Just Don't Do It

Sins of omission and commission

Jonathan Lewis

jonathanlewis.wordpress.com

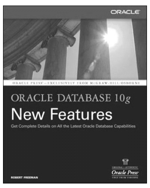
www.jlcomp.demon.co.uk

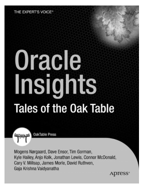
My History

Independent Consultant

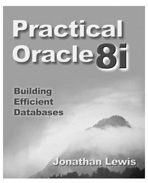
35+ years in IT
30+ using Oracle (5.1a on MSDOS 3.3)

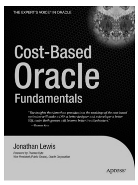
Strategy, Design, Review,
Briefings, Educational,
Trouble-shooting

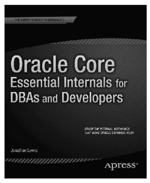




Oracle author of the year 2006
Select Editor's choice 2007
UKOUG Inspiring Presenter 2011
ODTUG 2012 Best Presenter (d/b)
UKOUG Inspiring Presenter 2012
UKOUG Lifetime Award (IPA) 2013
Member of the Oak Table Network
Oracle *ACE Director*
O1 visa for USA







How to spend less time on a job

- Don't do it
  - Do it less often
    - Do it in a quiet period
      - Do it more efficiently
        - Don't do it in little bits

Internal Avoidance Mechanisms

Infrastructure

- Indexing
- Materialized views
- Storage Indexes
- Zone Maps

Caching

- Scalar Subquery Caching
- Result Cache
- Deterministic functions
- Pragma UDF

Optimizer

- Partition elimination
- Bloom filters
- Join Elimination
- Partial Join Evaluation

The optimizer and run-time engine have ***many*** mechanisms for reducing work, or avoiding repeating work they have done once. We can learn from these principles.

Superfluous Updates (a)

```
                                                       CPU      Elapsd
 Physical Reads  Executions  Reads per Exec %Total Time (s)  Time (s) Hash Value
--------------- ------------ -------------- ------ -------- --------- ----------
      2,951,745            1    2,951,745.0   13.3   750.49   1306.68 3185433958
Module: JDBC Thin Client
update HISTORY SET FLAG=0 WHERE CLASS = 'x'
```

```
update  history set flag =  0
where   class  = 'x'
and     flag  != 0;
```

Important point: "flag" had been declared NOT NULL.

Updates a few hundred rows instead of 5 million.
This halved the elapsed time - but still did a very big tablescan

http://jonathanlewis.wordpress.com/statspack-distractions/
https://jonathanlewis.wordpress.com/2019/09/08/quiz-night-34/
(quiz answer: 12.2 makes the original statement more expensive)

This was from a ***statspack*** report taken from an overnight batch job. Step 1 - don't update data that isn't going to change (**<u>unless</u>** you really want to lock it anyway).

Superfluous Updates (b)

```
create index hst_idx on history(
        case when class = 'x' and flag != 0 then 1 end
);
```

*This index as small as it could be, identifies **exactly** the data we are interested in and no more, and is most unlikely to be used by any other SQL in the system.*

```
select column_name from user_ind_columns         -- find the hidden column name
where  table_name = 'HISTORY' and index_name = 'HST_IDX';

begin
   dbms_stats.gather_table_stats(
        user,
        'history',
        method_opt=> 'for all hidden columns size 1'
--      method_opt=> 'for columns sys_nc00019$ size 1'
   );
end;
/
```


Step 2: Add a high precision, ***minimum-risk*** index. Recent versions of Oracle collect index stats automatically but you still need to gather ***column*** stats.

Superfluous Updates (c)

```
select  state, flag from history
where   case when flag = 'x' and state != 0 then 1 end = 1
;

| Id | Operation                    | Name    | Rows | Bytes | Cost  |
|  0 | SELECT STATEMENT             |         |   28 |   196 |     5 |
|  1 |  TABLE ACCESS BY INDEX ROWID| HISTORY |   28 |   196 |     5 |
|* 2 |   INDEX RANGE SCAN           | HST_IDX |   28 |       |     1 |

Predicate Information (identified by operation id):
   2 - access(CASE  WHEN ("FLAG"='x' AND "STATE"<>0) THEN 1 END =1)


alter table t1 add x_status /* invisible */
generated always as (
        case when flag = 'x' and state != 0 then 1 end
) virtual
;
```


In 11g you're more likely to create a virtual column on the table and create an index on the virtual column. In 12c you can also declare the column invisible.

Don't repeat the work (a)

```
update  small_table t
set     fcr7 = (
                select fcr7 * 100               -- where's the alias?
                from    slow_view d
                where   d.monat = t.monat
                and     d.dl    = t.dl
        )
```

- The update takes 3 to 8 hours to run.
- There are 84 rows in ***small_table***
- It takes about 7 minutes to execute ***"select * from slow_view"*** returning 63 rows

See https://jonathanlewis.wordpress.com/2016/01/08/ctes-and-updates/ for write-up

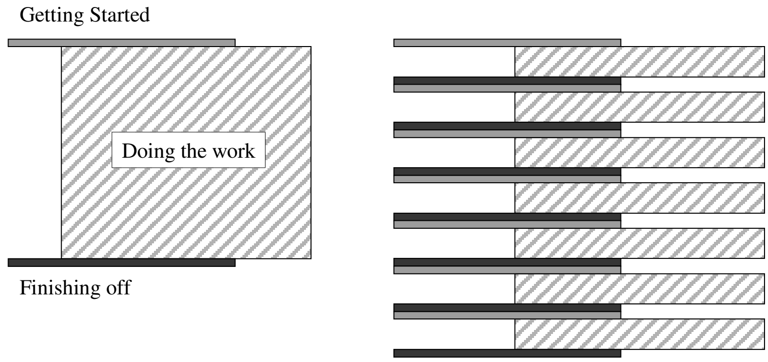
Don't repeat the work (b)

```
update  small_table t
set     fcr7= (
                with d as (
                select /*+ materialize */
                        fcr7, monat, dl
                from    slow_view
                )
                select  fcr7 * 100
                from    d
                where   d.monat = t.monat
                and     d.dl    = t.dl
     )
```

- The query against ***slow_view*** now runs once.
- The 84 row correlated update will scan a small "temporary table" 84 times
- It should take about 7 minutes to execute the statement

It might have been possible to rewrite the update as a MERGE command - but that might have been difficult. Maybe there was a safe way to use the SQL result cache.

Don't do it in little bits



SQL*Net round-trips
PL/SQL single row processing
Inline Scalar subqueries
Nested loops !
Buffer gets - costs !

Array Fetching (a)

This query takes ***28 seconds*** to run - how can I make it go faster ?

```
select   /*+ full(my_big_table)  */
         max (id) id
from
         my_big_table
group by
         other_id, event, company_id, security_id;
```

```
| Id  | Operation             | Name          | Rows  | Bytes |TempSpc|
|   0 | SELECT STATEMENT      |               | 7951K|   257M|       |
|   1 |  SORT GROUP BY        |               | 7951K|   257M|   365M|
|   2 |   PARTITION RANGE ALL|               | 7951K|   257M|       |
|   3 |    TABLE ACCESS FULL | MY_BIG_TABLE  | 7951K|   257M|       |
```

That's not bad for scanning and aggregating (at least) 257MB / 8 million rows of data.

A ***covering index*** with an index fast full scan was "a little" faster.
A full scan might ***avoid the sort*** - if it were possible (nulls and partitions make this harder)
Running ***parallel*** might be faster - or might give a clue about performance

The covering index was about half the size of the table. It's an expensive strategy with massive potential for unexpected side effects, and only 8 seconds saving.

Array Fetching (b)

Step 1: where do you spend the time ?

```
set autotrace on statistics

Statistics
        91       recursive calls
        10       db block gets
    224115       consistent gets
     10578       physical reads
         0       redo size
  25944773       bytes sent via SQL*Net to client
   1200334       bytes received via SQL*Net from client
    109080       SQL*Net roundtrips to/from client
         0       sorts (memory)
         1       sorts (disk)
   1636183       rows processed
```

set arraysize 1000 -- Path with index fast full scan dropped to 4 seconds

set JDBC connection property "defaultRowPrefetch" (default 10)
… etc.

You could (should) enable tracing but in this case ***autotrace*** held a big clue about the problem: small array fetches. (Why does someone want 1.6M "raw" rows anyway?)

Constant functions (a)

```
----------------------------------------------------------------------------------------------
|Id |Operation                   | Name               |Starts|A-Rows|   A-Time   |Buffers |
----------------------------------------------------------------------------------------------
| 0 |SELECT STATEMENT            |                    |    1 |    1 |00:00:11.70 |  38245 |
| 1 | NESTED LOOPS               |                    |    1 |    1 |00:00:11.70 |  38245 |
| 2 |  NESTED LOOPS              |                    |    1 |    1 |00:00:11.70 |  38244 |
|*3 |   HASH JOIN OUTER          |                    |    1 |    1 |00:00:11.70 |  38242 |
|*4 |    TABLE ACCESS FULL       | ICX_SESSIONS       |    1 |    1 |00:00:11.70 |  38165 |
|*5 |    TABLE ACCESS FULL       | FND_RESPONSIBILITY |    1 | 2192 |00:00:00.01 |     77 |
|*6 |   INDEX UNIQUE SCAN        | FND_USER_U1        |    1 |    1 |00:00:00.01 |      2 |
| 7 |  TABLE ACCESS BY INDEX ROWID| FND_USER          |    1 |    1 |00:00:00.01 |      1 |
----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   3 - access(RESPONSIBILITY_ID=ICX.RESPONSIBILITY_ID)
   4 - filter((ICX.DISABLED_FLAG<>'Y' AND ICX.PSEUDO_FLAG='N' AND
          icx.last_connect > sysdate@!-
          to_number(nvl(fnd_profile.value('icx_session_timeout'),'30'))/60/24))
   5 - filter(NVL(ZD_EDITION_NAME,'ORA$BASE')='V_20200519_1939')
   6 - access(USR.USER_ID=ICX.USER_ID)
```


The tablescan at operation 4 is taking 11.7 seconds for only 38,000 buffers. So what's the predicate being tested for every row. Note the PL/SQL function (FND is a clue).

Constant functions (b)

```
icx.last_connect > (select sysdate@! -
        nvl (fnd_profile.value ('icx_session_timeout'),'30')/60/24 from dual)

------------------------------------------------------------------------------------------
|Id |Operation              | Name               |Starts | A-Rows |   A-Time   | Buffers |
------------------------------------------------------------------------------------------
| 0 |SELECT STATEMENT       |                    |     1 |     14 |00:00:00.35 |   38584 |
|*1 | HASH JOIN RIGHT OUTER|                     |     1 |     14 |00:00:00.35 |   38584 |
|*2 |  TABLE ACCESS FULL    | FND_RESPONSIBILITY |     1 |   2192 |00:00:00.01 |      76 |
|*3 |  HASH JOIN            |                    |     1 |     14 |00:00:00.34 |   38508 |
| 4 |   TABLE ACCESS FULL   | FND_USER           |     1 |   2627 |00:00:00.01 |     121 |
|*5 |   TABLE ACCESS FULL   | ICX_SESSIONS       |     1 |     14 |00:00:00.34 |   38387 |
| 6 |    FAST DUAL          |                    |     1 |      1 |00:00:00.01 |       0 |
------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access(RESPONSIBILITY_ID=ICX.RESPONSIBILITY_ID)
   2 - filter(NVL(ZD_EDITION_NAME,'ORA$BASE')='V_20200519_1939')
   3 - access(USR.USER_ID=ICX.USER_ID)
   5 - filter((ICX.DISABLED_FLAG<>'Y' AND ICX.PSEUDO_FLAG='N' AND
          ICX.LAST_CONNECT>))
```


Technically the subquery against ***dual*** operates as a filter subquery - running once per row examined; but ***scalar subquery caching*** means it only needs to run once.

Responses Offered

Appropriate Responses

Which version of Oracle ?

Where is the time going ?

  Is the time spent on the select or the insert ?

Is this a single big batch load, or lots of small batch inserts ?

Is there any data in the table before you start

Bad response

"Never do in PL/SQL that which can be done in plain SQL"

There were a number of responses to the original question.
In this case many of them were good questions to clarify the nature of the problem.

"Never do in PL/SQL …"

```
declare
    cursor c1 is select * from t2;
    type c1_array is table of c1%rowtype index by binary_integer;
    m_tab c1_array;
begin
    open c1;
    loop
        fetch c1 bulk collect into m_tab limit 100;
        begin
            forall i in 1..m_tab.count save exceptions
                 insert into t1 values m_tab(i);
        exception
            when {ORA-24381} then ... -- exception handling code
        end;
        exit when c1%notfound;
    end loop;
    close c1;
end;
```


Q: Why do this instead of a simple ***"insert into t1 select * from t2"*** ? A: It's an efficient way to handle the occasional error without producing a massive rollback..

Never … ?

Time →

Pure SQL

Bulk collect / ForAll

} The standard comparison

Pure SQL with rollback due to failure

Bulk collect / Forall

} But what if it fails ?

Validation mechanisms to allow Pure SQL | Pure SQL

Bulk collect / Forall

} And when was it validated

I see this dogmatic criticism too often - *"You should use an SQL statement and not a PL/SQL procedure"*. This demonstrates an absence of a proper cost/benefit analysis.

SQL vs. PL/SQL (a)

I want a mechanism that breaks a table down into a number of chunks that (plus or minus 1) all hold the same number of essentially consecutive blocks.

Can it be done in pure SQL ?

Simplified starting point - assume the table is a single segment (non-partitioned)

Of course it can be done - but ***should*** it be done ?

```
with extents as (
        select  file_id, block_id, blocks
        from    dba_extents
        where   owner = upper('&m_owner')
        and     segment_name = upper('&m_segment')
),
expansion as (
        select  --+ materialize
                rownum id
        from dual
        connect by
                level <= (select max(blocks) from extents)
),
expanded_blocks as (
        select
                ext.file_id, ext.block_id, ext.blocks,
                ext.block_id + exp.id - 1       individual_block
        from
                extents         ext,
                expansion       exp
        where
                exp.id <= ext.blocks
),
tiled as (
        select
                file_id, block_id, individual_block,
                ntile(&m_tiles) over (order by file_id, individual_block) tile
        from
                expanded_blocks
),
ranges as(
        select
                file_id,
                tile,
                min(individual_block) start_block,
                max(individual_block) end_block
        from
                tiled
        group by
                file_id,
                tile
)
select
        *
from
        ranges
order by
        file_id, tile, start_block
;
```


Obviously PL/SQL can be used in the wrong circumstances, but it can be the perfect environment for "enhancing" the data after the SQL has acquired a suitable data set

SQL vs. PL/SQL (b)

```
with extents as (
        select  file_id, block_id, blocks
        from    dba_extents
        where   owner        = upper('&m_owner')
        and     segment_name = upper('&m_segment')
),
expander as (
        select  --+ materialize
                rownum id
        from dual
        connect by
                level <= (select max(blocks) from extents)
),
expanded_blocks as (
        select
                ext.file_id, ext.block_id, ext.blocks,
                ext.block_id + exp.id - 1       individual_block
        from
                extents         ext,
                expander        exp
        where
                exp.id <= ext.blocks
),                            -- e.g. 120 extents x 1,024 blocks
```

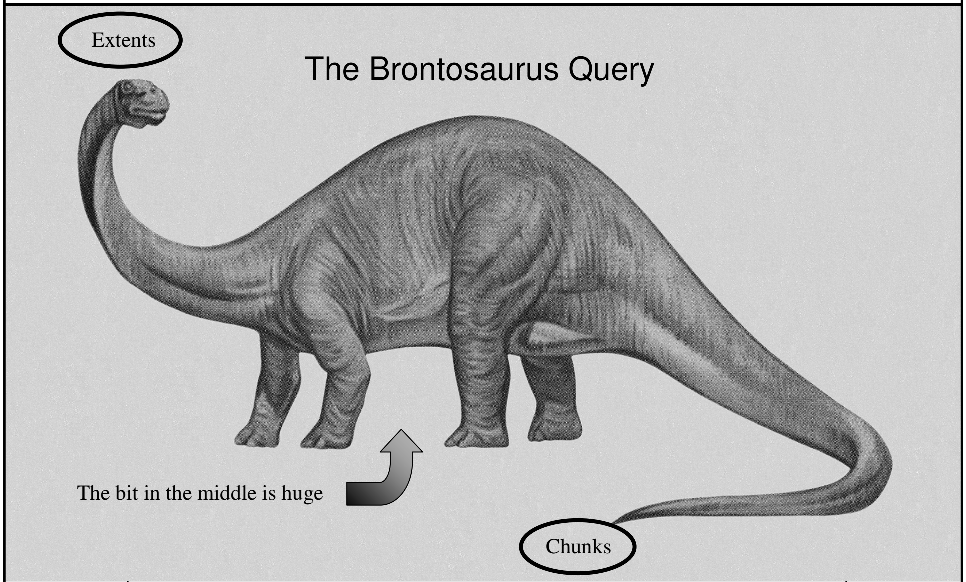

The (relatively) simple SQL solution is not efficient - we start at the scale of extents and expand to the scale of blocks, then contract to the scale of chunks required.

SQL vs. PL/SQL (c)

```
tiled as (
        select
                file_id, block_id, individual_block,
                ntile(&m_tiles) over (order by file_id, individual_block) tile
        from
                expanded_blocks
),
ranges as(
        select
                file_id,
                tile,
                min(individual_block) start_block,
                max(individual_block) end_block
        from
                tiled
        group by                  -- breaks up a chunk that crosses files
                file_id,
                tile
)
select
        {cosmetics for rowid ranges}
from    ranges
order by
        file_id, tile, start_block
;
```


For small objects the code is adequate - but it's new code for ***dbms_parallel_execute*** and you don't (usually) use that package for "small" objects.

SQL vs. PL/SQL (d)



SQL vs. PL/SQL (e)

Extent A	8	8	Chunk 1	51
Extent B	8	8		
Extent C	8	8		
Extent D	32	27		
		5	Chunk 2	51
Extent E	32	32		
Extent F	64	14		
		50	Chunk 3	50
	152			**152**

A picture of what we want gives us a strong hint that we should use a simple SQL statement and then count our way through the result (using PL/SQL)

SQL vs. PL/SQL (f)

The driving query of a PL/SQL loop solution

```
select
        file_id, block_id, blocks,
        sum(blocks) over() tot_blocks
from
        dba_extents
where
        owner        = upper('&m_owner')
and     segment_name = upper('&m_segment')
order by
        file_id, block_id
;
```

For a scalable pure SQL treatment see:
http://stewashton.wordpress.com/category/chunking-tables/

SQL vs. PL/SQL (g)

```
with extents_data as (
  select /*+ qb_name(extents_data) */
    o.data_object_id, e.file_id, e.block_id, e.blocks
  from dba_extents e
  join all_objects o
  on e.owner = o.owner
    and e.segment_name = o.object_name
    and e.segment_type = o.object_type
    and decode(e.partition_name,
            o.subobject_name, 0,
                              1
        ) = 0
  where e.segment_type like 'TABLE%'
    and e.owner = :owner
    and e.segment_name = :table_name
)
, extents_with_sums as (
  select /*+ qb_name(extents_with_sums) */
    sum(blocks) over() as tot_blks,
    sum(blocks) over(
      order by data_object_id, file_id, block_id
    ) - blocks as first_ext_blk,
    sum(blocks) over(
      order by data_object_id, file_id, block_id
    ) as next_ext_blk,
    e.*
  from extents_data e
)
```

```
, filtered_extents as (
  select /*+ qb_name(filtered_extents)*/ * from (
    select
      width_bucket(first_ext_blk-1, 0, tot_blks, :chunks)
        as prev_chunk,
      width_bucket(first_ext_blk, 0, tot_blks, :chunks)
        as first_chunk,
      width_bucket(next_ext_blk-1, 0, tot_blks, :chunks)
        as last_chunk,
      width_bucket(next_ext_blk, 0, tot_blks, :chunks)
        as next_chunk,
      e.*
    from extents_with_sums e
  )
  where prev_chunk < next_chunk
)
, expanded_extents as (
  select /*+ qb_name(expanded_extents) */
    first_chunk + level - 1 as chunk,
    prev_chunk, next_chunk, data_object_id, file_id,
    block_id, tot_blks, first_ext_blk
  from filtered_extents
  connect by first_ext_blk = prior first_ext_blk
    and prior sys_guid() is not null
    and first_chunk + level - 1 <= last_chunk
)
```

This is roughly two-thirds of an SQL statement that produces ***exactly*** the required result, slightly faster than the PL/SQL approach - but it's much harder to understand.

Cartesian Puzzle (a)

Spec: We have a **"big table"** with many **"attribute"** columns,
We have a small **"types"** table with corresponding columns and a **"score"**
For each row in the ***big_table*** find the **best match** from ***types*** table.
All the attribute columns in ***big_table*** are mandatory
At least one attribute in each row of the ***types*** table will be non-null.
There is always at least one partial match.

```
select
        bt.id, bt.v1,
        ty.category,
        ty.relevance
from
        big_table   bt,                 -- 500,000 rows
        types       ty                  --     900 rows
where
        nvl(ty.att1(+), bt.att1) = bt.att1
and     nvl(ty.att2(+), bt.att2) = bt.att2
and     nvl(ty.att3(+), bt.att3) = bt.att3
and     nvl(ty.att4(+), bt.att4) = bt.att4
;
```


The code means we have to compare every row in the big table with every row in the small table - for a total of 450 million intermediate rows "generated")

Cartesian Puzzle (b) - sample data

Big_table

ATT1	ATT2	ATT3	ATT4	ID
1	1	2	1	1
1	3	1	4	2

Types

ATT1	ATT2	ATT3	ATT4	CATEGORY	SCORE
1				XX	10
1			1	YY	20
1		1		ZZ	20

Results

1	1	2	1	1	
1				XX	10
1			1	YY	20

1	3	1	4	2	
1				XX	10
1		1		ZZ	20

big_table id = 1 fails to match the 3rd row of ***types*** because of the mismatch in ***att3***.
big_table id = 2 fails to match the 2nd row of ***types*** because of the mismatch in ***att4***.

Cartesian Puzzle (c)

```
with distinct_data as (
        select  /*+ materialize */
                distinct att1, att2, att3, att4          -- 400 rows!
        from    big_table
)
select  bt.id, bt.v1, ty.category, ty.relevance
from
        distinct_data  dd,  types  ty,  big_table  bt
where
        nvl(ty.att1(+), dd.att1) = dd.att1      -- "expensive" but small
and     nvl(ty.att2(+), dd.att2) = dd.att2      -- 900 types x 400 rows
and     nvl(ty.att3(+), dd.att3) = dd.att3      -- 360,000 tests
and     nvl(ty.att4(+), dd.att4) = dd.att4      -- (400 "best" results)
--
and     bt.att1 = dd.att1                       -- precise big join
and     bt.att2 = dd.att2
and     bt.att3 = dd.att3
and     bt.att4 = dd.att4
;
```


But how many distinct combinations are there in the big table ? Create a result set of the distinct set, do the match with that, then join with an exact match to the big table.

Cartesian Puzzle (d)

```
| Id  | Operation                    | Name                | Rows| Time      |
|   0 | SELECT STATEMENT             |                     | 520K| 00:00:30 |
|   1 |  TEMP TABLE TRANSFORMATION   |                     |     |          |
|   2 |   LOAD AS SELECT             | SYS_TEMP_0FD9D662C |     |          |
|   3 |    HASH UNIQUE               |                     | 400 | 00:00:30 |
|   4 |     TABLE ACCESS FULL        | BIG_TABLE           | 500K| 00:00:01 |
|*  5 |   HASH JOIN                  |                     | 520K| 00:00:01 |
|   6 |    NESTED LOOPS OUTER        |                     | 500 | 00:00:01 |
|   7 |     VIEW                     |                     | 400 | 00:00:01 |
|   8 |      TABLE ACCESS FULL       | SYS_TEMP_0FD9D662C | 400 | 00:00:01 |
|*  9 |     TABLE ACCESS FULL        | TYPES               |   1 | 00:00:01 |
|  10 |    TABLE ACCESS FULL         | BIG_TABLE           | 500K| 00:00:01 |
```

http://jonathanlewis.wordpress.com/2015/04/15/cartesian-join/

Execution time dropped from about 2 hours (almost pure CPU time) to less than 30 seconds.

Intermediates (a)

OTN: "This statement takes 7 hours to run , how do I reduce the time ?"

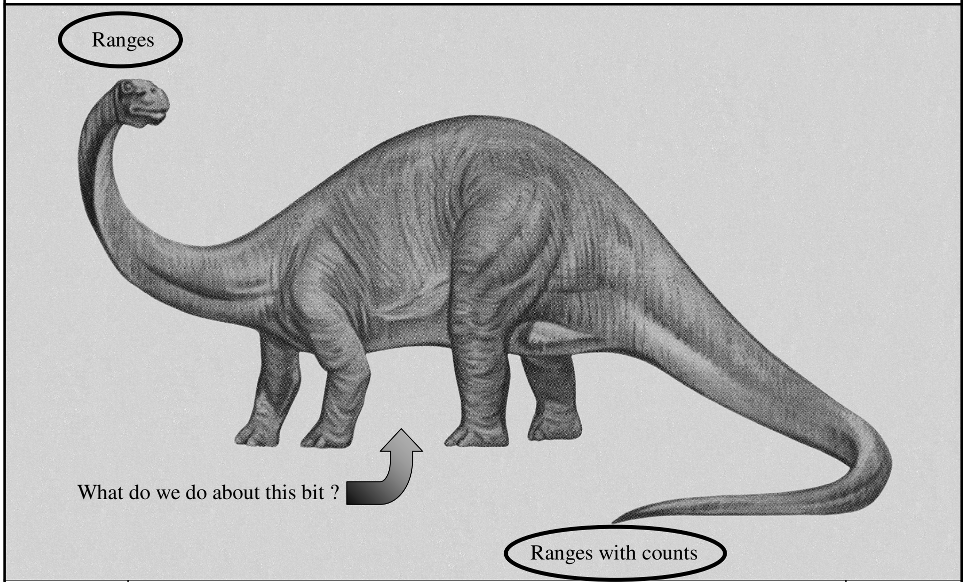
```
    SELECT 'ISRP-734', to_date('&DateTo', 'YYYY-MM-DD'),
           SNE.ID AS HLR
    ,      SNR.FROM_NUMBER||' - '||SNR.TO_NUMBER AS NUMBER_RANGE
    ,      COUNT(M.MSISDN) AS AVAILABLE_MSISDNS              -- 37,650 row result
    FROM
           SA_NUMBER_RANGES SNR                              -- 10,000 rows
    ,      SA_SERVICE_SYSTEMS SSS                            --  1,643 rows
    ,      SA_NETWORK_ELEMENTS SNE                           --    200 rows
    ,      SA_MSISDNS M                                      --    72M rows
    WHERE
           SSS.SEQ = SNR.SRVSYS_SEQ
    AND    SSS.SYSTYP_ID = 'OMC HLR'
    AND    SNE.SEQ = SSS.NE_SEQ
    AND    SNR.ID_TYPE = 'M'
    AND    M.MSISDN  >= SNR.FROM_NUMBER
    AND    M.MSISDN  <= SNR.TO_NUMBER
    AND    M.STATE  = 'AVL'
    GROUP BY
           SNE.ID,
           SNR.FROM_NUMBER||' - '||SNR.TO_NUMBER
;
```


Intermediates (b)

The plan showed a merge join outer between the tables ***sa_number_ranges*** and ***sa_msisdns*** which explodes the data massively before the *group by* contracts it

```
| Id | Operation                    | Name                | Rows | Bytes |TempSpc| Cost (%CPU)|
|  0 | SELECT STATEMENT             |                     |   53M|  3108M|       |   26M   (2)|
|  1 |  HASH GROUP BY               |                     |   53M|  3108M|   164G|   26M   (2)|
|  2 |   MERGE JOIN OUTER           |                     | 2438M|   138G|       |  195K  (15)|
|  3 |    SORT JOIN                 |                     | 1066 | 51168 |       |   21   (15)|
|* 4 |     HASH JOIN                |                     | 1066 | 51168 |       |   20   (10)|
|* 5 |      HASH JOIN               |                     |  328 |  8528 |       |   10   (20)|
|  6 |       TABLE ACCESS FULL      | SA_NETWORK_ELEMENTS |  146 |  1460 |       |    2    (0)|
|* 7 |       VIEW                   | index$_join$_002    |  328 |  5248 |       |    7   (15)|
|* 8 |        HASH JOIN             |                     |      |       |       |            |
|* 9 |         HASH JOIN            |                     |      |       |       |            |
|*10 |          INDEX RANGE SCAN    | SRVSYS_SYSTYP_FK_I  |  328 |  5248 |       |    2    (0)|
|*11 |          INDEX FAST FULL SCAN| E_NE_FK_I           |  328 |  5248 |       |    1    (0)|
| 12 |         INDEX FAST FULL SCAN | SRVSYS_PK           |  328 |  5248 |       |    1    (0)|
|*13 |      TABLE ACCESS FULL       | SA_NUMBER_RANGES    | 2219 | 48818 |       |   10    (0)|
|*14 |    FILTER                    |                     |      |       |       |            |
|*15 |     SORT JOIN                |                     |   13M|   167M|   622M|  169K   (2)|
|*16 |      TABLE ACCESS FULL       | SA_MSISDNS          |   13M|   167M|       |  104K   (2)|
```

The Brontosaurus Query



Intermediates (c)

There is no way around this join explosion if we use the tables as they are (even if we "hide" the join inside a pl/sql function) ***until 12c and pattern recognition***

Design an extract of ***sa_msisdns*** to run as part of this report mechanism.
Give each msisdn a row number (based on sorting the msisdns)
Create a unique index on (msisdn, {ordercolumn})

```
insert /*+ append */ into gtt_msisdns
select
        msisdn,
        row_number() over(order by msisdn)    counter
from
        sa_msisdns
where
        m.state  = 'AVL'
;
```

Costs: one big sort + write to table (less than two minutes for 40M msisdns)

Of course the drawback here is that we don't have a read-consistent result. But is a result that's out of date by 7 hours better than one that's inconsistent by 2 minutes

Intermediates (d)

Drive the query from *sa_number_ranges*, joined twice to the extract.

```
select
        rng.from_number, rng.to_number,
        from1.msisdn, from1.counter,
        to1.msisdn, to1.counter,
        1 + to1.counter - from1.counter range_count
from
        sa_number_ranges      rng,
        gtt_msisdns           from1,
        gtt_msisdns           to1
where
        from1.msisdn = (
                select min(gf.msisdn) from gtt_msisdns gf
                where gf.msisdn >= rng.from_number
        )
and     to1.msisdn = (
                select max(gt.msisdn) from gtt_msisdns gt
                where gt.msisdn <= rng.to_number
        )
;
```


It would be nice if there was a way of adding an index (optionally unique) to a "with subquery" clause, then we would effectively have our read-consistent GTT.

Intermediates (e)

```
| Id  | Operation                         | Name             |
|   0 | SELECT STATEMENT                  |                  |
|   1 |  NESTED LOOPS                     |                  |
|   2 |   NESTED LOOPS                    |                  |
|   3 |    TABLE ACCESS FULL              | SA_NUMBER_RANGES |
|*  4 |    INDEX RANGE SCAN               | GM_I1            |
|   5 |     SORT AGGREGATE                |                  |
|   6 |      FIRST ROW                    |                  |
|*  7 |       INDEX RANGE SCAN (MIN/MAX)  | GM_I1            |
|*  8 |   INDEX RANGE SCAN                | GM_I1            |
|   9 |    SORT AGGREGATE                 |                  |
|  10 |     FIRST ROW                     |                  |
|* 11 |      INDEX RANGE SCAN (MIN/MAX)   | GM_I1            |
```

On a test data set (40M msisdns, 10K number ranges) this query averaged 7 buffer gets per range to "count" the number of MSISDNs in that range

Run time: ca. 0.2 seconds

Intermediates (f) - match_recognize solution

Stew Ashton solutions

New technology (12c) - match_recognize()

Simple case - assume the ranges don't overlap.

```
select * from (
  select  from_number, to_number from number_ranges
  union all
  select  msisdn,      null      from msisdns
)
match_recognize(
  order by from_number, to_number            -- need an ordering
  measures a.from_number from_number,        -- the output columns
           a.to_number to_number,
           count(b.*) range_count
  pattern(a b*)                              -- define "patterns"
  define a as to_number is not null,         -- rules to identify
         b as from_number <= a.to_number     -- a "type" of row
);
```


See also: ***http://stewashton.wordpress.com/2015/12/12/summarize-data-by-range/*** for a solution with overlapping date ranges. Read-consistent, with runtime < 2 mins!

Intermediates (g) - worked example

```
insert into number_ranges values (3, 6);
insert into number_ranges values (8, 13);

insert into msisdns
select 2 * rownum - 1
from dual connect by rownum <= 10;

select * from (
  select from_number, to_number from number_ranges
  union all
  select msisdn, null from msisdns
)
order by from_number, to_number
;
```

FROM_NUMBER	TO_NUMBER
1	
3	**6**
3	
5	
7	
8	**13**
9	
11	
13	
15	
17	
19	

FROM_NUMBER	TO_NUMBER	RANGE_COUNT
3	6	2
8	13	3

With a small sample we can construct the intermediate result to see how Oracle is walking the data to find the pattern.

Intermediates (h)

```
| Id  | Operation                                          | Name          | Rows  |
|   0 | SELECT STATEMENT                                   |               |       |
|   1 |  VIEW                                              |               | 1001K|
|   2 |   MATCH RECOGNIZE SORT DETERMINISTIC FINITE AUTO|               | 1001K|
|   3 |    VIEW                                            |               | 1001K|
|   4 |     UNION-ALL                                      |               |       |
|   5 |      TABLE ACCESS FULL                             | NUMBER_RANGES |  1000 |
|   6 |      TABLE ACCESS FULL                             | MSISDNS       |  1000K|
```

Primary cost: one big sort

10032 trace

```
---- Sort Statistics ------------------------------
Input records                             1001000
Output records                            1001000
Total number of comparisons performed     8157115
  Comparisons performed by in-memory sort 8157115
Total amount of memory used               25400320
Uses version 2 sort
---- End of Sort Statistics -----------------------
```


See also: http://www.slideshare.net/stewashton/row-patternmatching12ctech14/ for a presentation on ***match_recognize()***. "Deterministic finite auto" is the ideal.

Conclusion

- Think technology
- Look for redundant updates
- Use array processing
- Avoid repeating expensive work
- PL/SQL may be better for special cases
- Intermediate tables are not always evil
- Think ***new*** technology
- Find the Brontosaurus