Documentum Application Performance and Tuning

Version 2

May 03, 2002

Author: Ed Bueché
Principal Engineer Capacity Planning and Performance
bueche@documentum.com

documentum

TABLE OF CONTENTS

Audience

This paper is meant for Documentum Application developers to aid in their design efforts and tuning strategies. The reader is assumed to have some basic knowledge of the following aspects of Documentum technology:

- Coding with the Documentum Foundation Classes (DFC) in either Java, Visual Basic, dmbasic, and/or C/C++.
- A basic understanding of the various components of the Documentum architecture including:
    - The eContent Server and its services,
    - The Desktop Client services,
    - The RightSite server,
    - The Web Development Kit (WDK) environment, and
    - The lower level dmcl calls and environment

Some important background documents include:

- Documentum eContent Server Fundamentals, DOC3-SVRFUNDUX-301 or DOC3-SVRFUNDWIN-701
- Documentum eContent Server API Reference Manual. DOC3-SVRAPIUX-301 or DOC3-SVRAPINT-JL00
- Using DFC in Documentum Applications, DOC3-USINGDFC-301
- Documentum Web Development Kit Tutorial, DOC3-WDKTUT-401
- RightSite Application Developer's Guide, DOC3-D3WAD-0798
- Migrating Documentum Desktop Client Applications, DOC3-DWMDC-4000

Accessing Objects Efficiently

The following section covers topics relating to the efficient access of Documentum objects. The Documentum eContent Server uses a flexible and extensible object-oriented data model for content and its associated meta-data. The power behind the data model lies not only in the flexible and extensible representation of an object and its attributes, but also in the ability to easily provide that information to the user. However, the usefulness of the data is impaired if retrieval is slow. Users do not appreciate sluggish interface response time. Hence, it serves developers well to pay attention to the cost of retrieving attribute information from the Docbase. This section covers some important aspects of retrieval including:

- Fetching an object vs. querying for some of its attributes
- DMCL cache sizing
- Data dictionary access for object type meta-data
- User name caching, and
- Creating complex objects efficiently.

Fetching and Querying for Objects

There are two main methods for retrieving attributes of an object from the Docbase: fetching the entire object and querying for a subset of its attributes. First, lets cover some basics relating to fetching an object. There are three main caches involved with Docbase objects. These are summarized in the following table.

Cache Name	Holds	Resides on	Is initialized when	Replacement Strategy	Size Controlled by
Dmcl client cache	Objects (attributes)	Client machine (or App server machine in case of Web)	An object is Fetched	LRU for non-modified objects	Dmcl.ini -
Server object cache	Objects (attributes)	EContent server machine	an object is fetched	LRU for non-modified objects	Server_config object
RDBMS buffer cache	RDBMS tables that make up an object	RDBMS machine	A query gets the rows of an object.	Typically LRU	RDBMS-specific initialization file

TABLE 1: Summary of Some Object caches

When an object is “fetched” all of its attributes are sent from the eContent server to the client and cached on the client. Once there its attributes can be accessed quickly. The object can also be manipulated once it is fetched (attribute values changed, and then “saved” to the Docbase, or the object can be locked through a checkout). The main method of causing an object to be fetched is through IDfSession::GetObject.

```
obj = (IDfSysObject ) session.getObject( objId );
objName = obj.getObjectName( );
contentType = obj.getContentType ();
```

Figure 1: Code Sample with GetObject

The important point to remember about an object fetch is that all of its attributes are retrieved. Therefore subsequent attribute access is inexpensive, but there is an initial performance penalty for loading all of the attributes. The following dmcl trace of the above DFC illustrates this behavior:

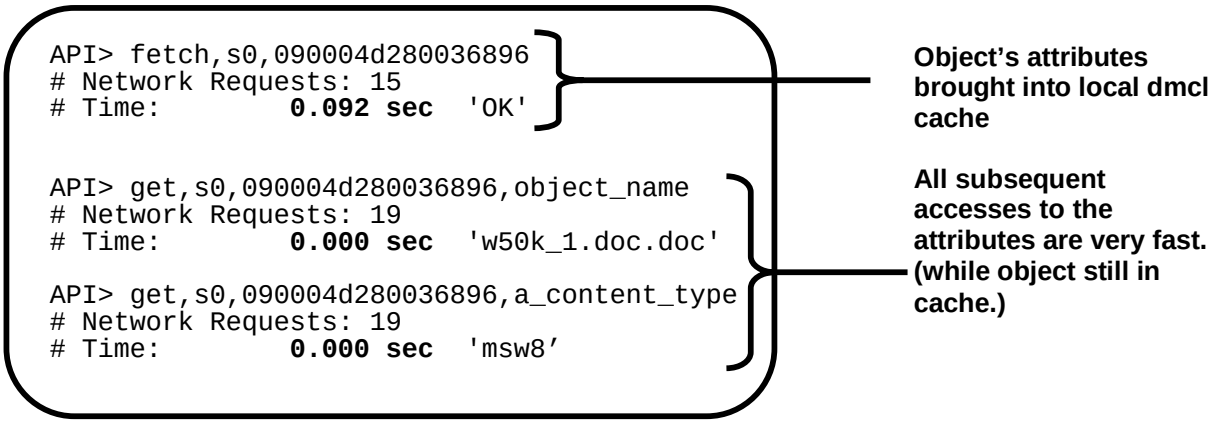


Figure 2: DMCL trace excerpt showing benefits of Object Fetch and caching

Note: DMCL tracing is described in more detail in Appendix A.

After an object has been fetched into the dmcl cache it will need to be fetched again to the client machine for the following reasons:

- The object is flushed from the dmcl cache (number of objects in cache is exceeded)
- The object's version stamp changes due to another user updating it
- It is requested by a different client session started on the same machine

We cover dmcl cache thrashing and version stamp changes in later sections. The important point about multiple sessions on the same client machine is that each session will have its own cache. If an object 'A' is fetched by one session it still has to be brought over again if fetched by a different session on the same machine.

The concern on the penalty of an object fetch is due to the fact that all of the object's attributes are sent to the requesting client and a network round-trip is needed for each object fetched. This is illustrated below. In this example, even if only the object's name and content type are later accessed, the GetObject call (which maps down to a dmcl fetch call) will have already brought over all of the object's attributes.

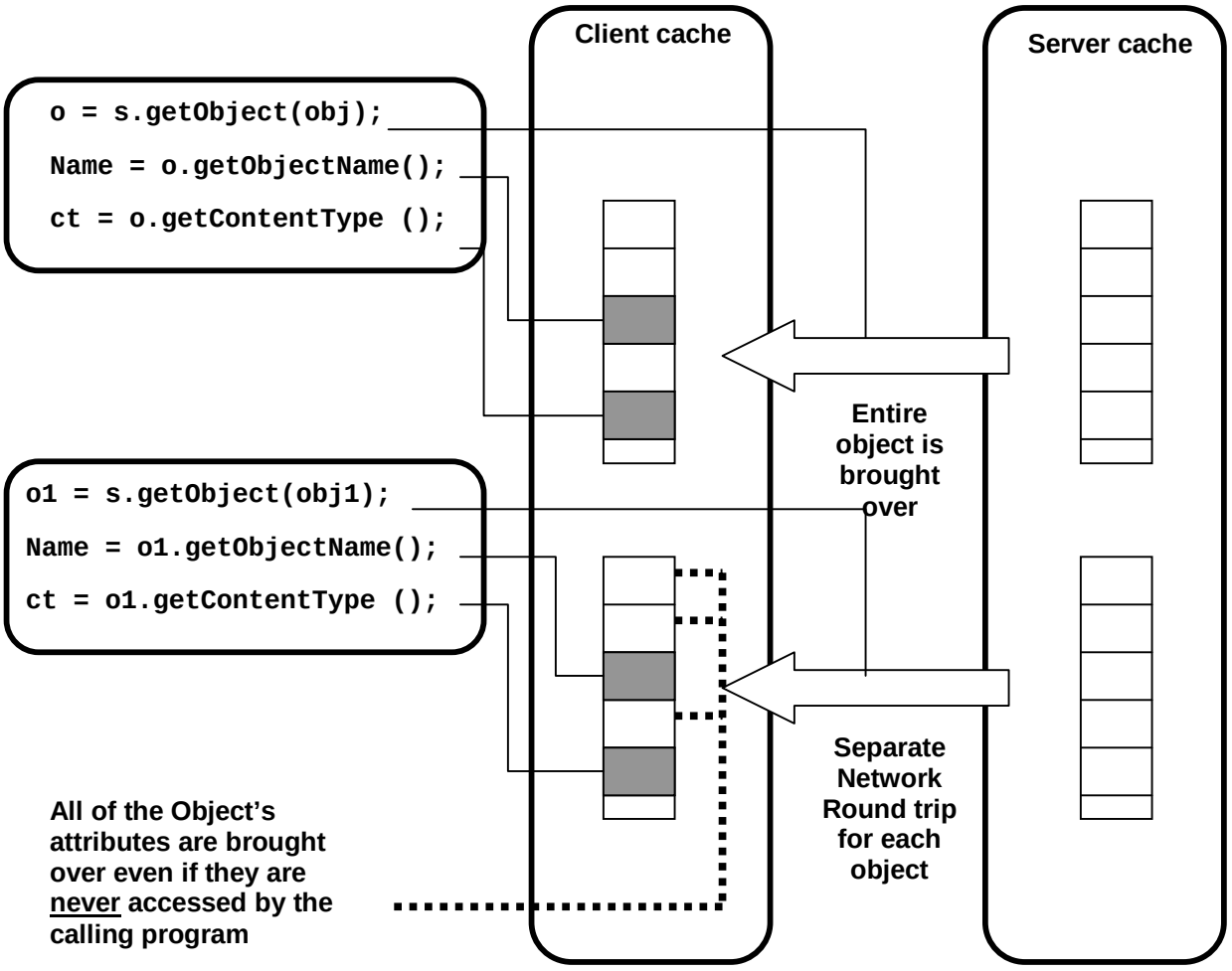


Figure 3: All Attributes & Separate Round Trip for Each fetch

Given this penalty object fetch (or DFC GetObject()) is not recommended under at least the following conditions:

- A set of objects need to be displayed (list),
- A small number of attributes need to be displayed, or
- The access to the attribute values of the object is localized to certain modules and not spread all throughout your program randomly.

The first point (avoid fetch in object display list) is extremely important. This is especially true when displaying a list of objects. This list could be, for example, the contents of a folder or the results of a search. A more efficient way to build an object list is to use the DfQuery interface. A DfQuery will issue a DQL query for the exact attributes needed for a set of objects and have them returned as a collection. An example is shown below.

```
DfQuery q0 = new DfQuery();
q0.setDQL("select r_object_id, object_name, a_content_type ….from
                                          where….");
IDfCollection c0 = q0.execute( session, q0.READ_QUERY );
while ( c0.next() ) {
       objId = c0.getId( "r_object_id" );
      objName = c0.getString( "object_name" );
       contentType = c0.getString( "a_content_type" );
       :
}
c0.close();
```

Figure 4: DfQuery example

The underlying architecture of the DfQuery mechanism is illustrated below.

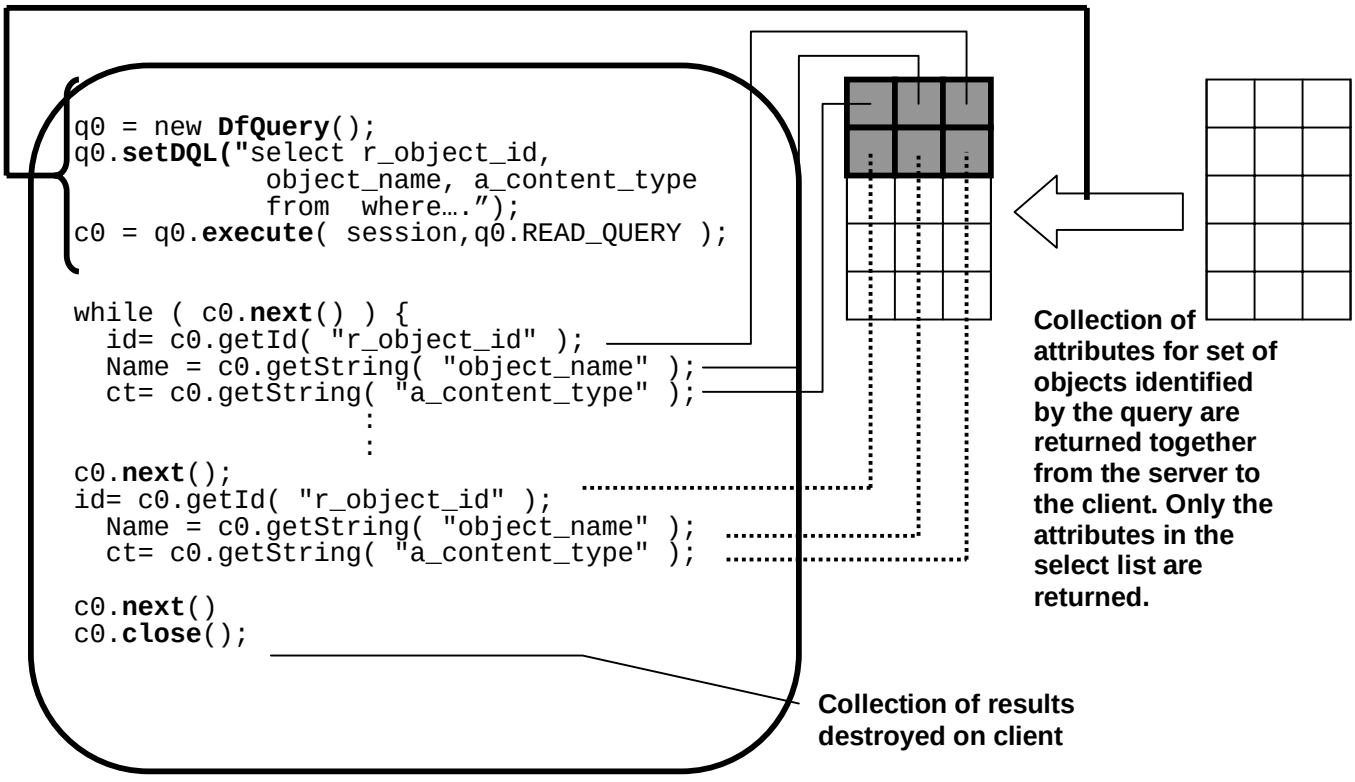


Figure 5: DfQuery Mapping of code to underlying operations

The benefits of using the query interface (illustrated below) are that:

- Only the needed attribute values are returned, lowering the network bandwidth needs,
- Fewer network round trips are required (attribute values packed in network packets) helping reduce the penalty of network latency, and
- Data is not cached in the dmcl or server cache (less work required to provide information to the user)

However, once the collection is closed then the query would need to be re-issued to get the results again.

A very common performance problem is to mix these two methods in a fashion that gives terrible performance (see below).

```
DfQuery q0 = new DfQuery();
q0.setDQL("select r_object_id, object_name, a_content_type
….from where….”);
IDfCollection c0 = q0.execute( session, q0.READ_QUERY );
while ( c0.next() ) {
      objId = c0.getId( "r_object_id" );
      obj = (IDfSysObject ) session.getObject( objId );
      objName = obj.getObjectName( );
      contentType = obj.getContentType ();
```

Figure 6: Example Code of Query/Fetch performance problem

The performance difference between the above method (Query/Fetch) and a “pure” Query/Collection list display (illustrated ealier in Figure 4) is shown below. In this test progressively larger numbers of objects are displayed (10 different folders had been populated with progressively larger numbers of objects). The eContent server resided on a dual processor, 800 Mhz Pentium III HP Lp1000r. In this example ten attributes of the dm_document objects are retrieved.

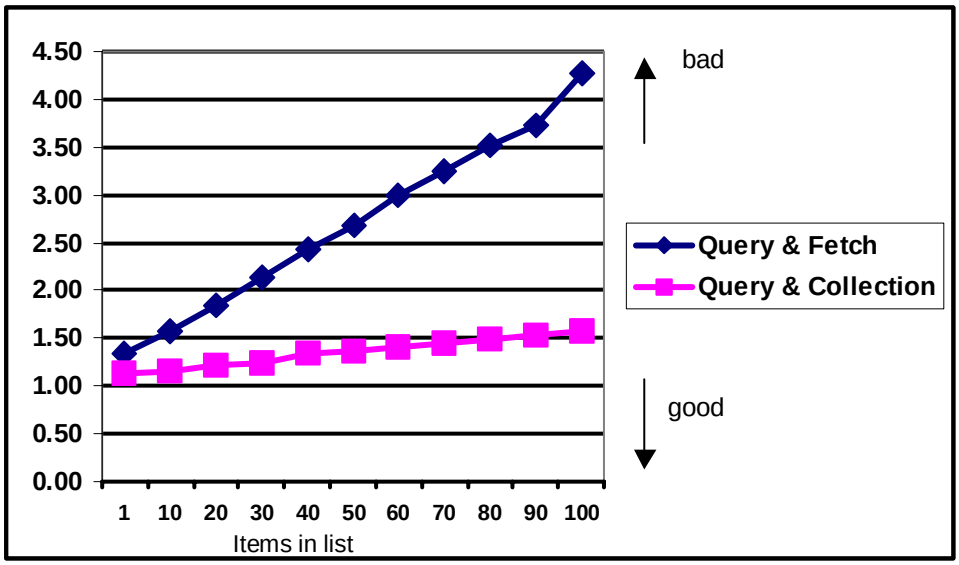


Figure 7: Performance Comparison of the two query methods

The above comparison is based on a 100M bps network infrastructure; the graph below shows how the two implementations fare on 56K bps bandwidth with 100 msecs one-way latency. In this graph we project the response time based on the number of messages exchanged and the amount of bytes sent each way. Actual response times are likely to be worse. The important point is that at 100 objects displayed the Query/Fetch strategy generates 20 times the number of round-trip network messages and almost 10 times the amount of bytes. This growth in network demand has a profound effect on response time, even with a small amount of objects.

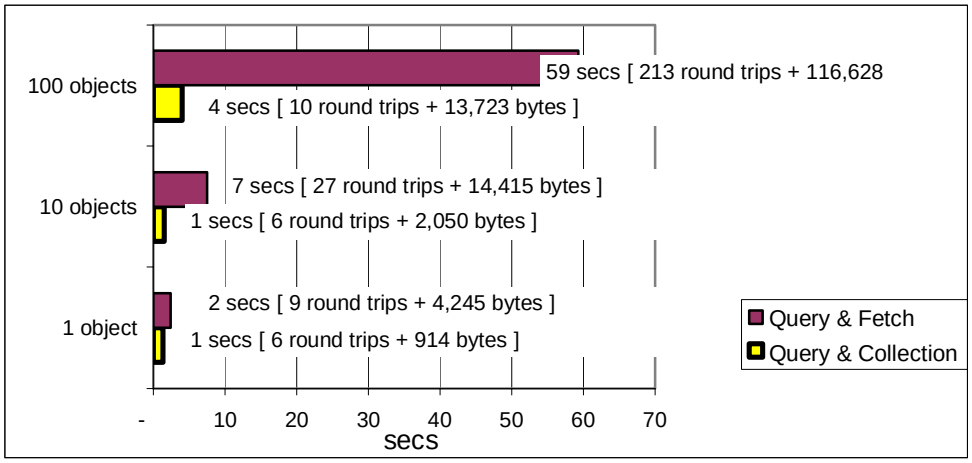


Figure 8: Impact of 56K bps and 100 msecs one-way latency on response time

Why does the combination of these two access methods perform so poorly? This is illustrated in Figure 9. The query will bring over the collection of attributes, however; only the object id is used. The other attributes in the collection are ignored. Then as each fetch occurs all of the object's attributes are returned costing a network round trip and lots of bandwidth since all of the attributes are sent. In this example, the collection is retrieved and then each object is fetched individually. All of the benefits of the query/collection have been lost and the user pays a terrible price in their response time.

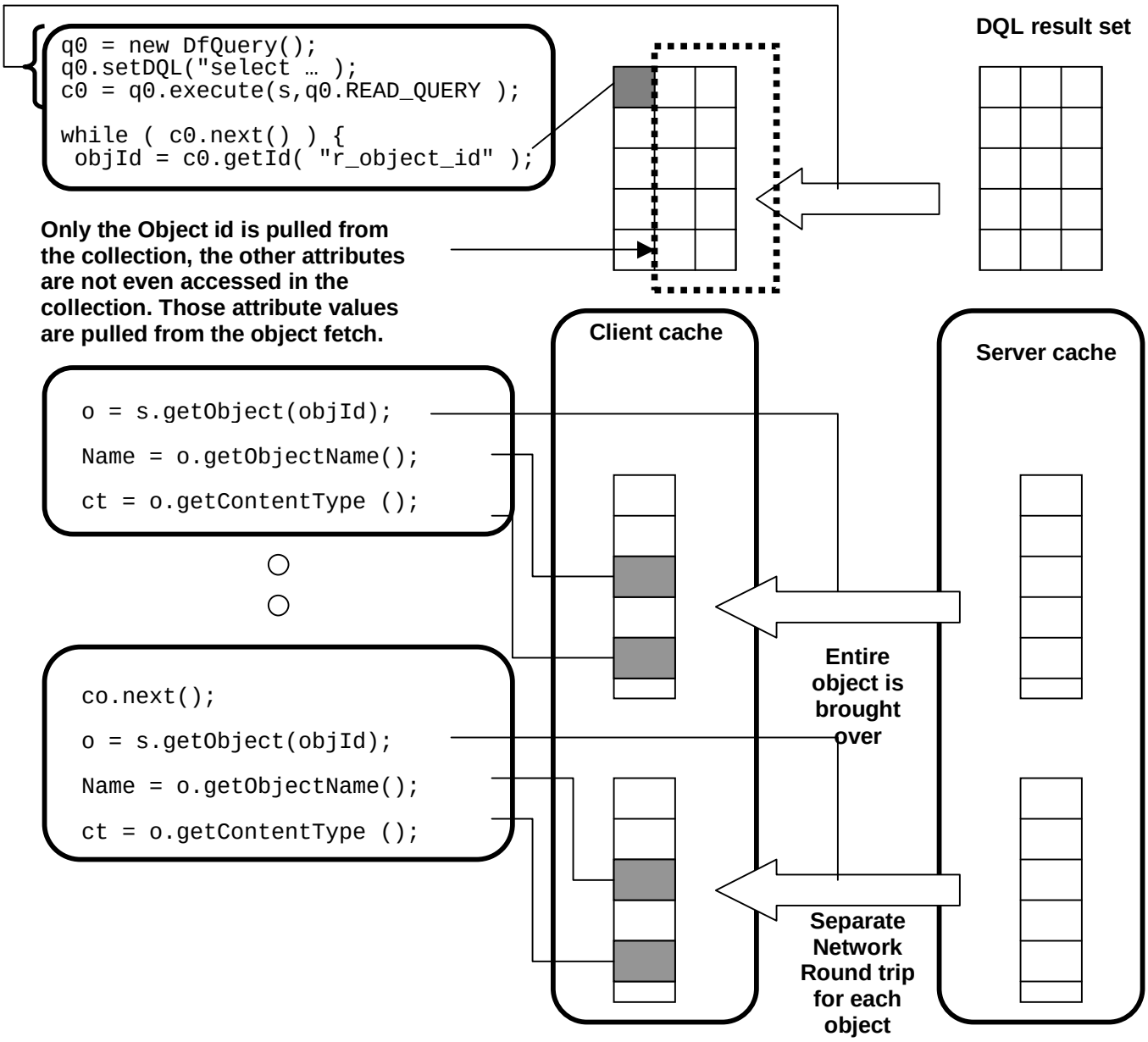


Figure 10: Query/Fetch under the covers

Since this is such an often-occurring coding problem, lets examine how one can spot it when looking at the dmcl trace. The following example illustrates the inefficient technique. Note how the trace shows that after getting the object id from the collection, the subsequent commands fetch the object or even start to query for the attributes directly from the object rather than through the collection.

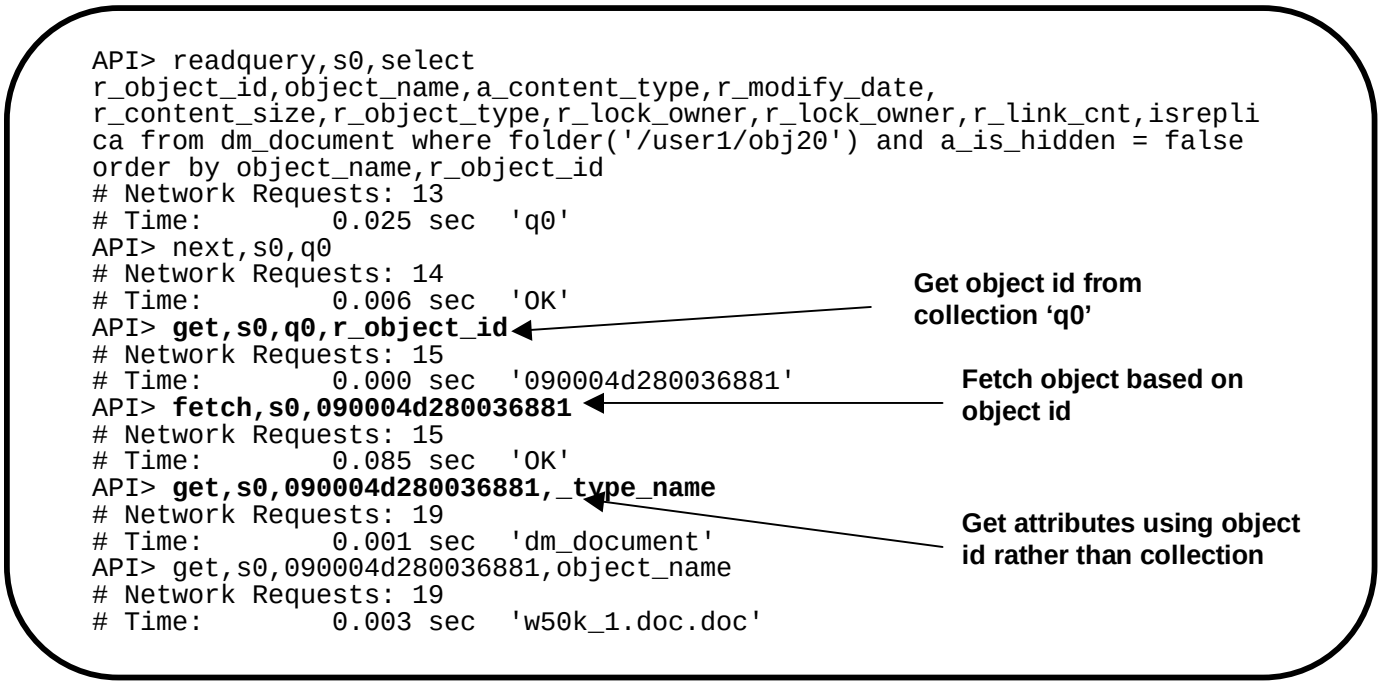


Figure 11: Illustration of the Query/Fetch problem from the dmcl trace level

Underlying eContent Server Caches: the Client DMCL cache

As mentioned earlier one of the big advantages of a Fetch call is that the object's attributes are cached in the local client DMCL cache. In this section we cover and illustrate some performance aspects of this cache. First, a Fetch of an object represents "optimistic locking" of the object by the Documentum eContent Server. That is, the fetch does not hold locks after retrieving the attributes. If an object's attributes are locally modified after this Fetch, then the subsequent "save" operation could fail if the object is already locked or has been changed since the object was brought into the DMCL cache. The typical failure is that there is an object version mismatch. This scenario is illustrated below.

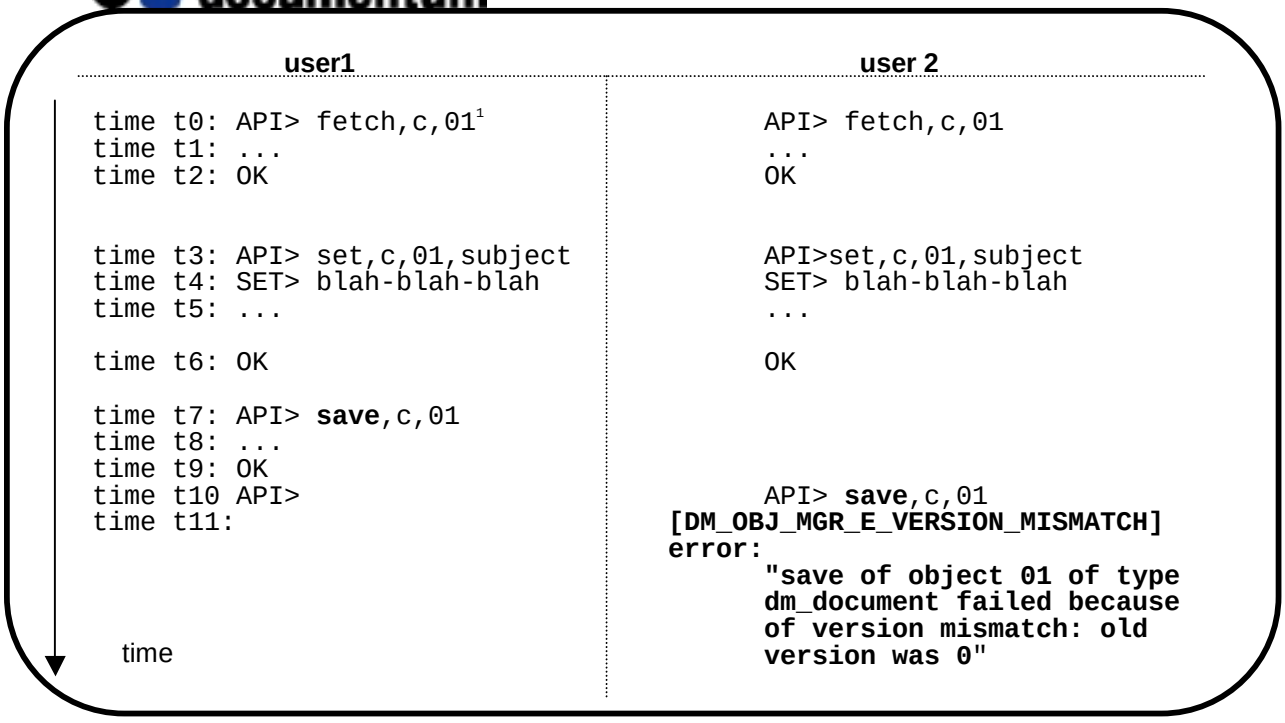


Figure 12: Object Version Error Illustrated

When an object is checked out, it is fetched and locked. This represents pessimistic locking and ensures that the "version" error cannot occur. However, the application developer still needs to handle the "object already locked" error shown below.

```
API> checkout,c,01
...
[DM_API_E_BADOBJ]error:  "Command checkout returned an invalid
document/object id."

[DM_SYSOBJECT_E_CANT_LOCK]error:  "Cannot lock doc01.doc sysobject,
 locked by user3."
```

Figure 13: Lock error illustrated

Next, as mentioned earlier the default size of the client dmcl cache is 100 objects. The size of this cache can have a profound impact on an application's performance. The biggest penalty occurs when the "object working set" (most frequently accessed objects) of an application is far larger than the cache itself. For example, suppose we change the size of the cache to 10 objects (with following entry in dmcl.ini file):

```
[DMAPI_CONFIGURATION]
client_cache_size = 10
```

Notice, then that after the 11th object is fetched, the first object is deleted from the cache to make room for the 11th object. This is "transparent" from a programming perspective. The next access to the 1st object succeeds, but only after the object is re-fetched (and the 2nd object is deleted from the cache to make room for the 1st one again). Hence, the response time increases.

[1] I've abbreviated the object id's in this example to ('01', '02', …, '11') in an effort to make the example more readable.

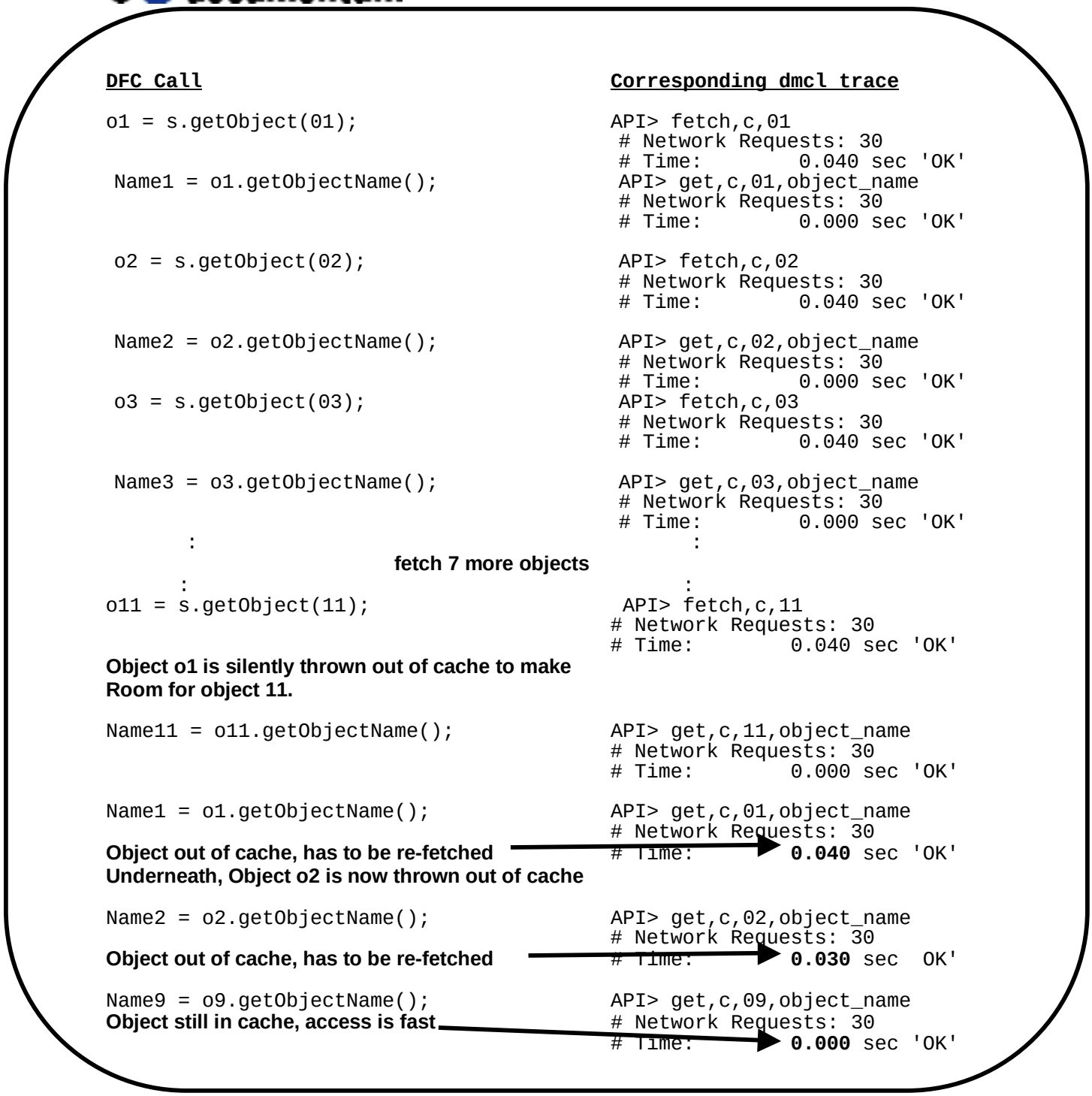


Figure 14: Example of Cache size Impact (DFC mapped to dmcl trace)

The correct sizing of the dmcl can have a fairly dramatic impact on XML and VDM-based applications that involve a large number of sysobjects. An XML document could be chunked down to a fairly small level (for example, at the paragraph level). For a large document this can result in the creation of hundreds or thousands of objects. The performance of an XML checkout/checkin could experience dramatic penalties as the various XML chunks are thrashed in and out of the dmcl cache.

Data Dictionary Access

If not efficiently accessed the eContent server data dictionary can lead to less-than desirable performance. The typical problem is a variant of the Fetch vs. Query/collection problem covered earlier. In many cases there is a need to access data dictionary information for a list of attributes. The 4.1 eContent server provided some apply methods that provided better access to result sets. These APIs were “undocumented” and will be phased out later, replaced by the 4.2 eContent server’s enhancements to the data dictionary. Lets cover these in more detail.

First, suppose we wanted to access the “is_searchable” property for the attributes of dm_document. Prior to the 4.1 eContent server this had to be done as shown below. As we will see later, it performs poorly.

```
IDfTypedObject theTypedObject;
IDfType theType =  s.getType( "dm_document" ) ;
int nNumAttr = theType.getTypeAttrCount();
for ( i = 0; i < nNumAttr; i++ )
    {
    String Name = theType.getTypeAttrNameAt( i );
     theTypedObject = s.getTypeDescription(     "dm_document",
                                                Name, id, ""  );

     bIsSearchable = theTypedObject.getBoolean( "is_searchable" );
     if ( bIsSearchable )
      {
        System.out.println("attribute is searchable" );
      }
      else
      {
        System.out.println ("attribute is NOT searchable" );
      }
 }
```

FIGURE 15: Getting is_searchable using DFC type operations

Now the poorly performing technique happens often enough to warrant an example dmcl trace. It might be hard to find in the code, but the behavior is much easier to spot in the trace of a poorly performing operation. The trace excerpt below comes from api executed against an 800 Mhz Hp lp1000r. Server machines that have only 400Mhz processors would fare much worse than 66 msecs per “tdm_” call.

```
API> get,s0,tdm_document.object_name,is_searchable
# Network Requests: 18
# Time:        0.066 sec  'T'
API> get,s0,030004d280000127,attr_name[1]
# Network Requests: 20
# Time:        0.000 sec  'r_object_type'
API> type,s0,dm_document,r_object_type
# Network Requests: 20
# Time:        0.000 sec  'tdm_document.r_object_type'
API> get,s0,tdm_document.r_object_type,is_searchable
# Network Requests: 20
# Time:        0.033 sec  'T'
```

Figure 16: Underlying dmcl trace for slow data dictionary access

In the 4.1 eContent server a faster method could be employed. Special “apply” methods were developed to obtain the needed data using a result set rather than a tdm_ call per attribute.

```
IDfList l1 = new DfList();
IDfList l2 = new DfList();
IDfList l3 = new DfList();
IDfCollection collection;
IDfTypedObject pDfTypedObject
String Name;
l1.append("TYPE_NAME");
l2.append("s");
l3.append("dm_document");
collection = s.apply(   "0000000000000000",
                        "GET_ATTRIBUTE_DD_INFO", l1,l2,l3   );
while ( collection.next() )
  {
   IDfTypedObject pDfTypedObject = collection.getTypedObject();
   int nNumAttr = collection.getValueCount("attr_name");
   for ( i = 0; i < nNumAttr; i++ )
     {
      Name = pDfTypedObject.getRepeatingString("attr_name", i);

      if (pDfTypedObject.getRepeatingBoolean("is_searchable", i))
      {
          System.out.println("attribute is searchable " );
      }
      else
      {
          System.out.println("attribute is NOT searchable");
      }
   }
}   //while ( collection.next() )
collection.close();
```

FIGURE 17: Getting is_searchable using 4.1 eContent Server apply method

The apply methods are very restrictive and will become obsolete in future releases of the eContent server. They have been replaced by a new set of Data Dictionary objects made available for direct query in the 4.2 version of the eContent server (see below).

```
DfQuery q0 = new DfQuery();
q0.setDQL("select attr_name,  is_searchable "
            + "from dmi_dd_attr_info where type_name = 'dm_document'“ +
                  “and nls_key = 'en'" );
IDfCollection c0 = q0.execute( session, q0.READ_QUERY );
while ( c0.next() )
   {
    String strAttrName = c0.getString( "attr_name" );
    bIsSearchable= c0.getBoolean( "is_searchable" );
    if (bIsSearchable) {
       System.out.println ( "attribute is searchable " );
    } else {
       System.out.println ( "attribute is NOT searchable ");
    }
}   //while ( collection.next() )
c0.close();
```

FIGURE 18: Getting is_searchable using 4.2 eContent Server Data Dictionary Object

The performance difference between these three methods is illustrated below. Again the machine running the eContent server was an 800 MHz dual CPU HP Lp1000r. There would be a larger penalty for the DFC type methods on systems with less MHz.

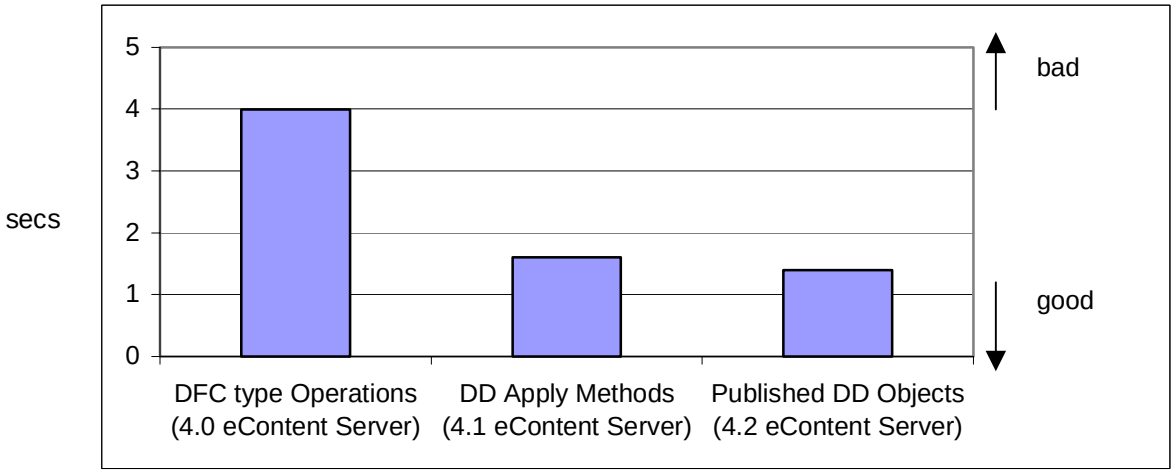


Figure 19: Response time of alternate data Dictionary Access to is_searchable property

The important point about the data dictionary access is that you can improve your performance dramatically by querying directly on the published data dictionary objects rather than just relying on the DFC type operations.

The data dictionary represents an object that is not expected to change often in production applications. Hence, the information retrieved is a good candidate for persistent caching on the client side. The Documentum Desktop Client provides a data dictionary cache interface called IDcDataDictionaryUtils that is documented in Appendix B. This does not, however, cover all of the items that can be retrieved from the data dictionary (e.g., is_searchable). Hence, it is likely that developers will want to construct their own persistent data dictionary caches in an effort to speed up access to their own favorite attributes from data dictionary objects. When setting up such a cache its important to be able to check consistency of the information in the cache. This is done by checking the *dd_change_count* in the *dmi_change_record* object. This count is incremented each time there is a modification to the data dictionary.

User Name Caching

There are many common dialogs' that allow one to select from a list of users. In enterprises with thousands of users (or less depending on the network infrastructure) the response time of displaying these dialogs will be poor if the user name list is downloaded each time the dialog is invoked. In this section we work through some caching techniques for the user name list that can be used in a persistent cache on the Desktop client side or a process-cache within an Application server.

It is easy enough to download all of the user names from the dm_user objects and create a cache. The most important question is: how do I check to make sure this cache is up-to-date? The key to the user-name-cache-coherency problem is the r_modify_date for the dm_user object, which indicates the date of the last change to that particular dm_user object.

Strategy #1: Simple Coherency

In the first example, we associate a single date for our entire user name cache and refresh the entire structure once any modification occurs on any of the dm_user objects. A query to check this would look as follows:

```
select max(r_modify_date) from dm_user
```

The following example illustrates how this could be done.

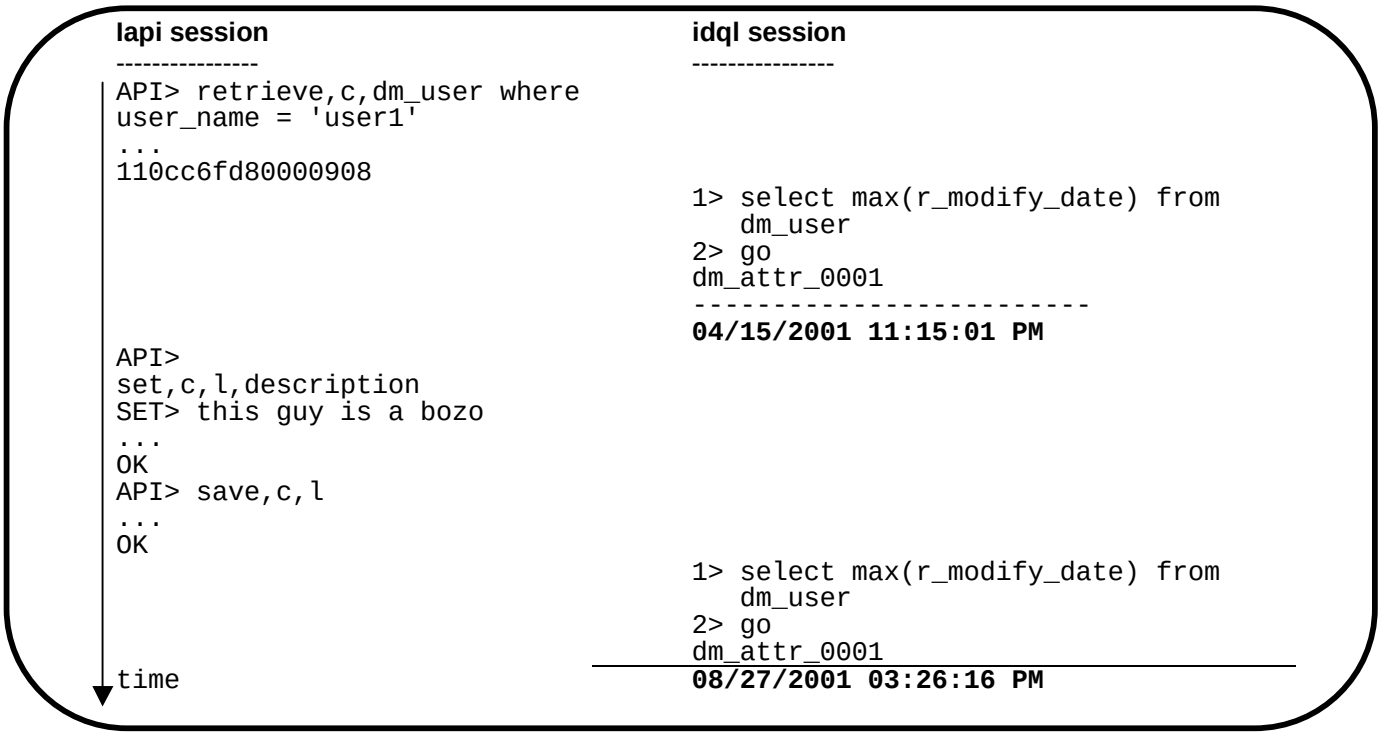


FIGURE 20: Illustration of caching user names using single global modify timestamp

The downside of this strategy is that if a single dm_user object changes out of 10,000 (for example) then the entire set of dm_user objects will be downloaded to the client. How often would this happen? An examination of some corporate Docbases indicated that although there were 20 to 30 changed dm_user objects in each month, the changes are typically clustered around less than 6 to 8 days in that month.

Strategy #2: Tracking changes to individual user names

To improve upon this we would have to cache not only the user name (and user_state), but its r_modify_date as well, as oppose to just saving the latest r_modify_date. Then the query to verify consistency would return all of the entries that have changed since the last time checked. This would be a far better strategy to employ when working with thousands of users and/or environments with low bandwidth and high latency.

Now one weakness of this strategy is that it is possible that a dm_user object was just deleted. In practice, Administrators are discouraged from deleting dm_user objects. In most cases the users are marked as invalid (i.e., 'can't log in': user_state = '1'). The worst thing that could happen if they are not "deleted" in this fashion is that a client cache could have the name of a user that no longer exists in the docbase. An attempt to include that user in some operation will fail (with an invalid user error).

However, if the administrators insist on deleting users then the full download of the user names could be restricted to when the following occurs:

- Some of the latest entries are new to the cache, and
- `select count(*) from dm_user` ≤ (old cache size + count of new entries)

Template-based Object Create Caching

Many customized Documentum applications create documents based on templates and utilize lifecycle and other eContent server services (like Virtual documents and/or relations). Unlike many tasks that are expensive, this complex-document-creation scenario has to be synchronous: the user selects the format, template, and service (e.g., life cycle) and presses the OK button and cannot continue work until the document is created. The expectation is that the complex object will be created and initialized quickly. Achieving this expected response time can be difficult depending on the object complexity.

One technique that has proven to be helpful for this is to create a cache of pre-baked objects (an object bag). The idea behind this is to create objects that are almost completely built up ahead of time so that when a user creates an object only a few quick operations need to be applied (like linking it to the user's folder and renaming it).

The maintenance of this cache and its properties are a little different from the ones we have covered so far in that it's a cache of objects that will potentially be used but might never be used. Earlier caches were built upon past references and access to objects. This cache is really built on things that might be accessed. Hence, we will refer to it as a "bag" instead. So, for example, suppose that we have 10 possible templates and will always apply one of 3 lifecycles to those documents during the creation time. If we wanted to cover all possibilities, then this implies at least 30 items in the bag (10 templates x 3 lifecycles). Since this serves a multi-user environment there will have to be multiple copies of each item so that there is enough items in the bag to serve the users before the bag is scheduled to be refreshed. If that number were set to 10 then the bag would have 300 objects.

The bag is maintained by a job that runs periodically which will re-fill the bag with the objects that have been depleted. In practice this job runs at nighttime, but is not restricted to. It could be run every hour if that was desired. The benefit of running it in off-peak hours is that it will have less of an impact on the on-line user response.

For this paper we provide an example application of this technique. There is a little more code involved so it has been put in a separate appendix. In this simple example the "application" will

create from templates and have the document part of a lifecycle when created. The create code is roughly as shown below. In our example we assume that the ACL and owner information of the pre-fab documents need not change from those of its initial creation.

```
Find template
Find lifecycle
Construct name of object in bag
Search the bag for this object
If found one then
      Check it out.
      Unlink it from template bag area
      Link it to user's area
      Rename it to what the user called it
      Save
Else not found
      Unlink it from template area
      Link it to user's area
      Rename it to what the user called it
      Saveasnew (to create the new object)
      Attach to life cycle
      Revert the template object
End
```

Figure 21: Create Algorithm

The most important point on the above algorithm relates to the naming convention of the objects in the bag. Each object in the bag is named from the template and the life cycle object id's. In our example, it looks as follows:

Lifecycle_object_id + template_object_id + template_ivstamp

For example:

`460004d28000c93f_090004d280037511_1`

The important point about the object name is that it encodes the current versions of the objects upon which it is based. This allows for changes to the templates or lifecycles to automatically invalidate the cached objects associated with them. The i_vstamp for an object is incremented each time the object is saved or modified. Normally a checkin operation will generate a new object id (when the version is incremented). The query that looks for the object id's for the lifecycle and template will only get the ids for the most current version. Hence once the template or lifecycle has been modified in this fashion, there won't be any objects in the bag that match it until the bag is updated.

However, it is possible for someone to checkin the template as the same version, keeping the object id the same. The i_vstamp would capture any such type of modification and invalidate the object in the bag when incremented.

The other aspect of this object-bag technique is the bag-refresh code. This is a job that runs periodically (perhaps once a day). Its algorithm is roughly as shown below.

```
Get a list of template id's and corresponding I_vstamps
Get object id's for life cycles
Store these in hash tables
Go through list of objects in bag
      If object_name does not match the template/lifecycle hash tables
            Delete it from bag
      Else
            If we need more of that object
                  Create more objects
For all of the lifecycle/template combinations that didn't have objects
      Create the correct number for the bag
```

Figure 22: Bag Refresh Algorithm

Since the bag refresh creates objects that are quite possibly never used (and even has to delete them when the template or lifecycle is modified) it will consume more CPU than a non-bag strategy. However, it is meant to consume its CPU during off-peak periods and to reduce the CPU that is required during the busy-hour online periods.

The performance of a create using this bag strategy is shown below. When a pre-created object is found in the object bag, response time is approximately 30% of the response time when it is not found in the bag and needs to be created to satisfy the user request. A bag hit is 3 times faster than creating the object on the fly each time.

It is important to note that the delays in creating the objects for the bag will not occur while a user waits for the creation (real time). The complex creation is performed at the back end with no user interaction.

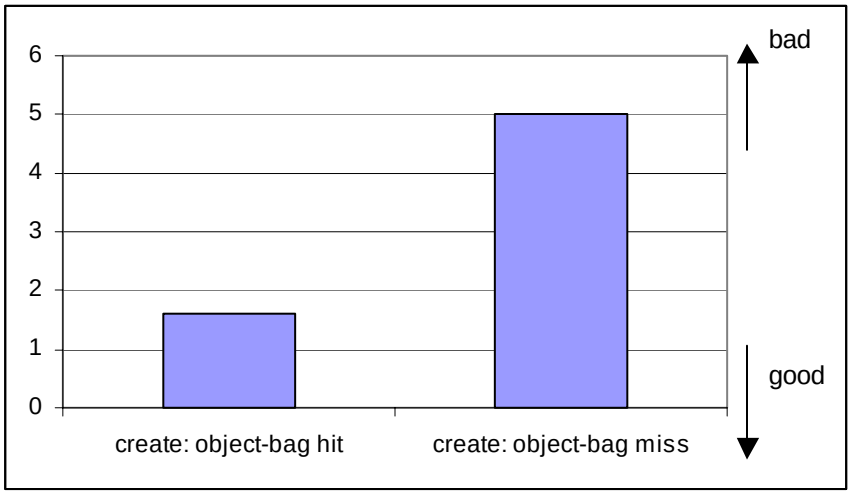


Figure 23: Performance Comparison of object-bag hit vs. miss

DQL-specific query techniques

There are many aspects of writing efficient DQL queries that are shared with SQL. However, there are some that are different. In this section, after a brief outline of the object-to-table model of Documentum, we cover the following DQL-specific query optimization areas:

1. Composite index restrictions in types
2. Multi-type join restrictions
3. Type depth performance
4. Queries against object attributes mirrored to full text engine

Object-to-Table Model

Documentum applies an object-oriented layer on top of the relational structure of the underlying database. The object attributes are actually stored in a number of tables. This is managed by the EContent server, and is in most cases transparent to the users and applications. The EContent server will translate DQL and API calls to the appropriate SQL statements for the RDBMS back-end. Each object type will have up to two corresponding tables in the database. For example, consider the object structure for company ACME:

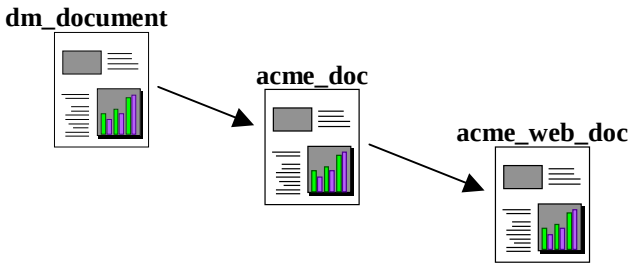
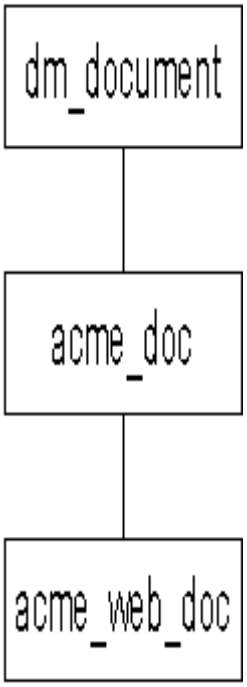


Figure 24: ACME's types

All corporate documents have a common set of attributes. These are defined in an object type called "acme_doc" which is a subtype of dm_document.

In addition, ACME is managing web content through the docbase, so they've defined a subtype of acme_doc that contains web-specific attributes. This hierarchy and a subset of the object attributes can be displayed as shown below.



dm_sysobject/dm_document

Attribute	Attribute Type	Repeating?
r_object_id	dmID	N
object_name	STRING	N
title	STRING	N
keywords	STRING	Y
i_contents_id	dmID	N

acme_doc (supertype dm_document)

Custom Attribute	Attribute Type	Repeating?
department	STRING	N
product	STRING	Y
document_class	STRING	N

acme_web_doc (supertype acme_doc)

Custom Attribute	Attribute Type	Repeating?
publication_date	TIME	N
review_date	TIME	N
expiry_date	TIME	N

Figure 25: Type Layout

In Documentum 4i, the underlying table structure is:

dm_sysobject_s
r_object_id
object_name
title
i_contents_id

dm_sysobject_r
r_object_id
i_position
keywords

acme_doc_s
r_object_id
department
document_class

acme_doc_r
r_object_id
i_position
product

acme_web_doc_s
r_object_id
publication_date
review_date
expiration_date

Figure 26: Underlying Table Layout

If a custom type consists only of additional single- or repeating-valued attributes, only the _s or _r table will be created. Note in the above example that there are no repeating attributes on acme_web_doc, so there is no acme_web_doc_r table.

Composite Index Restrictions for types

Indexes can be created on eContent server types to improve user-defined query response time. These user created indexes can also be composite indexes. However, the composite index cannot span a subtype. For example, suppose that we have the following type definitions:

```
Create type Foo ( x int, y int repeating )with Supertype dm_document;
Create type Bar (z int ) with Supertype Foo;
```

Figure 27: Example type definitions

And there is a query that accesses both x and z in the where clause so that the optimal "logical" index would be on that is on both x and z for a 'Bar' type object. If we tried to create such an index, it will fail. This is illustrated below.

```
Execute MAKE_INDEX with type_name = 'Bar',
                        attribute = 'x',attribute = 'z'
go
result
----------------
0000000000000000
(1 row affected)
[DM_TYPE_MGR_W_BAD_INDEX_ATTRIBUTE]warning:  "Attempt to create index on
type bar using illegal attribute 76"
```

Figure 28: Attempt to create a composite index across sub-types

This occurs because the attributes are stored in separate tables in the underlying RDBMS (in this case foo_s and bar_s) and indexes are only for single tables. There are two possible alternatives to creating the needed index:

- Flatten the type hierarchy so that the two attributes exist in the same table:

```
create type foo (x int, z int,y int repeating)with supertype dm_document
```

- Or take advantage of any special support that the underlying RDBMS has for join indexes. For example, with SQL Server 2000 one can create such an index outside of Documentum. Note, however, that join indexes are more expensive to update than normal indexes.

Another restriction on composite indexes is that one cannot mix non-repeating and repeating attributes in the same composite index. For example, the following DQL command will fail. This is shown below.

```
Execute MAKE_INDEX with type_name = ‘foo’,
                        attribute = ‘x’, attribute = ‘y’
go
result
-----------------
0000000000000000
(1 row affected)
[DM_TYPE_MGR_W_MIXED_INDEX]warning:  "Attempt to create index on type foo
with both repeating and single valued attributes."
```

Figure 29: Attempt to create a composite index with single valued and repeating attributes

Plan your data model wisely to ensure that it can be efficiently queried from.

Multi-type Join Restrictions

It is also important to understand the restrictions that center on multi-type joins. One important limitation is that repeating attributes cannot be part of the join. An example of this is shown below.

```
Create type Foo ( x int, y int repeating );
Create type Bar (z int );
 Register Table foo_r ( y int );

1> Select y from Foo, Bar where y = z
2> go
[DM_QUERY_E_REPEATING_USED]error:  "You have specified a repeating
attribute (y) where it is not allowed."

[DM_QUERY2_E_REPEAT_TYPE_JOIN]error:  "Your query is selecting repeating
attributes and joining types."
```

Figure 30: Example of repeating attribute in a type join

However, by registering the table foo_r the following query does succeed.

```
Select f.y from Foo_r f, bar b where f.y = b.z ;
```

Type Depth Performance

Another common question in data model design relates to the how nested the types should be. There are some specific limitations to the number of tables that can be in a single join (e.g., for Sybase this is 16), however, in most cases model complexity will be a bigger issue than join performance. The most likely cause of poor performance of deeply nested types is the inability to construct selective RDBMS indexes that provide good query performance (as seen in the previous composite index restriction examples).

Full Text Searchable Attributes

One method of circumventing the limitations of the underlying RDBMS for certain queries is to re-phrase those queries using Full Text searching in DQL. The 4.2 eContent server will automatically full text index string/char/text attributes. These can be searched using the full text extensions of DQL and can provide better search response time in certain cases than RDBMS queries (e.g., full text search is much faster than an RDBMS table scan for a large number of objects). Inefficient table scans normally occur in application searches due to usability requirements for search strings to be case-insensitive and to be able to use operators like CONTAINS. A typical object_name search given those "usability" needs would look like:

```
Select object_name from dm_document
where lower(object_name) like '%foo%'
```

Figure 31: A case-insensitive, CONTAINS query on object_name

The use of a function in the WHERE clause (i.e., lower(object_name)) disables index use in most RDBMS query optimizers. The wildcard operators '%' in the prefix of the search string does the same thing.

An application might encourage queries over a large set of attributes. Even if case insensitivity and CONTAINS are not required, it still may be undesirable to create RDBMS indexes on every searchable attribute in this set since additional RDBMS indexes increase the time to create or update an object.

An alternative is to utilize full text searches for attribute values that are mirrored into the Full Text engine. A full text DQL query on the above attribute would look as follows.

```
Select object_name from dm_document
search topic 'object_name <contains> *foo*'
```

Figure 32: A case-insensitive, CONTAINS Full Text DQL query

However, there are some differences in the search behavior of full text searches and normal char-based searches in an RDBMS. These include:

- The mirroring to the Full text engine of an object's attribute changes occurs 5 minutes after a change rather than during the update transaction. In addition, the attributes for content-less objects are not full text indexed and DQL predicate functions like FOLDER(DESCEND) don't have a full text analog. This makes the full text queries unattractive for applications like checking a folder for duplicate names prior to creating a document in that folder (cabinets and folders are content-less objects).

- Verity (the full text search engine) will treat non-alpha numeric characters as word separators. Hence, a search like:

  ```
  Select object_name from dm_document
  search topic 'object_name <contains> *foot_ball*'
  ```

  will return 'foot ball' as well as 'foot,ball' and 'ball,foot'.

Network Tips

In this section we cover tips to help users write applications that provide better response time over high latency and low bandwidth environments. In this section we cover the following topics:

- Using the dmcl trace to study client-side network behavior
- The batch_hint_size parameter
- Client cached queries
- Using the Proxy Recorder for studying Web-based client traffic

DMCL Tracing and Network Round-trip analysis

The 4i DMCL trace provides some network information that can be useful for Desktop Client applications. In addition to response time information, the number of round-trip messages is printed in the trace for each message. The example trace below illustrates this. The message counts are in bold in the following example.

```
API> get,c,serverconfig,r_server_version
# Network Requests: 11
# Time:        0.010 sec  '4.1.0.91 Win32.SQLServer7'
API> readquery,c,select r_object_id from dm_document where object_name =
'benchproj1_del1_d1.doc'
# Network Requests: 13
# Time:        0.200 sec  'q0'
API> next,c,q0
# Network Requests: 14
# Time:        0.000 sec  'OK'
API> get,c,q0,r_object_id
# Network Requests: 15
# Time:        0.000 sec  '090cc6fd80006ebf'
API> fetch,c,090cc6fd80006ebf
# Network Requests: 15
# Time:        0.300 sec  'OK'
API> get,c,l,object_name
# Network Requests: 18
# Time:        0.000 sec  'benchproj1_del1_d1.doc'
```

Figure 33: DMCL tracing of network operations

Batch Hint Size

Now lets study how to diagnose the impact of the 'batch_hint_size' parameter on network round trips. This parameter defines how many results should be sent in a single message from the server to the client. It is a hint because maximum message sizes might not allow the specified number of results to be stored in a single message from the server to the client. There is no setting that applies to all applications. The "correct" setting for this parameter is application dependent. If a list of objects needs to be displayed then under low bandwidth and high latency then more results per message is better. However, if each row in the query result is displayed separately then the 'time to first row' is very important and a lower value (perhaps even '1') is more appropriate. In general, a higher value is better because it leads to fewer network round trip messages.

Lets illustrate how this works. Suppose that we want to set this to 2 results per message. Then this is set in the dmcl.ini file in the following fashion:

```
[DMAPI_CONFIGURATION]
batch_hint_size = 2     # default is 20
```

Then our network information in the dmcl trace file shows us the effect of this parameter on our message traffic. Notice that after processing every two results the number of messages is increased by 1.

```
API> readquery,c,select r_object_id from dm_document where object_name
like 'benchproj1_%'
# Network Requests: 13
# Time:         0.170 sec  'q0'
API> next,c,q0
# Network Requests: 14
# Time:         0.000 sec  'OK'
API> get,c,q0,r_object_id
# Network Requests: 15
# Time:         0.000 sec  '090cc6fd80006ebf'
API> next,c,q0
# Network Requests: 15
# Time:         0.000 sec  'OK'
API> get,c,q0,r_object_id
# Network Requests: 15
# Time:         0.000 sec  '090cc6fd80006ec0'
API> next,c,q0
# Network Requests: 15
# Time:         0.000 sec  'OK'
API> get,c,q0,r_object_id
# Network Requests: 16
# Time:         0.000 sec  '090cc6fd80006ecd'
API> next,c,q0
# Network Requests: 16
# Time:         0.000 sec  'OK'
API> get,c,q0,r_object_id
# Network Requests: 16
# Time:         0.000 sec  '090cc6fd80006ed1'
API> next,c,q0
# Network Requests: 16
# Time:         0.000 sec  'OK'
API> get,c,q0,r_object_id
# Network Requests: 17
# Time:         0.000 sec  '090cc6fd80006ed2'
API> next,c,q0
# Network Requests: 17
# Time:         0.000 sec  'OK'
API> get,c,q0,r_object_id
# Network Requests: 17
# Time:         0.000 sec  '090cc6fd80006ece'
API> next,c,q0
# Network Requests: 17
# Time:         0.000 sec  'OK'
```

A new network message for every two results processed when the batch_hint_size = 2

Figure 34: Example of batch hint size

Cached Queries

Another important mechanism to improve performance on low-bandwidth and high-latency environments is the client cached queries feature. Client queries and their results can be cached locally on the client eliminating the need to send messages to the server. This mechanism is appropriate for infrequently changing data. For example, many applications will have global customizations files that are changed every couple of months. All of those queries can be cached locally and only refreshed when the global configuration files have been changed. This feature is enabled when:

- The query is marked as a cached query, either, by using the cachedquery dmcl method or in DFC. This is shown below.

```
IDfCollection c0 = q0.execute( session, q0.CACHE_QUERY );
```

Figure 35: Example call with cached query

- The effective_date is set properly in the dm_docbase_config object, and
- The dmcl.ini has `cache_queries = true`

The query and results are stored locally on the client and used until they have been deemed as invalid. Although there is a documented mechanism for handling cache coherency using the dm_docbase_config effective_date, I'd recommend that developers derive their own application specific coherency object and call the DFC flush() function once that object indicates that the queries need to be refreshed.

Lets illustrate the feature. Suppose that the dm_docbase_config effective date has been initialized properly and the cached_queries = true in the dmcl.ini. Then the query (shown below) will create the following entry in the cache.map file in:

`dmcl_root\dmcl\qrycache\hostname\docbaseid\username\cache.map`

```
DfQuery q0 = new DfQuery();
q0.setDQL("select r_object_id from dm_document where
object_name = 'foo.doc'");
IDfCollection c0 = q0.execute( session, q0.CACHE_QUERY );
while ( c0.next() ) {
      objId = c0.getId( "r_object_id" );
}
```

Figure 36: Cached query call example

For example, in our case above the file will look as follows (assuming our dmcl_root = c:\dctm and our docbase name is 'perf').

```
OBJ QUERY_CACHE_HDR_TYPE 3
1
xxx Sep 03 22:47:07 2001
A 4 perf
0OBJ QUERY_CACHE_TYPE 2
  1
A 80 select r_object_id from dm_document where object_name = foo.doc'
  1
A 90 c:\dctm\dmcl\qrycache\ENG075\000cc6fd\buec0sql.809\8001410d.NMU
0
```

Figure 37: cache.map after query is executed

The actual results are stored in that `8001410d.NMU` file. The contents, after our query was executed, are shown below.

```
TYPE QR 0000000000000000
 NULL
1
  r_object_id  STRING S 16
OBJ QR 1
A 16 090cc6fd80010d01
0
```

Figure 38: contents of `8001410d.NMU`

The formats of these files could change at any time. I show them here to just illustrate what is happening underneath the covers. The contents of `8001410d.NMU` (which is by the way a random file name) contain the object id result from the previous query.

To invalidate the cache it is just a matter of calling the DFC flush() function. This will delete both the query and result files for the calling user. As new cached query calls are made an in memory structure is created to hold the query and results. When the session disconnects these in memory structures are written to disk.

```
Session.flush(querycache);
```

Figure 39: Invalidate the query cache with flush()

The application-specific cache coherency mechanism can be as simple as the I_vstamp of an object that is updated by your application administrator or updated by the application itself. So when an application is started it can query that object's I_vstamp to compare it with one stored locally. If the I_vstamp has increased then the flush is called. If it has not then the subsequent cached queries will retrieve what was stored on disk.

Now, lets suppose we have two cached queries. If we run them again, our network trace shows us that the cache is being used properly to enable good performance. This is illustrated below.

```
API> get,c,serverconfig,r_server_version
# Network Requests: 11
# Time:        0.020 sec  '4.1.0.91 Win32.SQLServer7'
API> cachequery,c,select r_object_id from dm_document where object_name =
'foo.doc'
# Network Requests: 13
# Time:        0.010 sec  'q0'
API> next,c,q0
# Network Requests: 13
# Time:        0.000 sec  'OK'
API> get,c,q0,r_object_id
# Network Requests: 13
# Time:        0.000 sec  '090cc6fd80006ebf'
API> close,c,q0
# Network Requests: 13
# Time:        0.000 sec  'OK'
API> cachequery,c,select r_object_id from dm_document where object_name =
'bar.doc'
# Network Requests: 13
# Time:        0.000 sec  'q0'
API> next,c,q0
# Network Requests: 13
# Time:        0.000 sec  'OK'
API> get,c,q0,r_object_id
# Network Requests: 13
# Time:        0.000 sec  '090cc6fd80010d01'
API> close,c,q0
```

Figure 40: Example dmcl trace of cached queries

The 2 messages exchanged for the initial cache query are unavoidable setup overhead.

Proxy Recorder

The ProxyRecorder is a Documentum internal tool that is useful for diagnosing and analyzing browser-based user interfaces. In this section we illustrate how it works and how it can be used to diagnose a browser user interface design. The tool is “unsupported” and available on the Documentum developer website.

First, the ProxyRecorder is a tool that sits inbetween the browser and the HTTP server and monitors the HTTP/HTML traffic that passes between these two. It writes out a spreadsheet friendly summary of the information tracking messages and their latencies and size. It can be extremely useful to pinpoint the cause of poor browser-based interface performance.

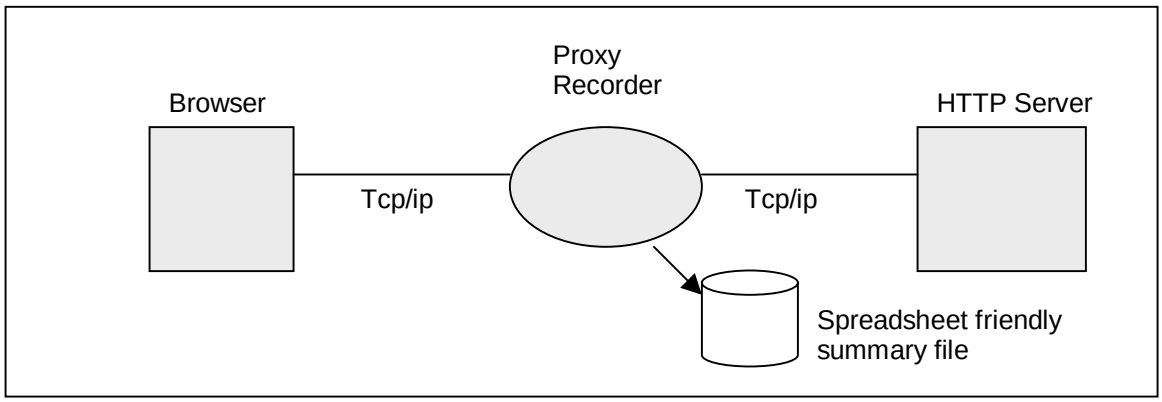


Figure 41: Proxy Recorder Architecture

A sample of the proxy recorder output looks as follows:

Time	ResStrt	ResEnd	ResDif	ReqLen	ResLen	
9:25:33	1512	1722	210	635	29967	/iteam/universal/iteam/centers/personal/quickView.jsp?com
9:25:23	1202	1212	10	758	3613	/RightSite/connect?DMW_DOCBASE=iteam30b&DMW_PA
9:25:22	1151	1161	10	649	2777	/wdk/cmd/exec.jsp?class=com.documentum.iteam.api.com
9:25:26	631	641	10	502	14734	/iteam/universal/iteam/toolbar/toolbar.jsp
9:25:30	391	391	0	555	3780	/iteam/universal/iteam/footer/footer.jsp
9:25:29	250	260	10	775	9394	/iteam/universal/iteam/centers/participant/participantTabBa
9:25:25	170	190	20	430	22833	/iteam/universal/iteam/iTeam.jsp?Reload=4298
9:25:31	151	151	0	653	2577	/iteam/universal/iteam/centers/participant/participantTabDis
9:25:29	110	110	0	485	3167	/iteam/universal/iteam/centers/participant/participantCompo

Figure 42: Sample Proxy Recorder Output

Where:
ResStrt is the time difference from sending the request to the start of the response.
ResEnd is the time difference from the sending of the request to the end of receiving the response.
ResDif is the time difference between starting to get the response and finishing getting the response.
ReqLen = request msg length
URL = the url
The timestamp values (all in milliseconds) are illustrated below.

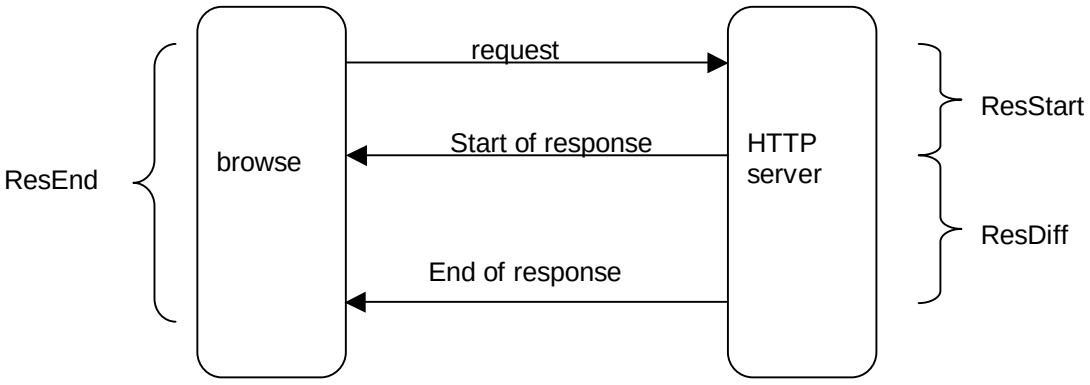


Figure 43: Illustration of Proxy Recorder Timings

Workflow and Lifecycle Tips

This section will cover some techniques that help to ensure good response time for workflow and lifecycle users. These two 4i features are highly customizable and they allow developers to define powerful actions to occur as a document moves through its lifecycle and workflow transitions. The best techniques are:

- Asynchronous execution of the operations through either separate threads or the Workflow agent, and
- Multiple-parallel workflow agents

Asynchronous Execution of Workflow and Lifecycle Commands

Application specific processing can be defined for lifecycle and workflow transitions. With Lifecycle, scripts can be defined to execute prior to and after entering a state. The customization can be setup such that the state transition depends on the successful run of the script. The state transition can also be made independent of the successful execution of the script.

These scripts perform any application defined set of tasks. If the normal promote method is called, then the execution of those custom scripts will be synchronous with respect to the promote call. If the scripts have long execution times this will translate to delays seen by the user.

There are many options available to execute lifecycle operations asynchronously. Using any of them can give the user near instantaneous response to their request. The lifecycle operation will not necessarily complete by the time their screen returns control, and hence, the semantics are more like e-mail (which returns immediately but does not imply that the e-mail has been received) or an online trading system (which indicates that the trade request has been accepted, but not completed yet). The methods available include the following:

- Use an existing Desktop Client lifecycle component which creates a separate thread to execute the lifecycle operation asynchronously, or

- Use the DFC `schedulePromote()` method to schedule the promote in the near future. In this case the promote operation is scheduled to run in the background as a server job.

An example of the schedulePromote() use is shown below. The code “schedules” the document promotion to occur in the next minute.

```
IDfSysObject obj2 = null;
Date dt = new Date();
DfTime dft ;
  System.out.println( "date = " + dt.toString() );
  if (   dt.getMinutes() == 59 ) { dt.setSeconds( 59 ); }
  else { dt.setMinutes( dt.getMinutes() + 1 ); }
  dft = new DfTime( dt.toString );
  System.out.println( "date = " + dft.toString() );
  ob2 = s.getObjectByQualification( "dm_sysobject where
                                    object_name = 'my_doc'");
  obj2.schedulePromote("Review", dft, false );
```

Figure 44: Schedule Promote Example

Although the response time of a synchronous promote call will vary depending on the backend "action script", the graph below illustrates the performance of schedulePromote() when used on a lifecycle that has "heavy lifting" action procs. In this example the action proc distributes inbox requests to users in a group and e-mails them as well. The user perceived response time of the synchronous promote is 8 times that of the schedulePromote() call.

The trade off of a schedulePromote vs. a synchronous promote is that even though the user "perceived" response time is quick, it will take longer to actually promote the document in the scheduled case than in the synchronous case. Our code example schedules the document promotion to occur 1 minute into the future. However, in practice this extra latency is not considered serious, an application employing this may want to consider handling the case that the user initiates the document promotion more than once as they wait for the initial promote to succeed.

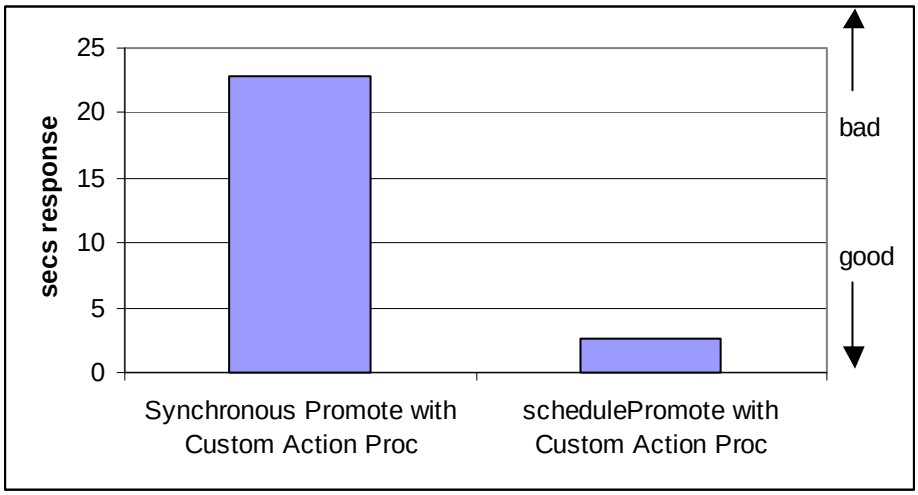


Figure 45: Example Response time of Synchronous vs. Asynchronous Promote

The schedulePromote call is best used in environments that don't support asynchronous execution on the client machine (e.g., Rightsite clients, WDK 4.2 clients).

Scripts can be defined for special processing to occur on workflow state transitions as well. By definition these scripts are performed automatically and asynchronously by the workflow agent.

Running Methods in a Java Application Server

eContent Server methods are an essential part of the automated workflow processing (they are the 'work' executed by the automatic workflow agent. They can also be used by applications that want to accomplish some DFC/dmcl processing on the eContent server machine. The 'method' invoked can be a docbasic script, a dmawk32 script, a standalone program, or a java class. When invoked the server creates a separate process to execute the script/program. In almost every case the method code will create a connection back to the eContent server to perform its tasks. The initialization overhead of a standard server method is primarily the creation of the process and the connection back to the eContent server.

The shift from the low-level dmcl api to the high level DFC api provided users with a higher level and simpler interface for building their Documentum applications. However, since the DFC is java-base, its startup costs are higher than the lower level C++ dmcl-based application written with Docbasic.

Starting a java-based server method in a persistent container (a Java application server) significantly reduces the startup overhead. Only the initial DFC call experiences the startup penalty, all subsequent ones do not. In addition, the application server enables the use of dmcl session pooling further minimizing the overhead associated with method connecting back to the eContent server.

In this section we outline how to create the infrastructure that allows Java-methods to run in java application servers.

The basic architecture of this is outlined below. The server invokes a program that takes its command line arguments and formats them into a url and parameter list and sends an HTTP GET to the application server and waits for the response. On the application server side the called JSP routine translates the url arguments back into normal command line arguments and passes those to the method, now called as a normal java method.

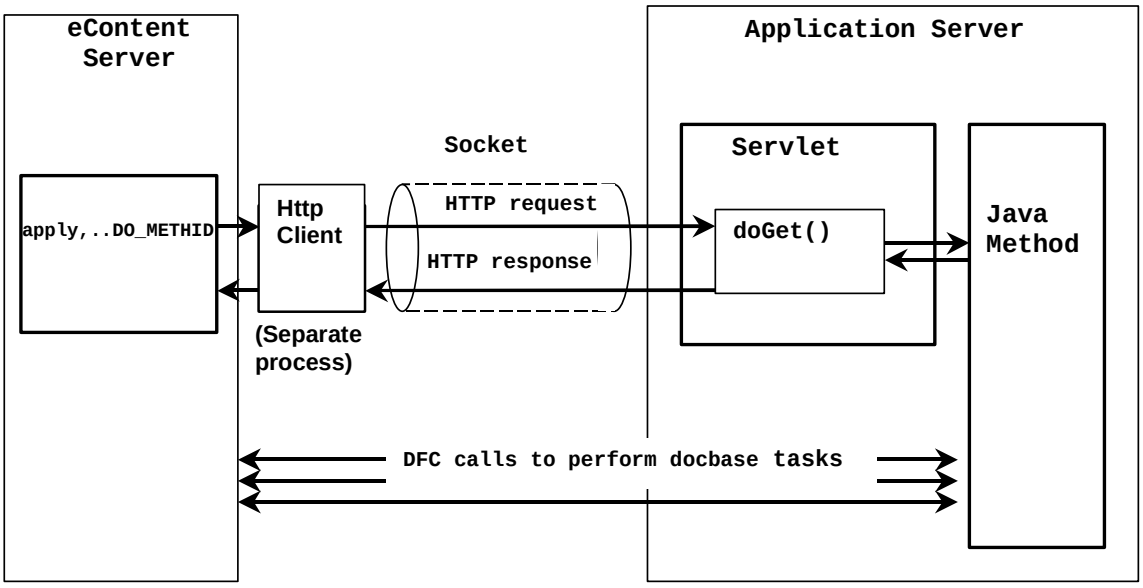


Figure 46: Architecture of Java-method running in an Application server

Lets first look at our method. This is simple, but captures the essence of what needs to be done. The source is shown in Appendix D. Notice that System.exit() cannot be called if this method is run in an application server. That is, the method source was designed to run either in an application server or on the command line.

Hence our method could be invoked as a normal server method. If it was the method verb would look as shown below:

```
java samplemethod -docbase_name mydocbase -user myuser -ticket mypassword
-executing_where app_server
```

In this case the samplemethod.class file is located in `%Documentum%\product\4.2\bin`.

The first step in moving the method to an app server is to put our http-sender program in place. In this example the program is a java-based program (see `HttpClient.java` in Appendix D). We recommend writing it in C or C++ for maximum efficiency (for a 4.X eContent Server). In the 5.0 eContent server, this logic will be part of the base server and the Http message will be sent directly

from the eContent server. Prior to sending the message this logic takes the command line arguments and translates them into url arguments.

On the server side the program invoked pulls out the arguments and translates them into command line arguments prior to sending them to the called routine. This source RemoteMainServlet.java is also in Appendix D.

The setup of the method with the application server will differ depending on which application server vendor is chosen. In our example, we used Catalina and did the following to set it up:

1. create a directory under `%CATALINA_HOME%\webapps\` called remoter.
2. create additional directories:
   a. `%CATALINA_HOME%\webapps\remoter\WEB-INF`
   b. `%CATALINA_HOME%\webapps\remoter\WEB-INF\classes`
   c. `%CATALINA_HOME%\webapps\remoter\WEB-INF\lib`
3. Put the web.xml (see appendix D) in `%CATALINA_HOME%\webapps\remoter\WEB-INF`
4. put dfc.jar in `%CATALINA_HOME%\webapps\remoter\WEB-INF\lib`
5. put samplemethod.class and RemoteMainServlet.class in `%CATALINA_HOME%\webapps\remoter\WEB-INF\classes`
6. Reboot Catalina

Once this is done the method verb can be changed to:

```
java HttpClient -docbase_name mydocbase -user myuser -ticket mypassword
-executing_where app_server
```

Although mileage may vary, the improved execution time for our simple method is about 1 second on an 866Mhz Intel PIII system for single user. The next version of the eContent server will improve upon this even further by eliminating the need to create the HttpClient process for each method invocation.

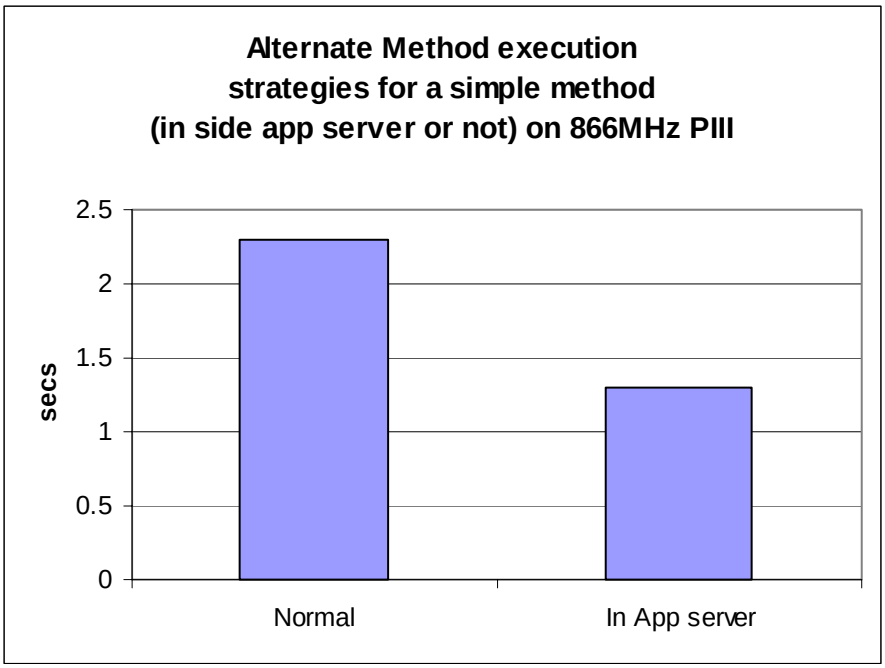


Figure 47: improvement in method execution time

Reducing Automatic Workflow Steps to Improve Latency

For some applications the latency time for automatic workflow processing is an important metric. There are several ways to tune the latency. Earlier we saw how the execution time of the server method can be reduced if the Java-based eContent server method is run in an application server. Another heuristic is to reduce the number of automatic workflow steps needed in the workflow cycle.

Figure 48 below shows a sample workflow. The three automatic tasks that are executed before a message is sent to the reviewer are high-lighted in red.

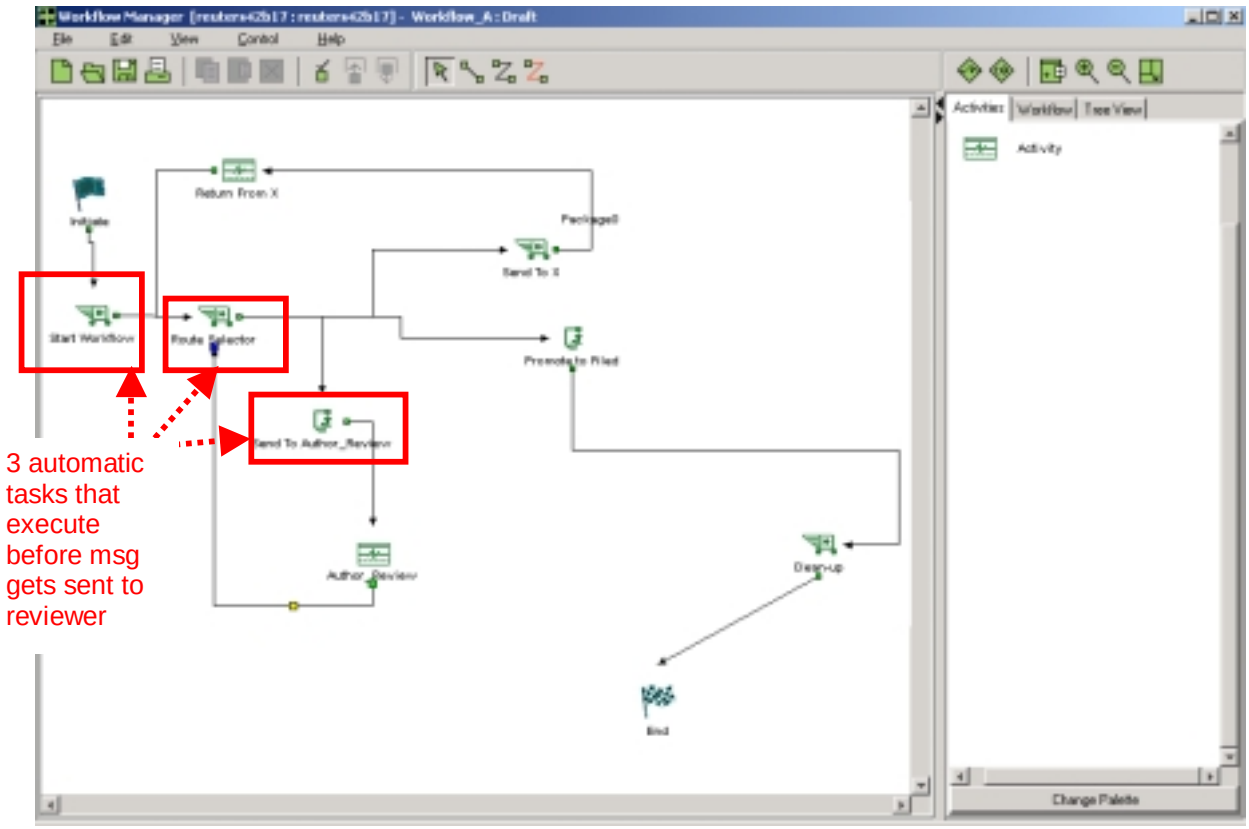


Figure 48: Initial workflow definition

The first automatic step 'Start Workflow' does almost nothing except execute a noop script. The second 'Route Selector' routes the document based on an attribute value of the document being routed (which identifies which node to send it to next). The third 'Send to Author_Review' does the meat of the custom processing. These processing steps mean that the document makes three trips through the automatic workflow queue before the review message is queued for the reviewer. In a multi-user environment each trip through the queue will also imply additional latency since the requests are executed in FIFO order.

Our first optimization was to eliminate the need for a separate 'Start Workflow' and 'Route Selector' automatic tasks. The only reason that the 'Start Workflow' node was used is that the first node in a workflow can only have an input 'port' (arc on this diagram) from the actual start. As is shown in the diagram the 'Route Selector' has inputs coming from not only the 'Start Workflow' but the 'Author Review' manual step and the 'Return from X' manual step. We were able to eliminate the Start Workflow step by creating two 'Route Selector' steps: one as an initial step and the other to receive

the inputs from the two manual processes. This modification of the workflow is shown below. It reduced the number of automatic workflow steps from 3 to 2. This can easily save several seconds off of the latency for the document in a heavy multi-user environment.

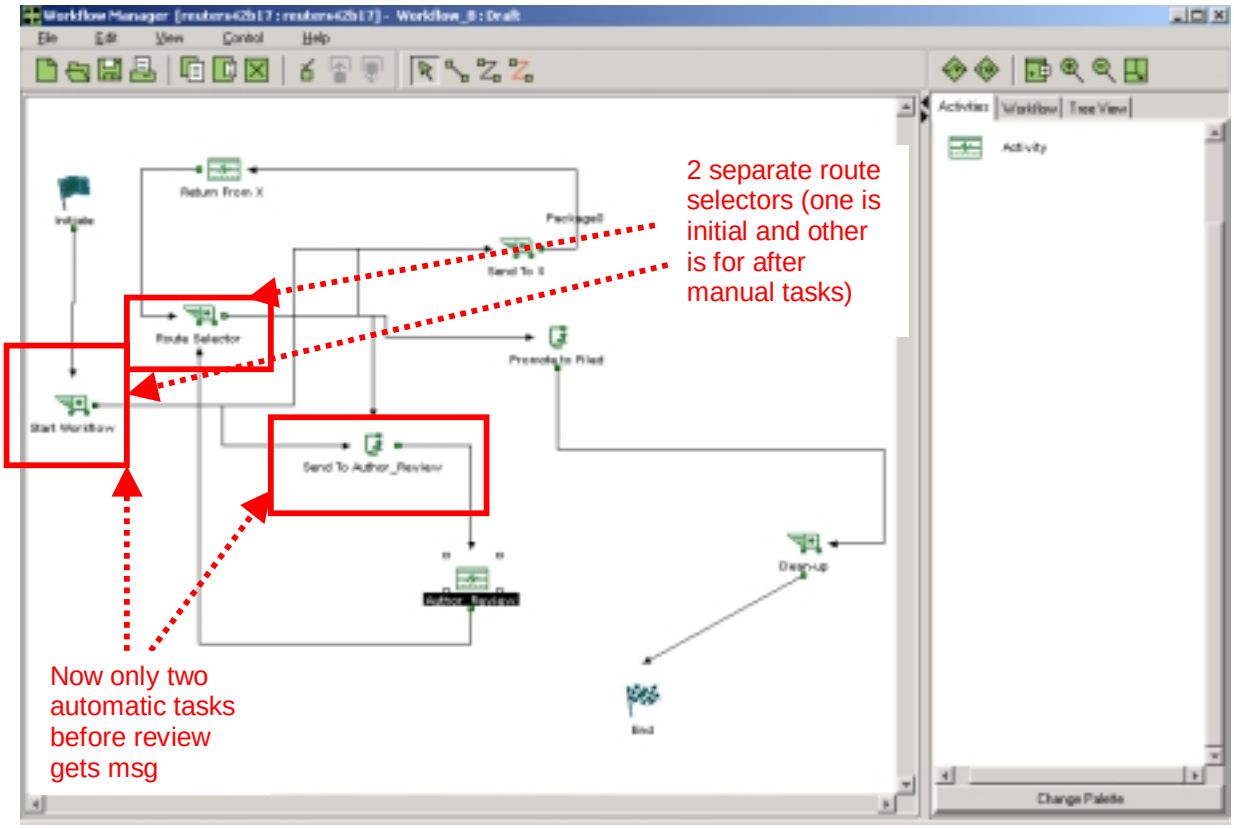


Figure 49: Reduction of 3 automatic steps to 2 automatic steps before Author Review

Multiple-Parallel Workflow Agents

As more users initiate automatic workflow tasks the queue of tasks waiting to be executed could grow large as the normal single workflow agent operates on the tasks. This is especially true for tasks with many steps. In this section we describe how to check the length of this queue and how to spawn multiple, parallel workflow agents in an effort to ensure that latency of tasks within the system is minimized.

First the length of the workflow automatic task queue can be checked with the following DQL query.

```
select count(*) from dmi_workitem where r_runtime_state in (0,1) and
r_auto_method_id <> '0000000000000000'
```

Figure 50: Query to check the workflow automatic task queue length

Second, to setup multiple workflow agents to service this queue it is necessary to add additional eContent servers for the same docbase. Each eContent server “instance” will add an additional workflow agent. These eContent server’s can run on the same machine as the other ones and client connections to them are load balanced by the Docbroker (which randomly picks from the list of available eContent servers each time a connection is established).

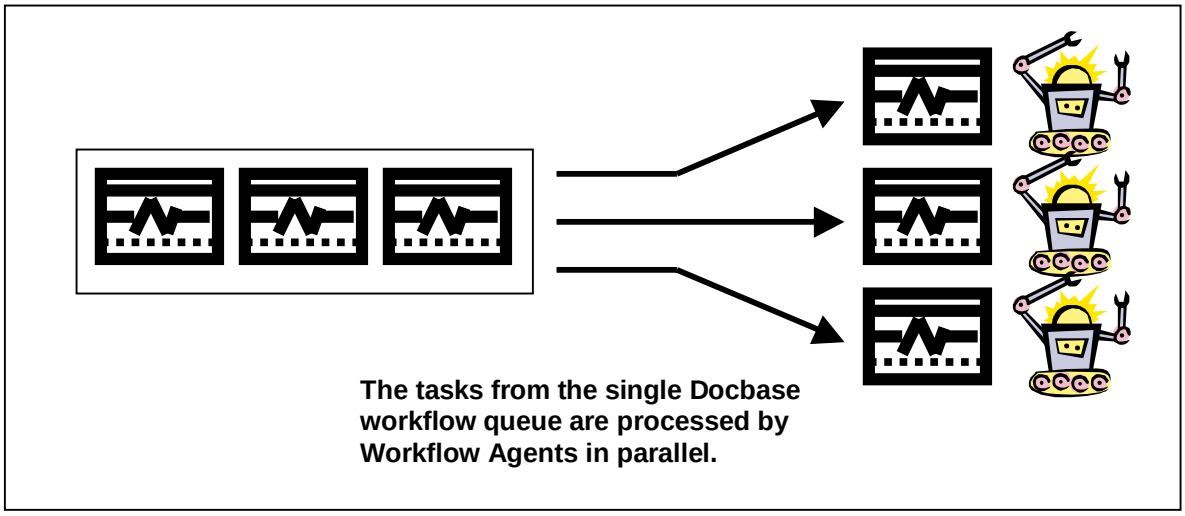


Figure 51: multiple parallel Workflow Agents servicing the single Docbase workflow queue

The following steps describe how to setup multiple eContent Servers on the same host with a single docbase:

1. Login to eContent Server and create a new server config object via IAPI. For example, assuming an existing eContent Server called “wcm” with a service name of “wcm” and is going to add a new one on the same host with the name of “wcm1”:

```
API> retrieve,c,dm_server_config
    {id-is-returned}
API> set,c,l,object_name
     wcm1
API> saveasnew,c,l
     OK
```

2. Create a new server.ini file for this newly created eContent Server:

- Navigate to your docbase subdirectory, for example, $DOCUMENTUM/dba/config/your-docbase-name
- Make a copy of server.ini. For example,
  $cp server.ini server1.ini
- Edit two lines in server1.ini:
  Service = wcm1
  Server_config_name = wcm1

3. Edit the /etc/services file to have the new eContent server name with a different port number. In this case the service name is wcm1.

4. Create START and SHUTDOWN eContent Server scripts. That is, dm_start_wcm1 and dm_shutdown_wcm1. Simply make a copy of dm_start_wcm and dm_shutdown_wcm and edit two lines in each. In START script, it is important to create a wcm1.log for this new server (server1.ini). In SHUTDOWN script, it is a matter of identifying the correct eContent server for the shutdown command by changing ‘iapi wcm …’ to ‘iapi wcm.wcm1 …’.

5. Boot the eContent Server using the startup script and verify that you can log into it by specifying the name and check the log file (wcm1.log) to ensure it is up and running properly:

   $ Idql **wcm.wcm1** –Uusername –Ppassword

On Windows NT the registry service entries for the additional eContent server's need to be "cloned" from the original ones and modified to reflect the new server.ini and config name in the service invocation. Cloning a registry service entry involves:

- writing out the key values (using the regedt32 Registry→Save Key menu item),
- adding a new key (using Edit→Add Key), and then
- restoring the old key values on the newly defined key (Registry→Restore)

The key for the service is located at:
`HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\DmServer`*`docbasename`*

Where docbasename is the name of the docbase involved.

The following graph illustrates the effect of additional parallel workflow agents on a 600+ concurrent user environment that makes heavy use of automatic workflow tasks. In this test we increased the number of parallel workflow agents from 10 to 15 and saw a 30% reduction in the peak workflow queue length.

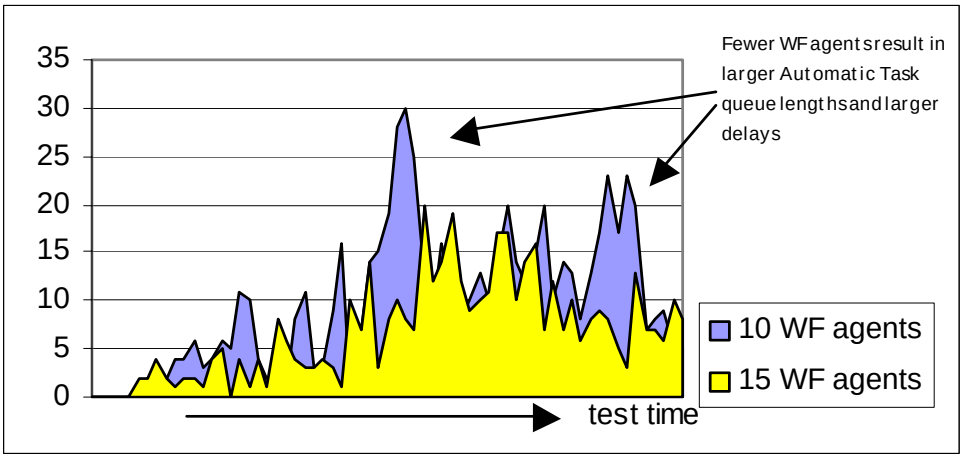


Figure 52: Automatic Workflow task queue length during 620 concurrent contributor run

Large Concurrent User Support

This section will cover concurrent user tuning issues. We will emphasize using session pooling and then cover blocking and deadlocking using SQL server as an example.

eContent Server Session Pooling

eContent server session pooling is a feature targetted for Web application servers that multiplex a large number of distinct users on a smaller number of eContent server sessions. Since the feature reduces the actual number of needed eContent server sessions it also reduces server memory needs and facilitates the scalability of the backend RDBMS. The pooling of sessions occurs on the application server side. Sessions are multiplexed through an explicit grab and release using the eContent server connect and disconnect methods. When a “connect” method occurs a free connection is picked from the pool. A “disconnect” method just releases a session back into the pool. If a connection cannot be found in the pool it will be created and placed in the pool.

Session pooling is enabled through the dmcl.ini file:

```
[DMAPI_CONFIGURATION]
connect_pooling_enabled = T
connect_recycle_interval = 100
```

The connect_recycle_interval represents the number of times a connection will be reused before it is explicitly destroyed. This is a longevity feature that ensures that the sessions in the pool are continually refreshed and memory leaks do not occur.

An application can derive maximum benefit from session pooling if it is structured and designed in a more “stateless” fashion. For example, a connect/disconnect call could occur for within every JSP page. This is illustrated in Figure 53. Documentum has a benchmark kit available on the developer Web site that illustrates a simple application of session pooling using JSP pages. This benchmark kit demonstrated an 85% reduction in needed eContent server sessions and corresponding database sessions. This is illustrated below were we compared the measured eContent server and Database sessions with those of the same application operating in a stateful-non-session pooling manner.

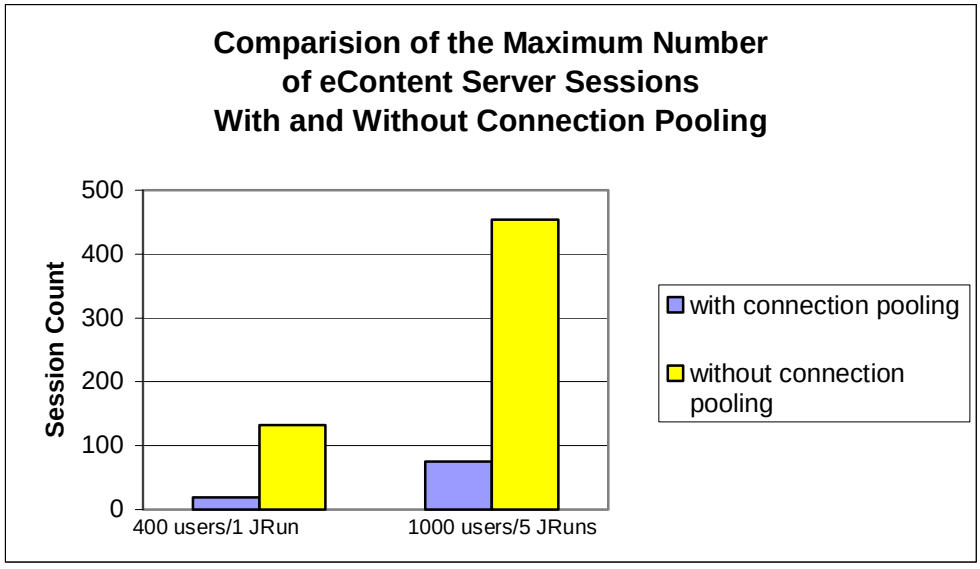


Figure 53: Reduction in server sessions with session pooling

These results are documented fully in a benchmark report that is available upon request.

Each of those JSP pages had the following structure to allow it to take advantage of session pooling.

```
%@ page language="java" %>
<%@ page import="com.documentum.fc.common.*"%>
<%@ page import="com.documentum.fc.client.*"%>
IDfSession session = null;
try    {
        String user = Request( "user" );
        String password = Request( "password" );
        String docbase = Request ( "docbase" );
        String folder = Request( "folder" );
        String domain = "";

        IDfLoginInfo li = new DfLoginInfo();
        li.setUser( user );
        li.setPassword( password );
        li.setDomain( "" );
        IDfClient dfc = DfClient.getLocalClient();

         // ----------------------------//
        // Do Useful DFC work here      //
       // ----------------------------//


      // Disconnect the session
       session.disconnect();



// Handle exceptions that can be thrown by above code
//
catch (DfException e)
    {
        out.println( "Unexpected error: " + e.getMessage() );
        try
        {
            if ( session != null )
                session.disconnect();
        }
        catch( DfException e1 )
        {
            // Do nothing.
        }
    }
%>
```

Figure 54: Sample DFC code that takes advantage of session pooling.

That benchmark passes the docbase,username, and password as an argument to every JSP page. This has a few security disadvantages (to say the least). One can provide a single trusted login to application servers using the getlogin server api method and login tickets. When a super user calls the getlogin method api for another user a ticket will be returned that can be used by that user name to login in (instead of sending a password). This feature allows for trusted logins between application servers and the eContent server. By initially setting up a connection to eContent server as a super user a trust relationship is established between the Application server and the eContent server. Additional connections can be established for other users without having to compromise password information. This behavior is illustrated below.

```
# illustrated using the iapi utility
API> getlogin,c,user1
...
DM_TICKET=000000000011b381
API> connect,sample_docbase,user1,DM_TICKET=000000000011b381
...
s1
API>
```

Figure 55: Illustration of trusted login setup

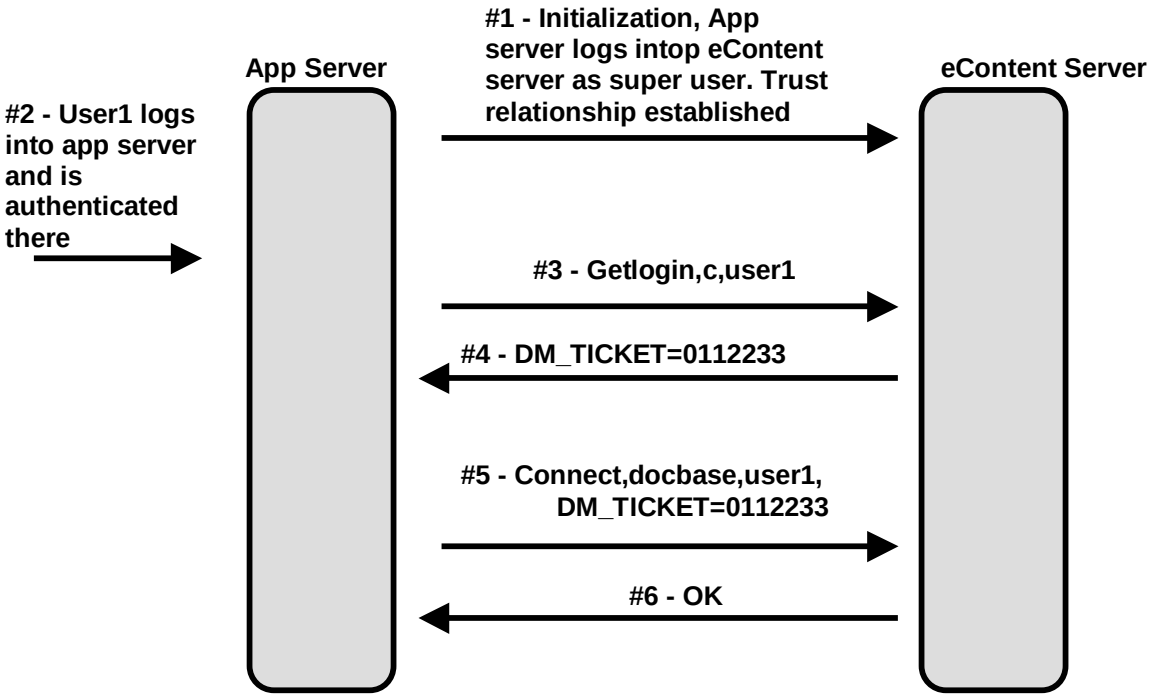


Figure 56: Illustration of App server trusted login scenario

Session pooling can also be enabled for the WDK 4.2 (although it is not officially supported). It can be enabled by adding the following lines in the config.xml.

```
<section>
      <name>Session</name>
...
      <property>
         <name>EnableConnectionPooling</name>
         <value>true</value>
      </property>
   </section>
```

Figure 57: How to enable session pooling in WDK 4.2 (unsupported feature)

RDBMS Blocking

The following section covers RDBMS blocking issues. Blocking is not an issue in a totally read-only environment. In environments in which updates are infrequent it might not even be noticeable. However, It can become noticeable in environments that frequently update objects and long response times are the result. Queries that execute in a fraction of a second in single user mode might take minutes when many users are working concurrently. In this section we discuss items that affect blocking, including:

- The concurrency mechanisms provided by the RDBMS
- Query plans (index selection, join strategies, and table scans), and
- Strategies to achieve and maintain optimal plans (statistics updates, design)

RDBMS Concurrency Mechanisms and Blocking

The concurrency control mechanisms used by the RDBMS vendors differ and these differences can alter the behavior of your system and lead to poor concurrent user performance. The most important difference to understand is Oracle's multi-version (shadow page) mechanism vs. the other RDBMS read-locking schemes. Multi-versioning allows Oracle to provide transaction isolation without read locks and predicate locks.[2] Readers are never blocked, only writers. That is, when a reader tries to read some data that is locked and being modified, then Oracle will create a pre-modification view of the data and place that in a separate "rollback segment" so the reader can view the non-modified record without blocking. However, Oracle does have write locks. If two threads try to modify the same record then one will get a write lock and the other will block until that write is committed (or rolled back).

Oracle is unique in this form of concurrency control. The other RDBMS will block readers because readers must obtain read locks on the data prior to accessing it. These two mechanisms are conceptually outlined below.

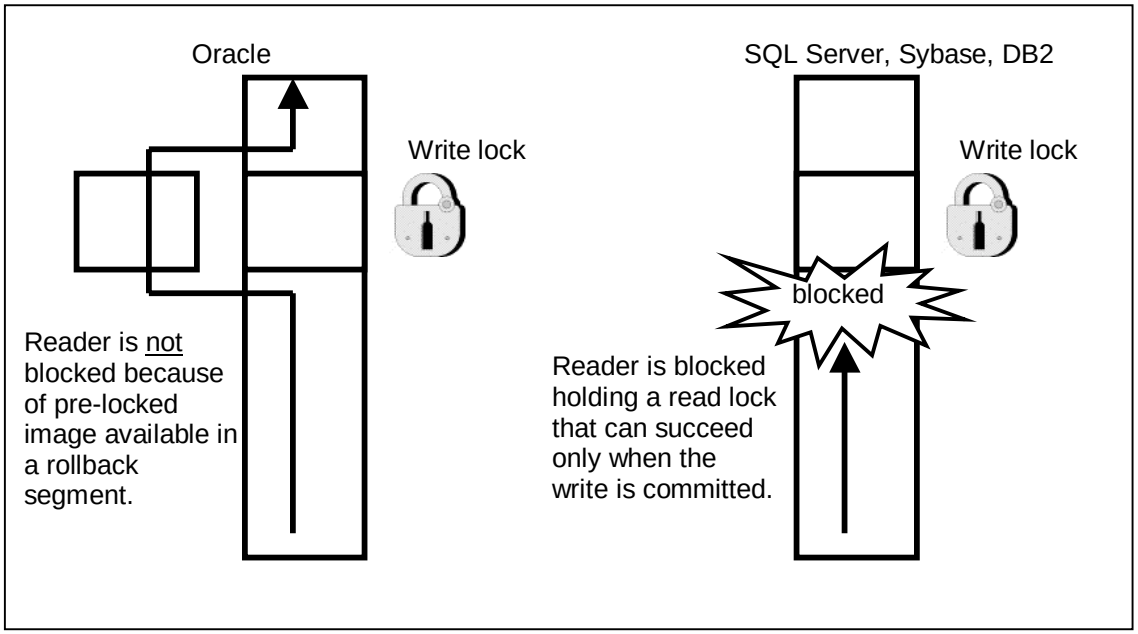


Figure 58: Illustration of Major difference in Concurrency control mechanisms

[2] A more complete discussion of Oracle's concurrency control mechanism can be found in "Concurrency Control, Transaction Isolation, and Serializability in SQL92 and Oracle7", an Oracle White paper, July 1995, Part No. A33745.

High capacity systems can be built with all of the RDBMS, however, assuming that all of the database systems have Oracle's concurrency behavior is a good way to have concurrency issues when moving to a different database system vendor.

Now lets illustrate this difference in some common blocking examples.

Updated Rows Blocking Readers

One of the simplest examples of how these mechanisms differ is illustrated in the case when one transaction is modifying a row and another user is doing a scan. Two blocking events could occur in the non-Oracle RDBMS:

- The writer could block the reader making the table scan take longer, or
- The reader could block the writer making the update take longer.

The former is illustrated below, where the scan is a table scan, but it could easily be a point query or range scan.

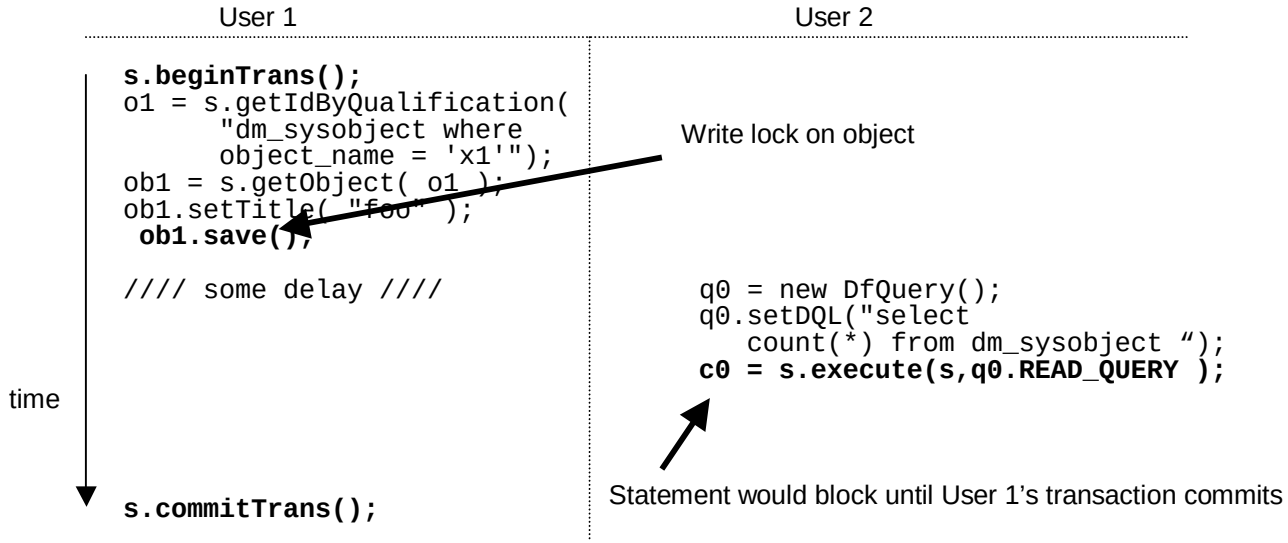


Figure 59: Example of Table Scan blocking with Non-Oracle RDBMS

In this case the modified record does not block any readers in Oracle because pre-modified images are written out to the Rollback segments for their viewing purposes.

In our example above, again, we use a table scan for illustrating the issue. However, the same behavior can occur if an underlying index just isn't selective enough. The following example illustrates this behavior.

Suppose that we want to view documents whose object_name's begin with 'foo' (this would select objects with names like 'foo', 'fool', and 'foot') and whose title begins with 'bar' (example, 'bar', 'bar1', and 'bart'). Suppose we have three objects in the Docbase that begin with 'foo'.

Object Id	Object_name	Title
1	foo	bar
2	foot	cat
3	fool	bar1

Figure 60: Sample data for blocking example

However, only two have a title that begins with ‘bar’.

If a user executes the following code (figure 53) to update the object with a title of ‘cat’. Then the following DQL query (figure 61) would block until the first user commits the above update.

```
session.beginTrans();
objId2 = session.getIdByQualification( "dm_sysobject where
object_name = 'foot' and title = ‘cat’ ");
obj2 = (IDfSysObject ) session.getObject( objId2 );
obj2.getTitle() ; // value is ‘cat’
obj2.setObjectName( "puff" );
obj2.save();
```

Figure 61: update code example

```
Select r_object_id from dm_document where object_name like ‘foo%’
and title like ‘bar%’
```

Figure 62: Query in update example

Why does this happen? Our query won’t return the object with a title of ‘cat’. The reason is that the query will initially use the object_name index and pick up records that begin with ‘foo’. The filter to select records beginning with ‘bar’ doesn’t happen until a later step in the query plan. Too late: the access plan tried to read a row that has an exclusive lock on it. Our query gets blocked before even reaching that ‘bar’ filter step in the plan. This is illustrated below.

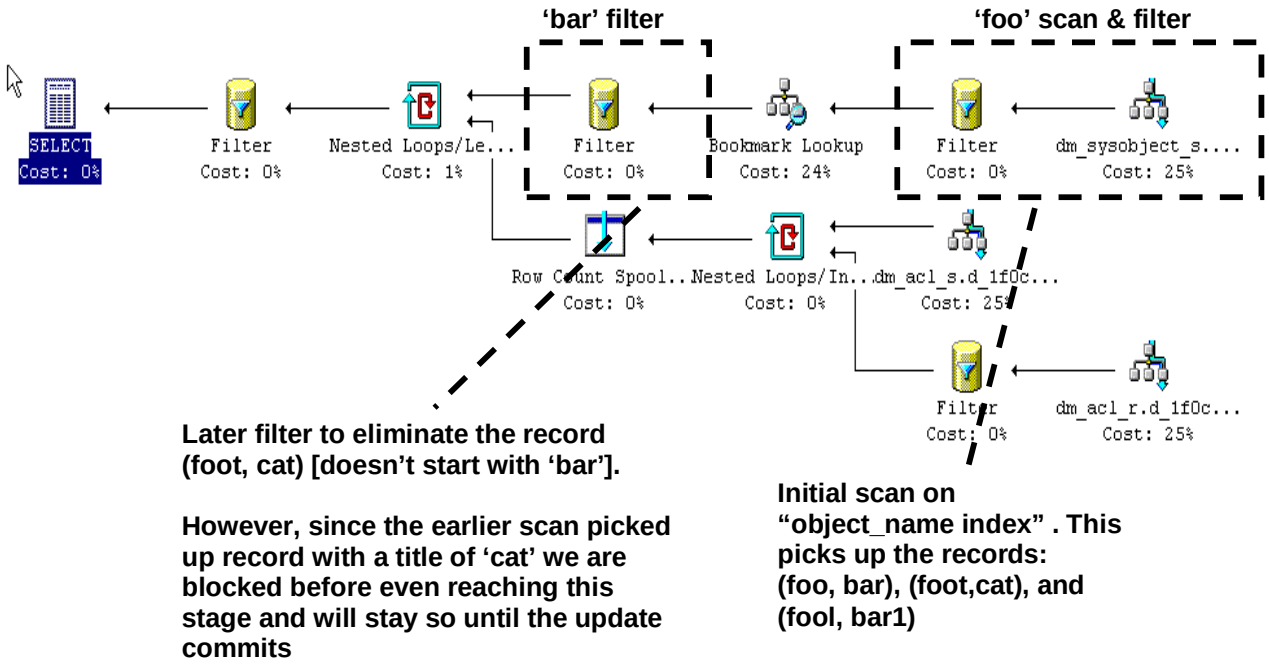


Figure 63: Example Query Plan

Inserted Rows Blocking Readers

Another, more subtle, blocking scenario occurs in cases when the reader blocks as a writer is inserting some, yet uncommitted, rows. In this case although the row has not yet been committed to the RDBMS, however, the row has been inserted and the reader still has to attempt to view it to see if it qualifies for the result set and hence blocks. This is illustrated in the example below.

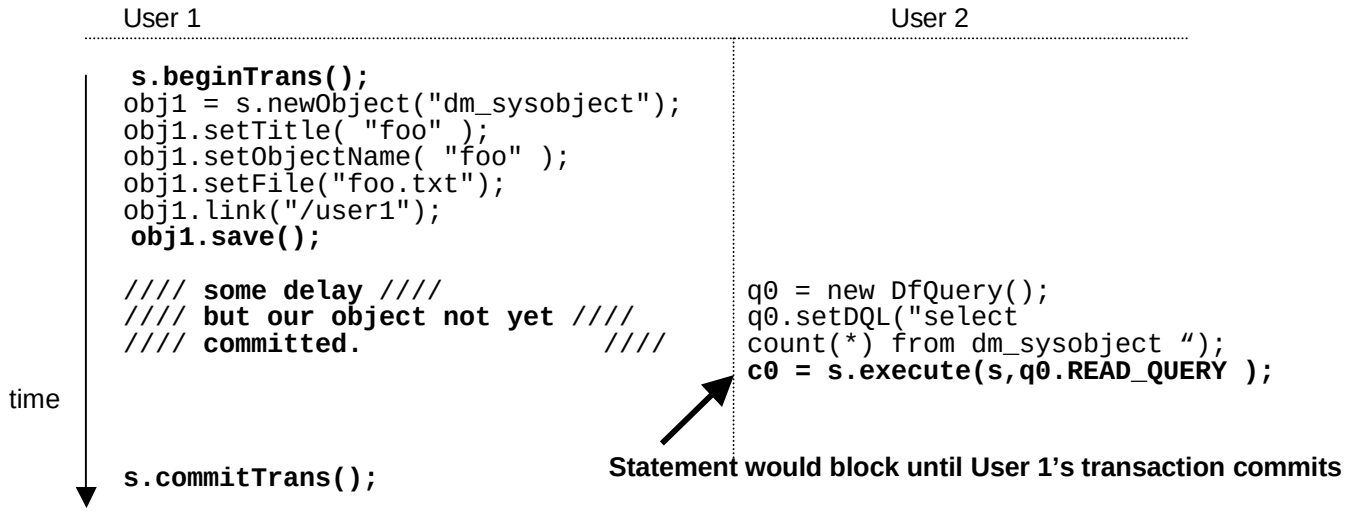


Figure 64: Example of Table Scan blocking with Non-Oracle RDBMS

It is instructive to examine the locking using the sp_lock command from SQL Server (illustrated in Figure 60) on this example. The important points include:

- First, large multi-step user defined transactions can lead to concurrency issues. This was a common issue for all of the examples shown so far. The longer a transaction takes to execute the more likely its bound to block other operations. There are two common techniques used to reduce the impact of large multi-step transactions:
  1. Pull out of the transactions queries that need not be performed within the transaction (this is called “shortening the critical path length”).
  2. Decompose the transaction or resources under transaction into smaller separate transactions.
- Second, the “select” actually “bumped into” the uncommitted record. The index scan of the query invited this behavior. Try to limit such queries from your application.
- Finally, an object create will insert into and update many tables and indexes. It provides plenty of opportunity to block other readers. The object bag technique described earlier might also reduce concurrency issues by reducing the number of items that are inserted into and updated during the on-line production hours. In this case the insert/update intensive operations happen in periods of low multi-user concurrency.

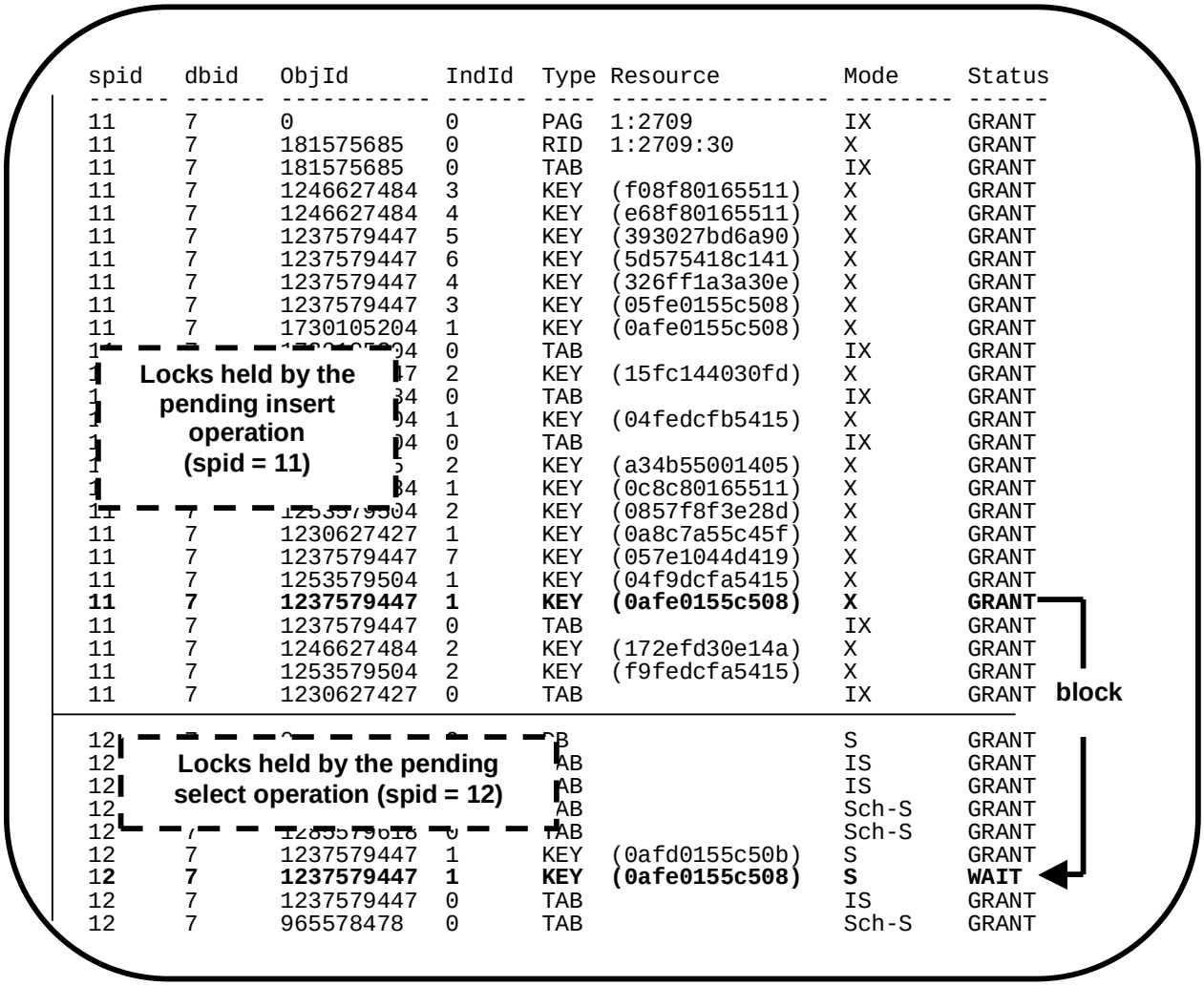


Figure 65: Sample SQL Server sp_lock output for the Insert example

Query Plans and Concurrency

The above examples illustrate how the access plan for the records can impact the concurrency performance of the application. The following items have been determined to have poor concurrency properties with non-Oracle databases:

- **Nested Loop Joins that pick the wrong join order**

  In this case the wrong join order implies that the “outer-table” of the nested loop join will cause the query to sift through more rows than if the table was chosen as the inner table. Since more rows are processed this not only leads to worse performance in isolation, but also leads to locking in concurrent user situations. The end result is the same, only the response time is lengthened.

  A nested loop algorithm operates roughly in the fashion outlined below.

```
Nested loop join of table A to table B on join column X and filter
columns A.Y and B.Z
(that is, select * from A, B where A.Y and B.Z and A.X = B.X )
Suppose  A is selected as the "outer table" and
         B is selected as the "inner table".
For all rows in A that match the initial selection criteria Y
   Get row from A
   For all rows in B that match the B join column X (A.X = B.X)
         Get a Row from B
         If B.Z = true then return row else discard
   Next B row
Next A row
```

Figure 66: Nested Loop Algorithm Outline

If the selection criteria for A.Y "selects" 1,000 rows 'A' on average,
A.X "selects" 10 rows from 'A' on average,
B.X "selects" 1 row from 'B' on average,
B.Z "selects" 4 rows from 'B' on average, then:

- If A is the "outer table" then 1,000 x 4 = 4,000 rows have to be processed
- If B is the "outer table" then about 40 rows will be processed.

That is, the columns (or attributes) on B are more selective than those of A and hence B should be the outer table.

- **Hash Joins and Merge Sort Joins**

Another important point about nested loop joins is that they are commonly the best join strategy for Documentum-based applications because the most common join that occurs is from the single-valued attribute table to the repeating value attribute table with r_object_id as the join column. This typically selects one or a few records from the single attribute table and a corresponding few records from the repeating attribute table.

There are other join algorithms than the nested loop described above. Under many conditions these will sift through more records than a nested-loop join for Documentum and hence have poor multi-user concurrency. The single user performance difference can be unnoticeable, but the multi-user performance can be much worse.

- **Table Scans**

Given the earlier examples it should be fairly obvious how a table scan will sift through many rows and have poor multi-user performance.

- **Indexes that are not selective enough or the wrong index**

As shown earlier, if an underlying index used in the query is not selective enough, then more records will be touched in the access operations than necessary. A good example of this would be in the earlier example when the object_name index selected the record with (bart, cat). A composite index on (object_name and title) would be a better, more selective index than one just on object_name in this case.

Strategies to Achieve and Maintain Optimal plans

The previous section outlined how non-optimal access strategies can lead to delays in large concurrent environments. The best ways to help ensure that plans are optimal include:

#1 - Check the plans ahead of time (prior to deployment, during application testing phases). There are two heuristics to this:

- Scale the number of objects to production levels. This will allow for testing of the underlying RDBMS optimizer and its ability to generate correct query plans in the presence of large data. In many cases also a bad plan will become apparent when a large amount of data has been loaded. Note that testing query performance does not require all objects to have content. Much disk space can be saved if a percentage of the test objects loaded are content-less.

- Check plans for the most frequently occurring queries.

Once sub-optimal plans have been identified then there are several options:

- Re-write the query to goad the optimizer in getting the correct plan
- Change the frequency of the query to ensure it is executed less often
- Add an index to make the plan optimal
- Force the selection of the proper index by defining a view (that forces the index) on the underlying _s and _r tables and register this view in the docbase.
- Program the statistics (Sybase) to generate the plan

#2 - Verify that the statistics job in the eContent server is run correctly and often.

RDBMS Deadlocking

A deadlock is an event within an RDBMS in which at least two separate threads (or users) are colliding over at least two data resources. The collision is the event that they need to lock both resources, however, they have only obtained half of the resource because the other thread (or user) has locked the other half. For instance, suppose two threads need to access and locks row in a table. Thread A has locked row 1 and is attempting to lock Row 4. Thread B has, however, already locked Row 4 and is attempting to lock Row 1. No progress can be made due to the existing exclusive locks and so the threads are in deadlock. Once the RDBMS detects this situation, it will rollback one of the thread's transactions to get out of the deadlock situation.

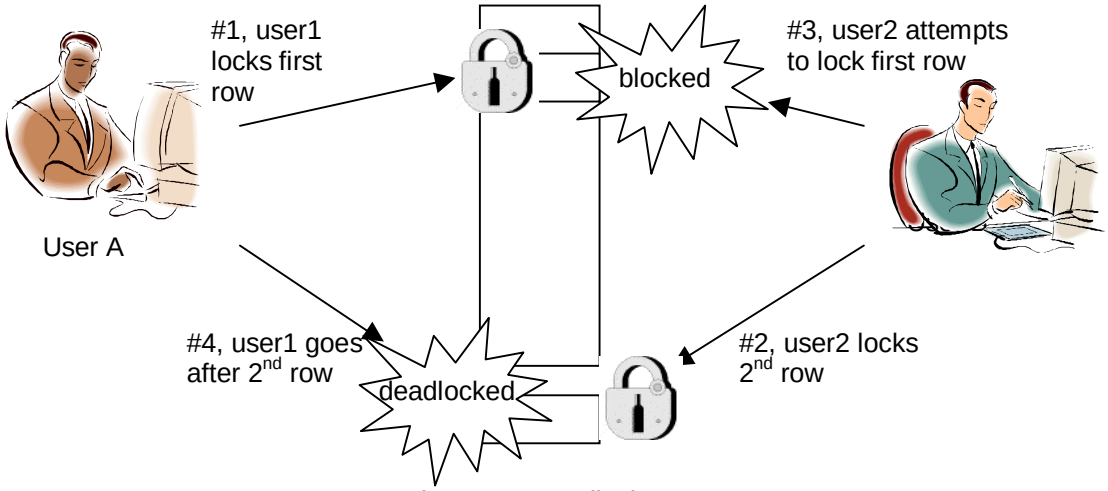


Figure 67: Deadlock Concept

The above example illustrates a deadlock in its most simple form. If 2nd resource that each thread is attempting to access is only for read purposes and the underlying RDBMS was Oracle then no deadlock would occur. Again, this is due to Oracle's multi-versioning concurrency control scheme that does not block readers. A deadlock would occur in Oracle only if the 2nd access is a modification attempt. The other RDBMS (Sybase, SQL Server, and DB2) would deadlock for read or modification access.

There are a couple of really important points to understand about deadlocks:

- They happen, don't assume they won't,
- They are typically caused by application design,
- Some can be designed "out" of an application, others cannot be avoided, and
- They can always be handled transparently.

In this section we:

- Illustrate how to code your DFC application to be deadlock tolerant
- Discuss how to diagnose a deadlock under Microsoft SQL Server
- Cover a couple of "real-world" case study examples

As mentioned before, once a deadlock occurs the RDBMS will rollback the transaction/operation of the thread that is chosen as a victim and return an error message back to the application. This victim gets an error return code and can now retry the transaction. If the user explicitly issued a beginTran() operation, then all of the modifications done after the beginTran() are rolled back. The process that is not a victim will just continue on with only a minor delay.

Detecting the deadlock at the application layer (DFC or DCML) is accomplished by parsing the error message returned by the eContent server. The message for SQL Server (and Sybase) will look something like:

```
STATE=40001, CODE=1205, MSG=[Microsoft][ODBC SQL Server Driver][SQL
Server]Your transaction (process ID #15) was deadlocked with another
process and has been chosen as the deadlock victim. Rerun your
transaction.
```

For Oracle it will look as follows:

```
[DM_SESSION_E_DEADLOCK]error:  "Operation failed due to DBMS Deadlock
error."
```

And for DB/2 it will be be shown as:

```
[DM_QUERY_E_CURSOR_ERROR]error:  "A database error has occurred during
the creation of a cursor (' STATE=40001, CODE=-911, MSG=[IBM][CLI
Driver][DB2/NT] SQL0911N  The current transaction has been rolled back
because of a deadlock or timeout.  Reason code "2".  SQLSTATE=40001').
```

Documentum has provided some deadlock retry logic within the EContent Server and it is enabled by default. However, this does not apply to all operations and it is best for application developers to not rely on it for any given sequence. The following code below describes how to handle or tolerate a deadlock in a fashion that is transparent for users. In all cases it is a matter of handling the deadlock in the exception handler for the function. For read-only operations, it is only a matter of rerunning the operation. For operations that involve some modification of the database, it would be necessary to ‘revert’ the object and reapply the changes before re-running the operation.

In our example we have two programs that attempt to change the title in two documents in reverse order. Such an example is a trivial one that could be avoided if they accessed the documents in the same order, however, as we will show later, there are many cases in which the programmer has little control over the access order for resources.

First, the base modification routine of our example is shown below. Again, the other thread is accessing object_name = x2 first and object_name = x1 second. A deadlock error is detected in the exception message. Once detected, the transaction is aborted (required) and the objects “reverted” (old changes discarded and refreshed with current Docbase values). The transaction abort is necessary to ensure that we can retry the operation without shutting down the session. Also in Oracle, unlike the other RDBMS, the “blocked (non-victim)” thread will not continue on making progress until the victim thread aborts the transaction.

The object ‘revert’ deserves a comment in that if the victim ‘reverts’ its value to the current docbase value before the ‘non-vicitim’ gets a chance to modify and save its results then the next attempt to execute this function by the victim is likely to lead to a “version mismatch” error (and hence, the operation should be re-tried again).

It is also noteworthy to point out where the different databases would actually block or get deadlocked. For oracle the reads are not blocked and so the deadlock doesn't occur until the 2nd save. For the other RDBMS the getObjectByQualification() causes the deadlock.

```
// sample deadlock clean code
public static int  update_objects( IDfSession session ) throws
IOException, InterruptedException, DfException{

    IDfId objId1;
    IDfSysObject obj1 = null;
    IDfId objId2;
    IDfSysObject obj2 = null;

    try {
        session.beginTrans();
        objId1 = session.getIdByQualification( "dm_sysobject where
                                                object_name = 'x1'");
        obj1 = (IDfSysObject ) session.getObject( objId1 );
        obj1.setTitle( "foo" );
        obj1.save();

        objId2 = session.getIdByQualification( "dm_sysobject where
                                                object_name = 'x2'");
        obj2 = (IDfSysObject ) session.getObject( objId2 );
        obj2.setTitle( "bar" );
        obj2.save();
        session.commitTrans();

        return 0;
    }
    catch (DfException e)
    {
        String msg = e.getMessage();
        if ( msg.toLowerCase().indexOf("deadlock") > 0 ||
             msg.toLowerCase().indexOf("version mismatch") > 0 ) {
            // we were deadlocked or ran into a version mismatch!!!
            session.abortTrans();
            if (obj1 != null ) obj1.revert();
            if (obj2 != null ) obj2.revert();
            return -1;
        } else {
            return -2;
        }

    }
}
```

Deadlock Occurs here for Sybase, SQL Server and DB2

Deadlock Occurs here for Oracle

If the "revert" occurs before the other thread can commit its changes then we are likely to run into a version mismatch error upon the next retry

Figure 68: update code example that handles deadlocks

The "calling" code is rather simple: just keep calling the function until it succeeds. This code is shown below. It assumes that the update_objects() function will take care of any cleanup after a deadlock occurs and that it is safe to retry the function.

```
while ( update_objects( session ) < 0 ) {

                retry++;
                if ( tracing)
                  System.out.println ("Failed " + retry + " time(s)");
                if ( retry > 4 ) {
                    System.out.println ("too many retrys, quitting");
                    break;
                }
                if (tracing)
                        System.out.println ("try again...");
}
System.out.println ("Succeeded");
```

Figure 69: Sample calling code

Actual production code would also protect against exceptions in the exception handling code for update_objects(). The reverts don't lead to additional deadlocks because they are not within a transaction and hence don't keep locks.

As is shown above there is only a small amount of code needed to make an application deadlock tolerant. Doing this extra work is a small price compared to the cost of trying to debug a deadlock situation that occurs in a multi-user test or production environment. The next sections give examples of the type of tools and work needed to debug code that is not deadlock tolerant. They are provided mainly to motivate the reader into making their code deadlock tolerant.

SQL Server tools for Deadlock analysis

Now, lets survey the most relevant tools for deadlocking. Our focus will be on the SQL Server, but the concepts covered are general for all of the databases. There are essentially two main tools for solving deadlock problems: 1204,1205 trace output and the SQL profiler.

1204,1205 DBCC TRACE Output

SQL Server will print out deadlock diagnostic information when the following trace flags have been enabled. To enable these trace flags do the following inside a SQL Server isql or Query analyzer tool window:

```
dbcc traceon(1204, 1205)
go
```

To disable this output do the following in a similar fashion:

```
dbcc traceoff(1204, 1205)
go
```

When a deadlock occurs these traceflags will print out some diagnostic information. The following is some output from a trace indicating a deadlock. We'll summarize the information now and then cover it in more detail later.

```
2000-02-17 11:49:21.37 spid2    *** Deadlock Detected ***
2000-02-17 11:49:21.37 spid2     ==> Process 16 chosen as deadlock victim
2000-02-17 11:49:21.37 spid2     == Deadlock Detected at: 2000-02-17 11:49:21.37
2000-02-17 11:49:21.37 spid2     == Session participant information:
2000-02-17 11:49:21.37 spid2     SPID: 12 ECID: 0 Statement Type: EXECUTE Line #:
1
2000-02-17 11:49:21.37 spid2     Input Buf:   s p _ c u r s o r o p e n     8
- - s e l e c t   r _ o b j e c t _ i d   f r o m   d m i _ q u e u e _ i t e m _
s   w h e r e   i t e m _ i d   =   ' 4 a 0 0 4 e 1 1 8 0 0 0 3 5 1 5 '   8
8      8
2000-02-17 11:49:21.37 spid2     SPID: 16 ECID: 0 Statement Type: EXECUTE Line #
-1
2000-02-17 11:49:21.37 spid2     Input Buf:   s p _ c u r s o r o p e n     &
cB   B   s e l e c t   d i s t i n c t   R . r _ o u t p u t _ p o r t   f r o m
d m i _ w o r k i t e m _ s   S ,   d m i _ w o r k i t e m _ r   R   w h e r e
S . r _ o b j e c t _ i d   =   R . r _ o b j e c t _ i d   a n d   S . r _ w o r
k f l o w _ i d   =   @ P 1     a n d   S . r _ a c t _ s e q n o   =   @ P 2
2000-02-17 11:49:21.37 spid2
2000-02-17 11:49:21.37 spid2     == Deadlock Lock participant information:
2000-02-17 11:49:21.37 spid2     == Lock: KEY: 7:1275151588:1 (8a03f72da6eb)
2000-02-17 11:49:21.37 spid2     Database: DM_AC_Accounting_docbase
2000-02-17 11:49:21.37 spid2     Table: dmi_workitem_s
2000-02-17 11:49:21.37 spid2     Index: d_1f004e1180000177
2000-02-17 11:49:21.37 spid2      - Held by: SPID 12 ECID 0 Mode "X"
2000-02-17 11:49:21.37 spid2      - Requested by: SPID 16 ECID 0 Mode "S"
2000-02-17 11:49:21.37 spid2     == Lock: KEY: 7:763149764:1 (b203bd628084)
2000-02-17 11:49:21.37 spid2     Database: DM_AC_Accounting_docbase
2000-02-17 11:49:21.37 spid2     Table: dmi_queue_item_s
2000-02-17 11:49:21.37 spid2     Index: d_1f004e118000016c
2000-02-17 11:49:21.37 spid2      - Held by: SPID 16 ECID 0 Mode "X"
2000-02-17 11:49:21.37 spid2      - Requested by: SPID 12 ECID 0 Mode "S"
2000-02-17 11:49:21.37 spid2
```

Note #1

Note #2

Note #3

Note #4

Notes on this output:

1. The statements shown represent the ones that are attempting to obtain locks. What is not shown in this dump are the statements that obtained the locks. In many cases this missing information is key to solving the deadlock problem. Note that both of these are actually read-only queries. The deadlock occurs because the statements that obtained the locks actually obtained exclusive locks. So by just looking at this output we can't tell what data modification statements were executed.

2. The line: `== Lock: KEY: 7:1275151588:1 (8a03f72da6eb)` is actually a terse way of identifying the table, index, and row being locked. `1275151588` identifies the table. This is the object id for the table that could be gotten from the SQL server sysobjects table (id). The number 1 that follows this (:1) indicates that this is occurring on the first index involved with the table. In this case this corresponds to the `d_1f004e1180000177` index defined by Documentum (this is a clustered index defined on r_object_id). The final value `8a03f72da6eb` represents an internal hash value created to represent the actual key value that is locked. Unfortunately, it is difficult (but not impossible) to map this hash value back to the actual row. We will discuss this later.

3. The terse information described in the above note is made more explicit by the next couple of lines:

```
2000-02-17 11:49:21.37 spid2     Table: dmi_workitem_s
2000-02-17 11:49:21.37 spid2     Index: d_1f004e1180000177
2000-02-17 11:49:21.37 spid2      - Held by: SPID 12 ECID 0 Mode "X"
2000-02-17 11:49:21.37 spid2      - Requested by: SPID 16 ECID 0 Mode "S"
2000-02-17 11:49:21.37 spid2     == Lock: KEY: 7:763149764:1 (b203bd628084)
```

   The table is identified (dmi_workitem_s) as well as the index involved ( ). In addition the next two lines indicate that SPID 12 had locked the key in exclusive mode (“X”) and that SPID 16 was trying to lock it in shared read mode (“S”). Database locking promotion is a fairly complex subject. However, for Documentum tables only a small subset matters and this small subset is fairly intuitive. That is, when a thread (or SPID, which stands for Server Process ID) locks a key (or row) in exclusive mode, then no other thread can lock it in any mode. When a row is locked in shared mode (“S”) then it can be locked by others in shared mode, but can’t be locked by a thread in exclusive mode. That is, a thread will block (sleep) if it attempts to lock in exclusive mode a row that is already locked in shared mode. It will sleep until the other thread releases this lock.

4. The final part of the output shows the other half of the locking story. Again, with a deadlock there has to be at least two threads (or SPIDs) and at least two resources (in this case keys, or rows). Notice that the hash value of the key in the second case (`b203bd628084`) is different from the one in the first case (`8a03f72da6eb`). They key question is WHAT ARE THOSE TWO ROWS??? This is important to understand in many cases because it may not be obvious from an inspection of the queries as to what rows intersect. In fact, by inspecting the query result sets, it may appear that there is no intersection of rows. Later we will go over a detailed example of this, and how to find the missing intersection.

SQL Profiler Output

This section describes how to use the SQL profiler to help diagnose and analyze deadlocks. The SQL profiler is a tool that captures event information. There are many types of events, but for the purpose of this discussion, the default ones (which include "statement events") are all that is important. That is, this tool captures the sequence of commands being sent from the Documentum Server threads to the SQL Server. It displays, if you will, the record of the conversation between the SQL Server and Documentum.

Capturing a Profile

The first task is to capture a SQL profile

To start the profiler select Start→Programs→SQL Server 7.0→Profiler. Then within the profiler select File→New→ Trace and the new trace dialogue appears (as shown below). Give the trace a name and also select the "Capture to File" option so that no trace information is lost. Finally, select the "Events" tab of this dialogue and add deadlocking events (a sub event of locking). See below.

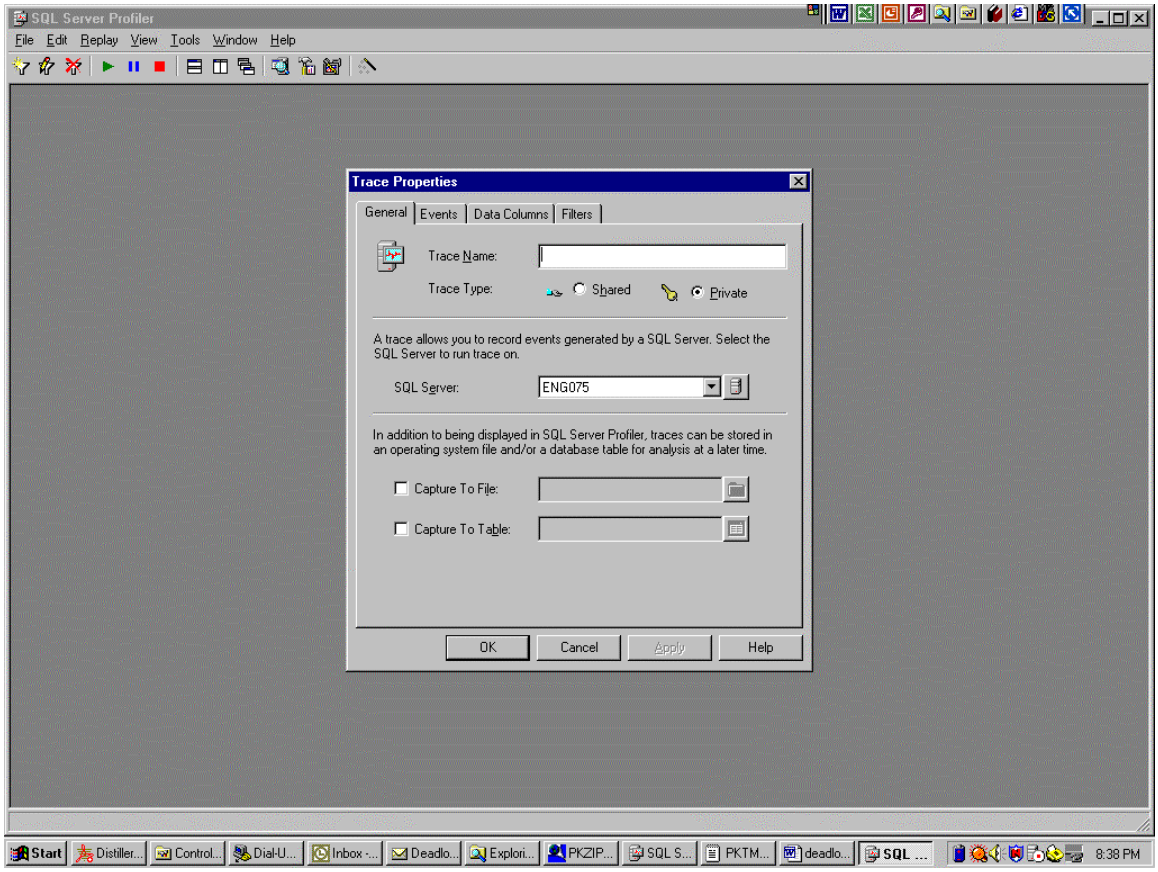


Figure 70: Starting a profile

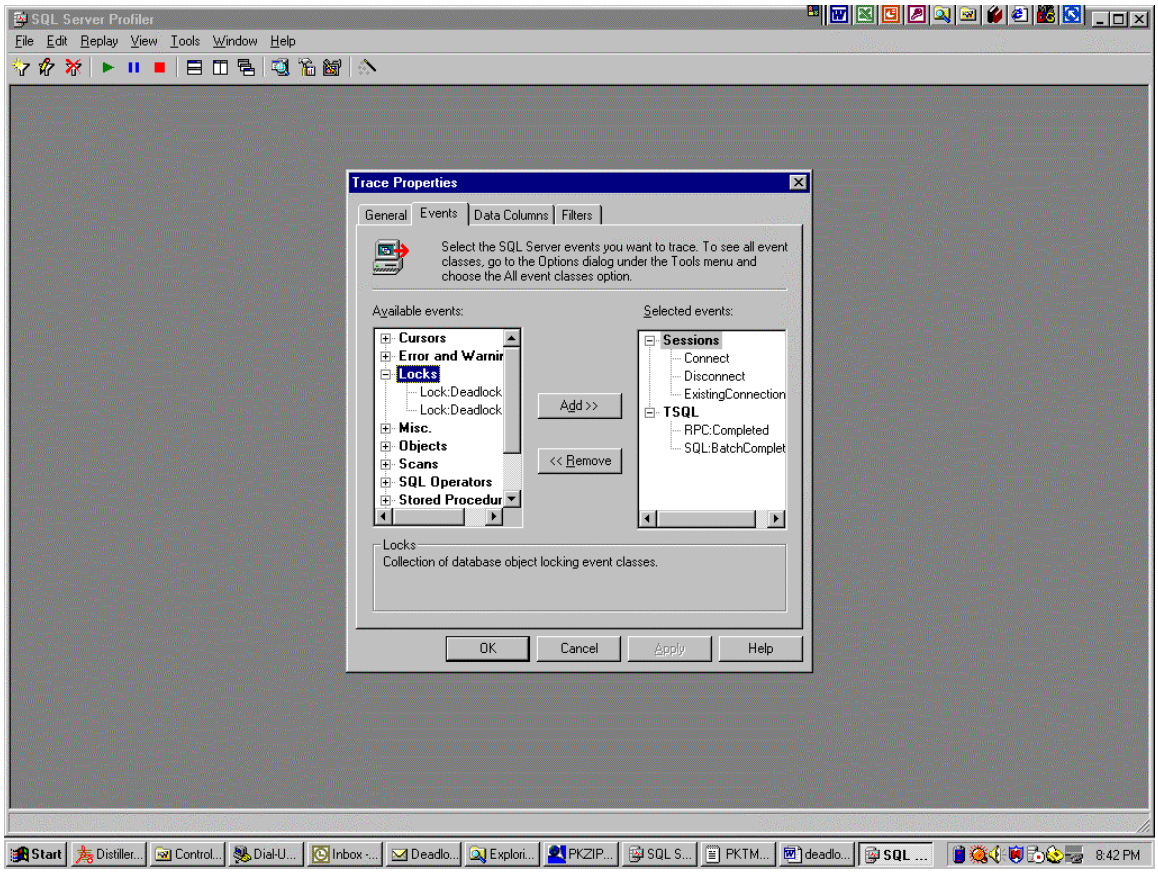


Figure 71: Selecting deadlock events for profile

Once this has been done then press the OK button and attempt to reproduce your deadlocking event.

Loading and Analyzing a Trace

Once a trace has been collected then it can be examined later. It is important to understand the sequence of calls because a deadlock always involves more than one locking statement and the error log trace output will only contain the queries/or statements that attempted to get locks (causing the deadlock event). As we mentioned earlier it won't contain the statements that actually locked the resources. In many cases it can be quite unclear as to what was happening unless the profile is obtained along with the errorlog deadlock information.

Now lets briefly examine the important fields of the trace output and describe how to manipulate a saved trace. An example trace is shown below. There are several fields in this display that are important to note:

- Event

  This column describes the event that was captured. For the most part the most important event is the RPC:Completed event. The Text column of this type of event includes the actual SQL that was run against the server. The Lock:deadlock events are also important because they indicate which threads were involved in a detected deadlock.

- Text

  The text column contains the information on what type of operation was run. This includes commands to run SQL, update, insert, and delete statements, and cursor operations. This is

the field that is used to determine what operation might have locked the resource in the first place.

- Duration

  This field is most useful for finding long running queries (not so important for deadlocks).

- SPID

  This field contains the server process id of the thread (or user) that executed the event. This can be correlated to the SPID listed in the SQL Server errorlog deadlock information.

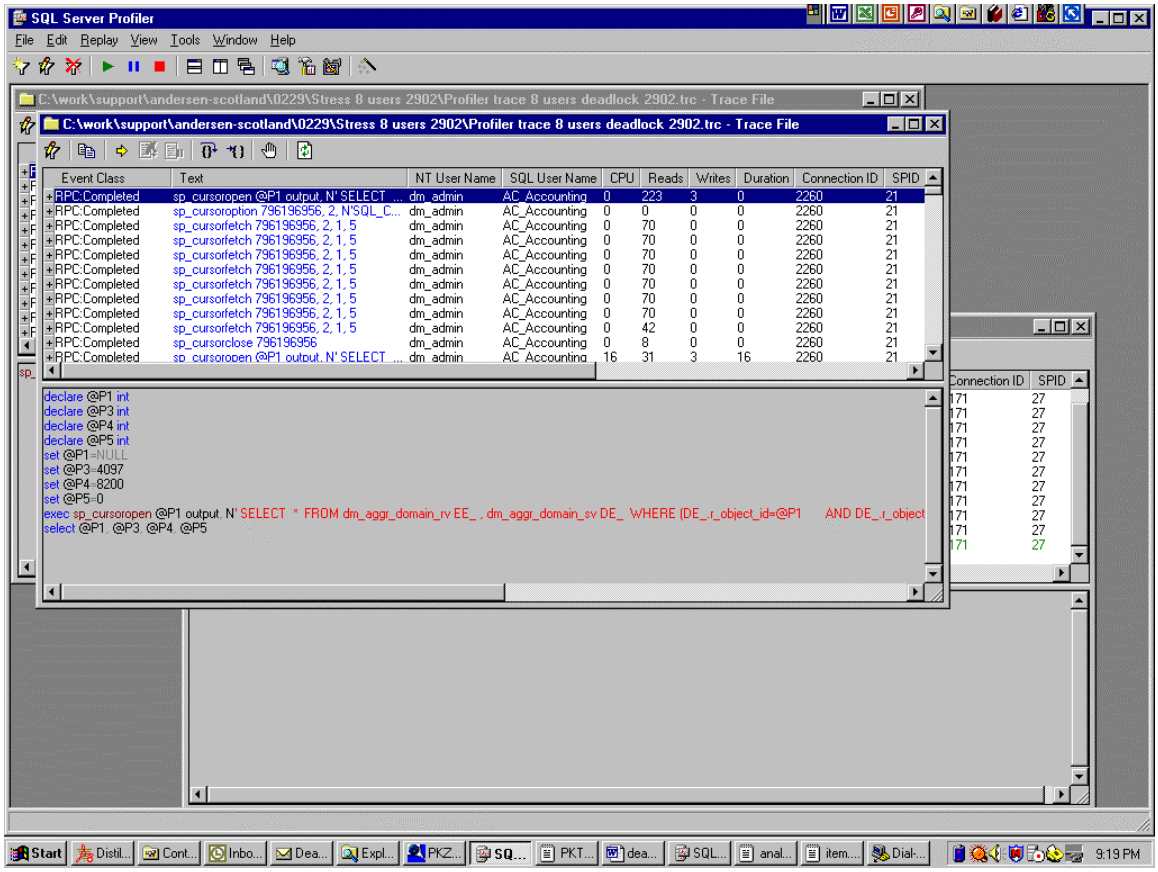


Figure 72: Profile output

Once the trace has been loaded it is also possible to create various views of the data by changing the properties of the profile. We can filter and display based on the values of the columns. For instance we can create a view of this trace that has only the data for the SPID 23 or perhaps we can include only data that is associated with UPDATE statements. To make these changes go to into effect bring up the properties dialogue for the trace and modify the values based on the correct tab in this dialogue.

Once a profile has been filtered by SPID, then were does one start to look for the most interesting queries? It is best to start at the “end”, that is, when the deadlock occurred and then work back in time to find the locking query. To find the “end” it is necessary to look for the deadlocking event in the profile. Select Edit→Find and in the dialogue that appears set the “Search Value” to ‘deadlock’

and the “Data Column” to ‘Event’. This will locate the deadlock events that were stored in the profile.
This deadlock event could have occurred quite far into a trace. To find the “starting point” of the relevant data we need only go back as far as the commit of the last transaction. Usually, there are not too many operations between the deadlock event and the last commit (for a particular filtered SPID). The commit of the last transaction will nearly always look as follows in the Text field:

IF @@TRANCOUNT > 0 COMMIT TRAN

For the thread under consideration, the deadlock has to occur between this last commit and the deadlock event. Note that Documentum uses implicit transactions. That is, the transaction will begin at the first data modification start (i.e., the first Update, Insert, or Delete). Now that the events have been “narrowed down” it is important to notice a few things:

a. Of the exclusively locked tables, how are they modified?
b. What is the order of read access and exclusive access?
c. What columns are updated? (especially for the tables associated with the deadlock)

We will cover this in more detail next. Note that having the SQL Profile is insufficient to solve the problem because the deadlock events in the profile do not indicate all of the diagnostic information as the ERRORLOG output does. However, unlike the ERRORLOG the actual “locking” statements are buried somewhere within the trace. One just has to find them.

Deadlock Examples

In this section we cover some deadlocks that are based on actual examples.

Deadlock Due to Index Update

The following section covers an example of a deadlock due to the need to update an index used by another query. An index can be a composite of several columns. If the row pointed to by one of those index entries is updated on one of those composite columns then that index will have to be updated. In this example an index was added on the dmi_queue_item_s table to optimize a customer-customized query. The index had the following definition:

```
create nonclustered index queue_test_1 on dmi_queue_item_s (    name, task_name,
router_id )
go
```

This index was the best to optimize the running of the customized Documentum Inbox query. Unfortunately, it has some concurrency issues. That is, if the column task_name is ever updated for a given row by any other query, then the index queue_test_2 will need to have that entry updated. To update that entry the index row needs to be locked exclusively.

How do we see this from the data? In this case the deadlock output looked as follows:

```
2000-02-23 09:53:39.40 spid2    *** Deadlock Detected ***
2000-02-23 09:53:39.40 spid2     ==> Process 19 chosen as deadlock victim
2000-02-23 09:53:39.40 spid2     == Deadlock Detected at: 2000-02-23 09:53:39.40
2000-02-23 09:53:39.40 spid2     == Session participant information:
2000-02-23 09:53:39.40 spid2     SPID: 19 ECID: 0 Statement Type: EXECUTE Line #:
0
2000-02-23 09:53:39.40 spid2     Input Buf:   s p _ c u r s o r o p e n     8
ç    s e l e c t   d i s t i n c t   q . s e n t _ b y ,   a . o b j e c t _ n a
m e ,   q . d a t e _ s e n t ,   q . t a s k _ n a m e ,   q . d u e _ d a t e ,
q . p r i o r i t y ,   q . t a s k _ s t a t e ,   q . r _ o b j e c t _ i d ,
q . i t e m _ i d ,   q . i t e m _ n a m e ,   q . e v e n t ,   q . d u e _ d a
t e ,   q . d a t
2000-02-23 09:53:39.40 spid2     SPID: 22 ECID: 0 Statement Type: UPDATE Line #:
1
```

```
2000-02-23 09:53:39.40 spid2     Input Buf:   s p _ e x e c u t e s q l     cî
î    U P D A T E   d m i _ q u e u e _ i t e m _ s     S E T   t a s k _ s t a t
e = ' a c q u i r e d ' , i _ v s t a m p =      2   W H E R E   ( r _ o b j e c
t _ i d = @ P 1             A N D   i _ v s t a m p = @ P 2             )
c.   .   @ P 1   v a r c h a r ( 1 7 ) , @ P 2   i n t   §    1b004e11800092ab  8
2000-02-23 09:53:39.40 spid2
2000-02-23 09:53:39.40 spid2     == Deadlock Lock participant information:
2000-02-23 09:53:39.40 spid2     == Lock: KEY: 7:763149764:12 (bc11f8c86228)
2000-02-23 09:53:39.40 spid2     Database: DM_AC_Accounting_docbase
2000-02-23 09:53:39.40 spid2     Table: dmi_queue_item_s
2000-02-23 09:53:39.40 spid2     Index: queue_test_1
2000-02-23 09:53:39.40 spid2      - Held by: SPID 19 ECID 0 Mode "S"
2000-02-23 09:53:39.40 spid2      - Requested by: SPID 22 ECID 0 Mode "X"
2000-02-23 09:53:39.40 spid2     == Lock: KEY: 7:763149764:1 (e40351b972ac)
2000-02-23 09:53:39.40 spid2     Database: DM_AC_Accounting_docbase
2000-02-23 09:53:39.40 spid2     Table: dmi_queue_item_s
2000-02-23 09:53:39.40 spid2     Index: d_1f004e118000016c
2000-02-23 09:53:39.40 spid2      - Held by: SPID 22 ECID 0 Mode "X"
2000-02-23 09:53:39.40 spid2      - Requested by: SPID 19 ECID 0 Mode "S"
2000-02-23 09:53:39.40 spid2
```

The deadlock output indicates that two indexes on the dmi_queue_item_s table are participating in this deadlock. The first is the queue_test_1 and the second is the default cluster index for the table `d_1f004e118000016c` (which is clustered on the r_object_id column). After looking at this output the following questions should be asked:

- Why is SPID 19 (a reader) attempting to access a row in the clustered index (`d_1f004e118000016c`)?
- Should SPID 19 be accessing the index queue_test_1?
- Why does SPID 22 need to access queue_test_1?

Let us answer these in order to unravel this deadlock:

Why is SPID 19 (a reader) attempting to access a row in the clustered index (`d_1f004e118000016c`)?

In SQL Server 7, if there is a clustered index on a table then the non-clustered indexes won't contain actual pointers to a row's position on a page, rather the pointer is a key to the clustered index that can be used to locate the row. More importantly for this example, the leaf level of a clustered index is the actual data itself. So any access to the data row itself is considered an access to the leaf level of the clustered index.

The reader is executing the following "inbox" query:

select distinct q.sent_by, a.object_name, q.date_sent, q.task_name, q.due_date, q.priority, q.task_state, q.r_object_id, q.item_id, q.item_name, q.event, q.due_date, q.date_sent, q.sent_by, q.message, q.task_number, q.task_state, q.item_type, q.item_id, q.content_type, q.stamp, q.router_id, q.instruction_page, q.priority, q.r_object_id, q.a_content_type, q.task_name from dbo.dmi_queue_item_sp q, dbo.dmi_package_sp p, dbo.aca_sap_document_sp a where (q.name='dmtest_acascanner1' and (q.task_state='dormant' or q.task_state='acquired' or q.task_state='ready') and (q.router_id!='0000000000000000' and q.router_id != ' ') and q.delete_flag=0 and (a.dispatch_group_check='dmtest_acascanner1' or a.dispatch_group_check='') and q.router_id = p.r_workflow_id and p.r_object_id in (select all r_object_id from dbo.dmi_package_r where (r_component_id=a.r_object_id))) order by 3 desc, 6 desc, 2 desc

We've underlined the references to the dmi_queue_item_s table to show that this query needs many (if not most) of the columns from that table. There is no "covering" index to return all of these fields (nor would it add much value to provide one given that there are so many fields being returned).

Hence, the reader has to get a shared lock on specific keys in the clustered index even if the keys are located from the index queue_test_1.

Should SPID 19 be accessing the index queue_test_1?

The SQL Server 7 product has a nice query analysis tool for looking into the plans for queries. It will diagram the plan in a "graph" form with icons and descriptive text. Unfortunately, one can't save that plan form in any convenient way (other than a screen dump which is a really poor way to show it). The other alternative is to use the SHOWPLAN_ALL option. Prior to executing the query in isql run the following command:

set SHOWPLAN_ALL on
go

This gives a textual (although extremely complicated) representation of the query plan. An excerpt from the plan for this query is shown below.

```
select distinct q.sent_by, a.object_name, q.date_sent, q.task_name, ….
  |--Sort(DISTINCT ORDER BY:([dmi_queue_item_s].[date_sent] DESC, [d….
       |--Merge Join(Inner Join, MERGE:([dm_sysobject_s].[r_object_i….
            |--Index Scan(OBJECT:([Andersen].[dbo].[dm_sysobject_s]….
            |--Nested Loops(Inner Join)
                 |--Sort(ORDER BY:([dmi_package_r].[r_component_id] …..
                 |    |--Merge Join(Inner Join, MANY-TO-MANY MERGE:(….
                 |         |--Sort(ORDER BY:([dmi_package_s].[r_obje…
                 |         |    |--Nested Loops(Inner Join)
                 |         |         |--Filter(WHERE:([dmi_queue_ite….
                 |         |         |    |--Bookmark Lookup(BOOKMAR….
                 |         |         |         |--Filter(WHERE:([dmi….
                 |         |         |              |--Index Seek(OB….
                 |         |         |--Index Seek(OBJECT:([Andersen….
                 |         |--Stream Aggregate(GROUP BY:([dmi_packag…
                 |              |--Index Scan(OBJECT:([Andersen].[db…
                 |--Index Seek(OBJECT:([Andersen].[dbo].[aca_sap_doc…
```

The "…. " on the right hand side of the above means that we've essentially cut off the right hand part of this output because it would not fit within the margins of this document. Since this information scrolls off so far to the right, it also is not very convenient for saving and viewing.

Lets focus on the inner part that indicates the usage of that index. The output is broken up into several columns. A column called "LogicalOp" indicates that an index seek was done on the dmi_queue_item_s table and the argument for this seek is shown in the next column as:

```
OBJECT:([Andersen].[dbo].[dmi_queue_item_s].[queue_test_1]),
SEEK:([dmi_queue_item_s].[name]='dmtest_acascanner1' AND
[dmi_queue_item_s].[task_state] BETWEEN 'acquired' AND 'acquired' OR
[dmi_queue_item_s].[name]='dmtest_acascanner1' AND [dmi_queue_item_s]
```

This shows that name, task_state, and router_id are used in the initial scan of records from the dmi_queue_item_s table. Hence, this index, which was added in this example, appears to be useful for the query.

So to answer the above question we need to know how SPID 22 was accessing records in the dmi_queue_item_s table.

Why does SPID 22 need to access queue_test_1?

But with a deadlock we always need two threads and two resources. The first resource appears to be a row within the dmi_queue_item_s table. Since the writer (SPID 22) is updating 'task_state' it will definitely have to lock that row in the dmi_queue_item_s table. But, in addition, it will have to lock any row in an index that uses that column as a key. In this case queue_test_1 also has 'task_state' as one of its columns, and so needs to be exclusively locked.

Now the deadlock appears to be understood. The reader accesses the queue_test_1 index to find some record X. It locks that index entry in shared mode. It then attempts to access the actual data row for record X. The writer has already locked record X to in exclusive mode (to update task_state) and now needs to lock the row in queue_test_1 pointing to this row so it can update that index row as well. This leads to a deadlock.

How can this particular deadlock be modified? Clearly if we take away the writer's need to access queue_test_1 then we won't have two resources to deadlock on. A simple approach will be to delete the queue_test_1 index and create another one without the task_state. That is, create a new index as follows:

```
create nonclustered index queue_test_1 on dmi_queue_item_s (   name, router_id )
go
```

But what will be the impact of this index on the query response time? In this case we ran the query three times with different indexes (the original one, the first queue_test_1 definition, and the new queue_test_1 definition above). Their response times are shown below:

Index used	Time to execute query	Logical io's to dmi_queue_item-s
Original (router_id)	1.435 secs	86,312
(name, task_state, router_id)	0.884 secs	327
(name, router_id)	0.912 secs	1,963

Hence, our original queue_test_1 index (on [name, task_state, router_id]) was the best index for this query, however, it lead to concurrency problems (deadlocks). By taking out task_state, we only slightly degrade the performance of the query (by 3%) and get rid of the concurrency problem.

This example is an important lesson on how indexes could be chosen to optimize for a particular query in isolation, and yet could lead to performance problems later in a multi-user environment.

Deadlocks Caused by Initial Scans That Aren't Selective Enough

The next most likely cause of a deadlock is from a lower level scan that requires more rows than expected to be accessed and locked. When the query normally returns it returns just a few rows, however, when the query is executing it's initial scan picks up more records than expected. Later "filters" are applied on those returned records to get to the small number of returned rows. Those initial scanned rows have to be locked and this leads to unexpected deadlocks with other users that appear to have no rows in common with the rows generated by this user's query.

This can be illustrated using the query shown earlier. If we take the custom index off of dmi_queue_item_s then that query will use an index on router_id to do the initial scan of dmi_queue_item_s. Since this index is not very selective it will generate a large rowset in this initial scan. These rows will be later "filtered out" using the rest of the criteria in the WHERE clause, but

until that occurs the returned rows have shared locks on them. This allows them to get into deadlocks with other operations that are working on a seemingly unrelated set of data. This is illustrated below.

For example, lets take the query from the previous example and whittle it down to a single scan of dmi_queue_item_s using the parts of the WHERE clause that can be used by an index. Suppose we had the index on (name, task_state, router_id). All components of this index will be used to locate rows in the following query:

```
select distinct count(*)  from dbo.dmi_queue_item_sp q where
(q.name='dmtest_acascanner1' and  (q.task_state='dormant' or
q.task_state='acquired' or q.task_state='ready')  and
(q.router_id!='0000000000000000' and q.router_id != ' ')       )
```

This query generates 76 rows on our sample database. An initial scan using such an index would lock 76 rows.

If we strip out the references to task_state in this query, then it will generate a rowset that is equivalent to one on with an initial scan of the index on (name, router_id):

```
select distinct count(*)  from dbo.dmi_queue_item_sp q where
(q.name='dmtest_acascanner1' and   (q.router_id!='0000000000000000' and
q.router_id != ' ')       )
```

This generates 637 rows from our sample database. That is, the additional 561 rows locked are some which could potentially deadlock with other update operations. These rows are later unlocked after subsequent filters are applied (namely the filter: `q.task_state='dormant' or q.task_state='acquired' or q.task_state='ready'`). But given the nature of the updates occurring, deadlocks are unlikely to occur with these extra rows.

However, suppose no custom index is being used and we have one with just (router_id). Then the following query:

```
select distinct count(*)  from dbo.dmi_queue_item_sp q
(q.router_id!='0000000000000000' and q.router_id != ' ')
```

generates 29,000+ locked rows in the initial scan making it very likely to cause deadlocks with other tasks. Again, once the filters are applied all but 76 will be unlocked, but in that window of initial scan and filter a deadlock could occur.

The solution is to attempt to ensure that the initial scan is as selective as possible. But notice that given the earlier case, the most selective index actually caused deadlock problems, so this is a heuristic, not a hard fast rule.

Deadlocks Caused by clashes between Inserts and Index Scans

Another possible deadlock occurs when the writer inserts a record (uncommitted) into a table (or index). The reader is scanning that index or table and needs to determine if that record is one that could fulfill its query. Now the uncommitted record can not possibly fit the query of the reader (by definition because it is uncommitted), but in SQL Server the reader only sees an exclusive lock on the row and does not know that it is an uncommitted record – hence it blocks.

Appendix A: Tracing Strategies for Documentum Applications

In this section we'll cover some of the tracing strategies that are useful in debugging performance problems. This will include enabling trace and some workflow/lifecycle specific techniques.

The DMCL Trace

Once it has been determined that a command has poor response time, the next most important task is to attempt to "sectionalize" the performance issue to client (desktop client or application server) or server software (eContent server or RDBMS). When developing an application, one "best practice" is to instrument an application such that the user perceived response time can be measured. This, however, may not always be possible, because, for example, one might only be customizing an existing application like the Documentum Desktop Client.

In such cases, Documentum provides some application API measuring capabilities that are sometimes useful in understanding why a particular command is not running as fast as desired. When this API tracing is enabled each DMCL command is written to a file along with its response time. This can look as follows:

```
# Time:        0.000 sec  'OK'
API> fetch,s0,0c246fa6800004c1
# Time:        0.601 sec  'OK'
API> get,s0,0c246fa6800004c1,i_has_folder[0]
# Time:        0.000 sec  'T'
API> readquery,s0,select "object_name", "a_content_type", "r_object_type",
"r_lock_owner", object_name, r_object_id, a_content_type, r_object_type,
r_lock_owner, r_link_cnt, isreplica,"object_name" from dm_sysobject where
folder(id('0c246fa6800004c1')) and a_is_hidden = false order by
"object_name",r_object_id
# Time:        6.699 sec  'q0'
API> count,s0,q0
# Time:        0.000 sec  '12'
API> get,s0,q0,_names[0]
# Time:        0.000 sec  'object_name'
```

Note: Command response time

Each line that begins with "API>" represents the command to sent via DMCL to the eContent Server and each line beginning with the "# Time:" represents the response time for that command in seconds. Hence in the case above most of the commands took less then 10 msec, except for the readquery command, which took almost 7 seconds.

If one or more queries are the problem, then this is typically the best way to identify them. This trace is also useful in determining whether the performance bottleneck is the software acting as a Documentum client or the Documentum eContent server/DBMS server. That is, given some scenario that maps to a series of DMCL calls, one need only compare the user perceived response time with the sum of all of the response times in the trace. For example, in the trace above the sum of all response times in the trace fragment is 7.3 secs. If the user perceived response time is 10 seconds, then our readquery method above is the major contributor and the solution will require some attention on the server machine. If the dmcl response times had added up to 2 seconds then our efforts would be better served by concentrating on the Application server or client side processing (or the networking between the systems).

FIGURE 73: How DMCL trace can help identify source of poor response time

Once a problem scenario has been identified as being either a client or a server issue, then more tools can be brought to bear to determine exactly what causes the problem. In this document we will survey many tools that can be useful in determining server-side performance issues. For client side performance issues it is typically best to employ software profilers (like the Numega TrueTime or Rational's Visual Quantify).

Enabling DMCL tracing

This API tracing is enabled in several ways:

1. By issuing the "trace" method to the server (see Documentum Reference Manual). For example at the IAPI prompt:

   ```
   IAPI> trace,c,10,'mytrace.trc'
   ```

   Which, in the above example, turns on the trace and directs the output into the file mytrace.trc.

   This API command can be issued from any language that can make DMCL API calls. For example, from Docbasic:

   ```
   Ret = dmAPIExec("trace,c,10,'mytrace.trc'")
   ```

   Or in DFC as:

   ```
   Session.traceDMCL( 10, "mytrace.trc" );
   ```

   In addition, the Documentum Desktop Client one can enable tracing in several ways:

   - By selecting the Help (menu) → Documentum Help (item) and pressing the CTRL key at the same time. When this is done an "interactive message testing" dialogue appears. The trace API method can be invoked in the same manner as in the IAPI from this dialogue. Once invoked, all Desktop client interactions will be traced until the end of the session or until the interactive message tester is brought up again and the trace disabled.
   - By selecting Properties → Advanced tab for the Desktop Client icon. For this to take affect for the Explorer integration it is necessary to either toggle between Online/Offline/Online or log off the Windows shell and then Log back in.

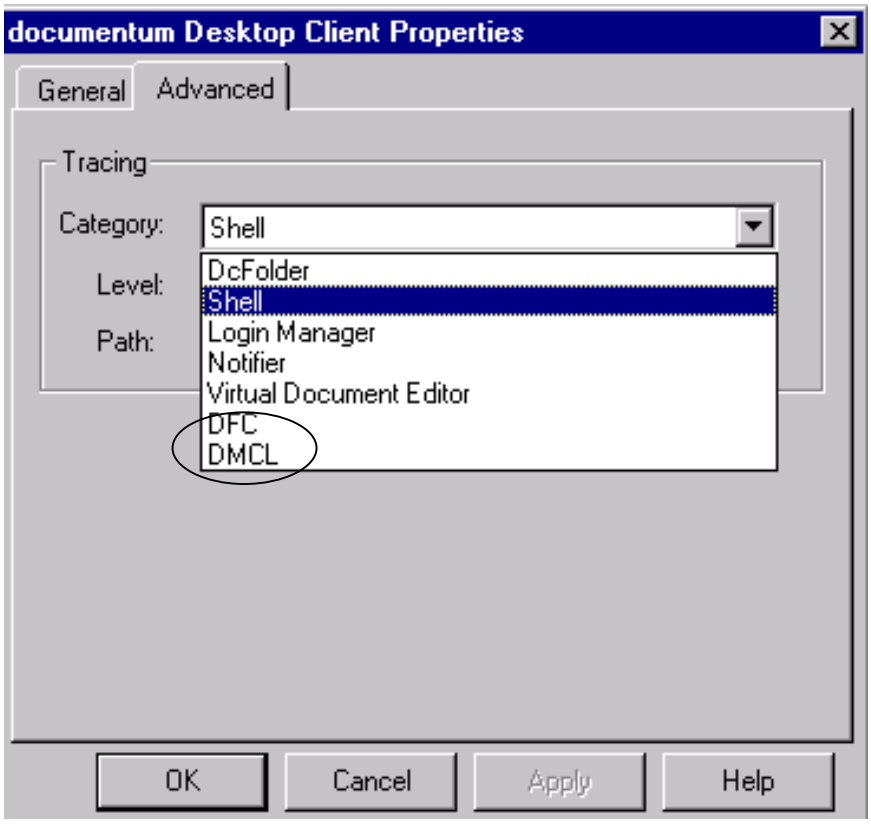


Figure 74: Enabling DMCL tracing through the Desktop Client Properties dialog

To disable tracing in general just re-issue the trace method with a level of 0:

```
trace,c,0
```

2. The tracing can also be enabled by turning it on via the client dmcl.ini file:

```
[DMAPI_CONFIGURATION]
trace_level = 10
trace_file = "c:\mytrace_folder"
```

The tracefile can either be a file or a directory. If it's a directory then a new (randomly named) trace file will be created in the directory for every dmcl session that is created.

This method is useful when the startup of an application needs to be examined or it is difficult to get the application to issue the trace method. For example, this is probably the easiest way to collect this trace with the Documentum RightSite server or with an Application server environment.

Tracing, however, is not always straightforward. Some Documentum server API's can be customized by user-defined scripts. These scripts need to enable tracing explicitly else the bottleneck might not be easily identified. For example, the eContent server can execute customized lifecycle and workflow scripts in the background. In those cases it is best to be able to enable tracing in the script itself, or to turn on the tracing on the eContent server machine using the

dmcl.ini method outlined above. Care must be taken to match up the dmcl trace files of the client machine with the ones generated on the eContent server machine.

Analyzing a DMCL Trace

Once a dmcl trace file is available it is in many cases useful to load this into a spreadsheet in order to further analyze it. Since the default format of the DMCL trace is not spreadsheet friendly it is necessary to perform a series of processing steps to move it into this format. This can be done using some filter language like awk or Perl. The following 2-line awk program will modify the dmcl file output so that each line contains two columns. The first column is the response time and the second is the DMCL command. A tab character separates the two columns.

```
/API>/ { cmd = $0 }
/# Time:/ { printf("%s\t%s\n", $3, cmd ); }
```

Figure 75: Two Line Awk program to reformat dmcl trace for spreadsheet

Lets suppose the above two-line program is in a file called sort.awk. Then to run this on a tracefile (called mytrace.txt) one would execute it in the following manner:

```
dmawk32 –f sort.awk < mytrace.txt     > mytrace.out
```

or

```
awk –f sort.awk  < mytrace.txt    > mytrace.out
```

Hence our trace shown earlier would look as follows.

```
0.601  API> fetch,s0,0c246fa6800004c1

0.000  API> get,s0,0c246fa6800004c1,i_has_folder[0]

6.699  API> readquery,s0,select "object_name", "a_content_type", "r_object_type",
"r_lock_owner", object_name, r_object_id, a_content_type, r_object_type,
r_lock_owner, r_link_cnt, isreplica,"object_name" from dm_sysobject where
folder(id('0c246fa6800004c1')) and a_is_hidden = false order by
"object_name",r_object_id

0.000  API> count,s0,q0

0.000  API> get,s0,q0,_names[0]
```

Figure 76: reformatted trace

Which is a format that can easily be loaded into a spreadsheet (pick Delimited option, not fixed width while in the import wizard). The data can now be sorted by the first column to find the most expensive operations.

One important note, however, is that at times some dmcl operations will generate multiple lines of output. These are cumulative. For example, multiple lines of trace are generated when an object is saved and it has several content pages. In the example below the time for the save is 594 msecs. The 141 msecs is an intermediate timing.

```
API> save,s2,090d3ca680073578
# Network Requests: 333
# Time:        0.000 sec  '>>>Using data server session to retrieve content <<<'
# Time:        0.141 sec  '>>>Using data server session to retrieve content <<<'
# Time:        0.594 sec  'OK'
```

Appendix B: IDCDataDictionaryUtils

Public Methods Accessible from IdcDataDictionaryUtils Interface:

InitAllObjectTypes (IDispatch* pSessionDisp, BSTR docbaseName)
Purpose:
Initialize data dictionary cache with the available object types and their display names from a docbase, whose name is <docbaseName>. This method reads client side permanent cache if this client cache is still valid. Otherwise, this method queries the docbase for the latest information.

InitAllFormats (IDispatch* pSessionDisp, BSTR docbaseName)
Purpose:
Initialize data dictionary cache with the available formats and their display names from a docbase, whose name is <docbaseName>. This method reads client side permanent cache if this client cache is still valid. Otherwise, this method queries the docbase for the latest information.

GetAllObjectTypes (BSTR docbaseName, /*[out]*/ VARIANT *allObjectTypes)
Purpose:
Returns the display names for all object types available in a docbase, whose name is <docbaseName> in a safe array, named <allObjectTypes>.

GetAllFormats (BSTR docbaseName, /*[out]*/ VARIANT *allFormats)
Purpose:
Returns the display names for all formats available in a docbase, whose name is <docbaseName> in a safe array, named <allFormats>.

GetObjectTypeDisplayName (IDispatch* pSessionDisp, BSTR objectType, /*[out, retval]*/ BSTR* pDisplayName)
Purpose:
Returns the display name for an object type in a docbase whose session is <pSessionDisp>.

GetFormatDisplayName(IDispatch* pSessionDisp, BSTR internalName, /*[out, retval]*/ BSTR* pDisplayName)
Purpose:
Returns the display name for a format in a docbase whose session is <pSessionDisp>.

GetAttributeDisplayName (IDispatch* pSessionDisp, BSTR objectType, BSTR internalName, /*[out, retval]*/ BSTR* pDisplayName)
Purpose:
Returns the display name for an attribute of an object type in a docbase whose session is <pSessionDisp>.

GetFormatCanIndex (IDispatch* pSessionDisp, BSTR internalName, /*[out, retval]*/ VARIANT_BOOL* pCanIndex)
Purpose:
Returns TRUE or FALSE value on whether a certain format in a docbase can be full-text indexed.

Sample Code 1:

```
Dim dataDictUtil as DCUTILMGRLib.DcDataDictionaryUtils
Dim objectTypes as Variant
Dim numTypes as Integer
Dim formats as Variant
Dim numFormats as Integer

‘ First, create Data Dictionary Utility Manager.
Set dataDictUtil = new DCUTILMGRLib.DcDataDictionaryUtils

‘ Initialize object types cache, then retrieve all object types display
name.
dataDictUtil.InitAllObjectTypes dfSession, dfSession.getDocbaseName
dataDictUtil.GetAllObjectTyeps dfSession.getDocbaseName, objectTypes

‘ Iterate the 2-dimensional array “objectTypes”
If VarType(objectTypes) = vbArray + vbVariant Then
      numTypes = UBound(allObjectTypes, 1) - 1

      For i = 0 To numTypes
            typeNameData.InternalName = objectTypes (i, 0)
            typeNameData.DisplayName = objectTypes (i, 1)
      Next i
End If

‘ Initialize formats cache, then retrieve all formats display name.
dataDictUtil.InitAllFormats dfSession, dfSession.getDocbaseName
dataDictUtil.GetAllFormats dfSession.getDocbaseName, formats

‘ Iterate the 2-dimensional-array “formats”
If VarType(formats) = vbArray + vbVariant Then
      numFormats = UBound(formats, 1)

      For i = 0 To numFormats - 1
            formatInfo.InternalName = formats(i, 0)
            formatInfo.DisplayName = formats(i, 1)
      Next i
End if

‘ Clean up.
Set dataDictUtil = nothing
```

Sample Code 2:

```
Dim dataDicUtil as DCUTILMGRLib.DcDataDictionaryUtils
Dim objTypeDisplayName as String
Dim formatDisplayName as String
Dim attributeDisplayName as String
Dim canFullTextIndex as boolean

‘ First, create Data Dictionary Utility Manager.
Set dataDictUtil = new DCUTILMGRLib.DcDataDictionaryUtils

‘ Get the display name for dm_document from a docbase specified in
dfSession.
objTypeDisplayName = utilMgr.GetObjectTypeDisplayName(dfSession,
“dm_document”)

‘ Get the display name for msw8 from a docbase specified in dfSession.
formatDisplayName = dataDictUtil.GetFormatDisplayName(dfSession, “msw8”)
```

```
‘ Get the display name for object_name attribute of dm_document object in
a docbase specified in dfSession.
attributeDisplayName = dataDictUtil.GetAttributeDisplayName(dfSession,
“dm_document”, “object_name”)

‘ Find out if a certain format can be full-text indexed.
CanFullTextIndes = dataDictutil.GetFormatCanIndex (dfSession, "msw8")

‘ Clean up.
Set dataDictUtil = Nothing
```

Appendix C: Object Bag Create/Refill code

Listing #1: Object Create code excerpt (variable declarations omitted):

```
        // This is the code that the user executes when creating an object.
        //
        // Note: In this example bot hthe template area and the object bag
        // are in a user's cabinet (e.g., /user1/obcr/template). In
        // practice it would actually be some application global
        // system area accessible by all users.
        //
        count = 0;
         DfQuery q0 = new DfQuery();
         q0.setDQL("select r_object_id,object_name,i_vstamp "
           + "from dm_document where folder('" + template_folder + "') "
           + "and a_is_hidden = false order by object_name,r_object_id" );
          c0 = q0.execute( session, q0.READ_QUERY );
         while ( c0.next() )
         {
             template_id = c0.getId( "r_object_id" );
             objName = c0.getString( "object_name" );
             temp_ivstamp = c0.getInt( "i_vstamp" );
             if ( objName.equals("template1.doc") ) {
                 System.out.println( "Found a template called " + objName );
                 break;
             }

         count++;
         }
         c0.close();

         // select the lifecycle that you want this file to follow
         q0.setDQL("select r_object_id,object_name,i_vstamp from dm_policy " );
         c0 = q0.execute( session, q0.READ_QUERY );
         while ( c0.next() )
         {
             lfObjId = c0.getId( "r_object_id" );
             lfobjName = c0.getString( "object_name" );
             lf_ivstamp = c0.getInt( "i_vstamp" );

             if ( lfobjName.equals("Test Lifecycle") ) {
                 System.out.println( "Found a lifecycle called " + lfobjName );
                 break;
             }

         }
         c0.close();

         // check the template cache for any objects that exist that are already
         // like this one.
         q0.setDQL("select r_object_id,object_name,a_content_type,r_modify_date, "
                 + "r_content_size,r_object_type,r_lock_owner,"
                 + "r_lock_owner,r_link_cnt,isreplica "
                 + "from dm_document where folder('" + template_cache + "') "
                 + "and a_is_hidden = false and "
                 + "object_name = '" + lfObjId + "_" +  template_id + "_" +
temp_ivstamp + "'");

         c0 = q0.execute( session, q0.READ_QUERY );
         objId = null;
         while ( c0.next() )
         {
             objId = c0.getId( "r_object_id" );
             objName = c0.getString( "object_name" );
             lockOwner= c0.getString( "r_lock_owner" );

             if ( lockOwner.equals("") ) {
                 // lets attempt to lock this object
```

```
                my_doc = (IDfSysObject) session.getObject( objId );
                System.out.println("found an cached object called " + objName);

                try {
                    my_doc.checkout();
                    System.out.println("was able to lock an item in the cache");
                    found_object_in_cache = true;
                    break;
                }
                catch (DfException e)
                {
                    // file might have just been checked out
                    // try the next file
                    System.out.println("file already be locked, try another");
                }
                objId = null;

            } else {
                System.out.println("right name, but locked by " + lockOwner );
            }
            if ( found_object_in_cache ) break;
        }
        c0.close();
        // if none exist then attach to a lifecycle, saveasnew, change name and
        // return to user
        if ( found_object_in_cache == false ) {
            my_doc = (IDfSysObject) session.getObject( template_id );
            my_doc.unlink("/" + user + "/obcr/Template");
        } else {
            my_doc = (IDfSysObject) session.getObject( objId );
            my_doc.unlink("/" + user + "/obcr/Template_Cache");
        }


        my_doc.link("/" + user + "/obcr");
        my_doc.setObjectName("my_doc");
        if ( found_object_in_cache == false ) {
            System.out.println("cache miss");
            objId = my_doc.saveAsNew(false);
            my_doc = (IDfSysObject) session.getObject( objId );
            my_doc.attachPolicy(lfObjId,"InProgress", "");


        } else {
            // if some exist then just change the name, save the object and
finish
            System.out.println("cache hit");
            my_doc.save();
        }
```

Listing #2: Object refill code (variable declarations omitted):

```
//
// This is the code that a job executes to fill the object bag.
//
// create a list of the object id's in the template cache
// put each id into a hash table for later lookup.
//
DfQuery q0 = new DfQuery();
q0.setDQL("select r_object_id,object_name,i_vstamp "
        + "from dm_document where folder('" + template_folder + "') "
        + "and a_is_hidden = false order by object_name,r_object_id" );
 c0 = q0.execute( session, q0.READ_QUERY );
while ( c0.next() )
{
    template_id = c0.getString( "r_object_id" );
    objName = c0.getString( "object_name" );
    temp_ivstamp = c0.getString( "i_vstamp" );
    Object [] o = new Object[3];
    o[0] = new String( template_id );
    o[1] = new String( objName );
    o[2] = new String( temp_ivstamp );
    template_hash.put( template_id, o );



}
c0.close();

//
// create hash table of possible life cycles
// select the lifecycle that you want this file to follow
//
q0.setDQL("select r_object_id,object_name from dm_policy " );
c0 = q0.execute( session, q0.READ_QUERY );
while ( c0.next() )
{
    lfObjId = c0.getId( "r_object_id" );
    lfobjName = c0.getString( "object_name" );

    Object [] o = new Object[3];
    o[0] =  lfObjId ;
    o[1] = new String( lfobjName );

    lf_hash.put( lfObjId.getId() , o );


}
c0.close();

//
// look through the cache and delete any object that no longer has
// a corresponding lifecycle or template
//

q0 = new DfQuery();
q0.setDQL("select r_object_id,object_name,r_lock_owner "
        + "from dm_document where folder('" + template_cache + "') "
        + "and a_is_hidden = false order by object_name,r_object_id" );
 c0 = q0.execute( session, q0.READ_QUERY );
while ( c0.next() )
{
    object_stale = false;
    objId = c0.getId( "r_object_id" );
    objName = c0.getString( "object_name" );
    r_lock_owner = c0.getString( "r_lock_owner" );
    if ( ! r_lock_owner.equals("") ) {

        System.out.println( "Template object  " + objName +
```

```
                " is locked by " + r_lock_owner);
            continue;

        } else {

            //
            // pull out the life cycle object id, template object id, and
            // ivstamp for template object.
            //

            if (objName.length() < 30) {
                System.out.println("name in cache too small (" +
                            objName.length() + ")");
                return ;
            }
            lf_id_part = objName.substring(0,16);
            template_id_part = objName.substring(17,33);
            template_ivstamp_part = objName.substring(34);

            System.out.println("parts " + template_id_part + "  "
                                        + template_ivstamp_part + " "
                                        + lf_id_part  );

             // see if the object has a corresponding template and
             // valid life cycle we check to see if the template and
             // documents exist and if they
             // have the proper i_vstamp. Destroy if they don't.

            Object [] o = (Object[]) template_hash.get( template_id_part );

            Object [] o1 = (Object[]) lf_hash.get( lf_id_part );

            if ( o != null && o1 != null ) {

                // we found it. Now check the ivstamp part.

                String vs = (String) o[2];
                if (vs.equals(template_ivstamp_part)) {
                    // object ok
                    System.out.println("object " + objName + " checks out");
                } else {
                    object_stale = true;
                }
             } else {
                    object_stale = true;
             }


            if ( object_stale ) {
                // destroy it
                System.out.println("object " +objName +
                                   " is stale, destroy it");
                IDfPersistentObject p = session.getObject(objId);
                p.destroy();
            }
          }



    }
    c0.close();

    // iterate through the life cycle and template hash tables to
    // make sure we have sufficient objects for each template / life cycle
    // combination

    Enumeration template_e = template_hash.elements();
    Enumeration lf_e = lf_hash.elements();

    while (lf_e.hasMoreElements() ) {
        Object o1[] = (Object []) lf_e.nextElement();
```

```
        IDfId lfid = (IDfId ) o1[0];

        while (template_e.hasMoreElements() ) {
            Object[] o = (Object[]) template_e.nextElement();
            String fn = lfid + "_" + o[0] + "_" + o[2];

            // run a quick query to see how many of this type of object
            // should be created.
            q0.setDQL("select count(*) as cnt from dm_document "
            + "where folder('" + template_cache + "') "
            + "and a_is_hidden = false and "
            + "object_name = '" + fn + "'");
            c0 = q0.execute( session, q0.READ_QUERY );
            while ( c0.next() )
            {
                cnt = c0.getInt( "cnt" );
            }
            c0.close();



            // now create that many..
            todo = limit - cnt;
            System.out.println("creating " + todo +
                               "objects with name " +fn );

            for (i=0; i<todo; i++) {
                // do the create and attach logic
                my_doc = (IDfSysObject)
                       session.getObject( new DfId((String) o[0]) );
                my_doc.revert();
                my_doc.setObjectName(fn);
                my_doc.unlink("/" + user + "/obcr/Template");
                my_doc.link("/" + user + "/obcr/Template_Cache");
                objId = my_doc.saveAsNew(false);
                my_doc = (IDfSysObject) session.getObject( objId );
                my_doc.attachPolicy(lfid,"InProgress", "");
                System.out.println("created " + fn + " #" + i);

            }
        }
    }
```

Appendix D: Listings for Server Methods in App Server

Listing #1: Our Samplemethod.java

```
import java.io.*;
import com.documentum.fc.client.*;
import com.documentum.fc.common.*;


public class samplemethod {

    private static final String className = "samplemethod ";

    static String docbase = "";
    static String user = "";
    static String runtime_environment = "";
    static String ticket = "";

    public static void main(String[] args) {

        remoteMain( args );
    }

    public static int remoteMain(String[] args) {
        long start = System.currentTimeMillis();
        long ends;
        System.out.println("Argument List");
        for (int i=0; i < args.length; i++) {
            System.out.println("Arg" + i + " : " + args[i]);
            if ( args[i].equalsIgnoreCase("-docbase_name") )
                docbase = args[i+1];
            if ( args[i].equalsIgnoreCase("-user") )
                user = args[i+1];
            if ( args[i].equalsIgnoreCase("-ticket") )
                ticket = args[i+1];
            if ( args[i].equalsIgnoreCase("-running_where") )
                runtime_environment = args[i+1];
        }


            IDfSession session = null;
            try {
                // connect to the docbase
                IDfClient client = DfClient.getLocalClient();
                DfLoginInfo loginInfo = new DfLoginInfo();
                loginInfo.setUser(user);
                loginInfo.setPassword(ticket);
                session = client.newSession(docbase, loginInfo);

                System.out.println("hello world");
            }
            catch (DfException e) {
                System.out.println("Method : " + className + " DFC error occured:
" + e.getMessage());
                if (runtime_environment.equals("app_server")){
                    return -1;
                }else{
                    System.exit(-1);
                }
            }
            finally {
                // Disconnect
                if (session != null) {
                    try {
                        session.disconnect();
                        ends = System.currentTimeMillis();
                        System.out.println("total time = " +
                                                            (ends - start) );
```

```
                        if (runtime_environment.equals("app_server")){
                            return 1;
                        }else{
                            System.out.println("return code = 1");
                            System.exit(1);
                        }
                    }
                    catch (DfException e) {
                        System.out.println(e.getMessage());
                    }
                }
            }
        return 1;
    }

}
```

Listing #2: Httpclient.java

```
/*
**  (c) Copyright Documentum, Inc. 2001
**  All Rights reserved.
*/

//package com.documentum.unsupported;

import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLEncoder;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
import java.io.PrintStream;


public class HttpClient {

    private static final String KEY_URL = "url";

    private final static String SERVLET_NAME = "remoter/remoter";
    private final static String SERVLET_DEFAULT_IP_ADDRESS =
"http://localhost:8080";
    final static String SERVLET_DEFAULT_URL = SERVLET_DEFAULT_IP_ADDRESS + "/" +
SERVLET_NAME;

    private final static String HTML_START_PARAGRAPH = "<p>";
    private final static String HTML_END_PARAGRAPH = "</p>";

    public final static int RETURN_FATAL = -1;
    public final static int RETURN_HTTP_INTERACTION_FAILED = -2;


    private PrintStream debugStream = null;

    public HttpClient() {
        debugStream = null;
    }

    public HttpClient(PrintStream debugStream) {
        this.debugStream = debugStream;
    }

    public int remoteMain(String[] args) {
        try {
            return remoteMain(args, SERVLET_DEFAULT_URL);
        }
        catch (MalformedURLException x) {
            debug(x.toString());
            return RETURN_FATAL;
        }
```

```
    }

    public int remoteMain(String[] args, String host, int port) {
        StringBuffer url = new StringBuffer(128);
        url.append("http://");
        url.append(host);
        url.append(':');
        url.append(port);
        url.append('/');
        url.append(SERVLET_NAME);
        try {
            return remoteMain(args, url.toString());
        }
        catch (MalformedURLException x) {
            debug(x.toString());
            return RETURN_FATAL;
        }
    }

    public int remoteMain(String[] args, String servletURL) throws
MalformedURLException {
        return remoteMain(servletURL + encodeQueryString(args));
    }

    private int remoteMain(String encodedUrl) throws MalformedURLException {
        int status = RETURN_FATAL;

        URL url = new URL(encodedUrl);
        debug("URL=" + encodedUrl);

        try {
            HttpURLConnection connection = createConnection(url);

            // Issue request & check response code.
            long start = System.currentTimeMillis();
            int httpStatus = connection.getResponseCode();
            debug("HTTP status=" + httpStatus + ", round trip time = " +
(System.currentTimeMillis() - start) + " millis");
            switch (httpStatus) {
            case HttpURLConnection.HTTP_OK:
                status = getResultCodeFromResponse(connection);
                break;

            default:
                status = RETURN_HTTP_INTERACTION_FAILED;
                break;
            }
        }
        catch (IOException x) {
            debug(x.toString());
            status = RETURN_FATAL;
        }

        return status;
    }

    private HttpURLConnection createConnection(URL url) throws IOException {
        HttpURLConnection connection = (HttpURLConnection) url.openConnection();
        connection.setUseCaches(false);
        connection.setRequestMethod("GET");
        return connection;
    }

    private int getResultCodeFromResponse(HttpURLConnection connection) {
        try {
            int status = RETURN_FATAL;
            BufferedReader reader = new BufferedReader(new
InputStreamReader(connection.getInputStream()));

            // We wait for up to a second, in 10 millisecond granularity, for the
input stream
```

```
            // to become available. It is most unlikely to take anywhere near
that long, but
            // we know for sure that there is sometimes a small delay.
            for (int i = 0; i < 100  &&  !reader.ready(); ++i) {
                try {
                    Thread.sleep(10);
                }
                catch (InterruptedException x) {
                    break;
                }
            }

            if (reader.ready()) {
                String payload = reader.readLine();
                if (payload != null) {
                    debug(payload);

                    // Extract return code from remote operation (the text of the
first HTML paragraph).
                    int sp1 = payload.indexOf(HTML_START_PARAGRAPH);
                    int ep1 = payload.indexOf(HTML_END_PARAGRAPH);
                    if (sp1 == -1  ||  ep1 == -1) {
                        debug("Couldn't find return code paragraph in response");
                        return RETURN_FATAL;
                    }
                    String stat = payload.substring(sp1 +
HTML_START_PARAGRAPH.length(), ep1);
                    status = Integer.parseInt(stat);
                }
            }
            return status;
        }
        catch (IOException x) {
            return RETURN_FATAL;
        }
    }

    private String encodeQueryString(String[] args) {
        // Construct query string.
        StringBuffer query = new StringBuffer();
        query.append('?');
        for (int i = 0; i < args.length; ++i) {
            if ((i > 0)  &&  ((i % 2) == 0))
                query.append('&');

            if ((i % 2) == 0) {
                query.append(URLEncoder.encode(args[i].substring(1)));
            }
            else {
                query.append('=');
                query.append(URLEncoder.encode(args[i]));
            }
        }

        return query.toString();
    }

    private void debug(String s) {
        if (debugStream != null) {
            debugStream.println(s);
        }
    }


    public static void main(String[] args) {
        String url = SERVLET_DEFAULT_URL;
        if (System.getProperty(KEY_URL) != null) {
            url = System.getProperty(KEY_URL);
        }

        HttpClient client = new HttpClient();
        int status = RETURN_FATAL;
```

```
        try {
            status = client.remoteMain(args, url);
        }
        catch (MalformedURLException x) {
            System.out.println("Servlet URL " + url + " is malformed");
            System.exit(RETURN_FATAL);
        }
        switch (status) {
        case RETURN_HTTP_INTERACTION_FAILED:
            System.out.println("Failed to establish a successful HTTP interaction
with the remote app server");
            System.exit(RETURN_FATAL);

        default:
            System.out.println("Publish returned status = " + status);
            break;
        }
        System.exit(status);
    }
}
```

Listing #3: RemoteMainServlet.java

```
/*
**  (c) Copyright Documentum, Inc. 2001
**  All Rights reserved.
*/

//package com.documentum.unsupported;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.SingleThreadModel;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpUtils;

import java.util.Enumeration;
import java.util.Hashtable;
import java.util.StringTokenizer;

import java.lang.reflect.Method;
import java.lang.reflect.InvocationTargetException;



/**
 * This is a servlet which invokes the main method of an configured class.
 * It generates a response which discloses the return value,
 * duration in milliseconds, and any exception message. The arguments to the
 * command line app are encoded as a URL query string in a GET or as an XML
payload in a POST.
 */
public class RemoteMainServlet extends HttpServlet
{
    private static final String INIT_PARAM_CLASS = "class";
    private static final String INIT_PARAM_DEBUG = "debug";
    private boolean debugging = false;
    private Class remoteMainClass = null;

    private void debug(String s) {
        if (debugging) {
            log(s);
        }
```

```
    }

    /*
       * This is a default implementation for HTML response generation.
       * This implementation is
       * tightly coupled with the HttpClient implementation,
       * and should not be changed without
       * a review of the change's effect on HttpClient.
     */
    private void doResponse(HttpServletResponse resp, int result, long duration,
String xMessage) throws ServletException {
        if (result != 0) {
            debug("doResponse: result=" + result);
        }
        try {
            resp.setContentType("text/html");

            StringBuffer sb = new StringBuffer();

            sb.append("<html><body>");

            sb.append("<h1>Method Return Value</h1>");
            sb.append("<p>" + result + "</p>");

            if (xMessage != null) {
                sb.append("<h1>Exception Message</h1>");
                sb.append("<p>" + xMessage + "</p>");
            }

            sb.append("<h1>Time Taken (milliseconds)</h1>");
            sb.append("<p>" + duration + "</p>");

            sb.append("</body></html>");

            PrintWriter pw = resp.getWriter();
            pw.print(new String(sb));
            pw.close();

            resp.setContentLength(sb.length());
        } catch (IOException x) {
            log("doResponse: " + x.getMessage());
        }
    }

    /*
     * Produces a response which can utilize the return value,
     * duration in milliseconds, and any
     * exception message.
     */
    private void doRequest(HttpServletResponse resp, String[] argv) throws
ServletException {
        String xMessage = null;

        // Now invoke the method, passing args from the query string.
        long start = System.currentTimeMillis();
        Integer result = new Integer(-1);
        try {
            Method main = remoteMainClass.getMethod("remoteMain", new Class[] {
String[].class });
            result = (Integer) main.invoke(null, new Object[] { argv });
        } catch (IllegalAccessException x) {
            log("doRequest: IllegalAccessException: " + x.getMessage());
            xMessage = x.getMessage();
        } catch (IllegalArgumentException x) {
            log("doRequest: IllegalArgumentException: " + x.getMessage());
            xMessage = x.getMessage();
        } catch (InvocationTargetException x) {
            log("doRequest: InvocationTargetException: " + x.getMessage());
            xMessage = x.getMessage();
        } catch (Throwable t) {
            t.printStackTrace();
        }
```

```
        long duration = System.currentTimeMillis() - start;

        // Now return a response.
        doResponse(resp, result.intValue(), duration, xMessage);
    }

    public void init() throws ServletException {
        String debugFlag = getInitParameter(INIT_PARAM_DEBUG);
        if (debugFlag != null) {
            if (debugFlag.equalsIgnoreCase("true")) {
                debugging = true;
            }
        }

        String remoteMainClassName = getInitParameter(INIT_PARAM_CLASS);
        if (remoteMainClassName == null) {
            remoteMainClassName = "samplemethod";
        }
        if (remoteMainClassName != null) {
            try {
                remoteMainClass = Class.forName(remoteMainClassName);
            }
            catch (ClassNotFoundException x) {
                log("Error loading dynamic remote main class: " +
                           x.getMessage());
                throw new ServletException("Error loading dynamic remote main
class", x);
            }
            catch (ClassCastException x) {
                log("Error loading dynamic remote main class: " +
                                       x.getMessage());
                throw new ServletException("Error loading dynamic remote main
class", x);
            }
        }
        else {

            log("No dynamic remote main class specified in init-params");
            throw new ServletException("No dynamic remote main class specified in
init-params");
        }
    }

    public void destroy() {
    }

    /**
       Upon receipt of a GET request, the servlet parses the
       query string and translates the
       value assertions into getopt style argument specifications with
       in an argv-style String
       array.
     */
    public void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {

        // Construct argument array from XML request document.
        String[] argv = null;

        if (req.getQueryString() != null) {
            try {
                // Parse query string.
                Hashtable params =
                           HttpUtils.parseQueryString(req.getQueryString());

                // Allocate appropriately sized argument array.
                argv = new String[params.size() * 2];

                // For each query string argument, construct a
                // Unix getopt argument pair.
                int i = 0;
                Enumeration keys = params.keys();
```

```
                while (keys.hasMoreElements()) {
                    String key = (String) keys.nextElement();
                    Object value = params.get(key);
                    String valueString = null;
                    if (value instanceof String) {
                        valueString = (String) value;
                    } else if (value instanceof String[]) {
                        // For multiply asserted arguments, assume
                        //  the last one is the correct one.
                        String[] valueStringArray = (String[]) value;
                        valueString = valueStringArray[valueStringArray.length -
1];
                    }
                    argv[i++] = "-" + key;
                    argv[i++] = valueString;
                }

            } catch (IllegalArgumentException x) {
                /* no query string */
            }
        }

        // Now do the request.
        doRequest(resp, argv);
    }
}
```

Listing #4: Web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">

<web-app>
    <servlet>
        <servlet-name>
            remoter
        </servlet-name>
        <servlet-class>
            RemoteMainServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>


    </servlet>
    <servlet-mapping>
        <servlet-name>
            remoter
        </servlet-name>
        <url-pattern>
            /remoter/*
        </url-pattern>
    </servlet-mapping>
</web-app>
```