Recent enhancements in Optimization
...
... and some not so recent

Jonathan Lewis
www.jlcomp.demon.co.uk

Who am I?

Independent Consultant.
21+ years in IT
18+ using Oracle
Strategy, Design, Review
Briefings, Seminars
Trouble-shooting
www.jlcomp.demon.co.uk
One of the directors of the UKOUG
Member of the Oak Table Network.
Oracle ACE
OTN Author of the Year 2006

Warning – “Fundamentals” doesn’t mean it’s simple, it means it is the
foundation which you must know in order to understand what’s going on.
Queueing

```sql
select ... 
from 
audience  aud
where 
  i_own_a_copy = 'NO'
or 
  ( 
    i_own_a_copy = 'YES'
    and my_copy_is_signed = 'NO'
  )
;
```

Why would you care ...  

... about recent enhancements ?  

• Some things are worth upgrading for
  – if you’re not already there

• Some features save you future time
  – if you know they exist

• You can get rid of ‘dirty’ code
  – less need for workarounds and tricks
Strategic Targets

- Use of CPU costing (system stats)
- Proper choice of optimizer mode
- Minimalist parameter setup
- Better diagnostics = less time wasted
  - (Elimination of dirty code)
  - (Optimum set of physical structures)
  - (Cost-effective use of statistics)

If you want to run a good system, you always need to be on the lookout for ways to get rid of the mass of garbage that has accumulated over the years.

Environment: Optimizer Mode

- all_rows obvious choice for DW / DSS
- first_rows_N subtle choice for OLTP

\[(n = 1, 10, 100, 1000)\]
\[/*+ first_rows(n) */
\[\text{rownum} \leq n -- 10g \text{ effect}\]

These can be set at the system level then changed at the session or statement level.
first_rows vs. first_rows_N

```sql
create table t1 as
select rownum id,
    mod(rownum,10000) modded,
    lpad(rownum,1000) padding
from {very large source}
where rownum <= 100000
;
alter table t1 add constraint t1_pk primary key(id);
create index t1_mod on t1(modded);
select padding from t1
where modded = 1
order by id;
```

We have a primary key on `id`, and an index that is fairly selective but has a bad clustering factor on the `modded` column. (10 scattered rows per value)

first_rows_N vs. first_rows

**Execution Plan (first_rows)**

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>111</td>
<td>109K</td>
<td>14495</td>
</tr>
<tr>
<td>* 1</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>T1</td>
<td>111</td>
<td>109K</td>
<td>14495</td>
</tr>
<tr>
<td>2</td>
<td>INDEX FULL SCAN</td>
<td></td>
<td>111</td>
<td>109K</td>
<td>100K</td>
</tr>
</tbody>
</table>

**Execution Plan (first_rows_10)**

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>111</td>
<td>109K</td>
<td>148</td>
</tr>
<tr>
<td>1</td>
<td>SORT ORDER BY</td>
<td></td>
<td>111</td>
<td>109K</td>
<td>148</td>
</tr>
<tr>
<td>2</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>T1</td>
<td>111</td>
<td>109K</td>
<td>113</td>
</tr>
<tr>
<td>* 3</td>
<td>INDEX RANGE SCAN</td>
<td>T1_MOD</td>
<td>111</td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

The impact of `first_rows` compared to `first_rows_10`. The older option is extremely aggressive about avoiding sorting and hashing.
Cost is Time (a)

dbms_stats.gather_system_stats('start')
   -- allow some time for typical processing
   dbms_stats.gather_system_stats('stop')

<table>
<thead>
<tr>
<th>SNAME</th>
<th>PNAME</th>
<th>PVAL1</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSSTATS_MAIN</td>
<td>CPUSPEED</td>
<td>559 M-ops/s</td>
</tr>
<tr>
<td>SYSSTATS_MAIN</td>
<td>SREADTIM</td>
<td>1.299 m/s</td>
</tr>
<tr>
<td>SYSSTATS_MAIN</td>
<td>MREADTIM</td>
<td>10.204 m/s</td>
</tr>
<tr>
<td>SYSSTATS_MAIN</td>
<td>MBRC</td>
<td>4 blocks</td>
</tr>
<tr>
<td>SYSSTATS_MAIN</td>
<td>MAXTHR</td>
<td>13938448 -- Max I/O rate</td>
</tr>
<tr>
<td>SYSSTATS_MAIN</td>
<td>SLAVETHR</td>
<td>244736 -- Avg slave I/O rate</td>
</tr>
</tbody>
</table>

External calibration of disk speeds may work better:
Orion (Oracle) or www.iozone.org or simple C

You then need to be aware that system statistics are strategic. You can give Oracle an idea about the resources and response times for your hardware.

Cost is Time (b)

9.2 Performance Tuning Guide and Reference:

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

Before 9i:
   mreadtim = sreadtim

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

The documentation actually tells us that the cost is the time to completion - in units of the single block read times. And that's always been the case.
Cost is Time (c)

<table>
<thead>
<tr>
<th>Plan Table columns</th>
<th>8i</th>
<th>9i</th>
<th>10g</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEARCH_COLUMNS</td>
<td>Not used</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>FILTER_PREDICATES</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>ACCESS_PREDICATES</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>TEMP_SPACE</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>IO_COST</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>CPU_COST</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>QBLOCK_NAME</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TIME</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROJECTION</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OTHER_XML</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEPTH</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OBJECT_ALIAS</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PLAN_ID</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

New versions of Oracle add extra columns to the plan table. Keep an eye on the defining scripts, and on the reporting scripts. Use a single public GTT.

Cost is Time (d)

dbms_stats.gather_system_stats('noworkload') -- 10g

<table>
<thead>
<tr>
<th>SNAME</th>
<th>PNAME</th>
<th>PVAL1</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSSTATS_MAIN</td>
<td>CPUSPEEDNW</td>
<td>913.641725 M-ops per sec</td>
</tr>
<tr>
<td>SYSSTATS_MAIN</td>
<td>IOSEEKTIM</td>
<td>10 millisec</td>
</tr>
<tr>
<td>SYSSTATS_MAIN</td>
<td>IOTFRSPEED</td>
<td>4096 bytes per millisec</td>
</tr>
</tbody>
</table>

Derived \( sreadtim = \)

time to get there + time to read block =

\[ \text{ioseektim} + \left(\frac{\text{db\_block\_size}}{\text{iotfrspeed}}\right) = \]

\[ 10 + \frac{8192}{4096} = \]

12 m/s

In the absence of a real sample, 10g uses some basic disk response times to derive \( sreadtim \) and \( mreadtim \). Then uses these in the same way as 9i.
Cost is Time (e)

Derived \( mreadtim = \)

time to get there + time to do the multiblock read =

\[ \text{ioseektim} + \left( \text{db}\_\text{file}\_\text{mbrc} \times \text{db}\_\text{block}\_\text{size} / \text{iotfrspeed} \right) = \]

10 + 8 (say) * 8192 / 4096 =

26 m/s

10g has three values for the multiblock read counts
the one you used to set - but get rid of it.
the value to attempt at run-time
the value to use for optimization calculations

If you set a value for \( \text{db}\_\text{file}\_\text{multiblock}\_\text{read}\_\text{count} \) then 10g uses that as the MBRC. But you should try to eliminate it from your spfile (init.ora).

Optimiser Parameters 8i

Optimise a query with event 10053 set in 8i, and you get the above set of parameters listed as the 'parameters used by the optimizer'
Optimiser Features 10g

This is the complete list of parameters from 10.2.0.1, which increases again (by another 5 items) as you go to 10.2.0.2 - see xSqlrscopy

Features affected by o_f_e

This is a list of the parameters that change value when you set the optimizer features back to 8.1.7 from 10.2.0.1 - a total of 69.
Don’t disable everything!

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>4</td>
<td>68</td>
<td>3 (0)</td>
</tr>
<tr>
<td>1</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>T1</td>
<td>4</td>
<td>68</td>
<td>3 (0)</td>
</tr>
<tr>
<td>2</td>
<td>BITMAP CONVERSION TO ROWIDS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>BITMAP AND</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>BITMAP CONVERSION FROM ROWIDS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*5</td>
<td>INDEX RANGE SCAN</td>
<td>T1_B1</td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>BITMAP CONVERSION FROM ROWIDS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*7</td>
<td>INDEX RANGE SCAN</td>
<td>T1_B2</td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

9i: _b_tree_bitmap_plans = false

10g: /*+ opt_param('_b_tree_bitmap_plans','false') */

Many people upgrade, then set the optimizer_features_enable back to an earlier version. Don’t. Check the plans, then adjust the matching parameter.

Six Steps of Optimization

• Query Transformation
• Cost Calculation
• Run-time features

And sometimes:
• Anomalous behaviour
• Diagnosis -- see afternoon session
• Fixes -- see afternoon session
Explain Plan (a)

explain plan for ...
select * from table(dbms_xplan.display);

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td></td>
<td>432 (2)</td>
<td>00:00:06</td>
</tr>
<tr>
<td>1</td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>VIEW</td>
<td></td>
<td>167</td>
<td></td>
<td>432 (2)</td>
<td>00:00:06</td>
</tr>
<tr>
<td>* 3</td>
<td>FILTER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>TABLE ACCESS FULL</td>
<td>EMP</td>
<td>20000</td>
<td>156K</td>
<td>62 (2)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>5</td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* 6</td>
<td>TABLE ACCESS FULL</td>
<td>EMP</td>
<td>3333</td>
<td>26664</td>
<td>62 (2)</td>
<td>00:00:01</td>
</tr>
</tbody>
</table>

Predicate Information (identified by operation id):
3 - filter("OUTER"."SAL") > (SELECT /*+ NO_UNNEST */ AVG("INNER"."SAL"
 FROM "EMP" "INNER" WHERE "INNER"."DEPT_NO"=:B1))
6 - filter("INNER"."DEPT_NO"=:B1)

Explain Plan (b)

```sql
select /*+ qb_name(main) */
    count(*)
from (select /*+ no_merge qb_name(results) */
      outer.*
    from emp outer
    where outer.sal > {
      select /*+ no_unnest qb_name(subq) */
        avg(inner.sal)
    from emp inner
    where inner.dept_no = outer.dept_no
    }

);```

The query I used included the 10g hint `qb_name` – giving a query block an explicit name. And every object has an alias (`emp1, emp2` would be better)
Explain Plan (c)

explain plan for ...
select * from table(dbms_xplan.display(null, null, 'ALL'));

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td></td>
<td>432 (2)</td>
<td>00:00:06</td>
</tr>
<tr>
<td>1</td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>VIEW</td>
<td></td>
<td>167</td>
<td></td>
<td>432 (2)</td>
<td>00:00:06</td>
</tr>
<tr>
<td>* 3</td>
<td>FILTER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>TABLE ACCESS FULL</td>
<td>EMP</td>
<td>20000</td>
<td>156K</td>
<td>62 (2)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>5</td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* 6</td>
<td>TABLE ACCESS FULL</td>
<td>EMP</td>
<td>3333</td>
<td>26664</td>
<td>62 (2)</td>
<td>00:00:01</td>
</tr>
</tbody>
</table>

Query Block Name / Object Alias (identified by operation id):
1 - MAIN
2 - RESULTS / from$_subquery$_001@MAIN
3 - RESULTS
4 - RESULTS / OUTER@RESULTS
5 - SUBQ
6 - SUBQ / INNER@SUBQ

With the option 'all', dbms_xplan shows you all query block names, and all object aliases. The projection (column selection) information is also printed.

Explain Plan (d)

select * from table(dbms_xplan.display_cursor); --10g

SQL_ID 8krn5pzw3hm, child number 0
select /*+ qb_name(main) */ count(*) from ( select /*+ no_merge qb_name(results) */ outer.* from emp outer where outer.sal > ( select /*+ no_unnest qb_name(subq) */ avg(inner.sal) from emp inner where inner.dept_no = outer.dept_no ) )

Plan hash value: 322796046

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td></td>
<td>432 (2)</td>
<td>00:00:06</td>
</tr>
<tr>
<td>1</td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>VIEW</td>
<td></td>
<td>167</td>
<td></td>
<td>432 (2)</td>
<td>00:00:06</td>
</tr>
</tbody>
</table>

Predicate Information (identified by operation id):
3 - filter("OUTER"."SAL">)
6 - filter("INNER"."DEPT_NO"=:B1)

In 10g you get some new options in dbms_xplan. One of them lets you query memory for execution plans. The default is "the most recent you executed".
### Explain Plan (e)

```sql
set serveroutput off
select /*+ gather_plan_statistics */ ...
select * from table(dbms_xplan.display_cursor((null, null,'ALLSTATS')));
```

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Starts</th>
<th>E-Rows</th>
<th>A-Rows</th>
<th>A-Time</th>
<th>Buffers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>00:00:00.07</td>
<td>1540</td>
</tr>
<tr>
<td>2</td>
<td>VIEW</td>
<td></td>
<td>1</td>
<td>167</td>
<td>9998</td>
<td>00:00:00.10</td>
<td>1540</td>
</tr>
<tr>
<td>* 3</td>
<td>FILTER</td>
<td></td>
<td>1</td>
<td></td>
<td>9998</td>
<td>00:00:00.07</td>
<td>1540</td>
</tr>
<tr>
<td>4</td>
<td>TABLE ACCESS FULL</td>
<td>EMP</td>
<td>1</td>
<td>20000</td>
<td>20000</td>
<td>00:00:00.04</td>
<td>220</td>
</tr>
<tr>
<td>5</td>
<td>SORT AGGREGATE</td>
<td>EMP</td>
<td>6</td>
<td>1</td>
<td>6</td>
<td>00:00:00.03</td>
<td>1320</td>
</tr>
<tr>
<td>* 6</td>
<td>TABLE ACCESS FULL</td>
<td>EMP</td>
<td>6</td>
<td>3333</td>
<td>20000</td>
<td>00:00:00.04</td>
<td>1320</td>
</tr>
</tbody>
</table>

---

You need the privileges to access:
- v$sql
- v$sql_plan
- v$sql_plan_statistics

---

You can enable `dbms_xplan` to report the rowsource execution statistics. This makes it easier to compare predicted costs with actual resource usage.

---

### Optimizer Enhancements

- **Cost-Based Transformation**
- Better use of available information
  - Outer joins
  - Constraints
- **Better elimination of redundant work**
  - Join elimination
  - Group by elimination
CBQT (a)

```sql
select
  outer.*
from
  emp outer
where
  outer.sal >(
    select
      avg(inner.sal)
    from
      emp inner
    where
      inner.dept_no = outer.dept_no
  );
```

There are many ways to transform SQL statements before calculating the cost. 8i and 9i used to follow heuristics (rules) – 10g works out the cost.

CBQT (b)

**Execution plan (8i autotrace)**

0  SELECT STATEMENT Optimizer=CHOOSE (Cost=33 Card=1000 Bytes=72000)
1  0  FILTER
2  1  TABLE ACCESS (FULL) OF 'EMP' (Cost=33 Card=1000 Bytes=72000)
3  1  SORT (AGGREGATE)
4  3  TABLE ACCESS (FULL) OF 'EMP' (Cost=33 Card=3334 Bytes=26672)

**Execution plan (9i autotrace)**

0  SELECT STATEMENT Optimizer=CHOOSE (Cost=100 Card=1000 Bytes=98000)
1  0  HASH JOIN (Cost=100 Card=1000 Bytes=98000)
2  1  VIEW (Cost=64 Card=6 Bytes=156)
3  2  SORT (GROUP BY) (Cost=64 Card=6 Bytes=48)
4  3  TABLE ACCESS (FULL) OF 'EMP' (Cost=35 Card=20000 Bytes=160000)
5  1  TABLE ACCESS (FULL) OF 'EMP' (Cost=35 Card=20000 Bytes=1440000)

8i would not unnest unless hinted (in most cases). 9i would unnest if it was possible to do so (in most cases). Usually unnesting was a better idea.
CBQT (c)

10g considers both options – and others:

```sql
SELECT
  "OUTER"."DEPT_NO" "DEPT_NO","OUTER"."SAL" "SAL",
  "OUTER"."EMP_NO" "EMP_NO","OUTER"."PADDING" "PADDING"
FROM
  "TEST_USER"."EMP" "INNER",
  "TEST_USER"."EMP" "OUTER"
WHERE
  "INNER"."DEPT_NO"="OUTER"."DEPT_NO"
GROUP BY
  "INNER"."DEPT_NO",
  "OUTER".ROWID,
  "OUTER"."PADDING",
  "OUTER"."EMP_NO",
  "OUTER"."SAL",
  "OUTER"."DEPT_NO"
HAVING
  "OUTER"."SAL">AVG("INNER"."SAL")
```

10g costs for the filter, and the unnest, and even costs for complex view merging of the unnested subquery – which is the SQL shown on this slide.

Outer Join effects (a)

```sql
select
  count(st.padding),
  count(lt.padding)
from
  small_table st,
  large_table lt
where
  st.id (+) = lt.nl
;
```

Historically Oracle always had to use the 'preserved' table as the first of the two tables in the join order.

But 10g introduced the 'right outer' join, which allows the 'preserved' table to be second in the join order.
Outer Join effects (b)

In the 9i plan Oracle has to create the hash table from the (large) preserved table: in 10g Oracle can create the hash table from the deficient table.

Execution Plan (9i autotrace)

0  SELECT STATEMENT Optimizer=ALL_ROWS (Cost=48 Card=1 Bytes=204)
1  SORT (AGGREGATE)
2  HASH JOIN (OUTER) (Cost=48 Card=10000 Bytes=2040000)
3  TABLE ACCESS (FULL) OF 'LARGE_TABLE' (Cost=28 Card=10000 Bytes=1020000)
4  TABLE ACCESS (FULL) OF 'SMALL_TABLE' (Cost=2 Card=29 Bytes=2958)

Execution Plan (10g autotrace)

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>204</td>
<td>52 (2)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>1</td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td>204</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* 2</td>
<td>HASH JOIN 10000</td>
<td>10000</td>
<td>1992K</td>
<td>52 (2)</td>
<td>00:00:01</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>TABLE ACCESS FULL</td>
<td>SMALL_TABLE</td>
<td>29</td>
<td>2958</td>
<td>2 (0)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>4</td>
<td>TABLE ACCESS FULL</td>
<td>LARGE_TABLE</td>
<td>10000</td>
<td>996K</td>
<td>49 (0)</td>
<td>00:00:01</td>
</tr>
</tbody>
</table>

Predicate Information (identified by operation id):  
2 - access("ST"."ID"(+)="LT"."N1")

In the 9i plan Oracle has to create the hash table from the (large) preserved table: in 10g Oracle can create the hash table from the deficient table.

Outer Join effects (c)

```sql
select t1.small_vc, t2.small_vc
from t1, t2
where t1.padding = 'xxxx'
and t2.id(+) = t1.id
and t2.small_vc = 'abc';
```

Neither query is really an outer join

Oracle 10g recognises more cases for eliminating outer joins. In both these cases the last predicate means the outer joins can be eliminated (in 10.2).

```sql
select t1.small_vc, t2.small_vc, t3.small_vc
from t1, t2, t3
where t1.padding = 'xxxx'
and t2.id(+) = t1.id
and t3.small_vc(+) = t2.small_vc
and t3.padding = 'xxxxx';
```
Outer Join effects (d)

Execution Plan (9i autotrace)

0  SELECT STATEMENT Optimizer=ALL_ROWS (Cost=13 Card=1 Bytes=242)
1  0   FILTER     ("T3"."SMALL_VC"="T2"."SMALL_VC")
2  1     HASH JOIN (OUTER)
3  2     MERGE JOIN (CARTESEAN) (Cost=8 Card=1 Bytes=227)
4  3       TABLE ACCESS (FULL) OF 'T1' (Cost=4 Card=1 Bytes=115)
5  3       BUFFER (SORT) (Cost=4 Card=1 Bytes=112)
6  5       TABLE ACCESS (FULL) OF 'T3' (Cost=4 Card=1 Bytes=112)
7  2       TABLE ACCESS (FULL) OF 'T2' (Cost=4 Card=1000 Bytes=15000)

Execution plan (10g autotrace)

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>242</td>
<td>19 (6)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>1</td>
<td>HASH JOIN</td>
<td></td>
<td>1</td>
<td>242</td>
<td>19 (6)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>2</td>
<td>MERGE JOIN CARTESIAN</td>
<td></td>
<td>1</td>
<td>227</td>
<td>12 (0)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>3</td>
<td>TABLE ACCESS FULL</td>
<td>T1</td>
<td>1</td>
<td>115</td>
<td>6 (0)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>4</td>
<td>BUFFER SORT</td>
<td></td>
<td>1</td>
<td>112</td>
<td>6 (0)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>5</td>
<td>TABLE ACCESS FULL</td>
<td>T3</td>
<td>1</td>
<td>112</td>
<td>6 (0)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>6</td>
<td>TABLE ACCESS FULL</td>
<td>T2</td>
<td>1000</td>
<td>15000</td>
<td>6 (0)</td>
<td>00:00:01</td>
</tr>
</tbody>
</table>

The (late) filter predicate that appears in the 9i plan was used as an extra access predicate for the hash join in the 10g plan.

Predicates & Constraints (a)

create table t1 (  
id number not null,  
v1 varchar2(40) not null,  
constraint t1_ck_v1 check (v1 = upper(v1))  
);  
create index t1_i1 on t1(v1);  
select *  
from t1  
where upper(v1) = 'SMITH'  
;

The optimizer can use constraints to create predicates. This has an interesting effect on how many extra access paths become available.
Predicates & Constraints (b)

Constraint: \( v_1 = \text{upper}(v_1) \)
Predicate: \( \text{upper}(v_1) = 'SMITH' \)
Conclusion: \( v_1 = 'SMITH' \)

-----------------------------------------------------------------------
<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>34</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>T1</td>
<td>1</td>
<td>34</td>
<td>2</td>
</tr>
<tr>
<td>* 2</td>
<td>INDEX RANGE SCAN</td>
<td>T1_I1</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
-----------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
2 - access("T1"."V1"='SMITH')
    filter(UPPER("T1"."V1")='SMITH') -- seems redundant

By applying transitive closure, Oracle can produce a new predicate that allows an index to be used where a tablescan was once the only option.

Abusing views

select ... from view_a, view_b where ... and flag_col = 1

create or replace view_a as
select ... from view_x, view_y, view_z ...

create or replace view_x as
select ... from view_1, view_2, view_3 ...

create or replace view_1 as /* !!! */
    select ... from table1 where ... flag_col in (2,3,7)
    union select ... from table2 where ... flag_col in (1,9)
    union select ... from table3 where ... flag_col in (5,11)

Views are not inherently bad - but too many people create views without defining the interface, then use them without knowing what they are for.
Join Elimination (a)

create or replace view v1 as
select
  gp.id id_gp, -- PK of grandparent
  p.id_gp id_gp, -- FK to grandparent
  p.id p_id, -- PK is (id_gp, id)
  gp.small_num_gp,
  p.small_num_p
from
  grandparent gp,
  parent p
where
  p.id_gp = gp.id;

So let’s build a view - then forget what it’s for and how to use it, and see
what Oracle does when we query it.

Join Elimination (b)

select small_num_p
from v1
where id_gp between 10 and 50;

Execution plan (10g autotrace)

| Id  |Operation                    |Name   |Rows | Bytes | Cost (%CPU)| Time     |
|  0 |  SELECT STATEMENT           |       |  43 |  516  |      51 (0)| 00:00:01 |
|  1 |   TABLE ACCESS BY INDEX ROWID|PARENT |   1 |   8   |      2 (0)| 00:00:01 |
|* 2 |    NESTED LOOPS             |       |  43 |  516  |      2 (0)| 00:00:01 |
|* 3 |    INDEX RANGE SCAN         |GP_PK  |  42 |  168  |      2 (0)| 00:00:01 |
|* 4 |    INDEX RANGE SCAN         |P_PK   |   1 |      |      1 (0)| 00:00:01 |

Predicate Information (identified by operation id):

3 - access("GP"."ID">=10 AND "GP"."ID"<=50)
4 - access("P"."ID_GP"="GP"."ID")
   filter("P"."ID_GP"<50 AND "P"."ID_GP">=10) -- transitive closure

Depending on the way you define your view, you may find that Oracle can
eliminate one of the tables from your queries.
Join Elimination (c)

Select small_num_p
From v1
Where id_gp between 10 and 50;

Execution plan (10g autotrace)

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>84</td>
<td>672</td>
<td>87 (0)</td>
<td>00:00:02</td>
</tr>
<tr>
<td>1</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>PARENT</td>
<td>84</td>
<td>672</td>
<td>87 (0)</td>
<td>00:00:02</td>
</tr>
<tr>
<td>* 2</td>
<td>INDEX RANGE SCAN</td>
<td>P_PK</td>
<td>84</td>
<td></td>
<td>2 (0)</td>
<td>00:00:01</td>
</tr>
</tbody>
</table>

Predicate Information (identified by operation id):
---------------------------------------------------
2 - access("P"."ID_GP">=10 AND "P"."ID_GP"<=50)

Because we only want information from table parent – and because the join to grandparent is not going to lose data, Oracle eliminates it.

Eliminating work

select max_id, n1
from (select /*+ no_merge */
      n1, max(id) max_id
      from t1
      group by n1
      ) t
where max_id > 8000

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>13</td>
<td>338</td>
<td>50</td>
</tr>
<tr>
<td>1</td>
<td>VIEW</td>
<td></td>
<td>13</td>
<td>338</td>
<td>50</td>
</tr>
<tr>
<td>* 2</td>
<td>FILTER</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>HASH GROUP BY</td>
<td></td>
<td>13</td>
<td>104</td>
<td>50</td>
</tr>
<tr>
<td>* 4</td>
<td>TABLE ACCESS FULL</td>
<td>T1</td>
<td>2000</td>
<td>36000</td>
<td>28</td>
</tr>
</tbody>
</table>

Predicate Information (identified by operation id):
---------------------------------------------------
2 - filter(MAX("ID")>8000) -- predicate push down
4 - filter("ID">8000)    -- predicate push down

The /*+ no_merge */ hint helps to emulate the behaviour of a stored view in a more complex query. Oracle has inferred a clever 'push' from the max().
Eliminating Sorting

```sql
select count(*) from ( 
    select /*+ no_merge */
    key_value, padding
    from
    t1
    order by
    key_value, padding
) 
```

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>1</td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>VIEW</td>
<td></td>
<td>65536</td>
<td>50</td>
</tr>
<tr>
<td>3</td>
<td>TABLE ACCESS FULL</td>
<td>T1</td>
<td>65536</td>
<td>50</td>
</tr>
</tbody>
</table>

Where has the `SORT ORDER BY` gone to?

The `/*+ no_merge */` forces Oracle to instantiate the view before counting the contents. But the optimizer manages to eliminate the redundant `order by`.

Conclusion

Review your optimizer mode

Use Explain Plan properly

Review your parameters (especially the funny ones)

Work on your constraints

Review your view definitions