Migrating ? Don't forget the Optimizer.

What is the most important component of the Oracle database engine? My vote goes to the optimizer. Everything the database does is SQL, and every piece of SQL has to be translated into something that can work efficiently. Whatever you do with your data, the optmizer gets involved.

So when you move to a new version of the database software make sure you know how the optimizer can help you, and how you can help the optimizer.

A Historical Perspective

When the Cost Based Optimizer first appeared, it suffered a bad press that didn't stop for a very long time – in fact, every release through to about 8.1 had people saying things like: *"the CBO is finally usable in this release"*. This used to puzzle me a bit, because I had managed to set up systems somewhere around version 7.2 that ran perfectly well on CBO.

I suspect the reason for my early success was that I happened to notice something that has since been explained mathematically: the CBO *"likes"* to do tablescans when the value for parameter ***db_file_multiblock_read_count*** is big, and *"prefers"* to use indexes when the value is small. And, as we now know, a few other parameters display similar characteristics, i.e. RBO liked them big but CBO needed them small.

CBO was completely different from RBO – but if you didn't discover the critical differences and kept on treating it like the RBO you suffered the pain for a long time.

History Repeats Itself

The CBO is a subtle and ever-evolving piece of code. Some releases of Oracle introduce minor enhancements, some releases introduce major enhancements, and some releases change the paradigm.

Unless you spot the major enhancements and paradigm shifts you can end up fighting the CBO for years, trying to work around remembered weaknesses instead of playing to new strengths – just as many people did in the shift from RBO to CBO.

So are there any changes in the CBO that could have as much impact on our thinking as the change from the RBO? The answer is yes.

In 9i Oracle added ***system statistics*** (which I rate as a new paradigm) and automatic ***workarea*** sizing (which I rate as a major enhancement).

In 10g we get various tuning ***advisors*** (which optimize your developers' time rather than the database itself) and ***SQL profiles*** (a major enhancement if you know what problem the feature is solving).

Across both versions we get ***dynamic_sampling*** – which was introduced as a very useful feature in 9i and is of particular benefit for Data Warehouse and Decision Support systems. But if you haven't paid any attention to it in 9i it could become a nuisance in 10g for anyone running an OLTP system.

The most important of all these changes is the introduction of ***system statistics*** – in fact I would go so far as to say that one of the key steps in migrating from Oracle 8 to Oracle 9 is to enable system statistics and work through the impact they have on your system. System statistics are really quite critical to getting the most out of the optimizer in 10g, and the best time to get familiar with system statistics is the moment you decide to migrate from 8i.

This feature is so significant that the rest of this article will focus on nothing else, and save ***dynamic_sampling*** and ***SQL profiles*** for another article. But remember as you read that every version and upgrade of the optimizer gives you new benefits, and failure to investigate and exploit them isn't just going to lead to *'no improvement'*, it may lead to *'negative improvement'*. Never forget how a little ignorance caused so much grief to users in the early years of the Cost Based Optimizer.

System Statistics

Prior to Oracle 9, the Cost Based Optimizer based its calculations on the number of I/O requests that would be needed to satisfy a query, using various constants to massage figures for tablescans, and throwing in a few rules to cater for things like caching of small indexes and so on. (See http://www.jlcomp.demon.co.uk/12_using_index_i.html for an introduction to this topic)

In the early days there were problems with the assumptions made and the limited set of features enabled; but as time passed assumptions were refined, algorithms improved, and new features implemented. But the side effect of estimating I/O requests was a persistent limitation.

In 9i, Oracle introduced ***CPU costing***, a mechanism that allowed the CPU cost of an operation to be included as part of the overall estimate. Initially, this feature was enabled only if you had collected ***system statistics***, but this is no longer true with the arrival of 10g.

So what does ***CPU costing*** do and what are ***system statistics***? Let's start with the system statistics, and a couple of calls to the ***dbms_stats*** package to demonstrate. (This example uses 9.2.0.4, and your account will need to be granted the role ***generate_system_stats*** for it to work.)

```
execute dbms_stats.gather_system_stats('Start');
-- some time delay while the database is under a typical workload
execute dbms_stats.gather_system_stats('Stop');
```

To see what you have done, you can query a table (owned by the ***SYS*** schema) called ***aux_stats$***. After gathering system statistics this will contain a few critical numbers that will be used by the new optimizer algorithms to calculate costs. (You may have to flush the ***shared_pool*** to invalidate existing execution plans, though, depending on your version of Oracle). The following query will show you the current settings:

```
select  pname, pval1
from    sys.aux_stats$
where   sname = 'SYSSTATS_MAIN';
```

The exact list of results is version dependent (the code is still evolving, some versions of Oracle gather more statistics than others) but you will probably see something like this:

```
PNAME                                PVAL1
------------------------------ ----------
CPUSPEED                              564
MAXTHR                           13899776
MBRC                                    6
MREADTIM                           10.496
SLAVETHR                           182272
SREADTIM                            1.468
```

Oracle 10g also introduces a few extra rows with values that are set as the database starts up:

```
CPUSPEEDNW                       904.86697
IOSEEKTIM                               10
IOTFRSPEED                            4096
```

Tablescans

There are two significant changes that apply to the optimizer cost calculations when system statistics are available. You will note first that ***sys.aux_stats$*** holds values for the following:

- ***sreadtim*** Average time for a single-block read request in milliseconds
- ***mreadtim*** Average time for a multi-block read request in milliseconds
- ***MBRC*** Average number of blocks in a multi-block read.

Using this information, Oracle can estimate how long it will take to do a ***tablescan*** (or ***index fast full scan***) The arithmetic is easy: it's just the number of multi-block reads needed to do the scan, multiplied by the average time to do a multi-block read. Ignoring the minor changes due to automatic segment space management, we just take the high-water mark, and work from there:

Time to completion = mreadtim * HWM / MBRC.

Rather then reporting this *"time to completion"* as the cost of the query, Oracle restates the time in terms of the equivalent number of single block reads. To do this, simply divide the time to completion by the average time for a single-block read.

Cost = time to completion / sreadtim

Or, putting the two formulae together and rearranging terms:

Cost of tablescan = (HWM / MBRC) * (mreadtim / sreadtim)

From this example, you can see that the cost of a query **is** the time to completion of a query, but expressed in units of single block reads rather than in proper time units.

When you start using ***system statistics***, the optimizer automatically starts to be more *"sensible"* when choosing between tablescans and indexed access paths because the cost of the multiblock reads used for tablescans will include a proper and appropriate time component.

Historically, the cost of a tablescan was simply:

Cost of tablescan = HWM / (modified db_file_multiblock_read_count).

This made little allowance for the fact that your choice of value for the parameter ***db_file_multiblock_read_count*** could be totally unrealistic, nor did it allow in any way for the extra time that an extremely large ***db_file_multiblock_read_count*** would take compared to a single block read.

This weakness is largely why Oracle created the ***optimizer_index_cost_adj*** parameter in 8.1.6 to allow you to introduce a factor that was similar in intent to the ***mreadtim*** that you collect in system statistics (you may have spotted the similarity between the ***mreadtim / sreadtim*** element in the new cost formula and the common method for estimating a sensible ***optimizer_index_cost_adj***). But there are some unexpected side effects to using the ***optimizer_index_cost_adj*** parameter that can cause problems (see http://www.jlcomp.demon.co.uk/18_oica_i.html for further details) and the mechanisms that come into play when you start using ***system statistics*** are much more robust.

It is still meaningful, by the way, to use the ***optimizer_index_cost_adj*** as a clue to table caching effects (specifically, what percentage of single block table reads are likely to turn into real read requests) even when using ***system statistics***. There are some indications in 10g, though, that even this will become unnecessary in the not too distant future.

CPU Costs

System statistics do more than correct for the I/O and time trade-off between single-block and multi-block reads. They also cater for two further enhancements (or corrections) to costing: first, Oracle can be even better at balancing tablescans against indexed access paths; secondly, Oracle can be smart about rearranging predicate order.

Note how the statistics include the apparent ***cpuspeed***, nominally in MHz. Don't be alarmed if this is nothing like the actual CPU speed of your system – the figure is probably just an internal calibration of a baseline operation that Oracle then uses to produce relative CPU costs of other operations. On one machine running at 2.8GHz, I typically come up with an apparent CPU of a few hundred MHz. Bear in mind that what you see is the speed of a single CPU, not the sum of all the CPUs in a multi-CPU system. [Update: since this article was written, the 10053 trace file has been modified to show that this speed is, indeed, in *"millions of Oracle operations per second"*]

Why does it help the optimizer to know the (apparent) speed of your CPU?

Consider an example where you have a choice:

- Option 1: use an index on a simple date column to find 20 scattered rows in a table.
- Option 2: use a tablescan to examine every row in the table, checking every single row in the table to see if the date column falls in the correct range.

Oracle may decide, based purely on the number and speed of single-block and multi-block reads, that a tablescan would be quicker. But how much CPU will this take if the tablescan requires a test like the following on 10,000 rows:

```
    date_col between to_date('01-Jan-2004') and to_date('02-Jan-2004);
```

CPU operations take time as well and if the number and nature of the tests that have to be performed on a tablescan requires a lot of CPU Oracle factors this into the equation and might switch a query from a CPU-intensive tablescan to an index range scan. You can see this from the formula in the 9.2 Performance Tuning Guide and Reference (A96533 p9-22):

```
    Cost = (
            #SRds * sreadtim +
            #MRds * mreadtim +
            #CPUCycles / cpuspeed
        ) / sreadtim
```

The ***#CPUCycles*** value in this equation is visible in the ***cpu_cost*** column of the newer versions of the ***plan_table*** used by the Explain Plan facility. (Another small, but important, detail of optimizer enhancements – always check to see how the ***explain plan*** facility has evolved).

In fact, if you use this column in the cost equation, the formula needs a fudge factor thrown in – ***cpuspeed*** is recorded in MHz *[update: actually millions of operation per second]*, and the other timings are given in milliseconds, so the CPU component of the formula looks as if it should be adjusted by a factor of 1,000, viz:

```
            #CPUCycles / (cpuspeed * 1000)
```

Predicate Order

But apart from the high-level choices, having some knowledge of CPU speed and the complexity of predicates can allow Oracle to do things that you might never consider in a manual tuning exercise. The best demonstration of this comes from a (slightly contrived) worked example. For the purposes of a repeatable test, the following code ran under 9.2.0.4 and used a locally managed tablespace with a uniform extent size of 1MB, and manual segment space allocation.

```
create table t1 as
select
        trunc(sysdate-1) + rownum/1440    d1,
        rownum                            n1,
        rpad('x',100)                     padding
from
        all_objects
where
        rownum <= 3000
;

alter table t1
    add constraint t1_pk primary key (d1,n1)
    using index (create index t1_pk on t1(d1,n1))
;

execute dbms_stats.gather_table_stats(user,'t1',cascade=>true)
```

Based on this data set, here are two queries which look virtually identical. Which one of them will be faster (the index hint is there just in case your test case has some unusual parameter settings that push the optimizer into doing a tablescan):

```
select  /*+ index(t1) */
        padding
from    t1
where   n1 = 2800
and     d1 >= trunc(sysdate)
;

select  /*+ index(t1) */
        padding
into    m_junk
from    t1
where   d1 >= trunc(sysdate)
and     n1 = 2800
;
```

Notice that the only difference between the two queries is the order of the predicates. If you run the queries through autotrace (***set autotrace on***), you'll find they both produce the same plan, and the same number of consistent gets to execute.

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=8 Card=1 Bytes=112)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=8 Card=1 Bytes=112)
   2    1     INDEX (RANGE SCAN) OF 'T1_PK' (NON-UNIQUE) (Cost=7 Card=1562)

Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          9  consistent gets
          0  physical reads
```

So when I put each query into a pl/sql loop that executes it 10,000 times, why is it that the first query takes 4.34 CPU seconds to run 10,000 times (at 2.8GHz), but the second query takes 13.42 CPU seconds if system statistics are ***not*** enabled; but when I enable system statistics both queries run in the shorter time.

The answer isn't visible in ***autotrace***, but if you run the two queries through Oracle's ***dbms_xplan*** package to get the full execution plan, you see the following for the **faster** query when system statistics are not enabled:

```
-------------------------------------------------------------------------------
| Id  | Operation                   |  Name       | Rows  | Bytes | Cost  |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |             |     1 |   110 |     8 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T1          |     1 |   110 |     8 |
|*  2 |   INDEX RANGE SCAN          | T1_PK       |  1562 |       |     7 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("T1"."D1">=TRUNC(SYSDATE@!) AND "T1"."N1"=2800)
       filter("T1"."N1"=2800 AND "T1"."D1">=TRUNC(SYSDATE@!))

Note: cpu costing is off
```

If you swap the order of the predicates (still without system statistics) to check the **slower** query you will spot a small change in the ***Predicate Information*** section:

```
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("T1"."D1">=TRUNC(SYSDATE@!) AND "T1"."N1"=2800)
       filter("T1"."D1">=TRUNC(SYSDATE@!) AND "T1"."N1"=2800)
```

However, when you enable system statistics (specifically it's the ***CPU costing*** component that counts), then it doesn't matter which way round you write the predicates, the query runs quickly and the execution plan shows that the ***filter()*** line operates the numeric check before the date test.

There are two important points to address here. First, why was there such a significant difference in performance, and secondly, what was Oracle doing when it found the quicker path automatically.

The difference in performance depended on the fact that the data set and query were designed to force Oracle to check both columns (d1 and n1) for every index entry covered by the index range scan, and the specific predicate values required Oracle to check 1,560 rows. Of those 1,560 rows, every single one passed the date test, but only one passed the numeric test. So, by switching the order of the predicates, I was actually choosing between:

1,560 date tests which pass and 1,560 numeric tests of which one passes

or

1,560 numeric tests of which one passes, and one subsequent date test.

The difference in performance is entirely due to the absence of 1,559 date tests.

How did Oracle manage to find that difference in performance when ***cpu_costing*** was enabled? I'm not entirely sure – and it may be a process that is still subject to enhancement over the next few minor releases anyway. What I do know is that I have several examples where Oracle chooses to switch a set of predicates so that numeric tests take place before date tests. In some cases the calculated cost of the query does not change (which makes it look like "numbers before dates" is a simple rule). In a couple of cases the calculated cost of the query changes (which makes it look like the optimizer has done some work on figuring out how many times each test would be executed).

Either way, when ***CPU Costing*** is enabled, a new code path comes into play that can improve the performance of some of your queries without changing the execution path, and you will only understand what's going on if you check the new ***explain plan*** output. (This re-arrangement of predicate order can be stopped by using the hint ***/*+ ordered_predicates */*** but the hint is deprecated in 10g).

Conclusion

Although system statistics appeared in Oracle 9i, they haven't really caught on very well at present. But they are important and are actually critical to certain optimisation options in 10g.

System statistics give the optimizer more of the truth about how your system really performs, and therefore allows the optimizer to produce a better match between estimated and actual query execution time.

System statistics also include details that allow the optimizer to trade I/O costs against CPU costs. At a gross level, knowledge of CPU speeds allow for better decisions on execution paths; but at a finer level of detail Oracle even allows for re-arranging the order of predicate testing to minimise CPU time after the path has been decided.

If you are still running 8i, and plan to migrate to Oracle 9i you should include system statistics as part of your migration plan. If you are running 9i without using system statistics you should probably enable them as soon as possible.

If you are migrating to 10g without first implementing system statistics on 9i then make some allowances for investigating system statistics in your test plans. If you don't you may spend a lot of time trying to understand why odd things are happening in all those places where you've put in a workaround for a problem that wouldn't have existed if you had been doing the right thing in the first place.

Further Reading

Database Performance Tuning Guide and Reference Release 2 (9.2) (A96533-01, p. 3-6 ff.): Gathering System Statistics

Jonathan Lewis is a freelance consultant with more than 18 years experience of Oracle. He specialises in physical database design and the strategic use of the Oracle database engine, is author of 'Practical Oracle 8i - Designing Efficient Databases' published by Addison-Wesley, and is one of the best-known speakers on the UK Oracle circuit. Further details of his published papers, presentations and seminars and tutorials can be found at http://www.jlcomp.demon.co.uk, which also hosts The Co-operative Oracle Users' FAQ for the Oracle-related Usenet newsgroups.