

# Tuning EMC Documentum Webtop: Advanced Search FTDQL Behavior for DFC 5.3 SP2 and SP3

*Applied Technology*

---

## **Abstract**

This white paper describes how to tune FTDQL behavior of the Advanced Search component in EMC<sup>®</sup> Documentum<sup>®</sup> Webtop. By using `dfc-dqlhints.xml` one could disable the FTDQL behavior requested by Advanced Search to address performance issues.

March 2007

---

---

Copyright © 2007 EMC Corporation. All rights reserved.

EMC believes the information in this publication is accurate as of its publication date. The information is subject to change without notice.

THE INFORMATION IN THIS PUBLICATION IS PROVIDED “AS IS.” EMC CORPORATION MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WITH RESPECT TO THE INFORMATION IN THIS PUBLICATION, AND SPECIFICALLY DISCLAIMS IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Use, copying, and distribution of any EMC software described in this publication requires an applicable software license.

For the most up-to-date listing of EMC product names, see EMC Corporation Trademarks on EMC.com

All other trademarks used herein are the property of their respective owners.

Part Number H2664

---

## Table of Contents

<b>Executive summary</b> .....	<b>4</b>
<b>Introduction</b> .....	<b>4</b>
Audience .....	5
Terminology .....	5
<b>FTDQL background</b> .....	<b>5</b>
<b>dfc-dqlhints.xml background</b> .....	<b>10</b>
Case No. 1: Always turning off FTDQL in Advanced Search .....	11
Case No. 2: Turning off FTDQL for specific WHERE clause operators or attributes .....	13
Case No. 3: Turning off FTDQL for specific types in the FROM clause .....	14
Case No. 4: Manipulating FTDQL hint while issuing other hints .....	15
Case No. 5: Turning off FTDQL when a date range is used .....	17
Case No. 6: Turning off FTDQL in case of FOLDER(DESCEND).....	17
Case No. 7: Turning off FTDQL for a specific repository.....	18
<b>Known limitations of Patch 9950</b> .....	<b>18</b>
<b>Conclusion</b> .....	<b>18</b>
<b>Appendix: DTD definition</b> .....	<b>19</b>

---

## Executive summary

By default in EMC® Documentum® 5.3, all sysobjects have their attributes and content (if present and indexable) full-text indexed. The attribute values are mirrored from the corresponding values in the database, which has several advantages. However, experience with full-text technology has shown poor performance with the full-text engine as compared to the database.

In Content Server the concept of FTDQL processing was to set up to support both relational and full-text search capabilities. These capabilities allow a query to locate objects using both the full-text and relational database. However, with the release of 5.3, `dfc-dqlhints.xml` was introduced to easily add DQL query hints to a query that was constructed by the DFC query builder. The DFC query builder constructs a query defined by the Advanced Search page.

This white paper lists several cases of when and how a `dfc-dqlhints.xml` file is used to disable the FTDQL behavior requested by Advanced Search by default, such as turning off FTDQL in Advanced Search, or turning off FTDQL when using a date range or for a specific repository.

## Introduction

The purpose of this white paper is to describe how to tune FTDQL behavior of the Advanced Search component in Documentum Webtop. In Documentum 5.3, Advanced Search was set up to leverage the Content Server's FTDQL hint for all Advanced Search queries. The FTDQL hint will cause the DQL WHERE clause (and any SEARCH clause of the DQL query) to be issued to the full-text, instead of just serving the SEARCH clause at the full-text and filtering on the rest of the WHERE clause with the relational database in a subsequent join.

In 5.3, by default, all sysobjects have their attributes and content (if present and indexable) full-text indexed. The attribute values are mirrored from the corresponding values in the database. This has several advantages:

- If a combination full-text and attribute query is issued, then this potentially allows the query to be made more selective at the full-text, and hence allow for more efficient total processing of the query.
- If the query is just an attribute-based query, then it can potentially support more efficient case-insensitive searches

However, experience with full-text technology has shown that several use cases perform poorly with the full-text engine as compared to the database. These use cases include:

- ID-based wildcard searches or date range queries that could have many hits to the full-text dictionary and cause much I/O
- Applications that are sensitive to the propagation latency due to the attribute changes into the full-text and/or perhaps need transaction-like semantics (vs. the several minute latency provided by the full-text index)
- Queries with FOLDER(DESCEND) predicates

In other cases, it might be just that the database performs the customer queries efficiently and the full-text index adds no value. Sending all Advanced Search queries to the full-text could unnecessarily burden it. For example, suppose that 10,000 users are issuing point queries (exact matches) on some "customer id," and only 100 users are performing proper full-text searches. Then the 10,000 users will be adding a burden to the full-text index without getting any value for it, and in addition, their large query volume increases the probability that the full-text environment will time out the queries.

Therefore, it may be highly desirable to change the default behavior of the Advanced Search to disable the FTDQL hint in several use cases. This white paper describes how to accomplish this using the `dfc-dqlhints.xml` configuration file that is set up on the same host as the Webtop application server. This file will influence the behavior of the query builder of Advanced Search.

---

The behavior of the system is relative to 5.3 SP2 and SP3 using Patch 9950. The behavior differs slightly in some ways from 5.3 SP1. The behavior of 5.3 SP1 is documented in other documents.

## **Audience**

This white paper is written for administrators who are considering tuning FTDQL behavior of the Advanced Search component in Documentum Webtop to reduce the amount of burden on database performance.

## **Terminology**

**Data join:** In this full-text/RDBMS context, join is the part of the query in which items found in one store are matched up with items found in the other store. For example, if part of the query goes to the full-text first, then the output from that query is a list of object IDs that are joined with the database to locate the rest of the object's attributes.

**FTDQL:** This DQL query processing technique was introduced in 5.3 to service the attribute and any full-text search clauses with the full-text engine rather than produce searches across both data stores. In FTDQL processing although the WHERE clause is being sent to the full-text, all values from the SELECT list are still obtained from the RDBMS once the rows have been determined from the full-text.

**Initial query scan:** Queries, whether they are generated by the full-text index or the relational database, are typically composed of an initial reading of data from disk (a scan) followed by subsequent filtering or joining with other scans. The initial query scan is important because in many cases it determines the amount of I/O that performed.

**Selective scan:** A selective scan is a sequence of reads to some large record store that involves only a few reads (the scan is very selective of the data it reads based on the items it's looking for).

**Selective query:** A selective query is a query that returns very few results out of a possible large number of results. For example, if a query returns one hit out of potentially 1 million records, then the query is selective. Most queries become more selective with the addition of many search items joined by operators like "AND". For example, an Internet search engine query on "rover" causes 91 million hits and a query on "mars" generates 360 million hits, but a query on "mars" and "rover" generates only 13 million hits and a query for "mars FOLLOWED BY rover" (which is done by just quoting the two terms "mars rover") generates only 1 million hits.

## **FTDQL background**

The following section outlines some important concepts of FTDQL processing in Content Server. The Documentum Query Language (DQL) supports both relational and full-text search capabilities. These capabilities allow a query to locate objects using both the full-text and the relational database. Prior to Documentum 5.3 a query that leveraged both repositories would contain a SEARCH clause (which provided search tokens for the full-text) and a WHERE clause (which provided search logic for the relational database). For example, to find the object IDs for all documents that have an object name of 'foo' but have "bar" in their content somewhere, the following query could be issued:

```
SELECT r_object_id FROM dm_document SEARCH DOCUMENT CONTAINS "bar" WHERE object_name = 'foo'
```

When such a query reached the Content Server it would issue the full-text query and obtain a result set that was then joined with the relational database result set for a combined result set. So, if in our above example the output from the full-text located three objects such as the following:

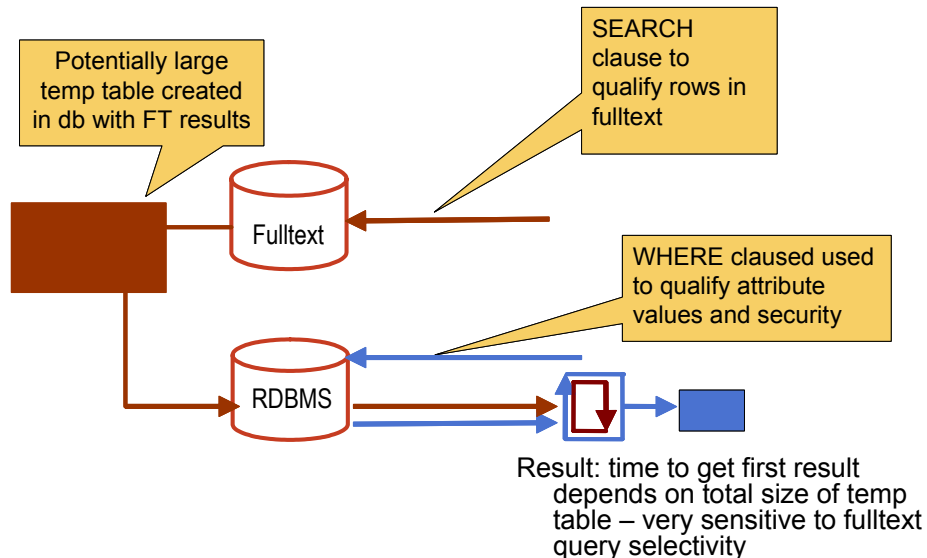
R_object_id	Text in full-text index
0900000180000109	Bar fox five ten 10 ggg
0900000180000110	Milk bar going there
0900000180000111	Bumping basket bar

And the relational database portion of the query located four objects:

R_object_id	Object_name
0900000180000109	Foo
0900000180000111	Foo
0900000180000113	Foo
0900000180000114	Foo

The join processing would create a temp table with the results from the full-text (object IDs) and then execute a relational join with the output from the relational database object\_name search. Then the joined result set will only have objects 0900000180000109 and 0900000180000111. This general technique is illustrated in Figure 1.

Some performance issues could occur if the full-text query was not selective (that is, it returned lots of results) because the resulting temp table created with the full-text results could be very large. In fact, it's quite possible that the final result set was small (only a few rows returned) but in getting to the answer, many rows had to be processed.



**Figure 1. Pre-5.3 full-text/RDBMS query processing**

Prior to 5.3 some of the RDBMS attributes could be full-text indexed and hence, the SEARCH clause could apply to the content of the document and whichever attributes had been full-text indexed. Suppose that “object\_name” was selected to be full-text indexed. Then, in our example above, the following query:

```
SELECT r_object_id FROM dm_document SEARCH DOCUMENT CONTAINS 'foo bar'
```

Would return at least 0900000180000109 and 0900000180000111 in the above example without having the WHERE clause. It would “at least” return those two because it might also return other documents that had “foo” and “bar” in the content or object\_name. For example, the above query could also return a document with the object name of “bar foo”. This is illustrated below.

R_object_id	Text in full-text index	Object name value in full-text index
0900000180000109	Bar fox five ten 10 ggg	foo
0900000180000115	This report is for summary	Bar foo
0900000180000111	Bumping basket bar	Foo

Prior to 5.3 one could modify the SEARCH clause so that the token “foo” was looked for only in the attribute object\_name.

```
SELECT r_object_id FROM dm_document SEARCH TOPIC 'bar <AND> object_name  
<CONTAINS> foo'
```

In 5.3 this technique was extended in the following ways:

- DQL was enhanced to convert WHERE clause syntax into the corresponding full-text query syntax and to issue the corresponding query to the full-text when the query was designated to run as “FTDQL”.
- By default, all sysobjects’ content and attributes were full-text indexed.

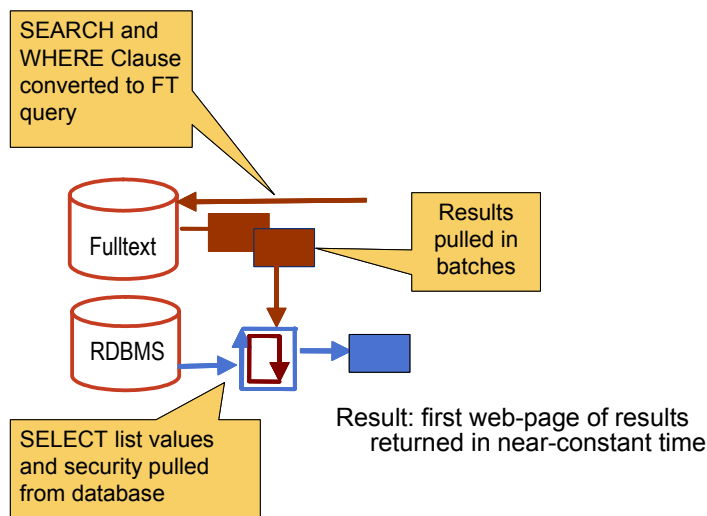
Our example query above can be expressed more naturally as:

```
SELECT r_object_id FROM dm_document SEARCH DOCUMENT CONTAINS "bar"  
WHERE object_name = 'foo'
```

And in this case the object\_name and content query both go to the full-text index (as shown in Figure 2).

Also illustrated in Figure 2 is a change to the query processing of an FTDQL query in that the results are returned from the full-text in batches that are joined (in batches) to the Content Server database queries. This eliminates the creation of the temporary table in the database and allows for some initial results to be returned faster. That is, it is no longer necessary for Content Server to consume the entire full-text result before it determined if any of the items found succeeded in further qualification.

This latter optimization improves response time, especially when the full-text query is not selective and returns many results. Prior to 5.3, an unselective full-text query would cause a large temp table to be built, leading to long response times. In FTDQL processing the temp table is not created. This optimization is possible because the results are returned in relevance order from the full-text index.



**Figure 2. 5.3 FTQL processing**

The FTDQL enhancement is documented in the *Content Server DQL Reference Manual*. It is important to note that this behavior in query processing is enabled in two ways:

- Explicitly by placing an `ENABLE(FTDQL)` at the end of the query
- Implicitly by combining a `SEARCH` and `WHERE` clause in a single query

`ENABLE (FTDQL)` allows for queries without a `SEARCH` clause to be processed by the full-text engine. So, for example the following query:

```
SELECT r_object_id FROM dm_document WHERE object_name = 'foo'
ENABLE(FTDQL)
```

Would be issued to the full-text to locate the objects, while the following:

```
SELECT r_object_id FROM dm_document WHERE object_name = 'foo'
```

Would be issued to the RDBMS to locate the objects.

In both cases, the values of the `SELECT` list are obtained from the relational database.

Note that the FTDQL query shown above could qualify documents whose `object_name` is “Foo” or “FOO” as well as “foo”. That is, if the `WHERE` clause is sent to the full-text engine, then it is doing a case-insensitive search, while the database is doing a case-sensitive search by default. To change the above database search to make it case-insensitive, modify it as follows:

```
SELECT r_object_id FROM dm_document WHERE upper(object_name) =
upper('foo')
```

In addition, the following query has the identical semantics to the FTDQL query without the calls to the `upper` function:

```
SELECT r_object_id FROM dm_document WHERE upper(object_name) =
upper('foo') ENABLE(FTDQL)
```

Webtop Advanced Search will, by default, add the `ENABLE(FTDQL)` hint for search queries it is directed to build. The FTDQL processing is sometimes undesirable at times in a customer environment. These scenarios include the following:

1. When the attribute searches cause many hits to the full-text index dictionary (wildcard ID searches or date range searches)
2. When the calling application depends on transactional update semantics

---

### 3. When too many search terms are added to the full-text query

In the second scenario the application could get unexpected results. That is, the full-text index is updated asynchronously after a transaction for an object is committed to the database. The transaction commit causes an event to be generated that will ultimately cause the full-text index to be updated with the new values. However, there will be a lag of time (at least a minute or two) before the full-text index has any values just committed to the database. FTDQL might need to be disabled to get around this latency issue.

In the first and third scenarios above it is possible that the full-text engine will consume more resources than the database to obtain the results, but worse yet, it is possible that the full-text index could “time out” the query and return an error. If wildcard searches are being performed on attributes that have ID-like values then the full-text query could hit the dictionary significantly more than if the wildcard is for English language words. These extra hits to the dictionary can cause the full-text engine to generate a significant amount of I/O.

In the third scenario it is possible that the query explicitly has more search terms. An example of this that occurs with Advanced Search is when the query builder is instructed to use the FOLDER(DESCEND) predicate in DQL.

So, by default, Webtop Advanced Search will not search in a particular folder but will issue the query repository-wide. This is designated using the Locations field of the Advanced Search form shown in Figure 3.

The screenshot shows the 'Locations' section with two radio buttons: 'dm\_notes' (selected) and 'Current location only: dm\_notes: /Edward Bueche'. An 'Edit' link is to the right. Below is the 'Object Type' dropdown menu set to 'Document (dm\_document)'. The 'Properties' section contains three input fields: 'Name' (dropdown), 'begins with' (dropdown), and 'foo' (text). A 'Remove' link is to the right of the 'foo' field. Below the 'Name' dropdown is a link 'Add another property'.

**Figure 3. Locations field of Advanced Search form**

If Locations is specified, then the FOLDER(DESCEND) predicate will be added to the query, as shown in Figure 4.

The screenshot shows the 'Locations' section with two radio buttons: 'dm\_notes' (unselected) and 'Current location only: dm\_notes: /Engineering' (selected). An 'Edit' link is to the right. Below is the 'Object Type' dropdown menu set to 'Sysobject (dm\_sysobject)'. The 'Properties' section contains three input fields: 'Name' (dropdown), 'begins with' (dropdown), and 'foo' (text). A 'Remove' link is to the right of the 'foo' field. Below the 'Name' dropdown is a link 'Add another property'.

**Figure 4. FOLDER(DESCEND) predicate added**

In FTDQL the FOLDER(DESCEND) predicate will cause the following processing:

1. The folder and all of its children folder IDs will be determined using a SQL query.

- 
2. These IDs will be added as search terms to the full-text query.
  3. The resulting full-text query will be executed.

For example, in our sample repository:

5,544 folders from:

```
SELECT count(*) FROM dm_folder WHERE FOLDER('/Engineering',  
DESCEND)
```

598 folders from:

```
SELECT count(*) FROM dm_folder WHERE  
FOLDER('/Engineering/Enterprise Systems', DESCEND)
```

134 folders from:

```
SELECT count(*) FROM dm_folder WHERE  
FOLDER('/Engineering/Enterprise Systems/Capacity Planning',  
DESCEND)
```

Hence, FOLDER(DESCEND) could significantly increase the number of search terms added to the full-text query (hundreds) and is *not* recommended when it is potentially so large.

## dfc-dqlhints.xml background

A mechanism was introduced in 5.3 to manipulate the construction of DQL queries by the Advanced Search component of Webtop without having to write code. In previous releases it was necessary to extend the search component by adding Java code that edited the finished query prior to it being issued to Content Server. This alternate mechanism in 5.3 was designed to make it easy to add DQL query hints to the query that was constructed by the DFC query builder. The DFC query builder constructs a query defined by the Advanced Search page.

The mechanism to accomplish this is an XML file called `dfc-dqlhints.xml`<sup>1</sup>. The path to the file is defined in the `dfc-fullproperties` file in a manner as follows:

```
dfc.dqlhints.file=c:\documentum\config\dfc-dqlhints.xml
```

The format of this XML file defines a series of “rules,” each of which have “conditions” that might be met, causing some hint to be applied to the query. The dtd of the XML file is self-referencing (meaning that although the code that parses the XML file format references a DTD it is not necessary to have one in the directory with the `dfc-dqlhints.xml` file).

To get Webtop to utilize the hint file properly, do the following:

1. Put the path of the `dfc-dqlhints.xml` file into the `dfcfull.properties` (as shown above).
2. Create the `dfc-dqlhints.xml` file in the position pointed to by the `dfc.dqlhints.file` variable.
3. Apply Patch 9950 for `dfc.jar`.
4. Reboot Webtop.

If the syntax of the file is incorrect then the Advanced Search will issue an error the first time and then ignore the faulty file in subsequent invocations. The examples shown in subsequent sections should be exactly as shown.

---

<sup>1</sup> The DTD that specifies the syntax for this file is provided in the “Appendix: DTD definition” section. This paper describes many, but not all, of the possible ways in which the file can be used.

---

In addition, when turning off FTDQL one must also change the default behavior of the Advanced Search so that it is explicitly case-insensitive. The full-text query by default is “case insensitive” and the Advanced Search takes advantage of this in its formulation of FTDQL queries. However, the database is by default “case sensitive” and must be explicitly made case-insensitive by adding upper() (or lower()) function calls to the WHERE clause attribute values and the search literals. Case-insensitivity can be enabled in Advanced Search. In wdk/config/advsearchex.xml, change the following:

```
<!-- default case sensitive property value for attribute constraints. -  
-->  
    <defaultmatchcase>true</defaultmatchcase>
```

To the following:

```
<!-- default case sensitive property value for attribute constraints.  
-->  
    <defaultmatchcase>false</defaultmatchcase>
```

That is, Advanced Search is implicitly leveraging the behavior that the following:

```
SELECT r_object_id FROM dm_document WHERE object_name = 'foo'  
ENABLE (FTDQL)
```

Has equivalent search semantics to the following:

```
SELECT r_object_id FROM dm_document WHERE upper(object_name) =  
upper('foo') ENABLE (FTDQL)
```

Because the full-text engine is case-insensitive (that is, in the FTDQL case, the upper() (or lower()) calls do nothing). Until <defaultmatchcase> is set to “false” the upper() (or lower()) function calls will not be placed on the query.

Note that case-insensitive queries to the RDBMS are harder to tune than case-sensitive ones. A case-insensitive RDBMS search will have to apply the upper() or lower() functions to the attribute value and search string in the WHERE clause. By default, this will disable the use of any RDBMS index on the attribute (column) causing a full table scan of the database table, making the query run slower. Hence, when using Oracle we recommend building a functional index on the desired column to ensure that case-insensitive RDBMS queries run efficiently. A similar, but different mechanism can achieve good performance on DB2 as well (generated column support).

In addition, once FTDQL is disabled there are several other behavior changes that might be noticed:

- Full-text results will be populated in a temp table to be joined with the RDBMS as was done in versions earlier than 5.3. Hence, the response time of NOFTDQL queries will be sensitive to non-selective search terms (the less selective, the longer the response).
- FTDQL queries return the results in relevance order. Non-FTDQL queries that involve full-text have to enable this order by putting SCORE on the SELECT list and ordering the results by SCORE.

To test whether the file is working properly one should turn on the dmcl trace of Advanced Search and validate that the query constructed by the Advanced Search is as desired.

The next sections outline several use cases in which the dfc-dqlhints.xml file is employed to disable the FTDQL behavior requested by Advanced Search by default.

## ***Case No. 1: Always turning off FTDQL in Advanced Search***

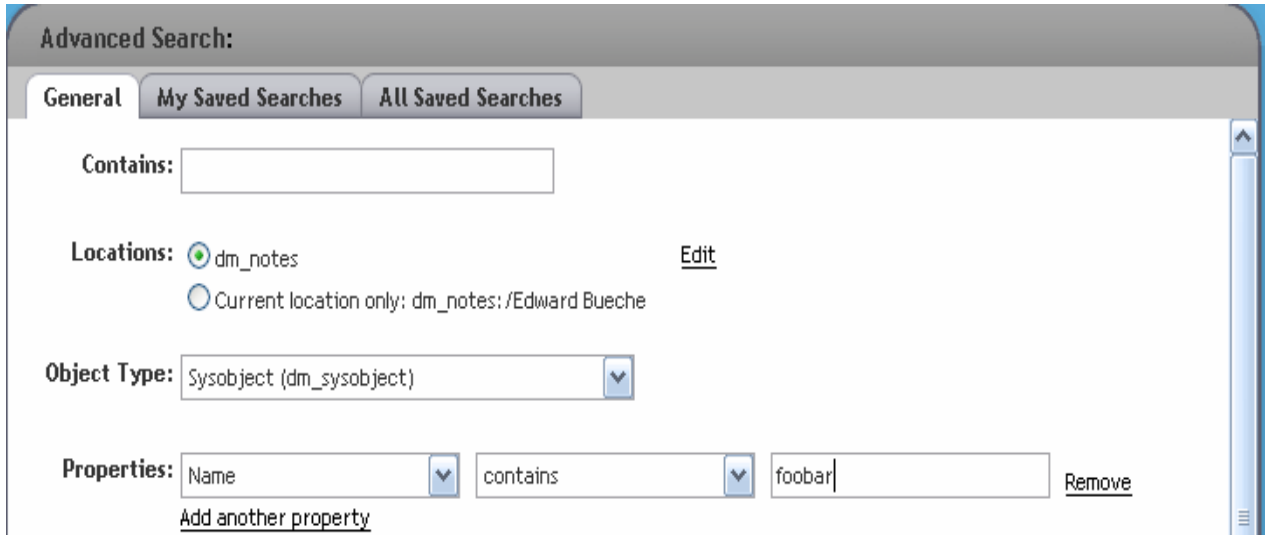
If the desired behavior of the customer is to turn off the FTDQL hint for all Advanced Search queries, then the dfc-dqlhints.xml file should have the following format:

---

Tuning EMC Documentum Webtop: Advanced Search FTDQL Behavior for  
DFC 5.3 SP2 and SP3  
Applied Technology

```
<?xml version="1.0"?>
  <!DOCTYPE RuleSet PUBLIC "dfc-dqlhints.dtd" "dfc-dqlhints.dtd">
  <RuleSet>
    <Rule>
      <DisableFTDQL/>
    </Rule>
  </RuleSet>
```

When the file has been set up in this fashion then a query issued by Advanced Search will not have the FTDQL hint added. For example, suppose a search is done for documents with an object\_name that contains “foobar” in the name, as shown in Figure 5:



**Figure 5. Search with “foobar” in the name**

Then without the dfc-dqlhints.xml defined the query issued would have looked as follows:

```
SELECT r_object_id, object_name,r_object_type,r_lock_owner,owner_name,
r_link_cnt,r_is_virtual_doc,
r_content_size,a_content_type,i_is_reference,
r_assembled_from_id,r_has_frzn_assembly,a_compound_architecture,
i_is_replica,r_policy_id,r_modify_date FROM dm_sysobject WHERE
(object_name LIKE '%foobar%' ESCAPE '\') AND (a_is_hidden = FALSE)
ENABLE (FTDQL)
```

Once the dfc-dqlhints.xml file has been put in place then the query would look as follows:

```
SELECT r_object_id, object_name,r_object_type,r_lock_owner,owner_name,
r_link_cnt,r_is_virtual_doc,
r_content_size,a_content_type,i_is_reference,
r_assembled_from_id,r_has_frzn_assembly,a_compound_architecture,
i_is_replica,r_policy_id,r_modify_date FROM dm_sysobject WHERE
(object_name LIKE '%foobar%' ESCAPE '\') AND (a_is_hidden = FALSE)
ENABLE (NOFTDQL) 2
```

<sup>2</sup> Actually, if Webtop is made case-insensitive for the RDBMS then the WHERE clause would have the string “foobar” and object\_name converted to upper- (or lower-)case in the WHERE clause itself.

---

Notice that at the end of the query ENABLE(FTDQL) has been changed to ENABLE(NOFTDQL).

Now, the “Contains” part of Advanced Search is used when a full-text query is desired against the content. It causes a SEARCH clause to be added to the query. Suppose, for example, that a content full-text query was desired as well as an attribute query.

The screenshot shows the 'Advanced Search' interface with three tabs: 'General', 'My Saved Searches', and 'All Saved Searches'. The 'General' tab is active. The 'Contains' field is set to 'global warming'. Under 'Locations', 'dm\_notes' is selected with a radio button, and there is an 'Edit' link. Below it, 'Current location only: dm\_notes:/Edward Bueche' is shown. The 'Object Type' dropdown is set to 'Sysobject (dm\_sysobject)'. The 'Properties' section shows a dropdown set to 'Name', followed by a dropdown set to 'contains', and an empty text field with a 'Remove' link. Below the text field is the text 'Add another property'.

**Figure 6. Advanced Search: Search for global warming documents**

Then Advanced Search with our dfc-dqlhints.xml file will issue the following query:

```
SELECT r_object_id,object_name,r_object_type,r_lock_owner,
owner_name,r_link_cnt,r_is_virtual_doc,
r_content_size,a_content_type,i_is_reference,
r_assembled_from_id,r_has_frzn_assembly,
a_compound_architecture,i_is_replica,r_policy_id,score,r_modify_date
FROM dm_sysobject SEARCH DOCUMENT CONTAINS 'global warming' WHERE
(object_name LIKE '%foobar%' ESCAPE '\') AND (a_is_hidden = FALSE)
ENABLE (NOFTDQL)
```

Without this explicit ENABLE(NOFTDQL) the query would implicitly run as FTDQL even if the ENABLE(FTDQL) was not present (because the query contains a “SEARCH DOCUMENT CONTAINS” clause). As mentioned earlier, using this NOFTDQL hint will cause the SEARCH part of the query to go to the full-text index, but the WHERE part of the query will be serviced by the RDBMS (the end result to come from the join between the two, as shown in Figure 1). Therefore, a temp table will be populated with the full-text result. The downside of such an approach is that if the full-text query is unselective then the temp table will be quite large, impacting response time in a negative fashion. In this case, “global warming” could have millions of hits in the full-text, causing a temp table that is of the same size to be created, and thus lengthening the response time significantly.

## **Case No. 2: Turning off FTDQL for specific WHERE clause operators or attributes**

The next use case is one in which it is highly desirable to turn off the FTDQL hint when certain operators are used in the WHERE clause. For example, suppose that it is known that object\_names are generated automatically (hence, have highly artificial names), but the values of the Subject attribute are natural language values. Hence, an Advanced Search of:

---

Object Type: Sysobject (dm\_sysobject) ▼

Properties: Name ▼ begins with ▼ foo | [Remove](#)  
[Add another property](#)

**Figure 7. Advanced Search of foo**

Which generates a query with object\_name LIKE 'foo%' might cause the full-text to find thousands of unique dictionary hits while a similar search against the Subject attribute might only generate one hit to the dictionary:

Object Type: Sysobject (dm\_sysobject) ▼

Properties: Subject ▼ begins with ▼ help | [Remove](#)  
[Add another property](#)

**Figure 8. Advanced Search of Subject “help”**

In addition, it might be that all LIKE queries are considered prone to timing out in the full-text due to its wildcard handling. In this case there is a desire to turn off FTDQL if the Advanced Search has operators such as BEGINS WITH, ENDS WITH, and CONTAINS, but to leave it in place for operators like =.

To turn off FTDQL for a particular attribute that is part of the WHERE clause with a wildcard operator, craft the dfc-dqlhints.xml file to look as:

```
<?xml version="1.0"?>
  <!DOCTYPE RuleSet PUBLIC "dfc-dqlhints.dtd" "dfc-dqlhints.dtd">
  <RuleSet>
    <Rule>
      <Condition>
        <Where condition="any">
          <Attribute operator="like">object_name</Attribute>
        </Where>
      </Condition>
      <DisableFTDQL/>
    </Rule>
  </RuleSet>
```

If multiple attributes need this treatment, just add additional <Attribute> lines with the other Attributes identified.

---

Note: You cannot turn off wildcard operators for all attributes in a single line at this time.

---

### **Case No. 3: Turning off FTDQL for specific types in the FROM clause**

It is also possible that the user environment knows that any query to a specific object type can bypass FTDQL and use the database for its attribute values. So, for example, if there was a subtype in the repository called km\_message and it was a subtype of dm\_document and the desired behavior of Advanced

---

Search was that any query to km\_message should not use FTDQL, then the dfc-dqlhints.xml file could have the following format:

```
<?xml version="1.0"?>
  <!DOCTYPE RuleSet PUBLIC "dfc-dqlhints.dtd" "dfc-dqlhints.dtd">
  <RuleSet>
    <Rule>
      <Condition>
        <From condition="any">
          <Type>km_message</Type>
        </From>
      </Condition>
      <DisableFTDQL/>
    </Rule>
  </RuleSet>
```

So, searching for a km\_document as:

Object Type:

Properties:    [Remove](#)

[Add another property](#)

**Figure 9. Searching for km\_document**

Will cause the FTDQL to be omitted, while searching for a normal dm\_document would not:

Object Type:

Properties:    [Remove](#)

[Add another property](#)

**Figure 10. Searching for dm\_document**

### ***Case No. 4: Manipulating FTDQL hint while issuing other hints***

The dfc-dqlhints.xml file also allows for additional hints to be applied to the query. Some of these might be extremely important for other reasons. For example, suppose that an application needed to change the underlying access method for SQL Server from keyset cursors to default result set for a particular type of query. In addition, the user wants to turn off the FTDQL. In this case, the user would need to add the NOFTDQL hint to the SQL\_DEF\_RESULT\_SET hint as shown below.

---

```

<?xml version="1.0"?>
  <!DOCTYPE RuleSet PUBLIC "dfc-dqlhints.dtd" "dfc-dqlhints.dtd">
  <RuleSet>
    <Rule>
      <Condition>
        <Where condition="any">
          <Attribute operator="like">subject</Attribute>
          <Attribute operator="like">object_name</Attribute>
        </Where>
      </Condition>
      <DQLHint> ENABLE(SQL_DEF_RESULT_SET 100, NOFTDQL) </DQLHint>
      <DisableFTDQL/>
    </Rule>
  </RuleSet>

```

This example also shows how the hint is applied only when LIKE is used on the subject or the object\_name. An important point is that when the “Conditions” are met then the DQLHINT is applied to the query (and overrides the DisableFTDQL clause).

Another point is that in the examples so far the “Conditions” of the dfc-dqlhints.xml file evaluate to a binary decision on the DisableFTDQL and the DQLHint. To apply a different rule based on some different conditions, then it’s a matter of adding additional rules. The following example shows how to apply different rules based on whether subject or object\_name are used in a LIKE clause for the query.

```

<?xml version="1.0"?>
<!DOCTYPE RuleSet PUBLIC "dfc-dqlhints.dtd" "dfc-dqlhints.dtd">
  <RuleSet>
    <Rule>
      <Condition>
        <Where condition="any">
          <Attribute operator="like">subject</Attribute>
        </Where>
      </Condition>
      <DQLHint> ENABLE(SQL_DEF_RESULT_SET 100, NOFTDQL) </DQLHint>
      <DisableFTDQL/>
    </Rule>
    <Rule>
      <Condition>
        <Where condition="any">
          <Attribute operator="like">object_name</Attribute>
        </Where>
      </Condition>
      <DQLHint> ENABLE(SQL_DEF_RESULT_SET 10) </DQLHint>
      <DisableFTDQL/>
    </Rule>
  </RuleSet>

```

This is very similar to the previous definition in semantics. However, much care should be taken in this case. In the SP2/3 Patch 9950 of DFC it is possible that if multiple rules apply then the software could generate a query with multiple ENABLE statements (causing a DQL syntax error). The rules must apply to a single query in a mutually exclusive manner for this patch.

---

## Case No. 5: Turning off FTDQL when a date range is used

Date range queries can be problematic for full-text index. To turn off FTDQL for date range queries, create the following definition for the dfc-dqlhints.xml file:

```
<?xml version="1.0"?>
<!DOCTYPE RuleSet PUBLIC "dfc-dqlhints.dtd" "dfc-dqlhints.dtd">
  <RuleSet>
    <Rule>
      <Condition>
        <Where condition="any">
          <Attribute>r_creation_date</Attribute>
          <Attribute>r_modify_date</Attribute>
          <Attribute>r_access_date</Attribute>
        </Where>
      </Condition>
      <DisableFTDQL/>
    </Rule>
  </RuleSet>
```

This will cause ENABLE(NOFTDQL) to be applied if any WHERE clause portion has a reference to r\_creation\_date, r\_modify\_date, or r\_access\_date.

## Case No. 6: Turning off FTDQL in case of FOLDER(DESCEND)

Problematic FOLDER(DESCEND) queries can be handled in the following manner. First, turn off FTDQL for all queries (across all repositories) using the following definition:

```
<?xml version="1.0"?>
<!DOCTYPE RuleSet PUBLIC "dfc-dqlhints.dtd" "dfc-dqlhints.dtd">
<RuleSet>
  <Rule>
    <Condition>
      <Docbase>
        <Name descend="true">*</Name>
      </Docbase>
    </Condition>
    <DisableFTDQL>
  </Rule>
</RuleSet>
```

To turn it off for a specific repository (say for Docbase = 'support'), create a definition that looked like:

```
<?xml version="1.0"?>
<!DOCTYPE RuleSet PUBLIC "dfc-dqlhints.dtd" "dfc-dqlhints.dtd">
<RuleSet>
  <Rule>
    <Condition>
      <Docbase>
        <Name descend="true">support</Name>
      </Docbase>
    </Condition>
    <DisableFTDQL>
  </Rule>
</RuleSet>
```

---

## Case No. 7: Turning off FTDQL for a specific repository

Note that the example in the previous section can be made more generalized in targeting when to disable full-text indexing for a repository. Suppose we have two repositories (dm\_notes and support) and we wanted to turn off FTDQL for the “support” repository (when date-range queries happen) and fire off a different hint for the dm\_notes repository under similar (but not identical) conditions. Modify the file in the following manner:

```
<?xml version="1.0"?>
<!DOCTYPE RuleSet PUBLIC "dfc-dqlhints.dtd" "dfc-dqlhints.dtd">
  <RuleSet>
    <Rule>
      <Condition>
        <Where condition="any">
          <Attribute>r_creation_date</Attribute>
          <Attribute>r_modify_date</Attribute>
          <Attribute>r_access_date</Attribute>
        </Where>
        <Docbase>
          <Name>support</Name>
        </Docbase>
      </Condition>
      <DisableFTDQL/>
    </Rule>
    <Rule>
      <Condition>
        <Where condition="any">
          <Attribute>r_creation_date</Attribute>
          <Attribute>r_modify_date</Attribute>
        </Where>
        <Docbase>
          <Name>dm_notes</Name>
        </Docbase>
      </Condition>
      <DQLHint>ENABLE(SQL_DEF_RESULT_SET 10)</DQLHint>
    </Rule>
  </RuleSet>
```

## Known limitations of Patch 9950

- If multiple rules succeed and if multiple DQLHint lines are defined for each rule, then the software could generate a query with a syntax error.
- If the “descend” flag is used and a DQLHint is defined then an ENABLE(NOFTDQL) could be issued incorrectly.

## Conclusion

In some cases, the database performs queries efficiently and full-text adds no value. Sending all Advanced Search queries to the full-text could unnecessarily burden it. This white paper covered some examples of when it's desirable to change the default behavior of Advanced Search to disable the FTDQL hint. Using the dfc-dqlhints.xml configuration file will influence the behavior of the query builder of Advanced Search. In addition, it has been noted that the behavior of the system is relative to 5.3 SP2 and SP3 using Patch 9950 and differs slightly from 5.3 SP1. The behavior of 5.3 SP1 is addressed in other documents.

---

## Appendix: DTD definition

This does not need to be present or within reach of the dfc-dqlhints.xml.

```
<!ELEMENT RuleSet (Rule*)>
<!ELEMENT Rule (Condition?, DQLHint?, DisableFullText?,
DisableFTDQL?)>
<!ELEMENT Condition (Select?, From?, Where?, Docbase?)>
<!ELEMENT DQLHint (#PCDATA)>
<!ELEMENT DisableFullText EMPTY>
<!ELEMENT DisableFTDQL EMPTY>
<!ELEMENT Select (Attribute+)>
  <!ATTLIST Select condition (all | any) \"all\">
<!ELEMENT From (Type+)>
  <!ATTLIST From condition (all | any) \"all\">
<!ELEMENT Where (Attribute+)>
  <!ATTLIST Where condition (all | any) \"all\">
<!ELEMENT Docbase (Name+)>
<!ELEMENT Attribute (#PCDATA)>
  <!ATTLIST Attribute operator
    (equal | not_equal | greater_than | greater_equal | less_than
| less_equal | like | not_like | is_null |
is_not_null | in | not_in | between)
    #IMPLIED>
<!ELEMENT Type (#PCDATA)>
<!ELEMENT Name (#PCDATA)>
<!ATTLIST Name descend (true | false) #IMPLIED>
```