

IT'S THE ALGORITHM, STUPID !

Stéphane Faroult, Oriole
sfaroult@oriole.com

A lot of attention is devoted (with some justification) to SQL tuning. However, focusing on individual SQL statements often misses an important point and is only half the story; a lot of impressive performance gains can be made by changing algorithms. This presentation will show why the combination of 'good' statements can be bad, and how new and not-so-new improvements to the SQL syntax can lead to dramatic performance gains by reducing the number of statements and simplifying algorithms.

1. Going farther than SQL tuning

Too often, a number of SQL statements which are not dreadful by themselves are combined in an utterly inefficient way. By closely – too closely – following the specified algorithms, developers, whose grasp of SQL is sometimes hardly adequate, often miss on the set-handling capabilities of Oracle, among other things, and rewrite in an inefficient way what Oracle can natively perform much faster. The aim of this paper is to bring attention to a number of common mistakes. Although PL/SQL is used throughout the paper as an example language, most of what follows is in no way PL/SQL specific and could as well be applied to Pro*C , JSQL, oraperl or whatever. It must be emphasized that improving algorithms, as defined in this paper, mostly consists in shifting a large part of the logic from the 'wrapper code', in which SQL statements are imbedded, to the SQL statements themselves – something which cannot be done without a strong command of the SQL language.

2. How "good" SQL can be relatively inefficient

The tuning process usually concentrates on finding the 'bad SQL' and tuning it, with often dramatic performance gains. However, some other candidates for tuning may be forgotten by beginners. Mediocre SQL executed very often (and not always easy to spot when bind variables are not used) may provide greater improvements. But even people with some experience can totally miss on SQL which on face value is beyond all reproach. We are going to illustrate this with three examples.

2.1. The UPSERT example

One of my favourite examples is the way many developers handle the usual :

```
If item is present
Then update it
Otherwise insert a new row
```

More often than not you see it coded as :

```
select count(*)
into cnt
from my_table
where primary_key = id;
if (cnt = 0)
then
  insert into my_table
```

```

    values (id, ...);
else
    update my_table
    set ...
    where primary_key = id;
end if;

```

Take any of these statements individually, it's hard to see how one can improve them. Take them globally, now. Granted, a primary key index search doesn't cost much. But it doesn't cost nothing, either. What is a primary key search? You hit the root of a tree, then descend it doing a few comparisons, then get a *rowid* (physical address); with this row address, you can reach the table block and use some kind of table inside it to reach the proper row (this is a simplified view). The initial select doesn't need to hit the table, since it just needs to count index entries. The insert goes to the table first, 'remembers' the address where the row has been inserted, then records it as a new index entry. The update, however, goes to both the index and the table. However, the select has gone through the index already, hasn't it? Since all we need from the index is the rowid, do we need to do the index search twice? Certainly not.

We could write an improved version returning the rowid in the SELECT and using it in the UPDATE – it would save one index search. But at this stage we can wonder why we couldn't directly update the table; Oracle provides a kind of system variable, `SQL%ROWCOUNT`, which tells how many rows were affected by the last SQL statement. Since updating nothing is not an error, we just have to write:

```

Update my_table
Set ...
Where primary_key = id;
If (SQL%ROWCOUNT = 0)
Then
    Insert
End if;

```

Note that we are doing exactly what the pseudo-code requests; however, we are using the ability of Oracle to give us a post-action report, instead of using the classical defensive technique of checking beforehand. Typically, basic coding (using nothing but primary keys) accesses 11 blocks when the row is found, and 22 when it is not. The version which directly tries the update respectively accesses 6 and 12 blocks. This may seem a minor gain; not so minor when such a statement is run thousands of times per hour – if you assume that you have 30% of inserts and 70% of updates, for instance, you will access in the second case only 55% of the blocks accessed in the first case. Elapsed time being more or less in line with the number of blocks accessed, it can make all the difference between the batch job which fits nicely in the nightly schedule and the batch job which doesn't.

2.2. The nested loop example

Another very common mistake is to explicitly code the iterative parts of a process. Because developers will follow blindly the algorithm, they will open a cursor – so far so good – loop, and inside the loop open another cursor which takes one or several of the values returned by the outer loop as parameters.

Simple example, let's say that we have a table `ORDERS` which references both `CUSTOMERS` and `ARTICLES`, and that we have to write a batch process to set a flag to mark customers who have not ordered anything in the past month as 'inactive' (presumably either to mail them to try to win them back – or to no longer send them an expensive catalogue).

In most cases the algorithm will be expressed as

```
For each customer
  If the customer has ordered nothing in the past month
    Mark the customer as inactive
```

Which in 99% of cases will be coded as follows :

```
declare
  cnt  number;
begin
  for rec in (select customer_id from customers)
  loop
    select count(*)
    into cnt
    from orders
    where date_entered > add_months(sysdate, -1)
      and customer_id = rec.customer_id;
    if (cnt = 0)
    then
      update customers
      set customer_active = 'N'
      where customer_id = rec.customer_id;
    end if;
  end loop;
end;
/
```

We have indexed the (`CUSTOMER_ID`, `DATE_ENTERED`) columns in the `ORDERS` table. When we have a look at the queries, everything just looks fine; indeed the outer `SELECT` scans all the customer ids and can find them in the primary key, which is certainly a far lesser evil than scanning the `CUSTOMERS` table (3,000 rows in the test database); and it's a step we cannot easily elude. The counting of recent orders uses a purposely-built index and the update uses no less than the primary key. In our test, 2371 rows had to be updated (we had 3500 orders), which was done in 6.5 seconds, using 3.8 seconds of CPU and 16683 logical reads.

This can however be improved; a minor change consists (as already suggested in the `UPSERT` example) in returning the `ROWID` with the `CUSTOMER_ID` in the `SELECT`, and using it to do the update (about the same elapsed time, but only 11941 logical reads). However, the best thing to do is obviously not to loop – or rather, let Oracle do it.

```
update customers c
set c.customer_active = 'N'
where not exists (select null
                  from orders o
                  where o.date_entered > add_months(sysdate, -1)
                    and o.customer_id = c.customer_id);
```

Very same result – but in about 3.2 seconds, 0.7 seconds of CPU and 8803 logical reads. And people who have been told again and again never to use `NOT IN` but always `NOT EXISTS` may be surprised to discover that on the test database :

```
update customers
set customer_active = 'N'
where customer_id not in (select /*+ MERGE_AJ */ customer_id
                        from orders o
                        where o.date_entered > add_months(sysdate, -
1));
```

does the very same thing in 3 seconds, only 0.46 seconds of CPU and a pitiful 2797 logical reads. Granted, without the hint it's by far the worst result!

The point is that between 'formal' specifications as expressed above and the final query there is a gap which may not be obvious to jump at once for a developer. However, when you compare the results of the query with decent PL/SQL code following closely the specs, we have nevertheless achieved a gain of more than 52% in elapsed time, almost 88% in CPU consumption and slightly above 83% in the number of logical reads. This is typically in the same order of magnitude as 'good SQL' v.s. 'bad SQL' – except that we had no bad SQL to start with – just a poor algorithm, or perhaps more exactly an algorithm which was unsuited to the set-processing capabilities of SQL..

2.3. The modular programming example

Young programmers are often eager to apply everything they have heard about 'good practice' – and especially modular programming. Object Oriented Programmers are especially redoubtable, since they are more often than not eager to provide one specialized function to return the value of each column in a row.

The following example is unfortunately a true one, directly taken from `DBA_SOURCE` in a production database :

```
Function CST_ExistDG (vTstId in number) Return number is
-- =====
    cpt    number;

Begin

    select count(*)
    into    cpt
    from    cst2_test_etape
    where   test_id = vTstId
    and     etp_id  = 3;

    if cpt = 0 then
        return 0;
    else
        return 1;
    end if;

End CST_ExistDG;
```

The author of this outstanding example of PL/SQL programming should deserve public flogging.

- a. First, if existence is just what is checked, a `WHERE ROWNUM = 1` is just what is needed. In the best case, the `count (*)` will do a fast full scan – which, however fast, will always be slower than stopping at the first occurrence. The advantage of `count (*)` is of course that it returns 0 when nothing is found. Handling the `no_data_found` exception was probably a bit above the talents of the developer.
- b. Second, as demonstrated in the ‘Insert or Update’ case above, an independent existence test is very rarely necessary and this function was perfectly dispensable.

This is a vicious piece of rotten software, because people just looking for ‘bad SQL’ are likely to totally overlook it. Oh, yes, the process is slow, oh yes, this function is executed a large number of times, but look, it’s a query which uses the two first columns in the primary key, there is nothing we can do to speed it up! No ugly full scan, and it’s fast! You probably need to buy another processor.

3. Spotting problem stored procedures

How can we spot ‘problem’ stored procedures? A careful code review is an obvious answer, but more a theoretical than a practical one – especially when you are dealing with applications which have already been in production for a while. Do you know how to identify problem SQL? Then you are pretty able to identify problem PL/SQL too.

3.1. Run-time checks

`V$SQL` (OR `V$SQLAREA`) is usually the place to start with to find the bad SQL statements, those which really hog your system down. It’s a fine place for PL/SQL too. Any call to a PL/SQL anonymous block or stored procedure is recorded inside `V$SQL`, with its number of executions, its number of physical reads, and, most importantly, the number of logical reads (`buffer_gets`) it required. Unfortunately, the various SQL statements they may call are also accounted for separately, and there is nothing to this author’s current knowledge to relate a SQL statement to the piece of PL/SQL which executed it. However, a query such as the following one can be quite useful in identifying ‘bad PL/SQL’ :

```
select sql_text, buffer_gets, round(100 * buffer_gets / t.tot, 1) PCT
from v$sql,
     (select sum(buffer_gets) tot
      from v$sql
      where command_type != 47
        and executions > 0) t
where command_type = 47
     and executions > 0
     and round(100 * buffer_gets / t.tot, 1) > 1
order by 2 desc
/
```

Value 47 for `command_type` identifies PL/SQL code; this helps you see how much they contribute to the total number of logical reads.

If you can afford to trace processes, `tkprof` may of course be able to show you interesting things. If the executions per second figure is good but the procedure still accounts for a significant percentage of the total, you can guess that something is wrong.

3.2. Source analysis

Source analysis of the automated sort can also be quite valuable at times. One thing which must be understood is that SQL was designed from the start to operate on SETS. PL/SQL brought a lot in terms of exception handling; however, something such as the beloved loop calling a cursor is of less obvious use. It has had its use – the restriction on using `CONNECT BY` with joins once made ancillary cursors quite useful. However, the advent of inline views with Oracle 7.2 has made a lot of usages of loops calling cursors redundant.

Loops may have their use – committing at regular intervals springs to mind, although even this is usually questionable since rollback segments (or undo tablespaces) should be sized to suit business processes. Most of the time they are very badly used. Which is why a very raw analysis of the number of loops in the code can provide a fairly good indicator of the quality of the code. A brute force query such as the following one :

```
col dummy noprint
set num 8
select substr(owner ||'.'||name, 1, 60) proc,
       type,
       floor(sum(decode(instr(upper(text), 'LOOP'), 0, 0, 1))/2)
loops,
       sum(decode(instr(upper(text), 'SELECT'), 0, 0, 1)) +
       sum(decode(instr(upper(text), 'INSERT'), 0, 0, 1)) +
       sum(decode(instr(upper(text), 'DELETE'), 0, 0, 1)) +
       sum(decode(instr(upper(text), 'UPDATE'), 0, 0, 1)) SQL,
       count(*) lines,
       (floor(sum(decode(instr(upper(text), 'LOOP'), 0, 0, 1))/2) *
       (sum(decode(instr(upper(text), 'SELECT'), 0, 0, 1)) +
       sum(decode(instr(upper(text), 'INSERT'), 0, 0, 1)) +
       sum(decode(instr(upper(text), 'DELETE'), 0, 0, 1)) +
       sum(decode(instr(upper(text), 'UPDATE'), 0, 0, 1))))/
       count(*) dummy
from dba_source
where owner not in ('SYS', 'SYSTEM', 'CTXSYS')
group by substr(owner ||'.'||name, 1, 60), type
having floor(sum(decode(instr(upper(text), 'LOOP'), 0, 0, 1))/2) > 1
       and sum(decode(instr(upper(text), 'SELECT'), 0, 0, 1)) +
       sum(decode(instr(upper(text), 'INSERT'), 0, 0, 1)) +
       sum(decode(instr(upper(text), 'DELETE'), 0, 0, 1)) +
       sum(decode(instr(upper(text), 'UPDATE'), 0, 0, 1)) > 1
order by 6 desc
/
```

can provide quite interesting pointers (even if the figures it computes are not fully accurate - no provision is made to ignore comments and one `SELECT` with a subquery counts as two `SELECT`s). Basically, the greater the number of loops and the number of SQL statements in relation to the number of lines of the procedure or package, the greater the room for improvement (in the query we multiply the number of loops by the number of SQL statements to take into account the devastating effect of SQL called in loops).

3.3. Testing for performance

Once the ‘bad ‘ PL/SQL is identified, rewriting it and checking how it works is not always very easy. A difficulty is that the so convenient `SQL*Plus` command

```
set autotrace on
```

simply doesn't work with a PL/SQL procedure. There is no such thing as an 'execution plan' for a PL/SQL block, however, some statistics can be collected if you are allowed to select from V\$SESSION, V\$STATNAME and V\$SESSTAT.

Here are two procedures to sandwich a PL/SQL block you want to test :

```
-- before.sql
set echo off
set timing off
set recsep off
column CPU noprint new_value before_cpu
column READS noprint new_value before_reads
select s_cpu.value CPU,
       sum(s_reads.value) READS
from sys.v_$session se,
     sys.v_$statname n_cpu,
     sys.v_$statname n_reads,
     sys.v_$sesstat s_cpu,
     sys.v_$sesstat s_reads
where n_reads.name in ('db block gets', 'consistent gets')
     and n_cpu.name = 'CPU used by this session'
     and n_cpu.statistic# = s_cpu.statistic#
     and n_reads.statistic# = s_reads.statistic#
     and s_cpu.sid = se.sid
     and s_reads.sid = se.sid
     and se.audsid = userenv('SESSIONID')
group by s_cpu.value
/
column CPU clear
column READS clear
```

will display nothing but blank lines but will collect values before your PL/SQL runs; immediately after your PL/SQL, run this :

```
-- after.sql
set echo off
set timing off
set recsep off
column CPU print format 999999
column READS print format 999999999999999
select s_cpu.value - &&before_cpu - 97 CPU,
       sum(s_reads.value) - &&before_reads - 10 READS
from sys.v_$session se,
     sys.v_$statname n_cpu,
     sys.v_$statname n_reads,
     sys.v_$sesstat s_cpu,
     sys.v_$sesstat s_reads
where n_reads.name in ('db block gets', 'consistent gets')
     and n_cpu.name = 'CPU used by this session'
     and n_cpu.statistic# = s_cpu.statistic#
     and n_reads.statistic# = s_reads.statistic#
     and s_cpu.sid = se.sid
     and s_reads.sid = se.sid
     and se.audsid = userenv('SESSIONID')
group by s_cpu.value
/
column CPU clear
column READS clear
```

3.4. Beyond PL/SQL ?

Obviously, all the algorithm weaknesses mentioned above are in no way peculiar to PL/SQL – you can write poor algorithms in any language (although developers familiar with one language are some times more prone to a specific type of mistake – Cobol programmers may be more keen on useless nested loops while object-oriented developers may feel better with a set of ‘methods’ executing one SELECT per column they want to be returned). However, since the source may be less readily available than when stored in the database, and since statements only appear as individual statements, spotting simply from database activity the poorly written programs is a daunting, although not quite impossible, task. There are a number of interesting queries to be run. Just try something like that :

```
select sql_text, executions, buffer_gets
from v$sql
where parsing_user_id != 0
order by 2 desc
/
```

(set pause on helps). Some statements may be extremely interesting to spot – especially those SELECTs from a single table or UPDATEs which have a single condition - WHERE ID = :b1 – ID is of the course the primary key – which are executed a HUGE number of times. What is wrong with that? If the primary key is a technical value – some numerical id generated by a sequence – you can be sure that the programming is bad. An end-user will never input an internal identifier. Somewhere, somehow, the internal identifier which is fed into the query has been itself collected via a query. It stinks of a reprogrammed join.

SELECT ... FOR UPDATE closely followed by an UPDATE of the same table with (usually) the very same WHERE clause is also an easily recognizable pattern.

3.5. Red flags

This is a (non exhaustive) list of things which should raise the attention :

- ❑ A succession of SQL statements referencing the same tables
- ❑ A succession of SQL statements with very similar WHERE clauses
- ❑ Calls in statements to functions which return the result of another SELECT
- ❑ SELECTs from a single table where the only reference in the WHERE clause is to an internally defined identifier which is derived from elsewhere.
- ❑ SELECT for UPDATE followed by the matching UPDATE
- ❑ SELECT COUNT(..) FROM ... - probably an existence test

Some of these constructs may be perfectly legitimate. Candidly asking why it was coded this way cannot hurt.

4. The SQL weapons

Spotting poor PL/SQL and trying to replace it with bright SQL is fine, but how can we do it ? The brightness of SQL depends on the talents of the developer. What matters is to avoid short-sightedness, to be able to ‘zoom out’ to see how several statements, functions, etc. can be combined together in possibly a single SQL statement, and then to ‘zoom in’ and make of this SQL statement a fine, efficient one. It would be too easy if there were one magical recipe, a panacea solving all problems.

What is our goal?

Basically, minimizing the number of calls to Oracle and the number of times we have to access each table. Don't think of data in Oracle as of things you can get at the shop round the corner, but as of things you buy at a distant mall, and you'll get it. Better to have a long shopping list and buy everything at once. Here are a few recipes, which are worth trying.

4.1. *Inline views*

Inline views are a very effective tool to avoid a succession of SELECTs from a series of distinct tables with pretty similar structures. Subtypes often are poorly handled in databases and you often encounter cases when you want to consolidate data which you have to collect from several tables. Assume for instance that you are a travel agency, that you sell airplane seats, cruises and coach rides and that depending on the product you sell you store the information about your customers in a different table. Now, let's say that you want to compute how much you sold between two dates.

The brutish, procedural approach favoured by many would be something like :

```
select sum(amount)
into flight_revenue
from flights
where sale_date between ... and ... ;
select sum(amount)
into cruise_revenue
from cruises
where sale_date between ... and ... ;
select sum(amount)
into coach_revenue
from coach_rides
where sale_date between ... and ... ;
total_revenue := flight_revenue + cruise_revenue + coach_revenue;
```

This can be replaced with a single :

```
select sum(amount)
into total_revenue
from (select amount
      from flights
      where sale_date between ... and ...
      union all
      select amount
      from cruises
      where sale_date between ... and ...
      union all
      select amount
      from coach_rides
      where sale_date between ... and ...);
```

It will usually get the same result faster – how much faster depends of depends on the complexity of each statement. But it's often an excellent means to diminish the number of statements in a procedure.

4.2. *Ode to join*

Many developers seem to live in a permanent fear of joins, probably because of persistent rumours saying that joins are costly. As a matter of fact, trying to hide joins

by using functions or cursors open in loops is just reprogramming them – which is likely to be done much more poorly than by the Oracle kernel (counter-examples coming below). The problem is not with having several tables in the `FROM` clause – the problem is with fetching data from a lot of different places, which is totally different. A reprogrammed join will have a single way to proceed : nested loops. This is strongly reductive of Oracle’s capabilities, which, besides nested loops also knows hash joins (very efficient when large volumes are returned) or merge joins, which also have their use – as the `NOT IN` example above proves it. Since its introduction, the cost-based optimiser has made a lot of progress and keeps improving with each release. By coding a complicated algorithm, you are not only reinventing the wheel and recoding what Oracle knows how to do better, but you are also locking it out of any chance of improvement in the future – ‘PL/SQL rewriting’ is still a long way off ...

4.3. WITH

Oracle9 introduces something which can be very interesting in some cases, which is subquery factoring. The idea of subquery factoring is to reference a subquery once and for all at the beginning of a SQL statement, assign a name to it and then reference the name – the subquery (which obviously musn’t be correlated) will be evaluated only once (`WITH (subquery) AS alias SELECT ...`) This can be quite useful in some cases – complex UNIONS referencing the same subquery at different places, typically.

4.4. Analytic functions

The analytic functions introduced with release 2 of Oracle8i can sometimes help replace hundreds of lines of PL/SQL code with a single SQL statement which will be incredibly faster. Even if a task looks like requiring procedural processing, the intra-row comparisons and computations provided by analytic functions may save the day.

4.5. When PL/SQL lends a hand

Now, after having hit hard PL/SQL, one must acknowledge that it can bring a number of very significative improvements. Here are a few examples.

4.5.a Bulk operations

Bulk operations (`BULK COLLECT` for `SELECT`s and `FORALL` for `INSERT`s, `DELETE`s and `UPDATE`s) are extremely efficient. If you want to commit every say 1000 rows, what about using PL/SQL tables, storing 1000 rows in them, and doing the operation in one call then committing instead of doing it 1000 times and committing will show a huge improvement. You still have loops, but of a gentler kind.

4.5.b Packaged variables and arrays

Packages have a big advantage : you can declare variables, and array of variables, inside them. And one of their nice features is that you can have an initialisation section called once (the first time you reference the package). If you really need some kind of lookup function which will be frequently called, it may be worth considering caching a small lookup table inside a PL/SQL in a package. Tests have proved with a 300-odd rows, suitably indexed reference table used for code conversion, that by loading everything with an initial `BULK COLLECT` and then performing a dumb sequential search on the PL/SQL table, 100,000 calls were performing roughly 17 times faster than executing the

SELECT at each call. Loading everything in memory is probably not something to recommend for OLTP, but it can be efficient for batch programs.

5. Conclusion

Yes, tracking the bad SQL is a very good thing to do. But no longer having tremendously bad SQL doesn't mean the end of the road – there is usually still a large room for improvement when you consider the code. People sometime blink at the idea of rewriting part of the code – it seems much more of an endeavour than simply adding an index here and a hint there. However, it is worth the effort – gaining a factor 60 is not that uncommon. Moreover, the resulting code is usually much simpler – and much easier to maintain.