Agenda

Who am I
Areas of optimisation
Past, Present and Future
Q and A

Most slides have a foot-note. This is a brief summary of the comments that I should have made whilst displaying the slide, and is there for later reference.
Who am I?

Independent Consultant.
19+ years in IT
16+ using Oracle
Strategy, Design, Review
Briefings, Seminars
Trouble-shooting
www.jlcomp.demon.co.uk

What is "Optimisation"?

- What do we want?
  - Better performance for less cash
- Better application code
  - done by the dba / developer
- Better tools
  - let the dba/developer do a better job, faster
- Better kernel code

When you think of optimisation, it's not just about ways to rewrite the SQL to make it go faster - Oracle keeps inventing new strategies, and new tools.
Case Study - Unnesting

• Subqueries can become inline views
• Subqueries can become anti or semi-joins
• The optimizer may do it for you
• You may have to do it by hand
• Sometimes it is the wrong thing to do
• Sometimes there are better options anyway

Correlated Subquery (1)

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

If you did not include the hint (which is not needed in Oracle 9) a query like this could be slow and expensive in Oracle 8.
Correlated Subquery (2)

Default execution path in 8.1.7 (has to be forced in 9i and 10g)

FILTER
  TABLE ACCESS (FULL) OF EMP (Cost=34 Card=1000)
  SORT (AGGREGATE)
    TABLE ACCESS (FULL) OF EMP (Cost=34 Card=1000)

The costs change dramatically with version - 10g gets it right.

Oracle 8 : 34
Oracle 9 : 34,034
Oracle 10 : 238 (34 + 6 * 34)

The execution plan suggests (incorrectly) that Oracle is doing a full
 tables scan and sort of the subqueried emp table for each row of the driving
table.

Correlated Subq - SQL fix 1

The developer's solution in 8i

select
  outer.*
from
  (select
      dept_no, avg(sal) av_sal
    from
      emp
    group by
      dept_no
  ) inner,
  emp outer
where outer.dept_no = inner.dept_no
and outer.sal > inner.av_sal;

But the introduction of in-line views gave you an option for rewriting it to go
faster (most of the time). See Gaja Vaidyanatha Performance Tuning 101.
Correlated Subq - plan

The execution path for the re-written SQL statement

```
SELECT STATEMENT Optimizer=CHOOSE (Cost=121 Card=1000)
  HASH JOIN (Cost=121 Card=1000)
    VIEW (Cost=86 Card=6)
      SORT (GROUP BY) (Cost=86 Card=6)
        TABLE ACCESS (FULL) OF EMP (Cost=34 Card=20000)
    TABLE ACCESS (FULL) OF EMP (Cost=34 Card=20000)
```

The plan shows that we have explicitly created a 'holding' view of the data, and then done a hash join from this temporary set to the main table.

Correlated Subq - Kernel Fix

Oracle Corp's solution in 9i and 10g
The new plan for the original query

```
SELECT STATEMENT Optimizer=CHOOSE (Cost=73 Card=1000)
  HASH JOIN (Cost=73 Card=1000)
    VIEW OF VW_SQ_1 (Cost=50 Card=6)
      SORT (GROUP BY) (Cost=50 Card=6)
        TABLE ACCESS (FULL) OF EMP (Cost=22 Card=20000)
    TABLE ACCESS (FULL) OF EMP (Cost=22 Card=20000)
```

Not always needed
e.g. already unique

But just as the in-line view solution became popular, Oracle Corp. made it redundant. In Oracle 9 (or with the `unnest` hint) you get this execution path.
Correlated Subq - Problems

Check init.ora / spfile parameters in 9i

_unnest_subquery = TRUE (was false)
enables unnesting of correlated subqueries

_unnest_notexists_sq = SINGLE (9i only)
unnest NOT EXISTS subquery with one or more tables if possible

Oracle 9i unnests MOST subqueries without costing them
Oracle 10g (usually) works out the cost of unnesting

Correlated Subq - SQL option

```sql
select dept_no, emp_no, sal
from (select
depth_no, emp_no, sal,
    avg(sal) over(
        partition by dept_no
    ) av_sal
from emp
) where sal > av_sal;
```

You may do better than unnesting anyway. When you **unnest** you still scan the emp table twice. If you switch to **analytic functions**, you scan it once.
System Statistics (1)

```
dbms_stats.gather_system_stats('start')
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    -- (2.8GHz)</td>
</tr>
<tr>
<td>SYSSTATS_MAIN</td>
<td>SREADTIM</td>
<td>1.299</td>
</tr>
<tr>
<td>SYSSTATS_MAIN</td>
<td>MREADTIM</td>
<td>10.204</td>
</tr>
<tr>
<td>SYSSTATS_MAIN</td>
<td>MBRC</td>
<td>4</td>
</tr>
<tr>
<td>SYSSTATS_MAIN</td>
<td>MAXTHR</td>
<td>13938448 -- PX choke</td>
</tr>
<tr>
<td>SYSSTATS_MAIN</td>
<td>SLAVETHR</td>
<td>3244736 -- PX choke</td>
</tr>
</tbody>
</table>

One of the major features of Oracle 9 that should be included as part of the standard migration is the CPU-based costing feature. It is mandatory in 10g.

System Statistics (2)

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

So “cost” is the elapsed time measured in units of single block reads.

This formula for cost appears in the 9.2 manuals. Effectively it says that cost really is supposed to be a prediction of elapsed time.
System Statistics (3)

Tablescan cost =

\[(\text{High Water Mark / MBRC statistic}) \times \]
\[(\text{mreadtim / sreadtim}) + \]
\[\text{a CPU component} \]

Using multiple block sizes?

Doubling the block size halves the HWM
which doubles the impact of the MBRC.
which halves the calculated cost of a tablescan.

There are a few little traps to costing - whether you use the traditional or new method - Changing the block size for a table will change the cost of

Improved tools - Explain Plan

```
select *
from t1
Where id1 = 14
   -- index col 1
and id3 between 1 and 20
   -- index col 3
and n1 = 'a'
   -- tab, number col
and v1 > 15;
   -- tab, varchar col
```

Apart from the new optimisation tricks, Oracle keeps enhancing the tools to let you see what's going on. This is a badly written query.
Improved tools - Explain Plan

```
select * from t1
where id1 = 14 -- index col 1
and id3 between 1 and 20 -- index col 3
and n1 = 'a' -- tab, number col
and v1 > 15; -- tab, varchar col
```

**Search_columns:**
2

**Access_predicates (index):**

(T1.ID1 = 14 AND T1.ID3 >= 1 AND T1.ID3 <= 20)

Always check for new columns in the *plan_table*, (utlxplan.sql, catplan.sql).
How much of the index goes into the start-key, stop-key calculations.

Improved tools - Explain Plan

```
select * from t1
where id1 = 14 -- index col 1
and id3 between 1 and 20 -- index col 3
and n1 = 'a' -- tab, number col
and v1 > 15; -- tab, varchar col
```

**Filter_predicates (index):**

(T1.ID3 >= 1 AND T1.ID3 <= 20)

**Filter_predicates (table):**

(T1.N1=TO_NUMBER('a') AND TO_NUMBER(T1.V1)>15)

What predicates have to be checked on every single entry from the leaf entry, and table entry. Implicit conversions suddenly become explicit and
Correlated Subq - revisited

FILTER
TABLE ACCESS (FULL) OF EMP (Cost=34 Card=1000)
SORT (AGGREGATE)
   TABLE ACCESS (FULL) OF EMP (Cost=34 Card=1000)

FILTER
"OUTER. SAL> (SELECT AVG(INNER.SAL)
   FROM EMP INNER
   WHERE INNER.DEPT_NO=:B1)"

TABLE ACCESS (FULL) OF EMP
SORT (AGGREGATE)
TABLE ACCESS (FULL) EMP "INNER.DEPT_NO=:B1"

The unhelpful filter line of the old Oracle 8 plan would become much more meaningful using the new columns in Oracle 9's explain plan

Improved tools - V$ Plan

v$sql_plan
v$sql_plan_statistics

(last_)CR_BUFFER GETS
(last_)CU_BUFFER GETS
(last_)DISK READS
(last_)DISK Writes
(last_)ELAPSED TIME
(last_)EXECUTIONS
(last_)OUTPUT ROWS
(last_)STARTS

for each line of the plan!

Not only do you have a better plan_table, you also have real-time excution plans made visible in Oracle 9i - with the actual run-time statistics.
Correlated Subq - yet again

```sql
SELECT outer.*
FROM emp outer
WHERE outer.sal > (           
  SELECT /*+ no_unnest */ AVG(inner.sal) 
  FROM emp inner             
  WHERE inner.dept_no = outer.dept_no 
); 
```

SELECT STATEMENT

<table>
<thead>
<tr>
<th>FILTER</th>
<th>(last_)Starts</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE ACCESS (FULL) OF 'EMP' (TABLE)</td>
<td>1</td>
</tr>
<tr>
<td>SORT (AGGREGATE)</td>
<td>6</td>
</tr>
<tr>
<td>TABLE ACCESS (FULL) OF 'EMP' (TABLE)</td>
<td>6</td>
</tr>
</tbody>
</table>

If we look at the correlated subquery again, we see that the execution plan is not telling us everything - the statistics (starts in particular) are very helpful.

Improved tools - tkprof

```
tkprof {tracefile} {outputfile} waits=yes -- default
```

Elapsed times include waiting on following events:

<table>
<thead>
<tr>
<th>Event waited on</th>
<th>Times</th>
<th>Max Wait</th>
<th>Total Waited</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL*Net message to client</td>
<td>111</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>SQL*Net message from client</td>
<td>110</td>
<td>83.97</td>
<td>103.47</td>
</tr>
<tr>
<td>SQL*Net more data from client</td>
<td>16</td>
<td>0.05</td>
<td>0.13</td>
</tr>
<tr>
<td>SQL*Net more data to client</td>
<td>48</td>
<td>0.02</td>
<td>0.05</td>
</tr>
<tr>
<td>SQL*Net message to dblink</td>
<td>11</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>SQL*Net message from dblink</td>
<td>11</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>log file sync</td>
<td>3</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Oracle 9i also allows you to summarise wait states in a trace file. Generated from a 10046 level 8 trace, or using the improved (legal) sql_trace facility.
At last we get to ...

10g

Oracle 10g is very different from it predecessors because a huge fraction of the change is about optimising your use of time - not the machine's.

Optimiser enhancements (a)

select *
from t1
where n1 = 1
and n2 = 1
and (n3+1 = 1 or n3+1 = 2 or n3+1 = 3);
and n4 = 1 -- 4-column index

Execution plan from 9i

TABLE ACCESS TEST_USER T1 (by index rowid)
  Filter (T1.N3+1 = 1 OR T1.N3+1 = 2 OR T1.N3+1 = 3)
  INDEX NON-UNIQUE TEST_USER I1 (range scan) (Columns 3)
    Access (T1.N1 = 1 AND T1.N2 = 1 AND T1.N4 = 1)
    Filter (T1.N4 = 1)

The optimiser is always being enhanced to improve the efficiency at all sort of levels. In 9.2, column n3 is checked only after you get to the table.
Optimiser enhancements (b)

```sql
select *
from t1
where n1 = 1
and n2 = 1
and (n3+1 = 1 or n3+1 = 2 or n3+1 = 3);
and n4 = 1  -- 4-column index
```

On the upgrade to 10g, a query like this might suddenly show a performance improvement because the check of column $n3$ can now be done in the index.

New optimiser options

```sql
select count(st.padding),count(lt.padding)
from small_table,large_table
where small_table.id(+) = large_table.n1;
```

Of course, there are inevitably some improvements from coding in new optimisation techniques and execution paths - for example outer hash joins.
New language extensions (1)

```
select
t2.event_id,
t1.chk_pt,
t2.score
from
(t2 partition by (event_id)
right join t1
  on (t2.chk_pt = t1.chk_pt)
)
order by
t2.event_id,
t1.chk_pt
;
```

Oracle has invented, and the ANSI committee has approved, the *partitioned outer join*. (Only for use with ANSI outer join syntax, though)

New language extensions (2)

<table>
<thead>
<tr>
<th>EVENT_ID</th>
<th>CHK_PT</th>
<th>SCORE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Buffer</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Buffer</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Buffer</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Enqueue</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>Enqueue</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Enqueue</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Enqueue</td>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>Wait</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Wait</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>Wait</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Wait</td>
<td>4</td>
<td>17</td>
</tr>
</tbody>
</table>

How can you get from the output on the left to the output on the right? A simple outer join will not do - we need an 'outer join per event id'.

Oracle 10g Evolution 27
### Better hints

- `index (t1, i1)` v9.2
- `index (t1 t1(col1, col2))` v10.1
- `no_index_ss (t3)` v10.1
- `no_index_ffs (t3 i1 i2 i3)` v10.1
- `leading (t3)` v9.2
- `leading (t3 t4 t1 t5 t2)` v10.1

The strangest hint in 10g

```sql
/*+ IGNORE_OPTIM_EMBEDDED_HINTS */
```

Some of the enhancements are very low key, and yet extremely effective. For example, *index hints* just got safer, and the *leading hint* is now really useful.

### Index coalesce - 8i and 9i

```sql
alter index i1 coalesce;
```

Under 8i and 9i, the coalesce could only produce a single block from a set of adjacent blocks - so missed many opportunities for improving the index.
Index coalesce - 10g

```
alter index i1 coalesce;
```

Oracle 10 can take a longer stream of index leaf blocks, and distribute the discovered rows more widely. (The branch block restriction still applies)

---

dbms_stats enhancements

```
procedure alter_stats_history_retention
procedure get_stats_history_retention
procedure restore_table_stats
procedure purge_stats
procedure gather_fixed_objects_stats
procedure gather_dictionary_stats
procedure lock_table_stats
procedure unlock_table_stats
```

There are plenty of enhancements to investigate in dbms_stats - including analysing the X$ objects, locking statistics, and recalling historical stats.
Cache hit statistics

```
INSERT INTO sys.cache_stats_1$(
    OBJ#, CACHED_SUM, CACHED_VAR,
    CACHED_NO, CACHED_DATE, LGR_LAST,
    LGR_SUM, LGR_VAR, LGR_NO, LGR_DATE,
    PHR_LAST, PHR_SUM, PHR_VAR
) VALUES (
    :1, :2, 0,
    CASE WHEN (:2>0) THEN 1 ELSE 0 END,
    SYSDATE, :3, 0, 0, 0, SYSDATE, :4, 0, 0
);
```

Some information that gets dumped to disc (by mmon) is an extract from v$segstat. This is one of the two ways in which the data is recorded.

Cache Hit Ratios resuscitated?

desc tab_stats$
desc ind_stats$

```
OBJ#           NOT NULL NUMBER
CAUCEDBLK      NUMBER
CACHEHIT       NUMBER
LOGICALREAD    NUMBER
```

Cache history is not currently used - but there are parameters waiting to be set to enable it.

And one reason the v$segstat data is recorded becomes apparent when you discover the ind_stats$ and tab_stats$ tables, which hold 'local' cache hit.
Parameters now set to TRUE:

_optimizer_correct_sq_selectivity
_optimizer_dim_subq_join_sel
_optimizer_join_sel_sanity_check
_optimizer_squ_bottomup
_partition_view_enabled
_push_join_union_view2
_right_outer_hash_enable
query_rewrite_enabled
skip_unusable_indexes

And one that's false: _optimizer_ignore_hints

How much **DID** you know about join order effects?

_optimizer_join_order_control = 3

There are 46 parameters that change their values if you change the setting for *optimizer_features_enabled* from 8.1.7.4, through 9.2.0.4 to 10.1.0.2

Undercover optimisation (1)

select user from dual;

begin
  m_user := user;
end;

Execution Plan

0    SELECT STATEMENT Optimizer=ALL_ROWS (Cost=2 Card=1)
1  0  **FAST** DUAL (Cost=2 Card=1)

Statistics

0  db block gets
0  consistent gets
1  rows processed

For all those over-zealous developers who like to include the username and time in every single procedure - the cost of hitting *dual* has dropped.
Undercover optimisation (2)

When 10g sees this

```sql
for rl in (select ... from all_objects) loop
  -- do something for each row
end loop; -- check v$sql.fetches
```

it does something like this

```sql
open cl; -- for select ... from all_objects
loop
  fetch cl bulk collect into m_array_var limit 100
  for i in 1..m_array_var.count loop
    -- do something for each row
  end loop;
  until cl%notfound
  close cl;
end loop;
```

Oracle 10g will take an inefficient pl/sql loop like the above 'traditional' form, and operate as if it were written in the 'array processing form'.

Undercover Optimisation (3)

```sql
for i_ct in 1..10000 loop
  execute immediate
  'select spare from parse_table where id = :v1'
  into m_dummy using i_ct;
end loop;
```

Latch:

<table>
<thead>
<tr>
<th></th>
<th>9.2.0.4</th>
<th>10.1.0.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared pool</td>
<td>30,000</td>
<td>20,000</td>
</tr>
<tr>
<td>library cache</td>
<td>60,000</td>
<td>40,000</td>
</tr>
<tr>
<td>library cache pin</td>
<td>60,000</td>
<td>40,000</td>
</tr>
</tbody>
</table>

Statistic:

<table>
<thead>
<tr>
<th></th>
<th>9.2.0.4</th>
<th>10.1.0.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>parse count (total)</td>
<td>10,000</td>
<td>10,000</td>
</tr>
<tr>
<td>session cursor cache hits</td>
<td>10,000</td>
<td>10,000</td>
</tr>
<tr>
<td>open cursors cumulative</td>
<td>10,000</td>
<td>(15)</td>
</tr>
</tbody>
</table>

Using 'execute immediate' to build strings in a pl/sql function is a popular method. The SQL doesn't get into the pl/sql cursor cache - until version 10g.
Undercover Optimisation (4)

for i in 1..10 loop
   begin
       insert into t1 values (i,i,i,i); -- 4 indexes
   end;
end loop;

Redo entries 50 0
Redo size 10,880 0
Redo copy latch (immediate) 50 0
Redo allocation latch (willing to wait) 50 0
Redo allocation latch (immediate) .. ..

Alert log: Running with 1 shared and 18 private strand(s).

Oracle 10g uses 'private redo threads'. For small transactions, Oracle delays publishing the redo. This reduces redo record count, overhead and latching.

Structural optimisation (1)

create table t1 (n1 number, v1 varchar2(10))
partition by range (n1)  
    partition p001 values less than (1),
    ...
    partition p199 values less than (199) 
);

alter table t1 drop partition p001;

update tabpart$ set part# = :1 where obj# = :2 and bo# = :3

<table>
<thead>
<tr>
<th>call</th>
<th>count</th>
<th>cpu</th>
<th>elapsed</th>
<th>disk</th>
<th>query</th>
<th>current</th>
<th>rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse</td>
<td>198</td>
<td>0.01</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Execute</td>
<td>198</td>
<td>0.10</td>
<td>0.06</td>
<td>0</td>
<td>199</td>
<td>202</td>
<td>198</td>
</tr>
<tr>
<td>Fetch</td>
<td>0</td>
<td>0.00</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>total</td>
<td>396</td>
<td>0.11</td>
<td>0.08</td>
<td>0</td>
<td>199</td>
<td>202</td>
<td>198</td>
</tr>
</tbody>
</table>

If you've ever dropped a partition from a partitioned table with a very large number of partitions, you may have noticed it takes some time. Trace it.
Structural optimisation (2)

```sql
select obj#, part#
from tabpart$
where bo# = (table object id);
```

Results 9.2.0.4

<table>
<thead>
<tr>
<th>OBJ#</th>
<th>PART#</th>
</tr>
</thead>
<tbody>
<tr>
<td>48008</td>
<td>1</td>
</tr>
<tr>
<td>48011</td>
<td>2</td>
</tr>
<tr>
<td>48013</td>
<td>3</td>
</tr>
<tr>
<td>48015</td>
<td>4</td>
</tr>
<tr>
<td>48017</td>
<td>5</td>
</tr>
</tbody>
</table>

Results 10.1.0.2

<table>
<thead>
<tr>
<th>OBJ#</th>
<th>PART#</th>
</tr>
</thead>
<tbody>
<tr>
<td>54815</td>
<td>10</td>
</tr>
<tr>
<td>54818</td>
<td>20</td>
</tr>
<tr>
<td>54820</td>
<td>30</td>
</tr>
<tr>
<td>54822</td>
<td>40</td>
</tr>
<tr>
<td>54824</td>
<td>50</td>
</tr>
</tbody>
</table>

```sql
select -- view tabpartv$
    obj#, dataobj#, bo#,
    row_number() over (partition by bo# order by part#), etc ...
from tabpart$
```

10g does it better; and when you check the data dictionary content, and the view definitions for the two versions, you find out why.

Tuning Advisor

```sql
select /*+ example 1 */ padding from t1 where n1 = 15;

select /*+ example 2 */ (list of columns)
from t1, t2
where t2.n2 = 15 and t2.n1 = 15
and t1.n2 = t2.n2 and t1.n1 = t2.n1
;

select /*+ example 3 */ n2
from t1
where n1 not in (select n1 from t2)
;
```

Possibly the most significant benefit of 10g is the tool that allows you to ask Oracle to find better optimisation paths for your SQL - like these examples.
Findings
The execution plan of this statement can be improved by creating one or more indices.

Recommendations
Consider running the Access Advisor to improve the physical schema design or creating the recommended index.

Benefit
99.66%

Apart from suggesting that an index would be appropriate, and giving an indication of the benefit, you can ask to see the resulting execution plan.
Rationale
Creating the recommended indices significantly improves the execution plan of this statement. However, it might be preferable to run "Access Advisor" using a representative SQL workload as opposed to a single statement.

This will allow to get comprehensive index recommendations which takes into account index maintenance overhead and additional space consumption.

Most importantly, each recommendation comes with a Rationale. The bit I particularly like is the reminder to test with a representative workload.
Not necessarily a bad thing.
Findings
A potentially better execution plan was found for this statement.

Recommendations
Consider accepting the recommended SQL profile.

execute :profile_name :=
    dbms_sqltune.accept_sql_profile(
        task_name => 'TASK_3201'
    )

On a different statement the advisor says it can't help me with the code, but it has produced a Profile (a bit like an Outline) that will help if enabled.
Tuning Advisor (2b)

```sql
select {columns} from dba_sql_profiles;
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>SYS_SQLPROF_040624083819387</th>
</tr>
</thead>
<tbody>
<tr>
<td>CATEGORY</td>
<td>DEFAULT</td>
</tr>
<tr>
<td>STATUS</td>
<td>ENABLED</td>
</tr>
</tbody>
</table>
| SQL_TEXT           | select count(t1.v1) ct_v1, count(t2.v1) ct_v2 from ...

```sql
dbms_sqltune.drop_sql_profile('SYS_SQLPROF_040624083819387');
```

If you accept a profile, you can see it in the view `dba_sql_profiles`. You cannot see what it is doing, though. But you can get its identity to drop it.

---

Tuning Advisor (2c)

Profiles can hold enhanced statistics:

```sql
OPT_ESTIMATE("SEL$1", TABLE, "T2"@"SEL$1", SCALE_ROWS=15)
```

Table T2 will return 15 times as many rows as expected

```sql
OPT_ESTIMATE(
    @"SEL$1",
    JOIN,
    ("T2"@"SEL$1", "T1"@"SEL$1"),
    SCALE_ROWS=15
)
```

The join T2 -> T1 will return 15 times as many rows as expected

Profiles often hold enhanced statistics to help the optimiser calculate costs more accurately. Unlike outlines, they do not hold a fixed path definition.
Tuning Advisor (3a)

Findings
The optimizer could not unnest the subquery at line ID 1 of the execution

Recommendations
Consider replacing "NOT IN" with "NOT EXISTS" or ensure that columns used on both sides of the "NOT IN" operator are declared "NOT NULL" by adding either "NOT NULL" constraints or "IS NOT NULL" predicates.
Rationale
A "FILTER" operation can be very expensive because it evaluates the subquery for each row in the parent query. The subquery, when unnested can drastically improve the execution time because the "FILTER" operation is converted into a join.

Be aware that "NOT IN" and "NOT EXISTS" might produce different results for "NULL" values.

Conclusion
• New features can improve application performance
• New kernel code improves the database performance
• New tools improve your performance