Just Don't Do It
Sins of omission and commission

Jonathan Lewis

jonathanlewis.wordpress.com

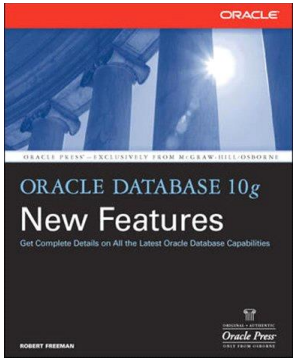
www.jlcomp.demon.co.uk

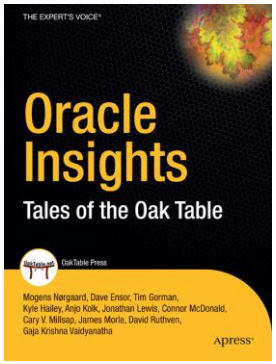
My History

Independent Consultant

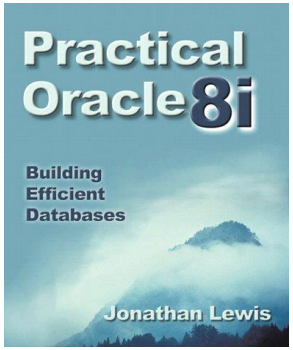
33+ years in IT
28+ using Oracle (5.1a on MSDOS 3.3)

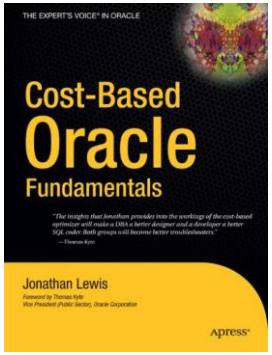
Strategy, Design, Review,
Briefings, Educational,
Trouble-shooting

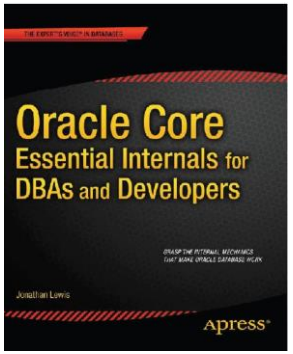




Oracle author of the year 2006
Select Editor's choice 2007
UKOUG Inspiring Presenter 2011
ODTUG 2012 Best Presenter (d/b)
UKOUG Inspiring Presenter 2012
UKOUG Lifetime Award (IPA) 2013
Member of the Oak Table Network
Oracle *ACE Director*
O1 visa for USA







How to spend less time on a job

- Don't do it
  - Do it less often
    - Do it more efficiently

How lazy is Oracle

- Storage Indexes
- Zone Maps

- Scalar Subquery Caching
- Deterministic Functions

- Partition Elimination
- Join Elimination

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 STATE = 0 WHERE FLAG = 'x'
```

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

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/

Superfluous Updates (b)

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


begin
   dbms_stats.gather_table_stats(
        user, 'history',
        method_opt=>
            'for all hidden columns size 1'
--          'for columns sys_nc00019$ size 1'
--          'for columns (case when flag = 'x' and state != 0 then 1 end) size 1'
   );
end;
/
```

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 then create an index on the virtual column. In 12c you can even declare the column invisible.

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 was possible (nulls and partitions make this harder)

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)

```
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.

Addressing the problem (a)

I can view a blog page if

- it belongs to a friend
- or it belongs to a friend of a friend

There is a friendship table

- (my_id, friend_id, ...) -- this is the PK
- If A is a friend of B, then B is a friend of A (by trigger)

I acquire ids by knowing names

- so I have my id, and the blog owner's id

Addressing the problem (b)

Strategy 1: check if it's my friend, then a friend's friend

```
select
        count(*)
from
        friends  fr
where
        fr.my_id     = :b1
and     fr.friend_id = :b2
;
```

```
select
        count(*)
from
        friends  fr1,
        friends  fr2
where
        fr1.my_id = :b1
and     fr2.my_id = fr1.friend_id
and     fr2.friend_id = :b2
;
```


The first statement checks if the two ids correspond to friends. If the count is zero the sql statement checks for "friend of a friend" - with a little PL/SQL for control.

Addressing the problem (c)

Strategy 2: - "don't do it in PL/SQL if it can be done in SQL" (wrong solution)

```
select  count(*) from (
        select  fr.friend_id
        from    friends fr
        where
                fr.my_id     = :b1
        and     fr.friend_id = :b2
        union all
        select  fr2.friend_id
        from    friends fr1,
                friends fr2
        where
                fr1.my_id     = :b1
        and     fr2.my_id     = fr1.friend_id
        and     fr2.friend_id = :b2
   )
;
```


The routine was too expensive, and the first attempt to improve performance used a mantra to fix the wrong problem. This made things worse.

Addressing the problem (d)

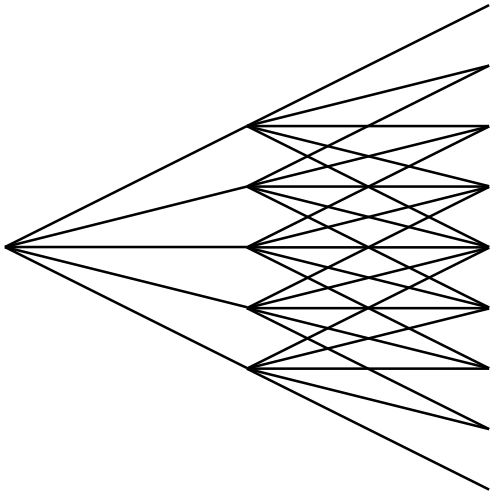
Strategy 3: the *join* is unnecessarily expensive – so ask a different question

```
select   count(*)
from     (
                select   fr.friend_id
                from     friends  fr
                where    fr.my_id     = :b1
                and      fr.friend_id = :b2
                union all
                (                           -- brackets for clarity
                  select fr.friend_id
                  from   friends  fr
                  where  fr.my_id = :b1
                  intersect
                  select fr.friend_id
                  from   friends  fr
                  where  fr.my_id = :b2
                )                           -- brackets for clarity
        )
;
```

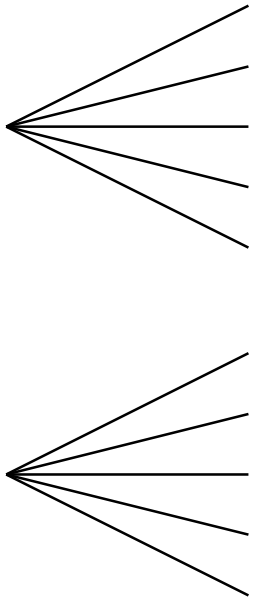

We change the English language from "are you a friend of a friend" - which suggests a join - to "do we have a friend in common" - which suggests an intersection.

Addressing the problem (e)

Join



Intersect



Rowids checked: n * n n + n

Graphically we can see that we have changed an "n-squared" (clearly non-scalable) problem into a "2n" (which means reasonably scaling) problem.

Addressing the problem (f)

```
select  /*+ gather_plan_statistics */
        count(*) from   dual
where   exists (
                select  fr.friend_id
                from    friends         fr
                where   fr.my_id        = :b1
                and     fr.friend_id    = :b2
                union all
                (
                        select  fr.friend_id
                        from    friends fr
                        where   fr.y_id = :b1
                        intersect
                        select  fr.friend_id
                        from    friends fr
                        where   fr.my_id = :b2
                )
        )
;
```


In fact we shouldn't ***count***, we can make things a little more efficient with a subquery that checks for existence - which means it can stop after the first subquery.

"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 others 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..

Cartesian Puzzle (a)

Target: We have a "big table" with many "attribute" columns,
We have a small "types" table with matching attribute columns
For each row in the ***big_table*** find the best possible 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")

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	RELEVANCE
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 (b)

```
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
and     nvl(ty.att3(+), dd.att3) = dd.att3
and     nvl(ty.att4(+), dd.att4) = dd.att4
--
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 (c)

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/

Intermediates (a)

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

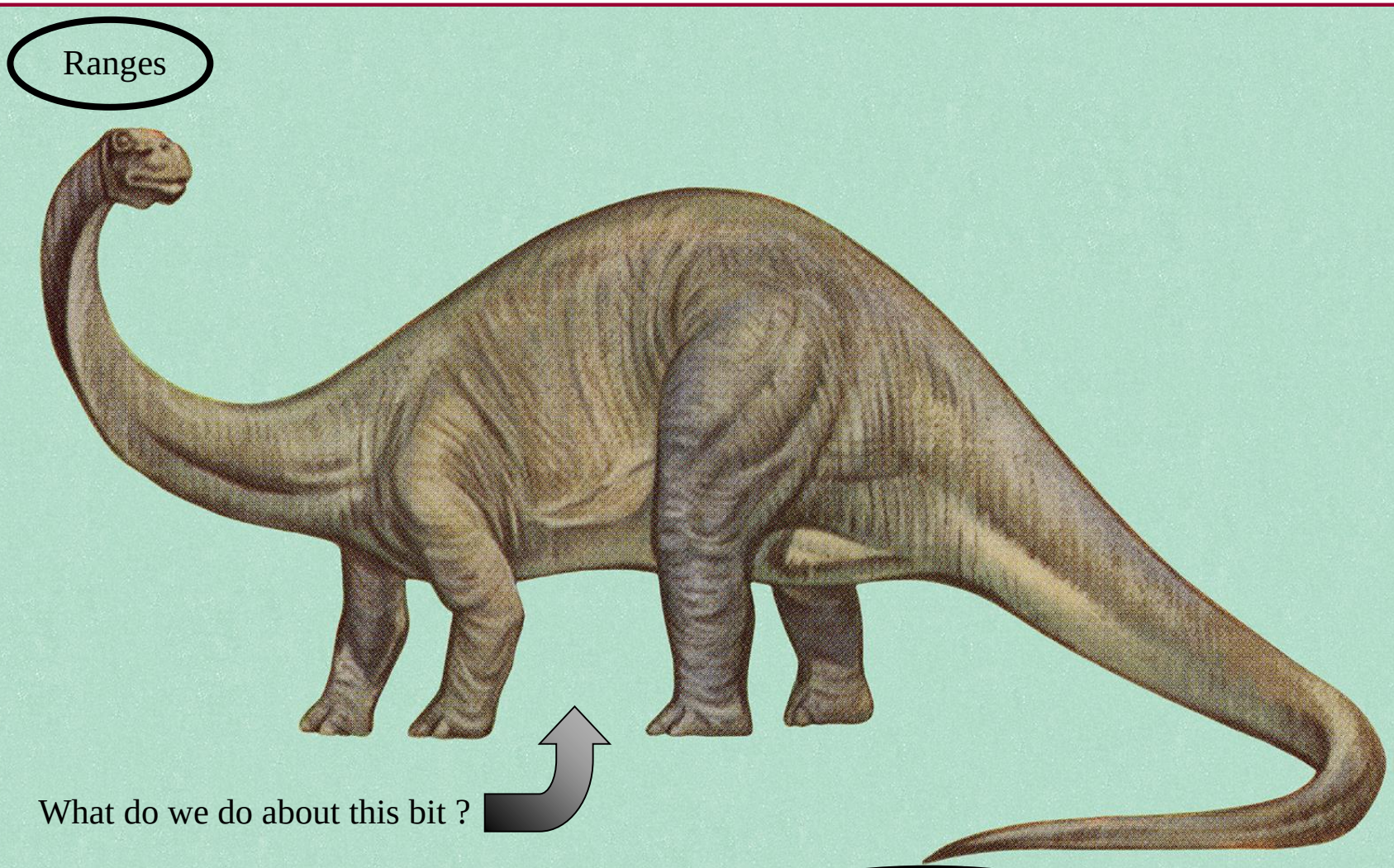
```
SELECT 'ISRP-734', to_date('&DateTo', 'YYYY-MM-DD'),
       SNE.ID AS HLR
,      SNR.FROM_NUMBER||' - '||SNR.TO_NUMBER AS NUMBER_RANGE
,      COUNT(M.MSISDN) AS AVAILABLE_MSISDNS             -- 37,650 row result
FROM
       SA_NUMBER_RANGES SNR                             -- 10,000 rows
,      SA_SERVICE_SYSTEMS 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)

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)

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,         -- how to recognize a type
         b as from_number <= a.to_number
);
```


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 minutes !

Intermediates (g)

```
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 (g)

```
| 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 -----------------------
```

Subquery Abuse (a)

```
select  count(id) cnt,
   case
        when exists(
                select 'x' from geo_apps apps
                where apps.userid = appo.id and apps.PO_ATTRIBUTE1='Y'
        )  then 'One'
        when exists(
                select 'x' from geo_apps apps
                where apps.userid = appo.id and apps.PO_ATTRIBUTE2='Y'
        )  then 'Two'
        ...   Up to PO_ATTRIBUTE20
   end
from
   geo_appo     appo                        -- 15 Million rows.
group by
   case
        when exists(
                select 'x' from geo_apps apps
                where apps.userid = appo.id and apps.PO_ATTRIBUTE1='Y'
        )  then 'One' …
```

Subquery Abuse (b)

Requirement:

Report

the number of rows where PO_ATTRIBUTE1 is the first one set to 'Y'
the number of rows where PO_ATTRIBUTE2 is the first one set to 'Y'
the number of rows where PO_ATTRIBUTE3 is the first one set to 'Y'
...
the number of rows where PO_ATTRIBUTE20 is the first one set to 'Y'
the number of rows where no PO_ATTRIBUTEn is set

Workload

Minimum: 2 * 15M executions of a (3 block) subquery
Worst case: 2 * 300M executions of a (3 block) subquery

Correct Approach

It's ***not necessarily*** a good idea to use scalar subqueries in the select list.
Do a join

The strategy can be good - especially if it makes great use of scalar subquery caching
e.g: http://savvinov.com/2016/04/25/unstable-query-performance-a-case-study/.

Subquery Abuse (c)

```
select  count(*) cnt, first_attribute
From    (
        select
                case
                        when apps.PO_ATTRIBUTE1='Y'     then 'One'
                        when apps.PO_ATTRIBUTE2='Y'     then 'Two'
                        ...
                        when apps.PO_ATTRIBUTE20='Y'    then 'Twenty'
                                                        else 'Null'
                end as first_attribute
        from
                geo_appo  appo, geo_apps  apps
        where
                apps.userid(+) = appo.id
        )
group by
        first_attribute
order by
        count(*) desc
;
```

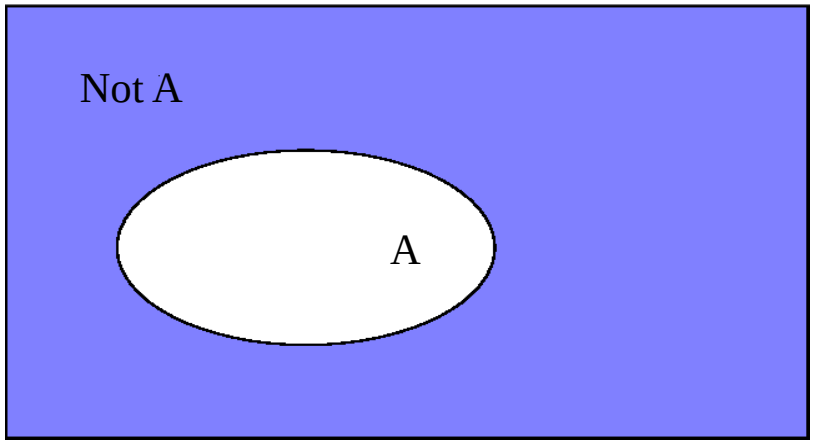

In this case using a join to find the single relevant row with a case statement based on the values in that row would probably be efficient enough.

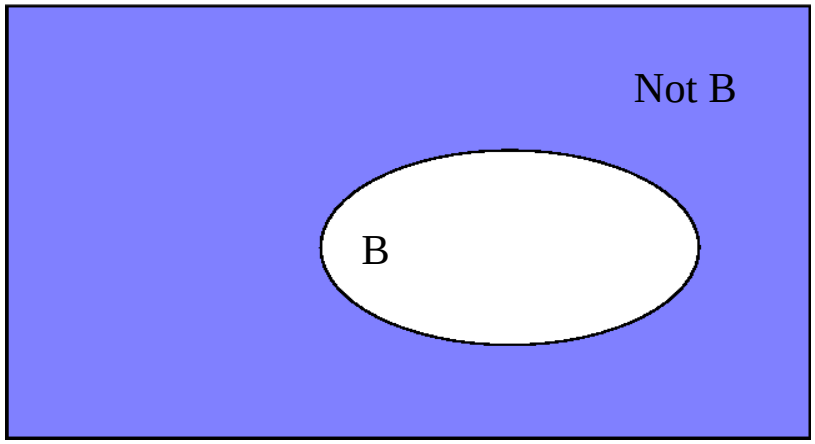
Subquery Elimination (a)

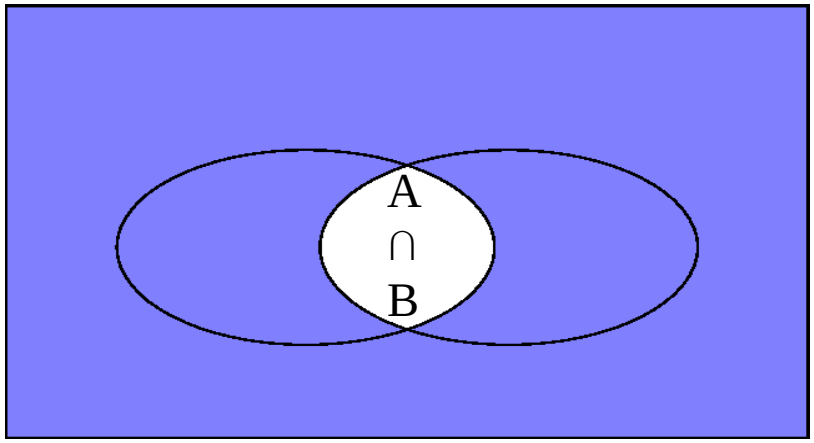
```
select
        a.hotel_code
from    lf_hotel_temp a                         -- 270,000 rows
where
        a.service_id = :p_service_id            -- one of 3 values
and     (
            not exists (
                select  *
                from    lf_ts_roomtype_properties b
                where   b.hotel_code = a.hotel_code
            )
         or not exists (
                select  *
                from    lf_gta_roomtype_properties b
                where   b.hotel_code = a.hotel_code
            )
         or not exists (
                select  *
                from    lf_hb_roomtype_properties b
                where   b.hotel_code = a.hotel_code
            )
        )
```


A first step - which may reduce confusion or introduce insight is to think about re-arranging the predicates using logical equivalence.

Logical Equivalence







not A ***or*** not B == not (A ***and*** B)

not A ***or*** not B ***or*** not C == not (A ***and*** B ***and*** C)

Subquery Elimination (b)

```
select
        a.hotel_code
from    lf_hotel_temp a                     -- 270,000 rows
where
        a.service_id = :p_service_id        -- one of 3 values
and     not (
                exists (
                   select  *
                   from    lf_ts_roomtype_properties b
                   where   b.hotel_code = a.hotel_code
                )
             and exists (
                   select  *
                   from    lf_gta_roomtype_properties b
                   where   b.hotel_code = a.hotel_code
                )
             and exists (
                   select  *
                   from    lf_hb_roomtype_properties b
                   where   b.hotel_code = a.hotel_code
                )
        )
```


The original query is aiming to "find hotels which ***do not have all*** their related data". Start (next slide) with "find hotels where data exists in all three related tables".

Subquery Elimination (c)

Subquery approach - large number of small actions (cp. Nested loop join)
Alternative - brute force, just once (cp. Hash join)

The optimum strategy depends on the data and the indexing.
Brute force: Step 1 - find all hotels which have data in all three related tables

```
select  hotel_code
from    lf_ts_roomtype_properties
where   hotel_code is not null
intersect
select  hotel_code
from    lf_gta_roomtype_properties
where   hotel_code is not null
intersect
select  hotel_code
from    lf_hb_roomtype_properties
where   hotel_code is not null
```

Subquery Elimination (d)

```
select
        a.hotel_code
from    lf_hotel_temp a                         -- 270,000 rows
where   a.service_id = :p_service_id  -- one of 3 values
minus   (
        select  hotel_code
        from    lf_ts_roomtype_properties
        where   hotel_code is not null
        intersect
        select  hotel_code
        from    lf_gta_roomtype_properties
        where   hotel_code is not null
        intersect
        select  hotel_code
        from    lf_hb_roomtype_properties
        where   hotel_code is not null
        )
```


We now subtract the hotels that have all their related data from the list of hotels that we identify from ***lf_hotel_temp***.

Subquery Elimination (e)

Execution plan for the set-based query

```
| Id  | Operation                | Name                          |
|   0 | SELECT STATEMENT         |                               |
|   1 |  MINUS                   |                               |
|   2 |   SORT UNIQUE NOSORT     |                               |
|*  3 |    INDEX FULL SCAN       | LF_HOTEL_TEMP_PK              |
|   4 |   INTERSECTION           |                               |
|   5 |    INTERSECTION          |                               |
|   6 |     SORT UNIQUE          |                               |
|   7 |      TABLE ACCESS FULL   | LF_TS_ROOMTYPE_PROPERTIES     |
|   8 |     SORT UNIQUE          |                               |
|   9 |      TABLE ACCESS FULL   | LF_GTA_ROOMTYPE_PROPERTIES    |
|  10 |    SORT UNIQUE           |                               |
|  11 |     TABLE ACCESS FULL    | LF_HB_ROOMTYPE_PROPERTIES     |
```

With a suitable index the full scan with sort could be an full scan with "nosort"

Subquery Elimination (f)

Possible execution plan for the original query - likely in 12c

```
| Id  | Operation                | Name                          |
|   0 | SELECT STATEMENT         |                               |
|*  1 |  HASH JOIN RIGHT ANTI    |                               |
|   2 |   VIEW                   | VW_SQ_1                       |
|*  3 |    HASH JOIN             |                               |
|*  4 |     HASH JOIN            |                               |
|   5 |      TABLE ACCESS FULL   | LF_GTA_ROOMTYPE_PROPERTIES    |
|   6 |      TABLE ACCESS FULL   | LF_HB_ROOMTYPE_PROPERTIES     |
|   7 |     TABLE ACCESS FULL    | LF_TS_ROOMTYPE_PROPERTIES     |
|*  8 |    INDEX FULL SCAN       | LF_HOTEL_TEMP_PK              |
```


I was impressed to find the optimizer ***can*** do the same OR -> AND conversion - and then converts the three subqueries into a single join subquery and unnests to anti-join

Conclusion

- Think technology
- Look for redundant updates
- Use array processing
- Review the requirement
- You can visit a table more than once
- Temporary tables are not evil
- Where's the Brontosaurus
- Rethink subqueries