Manipulating CBO in an OLTP Environment

Introduction

The year is 2006. It is time to say hello to Oracle 10g and it is time to say goodbye to the venerable RBO (Rule Based Optimizer) that we all used to resort to so frequently when a query just wouldn't want to use an index. With Oracle 10g, RBO is still here, but is no longer documented or supported. This article is an attempt to give philosophical support to Jonathan Lewis book which provides the gory details of the cost based optimizer.Why was RBO so much loved and why is it so hard to part from it? There are many reasons, but the most important are the following:

- It was easy and simple to understand. The main philosophy of the RBO was: "if there is an index, use it". Rule based optimizer was only looking into availability of certain access paths. Access paths were ranked and the path with the best rank was used. That was it.
- It wasn't changing much. RBO pretty much remained the same from Oracle 5.1.22 until 9.2. This is more then a decade without change. RBO served us well for more then a decade. It's an enviable record indeed.
- It was stable. You could move database from one platform to the next, from one machine to another, from one disk farm to another and plans would never change. If a query with an embedded /*+ RULE */ hint was using index access path On Oracle 7.3.4 on a Windows NT machine, it would use index access path on Oracle 9.2 on a HP SuperDome. Queries did not need to be re-tuned for each new version and each new platform. Developers would develop applications on a small Windows 2k server, run EXPLAIN PLAN and the plan would remain the same on the big box, without particular effort from the programmer or a DBA.
- It was easy to manipulate. We all remember tricks like adding 0 to number or date columns or concatenating with empty strings, in case of character columns, to prevent an index from being used when it wasn't desired. The only place to manipulate RBO was within the SQL statement itself.

As you can see, none of that is true for the CBO (Cost Based Optimizer). First, CBO is not yet a finished product. It is very far from reaching a stable state. I suspect that Oracle will kill it if it ever does approach a stable state. CBO is changing not just between versions, but between patch sets as well. CBO is so complex then an authority like Jonathan Lewis had to write a book, several hundreds pages in length solely to deal with CBO. Second, with machine calibration introduced in Oracle 9i system statistics, collected by DBMS_STATS.GATHER_SYSTEM_STATS) your plans are likely to change even when moving an application from a development box to the QA box and then, again, when moving it to production. In other words, CBO messed up the whole development process. This article is an attempt to answer the obvious question: how to deal with all this change and instability.

Before we start going into the technical details, we have to observe one more crucial difference between RBO and CBO: RBO just wasn't very good with large reports or in a data warehouse environment. RBO was an OLTP beast, pure and simple. And it was very good at it. CBO is a people pleaser: it tries to be everything to everybody. CBO can deal with an index access path as well as a hash join, star schema, query rewrite and materialized views. As the title suggests, this article restricts itself to an OLTP environment as it is precisely the environment which suffers the most from the retirement of the RBO.

Painkillers

Tools to mitigate the separation anxiety are many: hints, outlines, fiddling with "OPTIMIZER_INDEX" parameters, manipulating index statistics directly by using DBMS_STATS.SET_INDEX_STATS and last but not least, the proper application design.

Actually, proper design is the most important method. If your database is designed well, your SQL is going to be natural and simple. The untold secret of the trade is that bad SQL comes from trying to access data which wasn't designed to allow easy access. That results in N-way joins, using at least one non-indexed column and massive DECODE and PL/SQL functions created in order to be able to retrieve the data. There are few universally known rules here:

- Every table must have a primary key.
- Every table must contain timestamp, type of action performed and who did it, as a very minimum. Other identifying data, like the client id, contract id or alike should also be present.
- Columns frequently used in queries must be indexed.
- Same item must be called the same in every table.
- Business rules should be enforced by the database (triggers) and not the application. This is usually hotly debated topic as application developers feel that an application should be the place where business rules are enforced. The problem arises when there is more then one application handling the data. There is no way to make sure that two programmers will not differ in their implementations, which would lead to inconsistencies.

We have all seen applications breaking at least one of these rules. I've also seen applications breaking them all. Especially so called “legacy applications”, precisely the applications designed for the RBO, are prone to break those rules. When this is the case, one needs to consider redesigning the application. Applications have their life cycle and after 6-8 years they have to go through a complete overhaul to remain useful. If your application was designed for RBO, it was likely designed before 2002 and it a major face lift should be in order. Sometimes, that is not an option, mainly for business reasons. So, now we have a problem: we have an old application, designed for RBO which we have to move to Oracle 10g. We cannot redesign, the most we can do is to export/import the application from an old version of the database. The next best option, after redesigning the application is to, somehow, make CBO behave like RBO on steroids. That is achieved by manipulating two parameters, OPTIMIZER_INDEX_CACHING and OPTIMIZER_ INDEX_COST_ADJ. Those two parameters regulate the following things:

- OPTIMIZER_INDEX_CACHING makes CBO assume that for any index certain percentage of blocks is cached. If the value of the parameter is 80, CBO assumes that 80% of the index blocks are cached. According to Jonathan's book and private communication, this parameter has no effect on single table access paths, it only affects in-list iterators and nested loop joins.
- OPTIMIZER_ INDEX_COST_ADJ parameter defines the cost of a single-block I/O as a percentage of a cost of a multi-block I/O request. Single block I/O is most frequently used when retrieving data from an index, while multi-block I/O is normally used when retrieving data from a table. Reference manual claims that this parameter determines a cost of an I/O against index as a percentage of a cost of an I/O against tables. I have to thank Jonathan Lewis again for this clarification.

These two parameters can be manipulated both on the instance level, with ALTER SYSTEM or at the session level, with ALTER SESSION. Those two parameters will make CBO behave like RBO: if an index is there, it will be used. Unfortunately, this approach suffers from the same shortcoming as RBO: if an index is there and we don't want it used, we have to explicitly disable it, by using hints or otherwise. That is something that we can live with, in the OLTP world. Of course, this is the real world so databases are almost never pure OLTP databases. Every OLTP database that I've ever seen runs some reports, processes to supply data to the data warehouse, periodical billing or some other kind of a batch job, which is usually bad suited for an OLTP environment. The owners of those jobs are happy with CBO as it is and don't want it skewed toward index usage. In such cases we have to manipulate those parameters on the session level. Applications written in Java, using EJB and an application server like WebLogic or WebSphere can help us immensely here. Such applications make use of the application server pooling and are all contacting database using the same user id.

This is, of course, a pain to debug and monitor as it is almost impossible to connect a physical user to his database session, but can be helpful if we want to fiddle with the CBO. If we cannot alter OPTIMIZER

INDEX parameters on the instance level, we can do it on the session level. Having only a single user to do it for makes things easier. The right way of doing it, without having to ask the application server administrator is through a database trigger, like this one:

```
CREATE TRIGGER OIPARAM
AFTER LOGON
ON SCOTT.SCHEMA
DECLARE
OIC VARCHAR2(128):= 'ALTER SESSION SET OPTIMIZER_INDEX_CACHING=80';
OICA VARCHAR2(128):= 'ALTER SESSION SET OPTIMIZER_INDEX_COST_ADJ=25';
BEGIN
EXECUTE IMMEDIATE OIC;
EXECUTE IMMEDIATE OICA;
END;
/
```

This is a simple database event trigger which sets the OPTIMIZER_INDEX parameters to values which guarantee "rule based" behavior by using the "EXECUTE IMMEDIATE" command. On a well oiled OLTP systems, one can reasonably expect large part of index blocks to be cached. Together with CPU costing (enabled by gathering system statistics) it usually provides good enough emulation of RBO.

In case with mixed database with very many users, creating a trigger for each one is not practical. In that case, we have to manipulate the statistics directly. The parameter that we can manipulate is called clustering factor and measures the "degree of disorder" of the table with respect to the given index. Clustering factor of an index is calculated by inspecting all keys in index sequentially and adding one whenever block change is encountered. Clustering factor for single column index on column COL is always between the following two values:

```
SELECT COUNT(DISTINCT DBMS_ROWID.ROWID_BLOCK_NUMBER(ROWID))
FROM OUR_TABLE
WHERE COL IS NOT NULL
```

and

```
SELECT COUNT(ROWID) FROM OUR_TABLE WHERE COL IS NOT NULL
```

Clustering factor is a very important attribute, significantly influencing optimizer decision whether to use the index or not. Higher the clustering factor is, less likely is that the index will be chosen by the CBO. By setting the clustering factor low or high, we can suggest the optimizer whether to use the index or not. Clustering factor seems trivial but is actually more important then it seems. For instance, in Oracle 10.2 all tablespaces have automated segment space management by default. That usually results in more blocks in the table and higher clustering factor. Some plans are likely to go awry after upgrade to 10.2. So, how do we set statistics?

Let's create an additional index on the table SCOTT.EMP:

```
CREATE INDEX "SCOTT"."EMP_ENAME_I" ON "SCOTT"."EMP" ("ENAME")
```

When this index is analyzed, we get the following from DBA_INDEXES:

```
  1  SELECT CLUSTERING_FACTOR
  2  FROM DBA_INDEXES
  3* WHERE OWNER='SCOTT' AND INDEX_NAME='EMP_ENAME_I'
SQL> /



CLUSTERING_FACTOR
```

```
-----------------
               1
```

OK, let's now execute a query with AUTOTRACE on and pay attention to the overall cost:

```
  1* select empno,ename,job from scott.emp where ename='CLARK'
SQL> /

     EMPNO ENAME      JOB
---------- ---------- ---------
      7782 CLARK      MANAGER


Execution Plan
----------------------------------------------------------
Plan hash value: 549418132

--------------------------------------------------------------------------------
----

| Id  | Operation                   | Name        | Rows  | Bytes | Cost (%CPU)| Time
|

--------------------------------------------------------------------------------
----

|   0 | SELECT STATEMENT            |             |     1 |    18 |     2   (0)|
00:00:01 |

|   1 |  TABLE ACCESS BY INDEX ROWID| EMP         |     1 |    18 |     2   (0)|
00:00:01 |

|*  2 |   INDEX RANGE SCAN          | EMP_ENAME_I |     1 |       |     1   (0)|
00:00:01 |

---
```

So, index is used and cost is 2. Let's now “fix” the statistics for that index:

```
begin
DBMS_STATS.SET_INDEX_STATS (
ownname  =>'SCOTT',
indname  => 'EMP_ENAME_I',
clstfct  => 100);
end;
```

Now, let's check the result from DBA_INDEXES:

```
  1  SELECT CLUSTERING_FACTOR
  2  FROM DBA_INDEXES
  3* WHERE OWNER='SCOTT' AND INDEX_NAME='EMP_ENAME_I'
SQL> /

CLUSTERING_FACTOR
-----------------
              100

SQL>
```

Now, we will re-execute our simple query, asking for ENAME='CLARK', with AUTOTRACE on:

```
SQL> select empno,ename,job from scott.emp where ename='CLARK';

     EMPNO ENAME      JOB
---------- ---------- ---------
      7782 CLARK      MANAGER


Execution Plan
----------------------------------------------------------
Plan hash value: 3956160932

--------------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |     1 |    18 |     2   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| EMP  |     1 |    18 |     2   (0)| 00:00:01 |
```

Suddenly, the index is no longer used as the new clustering factor makes it much more expensive. Let's force the issue and make the query use the index by using hint:

```
  1  select /*+ index(e emp_ename_i) */
  2         empno,ename,job
  3  from scott.emp e
  4* where ename='CLARK'
SQL> /

     EMPNO ENAME      JOB
---------- ---------- ---------
      7782 CLARK      MANAGER


Execution Plan
----------------------------------------------------------
Plan hash value: 549418132

--------------------------------------------------------------------------------------------
----

| Id  | Operation                   | Name        | Rows  | Bytes | Cost (%CPU)| Time
|

--------------------------------------------------------------------------------------------
----

|   0 | SELECT STATEMENT            |             |     1 |    18 |     5   (0)|
00:00:01 |

|   1 |  TABLE ACCESS BY INDEX ROWID| EMP         |     1 |    18 |     5   (0)|
00:00:01 |

|*  2 |   INDEX RANGE SCAN          | EMP_ENAME_I |     1 |       |     1   (0)|
00:00:01 |
```

The cost is now 5. We didn't change the table, we didn't change the index, the only thing that was hacked was the index statistics. It works both ways. We can make index “good” or “bad”, depending on our intentions.

What are the problems with this approach? First, the next time that the index is analyzed, the clustering factor will be reset:

```
SQL> analyze index scott.emp_ename_i compute statistics;

Index analyzed.

SQL> select clustering_factor from dba_indexes
  2  where owner='SCOTT' and index_name='EMP_ENAME_I';

CLUSTERING_FACTOR
-----------------
                1

SQL>
```

Every database that worked with has some kind of statistics computation job, executed in regular intervals. In other words, our clever hack will be gone after certain period of time, changing the execution plans of the SQL statements without warning. That can be contravened by using DBMS_STATS.LOCK_TABLE_STATS procedure, but that defeats the purpose of having data regularly analyzed. It is very likely that the statistics will not have to be manipulated just for a single table and a single index, we will have to manipulate it for the set of tables, comprising a significant part of our tables, therefore making periodical statistics computation an exercise in futility.

So, how necessary periodic statistics computation really is? To make the long story short, it isn't. Every statistics re-computation will change execution plans, sometimes quite unpredictably. What is the purpose of having an optimizer statistics? The purpose of statistics c is to have a good input values for the CBO to be able to come up with intelligent plans. As long as the statistics reflects realistic relationships among the object sizes, it doesn't need to be changed. The so called “Ensor's paradox” (named after Dave Ensor, famous Oracle DBA and an Oak Table member) states that the only time when it is safe to collect statistics is when the statistics is not needed. Essentially, following the usual business logic, after each statistics computation, the system should undergo an extensive QA test, to make sure that the overall performance is still acceptable. Collecting statistics is something that should be done rarely, only when the data volume changes significantly.

Such a “lazy” model should also be adopted to allow for a normal development process. The DBMS_STATS package can also be used to transfer statistics between systems, making sure that the development, staging, QA and production systems all have the same statistics, making those EXPLAIN PLAN commands done by the developers relevant in production.

Having stable and dependable statistics should be a paramount in a business environment and yet I find it very hard to convince production DBA people not to collect statistics on a daily, weekly or monthly basis. Collecting up-to date statistics is obviously some kind of religious ritual which has exactly the effect it is supposed to alleviate: it introduces ghosts, spirits and daemons in your databases. How many times have I seen questions like “It worked fine yesterday, but it is extremely slow today. Nothing was changed. Please, help me”? More often then not, the root of the evil is in statistics. To add insult to the injury, Oracle 10g has automated job that collects statistics periodically, so even if the DBA didn't create the job it still exists and runs, unless explicitly disabled:

```
SQL> connect sys@local as sysdba
Enter password:
Connected.
SQL> call dbms_scheduler.disable('GATHER_STATS_JOB');

Call completed.
```

Finally, what are the shortcomings of fiddling with statistics? Well, for one, it's global in scope. By hacking statistics and converting a “bad index” into a “good index”, you can inadvertently affect any application using the underlying table, not just an OLTP ones. If a table is used mainly for reporting purposes and the indexes it

has have to be referenced by hints to be used, then hacking the clustering factor will mess up the access path for the reports accessing the table.

Conclusion

There are several methods for achieving acceptable performance and stability in an OLTP environment, with varying degree of sophistication. Each method has its scope, environment for which it is appropriate, its advantages and its shortcomings. The last method, fiddling with statistics requires quite a bit of knowledge and understanding and is usually not recommended. DBA will usually have a hard time convincing management that statistics should not be collected on a regular basis or that "regular basis" should mean "once or twice a year". Therefore, I find recommending the parameter based "silver bullet" approach most effective and the easiest to implement. None of these methods is a solution. CBO is still very much in development and occasional excitements are more or less guaranteed, from version to version and even from patchset to patchset.. Combination of logic and common sense can achieve an acceptable performance for an OLTP system, even with all the changes that optimizer is experiencing. It is also possible to organize stable and dependable development process around CBO although it takes a little bit more planing then with the RBO.