

SQL performances : when several finely tuned statements sum up to bad programming

When performance issues arise and when, after rounding up the usual suspects, you begin tracing your programs, you usually end up pouring over tkprof output, looking for full scans of large tables and trying to individually tune each SQL statement. This usually yields good results, but it is a far cry from the complete picture.

The typical example to illustrate the point, is the often encountered update or insert-if-not-found case, applied to a table which has a suitable primary key defined.

The PL/SQL beginner will write it the following way :

```
select count(*)
into v_count
from my_table
where id = v_id;
if (v_count = 0)
then
  insert into my_table(id, ...)
  values (v_id, ...);
else
  update my_table
  set ...
  where id = v_id;
end if;
```

Taken individually, each one of the three statements is perfect. However, a seasoned Oracle user will probably wonder why, in the case of the update, we are scanning the primary key index twice, once to count occurrences, and a second time to find the data block address in order to be able to do the update.

Therefore, our experienced user will rather write the following :

```
begin
  -- Get the address of the block to update
  select rowid
  into v_rowid
  from my_table
  where id = v_id
  for update of ..;
  -- Do the update
  update my_table
  set ...
  where rowid = v_rowid;
exception
  when no_data_found then
    insert into my_table(id, ...)
    values (v_id, ...);
end;
```

All this is fine, but how can we actually prove that one method is better than the other? A smart way to measure statement efficiency is to check how many buffers are accessed to process it; much more than CPU time or any such ‘classical’ value it gives you a fairly everything-else-independent quantity (although physical disorganization, such as row-chaining, would blur the picture and somehow show up as the number of block buffers would be higher than normal.). These values are given in columns ‘query’ and ‘current’ in the tkprof output, but one can also get them easily by running a simple script such as *bufgets.sql* :

```
select sum(s.value) buffers_accessed
from v$mystat s,
     v$statname n
where n.name in ('db block gets', 'consistent gets')
      and n.statistic# = s.statistic#
/
```

before the statement to test, run the statement, run *bufgets.sql* once again and compute the difference.

The two competing PL/SQL codes have been benched on a 50,000 rows table (454 Oracle blocks), with a single index (primary key).

In both cases, the processing of a new key ‘costs’ 8 blocks. By measuring independently the INSERT statement, we can determine that the (in fact similar) unsuccessful SELECTs cost 4 in both cases, and the INSERT 4 as well. However, the beginner’s code will cost 13 when processing an already existing key, and the old Oracle hand’s one only 7. Assuming that 10% of values are new ones, the scores for 100 lines will be :

Beginner : $90 \times 13 + 10 \times 8 = 1250$

Old timer : $90 \times 7 + 10 \times 8 = 710$, or almost 2 times as efficient, although the beginner has done no obvious mistake...

As exceptions begin to pop-up, one can wonder why we bother to SELECT first, and whether it would not be more efficient to let Oracle do most of the job itself. We would then have two cursors instead of three, which should be a minor gain both in terms of SGA memory consumption and parsing. But here again, we can write it in two different ways :

```
update my_table
set ...
where id = v_id;
if SQL%ROWCOUNT = 0
then
  insert into my_table(id, ...)
  values (v_id, ...);
end if;
```

or

```

begin
  insert into my_table(id, ...)
    values (v_id, ...);
exception
  when dup_val_on_index then
    update my_table
      set ...
      where id = v_id;
end;

```

Intuitively, one may think that if we have more rows to update, the first case will be better, and if we have more rows to insert, the second will execute faster. In fact, this is more than half wrong.

Let's use our 'buffer access' cost :

If we try insert first :

New key : cost = 4 blocks
 Already existing key : cost = 22 blocks

If we try update first :

New key : cost = 8 blocks
 Already existing key : cost = 6 blocks

In fact, the most efficient way is to update, and insert if not found, even if odds are 50/50. The reason is that the 'duplicate value' event is detected only after having inserted the index(es), which is done AFTER having inserted the row (and updated the rollback segments), whereas before updating you must search the index and immediately see if the value is here or not - in other words, the insert generates a 'post' exception, whereas the update generates a 'pre' exception (kind of).

Here is what happens in both cases :

Try insert first :

- a) Update the data block, write the rollback segment, update index(es), write the rollback segment.

If the data already exists, it is likely that all the indexes will not be updated, although we have no certainty that unique indexes are inserted first. We can be pretty sure that some internal rollback will take place somewhere. Then you will have to switch to the update, which means :

- b) searching the index once again (not very likely that Oracle will keep track of the rowid of the existing line), update the datablock, write the rollback segment, and possibly update some indexes

as well.

Try update first :

Search the index. If found, same processing as b) above. If not found, same processing as a). Much more straightforward.

If we once again compute the cost for 100 rows of which 10 are new :

Insert first :

$$90 \times 22 + 10 \times 4 = 2020,$$

or

almost twice as much as the unsuspecting beginner...

Update first :

$$90 \times 6 + 10 \times 8 = 620,$$

or

more than 10% better than our previous best case...

It could be slightly different of course in a real-life case with more indexes.

However if you wish to find out how effective “**insert first**” is the number of new lines is the key. So how many new lines equates to the cost factor.

We can compute the resultant cost factor by:

$$\begin{aligned} (100 - \text{new_lines}) \times 22 + 4 \times \text{new_lines} &< (100 - \text{new_lines}) \times 6 + 8 \times \text{new_lines} \\ 1600 &< 20 \times \text{new_lines} \end{aligned}$$

As a result we now have our rule of thumb:

The **insert first** becomes less costly only if

you have 80% of rows to insert

against 20% of rows to update.

So much for intuition...

A conclusion ? Watch not only your statements, but your algorithms too!