EXECUTIVE OVERVIEW
This paper covers three improvements in SQLDeveloper version 4.2 Explain Plan/Autotrace facility:

- the process gathering execution statistics in the background and the ability to cancel execution (with ability to see partial statistics)
- hyperlinked object names with convenient access to object statistics
- hint management allowing performance analyst to tune the statement

It can be argued that together with earlier amendments, such as side-by-side plan comparison and performance hotspot identification they provide a compelling alternative to popular facilities such as DBMS_XPLAN.

INTRODUCTION
SQL Optimization is well known to be difficult problem. Statistics information is incomplete, robust cost metrics is elusive, and the search space is explosive. The optimization goals are often conflicting. The very first idea that every SQL performance analyst discovers: "The optimization is only as good as its cost estimates". Those fundamental limitations are eloquently summarized in essay by Guy Lohman Is Query Optimization a “solved” problem?

In this article we describe 3 improvements which make performance analyst's job easier.

CANCELING LONG RUNNING QUERIES
A query execution can take ridiculously long time. Optimizer group members coined the term "catastrophic plan" for poorly optimized statements where execution difference between wrong plan and good plan differs in orders of magnitude. However, if the statement runs for hours or even days, not everybody would be willing to witness it running till the end. In many cases gathering partial execution statistics is enough for identifying hot spots. Here is how to do it in SQLDeveloper version 4.2.
Let's “tune” the query

```sql
select count(*) from all_objects, all_objects
```

A reader may object that it is quite a dubious query. However, it suffices for our purposes; most important is the fact that it is guaranteed not to finish in at least 5 min. After starting SQLDeveloper autotrace, a user witnesses the following screen:

There the running autotrace task indicates some activity, while the rowsource tree in the Autotrace output panel and V$MYSTATS data are both empty. After 1 minute elapses a user would notice the Autotrace panel refreshed with some data.
As it is evident from rowsource tree, the query against V$SQL_PLAN succeeded, but the adjacent V$SQL_PLAN_STATISTICS info is empty. This is just an artifact of rowsource level statistics gathering in oracle. On the other hand, statement level V$MYSTAT data seems to be genuine and reflects the amount of work which has been done so far. After two minutes, we notice buffer reads increase:

Since it is not clear how long would it take to run this statement (or even if it is going to finish at all without triggering exception such as memory/storage overflow) let's cancel the task.

Canceling running autotrace in SQLDeveloper 4.2 is not just canceling the task (which would result in busted database session). The only way to reliably stop statement execution in oracle is killing the session, and this is what happens when user cancels autotrace task. Therefore, execution of

```
select count(*) from all_objects, all_objects
```

in the main thread returns exception, which is reported like this
After acknowledging this dialog message a user is advised to wait a little, because it would take some time for task finishing process to collect the final rowsource and statement level statistics. At the end, one can see meaningful V$SQL_PLAN_STATISTICS info:

As a sanity check, we witness pretty close buffer reads numbers both on rowsource and statement level. With partial statistics at hand, performance analyst is expected to proceed to identify hotspots. Clearly, (s)he has to be careful when comparing estimated numbers with partial statistics, because partial statistics (as it is implied by its name) is incomplete!
OBJECT HYPERLINKS

This is pretty obvious feature, which by some mysterious reason has been missing for so many releases. In a word, following object name hyperlink would open object viewer, where the most relevant information is statistics:
HINT MANAGEMENT

The next objective after identifying performance bottleneck is fixing the access path. Here we take little terminology detour together with rudimentary tutorial on SQL optimization.

SQL Optimization 101

SQL queries greatly vary in their complexity, but in the most basic form it is conjunctive query, which is informally known as "select-project-join". With the ubiquitous HR schema, the example of such query might be:

```sql
select count(*)
from employees e, departments d, locations l
where e.department_id = d.department_id
and l.location_id = d.location_id
and l.country_id = 'US'
```

Presented with such a query, what kind of program implementing it a naive software developer would write? A typical strategy would be searching locations table for the records with country code 'US' and list all corresponding location numbers. Then, find all the departments records with matching location number. We proceed by extracting list of department numbers from those records and identify all the employees that match (i.e. work for) those departments. Count them and, done!

At this point a reader familiar with database theory basics might be tempted to ask where does the query say that we are counting employees only? However, we'll not pursue this venue and focus on the crux of the issue: the access path that naive developer just have described is reasonable, but not the only possible execution strategy. Equally feasible might be scanning employees table, first, then matching all those records with departments, and finally matching those combined employees-departments records with locations and filter out every composite employees-departments-locations record that doesn't fit the required country_id. “But this is slower!” -- might naïve programmer exclain. Not necessarily so, and the reader is advised to reflect a minute or two for a counterexample.

Matching records from adjacent tables is performed via formal database operation of natural join. The standard symbol for natural join is bow tie \( \bowtie \), so that the crux of the first [naive] execution strategy is formally

\[
(l \bowtie d) \bowtie e
\]

while the second is

\[
(e \bowtie d) \bowtie l
\]

\(^2\)This fancy term stems from database theory community, where each word refers to Relational Algebra operator, so that the compound expression describes sequence of 3 operators.

\(^3\)“Naive” nowadays is synonymous with “NoSQL”
LEADING Hint

All together there is 12 way to arrange table order in the above example:

(l  d)  e         e  (l  d)  
⋈ ⋈ ⋈ ⋈  
(d  l)  e         e  (d  l)  
⋈ ⋈ ⋈ ⋈  
(d  e)  l         l  (d  e)  
⋈ ⋈ ⋈ ⋈  
(e  d)  l         l  (e  d)  
⋈ ⋈ ⋈ ⋈  
(l  e)  d         d  (l  e)  
⋈ ⋈ ⋈ ⋈  
(e  l)  d         d  (e  l)  
⋈ ⋈ ⋈ ⋈  

For technical reasons, oracle optimizer considers only those possibilities listed in the left column\(^4\). With this limitation the parenthesis become redundant, and the sequence of join operations can be specified solely as a sequence of tables (table aliases to be more precise). This is the essence of the LEADING hint which specifies the succession of tables. Oracle RDBMS manual describes the syntax for each hint, but applying them in practice often is quite arcane. Let see if SQL Developer can help.

For illustrative purposes in this section we use Explain Plan. This is the fastest way to check if a suggested hint influences the execution plan. It is not sufficient if one wants to establish that hinted plan is actually faster. With this disclaimer the execution plan in my environment is

\(^4\)Even though so called “bushy” join trees introduced in recent oracle versions allowed the right column orders, they require more sophisticated LEADING hint.
This is as far from naïve execution as it can possibly get, and our first goal is to influence the naïve plan with hints.

One under appreciated but invaluable repository of hints is the OTHER_XML column. Among other things it contains a full list of hints which guarantees to fix the access path regardless of other factors influencing optimizer decisions (such as statistics). First, the most obvious suggestion is introducing ability to copy any hint with minimal effort (such as click of the mouse). Here is this list of hints in our example together with context menu invoked over the ALL_ROWS hint:

In this section we are focused specifying table joining order, which is prerogative of the LEADING hint:
Here we see that only 3 join orders are listed (not 6):

d  l  e

l  d  e

e  l  d

This is because exhausting list of all permutations is too large to fit into the context menu; consequently it has been limited to varying the first table in join order only. We'll return to this issue and explain how a user can reposition any other table into arbitrary position when discussing TABLE ACCESS rowsource node context menu.

The SEL$1 is the name of the query block and it is the only one in our select-project-join query. Hints with single query block can omit query block references, and in this simplistic form they are described in oracle SQL tuning reference manual. Before moving on to more sophisticated queries with multiple query blocks, lets fix the join order to influence our initial naïve query execution strategy. Fortunately the desired join order is on the context menu item list

l  d  e

The explain plan changes to

![Explain Plan Image]

There we have labeled the sequence of rowsource executions with numbers, and it is indeed the desired join evaluation order.
Subqueries

One of the most influential ideas in SQL is query composition. Introducing just one extra subquery is the most straightforward way to amend our example to obtain more than one query block

```
select count(*)
from employees e, departments d, locations l
where e.department_id = d.department_id
and l.location_id = d.location_id
and l.country_id
  in (select country_id from countries
       where country_name like 'United States%')
```

Let see what unhinted Explain Plan for this query is

![Explain Plan Image]

Apparently, it is 4 table join! Once again, hints in OTHER_XML describe why this is happening
Initially there were two query blocks: main query SEL$1, and subquery SEL$2. Then optimizer decided to unnest the subquery and subsequently merge it into the main query query block. The resulting query block has been given internal name SEL$5DA710D3. The join of four tables is constructed in the context of this query block as it is evident by the LEADING hint. Now, just as we did in last section a user can experiment how join order affects query performance of the transformed query with just a single mouse click.

Hints in the Rowsource Tree

There are also hints available in the rowsource tree. TABLE ACEESS node is equipped with two kinds of hints

Here we see a set of LEADING hints. Unlike similar hints in the OTHER_XML section of the Explain Plan, it is the position of the table corresponding the current node (i.e. TABLE ACCESS for DEPARMENTS) that is varied. If performance analyst would like to swap position of some other table, then s(he) can simply invoke context menu for TABLE ACCESS rowsource node for that object.

5“What the transformed query looks like” is perhaps the most natural question to ask. Unfortunately, there is no satisfactory answer to that. Optimizer represent queries internally as trees with nodes which aren't always having direct SQL equivalent, so the translation of this internal representation to SQL is not always possible.
TABLE ACCESS rowsource node is also equipped with index hints. These are the indexes as queried straight from the dictionary, and no effort has been made to investigate the predicates which affect index applicability.

The last hint helper implemented in SQL developer is join method. Invoking context menu either over join method hint in OTHER_XML, or, alternatively, over join in the rowsource tree suggests hints with either of three methods:

Here we see that outer most join operation – NESTED LOOPS – performs join over some intermediate table build with preceding joins and EMPLOYEES table in the second argument. Changing default NESTED LOOPS method into, say, HASHJOIN can be accomplished with single mouse click.

And finally, suppose we already have some hints in the SQL query that we are tuning. What a result of choosing different hint would be? The tool tries to guess whether adding new hint, or replacing one is the most appropriate. For example, if one have chosen a different LEADING hint, the existing LEADING hint for the same query block is replaced. Likewise, if user changes join method, the prior join method is overridden. All other types of hints are appended to the existing commented hints section of the query.