Index Costing Threat

> In the previous post I described a problem with the strategy that the optimizer uses for costing an index fast full scan, and the alternative strategy that Oracle had supplied at some point in the 10g timeline to allow a more rational cost to be calculated in special cases. In an earlier post still I had described the problem with

In ***the previous post*** I described a problem with the strategy that the optimizer uses for costing an ***index fast full scan***, and the alternative strategy that Oracle had supplied at some point in the 10g timeline to allow a more rational cost to be calculated in special cases. In ***an earlier post still*** I had described the problem with the way Oracle derived the ***clustering_factor*** of an index that resulted in the costs of index range scans (and full scans) being given a cost that was too high.

No matter how many things you find out about inconsistencies in the costing algorithms there always seem to be just a few more traps waiting for you on your next assignment – and some of them even turn out to be oddities that you had noticed several years ago but had forgotten about because you "just knew" they couldn't possibly turn into a production problem. Here's one such oddity that I discovered about 5 years ago, and have just rediscovered as a (known) bug – fortunately a bug with at least two workarounds.

Let's start by creating the demo data that I'll be accessing with a simple piece of code that I'm going to hint in two different ways:

```
rem
rem     Script:         oica_iffs.sql
rem     Author:         Jonathan Lewis
rem

create table t1 nologging
as
with generator as (
        select  --+ materialize
                rownum id
        from dual
        connect by
                level <= 1e4
)
select
        lpad(trunc(dbms_random.value(0,10000)),10)      v1,
        rpad('x',20)                                    padding
from
        generator       v1,
        generator       v2
where
        rownum <= 1e6
;

create index t1_i1 on t1(v1) nologging;
```

In 12c, of course, basic statistics for both the table and the index would be created as the code creates the objects, whereas only the index statistics would be created in 11g and I'd have to create both table and index stats manually for 10g – which is what I'll do next before querying the data. You'll note that the index is on a column of type *varchar2()*, populated with a string of exactly

10 characters in every row. With 1 million rows in the table this will give us an index of roughly 1e6 * (11 + 7 + 4) * 100/90 = 25MB … which means a little over 3,000 leaf blocks.

```
begin
        dbms_stats.gather_table_stats(
                ownname          => user,
                tabname          =>'T1',
                cascade          => true,
                method_opt       => 'for all columns size 1'
        );
end;
/

set autotrace traceonly explain

select  /*+ index(t1) */
        count(*)
from    t1
where   v1 is not null
;

select  /*+ index_ffs(t1) */
        count(*)
from    t1
where   v1 is not null
;

alter session set optimizer_index_cost_adj = 1;

select  /*+ index(t1) */
        count(*)
from    t1
where   v1 is not null
;

select  /*+ index_ffs(t1) */
        count(*)
from    t1
where   v1 is not null
;

alter session set optimizer_index_cost_adj = 100;
set autotrace off
```

Note that I've invoked both queries twice. The important test here is what happens to the costs of the two queries when we set the parameter ***optimizer_index_cost_adj*** to a very small value. Now, for a long time (since 9i, basically) the parameter has been viewed as a highly undesirable parameter; to a large extent it was a fix to work around some of the problems of over-costing that should have disappeared when system statistics were introduced. Unfortunately various 3rd party applications still insist that this parameter should be set to a ridiculously small value (and often insist that the ***optimizer_index_caching*** should be set to a ridiculously large value), and a fair number of home-grown applictions still use a non-default value (albeit a more realistic value, perhaps) for the parameter.

Here are the plans when we leave the parameter at its default value:

```
-----------------------------------------------------------
| Id  | Operation         | Name  | Rows  | Bytes | Cost  |
-----------------------------------------------------------

|   0 | SELECT STATEMENT  |       |     1 |    11 |  3070 |
|   1 |  SORT AGGREGATE   |       |     1 |    11 |       |
|*  2 |   INDEX FULL SCAN| T1_I1 |  1000K|    10M|  3070 |
-----------------------------------------------------------

-------------------------------------------------------------------
| Id  | Operation             | Name  | Rows  | Bytes | Cost  |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT      |       |     1 |    11 |   467 |
|   1 |  SORT AGGREGATE       |       |     1 |    11 |       |
|*  2 |   INDEX FAST FULL SCAN| T1_I1 |  1000K|    10M|   467 |
-------------------------------------------------------------------
```

The cost of the index fast full scan is much lower than the cost of the index full scan – and that seems reasonable in this case. Given these figures for the full cost we might expect that setting the ***optimizer_index_cost_adj*** to 1 would reduce the two costs by a factor of 100 – in other words to 31 and 5 respectively. This is what I actually got:

```
-----------------------------------------------------------
| Id  | Operation         | Name  | Rows  | Bytes | Cost  |
-----------------------------------------------------------
|   0 | SELECT STATEMENT  |       |     1 |    11 |    31 |
|   1 |  SORT AGGREGATE   |       |     1 |    11 |       |
|*  2 |   INDEX FULL SCAN| T1_I1 |  1000K|    10M|    31 |
-----------------------------------------------------------

-------------------------------------------------------------------
| Id  | Operation             | Name  | Rows  | Bytes | Cost  |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT      |       |     1 |    11 |   467 |
|   1 |  SORT AGGREGATE       |       |     1 |    11 |       |
|*  2 |   INDEX FAST FULL SCAN| T1_I1 |  1000K|    10M|   467 |
-------------------------------------------------------------------
```

The cost of the index full scan is adjusted as expected, but the cost of the index fast full scan doesn't change – and in this case it means the choice of execution plan would change as we change the ***optimizer_index_cost_adj***. In fact we don't have to go to the extreme value of 1; looking at the costs above we can infer that (to a first approximation) if we set the parameter to floor(467/3070) = 15 then the cost of the full scan would become a little less than the cost of the fast full scan and the default plan would change.

An interesting (and important) point **for my example** is that this probably wouldn't make much difference to performance because the index has just been created; the FULL scan would probably do a large number of extremely efficient *"db file parallel read"* calls because all the required index leaf blocks would be nicely arranged with lots of "logically adjacent" leaf blocks also being physically adjacent. For an index that had grown slowly over time, though, with lots of index leaf blocks split as the index grew, we would probably see a lot of single block *"db file sequential read"* waits because logically adjacent leaf blocks would be physically randomly scattered throughout the index segment.

So What ?

It's relatively easy to find "theoretical" issues with the optimizer – such as this one where the arithmetic looks inappropriate – but the first question you have to ask when you discover such a problem is how much it's likely to matter and whether there's a viable workaround.

In this case I suspect my initial response must have been to ignore the problem because:

- No-one should be messing around with that parameter
- An index fast full scan is a rarely a useful execution path anyway
- If you really need it you can hint it

Of course I had overlooked the fact that various 3rd party applications (like Siebel and SAP) have installation directives that insist on all sorts of silly values for optimizer parameters – including the setting ***optimizer_index_cost_adj*** to 1; even so the chances of Oracle picking a full scan over a fast full scan seemed fairly remote except for the type of scenario where people end up looking closely at long running SQL anyway to see how they can optimize it in detail.

Inevitably, of course, once you start combining enough Oracle features you can always find a way to introduce a problem that has been around for years that no-one has noticed before. And then you discover that the problem was fixed years ago, but no-one really got around to mentioning it. There is a workaround to this optimizer anomaly – and I don't know why it hasn't been made the default behaviour and published with a "Notable change in behaviour" warning.

In Oracle 10.2.0.5 (and possibly earlier versions of 10g, though I can't test that) event 38085 was coded to bypass this issue. Set the event to level 1 and the optimizer applies the ***optimizer_index_cost_adj*** to the cost of index fast full scans:

```
alter session set events '38085 trace name context forever, level 1';

select ...

-------------------------------------------------------------------
| Id  | Operation             | Name  | Rows  | Bytes | Cost  |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT      |       |     1 |    11 |     5 |
|   1 |  SORT AGGREGATE       |       |     1 |    11 |       |
|*  2 |   INDEX FAST FULL SCAN| T1_I1 |  1000K|    10M|     5 |
-------------------------------------------------------------------
```

So we can fix the problem, and we can fix it in a global fashion rather than having to mess around with hints. Will we ever need to fix it ? Possibly – for example this event is listed in a configuration document I've seen for SAP, perhaps as a necessary consequence or their requirement that ***optimzer_index_cost_adj*** be set to 1, so there may be cases where the problem shows up in "typical" processing.

For a specific example of the problem having an impact that you probably wouldn't have predicted and might not recognise there's a quirky little bug report (Bug 14690310) on MoS which complains about the performance of a dynamic sampling query against a partitioned indexed organized table in 11.2.0.3 because the optimizer does an "index full scan" instead of an "index (sample) fast full scan". The analysis of the problem given in the bug report looks incomplete, but there can be little doubt that the failure to apply the ***optimizer_index_cost_adj*** to the calculated cost of the index fast full scan was a contributing factor and I have constructed a test case that demonstrates the point and shows that the 38085 event addresses the problem.

Summary

It's always possible to find anomalies and inconsistencies in the optimizer's calculations, particularly when you try to tweak rarely used parameters. The ***optimizer_index_cost_adj*** has been recognised as a source of undesirable side effects of a long time but can still be found in recent installations. One of the "destabilising" side effects of setting the parameter (especially to very low values) is that the optimizer may start to do single block *"index full scans"* when it would be better to do multi-block *"index fast full scans"*. There are two global options you can use to counteract this defect: either increase the setting for the ***optimizer_index_cost_adj*** (ideally back to the default, of course) or set event 38085 to level 1.