 (R_OBJECT_ID, ACCOUNT, CHECK_NUMBER, TRANSACTION_DATE, AMOUNT, I_PARTITION,
  OBJECT_NAME, I_SHARING_PARENT, R_PAGE_CNT, I_VSTAMP)
```

This last file can be used to launch all the other files:

LoadUp.cmd contains the following commands:

```
start load_dmi_object.cmd
start load_dmr_content_r.cmd
start load_dmr_content_s.cmd
start load_test_payment_check_r.cmd
start load_test_payment_check_s.cmd
```

When you have finished loading the data, it is a good time to verify that the load has gone correctly. Review the log files and be sure that the number of rows loaded is what you expected.

4.5.7 Create the indexes

If you created the offline tables as suggested, using the switch_out operation, the indexes should already be in place, so there is no need to create them.

If you created the offline tables manually, you must also create the indexes. The offline tables must have indexes when the partition exchange occurs.

Here are examples of creating indexes for the example. The index table names are not critical, but must be distinct from any other table names. Each of the SQL command files that creates the indexes uses the PARALLEL option to make use of parallel processing in Oracle.

Note: If you experience difficulties with the following information, please consult the Oracle documentation or Oracle support for assistance.

Here are the three files with commands to create the indexes.

DMI_OBJECT_TYPE_INDEX.sql contains the following commands:

```
select to_char(sysdate, 'Dy DD-Mon-YYYY HH24:MI:SS') as "Current Time" from dual;
create UNIQUE index DMI_OBJECT_TYPE_UNIQUEx
on DMI_OBJECT_TYPEx(
R_OBJECT_ID
,I_TYPE
,I_PARTITION
)
TABLESPACE DM_TECHPUBS_INDEX
PARALLEL 7
NOLOGGING
/
select to_char(sysdate, 'Dy DD-Mon-YYYY HH24:MI:SS') as "Current Time" from dual;
```

DMR_CONTENT_INDEX.sql contains the following commands:

```
select to_char(sysdate, 'Dy DD-Mon-YYYY HH24:MI:SS') as "Current Time" from dual;
create  index DMR_CONTENT_Rx_1
on DMR_CONTENT_Rx(
PARENT_ID
,PAGE
)
TABLESPACE DM_TECHPUBS_INDEX
PARALLEL 7
NOLOGGING
/


create UNIQUE index DMR_CONTENT_Rx_2
on DMR_CONTENT_Rx(
R_OBJECT_ID
,I_POSITION
,I_PARTITION
)
TABLESPACE DM_TECHPUBS_INDEX
PARALLEL 7
NOLOGGING
/


create  index DMR_CONTENT_Sx_1
on DMR_CONTENT_Sx(
DATA_TICKET
,STORAGE_ID
,FORMAT
)
TABLESPACE DM_TECHPUBS_INDEX
PARALLEL 7
NOLOGGING
/


create  index DMR_CONTENT_Sx_2
on DMR_CONTENT_Sx(
I_PARKED_STATE
,R_OBJECT_ID
)
TABLESPACE DM_TECHPUBS_INDEX
PARALLEL 7
NOLOGGING
/


create UNIQUE index DMR_CONTENT_Sx_3
on DMR_CONTENT_Sx(
R_OBJECT_ID
,I_PARTITION
)
TABLESPACE DM_TECHPUBS_INDEX
PARALLEL 7
NOLOGGING
/
select to_char(sysdate, 'Dy DD-Mon-YYYY HH24:MI:SS') as "Current Time" from dual;
```

TEST_PAYMENT_INDEX.sql contains the following commands:

```
select to_char(sysdate, 'Dy DD-Mon-YYYY HH24:MI:SS') as "Current Time" from dual;
create UNIQUE index TEST_PAYMENT_CHECK_Rx_1
on TEST_PAYMENT_CHECK_Rx(
R_OBJECT_ID
,I_POSITION
,I_PARTITION
)
TABLESPACE DM_TECHPUBS_INDEX
```

```
PARALLEL 7
NOLOGGING
/


create UNIQUE index TEST_PAYMENT_CHECK_Sx_1
on TEST_PAYMENT_CHECK_Sx(
R_OBJECT_ID
,I_SHARING_PARENT
,I_PARTITION
)
TABLESPACE DM_EXCHANGE_INDEX
PARALLEL 7
NOLOGGING
/
select to_char(sysdate, 'Dy DD-Mon-YYYY HH24:MI:SS') as "Current Time" from dual;
```

Finally, here is a file that will run all three of the index creation files. Again, you must substitute the username/password of your repository for Exchange/Exchange.

Create_Index.cmd contains the following commands:

```
start sqlplus Exchange/Exchange@ORCL @DMI_OBJECT_TYPE_INDEX.sql
start sqlplus Exchange/Exchange@ORCL @DMR_CONTENT_INDEX.sql
start sqlplus Exchange/Exchange@ORCL @TEST_PAYMENT_INDEX.sql
```

4.5.8 Switch the data into the partition

Now that the offline tables have been loaded with data, and the indexes have been generated, we are ready to create the partition exchange script. In this example, we are switching in two dmr_content database tables, dmr_content_r and dmr_content_s, the two custom lightweight object tables, test_payment_check_s and test_payment_check_r, and the dmi_object_type table.

The switch_in operation can be performed online or offline, but we will do this online for our example. Be sure that nothing will access the switch partition during the switch. While the individual tables are switched into the partition, the information in the partition is not consistent. Again, the commands used for an originally partition-enabled repository are different from those for an originally non-partition-enabled repository.

4.5.8.1 Originally non-partition-enabled

Use the following command to switch the data from the temporary tables into the partition:

```
EXECUTE partition_operation WITH "operation"='switch_in',
"partition_scheme"='example_sch',
"temp_table_suffix"='x',"partition_name"='P2',"execute_now"=TRUE
```

4.5.8.2 Originally partition-enabled

Use the following two commands to switch the from the temporary tables into the partition:

```
EXECUTE partition_operation WITH "operation"='switch_in',
"partition_scheme"='dm_sysobject_sch',
"temp_table_suffix"='x',"partition_name"='P2',"execute_now"=TRUE

EXECUTE partition_operation WITH "operation"='switch_in',
"partition_scheme"='dmr_content_sch',
"temp_table_suffix"='x',"partition_name"='P2',"execute_now"=TRUE
```

4.5.9 Verify that the data has been loaded

Now that the data has been loaded into the partition, examine the results. The directly loaded data is now accessible in your repository.

Chapter 5

Lightweight SysObjects

5.1 Lightweight type

A lightweight type is a type whose implementation is optimized to reduce the storage space needed in the database for instances of the type. All lightweight SysObjects types are subtypes of a shareable type. When a lightweight SysObject is created, it references a shareable supertype object. As additional lightweight SysObjects are created, they can reference the same shareable object. That shareable object is called the lightweight SysObject's parent, and the lightweight SysObject is the child. Each lightweight SysObject shares the information in its shareable parent object. In that way, instead of having multiple nearly identical rows in the SysObject tables to support all the instances of the lightweight type, a single parent object exists for multiple lightweight SysObjects.

You may see a lightweight SysObject referred to as a lightweight object, or sometimes abbreviated as LWSO. All of these terms are equivalent.

Lightweight objects are useful if you have a large number of attribute values that are identical for a group of objects. This redundant information can be shared among the LWSOs from a single copy of the shared parent object. For example, Enterprise A-Plus Financial Services receives many payment checks each day. They record the images of the checks and store the payment information in SysObjects. They will retain this information for several years and then get rid of it. For their purposes, all objects created on the same day can use a single ACL, retention information, creation date, version, and other attributes. That information is held by the shared parent object. The LWSO has information about the specific transaction.

Using lightweight SysObjects can provide the following benefits:

- Lightweight types take up less space in the underlying database tables than a standard subtype.
- Importing lightweight objects into a repository is faster than importing standard SysObjects.

5.2 Shareable type

A shareable type is a type whose instances can share its property values with instances of lightweight types. It is possible for multiple lightweight objects to share the property values of one shareable object. The shareable object that is sharing its properties with the lightweight object is called the parent object, and the lightweight object is called its child.

5.3 Working with lightweight objects

In many contexts, lightweight objects behave similar to standard sysobjects. Any operation that you perform on a lightweight object that affects only the attributes of the lightweight type will behave similar to a standard object. You can read and modify any of the values for attributes of the lightweight type, just as you would for a normal SysObject. You can read (get) any of the attribute values that you normally can that are part of the parent type or its supertypes. However, if you attempt to set the values of attributes of the parent type (or the inherited attributes from its supertypes), a lightweight object behaves differently from a normal object.

Since the attribute values of the shared parent are used by all the other lightweight object children, you cannot change the parent's attributes without affecting all of the children. To prevent changing the attributes of the shared parent, but to allow changing those attributes for a specific child, a copy of the shared parent object is created for the private use of the child lightweight object. When a lightweight object has a private parent object, we call this a materialized lightweight object. Now the child lightweight object and its private parent behave similar to a normal object.

You can explicitly materialize a lightweight object by using the Foundation Java API materialize method, or the lightweight object can automatically materialize when you attempt to modify shared parent attributes. When you create the lightweight type, you specify if automatic materialization is allowed. If you don't allow automatic materialization, any operation that would cause materialization generates an error.

A LWSO that is materialized can be unmaterialized. After performing the operations that require materialization, you can either reparent the LWSO to the original shared parent, or to another shared parent.

A listing of methods and their effects is shown in the section, “Applying SysObject APIs to lightweight objects” on page 59. Some common things to be aware of are:

- folders—the link/unlink API will cause materialization, since it affects the shared parent. Place the shared parent in a folder before attaching any child LWSOs. Therefore, all the child LWSOs must be in the same folder as their shared parent.
- content type—setting content type will also cause materialization. Set the content type on the parent before attaching the child LWSOs. All the content for the children will be of the same type.
- lifecycle—participating in a lifecycle requires the LWSO to be materialized.

5.4 Storing lightweight subtype instances

A lightweight type is a subtype of a shareable type, so the tables representing the lightweight type store only the properties defined for the lightweight type. The values for inherited properties are stored in rows in the tables of the shareable type (the supertype of the lightweight type). In standard objects, the r_object_id attribute is used to join the rows from the subtype to the matching rows in the supertype. However, since many lightweight objects can share the attributes from their shareable parent object, the r_object_id values will differ from the parent object r_object_id value. For lightweight objects, the i_sharing_parent attribute is used to join the rows. Therefore, many lightweight objects, each with its own r_object_id, can share the attribute values of a single shareable object.

5.4.1 Materialization and lightweight SysObjects

When a lightweight object shares a parent object with other lightweight objects, we say that the lightweight object is unmaterialized. All the unmaterialized lightweight objects share the attributes of the shared parent, so, in effect, the lightweight objects all have identical values for the attributes in the shared parent. This situation can change if some operation needs to change a parent attribute for one of (or a subset of) the lightweight objects. Since the parent is shared, the change in an attribute would affect all the children. If the change should only affect one child, that child object needs to have its own copy of the parent. When a lightweight object has its own private copy of a parent, we say that the object is materialized. Documentum CM Server creates rows in the tables of the shared type for the object, copying the values of the shared properties into those rows. The lightweight object no longer shares the property values with the instance of the shared type, but with its own private copy of that shared object.

For example, if you checkout a lightweight object, it is materialized. A copy of the original parent is created with the same r_object_id value as the child and the lightweight object is updated to point to the new parent. Since the private parent has the same r_object_id as the lightweight child, a materialized lightweight object behaves similar to a standard object. As another example, if you delete a non-materialized lightweight object, the shared parent is not deleted (whether or not there are any remaining lightweight children). If you delete a materialized lightweight object, the lightweight child and the private parent are deleted.

When, or if, a lightweight object instance is materialized is dependent on the object type definition. You can define a lightweight type such that instances are materialized automatically when certain operations occur, only on request, or never.

5.4.2 Example of materialized and unmaterialized storage

This section illustrates by example how lightweight objects are stored and how materialization changes the underlying database records. Note that this example only uses the _s tables to illustrate the implementation. The implementation is similar for _r tables.

Suppose the following shareable and lightweight object types exist in a repository:

- customer_record, with a SysObject supertype and the following properties:

  ```
  cust_name string(32),
  cust_addr string(64),
  cust_city string(32),
  cust_state string(2)
  cust_phone string(24)
  cust_email string(100)
  ```

- order_record, with the following properties:

  ```
  po_number string(24)
  parts_ordered string(24)REPEATING
  delivery_date DATE
  billing_date DATE
  date_paid DATE
  ```

  This type shares with customer_record and is defined for automatic materialization.

Instances of the order record type will share the values of instances of the customer record object type. By default, the order record instances are unmaterialized.

The order record instances represented by objID_2 and objID_3 share the property values of the customer record instance represented by objID_B. Similarly, the order record object instance represented by objID_5 shares the property values of the customer record object instance represented by objID_Z. The i_sharing_type property for the parent, or shared, rows in customer_record are set to reflect the fact that those rows are shared.

There are no order record-specific rows created in customer_record_s for the unmaterialized order record objects.

Because the order record object type is defined for automatic materialization, certain operations on an instance will materialize the instance. This does not create a new order record instance, but instead creates a new row in the customer record table that is specific to the materialized order record instance.

Materializing the order record instances created new rows in the customer_record_s table, one row for each order record object, and additional rows in each supertype table in the type hierarchy. The object ID of each customer record object representing a materialized order record object is set to the object ID of the order record object it represents, to associate the row with the order record object. Additionally, the i_sharing_type property of the previously shared customer record object is updated. In the order record objects, the i_sharing_parent property is reset to the object ID of the order record object itself.

5.5 Content support for lightweight SysObjects

Lightweight SysObjects support content storage similarly to regular SysObjects. A lightweight object uses the same storage area and format as its parent and will have the same content facility support as regular SysObjects. That is, a lightweight object can have contents on any of the currently supported storage areas such as distributed store, turbo store, blob store, file store, CA store and external store. Consequently, Accelerated Content Services and Branch Office Caching Services will support lightweight objects as well.

The getFile (and setFile) APIs on a lightweight object will retrieve (and set) the proper content on the lightweight object (and update the page count on the object). Similarly, the addRendition and removeRendition APIs will add and remove the proper renditions to the lightweight object, respectively.

Note that a lightweight object and its original parent object do not share contents. Therefore, if a lightweight object does not have its own content, then the getFile API will not return content from its parent. When a lightweight object is materialized, if it has its own contents, then those contents will point to the cloned parent (note that this is a no-op since the cloned parent would have the same id as the lightweight object); otherwise, the cloned parent will have no contents.

5.6 Using DQL to create lightweight system object types

To use lightweight SysObjects, you will first need to have a shareable parent type, and a lightweight child type.

5.6.1 Create a shareable type

For the shareable type, either alter an existing type or create a new shareable type. Use the following DQL commands to alter or create a shareable type:

```
CREATE SHAREABLE TYPE type_name
[(property_def {,property_def})]
[WITH] SUPERTYPE parent_type [PUBLISH]
```

or

```
ALTER TYPE type_name SHAREABLE [PUBLISH]
```

After the shareable type exists, the attribute type_category in dm_type will have a value 0x00000002 to indicate that a type is a shareable type. A shareable type will have the following new attributes:

- i_sharing_type

  —an ID attribute records the ID of the topmost lightweight type which the sharing lightweight object is an instance of. This attribute will be a null ID before the shareable instance is shared by any lightweight object. After it is shared by the first lightweight object, the following sharing lightweight objects need to be instances of the topmost lightweight type or its subtypes.

- i_orig_parent

  —an ID attribute records the original parent ID of a materialized lightweight object. This attribute will record the ID of the shareable instance when the instance is first shared by any lightweight object and remain the same afterwards even after the lightweight object is materialized and the shareable instance is cloned for the materialized lightweight object.

- allow_propagating_changes

  —a Boolean flag indicating whether any changes applied to the shareable instance which has the consequence of propagating the changes to all its children are allowed. By default, this flag is set to false.

The following are the changes that need to be propagated:

- Changes to any fulltext-indexable attributes.
- Changes to any retention-related attributes. That is, adding new values to the i_retainer_id or dropping existing values from the i_retainer_id.

Any shareable object is not allowed to be versioned or deleted if it is shared by at least one lightweight object. A shareable type is not allowed to become a non-shareable one. A shareable type can only be dm_sysobject or any of its subtypes, but dm_sysobject is not shareable by default, but must be altered.

5.6.2 Create the lightweight type

After you have defined the shareable type, use the following DQL command to create the lightweight type:

```
CREATE LIGHTWEIGHT TYPE type_name
    [(property_def {,property_def})]
 SHARES a_shareable_type
[AUTO MATERIALIZATION |
     MATERIALIZATION ON REQUEST |
     DISALLOW MATERIALIZATION]
[FULLTEXT SUPPORT NONE |
  FULLTEXT SUPPORT |
  FULLTEXT SUPPORT LITE | BASE [ADD attribute_list | ALL]
]
 [PUBLISH]
```

If no optional clauses are specified, the default is AUTO MATERIALIZATION and FULLTEXT SUPPORT NONE. The LIGHTWEIGHT TYPE clause tells server that the new type is a lightweight type and the SHARES clause tells server from what shareable type (and its subtypes) the instances of the new type are sharing. That is, it defines the legal type of objects that can appear in the i_sharing_parent attribute of the type.

A lightweight type can define any of its own attributes as non-qualifiable. If a lightweight type has any non-qualifiable attribute, then it will have its own property bag columns defined on its own base tables. However, the interface will make sure that all property bags of a lightweight object appear as one property bag for the object.

The super type of a lightweight type recorded in dm_type is its parent type. The type_category attribute of dm_type for a lightweight type is set to 0x00000004. A new attribute, shared_parent_name, is added to dm_type to store the name of the shared parent type. This new attribute will help to quickly identify the shared parent type for the subtypes of a lightweight type.

Note that the custom attributes, if any, can not overlap with parent type except the following attributes:

- Subject
- Title

Otherwise, an error will be reported.

You can further subtype the lightweight type using the existing WITH SUPERTYPE clause. The new subtype will have the same type hierarchy as usual. However, you can't mix up the SHARES clause with the WITH SUPERTYPE clause in the same CREATE TYPE or CREATE LIGHTWEIGHT TYPE statement.

After the lightweight type exists, the attribute type_category in dm_type will have a value 0x00000004 to indicate that a type is a lightweight type. In addition to the custom attributes, the lightweight type will have the following attributes:

- r_object_id

  —an ID attribute.

- i_vstamp

  —an integer attribute recording the version stamp of the object. This attribute serves as the mechanism for concurrency control.

- i_partition

  —an integer indicating the partition where the object resides. This attribute will exist only when the lightweight type is declared for partitioning. A lightweight object will reside on the same partition as its parent object.

- i_sharing_parent

  —an ID attribute. This attribute points to the shared parent for the lightweight object. The object will inherit all attributes and policies (that is, ownership, security, retention, aspect, and so on) from the shared parent. This attribute must point to a valid ID to share the properties from the parent. A parent object is not allowed to be destroyed if there are lightweight objects pointing to it. When a lightweight object is materialized, this attribute will store the same ID as the object itself.

- object_name

  —a CHAR(255) attribute

- r_page_cnt

—an integer indicating the number of content files associated with the lightweight object.

5.7 Querying lightweight SysObjects

You can query a lightweight subtype similar to any other regular type. The only difference is that, by default, the parent type will be joined together for the query.

5.7.1 LITE keyword

Add the keyword LITE (similar to the keyword ALL), after the lightweight type name (or any of its subtypes) to only query against the lightweight type without the parent type. For example, if you issue the following DQL statement:

```
SELECT attrs_in_both_types
FROM dm_image_ingestion
WHERE predicates_involve_attrs_in_both_types
```

where dm_image_ingestion is the lightweight type that has dm_sysobject as its shareable parent type, then the query will search the base tables of both dm_image_ingestion and dm_sysobject. However, the following DQL statement will only query against dm_image_ingestion:

The following statement will only query against attributes defined in the lightweight type:

```
SELECT only_attrs_in_the_lightweight_type
    FROM a_lightweight_type (LITE)
    WHERE predicates_involve_only_attrs_in_the_lightweight_type
```

You can still issue other DQL statements against any lightweight type similar to a regular type. This includes the FTDQL hint among others. The LITE keyword can only be used by the superuser.

5.7.2 HIDE_SHARED_PARENT DQL hint

The HIDE_SHARED_PARENT DQL hint directs Documentum CM Server to return only the rows in the query results that are not shared parents. In order to accomplish this, the server will add two additional attributes, r_object_id and r_object_type to the SQL statement select list, if not already there (only applies to dm_sysobject or any of its subtypes). The server will run the SQL query against the database and then for each qualified row it will get the value of r_object_type, fetch the type object and check to see if it is a shareable type. For non-shareable types it will just return the row, but for shareable types it will do the following:

- Look in the sysobject cache for the r_object_id object, or issue an SQL statement if the object is not in the cache, and examine the i_sharing_type.
- Return the row if i_sharing_type is empty, or skip the row if it is not empty (indicating that the row is from a shared parent).

The results of a query that uses the HIDE_SHARED_PARENT DQL hint will not contain any shared parents. A side effect of this hint is that queries will show shared parents without child objects, but the shared parents will disappear from the results when a child LWSO is attached. Webtop uses this hint by default.

This hint does affect performance, since there are the additional checks required to determine whether the result contains shared parents.

5.8 Displaying shared parents of lightweight SysObjects

By default, Web Development Kit applications such as Documentum Webtop and Documentum Administrator do not display shared parents or LWSOs. If you create a shared parent, link it to a folder, and then create a set of LWSOs for that shared parent, you will not see any of those documents displayed in your browser.

Web Development Kit applications, by default, use the HIDE_SHARED_PARENT DQL hint when displaying the documents in a folder. This hint will prevent any Web Development Kit application from displaying any shared parent objects that have LWSOs pointing to them. You can change this default behavior by modifying the application.

You will need to have access to the application server where the Web Development Kit application is deployed, or be able to modify the application WAR file before it is deployed.

You will find the switch that controls the display the shared parents in the wdk/app.xml file of the application, but you will want to add and modify that switch in the custom/app.xml file of the application. By modifying this file, it will be easier to move any customizations to other Web Development Kit applications, and to move them to later versions. Settings in the custom/app.xml file override values in other app.xml files. To add the switch to custom/app.xml, add the following element into the file, inside of the <application></application> element:

```
<lightweight-sysobject>
    <hide-shared-parent>false</hide-shared-parent>
</lightweight-sysobject>
```

There may already be some other elements in the application element, but the order does not matter.

You must restart your application for the change to take effect, if it is already deployed.

5.9 Using Foundation Java API on lightweight SysObjects

Foundation Java API provides methods to create and modify lightweight SysObjects. Many of the methods used to manipulate standard SysObjects apply to lightweight objects, but there are some differences to be aware of.

An unmaterialized lightweight object behaves similar to a standard SysObject for operations that only modify the attributes of the lightweight type. In this case, operation modifies only the lightweight attributes from the lightweight object.

An unmaterialized lightweight object also behaves similar to a standard SysObject for operations that only read the attributes of the object. In this case, the get operation retrieves the lightweight attributes from the lightweight object, and retrieves the parent attributes from the parent.

Operations that modify the attributes of the shared parent cause the lightweight object to materialize (if allowed).

A materialized lightweight object behaves similar to a standard object, in that you can read and modify the attributes of the lightweight and parent object just as with a normal SysObject.

5.9.1 IDfLightObject

The IDfLightObject interface provides methods to manipulate lightweight SysObjects.

- void *dematerialize*()

  Turns a materialized lightweight SysObject back into an unmaterialized lightweight SysObject by reparenting to the original shared parent.

- IDfId *getSharingParent*()

  Returns the ID of the lightweight SysObject's parent.

- boolean *isMaterialized*()

  Indicates whether the lightweight SysObject is materialized.

- void *materialize*()

  Explicitly triggers the object to be materialized. A private parent copy of the shared parent is created, and the object is reparented to the private parent.

- void *reparent*(IDfId newParentId)

  The lightweight SysObject is set to point to the new shareable parent object. Use this method to reparent one child object. Use the method IDfSession.reparentLightObjects() to reparent a list of lightweight SysObjects.

5.9.2 IDfSession

Methods have been added to IDfSession to create and reparent lightweight SysObjects.

- IDfPersistentObject *newLightObject*(String typeName, IDfId parentId)

  Use this method to create a new lightweight SysObject. The typeName must be an existing lightweight type, and the parentId must be an existing object of the shareable type shared by the lightweight type.

- void *reparentLightObjects*(IDfId newParentId, IDfList childIds)

  The method reparents a list of lightweight SysObjects to a new parent. The newParentId must be an existing object of the shareable type shared by the lightweight objects. The childIds must all be of the lightweight type that shares the newParentId type. To reparent a single lightweight object, use the IDfLightObject.reparent() method.

5.9.3 Operations not allowed on a parent

A parent object that has lightweight objects pointing to it cannot be versioned and cannot be destroyed. Similarly, a lightweight object cannot set its i_sharing_parent to any object which has been versioned.

5.9.4 Foundation Java API interface support

Foundation Java API supports casting lightweight objects to the same interface as their sharing parents. For example, if a lightweight object shares a SysObject, then the object can be cast with IDfSysObject and apply the APIs in that interface.

5.9.5 Applying SysObject APIs to lightweight objects

Whether a SysObject API is invocable on lightweight object is based on the principle that the lightweight object is supposed to be sharing (or under the control of) its parent’s attributes, not updating them (unless the lightweight object has been materialized, then it behaves similar to a standard object). Therefore, almost all the getters and status checking APIs will work transparently on the parent.

There are some APIs such as getObjectName/setObjectName (and getTitle/setTitle or getSubject/setSubject if title or subject is defined in the lightweight type), getPageCount, and lock, which will be applied to the lightweight object instead of the parent object, because the lightweight object either has the same attribute or it makes sense to perform the operation on the object.

Content attribute related setter APIs such as setStorageType and setContentType will cause the lightweight object to be materialized, if the type declares auto materialization. Therefore, you should set those attributes on the shared parent before parenting the lightweight objects if you want the objects to be unmaterialized. If the lightweight object materializes and has contents, then the contents will be migrated to the new storage area for the setStorageType API or disallowed for setContentType API. Direct content-related setter APIs (such as setFile and setContent) will set content on the lightweight object and do not automatically cause materialization.

Other setter APIs (such as setStatus) are not allowed unless the type declaration is AUTO MATERIALIZATION, in which case the original parent object will be cloned with the changes and the lightweight object will point to the cloned parent.

The following sections will discuss this matter in detail.

Note: The following sections will not give an exhaustive list; they just give you the basic ideas of how the APIs work with lightweight SysObjects.

5.9.5.1 APIs that work on parent SysObjects transparently

Almost all the getters such as getAcl (except the getObjectName and getPageCount, and the getTitle and getSubject, which return the respective values of the object_name and page count from the lightweight object) and the status checking APIs (such as isFrozen) will directly return the corresponding attribute values from the parent object.

5.9.5.2 APIs that work on lightweight objects only

If an API is applied to an attribute which has the same name as the parent, then it will be applied to the lightweight object. APIs such as getPageCount, getObjectName and setObjectName (and perhaps getTitle and getSubject) are in this category. For the lock API, it makes sense to perform the operation on the lightweight object. All the content-related getter and setter APIs will be applied to the lightweight object, accordingly.

5.9.5.3 APIs that cause lightweight objects to be materialized

The following APIs will cause the lightweight object to be materialized so that existing applications will work properly:

- All the non-content related setters.
- If the lightweight object does not have any content, then content attribute related APIs such as setStorageType and setContentType will cause the lightweight object to be materialized. Otherwise, the setStorageType will cause the contents to be migrated to the new storage area after the materialization and the setContentType will be disallowed.
- The link/unlink API. This implies that all of the lightweight child objects must be in the same folder as their shared parent. You can link the shared parent object to a folder before adding lightweight children.
- The checkout/checkin API. Note that both lightweight objects and the cloned parent will have new versions.
- Saveasnew. If saveasnew is applied to a non-materialized lightweight object, then only the lightweight object will be cloned as a new object pointing to the

same parent. If it is applied to a materialized lightweight object, then both the lightweight object and its parent will be cloned as another new materialized lightweight object.

- Branch. If branch is applied to a non-materialized lightweight object, it is disallowed. If it is applied to a materialized lightweight object, then both the lightweight object and its parent will be cloned as a branched version of the original one.
- Set retention.
- Set VDM children and assemble APIs.

5.9.5.4 APIs that are disallowed on lightweight objects

The following APIs are not allowed on lightweight objects:

- It is not allowed to register a lightweight object for auditing or event notification before it is materialized. If a parent is registered for auditing or event notification, then the child will be audited or notified. Moreover, when the child is materialized, the cloned parent will be automatically registered for auditing or event notification. We will allow the lightweight type itself to be registered which will result in all the lightweight objects of that type to be audited or notified.
- Replication.
- Create a distributed reference to a lightweight object unless it is materialized.
- Web caching and web publishing.
- Be visible in a folder unless it is materialized. This will change, subject to component support.
- Freeze and unfreeze, unless it is materialized.
- Prune, unless materialized.
- Inbox related APIs (such as route, forward, and so on) unless materialized.
- Lifecycle related APIs (such as attach, promote, and so on) unless materialized.

5.9.5.5 Operations applied to a SysObject that affects a lightweight object

If the parent object is registered for auditing or event notification, then changes to the lightweight object will be audited or notified accordingly. Setting the policy-related attributes (for example: security, retention, owner, lifecycle, and others) on the parent object will affect the lightweight object as well. Changing the parent’s a_storage_type will only affect new contents created for the lightweight objects, not for existing contents.

5.9.5.6 Operations using a lightweight object in place of a SysObject

After a lightweight object is materialized, you can use it in any place where a SysObject is allowed. However, if the lightweight object has not been materialized, then the application needs to be enhanced so that it knows how to deal with the lightweight object. For example, you will be able to place the lightweight object in a dm_relation object as the parent or the child. However, the application using the relationship needs to know how to deal with the special type in the relationship.

5.10 Using DQL to migrate standard objects to lightweight objects

If you want to change your current data model and move standard SysObjects into shareable parents and lightweight objects, you can use the administrative method MIGRATE_TO_LITE. One form of this command will take all the standard objects of a particular type and split them into a lightweight SysObject and shareable parent SysObject. Since each standard SysObject is converted to a lightweight SysObject that has its own shareable parent, this method does not reduce the data footprint, but can be an intermediate step towards this goal.

The other form of this method allows you to specify a series of SQL query predicates to select a set of lightweight objects and parent them with a specified shareable parent.

5.11 Lightweight SysObject example

The following example is a basic Foundation Java API program that uses a loop to create some lightweight SysObjects with content. Since setting the content type or storage type on a lightweight object will cause it to materialize, these attributes are set on the shared parent first. The following is a snippet from the example program that creates the shared parent, and then creates a lightweight child. The shareable parent type is test_payment_bank, and the child lightweight type is test_payment_check. Just a few of the method calls are shown in this snippet, but the full program follows:

```
            ...
            sysObj = (IDfSysObject)session.newObject("test_payment_bank");
            ...
            sysObj.save();
            objId = sysObj.getObjectId();
            ...
                sysObj =
(IDfSysObject)session.newLightObject("test_payment_check",objId);
```

The following is a simple standalone example program that creates a shared parent object and ten lightweight objects that share the parent. Be sure to have a copy of `dfc.properties` in the classpath for the program:

```
/*
 * lwsoExample.java
 * This example assumes that you have created two types, test_payment_bank
 * and test_payment_check. Here are the two DQL statements to create the types:
CREATE SHAREABLE TYPE test_payment_bank (
bank_code integer,
routing integer
) WITH SUPERTYPE dm_sysobject PUBLISH

CREATE LIGHTWEIGHT TYPE test_payment_check (
account integer,
check_number integer,
transaction_date date,
amount float
) SHARES test_payment_bank PUBLISH
 * This example also assumes that there is a check image GIF file in the
 * working directory (use any GIF to stand in for a check image), and that
 * the repository name, username, and password are passed to the program
 * on the command line.
 */
import com.documentum.com.DfClientX;
import com.documentum.com.IDfClientX;
import com.documentum.fc.client.*;
import com.documentum.fc.common.*;
public class lwsoExample {
    public void begin(String username, String password, String docbaseName) {
        IDfClientX clientx;
        IDfClient client;
        IDfSession session = null;
        IDfSessionManager sMgr = null;
        IDfSysObject sysObj;
        IDfId objId;
        try {
            // Create a Client object
            clientx = new DfClientX();
            client = clientx.getLocalClient();
            // Create a Session Manager object
            sMgr = client.newSessionManager();
            // Create an IDfLoginInfo object
            IDfLoginInfo loginInfoObj = clientx.getLoginInfo();
            loginInfoObj.setUser(username);
            loginInfoObj.setPassword(password);
            // Bind the Session Manager to the login info
            sMgr.setIdentity(docbaseName, loginInfoObj);
            session = sMgr.getSession(docbaseName);
             // Create Parent
            sysObj = (IDfSysObject)session.newObject("test_payment_bank");
            sysObj.setObjectName("check_parent");
            // Storage type MUST be set on the parent, otherwise the
            // lightweight object will be materalized.
            // An alternative is to use the DQL statement:
            // ALTER TYPE test_payment SET DEFAULT STORAGE 'filestore_01'
            sysObj.setStorageType("filestore_01");
            // Content type MUST be set on parent, otherwise the
            // lightweight object will be materalized.
            sysObj.setContentType("gif");
            // Set shared attributes
            sysObj.setInt("bank_code", 1);
            sysObj.setInt("routing", 231381116);
            sysObj.save();
            objId = sysObj.getObjectId();
            for (int i=0; i<10; i++) {
                            // Create 10 childern
                IDfTime currDate = new DfTime();
                sysObj = (IDfSysObject)session.newLightObject(
                        "test_payment_check",objId);
                sysObj.setObjectName("check_" + i);
                sysObj.setFile("check.gif");
                // Set individual attributes
                sysObj.setInt("check_number",i);
                sysObj.setTime("transaction_date", currDate);
                sysObj.setInt("account", 1000000);
                sysObj.setDouble("amount", 25.00);
                sysObj.save();
            }} catch(Throwable e) {
            if(e instanceof DfException) {
                System.out.println("DFC Exception:");
                String s = ((DfException)e).getStackTraceAsString();
                System.out.println(s);}
                        else {
                System.out.println("Non DFC Exception");
                e.printStackTrace();
            }} finally {
            if (session != null)
                sMgr.release(session);
        }}
    public static void main(String[] args)
    throws java.lang.InterruptedException {
        String docbase = args[0];
        String user = args[1];
        String password = args[2];
        new lwsoExample().begin(user, password, docbase);
    }
}
```

Chapter 6

PARTITION_OPERATION method

The information in this chapter is from *OpenText Documentum Content Management - Server DQL Reference Guide (EDCCS250400-DRD)*, describing the PARTITION_OPERATION administration method.

6.1 PARTITION_OPERATION

Creates partitioning scheme objects and SQL scripts to control repository partitioning.

Table 6-1: PARTITION_OPERATION arguments

Argument	Datatype	Value	Description
operation	string	listed in description column	Defines the operation that you want to execute. The only values allowed are: • create_scheme • db_partition • add_partition • delete_partition • split_partition • merge_partition • switch_out • switch_in
partition_scheme	string	*scheme_name*	
partition_name	string	*partition_name*	The name of the partition. Some databases may restrict the choices for a partition name. For example, choosing default as a partition name causes an error when you execute the partitioning script.
range	integer	*right_limit*	Uppermost i_partition value of an object placed in this partition is equal to (*right_limit* - 1).

Argument	Datatype	Value	Description
tablespace	string	*storage_name*	The tablespace for the partition. You can find the tablespace name for your repository by accessing the underlying database. By default, the name is dm_<repository_name>_docbase.
type_name	string	*type_name*	Specifies a type to partition. Must be a partitionable supertype. Create the type with CREATE PARTITIONABLE TYPE, or modify a type to be partitionable with ALTER TYPE ... ENABLE PARTITION.
table_name	string	*table_name*	Specifies a registered table. The table must already have an i_partition column. Do not use this for repository-created tables; use the type_name argument for those tables.
owner_name	string	*owner_name*	The table owner name.
source_name	string	*source_name*	The name of the partition to split.
temp_table_suffix	string	*temp_table_suffix*	The suffix for the temporary table. When tables are switched in or out, the table name must be the name of the type or registered table with the suffix applied. Use a short, one or two character suffix.

6.1.1 Syntax

To create a partitioning scheme object:

```
EXECUTE partition_operation WITH
operation='create_scheme',partition_scheme='scheme_name'{,partition_name='partition_name'
,range=right_limit,tablespace='storage_name'}
```

To generate a script to apply a partition scheme to types and/or registered tables:

```
EXECUTE partition_operation WITH
operation='db_partition',partition_scheme='scheme_name'{,type_name='type_name'}
{,table_name=table_name,table_own='table_owner'}
```

To add a partition (range must be greater than the upper bound of the scheme):

```
EXECUTE partition_operation WITH
operation='add_partition',partition_scheme='partition_scheme'{,partition_name='partition_
name',range='right_limit',tablespace='storage_name'}
```

To delete a partition (must be last partition in the scheme):

```
EXECUTE partition_operation WITH
operation='delete_partition',partition_scheme='partition_scheme',partition_name='partitio
n_name'
```

To split a partition (range must be one of the values defined for the partition to be split, the source partition):

```
EXECUTE partition_operation WITH
operation='split_partition',partition_scheme='partition_scheme',source_name='source_name'
,partition_name='partition_name',range='right_limit',tablespace='storage_name'
```

To merge a partition with the one to its left (with a lower range):

```
EXECUTE partition_operation WITH
operation='merge_partition',partition_scheme='partition_scheme',partition_name='partition
_name'
```

To switch out a partition to a temporary table:

```
EXECUTE partition_operation WITH
operation='switch_out',partition_scheme='partition_scheme',temp_table_suffix='temp_table_
suffix,partition_name='partition_name',execute_now=TRUE|FALSE
```

To switch data into a partition from a temporary table:

```
EXECUTE partition_operation WITH
operation='switch_in',partition_scheme='partition_scheme',temp_table_suffix='temp_table_s
uffix,partition_name='partition_name',execute_now=TRUE|FALSE
```

6.1.2 Arguments

Table 6-2: PARTITION_OPERATION arguments

Argument	Datatype	Value	Description
operation	string	listed in description column	Defines the operation that you want to execute. The only values allowed are: • create_scheme • db_partition • add_partition • delete_partition • split_partition • merge_partition • switch_out • switch_in
partition_scheme	string	*scheme_name*	
partition_name	string	*partition_name*	The name of the partition. Some databases may restrict the choices for a partition name. For example, choosing default as a partition name causes an error when you execute the partitioning script.
range	integer	*right_limit*	Uppermost i_partition value of an object placed in this partition is equal to (*right_limit* - 1)
tablespace	string	*storage_name*	The tablespace for the partition. You can find the tablespace name for your repository by accessing the underlying database. By default, the name is dm_<repository_name>_docbase.

Argument	Datatype	Value	Description
type_name	string	*type_name*	Specifies a type to partition. Must be a partitionable supertype. Create the type with CREATE PARTITIONABLE TYPE, or modify a type to be partitionable with ALTER TYPE ... ENABLE PARTITION.
table_name	string	*table_name*	Specifies a registered table. The table must already have an i_partition column. Do not use this for repository-created tables; use the type_name argument for those tables.
owner_name	string	*owner_name*	The table owner name.
source_name	string	*source_name*	The name of the partition to split.
temp_table_suffix	string	*temp_table_suffix*	The suffix for the temporary table. When tables are switched in or out, the table name must be the name of the type or registered table with the suffix applied. Use a short, one or two character suffix.

6.1.3 Return value

For create_scheme, PARTITION_OPERATION returns a Boolean value and an object id. The Boolean value indicates if the operation succeeded. The object ID is either the ID of the partition scheme object, or the object ID of the SQL script that is run against the database to perform the partitioning operation.

Note: One way to retrieve the script is to issue the following DQL statement (assume that the result of PARTITION_OPERATION returned the object ID 0855706080007c19):

```
EXECUTE get_file_url FOR '0855706080007c19' WITH "format"='text'
```

That statement will return the location of the file.

6.1.4 Permissions

You must have Sysadmin or Superuser privileges to use all the operations of this method. If you have type owner privileges, you can use the db_partition operation for that type.

In Oracle installations, you must be logged in as sysdba to run the script.

6.1.5 Descriptions

Use this method to create a partitioning scheme object (dm_partition_scheme). This object will record the types and tables that use the scheme, the name and range of each partition, and the tablespace (or filegroup) where each partition is stored. When a partitioning scheme object is first created, there are no types or registered tables associated with it, just the name, range and tablespace for the partitions in the scheme. When you associate types or registered tables with a partitioning scheme, a script is created that must be run against the underlying database to partition the database tables to match the partitioning scheme. Documentum CM Server does not run this script since it may take a long time to perform the partitioning on a large database, and you may want to examine the script before running it.

You also use this method to modify a partition scheme. Modifying the scheme also creates a database partitioning script that Documentum CM Server runs against the database for you.

When you run the script-creating variations of this method, the method returns the r_object_id of the partitioning script to run in your database. The partition scheme object information and the database are not consistent until the script is run, so the is_validated property of the partitioning scheme object is set to FALSE. After the script is created, you must run it against the underlying database, using the database's SQL facility. The scripts make the changes to the database that reflect the information in the partition scheme object. As the script is successfully run against the database, the is_validated property of the partitioning scheme object is set to TRUE.

If you did not enable partitioning when you created your repository, create a scheme, and then use the db_partition operation without specifying any tables or types to partition the repository intrinsic types.

The following are the intrinsic type hierarchies that are partitioned when you create a repository with partitioning enabled, or you later enable partitioning:

- dm_acl
- dm_sysobject
- dmr_containment
- dm_assembly
- dm_relation
- dm_relation_type
- dmi_otherfile
- dmi_replica_record
- dmr_content
- dmi_subcontent
- dmi_queue_item

6.1.6 Originally partition-enabled repository compared to originally non-partitioned repository

You can use partitioning even if you do not create a partition-enabled repository. You can partition the repository after it has been created. In either case, you will have a partitioned repository.

If you create a partition-enabled repository, the 11 intrinsic type hierarchies listed in the previous section will be partitioned, each in its own partition scheme. If you want more than one of those hierarchies to be partitioned identically, you will need to run PARTITION_OPERATION on each hierarchy you want to modify separately. However, you can use separate partitioning schemes on those hierarchies if you need to for your application. There is no way to merge the partition schemes so that multiple hierarchies are configured with a single scheme. You can add custom type hierarchies (where the topmost custom type specifies supertype NULL), to any of the intrinsic type schemes and manage them together, if needed.

In contrast, if you partition-enable your repository after creation, all the intrinsic types are configured with a single partitioning scheme. In this case, all the type hierarchies are partitioned identically. There is no way to separate them into different partition schemes if you later decide you need to do this.

You should plan your repository partitioning carefully before deciding which alternative meets your needs.

6.1.7 Create_scheme

To create a partition scheme, use the create_scheme operation. You will specify a name for the scheme. Each scheme has a unique name. In this document we use the convention that the name of a partition scheme ends with the characters _sch. In addition to the scheme, you will specify one or more partition name, range, and tablespace. The name is used to refer to the partition. The range is a single integer that is one larger than the largest value in the i_partition attribute for any objects in that partition. The tablespace specifies where the partition is stored. The r_object_id of the partition scheme object is returned.

The following command creates a partition scheme named my_partition_sch, with four partitions named zero, first, second, and third. Partition zero contains objects with i_partition values from 0 to 9, partition first contains objects with i_partition values from 10 to 19, partition second has 20 to 29, and partition third has 30 to 39. Notice that the range for partition zero is set to 10, but partition zero contains objects with i_partition values from 0 to 9:

```
EXECUTE partition_operation with "operation"='create_scheme',
"partition_scheme"='my_partition_sch',
"partition_name"='zero',range=10,"tablespace"='dm_techpubsglobal_docbase',
"partition_name"='first',range=20,"tablespace"='dm_techpubsglobal_docbase',
"partition_name"='second',range=30,"tablespace"='dm_techpubsglobal_docbase',
"partition_name"='third',range=40,"tablespace"='dm_techpubsglobal_docbase'
```

6.1.8 Db_partition

To apply a partition scheme, use the db_partition operation. You specify the partition scheme to apply, and the types and tables to partition. This operation adds the specified types and registered tables to the partition scheme object, and creates a script that partitions the database tables. A type or registered table can be associated with only one partition scheme, so the command will fail if a type or registered table is listed in any other partition scheme.

Use this operation to partition-enable a repository that was not originally created as partition-enabled. If you run this operation and do not specify any types or registered tables, the base types will be partitioned. In addition, record the note about increasing the number of cursors for an Oracle installation.

After you have created a partition scheme, perform these steps to partition the repository:

1. Use the db_partition operation of the PARTITION_OPERATION administrative method to create the partitioning script.
2. Stop the Documentum CM Server.
3. Execute the partitioning script against the database.
4. Restart the Documentum CM Server.

Note: Do not execute the script if there have been changes to the repository schema after it was generated. Create a new script if the schema changes.

The following command creates a script to apply my_partition_sch to the type, my_type. The command returns the r_object_id of the script, and sets the is_validated property of my_partition_sch to FALSE:

```
EXECUTE partition_operation with "operation"='db_partition',
"partition_scheme"='my_partition_sch',"type_name"='my_type'
```

After running the script, the is_validated property is TRUE, and the tables in the underlying database are partitioned as set out in the my_partition_sch object.

6.1.9 Add_partition

To add a partition, use the add_partition operation. You specify the partition name, range, and tablespace for each partition to add. The range of the new partition must be greater than the upper bound of the current partition scheme. Any types or registered tables already attached to this partition scheme will be modified. This operation modifies the partition information in the partition scheme object, and creates and runs a script to modify the partitioning in the database.

The following command adds a partition to the partition scheme my_partition_sch, and runs a script to add the additional partition to the database tables. In this example assume that my_type was partitioned in the earlier db_partition example:

```
EXECUTE partition_operation WITH "operation"='add_partition',
"partition_scheme"='my_partition_sch'
,"partition_name"='fourth',range=50,"tablespace"='dm_techpubsglobal_docbase'
```

6.1.10 Delete_partition

To delete a partition, use the delete_partition operation. The partition must be the last partition in the partition scheme. Similar to add_partition, this operation modifies the partition information in the partition scheme object, and runs a script to modify the partitioning in the database.

The following command removes the last partition (created in the add_partition example):

```
EXECUTE partition_operation WITH "operation"='delete_partition',
"partition_scheme"='my_partition_sch',"partition_name"='fourth'
```

6.1.11 Split_partition

To split a partition, use the split_partition operation. You specify a source partition to split, and the name, range, and tablespace of the new partition. The range value must fall into the defined range of the source partition. The new partition is to the left of the source partition (the new partition range is lower than the modified source partition).

The following command splits the zero partition into partitions five and zero:

```
EXECUTE partition_operation WITH "operation"='split_partition',
"partition_scheme"='my_partition_sch',
"source_name"='zero',
"partition_name"='five',range=5,"tablespace"='dm_techpubsglobal_docbase'
```

If you used the previous examples, before executing this command, the partition names and ranges are: zero (10), first (20), second (30), third (40). After executing this command, they are: five (5), zero (10), first (20), second (30), third (40).

6.1.12 Merge_partition

To merge two partitions, use the merge_partition operation. This operation merges the partition specified into the partition on its left (into the lower range numbered partition).

The following command merges the partition named five into the next lower range partition (partition zero, if you have used the previous examples):

```
EXECUTE partition_operation WITH "operation"='merge_partition',
"partition_scheme"='my_partition_sch',"partition_name"='five'
```

6.1.13 Switch_out

To move data out of a partition into temporary tables, use the switch_out operation. For some databases (SQL Server), the temporary tables (if they have already been created), must be empty. After the switch_out operation, the temporary table contents will be in the partition, and the data previously in the partition will reside in the temporary tables for the types and registered tables in that partition scheme. If you specify execute_now as TRUE, Documentum CM Server will execute the SQL script and switch out the data, otherwise you will need to run the script yourself. The temporary table names will match the type or registered table name that you switch out with the temp_table_suffix added.

While switch_out is executing, the data in the partition is not consistent, so you must insure that there is no repository access to that partition during the switch.

The following example switches out the data in the second partition into a temporary table named my_type_x (my_partition_sch only has one type, my_type, in it). Since execute_now is specified, you do not need to run the SQL script; it is done for you:

```
EXECUTE partition_operation WITH "operation"='switch_out',
"partition_scheme"='my_partition_sch',
"temp_table_suffix"='_x',"partition_name"='second',"execute_now"=TRUE
```

6.1.14 Switch_in

To move data into a partition from temporary tables, use the switch_in operation. The partition must be empty for SQL Server. Create the temporary tables and fill them with the data to switch into the partition. After the switch_in operation, the temporary table contents will be in the partition, and the data previously in the partition will reside in the temporary tables. If you specify execute_now as TRUE, Documentum CM Server will execute the SQL script and switch out the data, otherwise you will need to run the script yourself. The temporary table names will match the type or registered table name that you switch out with the temp_table_suffix added.

You must have created all the temporary tables to switch into the empty partition. One quick way to create those tables, is to use the switch_out operation on an empty partition. When the operation executes, it will create all the required tables. Since the partition was empty, all the tables will be, too. You can then fill the tables with data.

While switch_in is executing, the data in the partition is not consistent, so you must insure that there is no repository access to that partition during the switch.

The following example switches in the data in the from the temporary table named my_type_x (my_partition_sch only has one type, my_type, in it) into the partition named second. Since execute_now is specified, you do not need to run the SQL script; it is done for you:

```
EXECUTE partition_operation WITH "operation"='switch_in',
"partition_scheme"='my_partition_sch',
"temp_table_suffix"='_x',"partition_name"='second',"execute_now"=TRUE
```

6.1.15 Oracle installations

Note: If you experience difficulties with the following information, please consult the Oracle documentation or Oracle support for assistance.

In order to use this method with an Oracle installation, you must run the script as SYSDBA. Additionally, if you are partitioning a non-partitioned database, you may want to increase the number of open database cursors, or the script may exit with the error:

```
ORA-0100 Max Opened Cursors Exceeded
```

To increase the number of cursors, consult your Oracle documentation. It may tell you to use a command similar to:

```
ALTER SYSTEM SET OPEN_CURSORS=2000 SCOPE=MEMORY SID='*';
```

For example, if dmadmin is the installation owner:

```
C:\Documents and Settings\dmadmin>sqlplus "/as sysdba"
@ C:\Documentum\data\testenv\content_storage_01\00000057\80\00\01\19.txt
```

Additionally, if you are partitioning a non-partitioned database, you may want to increase the number of open database cursors, or the script may exit with the error:

```
ORA-0100 Max Opened Cursors Exceeded
```

to alter the number of open cursors.

If you exit the script with an error (from inadvertently exceeding the number of open cursors, for example), correct the error, and rerun the script, you may see an error message as follows:

```
ORA-12091: cannot online redefine table "TECHPUBS"."DMC_WFSD_ELEMENT_S" with
materialized views
```

or as follows:

```
ORA-23539: table "TECHPUBS"."DM_PLUGIN_R" currently being redefined
```

caused by leftover temporary items from the previous script failure. One way to correct this error is to run a command similar to:

```
execute DBMS_REDEFINITION.ABORT_REDEF_TABLE('<Schema Name>','<Table Name>'
,'<Table Name>I');
```

Where <Table Name>I is the intermediate table name used for the redefinition. From the first error message, we would use this command:

```
execute DBMS_REDEFINITION.ABORT_REDEF_TABLE('TECHPUBS','DMC_WFSD_ELEMENT_S'
,'DMC_WFSD_ELEMENT_SI');
```

or from the second:

```
execute DBMS_REDEFINITION.ABORT_REDEF_TABLE('TECHPUBS','DM_PLUGIN_R'
,'DM_PLUGIN_RI');
```

6.1.16 Switch in and switch out partitions

Switching data into a partition must be carefully planned. Typically, you will create a number of offline tables to load with data, use whatever native database method is available to load the tables into temporary tables in the database, create the table indexes, and then swap the tables into the prepared repository partition. This technique can load large amounts of data into a repository while causing a minimum of disruption to normal use of the repository. This technique can also be used to remove large amounts of data from a repository by switching out a partition for empty offline tables.

The typical steps you would take to do a partition exchange involve the following:

1. Create the offline tables.
2. Load the offline tables.

   Load the tables with data using whatever methods are available to you with your database.
3. Create the offline index tables.

   The offline tables must index the same properties as the online objects do. The schema must be identical. Create the offline index tables in the same tablespace as the current online indexes.
4. Exchange the partition for the offline tables.

   Run the PARTITION_OPERATION administration method to switch in or switch out the data. After following these steps, the data that was previously in the offline tables is in the online partition, and the previously online data is now in the offline table.
5. Validate the exchange.

   Check the online data to be sure that the exchange was successful.

Chapter 7

GENERATE_PARTITION_SCHEME_SQL method

The information in this chapter is from *OpenText Documentum Content Management - Server DQL Reference Guide (EDCCS250400-DRD)*, describing the GENERATE_PARTITION_SCHEME_SQL administration method. This method is deprecated.

7.1 GENERATE_PARTITION_SCHEME_SQL – Deprecated

Starting from the 6.6 release, this method is deprecated. See "PARTITION_OPERATION" on page 65, for its replacement. Creates an SQL script to control repository partitioning.

7.1.1 Syntax

To generate a script to partition a database:

```
EXECUTE generate_partition_scheme_sql WITH [operation='db_partition',]
[type_name='type_name',|table_name='regtable_name',[owner_name='owner_name',]]
[last_partition ='partition_name',last_tablespace='tablespace',]
partition_name='partition_name',range=integer,tablespace='tablespace'
{,partition_name='partition_name',range=integer,tablespace='tablespace'}
[,include_object_type={TRUE|FALSE}]
```

To generate a script to add partition(s) to a database:

```
EXECUTE generate_partition_scheme_sql WITH operation='add_partition',
[type_name='type_name',|table_name='regtable_name',[owner_name='owner_name',]]
partition_name='partition_name',range=integer,tablespace='tablespace'
{,partition_name='partition_name',range=integer,tablespace='tablespace'}
[,include_object_type={TRUE|FALSE}]
```

To generate a script to exchange a partition:

```
EXECUTE generate_partition_scheme_sql WITH operation='exchange_partition',
[temp_table_suffix='temp_table_suffix'],
type_name='type_name',
partition_name='partition_name'
,include_object_type={TRUE|FALSE}
```

```
EXECUTE generate_partition_scheme_sql WITH operation='exchange_partition',
[temp_table_suffix='temp_table_suffix'],
table_name='regtable_name',[owner_name='owner_name',]
partition_name='partition_name'
,include_object_type={TRUE|FALSE}
```

7.1.2 Arguments

Table 7-1: GENERATE_PARTITION_SCHEME_SQL arguments

Argument	Datatype	Value	Description
operation	string	db_partition, add_partition, or exchange_partition	Defines the operation that you want to execute. The only values allowed are: db_partition, add_partition, and exchange_partition. Db_partition creates a script to partition the database, add_partition creates a script to add a partition to the database, and exchange_partition creates a script to exchange a partition.
type_name	string	*type_name*	Specifies a type to partition. Must be a supertype.
table_name	string	*regtable_name*	Specifies a registered table. The table must already have an i_partition column. Do not use this for repository-created tables; use the type_name argument for those tables.
owner_name	string	*owner_name*	The table owner name.
last_partition	string	*partition_name*	The name of the partition for objects whose i_partition value is larger than the highest range defined.
last_tablespace	string	*tablespace*	The tablespace of the last partition.
partition_name	string	*partition_name*	The name of the partition.

Argument	Datatype	Value	Description
range	integer	*integer*	Uppermost i_partition value of an object placed in this partition.
tablespace	string	*tablespace*	The tablespace for the partition.
temp_table_suffix	string	*temp_table_suffix*	The suffix for the temporary table. The default is “x”.
include_object_type	Boolean	TRUE | FALSE	Whether to include the internal table dmi_object_type in partitioning. The table can only be included if it has already been partitioned.

7.1.3 Return value

GENERATE_PARTITION_SCHEME_SQL returns an object id for the text file containing the SQL script to generate the partitioning.

7.1.4 Permissions

You must have Sysadmin or Superuser privileges to use this method.

In Oracle installations, you must be logged in as sysdba to run the script.

7.1.5 Description

Note: Starting from the 6.6 release, this method is deprecated and is replaced by PARTITION_OPERATION.

Use this method to generate a database partitioning script. After the script is generated, it is run against the underlying database, using the database's SQL facility. For all versions of this method, except for add_partition, stop Documentum CM Server before you run the script against the database, then restart Documentum CM Server for the changes to take effect. For the add_partition method, you do not need to stop or restart Documentum CM Server.

The first form of the command, in which operation='db_partition', allows you to specify a type or a table to partition. If you do not specify a type or table, the generated script will partition all the partitionable types in the repository if it is run. You would use this command to partition a repository that was upgraded from an earlier version. The first range value means that objects with i_partition values from 0 to the first value will be in the first partition. The second partition will contain objects with i_partition values greater than first range value up to the second range value. Each subsequent partition will contain all the objects with i_partition values greater than the previous partition and up to its value. The last partition contains those objects with i_partition values greater than any previously specified partition.

The second form of the command, in which operation='add_partition', allows you to add partitions. This form is similar to the first form, but the first added partition begins with values greater than previously defined partitions. If there happen to be objects in the last partition whose i_partition values fall into the new partition's range, they will be moved into the new partition.

If you create a new repository with partitioning enabled, there is only one partition, called the last partition. You can then customize your repository by adding partitions. In this case, the first added partition range goes from 0 to the value you specified.

The final two versions, in which operation='exchange_partition', allow you to exchange a partition with a schema-identical offline table in the database tablespace. Commonly, you would use this feature to load a large number of objects into a table and then swap the table into the partition.

7.1.6 Oracle installations

In order to use this method with an Oracle installation, you must run the script as SYSDBA. For example, if dmadmin is the installation owner:

```
C:\Documents and Settings\dmadmin>sqlplus "/as sysdba"
@ C:\Documentum\data\testenv\content_storage_01\00000057\80\00\01\19.txt
```

Additionally, if you are partitioning a non-partitioned database, you may want to increase the number of open database cursors, or the script may exit with the error:

```
ORA-0100 Max Opened Cursors Exceeded
```

To increase the number of cursors, consult your Oracle documentation. It may tell you to use a command similar to:

```
ALTER SYSTEM SET OPEN_CURSORS=2000 SID='*' SCOPE=MEMORY;
```

or for Oracle versions earlier than Oracle 11:

```
ALTER SYSTEM SET OPEN_CURSORS=2000 SCOPE=MEMORY SID='*';
```

to alter the number of open cursors.

If you exit the script with an error (from inadvertently exceeding the number of open cursors, for example), correct the error, and rerun the script, you may see an error message as follows:

```
ORA-12091: cannot online redefine table "TECHPUBS"."DMC_WFSD_ELEMENT_S" with
materialized views
```

caused by leftover temporary items from the previous script failure. One way to correct this error is to run a command similar to:

```
execute DBMS_REDEFINITION.ABORT_REDEF_TABLE('<Schema Name>', '<Table Name>'
,'<Table Name>I');
```

Where <Table Name>I is the intermediate table name used for the redefinition. From the previous error message, we would use this command:

```
execute DBMS_REDEFINITION.ABORT_REDEF_TABLE('TECHPUBS', 'DMC_WFSD_ELEMENT_S'
,'DMC_WFSD_ELEMENT_SI');
```

7.1.7 SQL Server installations

If you are using a SQL Server installation, use filegroup names as values for the tablespace attributes, since SQL Server filegroups correspond to tablespaces.

For example, use:

```
tablespace='filegroup'
```

and

```
last_tablespace='filegroup'
```

for SQL Server installations.

7.1.8 Partition exchange

Partition exchange must be carefully planned. Typically, you will create a number of offline tables to load with data, use whatever native database method is available to load the tables, create the table indexes, and then swap the tables into the prepared repository partition. This technique can load large amounts of data into a repository while causing a minimum of disruption to normal use of the repository. This technique can also be used to remove large amounts of data from a repository by swapping out a partition for small offline tables.

The typical steps you would take to do a partition exchange involve the following:

1. Identify the offline tables to create.
2. Load the offline tables.

   Load the tables with data using whatever methods are available to you with your database.
3. Create the offline index tables.

   The offline tables must index the same properties as the online objects do. The schema must be identical. Create the offline index tables in the same tablespace as the current online indexes.
4. Exchange the partition for the offline tables.

   Run the GENERATE_PARTITION_SCHEME_SQL administration method to generate the partitioning script, stop Documentum CM Server, run the script against the RDBMS, and restart Documentum CM Server. After following these steps, the data that was previously in the offline tables is in the online partition, and the previously online data is now in the offline table.

5. Validate the exchange.

   Check the online data to be sure that the exchange was successful.

The *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)* contains more information about executing a partition swap.

7.1.9 Examples

This example generates a script to partition all the partitionable types in a repository. After the script is run, partitionable objects with an i_partition value of 0 to 10 will be stored in partition P1 of the database. Partitionable objects with an i_partition value of 11 to 20 will be stored in partition P2 of the database.

```
EXECUTE generate_partition_scheme_sql WITH
"partition_name"='P1',"range"=10,"tablespace"='dm_techpubs_docbase',
"partition_name"='P2',"range"=20,
"tablespace"='dm_techpubs_docbase'
```

To get the script, you can issue the following DQL statement (assume that the result of the earlier method returned the object ID 0855706080007c19): `EXECUTE get_file_url FOR '0855706080007c19' WITH "format"='text'`

Chapter 8

MIGRATE_TO_LITE method

The information in this chapter is from *OpenText Documentum Content Management - Server DQL Reference Guide (EDCCS250400-DRD)*, describing the MIGRATE_TO_LITE administration method.

8.1 MIGRATE_TO_LITE

Migrates standard objects to lightweight objects.

8.1.1 Syntax

To generate, and optionally run, a script to split standard objects into a shareable parent object and a lightweight object:

```
EXECUTE migrate_to_lite WITH source_type='source_type'

,shared_parent_type='shared_parent_type'
,execution_mode='execution_mode_split'[,recovery_mode=TRUE|FALSE]
,parent_attributes='parent_properties'
[ft_lite_add='property_list' | ft_base_add='property_list']
```

To generate, and optionally run, a script to migrate standard objects to lightweight objects:

```
EXECUTE migrate_to_lite WITH

shared_parent_type='shared_parent_type'

[,lightweight_type='lightweight_type']

,execution_mode='execution_mode'[,recovery_mode=TRUE|FALSE]

{,parent_sql_predicate='parent_sql_predicate'}

{,parent_id='ID'}

[,default_parent_id='ID']
[ft_lite_add='property_list' | ft_base_add='property_list']
```

8.1.2 Arguments

Table 8-1: MIGRATE_TO_LITE arguments

Argument	Datatype	Value	Description
source_type	string	*source_type*	Specifies the name of the source type which needs to be split. This type must be a dm_sysobject subtype, not the dm_sysobject itself. After the conversion, this type will become the lightweight type whose shared parent type is specified in the following parameter, shared_parent_type.
shared_parent_type	string	*shared_parent_type*	Defines the shareable type to use. If the specified type is not a shareable type, then it will be converted into one. The specified type needs to be a subtype of dm_sysobject and cannot be dm_sysobject. If the specified type is already a shareable type, then the lightweight_type parameter must be specified.
execution_mode	string	*execution_mode_split*	Specifies the operation to perform. The allowable values for the first form of the method are: • Split and Finalize • Split without Finalize • Finalize These values are described in detail in "Execution mode split values" on page 89.

Argument	Datatype	Value	Description
		execution_mode	Specifies the operation to perform. The allowable values for the second form of the method are: • Generate Script Only • Run and Finalize • Run without Finalize • Cancel • Finalize These values are described in detail in "Execution mode values" on page 89.
recovery_mode	boolean	TRUE|FALSE	Use this flag when the EXECUTION_MODE is either Run and Finalize, Run without Finalize, Split and Finalize, or Split without Finalize. The default is FALSE. If the flag is set to TRUE, then the internally-created interim types (and tables) will be dropped before the process begins. For Split and Finalize and Split without Finalize the shared parent will be reused but the interim child types (and tables) will be dropped.

Argument	Datatype	Value	Description
parent_attributes	string	*parent_properties*	Specifies properties in the source type which will be split into the parent type specified in shared_parent_type. The parameter contains a list of comma-separated property names from the source type. The remainder of the properties in the source_type will remain there.
lightweight_type	string	*lightweight_type*	If the specified type is a standard type, it will be converted to a lightweight type. The type needs to be a direct subtype of the one specified in shared_parent_type. If the specified type is already a lightweight type, then you must also set recovery_mode to TRUE, and specify which lightweight objects go with which parents. Use the parent_sql_predicate, parent_id, and default_parent_id arguments to specify lightweight objects and parents.

Argument	**Datatype**	**Value**	**Description**
parent_sql_predicate	string	*parent_sql_predicate*	This repeating string parameter specifies an SQL (not DQL), predicate qualifying a set of lightweight SysObjects whose parent will be set to the following parameter, parent_id. This parameter is allowed only when the execution_mode is either Run and Finalize or Finalize.
parent_id	ID	*ID*	This repeating ID parameter corresponds to the parent_sql_predicate parameter. The lightweight SysObjects that match the SQL predicate will be assigned to this parent. The specified ID needs to be a legal ID. This parameter is allowed only when the execution_mode is either Run and Finalize or Finalize.

Argument	**Datatype**	**Value**	**Description**
default_parent_id	ID	*ID*	This ID parameter is used for the remaining materialized lightweight SysObjects created by this method that are not qualified with any predicate specified in a parent_sql_predicate parameter. If this parameter is not specified, then the i_sharing_parent property of all the remaining lightweight SysObjects will be set to the ID of the lightweight object itself. That is, they will be considered as materialized. This parameter is allowed only when the execution_mode is either Run and Finalize or Finalize.
ft_lite_add	string	*property_list*	A comma separated list of the properties to add to the fulltext index for the lightweight type, or the string 'ALL' to add all the lightweight properties.
ft_base_add	string	*property_list*	A comma separated list of the properties to add to the fulltext index for the shareable parent type, or the string 'ALL' to add all the shareable properties.

8.1.3 Return value

MIGRATE_TO_LITE returns an object id for the text file containing the SQL script to do the migration. The script will be generated regardless of the execution_mode specified.

8.1.4 Execution mode split values

For the first form of the method, execution_mode can take on the following values:

- Split and Finalize—Generates the script and immediately runs it.
- Split without Finalize—Generates the script, but the original types are not changed. You can then validate the changes before making the changes permanent.
- Finalize—Makes the changes. For this mode, only the source_type, shared_parent_type, and execution_mode parameters are required.

When you use this command to split a type, the indexes on the type attributes are dropped. You will need to evaluate which attributes to index, and create those indexes, after splitting the type into a lightweight and a shareable type.

8.1.5 Execution mode values

For the second form of the method, execution_mode can take on the following values:

- Generate Script Only—Generates the script, but does not execute it.
- Run and Finalize—Generates the script and immediately runs it.
- Run without Finalize—Generates the script and immediately runs it, but the original types are not changed. All the changes will be applied to corresponding internally-created types (and tables) with the name dm_lw*_id_of_my_parent_type* and dm_lw*_id_of_my_child_type* created with the changes. You can then validate the changes before making the changes permanent.
- Cancel—Allows you to remove the interim types and tables created by the Run without Finalize mode. Typically you would do this if you decided not to finalize the changes. After removing the interim types and tables, the method ends.
- Finalize—Makes the changes permanent by replacing the original types with the internal types and dropped the internal types afterwards. If this option is provided when there is nothing to swap, a warning will be reported back to client and recorded in server log.

8.1.6 Permissions

You must have Superuser privileges, or be the owner of the involved types, to use this method. But also see the note for Oracle installation users following the description section.

8.1.7 Description

Use this method to migrate standard types and objects to shareable and lightweight types and objects. This method can also be used to reparent lightweight objects.

The first form of the command is used to convert a standard type to a lightweight and a shareable type. In this form of the command, the shareable type is formed by splitting off properties from the original standard type. The remaining properties are used to create the lightweight type. Specify the name of the standard type to split in the source_type parameter, the new shareable type in the shared_parent_type parameter, and list the properties to split off of the original type in the parent_attributes parameter. When you use this method to split standard objects, each lightweight object has its own private parent. In other words, each lightweight object is materialized. You can then reparent the lightweight objects by using the second form of the command, specifying parent_sql_predicates and parent_ids to point the lightweight objects to shareable parents.

The second form of the command does not move properties from one type to another. You will use it when all the non-inherited standard type properties will remain with the lightweight type. You can also use this form to reparent lightweight objects.

If you use the second form of the command and do not specify a lightweight type, and you specify a standard type in the shared_parent_type argument, the standard type is changed into a shareable type.

The second form can convert standard objects to lightweight objects and parent them with specified shareable parents. To do this, you specify a shared_parent_type and a lightweight_type. Conceptually, you can imagine that all the standard objects specified by the lightweight_type argument are converted to materialized lightweight objects pointing to private parents of the shared_parent_type. Next, all the objects of the lightweight type that match the parent_sql_predicate argument are made to point to the shareable parent with the associated parent_id. The method continues associating each group of lightweight objects that matches each subsequent parent_sql_predicate and parenting that group with the associated shareable parent. If any unconverted materialized lightweight objects remain after converting each parent_sql_predicate group, the remaining objects are parented with the default_parent_id shareable parent, if specified. Typically, you would create some shareable parents ahead of time, and then use this method to associate that list of objects with the groups defined by the parent_sql_predicate.

To use this method to reparent lightweight objects, specify the shared_parent_type and the lightweight_type, the parent_sql_predicate to select which objects to reparent and the parent_id to reparent to. You must also specify recovery_mode as

TRUE, or the method will not work with already converted objects of lightweight_type.

8.1.8 Fulltext support

Use the ft_lite_add and ft_base_add arguments to specify which properties are included in the fulltext index. These arguments add the shareable parent type properties and the lightweight type properties to fulltext indexing. The ft_lite_add variations add the properties specified for the lightweight type, and the ft_base_add variations add the properties specified for the shareable parent type.

Note: In order to use this method with an Oracle installation, the user account that Documentum CM Server uses to access the database must have enhanced privileges. The following example assumes that you have logged into SQLPLUS as the SYS user, in SYSDBA mode to grant these privileges to the repository_1 user:

```
grant DBA to repository_1;
```

These are powerful privileges and should be revoked when the migration is completed.

8.1.9 Examples

This example generates a script to convert usertype_1 to a shareable type. You do not execute this script against your database. It is created so you can examine the commands that will be issued when you later execute this method using another form of the method, either Run and Finalize or Run without Finalize:

```
EXECUTE  migrate_to_lite WITH "shared_parent_type"='usertype_1'
,"execution_mode"='Generate Script Only'
```

To get the script, you can issue the following DQL statement (assume that the result of the earlier method returned the object ID 0855706080007c19):

```
EXECUTE get_file_url FOR '0855706080007c19' WITH "format"='text'
```

This example converts usertype_2 to a shareable type in one operation:

```
EXECUTE  migrate_to_lite WITH "shared_parent_type"='usertype_2'
,"execution_mode"='Run and Finalize'
```

This example converts usertype_3 to a shareable type in two steps. After step one, you can examine the script and the temporary types and tables to verify the repository changes. For step one, use the Run without Finalize execution_mode to run the script and create the temporary tables:

```
EXECUTE  migrate_to_lite WITH "shared_parent_type"='usertype_3'
,"execution_mode"='Run without Finalize'
```

Then, for step two, use the Finalize execution_mode to swap in the temporary tables and commit the change:

```
EXECUTE  migrate_to_lite WITH "shared_parent_type"='usertype_3'
,"execution_mode"='Finalize'
```

This example converts an existing standard type, test_payment_check, and all its objects, into a lightweight type and a shareable type, test_payment_bank. Each lightweight object will have its own private parent object. The parent type will take the attributes bank_code and routing, but the other attributes will be part of the lightweight type.

In this example, there are already objects created of type test_payment_check. You can create the type using the following command:

```
CREATE TYPE "test_payment_check" (
 account integer,
 check_number integer,
 transaction_date date,
 amount float,
 bank_code integer,
 routing integer
 ) WITH SUPERTYPE "dm_document" PUBLISH
```

You can also use the batch example program to create a number of objects for this example.

Use the following command to split the test_payment_check objects into lightweight objects of type test_payment_check, and private parents of type test_payment_bank. The bank_code and routing attribute values will be associated with the private parent, and the other attributes will be associated with the lightweight objects. The attributes account and check_number of the (now lightweight) type test_payment_check will be fulltext indexed:

```
EXECUTE migrate_to_lite WITH "source_type"='test_payment_check'
,"shared_parent_type"='test_payment_bank'
,"execution_mode"='Split and Finalize'
,"parent_attributes"='bank_code,routing'
,"ft_lite_add"='account,check_number'
```

The following example takes the materialized lightweight objects converted by the last example and reparents them. Objects with check_numbers less than five are reparented to a separately created test_payment_bank shareable parent, and the remaining materialized objects are reparented to another separately created shareable parent:

```
EXECUTE "migrate_to_lite" WITH
"shared_parent_type"='test_payment_bank',"lightweight_type"='test_payment_check'
,"execution_mode"='Run and Finalize',"parent_sql_predicate"='check_number<5'
,"parent_id"='0925392d80001d5d'
,"default_parent_id"='0925392d80001d5e'
```

Chapter 9

Sample program for generating metadata for partition exchange

The information in this program is used as a step in the partition exchange example. It creates a shared parent object in the repository, then creates five text files that contain the object metadata to be ingested into the repository (all these objects are lightweight SysObjects that point to the shared parent object). The text files will subsequently be loaded into offline tables in the repository database. After the offline tables are loaded, they are exchanged for an online partition. After the partition exchange, the lightweight objects are in the repository, and can be manipulated similar to any other lightweight objects.

This program is modified from an earlier version to support the additional single valued attribute, i_other_contents, added to dmr_content in release 6.6.

Note: This program is provided as an example only. If you require assistance with writing programs for partitioning, contact OpenText Global Technical Services.

```
/*
 * loader.java
 *
 *
 * This example assumes that you have created two types, test_payment_bank
 * and test_payment_type. Here are the two DQL statements to create the types:
 CREATE SHAREABLE TYPE test_payment_bank (
 bank_code integer,
 routing integer,
 branch integer repeating
 ) WITH SUPERTYPE dm_sysobject PUBLISH

 CREATE LIGHTWEIGHT TYPE test_payment_check (
 account integer,
 check_number integer,
 transaction_date date,
 amount float,
 deductions float repeating
 ) SHARES test_payment_bank PUBLISH

 * This example also assumes that you have created an external store
 * filestore_ex and that there is a check image TIFF file in the
 * external store (use any TIFF to stand in for a check image).
 *
 * When you run this program, the repository name, username, password,
 * number of docs to create, and the partition number to place them in,
 * are passed to the program on the command line.
 *
 */
package com.documentum.docs.archive.examples;


import com.documentum.com.DfClientX;
import com.documentum.com.IDfClientX;
import com.documentum.fc.client.*;
import com.documentum.fc.common.*;
import java.io.FileWriter;
import java.io.PrintWriter;
import java.io.File;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.text.Format;
import java.util.Date;
import java.net.InetAddress;

public class loader
{
    // Login Info - will obtain from command-line args
    String docbaseName;  // args[0]  ("train5")
    String userName;     // args[1]  ("dmadmin")
    String password;     // args[2]  ("training")
    int numDocs;         // args[3]  (10000)
    int partition;       // args[4]  (150)

    Date start_time;
    Date end_time;

    //variables used for most programs
    IDfSessionManager sMgr;
    IDfSession session;
    IDfSysObject sysObj;

    IDfId parentObjId, childObjId, contentObjId;

    IDfCollection r_id_col;
    IDfCollection dmr_col;

   public void begin(String[] args)
   {

       try
       {
           start_time = new Date();
           end_time = new Date();

           start_time.setTime(System.currentTimeMillis());

           System.out.println("Started @ " +
                   DateFormat.getDateTimeInstance().format(start_time));

           parseCmdLine(args);

           // Logon
           sMgr = logOn(userName, password, docbaseName);
           session = sMgr.getSession(docbaseName);

           // Create Parent
           parentObjId = CreateParent(session);

           int amount = 0;
           int maxIds = 10000;  // max number of r_object_id's to retrieve at one time

           // open files
           PrintWriter dmiObjectType =
                   new PrintWriter(new FileWriter("dmiObjectType.txt"));
           PrintWriter testPaymentCheckS =
                   new PrintWriter(new FileWriter("testPaymentCheckS.txt"));
           PrintWriter testPaymentCheckR =
                   new PrintWriter(new FileWriter("testPaymentCheckR.txt"));
           PrintWriter dmrContentS =
                   new PrintWriter(new FileWriter("dmrContentS.txt"));
           PrintWriter dmrContentR =
                   new PrintWriter(new FileWriter("dmrContentR.txt"));

           int typeId = getTypeID(session, "test_payment_check");
```

```
            String fileStoreId = getFileStoreID(session, "filestore_ex");

            String formatTypeId = getFormatID(session, "tiff");

            InetAddress addr = InetAddress.getLocalHost();
            String hostname = addr.getHostName();

            Format formatter = new SimpleDateFormat("dd-MMM-yy");
            String set_time = formatter.format(start_time);
            for (int j=0; j<numDocs/maxIds; ++j)
            {
                 // get collection of r_object_id's for child objects - tag is always 08
                 r_id_col = getNewObjectIds(session, "08", String.valueOf(maxIds));
                 r_id_col.next();

                 // get collection of r_object_id's for content - tag is always 06
                 dmr_col = getNewObjectIds(session, "06", String.valueOf(maxIds));
                 dmr_col.next();

                // For this example, we are just using one image over and over again.
                // In a real application, you would specify which content file
                // goes with each child object.
                String filePath = "c:/ExternalStore/paycheck.tif";
                File file = new File(filePath);


                for(int i=0; i < maxIds; ++i)               {
                    // get child r_object_id
                    childObjId = r_id_col.getRepeatingId("next_id", i);

                    // record for DMI_OBJECT_TYPE (one record for each child)
                    dmiObjectType.println(childObjId.toString() + ", " + typeId +
                            ", " + partition);

                    // record for TEST_PAYMENT_CHECK_S (one record for each child)
                    if (i%25 == 0)
                        amount = i + j;

                    testPaymentCheckS.println(childObjId.toString() + ", " +
                            (1000000 + amount) + ", " + (i + j + 1) + ", " +
                            set_time + ", " + (amount + 25.00) + ", " + partition +
                            ", check_" + (i + j + 1) + ", " + parentObjId + ", 1, 0");

                    // record for TEST_PAYMENT_CHECK_R (one record for each
                    // repeating attribute)
                    testPaymentCheckR.println(childObjId.toString() + ", -1, " +
                            partition + ", 10.00");
                    testPaymentCheckR.println(childObjId.toString() + ", -2, " +
                            partition + ", 20.00");

                    // get child content r_object_id
                    contentObjId = dmr_col.getRepeatingId("next_id", i);

                    // record for DMR_CONTENT_S (one record for each child with
                    // content)
                    dmrContentS.println(contentObjId.toString() + ", 0, 1," +
                            fileStoreId + ", 0, 0, " + file.length() + ", tiff, "
                            + formatTypeId + ", 0, 0, 0, 0, \" \", 0, \" \", " +
                            hostname + ", " + filePath + ", " + set_time +
                            ", 0, \" \", \" \", 0, 0000000000000000, 0000000000000000, "
                            + file.length() + ", \" \", 0, 0, " + partition +
                            ", 0, 0");

                    // record for DMR_CONTENT_R (one record for each child with
                    // content)
                    dmrContentR.println(contentObjId.toString() + ", -1, " +
                            partition + ", "+ childObjId.toString() +
                            ", 0, , , , , " + formatTypeId +
                            ", 0, 0, 0, 0, , tiff, , , , , , , , , , , ");
                }
                r_id_col.close();
                dmr_col.close();
             }

            dmiObjectType.close();
            testPaymentCheckS.close();
            testPaymentCheckR.close();
            dmrContentS.close();
            dmrContentR.close();
        }
        catch(Throwable e)
        {
            if(e instanceof DfException)
            {
                System.out.println("DFC Exception:");
                String s = ((DfException)e).getStackTraceAsString();
                System.out.println(s);
            }
            else
            {
                System.out.println("Non-DFC Exception");
                e.printStackTrace();
            }
        }

        finally
        {
            if (session != null)
                sMgr.release(session);

            end_time.setTime(System.currentTimeMillis());
            System.out.println("Finished @ " +
                    DateFormat.getDateTimeInstance().format(end_time));
            System.out.println("Elapsed " +
                    CalculateTimeDiff(start_time, end_time) + " sec");

        }
    }

     private static IDfSessionManager logOn(String userName, String password,
             String docbaseName) throws Throwable
     {
         //create a Client object
         IDfClientX clientx = new DfClientX();
         IDfClient client = clientx.getLocalClient();

         //create a Session Manager object
         IDfSessionManager sMgr = client.newSessionManager();

         //create an IDfLoginInfo object named “BatchTest”
         IDfLoginInfo loginInfoObj = clientx.getLoginInfo();
         loginInfoObj.setUser(userName);
         loginInfoObj.setPassword(password);

         //bind the Session Manager to the login info
         sMgr.setIdentity(docbaseName, loginInfoObj);
         return sMgr;
     }

    private static IDfId CreateParent(IDfSession session) throws Throwable
    {
        IDfSysObject sysObj = (IDfSysObject)session.newObject("test_payment_bank");
        sysObj.setObjectName("check_parent");

        // Content type and storage must be set on parent, else lightweight
        // object is materalized
        sysObj.setStorageType("filestore_ex");
        sysObj.setContentType("tiff");

        // set shared attributes
        sysObj.setInt("bank_code", 1);
        sysObj.setInt("routing", 231381116);
        sysObj.setRepeatingInt("branch",0,1);
        sysObj.setRepeatingInt("branch",1,2);

        sysObj.save();

        return sysObj.getObjectId();
    }

    // This apply method will reserve a list of object IDs from the repository.
    // After the reservation, the IDs will not be used by the repository, so you may want
    // to set the number of reserved IDs large enough to cut down on the ID requests,
    // but not so large as to use up IDs if the process fails, or there are leftover IDs
    // when the objects are created. The equivalent API command is:
    // apply,c,NULL,NEXT_ID_LIST,TAG,I,08,HOW_MANY,I,100
    private static IDfCollection getNewObjectIds(IDfSession session, String tag,
            String number) throws Throwable
    {
        IDfList args = new DfList();
        IDfList types = new DfList();
        IDfList values = new DfList();

        args.appendString("TAG");
        types.appendString("I");
        values.appendString(tag);

        args.appendString("HOW_MANY");
        types.appendString("I");
        values.appendString(number);

        return session.apply("NULL", "NEXT_ID_LIST", args, types, values);
    }

    // select i_type from dm_type_s where name = 'test_payment_check'
    private int getTypeID(IDfSession session, String type) throws Throwable
    {
        IDfClientX clientx = new DfClientX();
        IDfQuery q = clientx.getQuery(); //Create query object
        q.setDQL("select i_type from dm_type_s where name = '" +
                type + "'"); //Give it the query

        IDfCollection col = q.execute(session, IDfQuery.DF_READ_QUERY);
        col.next();

        int result = col.getInt("i_type");

        col.close();

        return  result;
    }

    // select r_object_id from dm_store_s where name = 'filestore_ex'
    private String getFileStoreID(IDfSession session, String filestore)
    throws Throwable
    {
        IDfClientX clientx = new DfClientX();
        IDfQuery q = clientx.getQuery(); //Create query object
        q.setDQL("select r_object_id from dm_store_s where name = '" +
                filestore + "'"); //Give it the query

        IDfCollection col = q.execute(session, IDfQuery.DF_READ_QUERY);
        col.next();

        IDfId result = col.getId("r_object_id");

        col.close();

        return  result.toString();
    }

     // select r_object_id from dm_format_s where name = 'tiff'
     private String getFormatID(IDfSession session, String format) throws Throwable
     {
         IDfClientX clientx = new DfClientX();
         IDfQuery q = clientx.getQuery(); //Create query object
         q.setDQL("select r_object_id from dm_format_s where name = '" +
                 format + "'"); //Give it the query

         IDfCollection col = q.execute(session, IDfQuery.DF_READ_QUERY);
         col.next();

         IDfId result = col.getId("r_object_id");

         col.close();

         return  result.toString();
     }

    private static long CalculateTimeDiff(Date start_time, Date end_time)
    {
       long time_value;
       Date diff_time = new Date();

       diff_time.setTime(end_time.getTime() - start_time.getTime());
       time_value = (end_time.getTime() - start_time.getTime()) / 1000;
       return time_value;
    }
     /**
      * void parseCmdLine(String[] args)
      *
      * Copies args[] variables into friendlier-named variables
      */
     void parseCmdLine(String[] args)
     {
         if(args.length > 0)
         {
             for(int i = 0; i < args.length; ++i)
             {
                 switch(i)
                 {
                     case DOCBASE:   // 0
                         docbaseName = args[i];
                         break;
                     case USERNAME:  // 1
                         userName = args[i];
                         break;
                     case PASSWORD:  // 2
                         password = args[i];
                         break;
                     case NUMDOCS:  // 3
                         numDocs = Integer.parseInt(args[i]);
                         break;
                     case PARTITION:  // 4
                         partition = Integer.parseInt(args[i]);
                         break;
                 }
             }
         }
         else
         {
             help();
             System.exit(1); // you never return from this method
         }
     } //end: parseCmdLine(String[] args)
     public static void main(String[] args)
     {
         new loader().begin(args);
     }
     private static void help()
     {
         String[] helpList =
         {
             "You must set arguments before executing this program.",
             "Usage: java loader docbase user pwd numDocs partition",
             "",
             "Where: ",
             "      docbase      Docbase name",
             "      user         user name for the Docbase",
             "      pwd          password",
             "      numDocs      number of Docs to load",
             "      partition    partition number to load into",
             "",
             "Example from command line:",
             "",
             "   java loader train5 tuser1 password 10000 150",
         };
         for(int i = 0; i < helpList.length; ++i)
         {
             System.out.println(helpList[i]);
         }
     } //end: help()
     static final int DOCBASE       =0;
     static final int USERNAME      =1;
     static final int PASSWORD      =2;
     static final int NUMDOCS       =3;
     static final int PARTITION     =4;
 }
```