

UNDERSTANDING CORRELATED SUBQUERIES

© Copyright Oriole Corporation, 2001

There must be literally thousands of statements in use today which would run faster (in some cases incredibly so) if a correlated subquery had been used. It's all too easy to write code which is syntactically correct, works perfectly well and runs quickly on test data. When those data volumes grow as they are apt to do in real life, jobs can take longer and longer to complete until sometimes they become totally unacceptable.

We have seen for example a sixteen hour job, which after a five minute rewrite, thereafter regularly completed in ten minutes. Even more dramatically (although at first it doesn't sound like it) a four minute job was reduced to under two seconds.

The problem is of course is that if you've got a job which takes sixteen hours you have to do something about it, whereas most people will live with a job that takes four minutes when it should take two seconds. The danger is of course that if that piece of code is run 15 times per day, it's taking up one hour when it should only be taking thirty seconds. If it's run 10 times an hour then it's taking 16 hours per day instead of 8 minutes.

What do you do if you have these performance problems?

One popular solution is to invest \$000s in more hardware.

Well, invest isn't really the right word for squandering money unnecessarily is it?

The other solution is to seek and destroy the offending code (well, rewrite it anyway).

How do you identify which statements need to be examined?

Oriole's ORISNOOP will identify the statements which use the most resources. It doesn't necessarily mean that the statement which uses the most resources is inefficient but it's the best place to start.

Having identified the candidate statements how do we know if a correlated subquery is appropriate?

The dramatic speed advantage of a correlated subquery primarily comes when the correlated subquery is run in conjunction with a NOT EXISTS used in preference to an un-correlated query with a NOT IN. While it's true that this construct isn't always faster, in the situations where it is faster, as has been said already, it sometimes seems almost impossibly so.

The secret of why a correlated subquery and NOT EXISTS can be so much faster lies in the indexes. A NOT IN will never use an index.

If an un-correlated subquery is used with a NOT IN then, as normal, the subquery is run first and every row returned by the main query will examine every row returned by the subquery to see if the sought value is there. So if the subquery returns 500,000 rows and the main query returns 200,000 rows and you only want the values from the main query which aren't in the subquery, then there will be 100bn comparisons. If you use a correlated subquery with a 'not exists' and the sought value is indexed, then for each of the 200,000 rows returned by the subquery the index will be interrogated and it will be quickly established whether the value is present or not. 200,000 vs. 100,000,000,000. A correlated subquery is one which changes the order in which queries are run. Normally an inner or subquery runs to completion and the result of the subquery is 'plugged into' the outer or main query which is then run.

With a correlated subquery however, the order in which the queries are run is reversed. The outer query is fired first and as it returns each row, the subquery is run using information supplied by the result of the main query.

It can be a very difficult concept to get across without examples so here's one I hope will help.

If you imagine that you are a large telecom company. By the nature of your business, in many cases your suppliers will also be your customers. Imagine that you want to write to those suppliers who aren't also customers, to ask them why you should continue to do business with them if they don't use your services (I know that this would never happen but for the sake of argument :-)

The code which almost writes itself is:

```
select s.code#  
  from supplier s  
 where s.code# not in (select c.code#  
                      from customer c);
```

Remember that the 'not in' will result in the index on customer.code being ignored, so the customer table will be searched, all rows will be returned and then the code in every row on the supplier table will be returned and compared with EVERY value returned by the subquery. This statement could be rewritten very quickly as:

```
select s.code#
```

```
from supplier main
where not exists (select 'anything you like'
                 from customer c
                 where c.code = main.code);
```

NB The table alias of "main" is only used to show how the correlation is achieved but while sometimes it's not even necessary to have an alias at all it clarifies things if aliases are used. In fact it's good programming practice to use aliases in every situation where more than one table is referred to in a statement, since it saves time when *parsing*. It also prevents you from unintentionally "correlating" a subquery, which might happen if in your subquery, you inadvertently refer to a column which doesn't exist in that subquery's table but is present in the outer query's table.

Anyway, back to the explanation of the correlated subquery above. In this case the outer query will run first and the code returned by the first row will be passed to the subquery and the subquery will run using that value. Using the index the value will either be found to be in the customer table index or not. If it isn't then the 'not exists' will be found to be true and the main query row will be output. If the value was found in the index then the 'not exists' will be false ('exists' would be true) and that main query row will be discarded.

As I indicated above it's not as straightforward as saying (to paraphrase George Orwell) "'not in' bad 'not exists' good" but it is true to say that when not in's good it's very very good and when it's bad it's horrid :-)