# 

# Building a "Faceted Navigation" Application with Oracle Text 12c

Contents

[Building a "Faceted Navigation" Application with Oracle Text 12c 1](#_Toc329713896)

[Introduction 2](#_Toc329713897)

[The MVDATA Section 4](#_Toc329713898)

[The "Camera Shop" Example 4](#_Toc329713899)

[Recap 10](#_Toc329713900)

[Doing away with SELECT and CONTAINS - Using the Result Set Interface 10](#_Toc329713901)

[Using SDATA Columns 13](#_Toc329713902)

[Advanced - Bulk updates to MVDATA 15](#_Toc329713903)

[Handing Dates - "Bucketing" Date Ranges 22](#_Toc329713904)

[Hierarchical Facets 24](#_Toc329713905)

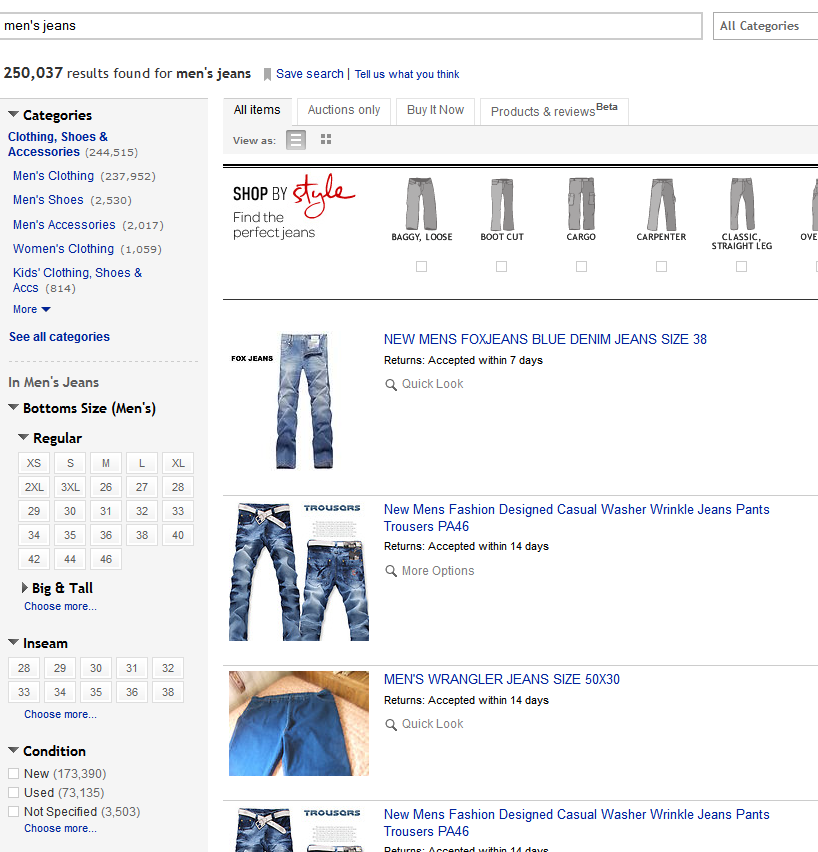
["Top N" facets 24](#_Toc329713906)

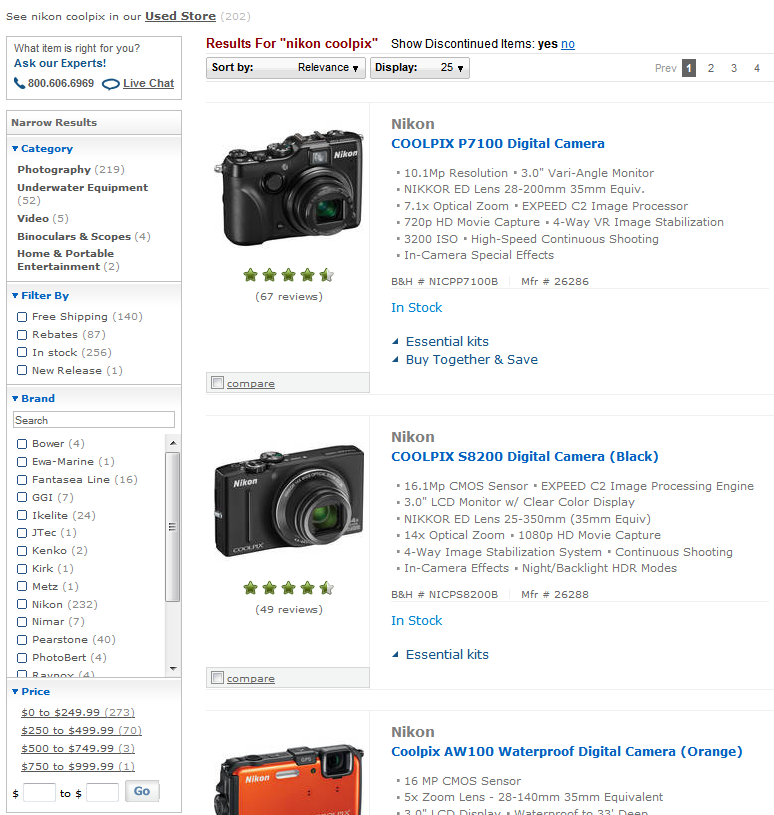
## Introduction

Faceted navigation is familiar to all of us from many sites on the web, particularly shopping sites.

When looking for items to purchase, it is common to first perform a full-text search, for example I might search for "men's jeans". The site will then present me with a list of results, with a list of characteristics, or "facets" listed alongside them. Commonly, each facet will have a range of values, and a count will be displayed of the total number of hits for each facet value.

A couple of examples follow:





The sites will normally allow you to "click through" on a facet value in order to select only the results with that facet, or alternatively to select a number of facet values to restrict the search to those values. Such methods are a powerful way to navigate through a set of results. Indeed, in some cases text search may not be used at all, but the user may navigate to a final result simply by clicking on successively narrower facet selections.

Creating an application which uses faceted navigation is not easy using standard SQL. Generally, you only want to display the first page of results, and yet you want to get summary counts for all the facets involved. This can be achieved by either:

1. Fetching all the results, including facet values, and counting the facet values within the application
2. Running multiple queries - one to fetch the first page of results, and then summary queries for each of the facets.

Neither of these are particularly simple, nor easy to program. So in 12c, Oracle Text has introduced new data models and extended some existing capabilities to make faceted navigation much easier.

## The MVDATA Section

Users of Oracle Text will be familiar with various section types currently in use:

* Zone sections - general purpose text sections
* Field sections - high performance text sections
* SDATA sections - "*S*tructured *DATA*" sections for numbers and dates
* NDATA sections - for name searches (and other inexact matches)

Oracle Text 12c introduces the MVDATA section (for *M*ulti *V*alued *DATA*). This, as its name suggests, may have multiple values in the same section.

MVDATA accepts only numeric values. In order to build a faceted navigation application, we associate each facet value with a unique number. We can chose to have an MVDATA section for each facet type ("clothing style", "waist size", "color", etc.) or we can just have a single MVDATA section for ALL facet types. This second approach is simpler and is the one we will concentrate on for the rest of this document.

It is important to understand that Oracle Text does not know or care what any particular number in an MVDATA section represents. It is entirely up to the application to keep track of the mapping between MVDATA numbers and "real world" characteristics.

MVDATA values can be loaded in several ways:

1. The traditional xml-style section can be used within the main text. For example the text might contain:  
   <author>J Smith</author>  
   <title>My Book</title>  
   <mvvalues>7,12,15</mvvalues>  
   where 7,12 and 15 represent different facet values associated with this book.
2. A separate column can be defined as an MVDATA\_COLUMN and can contain a comma-separated list of values.
3. The facet values may be bulk-loaded using CTX\_DDL.INSERT\_MVDATA\_VALUES after the main index has been created.

## The "Camera Shop" Example

We will look at the coding needed to produce a simple faceted navigation application. We are not going to consider *any* presentation or user interface issues - this example is purely about the underlying code necessary to produce such an application. It is anticipated that further examples will show how to embed this code into an ApEx application, but in this document we will be looking at the code only.

The code, which is also available in a single file at <TODO: file location> is designed to be run in SQL\*Plus or SQL Developer. Sometimes SQL\*Plus constructs will be used which are not necessary or relevant in SQL Developer. Things that are in blue need to be entered, things in red are output from SQL\*Plus. For brevity and clarity, DROP statements have not been included. Run this in a newly created schema, or see the full script which includes DROP statements so the examples are re-runnable.

For the demo, we will have three facet types:

* Resolution - the number of megapixels
* Body Type - eg. Compact, Bridge or Single-lens Reflex (SLR)
* Price - divided into ranges such as 100-150, 150-200 etc.

Remember that these facet types are known only to the application - Oracle Text deals only with numbers and does not know what the numbers represent. We'll create a table which is going to keep track of the "meaning" of each facet value - or facet\_id as we're calling it here:

CREATE TABLE facettable(   
 facet\_id NUMBER, -- this will be the number used in MVDATA  
 facet\_name VARCHAR2(40), -- facet name, or category  
 facet\_value VARCHAR2(20), -- facet value, or a division in category  
 facet\_group NUMBER -- defines the display order for category  
);

Now we'll populate this with our facet ids, and what they actually represent. Obviously a real application would have many more rows than this!

INSERT INTO facettable values( 1, 'Resolution', '10MP', 1 );  
INSERT INTO facettable values( 2, 'Resolution', '12MP', 1 );  
INSERT INTO facettable values( 3, 'Resolution', '15MP', 1 );  
INSERT INTO facettable values( 4, 'Resolution', '> 15MP', 1 );  
  
INSERT INTO facettable values( 5, 'Price', '100-150', 2 );  
INSERT INTO facettable values( 6, 'Price', '150-200', 2 );  
INSERT INTO facettable values( 7, 'Price', '200-250', 2 );  
INSERT INTO facettable values( 8, 'Price', '> 250', 2 );  
  
INSERT INTO facettable values( 9, 'Body Type', 'Compact', 3 );  
INSERT INTO facettable values( 10, 'Body Type', 'Bridge', 3 );  
INSERT INTO facettable values( 11, 'Body Type', 'SLR', 3 );

Strictly speaking, we're duplicating some information here - facet\_name and facet\_group are the same for any row so we could have normalized them out, but this way our example should be a bit clearer.

Now we're going to create the actual data our application is searching on. This will be in a table called "PRODUCTS", which we'll load with a selection of products including their facets:

CREATE TABLE products( text varchar2(255) );  
  
INSERT INTO products values('Nikon C400 <facetlist>1,5,9</facetlist>');  
INSERT INTO products values('Nikon C401 (Nikon USA)<facetlist>1,5,9</facetlist>');  
INSERT INTO products values('Nikon B40 <facetlist>1,6,10</facetlist>');  
INSERT INTO products values('Nikon SLRX <facetlist>4,8,11</facetlist>');

You can see we have just four products. Each has an MVDATA section (we'll define it soon) with the facet values for that product. For example the Nikon C400 has a resolution of 10MP (facet \_id = 1), a price in the range 100-150 (facet\_id = 5) and a body type of "compact" (facet\_id = 9).

Now we need to define the MVDATA section and create the index. Note that to use MVDATA sections, you *must* turn on the BIG\_IO option, so we'll define a storage option in order to set that.

EXECUTE ctx\_ddl.drop\_section\_group ( 'sec\_grp' )  
EXECUTE ctx\_ddl.create\_section\_group( 'sec\_grp', 'BASIC\_SECTION\_GROUP' )  
EXECUTE ctx\_ddl.add\_mvdata\_section ( 'sec\_grp', 'facetlist', 'facetlist' )

EXECUTE ctx\_ddl.drop\_preference ( 'storage' )  
EXECUTE ctx\_ddl.create\_preference ( 'storage', 'BASIC\_STORAGE' )  
EXECUTE ctx\_ddl.set\_attribute ( 'storage', 'BIG\_IO', 'TRUE' )  
  
CREATE INDEX productsindex ON products(text )INDEXTYPE IS ctxsys.context  
PARAMETERS( 'section group sec\_grp storage storage' );

Now we have an index which is searchable. Let's do a simple search which specifies that we want to find products with "nikon" in the text, 10MP resolution, and a price in either of the ranges 100-150 or 150-200. When all the MVDATA values need to be present (even if there is only one) we use the MVAND operator. When any of the MVDATA values can be present we use the MVOR operator. 10MP resolution is facet\_id = 1, price 100-150 is facet\_id = 5 and price 150-200 is facet\_id = 6, so our query would be:

SELECT \* FROM products WHERE CONTAINS( text,   
 'nikon AND MVAND(facetlist, (1) ) AND MVOR(facetlist, (5,6))'  
) > 0;

Let's run that:

TEXT  
--------------------------------------------------------------------------------  
Nikon C400 <facetlist>1,5,9</facetlist>  
Nikon C401 (Nikon USA)<facetlist>1,5,9</facetlist>  
Nikon B40 <facetlist>1,6,10</facetlist>

So we've fetched the data we want.

But in order to create our faceted navigation application, we need to summarize the facet information, so that for example we can show a count of all the results which are in the price range 100-150. We can't do that in plain SQL, instead we must define what's known as a "*Result Set Descriptor* " (RSD). The RSD is a block of XML-style code which defines the information we want to get back. A result set descriptor can be used in a *query template.* Query templates have been around a while, if you're not familiar with them it's worth checking the documentation.

The query template is used in a SQL "contains" clause, and the results of the query are returned in the normal manner - except for the information which was requested in the Result Set Descriptor. This can be accessed from a *package variable* in the ctx\_query PL/SQL package. The variable is named result\_set\_document.

Let's see that in action. We'll use a SQL\*Plus bind variable to hold our query template, and assign the value to it in an anonymous PL/SQL block:

VARIABLE query CLOB  
  
BEGIN  
 :query := '  
<query>  
 <textquery>  
 nikon AND MVAND( facetlist, (1) ) AND MVOR( facetlist, (5,6) )  
 </textquery>  
 <score datatype="INTEGER"/>  
 <ctx\_result\_set\_descriptor>  
 <group mvdata = "facetlist">  
 <group\_values>  
 <value id = "1"/>  
 <value id = "2"/>  
 <value id = "3"/>  
 <value id = "4"/>  
 <value id = "5"/>  
 <value id = "6"/>  
 <value id = "7"/>  
 <value id = "8"/>  
 <value id = "9"/>  
 <value id = "10"/>  
 <value id = "11"/>  
 </group\_values>  
 <count/>  
 </group>  
 </ctx\_result\_set\_descriptor>  
</query>  
';  
END;  
/

We can see the Result Set Descriptor here. It's listing the values that it wants to fetch back (in this case, all of them), and specifying that it wants a count for each group.

We'll run a contains query with this query template. First of all, though, we need to provide a temporary LOB for the result set document to be written to:

EXECUTE dbms\_lob.createTemporary( ctx\_query.result\_set\_document, true )

Now run the query:

select \* from products where contains( text, :query ) > 0;

(Note: if running this in an application, it is important to close the cursor used before trying to fetch the result set document)

The output of this query would be the same as earlier, because the main query specified is the same as we used in the simple CONTAINS example:

TEXT  
--------------------------------------------------------------------------------  
Nikon C400 <facetlist>1,5,9</facetlist>  
Nikon C401 (Nikon USA)<facetlist>1,5,9</facetlist>  
Nikon B40 <facetlist>1,6,10</facetlist>

But more interestingly, what's been written into ctx\_query.result\_set\_document? It's not trivial to look at that in SQL\*Plus. We can do

SET long 50000  
VARIABLE res CLOB  
EXECUTE :res := ctx\_query.result\_set\_document  
PRINT res

which produces

RES  
--------------------------------------------------------------------------------  
<ctx\_result\_set><groups mvdata="FACETLIST"><group value="1"><count>3</count></gr  
oup><group value="5"><count>2</count></group><group value="9"><count>2</count></  
group><group value="6"><count>1</count></group><group value="10"><count>1</count  
></group><group value="2"><count>0</count></group><group value="3"><count>0</cou  
nt></group><group value="4"><count>0</count></group><group value="7"><count>0</c  
ount></group><group value="8"><count>0</count></group><group value="11"><count>0  
</count></group></groups></ctx\_result\_set>

But it would be far more readable if viewed that as an *xmlType* object - that way it would get automatically formatted for us:

SELECT xmltype( :res ) FROM dual;

This produces the output:

XMLTYPE(:RES)  
--------------------------------------------------------------------------------  
<ctx\_result\_set>  
 <groups mvdata="FACETLIST">  
 <group value="1">  
 <count>3</count>  
 </group>  
 <group value="5">  
 <count>2</count>  
 </group>  
 <group value="9">  
 <count>2</count>  
 </group>  
 <group value="6">  
 <count>1</count>  
 </group>  
 <group value="10">  
 <count>1</count>  
 </group>  
 <group value="2">  
 <count>0</count>  
 </group>  
 <group value="3">  
 <count>0</count>  
 </group>  
 <group value="4">  
 <count>0</count>  
 </group>  
 <group value="7">  
 <count>0</count>  
 </group>  
 <group value="8">  
 <count>0</count>  
 </group>  
 <group value="11">  
 <count>0</count>  
 </group>  
 </groups>  
</ctx\_result\_set>

That's far more readable for debugging, though we don't necessarily need to do that in an application.

OK, so we've got a list of facet ids (the group value="N" part) and associated counts. What are we going to do with them? If our application is XML-based, maybe we could use that XML directly, perhaps transforming it with XSLT into something to display on-screen. But if our application is SQL based, we probably want to convert the XML into something that looks more like a SQL table. That will also make it much easier to join back to our facettable which tells us what all the facet ids actually mean. We can use the XML DB function XMLTABLE to convert from XML to a SQL table.

First we need to put our result set document into a table with an XMLType column:

CREATE TABLE res\_output ( res XMLTYPE );

EXECUTE INSERT INTO res\_output values ( xmltype(:res) )

Then we'll use the XMLTABLE function:

SELECT rs.facet\_id, rs.facet\_count  
 FROM res\_output r, XMLTABLE   
('/ctx\_result\_set/groups/group'  
PASSING r.res  
 COLUMNS   
 facet\_id NUMBER PATH '@value',  
 facet\_count NUMBER PATH 'count/text()'  
) as rs;

This will produce the output:

FACET\_ID FACET\_COUNT  
---------- -----------  
 1 3  
 5 2  
 9 2  
 6 1  
 10 1  
 2 0  
 3 0  
 4 0  
 7 0  
 8 0  
 11 0  
  
11 rows selected.

which is great - there are all our facet values and the number of counts for each.

An alternate form which uses the SQL\*Plus bind variable res and avoids the need for an extra tables would be as follows. The output is identical:

SELECT rs.facet\_id, rs.facet\_count  
 FROM XMLTABLE  
('/ctx\_result\_set/groups/group'  
PASSING XMLTYPE(:res)  
 COLUMNS  
 facet\_id NUMBER PATH '@value',  
 facet\_count NUMBER PATH 'count/text()'  
) as rs

In many of the following examples, we have used a temporary table "res\_output". All of the examples can be modified to use a bind variable as above if preferred.

We can use SQL to join the previous query back to the facettable, and to remove all the facets with zero counts, like this:

break on FACET\_NAME skip 1  
  
SELECT ft.facet\_name, ft.facet\_value, rs.facet\_count  
 FROM facettable ft, XMLTABLE   
('/ctx\_result\_set/groups/group'  
PASSING XMLTYPE(:res)  
 COLUMNS   
 facet\_id NUMBER PATH '@value',  
 facet\_count NUMBER PATH 'count/text()'  
) as rs   
where rs.facet\_id = ft.facet\_id  
and rs.facet\_count > 0  
order by ft.facet\_group, rs.facet\_count desc;

The output from that looks like:  
  
FACET\_NAME FACET\_VALUE FACET\_COUNT

--------------------------- ------------- -----------  
Resolution 10MP 3  
  
Price 100-150 2  
 150-200 1  
  
Body Type Compact 2  
 Bridge 1

which looks much more likes something we could use in an application.

## Recap

What have we learned so far?

1. MVDATA sections allow you to store multiple numeric values in a single section.
2. The "meaning" of these values is entirely down to the application
3. You can include MVDATA values in a CONTAINS clause using the MVOR and MVAND operators
4. You can fetch back summary information for MVDATA values using a Result Set Descriptor which can be included in a query template
5. The result set document is returned in the PL/SQL package variable ctx\_query.result\_set\_document
6. We can use XML DB functions to process the XML-based result set document into a SQL table for further processing with SQL.

## Doing away with SELECT and CONTAINS - Using the Result Set Interface

The Result Set Interface (RSI) is also new in 12c. This allows you to run queries without using a SQL SELECT at all. With the result set interface, you define a Result Set Descriptor (much as we did in the query template example above), and then call ctx\_query.result\_set with that descriptor. The procedure returns a result set document which normally contains two components:

1. A "hitlist". That is, the top N documents that satisfy the query
2. Summary information about the entire result set.

The hitlist part is somewhat similar to what you'd get from a normal SELECT cursor, but can only return

* ROWID
* SCORE
* SDATA values

The summary information typically contains group counts for SDATA or MVDATA values.

Let's define two SQL\*Plus variables, one which will hold the Result Set Descriptor, and one which will hold the output Result Set Document:

variable rsd clob  
variable rsout clob

Then we'll populate the result set descriptor with the appropriate XML:

begin  
 :rsd := '  
<ctx\_result\_set\_descriptor>  
 <count />  
 <hitlist start\_hit\_num="1" end\_hit\_num="5" order="score desc">  
 <score />  
 <rowid />  
 </hitlist>  
 <group mvdata = "facetlist">  
 <group\_values>  
 <value id = "1"/>  
 <value id = "2"/>  
 <value id = "3"/>  
 <value id = "4"/>  
 <value id = "5"/>  
 <value id = "6"/>  
 <value id = "7"/>  
 <value id = "8"/>  
 <value id = "9"/>  
 <value id = "10"/>  
 <value id = "11"/>  
 </group\_values>  
 <count/>  
 </group>  
</ctx\_result\_set\_descriptor>  
';  
 dbms\_lob.createtemporary(:rsout, true);  
end;  
/

You will see this is very similar to the earlier RSD that we used in a query template, except that it now contains a "hitlist" section as well, specifying that we want to fetch the score and rowid for the first five hits, ordered by score (highest first).

If we pass this into ctx\_query.result\_set, we will get a Result Set Document returned containing our results. As before, we'll put this into an XMLType field so it can be more easily read:

exec ctx\_query.result\_set( 'productsindex', 'nikon', :rsd, :rsout)  
  
delete from res\_output;  
insert into res\_output values (xmltype(:rsout));  
  
select \* from res\_output;

which will output:

RES  
--------------------------------------------------------------------------------  
<ctx\_result\_set>  
 <hitlist>  
 <hit>  
 <score>6</score>  
 <rowid>AAAW3NAABAAAXSBAAB</rowid>  
 </hit>  
 <hit>  
 <score>3</score>  
 <rowid>AAAW3NAABAAAXSBAAA</rowid>  
 </hit>  
 <hit>  
 <score>3</score>  
 <rowid>AAAW3NAABAAAXSBAAC</rowid>  
 </hit>  
 <hit>  
 <score>3</score>  
 <rowid>AAAW3NAABAAAXSBAAD</rowid>  
 </hit>  
 </hitlist>  
 <count>4</count>  
 <groups mvdata="FACETLIST">  
 <group value="1">  
 <count>3</count>  
 </group>  
 <group value="5">  
 <count>2</count>  
 </group>  
 <group value="9">  
 <count>2</count>  
 </group>  
 <group value="4">  
 <count>1</count>  
 </group>  
 <group value="6">  
 <count>1</count>  
 </group>  
 <group value="8">  
 <count>1</count>  
 </group>  
 <group value="10">  
 <count>1</count>  
 </group>  
 <group value="11">  
 <count>1</count>  
 </group>  
 <group value="2">  
 <count>0</count>  
 </group>  
 <group value="3">  
 <count>0</count>  
 </group>  
 <group value="7">  
 <count>0</count>  
 </group>  
 </groups>  
</ctx\_result\_set>

As before, we could process this using XML DB functions, but we won't go further into that here.

## Using SDATA Columns

Because our hitlist in RSI is limited to ROWID, SCORE and SDATA fields, it makes sense to put anything that we might want to fetch back into SDATA fields, so we don't need to do separate lookups using ROWID to get the data to display.

We can create SDATA fields in two ways. Firstly we can include them in the main text in the form <fieldname>data</fieldname>. A common way to do that might be to use the MULTI\_COLUMN\_DATASTORE to copy the data from another column in the table.

Alternatively we can use the FILTER BY clause when we create the index. Originally introduced as part of "Composite Domain Indexes" feature in 11g, this clause will automatically create an SDATA section from any columns specified.

Before running this section, drop the PRODUCTS table you created in the previous example. This time, rather than putting the facetlist directly into the text we're going to store it in a separate column and index it in the text using the MULTI\_COLUMN\_DATASTORE. We could chose, as mentioned above, to create our SDATA sections the same way, but we won't - we'll use FILTER BY to create the SDATA sections from the columns.

CREATE TABLE products(   
 model varchar2(249), -- max length for SDATA col  
 price number,  
 facetlist varchar2(255)  
);  
  
-- Here is the data for indexing.  
  
INSERT INTO products values( 'Nikon C400', 129, '1,5,9' );  
INSERT INTO products values( 'Nikon C401 (Nikon USA)', 149, '1,5,9' );  
INSERT INTO products values( 'Nikon B40', 190, '1,6,10' );  
INSERT INTO products values( 'Nikon SLRX', 445, '4,8,11' );

Here's how we would create the index using FILTER BY in the "create index" statement:  
  
EXECUTE ctx\_ddl.create\_preference ( 'mc\_ds', 'MULTI\_COLUMN\_DATASTORE')  
EXECUTE ctx\_ddl.set\_attribute ( 'mc\_ds', 'COLUMNS', 'model, facetlist')  
  
EXECUTE ctx\_ddl.create\_section\_group( 'sec\_grp', 'BASIC\_SECTION\_GROUP' )  
   
EXECUTE ctx\_ddl.add\_mvdata\_section ( 'sec\_grp', 'facetlist', 'facetlist' )  
  
EXECUTE ctx\_ddl.create\_preference ( 'storage', 'BASIC\_STORAGE' )  
EXECUTE ctx\_ddl.set\_attribute ( 'storage', 'BIG\_IO', 'TRUE' )  
  
CREATE INDEX productsindex ON products(model )INDEXTYPE IS ctxsys.context  
FILTER BY model, price  
PARAMETERS( 'datastore mc\_ds section group sec\_grp storage storage' );

Alternatively, we could have copied model and price in with the multi\_column\_datastore, and defined these as SDATA sections like so:

EXECUTE ctx\_ddl.create\_preference ( 'mc\_ds', 'MULTI\_COLUMN\_DATASTORE')  
EXECUTE ctx\_ddl.set\_attribute ( 'mc\_ds', 'COLUMNS', 'model as text, model, price, facetlist')  
  
EXECUTE ctx\_ddl.create\_section\_group( 'sec\_grp', 'BASIC\_SECTION\_GROUP' )  
EXECUTE ctx\_ddl.add\_mvdata\_section ( 'sec\_grp', 'facetlist', 'facetlist' )  
EXECUTE ctx\_ddl.add\_sdata\_section ( 'sec\_grp', 'model', 'model' );  
EXECUTE ctx\_ddl.add\_sdata\_section ( 'sec\_grp', 'price', 'price' );  
  
EXECUTE ctx\_ddl.create\_preference ( 'storage', 'BASIC\_STORAGE' )  
EXECUTE ctx\_ddl.set\_attribute ( 'storage', 'BIG\_IO', 'TRUE' )  
  
CREATE INDEX productsindex ON products(model )INDEXTYPE IS ctxsys.context  
PARAMETERS( 'datastore mc\_ds section group sec\_grp storage storage' );

Note the column\_list in the MULTI\_COLUMN\_DATASTORE : model appears twice. That's because we want it to be indexed as ordinary indexable text, as well as being treated as an SDATA section so we can fetch it in the hitlist.

Whichever way we create the index, the query we run against it is identical. As before, we'll create a Result Set Descriptor, but this time as well as score and rowid, we'll request the SDATA sections model and price. Then we'll call ctx\_query.result\_set as before:

begin  
 :rsd := '  
<ctx\_result\_set\_descriptor>  
 <count />  
 <hitlist start\_hit\_num="1" end\_hit\_num="5" order="score desc">  
 <score />  
 <rowid />  
 <sdata name="model"/>  
 <sdata name="price"/>  
 </hitlist>  
 <group mvdata = "facetlist">  
 <group\_values>  
 <value id = "1"/>  
 <value id = "2"/>  
 <value id = "3"/>  
 <value id = "4"/>  
 <value id = "5"/>  
 <value id = "6"/>  
 <value id = "7"/>  
 <value id = "8"/>  
 <value id = "9"/>  
 <value id = "10"/>  
 <value id = "11"/>  
 </group\_values>  
 <count/>  
 </group>  
</ctx\_result\_set\_descriptor>  
';  
 dbms\_lob.createtemporary(:rsout, true);  
end;  
/  
  
exec ctx\_query.result\_set( 'productsindex', 'nikon', :rsd, :rsout)  
delete from res\_output;  
insert into res\_output values (xmltype(:rsout));  
select \* from res\_output;

The output from this (abbreviated here for space reasons) is:

RES  
--------------------------------------------------------------------------------  
<ctx\_result\_set>  
 <hitlist>  
 <hit>  
 <score>6</score>  
 <rowid>AAAXUbAABAAAXSBAAB</rowid>  
 <sdata name="MODEL">Nikon C401 (Nikon USA)</sdata>  
 <sdata name="PRICE">149</sdata>  
 </hit>  
 <hit>  
 <score>3</score>  
 <rowid>AAAXUbAABAAAXSBAAA</rowid>  
 <sdata name="MODEL">Nikon C400</sdata>  
 <sdata name="PRICE">129</sdata>  
 </hit>  
 . . .  
 </hitlist>  
 <count>4</count>  
 <groups mvdata="FACETLIST">  
 <group value="1">  
 <count>3</count>  
 </group>  
 <group value="5">  
 <count>2</count>  
 </group>  
 . . .  
 </groups>  
</ctx\_result\_set>

We can see that now we're getting back the model name and price as part of the hitlist section. We could also have fetched a "snippet" there, but that's a topic for another paper. All this should give us all the data we need to build a fully-fledged faceted navigation application using Oracle Text.

## Advanced - Bulk updates to MVDATA

In the previous examples, we've dealt only with MVDATA fields which are part of the text of the document. Sometimes it can be advantageous to update MVDATA values separately from the text. There are the procedures which can be used for this in the CTX\_DDL package:

INSERT\_MVDATA\_VALUES  
UPDATE\_MVDATA\_SET  
DELETE\_MVDATA\_VALUES

As with any other CTX\_DDL procedure, if you wish to call these from a PL/SQL procedure, you must explicitly grant EXECUTE privilege on CTX\_DDL to your user - having the role CTXAPP is not sufficient to call procedures from within PL/SQL procedures.

INSERT\_MVDATA\_VALUES and UPDATE\_MVDATA\_SET are similar, the difference being that the UPDATE procedure clears any existing values for the specified section and rowid(s).

All of the procedures take the same set of arguments:

|  |  |  |
| --- | --- | --- |
| **Name** | **Data** **Type** | **Description** |
| IDX\_NAME | VARCHAR2 | The name of your index |
| SECTION\_NAME | VARCHAR2 | The name of the MVDATA section to update |
| MVDATA\_VALUES | SYS.OdciNumberList | The MVDATA values |
| MVDATA\_ROWIDS | SYS.OdciRidList | A list of ROWIDs to update |
| PART\_NAME | VARCHAR2 | Partition name (for LOCAL indexes) |

OdciNumberList and OdciRidList are PL/SQL tables. They can be initialized to have a fixed number of elements, and/or extended as necessary. Very often you will only write a single value to one or other of these tables.

All of these procedures are transactional - any changes will be immediately visible in the index without the need to SYNChronize the index.

Let's look at an example where we use a facet to track the stock of a product. Stock levels often change frequently, and it's easier and faster to update the relevant facet by means of the above functions than to modify the indexable text of the document and reindex it.

This example builds on the previous example, but introduces two new columns to the PRODUCTS table, STOCK and REL\_DATE. REL\_DATE will be used in a later example.

We're going to add an extra column to the facet\_table for this example, so we can order the final results by something other than the facet count:

CREATE TABLE facettable(   
 facet\_id NUMBER, -- this will be the number used in MVDATA  
 facet\_name VARCHAR2(40), -- facet name, or category  
 facet\_value VARCHAR2(20), -- facet value or division within category  
 facet\_group NUMBER, -- defines the display order for categories  
 facet\_display\_order NUMBER -- order to display facets (new column)  
);

Now we'll load the facet values. For brevity, I won't include all the previous values. You can, if you like, include them with zero values (or values of your choice) for the facet\_display\_order column.

INSERT INTO facettable values( 12, 'Stock Level', '0', 4, 7 );   
INSERT INTO facettable values( 13, 'Stock Level', '1', 4, 6 );   
INSERT INTO facettable values( 14, 'Stock Level', '2', 4, 5 );   
INSERT INTO facettable values( 15, 'Stock Level', '3', 4, 4 );   
INSERT INTO facettable values( 16, 'Stock Level', '4', 4, 3 );   
INSERT INTO facettable values( 17, 'Stock Level', '5', 4, 2 );   
INSERT INTO facettable values( 18, 'Stock Level', '> 5', 4, 1 );

Now we'll create the new products table:

CREATE TABLE products(   
 model VARCHAR2(249), -- max length for SDATA col  
 price NUMBER,  
 stock NUMBER,  
 rel\_date DATE,  
 facetlist VARCHAR2(255)  
);

Now we'll create the data in the products table, and the index on it. Note that the facet\_list column does NOT include any of the "new" facet values for stock level - they will be populated after loading by a one-off procedure.

INSERT INTO products VALUES( 'Nikon C400', 129, 10, SYSDATE-10, '1,5,9' );  
INSERT INTO products VALUES( 'Nikon C401 (Nikon USA)', 149, 5, SYSDATE-8, '1,5,9' );  
INSERT INTO products VALUES( 'Nikon B40', 190, 2, SYSDATE-3, '1,6,10' );  
INSERT INTO products VALUES( 'Nikon SLRX', 445, 0, SYSDATE, '4,8,11' );  
  
EXECUTE ctx\_ddl.drop\_preference ( 'mc\_ds' )  
EXECUTE ctx\_ddl.create\_preference ( 'mc\_ds', 'MULTI\_COLUMN\_DATASTORE')  
EXECUTE ctx\_ddl.set\_attribute ( 'mc\_ds', 'COLUMNS', 'model, facetlist')  
  
EXECUTE ctx\_ddl.drop\_section\_group ( 'sec\_grp' )  
EXECUTE ctx\_ddl.create\_section\_group( 'sec\_grp', 'BASIC\_SECTION\_GROUP' )  
   
EXECUTE ctx\_ddl.add\_mvdata\_section ( 'sec\_grp', 'facetlist', 'facetlist' )  
EXECUTE ctx\_ddl.add\_mvdata\_section ( 'sec\_grp', 'stockfacets', 'stockfacets' );  
EXECUTE ctx\_ddl.add\_mvdata\_section ( 'sec\_grp', 'datefacets', 'datefacets' );  
  
EXECUTE ctx\_ddl.drop\_preference ( 'storage' )  
EXECUTE ctx\_ddl.create\_preference ( 'storage', 'BASIC\_STORAGE' )  
EXECUTE ctx\_ddl.set\_attribute ( 'storage', 'BIG\_IO', 'TRUE' )  
  
CREATE INDEX productsindex ON products(model )INDEXTYPE IS ctxsys.context  
FILTER BY model, price  
PARAMETERS( 'datastore mc\_ds section group sec\_grp storage storage' );

Note that we created a new MVDATA section called "stockfacets". This isn't strictly necessary - we could have used the same facet section as before, but this will make the example slightly easier. And we will definitely need the new "datefacets" section later.

Next we'll create a procedure that should be run AFTER the index has been created, which will initially populate the stockfacet MVDATA section:

CREATE OR REPLACE procedure set\_stock\_levels IS  
 rowids SYS.odciRidList;  
 facets SYS.odciNumberList;  
 stock\_facet INTEGER;  
BEGIN  
 -- initialize collections  
 rowids := SYS.odciRidList();  
 facets := SYS.odciNumberList();  
  
 FOR c IN ( SELECT rowid, stock FROM products ) LOOP  
 CASE c.stock  
 WHEN 0 THEN stock\_facet := 12;   
 WHEN 1 THEN stock\_facet := 13;   
 WHEN 2 THEN stock\_facet := 14;   
 WHEN 3 THEN stock\_facet := 15;   
 WHEN 4 THEN stock\_facet := 16;   
 WHEN 5 THEN stock\_facet := 17;   
 ELSE stock\_facet := 18;  
 END CASE;  
  
 facets.EXTEND(1);  
 facets(facets.LAST) := stock\_facet;  
  
 rowids.EXTEND(1);  
 rowids(rowids.LAST) := c.rowid;  
  
 ctx\_ddl.insert\_mvdata\_values (  
 idx\_name => 'productsindex',   
 section\_name => 'stockfacets',   
 mvdata\_values => facets,   
 mvdata\_rowids => rowids );  
  
 rowids.DELETE;  
 facets.DELETE;  
  
 END LOOP;   
END set\_stock\_levels;

There are a few things to note here:

1. The "table collections" rowids and facets must be initialized. In this case we initialized them empty in the first two lines, then extended them later to hold values, and deleted those values after use. In fact in this example, each table only holds a single value, so we could have simplified the code by initializing them with size 1, not bothering with EXTEND or DELETE and referring to facets(1) rather than facets(facets.LAST). However, the code as we've written it here is more general, so more easily reused if we want to update multiple values or multiple rows. To see the simpler form of code with single value tables, see the trigger code which follows .
2. The "stock\_facet" values 12 through 18 which represent the different stock levels are hard-coded in two places - in this code and in the SQL which loads the facet\_table. This is a potential source of error if one is changed but not the other. Ideally we should look the facet values up from a table, or use the same fixed constants in both places.
3. Rather than doing this after loading, we might have chosen to create an AFTER INSERT trigger on the table. Indeed, we would probably need an AFTER INSERT trigger to handle new records that are inserted later, but we're going to ignore that for now as the code will be very similar to the ON UPDATE trigger that follows.

Running this procedure with

EXECUTE set\_stock\_levels;

Will populate all the facet values.

Next we're going to create a trigger which will update the stock facet whenever the stock column of the products table changes:

CREATE OR REPLACE TRIGGER stock\_facet\_trigger  
AFTER UPDATE ON products  
FOR EACH ROW  
DECLARE   
 rowids SYS.odciRidList;  
 facets SYS.odciNumberList;  
 stock\_facet INTEGER;  
BEGIN  
 -- initialize collections  
 rowids := SYS.odciRidList(1);  
 facets := SYS.odciNumberList(1);  
  
 CASE :OLD.stock  
 WHEN 0 THEN stock\_facet := 12;   
 WHEN 1 THEN stock\_facet := 13;   
 WHEN 2 THEN stock\_facet := 14;   
 WHEN 3 THEN stock\_facet := 15;   
 WHEN 4 THEN stock\_facet := 16;   
 WHEN 5 THEN stock\_facet := 17;   
 ELSE stock\_facet := 18;  
 END CASE;  
  
 rowids(1) := :OLD.rowid;  
 facets(1) := stock\_facet;  
  
 ctx\_ddl.delete\_mvdata\_values (  
 idx\_name => 'productsindex',   
 section\_name => 'stockfacets',   
 mvdata\_values => facets,   
 mvdata\_rowids => rowids );  
  
 CASE :NEW.stock  
 WHEN 0 THEN stock\_facet := 12;   
 WHEN 1 THEN stock\_facet := 13;   
 WHEN 2 THEN stock\_facet := 14;   
 WHEN 3 THEN stock\_facet := 15;   
 WHEN 4 THEN stock\_facet := 16;   
 WHEN 5 THEN stock\_facet := 17;   
 ELSE stock\_facet := 18;  
 END CASE;  
  
 rowids(1) := :NEW.rowid;  
 facets(1) := stock\_facet;  
  
 ctx\_ddl.insert\_mvdata\_values (  
 idx\_name => 'productsindex',   
 section\_name => 'stockfacets',   
 mvdata\_values => facets,   
 mvdata\_rowids => rowids );  
  
END stock\_facet\_trigger;

Notes:

1. We're deleting the facet value associated with the old stock value, and inserting the facet value for the new stock value. If the MVDATA section being used is *only* used for stock level (as it actually is here) then we could have done without the delete and used update\_mvdata\_set rather than insert\_mvdata\_values. However the code as used here - with the delete - will work if the MVDATA section is also used for other facet types.
2. The same comments about hard-coded facet values applies here. And of course we've used the same block of code (the CASE section) three times now so it's an obvious candidate for putting into a callable procedure.
3. If we made an AFTER INSERT trigger, it would be similar to this but of course would not require the section which deletes the existing facet value.

Now let's test the procedure and the trigger. We previously ran the set\_stock\_levels procedure, so we'll just run a Result Set Interface query and check the results:

variable rsout clob  
variable res clob  
variable rsd clob  
  
begin  
 :rsd := '  
<ctx\_result\_set\_descriptor>  
 <count />  
 <hitlist start\_hit\_num="1" end\_hit\_num="5" order="score desc">  
 <score />  
 <rowid />  
 <sdata name="model"/>  
 <sdata name="price"/>  
 </hitlist>  
 <group mvdata="stockfacets">  
 <group\_values>  
 <value id = "12" />  
 <value id = "13" />  
 <value id = "14" />  
 <value id = "15" />  
 <value id = "16" />  
 <value id = "17" />  
 <value id = "18" />  
 </group\_values>  
 <count/>  
 </group>  
</ctx\_result\_set\_descriptor>  
';  
END;  
/  
  
BEGIN  
 dbms\_lob.createtemporary(:rsout, true);  
 ctx\_query.result\_set( 'productsindex', 'nikon', :rsd, :rsout);  
 DELETE FROM res\_output;  
 INSERT INTO res\_output VALUES ( xmlType(:rsout) );  
END;  
/  
-- Now get the facets and counts:  
  
SELECT ft.facet\_name, ft.facet\_value, rs.facet\_count  
 FROM facettable ft, res\_output r, XMLTABLE   
('/ctx\_result\_set/groups/group'  
PASSING r.res  
 COLUMNS   
 facet\_id NUMBER PATH '@value',  
 facet\_count NUMBER PATH 'count/text()'  
) as rs   
where rs.facet\_id = ft.facet\_id  
and rs.facet\_count > 0  
order by facet\_display\_order;

This produces the output:

FACET\_NAME FACET\_VALUE FACET\_COUNT  
---------------------------------------- -------------------- -----------  
Stock Level > 5 1  
Stock Level 5 1  
Stock Level 2 1  
Stock Level 0 1

We can then update the stock level for "Nikon B40" to 5, and re-run our query (we'll re-use the same Result Set Descriptor):

UPDATE products SET stock = 5 WHERE model = 'Nikon B40';  
  
BEGIN  
 dbms\_lob.createtemporary(:rsout, true);  
 ctx\_query.result\_set( 'productsindex', 'nikon', :rsd, :rsout);  
 DELETE FROM res\_output;  
 INSERT INTO res\_output VALUES ( xmlType(:rsout) );  
END;  
/  
  
SELECT res FROM res\_output; --  
  
-- Now get the facets and counts:  
  
SELECT ft.facet\_name, ft.facet\_value, rs.facet\_count  
 FROM facettable ft, res\_output r, XMLTABLE   
('/ctx\_result\_set/groups/group'  
PASSING r.res  
 COLUMNS   
 facet\_id NUMBER PATH '@value',  
 facet\_count NUMBER PATH 'count/text()'  
) as rs   
where rs.facet\_id = ft.facet\_id  
and rs.facet\_count > 0  
order by facet\_display\_order;

Output is now:

FACET\_NAME FACET\_VALUE FACET\_COUNT  
---------------------------------------- -------------------- -----------  
Stock Level > 5 1  
Stock Level 5 2  
Stock Level 0 1

And we can see that there are now two items with a stock level of 5.

## Handing Dates - "Bucketing" Date Ranges

Our next example will extend the previous one, reusing the same products table, index and facets table. First we'll load some extra facet values:

INSERT INTO facettable values( 19, 'Release Date', 'Today', 5, 1 );  
INSERT INTO facettable values( 20, 'Release Date', 'This week' ,5, 2 );  
INSERT INTO facettable values( 21, 'Release Date', 'Last week' ,5, 2 );  
INSERT INTO facettable values( 22, 'Release Date', 'Older' ,5, 2 );

You'll see that rather than having fixed date values for particular dates, we have a set of ranges, or "buckets" for the dates. And since they're relative, we will need to update the facet values associated with each document on a daily basis (if the ranges were something like "January", "February" etc that wouldn't of course be necessary).

So we'll create a procedure update\_date\_facets, which will update all documents for a particular date range. Since we're using a separate MVDATA section "datefacets" for the dates, we don't need to worry about deleting any previous values, we can use update\_mvdata\_set to override any previous values for the rowid in question:

CREATE OR REPLACE PROCEDURE update\_date\_facets (  
 date\_label VARCHAR2,  
 min\_age INTEGER,  
 max\_age INTEGER  
) IS   
 rowids SYS.odciRidList;  
 facets SYS.odciNumberList;  
 num INTEGER;  
BEGIN  
 -- initialize collections  
 rowids := SYS.odciRidList();  
 facets := SYS.odciNumberList();  
  
 -- Fetch the relevant facet id for the label (facet value, eg "Today")  
 SELECT facet\_id INTO num FROM facettable   
 WHERE facet\_name = 'Release Date' AND facet\_value = date\_label;  
 facets.EXTEND(1);  
 facets(facets.LAST) := num;  
  
 -- Now get all the rowids which match the specified date range  
 FOR c IN ( SELECT rowid FROM products   
 where TRUNC(SYSDATE) - TRUNC(rel\_date) BETWEEN min\_age AND max\_age ) loop  
 rowids.EXTEND(1);  
 rowids(rowids.LAST) := c.ROWID;  
 END LOOP;  
  
 IF rowids.COUNT > 0 THEN  
 -- update the MVDATA values  
 -- need to explicitly GRANT EXECUTE ON CTXSYS.CTXDDL to this user  
 ctx\_ddl.update\_mvdata\_set (  
 idx\_name => 'productsindex',   
 section\_name => 'datefacets',   
 mvdata\_values => facets,   
 mvdata\_rowids => rowids );  
 END IF;  
  
END update\_date\_facets;

We can invoke this procedure for each of our "range buckets" by calling it with the label and the min\_age and max\_age in days for that label:

EXECUTE update\_date\_facets('Today', 0, 0)  
EXECUTE update\_date\_facets('This week', 1, 6)  
EXECUTE update\_date\_facets('Last week', 7, 13)  
EXECUTE update\_date\_facets('Older', 14, 9999)

And we can run a query against that:

begin  
 :rsd := '  
<ctx\_result\_set\_descriptor>  
 <count />  
 <hitlist start\_hit\_num="1" end\_hit\_num="5" order="score desc">  
 <score />  
 <rowid />  
 <sdata name="model"/>  
 <sdata name="price"/>  
 </hitlist>  
 <group mvdata="datefacets" topn="10">  
 <count/>  
 </group>  
</ctx\_result\_set\_descriptor>  
';  
 dbms\_lob.createtemporary(:rsout, true);  
end;  
/  
  
EXECUTE ctx\_query.result\_set( 'productsindex', 'nikon', :rsd, :rsout)  
  
insert into res\_output values (xmltype(:rsout));  
  
SELECT ft.facet\_name, ft.facet\_value, rs.facet\_count  
 FROM facettable ft, res\_output r, XMLTABLE   
('/ctx\_result\_set/groups/group'  
PASSING r.res  
 COLUMNS   
 facet\_id NUMBER PATH '@value',  
 facet\_count NUMBER PATH 'count/text()'  
) as rs   
where rs.facet\_id = ft.facet\_id  
and rs.facet\_count > 0  
order by ft.facet\_group, rs.facet\_count desc;

Which produces output:

FACET\_NAME FACET\_VALUE FACET\_COUNT  
---------------------------------------- -------------------- -----------  
Release Date Last week 2  
Release Date This week 1  
Release Date Today 1

## Hierarchical Facets

Sometimes it helps to display facets in a hierarchical fashion. For example a location facet might be represented as

AMERICAS (5)  
 -> Canada (2)  
 -> Quebec (2)  
 -> USA (3)  
 -> New York (2)  
 -> Chicago (1)

EUROPE (4)   
 -> France (4)  
 -> Paris (2)  
 -> Marseilles (2)

It can be seen that the count at each level is the sum of all the counts below.

It will probably not surprise the reader to learn that Oracle Text is not concerned with the hierarchy of facets. Instead, there are two ways this can be modelled:

1. The facets can be associated only with the "twig" nodes - in this case, cities. Then the application would need to keep track of the relationship between nodes, and also count up the sum counts at each level.
2. Facets can be associated with all nodes on the tree. An item with the branch node of New York would also have facets for USA and AMERICAS associated with it. Oracle Text would still need to keep track of the relationship between nodes for display purposes (probably by having a PARENT column or similar in the facet table), but it would not have to sum up the counts in order to display the counts at a higher level. This is the recommended approach.

## "Top N" facets

In the code examples above, we have had to list all the required group values in Result Set Descriptors:

<group mvdata="facetlist">  
 <group\_values>  
 <value id = "1"/>  
 <value id = "2"/>  
 </group\_values>  
 <count/>  
 </group>

This is a little awkward if we add new facet values. There is an alternative syntax where we can say that we want the "Top N" facets returned - the top facets ordered by their counts. We should be able to say something like:

<group mvdata="stockfacets" topn="10">  
 <count/>  
 </group>

Unfortunately at the time of writing, a bug (internal bug number 1426981**)** prevents this syntax from working correctly. It usually causes a syntax error to be thrown, but also causes other more serious issues including ORA-0600 internal corruption errors.

I hope to be able to revise the document once the bug is fixed.

This document copyright Oracle, 2010.

Code used in this document is provided for example purposes only. It may be freely copied or reused but users must take responsibility themselves for the code, including any bugs or errors. Oracle Support will treat this code or any code derived from it as custom application code. The author is happy to accept feedback or questions but cannot guarantee to reply in a timely manner.

Roger Ford : roger.ford@oracle.com, July 2010

Document version 1.0 for Oracle Database 12c Beta 2 release.

