JDeveloper and the Oracle Database - Performing Together

Accessing the Oracle Database

To many Java developers the Oracle database is simply a data store and can be left to get on with its work without interference. More traditional Oracle developers understand that the database is a sophisticated piece of software that offers a great deal of functionality, including an optimization engine that determines the path used to access your data. The optimizer's task is to determine the possible access paths to the data and work out which access path will use the least resources. When we write our own code we can make sure that it is well designed and does not include anything that will adversely impact on the optimization of the statement. When we are using ADF it is important to understand that the SELECT statements in particular are being written for us, and therefore we need to understand the effect of some of the JDeveloper options that govern the writing of these statements.

This article considers the performance implications of the way we design our ADF applications. Note that it is focused on performance tuning of ADF applications from the perspective of the database, not on tuning the middle tier, which is a whole separate topics of its own.

Determining the Access Path

Oracle has a number of available methods for accessing data but the most commonly used will be one of the following:

- A full table scan reads all the data in a table and then filters out the rows that are needed. This is equivalent to reading a reference book and skipping over the bits that you are not interested in. It is most efficient when we are accessing a large proportion of the information in the table.
- An index scan looks up the information in an ordered index, finds out the location of the data and then accesses it. It is equivalent to using the index of a book to look up a particular topic, find the pages that are relevant and then go to those pages. It is most efficient when we are accessing a small proportion of the information.

One of the first techniques we should learn is how to determine the statement that was used to access our data, and how many resources were used. Experienced Oracle developers are normally familiar with these techniques, but Java developer may not be. A simple technique for seeing what actually happens in the database follows.

Identify the likely statements in the shared pool. This statement will find all SQL that refers to the BOOKINGS table.

```
SELECT  sql_id, sql_fulltext,
        executions, trunc(buffer_gets/nullif(executions,0)) getsper, buffer_gets gets
FROM    v$sqlstats
WHERE   UPPER(sql_text) LIKE '%BOOKINGS%'
AND     sql_text not like '%v$sqlstats%'
```

SQL_ID	SQL_FULLTEXT	EXECUTIONS	GETSPER	GETS
0wznfb80b6zzx	SELECTBookingsLarge0.BOOKING_NO,Booki...	1	107	107

Figure 1 - This shows the result of running the statement in SQL Developer

Clicking on the SQL_FULLTEXT column shows the full statement.

```
SELECT
BookingsLarge0.BOOKING_NO,
BookingsLarge0.EVENT_NO,
BookingsLarge0.RESOURCE_CODE,
BookingsLarge0.CHARGEABLE,
BookingsLarge0.MADE_BY,
BookingsLarge0.QUANTITY,
BookingsLarge0.COST,
BookingsLarge0.STATUS,
BookingsLarge0.COMMENTS
FROM  BOOKINGS_LARGE BookingsLarge0
WHERE booking_no = :pBookingNo
```

Paste the sql_id from the above results into the following statement to find the access path:

```
SELECT object_name, operation, options
FROM   v$sql_plan
WHERE  sql_id = ' 0wznfb80b6zzx'
```

OBJECT_NAME	OPERATION	OPTIONS
(null)	SELECT STATEMENT	(null)
BOOKINGS_LARGE	TABLE ACCESS	BY INDEX ROWID
BOOKLG_PK	INDEX	UNIQUE SCAN

Figure 2 - Results from querying V$SQL_PLAN

From the results above we can see that Oracle used an index to access the single unique Booking_No from the table. Now we know how to find out what Oracle does, we can look at the effect of some of the View Object Options

View Objects

In an ADF application the SELECT statements that will be generated are determined by the View Objects (VO), an ADF Business Component. Some VO properties can impact on performance.

the Where Clause

There are three main ways we can define the WHERE clause in our VOs:

- By hard coding them using literals
- Using bind variables
- Using View Criteria

I think we can discount the first option in most cases as inflexible and unlikely to make the best use of the library cache.

Bind Variables

We can define conditions in our View Object using bind variables. This results in code such as:

Booking_no = :pBookingNo

If we are supplying a value for the variable pBookingNo then this code is efficient and Oracle can decide to use an index to access the data. However in situations where we do not supply a value for the bind variable we will get no data returned.

If we use code such as the following, an index cannot be used:

Booking_no = :pBookingNo or :pBookingNo IS NULL

We can implement a much more flexible and better performing approach using View criteria

View Criteria

A View Criteria is a named object that defines a set of WHERE clause conditions that can be applied to a VO. A VO can have multiple view criteria defined. It is a highly flexible approach since, when we expose the VO we can choose to enable or disable various View Criteria. The settings we apply to the Bind Variables and the View Criteria will affect performance.

Example 1:

Bind Parameter – Required = Y

View Criteria – Optional

View Criteria – Ignore Null Values = Y

If we use this approach the SQL statement will be constructed as follows:

```
SELECT
BookingsLarge0.BOOKING_NO,
BookingsLarge0.EVENT_NO,
```

```
BookingsLarge0.RESOURCE_CODE,
BookingsLarge0.CHARGEABLE,
BookingsLarge0.MADE_BY,
BookingsLarge0.QUANTITY,
BookingsLarge0.COST,
BookingsLarge0.STATUS,
BookingsLarge0.COMMENTS
FROM  BOOKINGS_LARGE BookingsLarge0
WHERE ((((BookingsLarge0.BOOKING_NO = :pBookingNo )
       OR (:pBookingNo IS NULL))))
```

This code cannot use the index on BOOKING_NO.

Example 2:
Bind Parameter – Required = N
View Criteria – Optional
View Criteria – Ignore Null Values = Y
In this case the code construction is dependent on whether a value is supplied for :pBookingNo.
The following code is produced when the user supplies a value for :pBookingNo:

```
SELECT
BookingsLarge0.BOOKING_NO,
BookingsLarge0.EVENT_NO,
BookingsLarge0.RESOURCE_CODE,
BookingsLarge0.CHARGEABLE,
BookingsLarge0.MADE_BY,
BookingsLarge0.QUANTITY,
BookingsLarge0.COST,
BookingsLarge0.STATUS,
BookingsLarge0.COMMENTS
FROM  BOOKINGS_LARGE BookingsLarge0
WHERE (((BookingsLarge0.booking_no = :pBookingNo)))
```

The following code is produced when the user does not supply a value for :pBookingNo:

```
SELECT
BookingsLarge0.BOOKING_NO,
BookingsLarge0.EVENT_NO,
BookingsLarge0.RESOURCE_CODE,
BookingsLarge0.CHARGEABLE,
BookingsLarge0.MADE_BY,
BookingsLarge0.QUANTITY,
BookingsLarge0.COST,
BookingsLarge0.STATUS,
BookingsLarge0.COMMENTS
FROM  BOOKINGS_LARGE BookingsLarge0
```

The WHERE clause predicate is only added when the parameter value is provided, which allow the best access path to be used.

Other Techniques Which Affect Performance

Using Functions in the Select

It is commonly understood that using PL/SQL function calls in a SELECT statement can affect performance, since the function may need to be executed for every single row of the results set. In ADF developers/designers tend to focus on the number of rows being returned to the middle tier. It is important to understand that a statement containing a function, which returns 10 rows at a time to the middle tier is not necessarily executing the function only 10 times. In particular, if we are sorting the

rows prior to returning the first 10 rows, then the function may be executed many more than 10 times. The ADF components provide controls which make it very easy for users to perform sorts. If the data needs to be sorted before returning the first set of rows then all the rows which match the selection criteria need to be read, and the function executed. This could mean executing a function thousands of times before the first set of rows is returned.

Transient Attributes

In some cases, rather than fetching derived values using a database function we can calculate the values using a transient attribute in the Entity Object. If the derived value we require is in a different Entity Object then we can base a View Object on multiple associated Entity Objects to obtain the derived value. This will perform better than the approach in which we use a database function. It is important however to include the source attributes of the derived expression in the View Object in addition to the derived value. This prevents multiple accesses to the linked table and ensures all the data is fetched in a single join.

Fetching Multiple Values from the Same Tables

In general when accessing data we should avoid repeatedly going back to the same database rows to fetch a number of columns. When we visit the database we want to fetch everything we need in one statement if possible. Consider the following type of statement. In this example the developer has defined 4 transient attributes, each of which gets a different value from a set of tax tables:

```
SELECT          taxFileNo,
                getTaxCalc1(taxFileNo),
                getTaxCalc2(taxFileNo),
                getTaxCalc3(taxFileNo),
                 getTaxCalc4(taxFileNo)
FROM            TaxDetailsTable
```

The data in this case is coming from the same rows - those that correspond to a particular tax file number, but four separate functions are firing to get the values. In this case we can take the following approach to minimize our trips to the database:

Create a single function that returns all the values we want separated by a character such as |.

Return that value to a transient attribute.

Create 4 additional transient attributes each of which will be based on a Groovy expression that extracts the part of the concatenated value that is appropriate to that attribute.

Modified Columns

Most developers will be aware that, if you modify a column in the predicate it cannot use an index on the column (although it would be able to use a function based index.)

Example:

```
WHERE UPPER(name) = :pName
```

This cannot use an index on Name.

When you are entering a View Criteria for a View Object , if you set the Ignore Case checkbox on, then this is the code that will be generated. If you know the data is stored in upper case then ensure the checkbox is off, if not you may need to create a function based index.

Validation

JDeveloper offers a number of declarative validation options. Some of these may perform poorly.

Using an IN View Accessor validation rule such as the example shown will perform a full scan of the table on which the View Accessor is based. In this example, when a user entered a value for EventNo into a screen it needed to scan 100,000 blocks to validate the value.

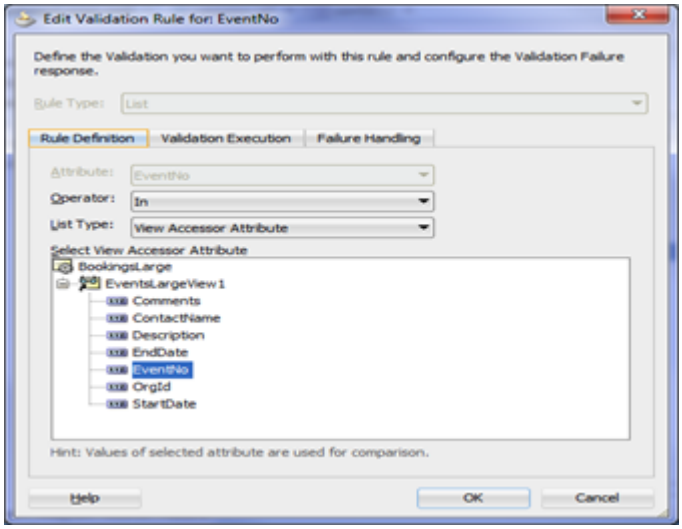

Figure 3 - Validation using IN

Using a Key Exists validation instead will perform an indexed lookup of the underlying table and will perform very much better.

Using View Accessors to Display Additional Information

View Accessors can also be used to display additional information on a screen, such as the name associated with an identifier. As an example we could use a Choice list based on a View Accessor which fetches an Organisation's name rather than displaying just the Organisation Id in one of the organisation's events. This approach may require a full scan of the organisations table. It is a much better approach to base the View Object on a join of the Events and Organisations tables by including both Entity Objects in the Events View Object.

Skewed data

The Oracle 11g database uses a feature known as Adaptive Cursors to identify values for bind variables in which the data is skewed and a particular bind variable value may return either a very small or very large percentage of the table. If I execute a statement with a minority value in the bind variable, then a majority (twice) it reassesses the plan and makes a different decision. You will find that in an ADF application, because the database session maintains a link to the statement in the shared cursor cache from its session cursor cache, the adaptive cursor functionality only works when a user from another session executes the statement. Regardless of the source of the statement, it also means that Oracle will get the access path wrong once before the new path is accessed.

You can get round this feature by identifying statements which use bind variables against skewed data and adding the BIND_AWARE hint to the Query Optimizer Hint property of the View Object.

ADF Query Components

The ADF Faces libraries in 11g offer some very attractive looking query facilities such as the ADF Query Panel with Table. While these provide very flexible query functionality to your users, you need to be aware that they also allow them to enter criteria which may impact on performance.

In this example we are allowing our users to perform a wide range of searches against the Bookings_Large table. If they happen to choose criteria where there is an equality clause and an index on the column the performance will be fine. If they choose non indexed columns, or use operators such as *Contains* then we must expect a full table scan.

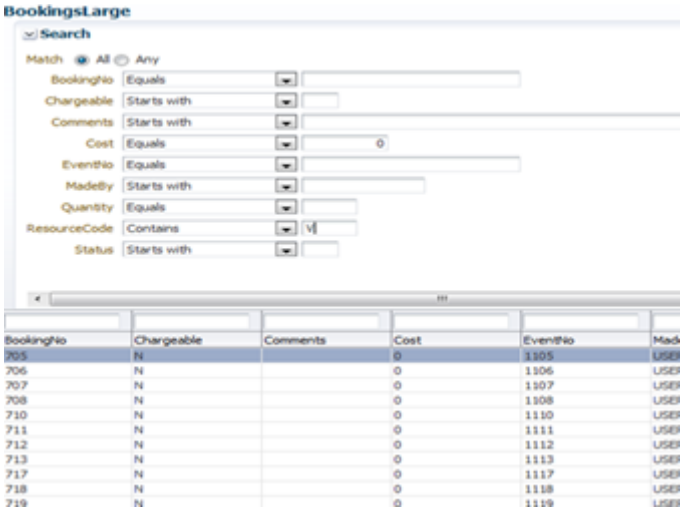

Figure 4 - ADF Query panel with table

Note that the full table scan on its own is not necessarily a problem. If we are fetching the first 20 rows with no sorting then it will perform well. If however the user chooses to sort the data, or if we have ordered the data in the View Object then in order to display the screen all rows which match the selection criteria must be sorted. This could be a large number depending on the selection criteria.

In situations where performance is critical and you have large data sets it may be better to avoid the more flexible query facilities provided by ADF and write your own more controlled query functionality.

Application Module Pooling and Database Connections

Database Sessions

Connections to the database are managed by the Application Module. The mechanism for application module pooling and in particular the way in which multiple application modules will nest depends on the whether you are using a JDeveloper version pre or post 11.1.2.* . However the basic rule that applies across these versions is that we would want to minimize the number of database connections used by a screen.

Use the same data source

If a page uses data from multiple Application Modules it is important that they use the same named data source. Even if the schema to which the data sources connect is the same, using a different named data source will result in multiple connections.

Using New Transaction

If we use an approach where we display a number of Bounded Task Flows (BTF) in regions on a page, and these BTFs all have their transaction management set such that they do not shared data scope, and use a new transaction, then each BTF will require its own database session. In some cases this is a necessary evil in order to implement the business requirements, however it is important to be aware of the implications from a performance point of view. With a large number of users and multiple sessions per screen the number of database sessions could be significant.

Passivation

Passivation is the method by which data is written out from the Application Module Pool to temporary persistent storage in order to free up memory in the middle tier. This will normally be to the PS_TXN table in the database. In a production environment, running with Application Module Pooling enabled means that the number of times passivation occurs is minimized. However in order to test that our

application code functions correctly when passivation occurs we will normally develop and test our application with Application Module pooling turned off.

If you are assessing your application for performance, you will find very different behavior with Application Module pooling turned off. As an example you are likely to find that queries against the database are issued multiple times in order to populate a screen. In some tests a simple query screen resulted in 3 executions of the same underlying statement to populate the screen. Clearly if the query is slightly slow to begin with this is going to amplify the effect. If you provide the application to your users for user acceptance testing they will receive a poor impression of performance if application module pooling is still turned off. The message here therefore is to make sure that, while you should have pooling turned off for development and testing, enable it again for user acceptance testing and production.

Conclusion

While many Java developers will consider the Oracle database to be a simple persistent data store, those of us who have worked with the database for some years know that it is a highly sophisticated product, with a vast array of features including advanced query optimization. It is important that applications that are written using the ADF framework to access the database do not negate these benefits. If developers have an understanding of the way the optimizer works and the effect of various JDeveloper development options on performance then we can expect to build better applications in which the ADF framework and the Oracle database complement each other and perform in an optimum manner.

Author name: Penny Cookson

Company: Sage Computing Services

About the author bio: Penny Cookson has over twenty five years' experience of working with Oracle products. Penny is the Managing Director for SAGE Computing Services, which provides consulting and training in the Oracle database and development tools throughout Australia. She was Oracle Magazine's Educator of the Year in 2004 and is an Oracle ACE.

Author contact details: penny@sagecomputing.com.au