High Speed PLSQL

- No Secrets, No Shortcuts

- Avoiding the worst practices.

- By Rodger Lepinsky
- North East Oracle User's Group
- http://www.noug.com
- Bentley College, Waltham, Massachusetts
- March 17, 2004

Who is Rodger Lepinsky?

- Oracle DBA who has worked with databases since 1992, and with Oracle since 1995. Versions 7 to 9i.

- Design
- Development
- Warehousing
- Tuning
- Database Administration

High Speed PLSQL

- Did you ever hear of the execution of the programmer? :-)

High Speed PLSQL

- The moral of the story:
- When we extrapolate the logic, we don't always get the intended result.
- What some people do for "performance", actually slows things down.

Synopsis

- There is much talk about tuning Oracle lately. Most of this focuses on DBA techniques, often done after the code has been put into production.

- These are very useful. However, in my experience, the most dramatic increases in speed have come from rewriting the SQL, and/or changing the data model.

- Ie. From hours, to minutes.

Synopsis

- As a DBA, I've frequently used statspack, or utlbstat/utlestat to find the source of the longest waits. (see: www.oraperf.com)
- Almost always, the number one wait is db file sequential / scattered read. The recommendation is to tune the SQL statements.
- (As differentiated from latches, redo log switches, library cache parsing, control files, extent allocations, etc.)

Output from Oraperf.com

Event	Time	Percentage
db file scattered read	8195	70.40%
db file sequential read	1743	14.97%
SQL*Net message from dblink	843	7.24%
latch free	249	2.14%
write complete waits	203	1.74%
SQL*Net more data from dblink	165	1.42%
...		

Output from Oraperf.com

Advice

The advice is given in the order of the most impact of the total response time. The percentage gain is taken of the response time.

Maximum Gain (%)	What	Detail
0	Check SQL*Net Configuration for SDU and TDU settings	Also check the usage of array fetch. Also check the usage of array inserts.
1	Tune the remote database that is accessed through dblinks	Tuning the remote database will help to improve the response time on this node.
82	Reduce the number of buffer gets	Check your SQL statements and try to optimize them.

Developer Tuning Techniques.

- You don't need to be a DBA to write fast queries in Oracle. Some techniques available to developers are:

- Efficient SQL coding
- Good Relational Design, using Primary Keys and Foreign Keys
- Set timing on, set feedback on, (not tuning per se, but does give metrics to start with) set autotrace on

More Developer Tuning techniques

- Explain plan
- SQL Trace / tkprof
- Indexes
- Hints
- Data Dictionary queries (if you have the rights to read the views)

Developer Tuning Techniques

- Of all these techniques, this presentation will focus primarily on:

- Coding techniques for SQL and PLSQL.

Developer Tuning Techniques

- This is not Intro to PLSQL.
- It assumes that you already know PLSQL.

- Little in here will be new to experienced Oracle people.

- What may be new is how I have put everything together.

Developer Tuning Techniques

- There are no secrets.
- There are also no shortcuts.
- Writing fast code does require some thought.
- This is differentiated from just charging ahead coding.

Principle Number One
Hard Drives Are Slow

- Between hard drives, CPU, and memory, the hard drives are the slowest component of any computer system!

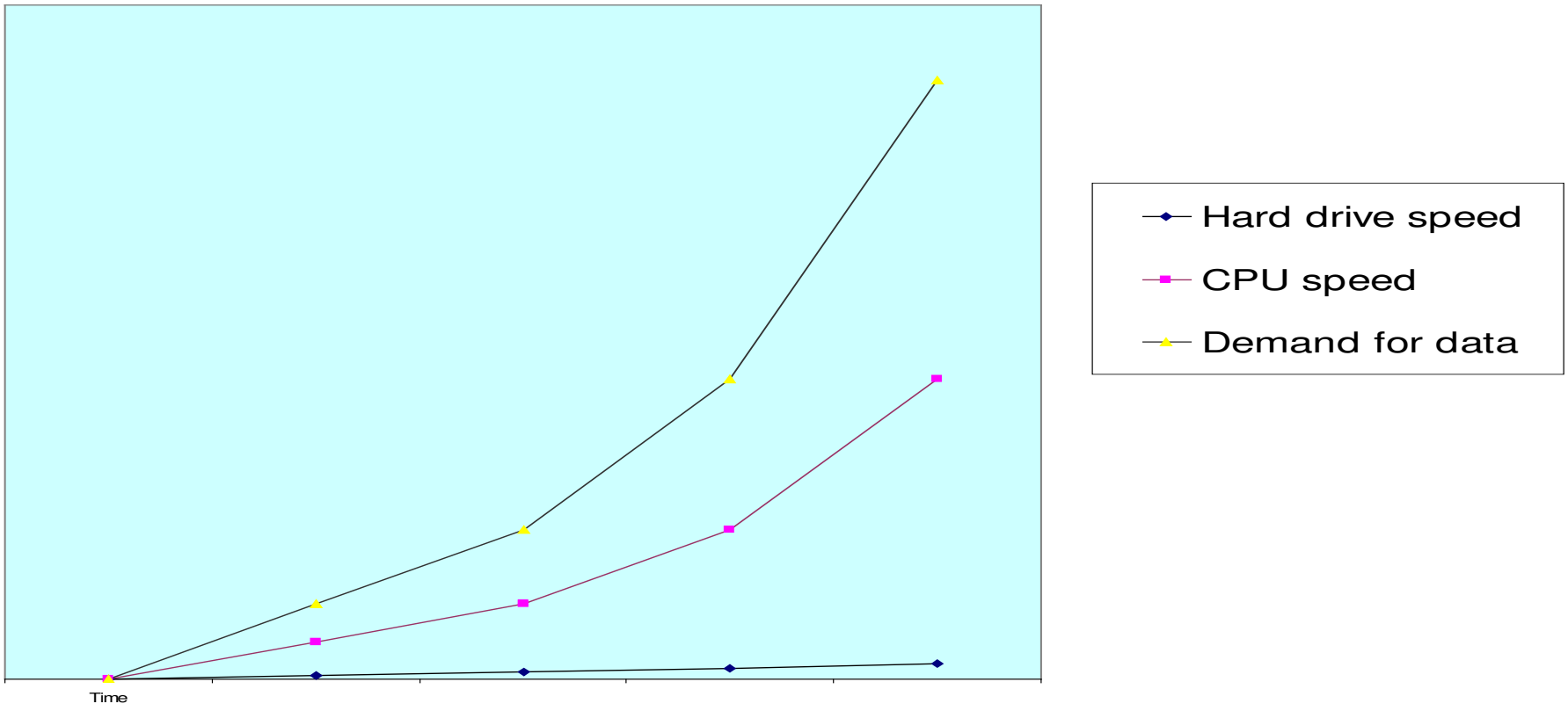
- They are thousands of times slower than CPU or memory.

- This is true whether you have worked on COBOL on a mainframe, Foxpro on LAN, or distributed Oracle system on Unix.

Speed of Hard Drives

- In 2001, I was researching on how to insert one billion rows a day (11,000 per second) into Oracle.
- I came across a chart that showed the increases in speed between hard drives, CPU, and the demand for data.
- I haven't been able to find the slide, but the general idea looked like the following:

Speed Increases
CPU vs. Hard Drives



Speed of Hard Drives

- While CPU speeds are going up exponentially, hard drive speeds are only going up in a more linear fashion.
- And, the demand for data was going up even faster than the speed increases in CPU.

- Consider 5 years ago, 1999.
- Typical Intel PC was 350 mhz. Now: 2.8 Ghz.
- Hard drives: 5K RPM. Now: 7.5K to 15K RPM

Speed of Hard Drives

- Compared to the solid state components in a PC, hard disks have by far the worst performance. And even as hard disks improve in speed, CPUs, video cards and motherboards improve in speed even faster, widening the gap. Thus, hard disks continue to constrain the overall performance of many systems.
- In the amount of time it takes to perform one random hard disk access, one of today's CPUs can execute over a million instructions! Making the CPU fast enough to process *two* million instructions while it waits doesn't really gain you much unless it has something to do with that time.
- Source: StorageReview.com

Principle Number One
Hard Drives Are Slow

- Yet, so many developers worry about trimming lines from their source code, but never reduce reads to the hard drives.

- At best, this can only give marginal increases in speed.

- When tuning SQL or PLSQL, always try to reduce the number of reads to the hard drive.

Speed of Hard Drives

- Significance:
- Oracle is a RDBMS.
- It stores and manages data on hard drives; the slowest part of any computer system.
- Even though it is a great tool, if used incorrectly, Oracle -can- be slow.

- I know. I've had to maintain, or speed up a number of slow, poorly developed, systems.

Lessons - My Uniface Experience

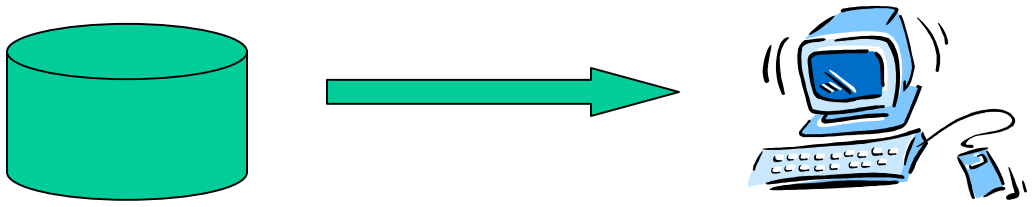
- Uniface was a client server product that I used in the nineties.

- No SQL language support..
- No GROUP BY or DISTINCT clause.

- First assignment, GROUP BY report in the native Uniface language.
- A tough problem.

Lessons - My Uniface Experience

- As I used the product, it became apparent that it was actually dragging ALL the data from Oracle, in to the client!
- The clients were running Windows 3.1, that had only 16 megs of RAM!



Lessons - My Uniface Experience

- When it dragged information into the client, it didn't just drag in the result set.

- Say there was one million detail rows, that would GROUP BY to one thousand rows. It would actually drag all one million rows into the client! The "group by" would then need to be coded with loops of sums and deletes.

Lessons - My Uniface Experience

- Estimated finish time:
- 10 hours.
- If it finished at all.

Lessons - My Uniface Experience

- How to fix?

Lessons - My Uniface Experience

- How to fix?
- Read lots of documentation on Uniface, and Oracle.

Lessons - My Uniface Experience

- How to fix?
- Read lots of documentation on Uniface, and Oracle.
- Create an entity in Uniface. Dynamically create views in Oracle with a user defined Where clause.

Lessons - My Uniface Experience

- How to fix?
- Read lots of documentation on Uniface, and Oracle.
- Create an entity in Uniface. Dynamically create views in Oracle with a user defined Where clause.
- Then just use a simple Uniface retrieve.

Lessons - My Uniface Experience

- End result.

- Report finished running in 5 minutes.
- Approximately 120 times faster.
- (Assuming that the original report would have finished.)

- Using slow software and equipment, I still had to make things work fast.

PRINCIPLE NUMBER TWO

- Conclusion:
- The database server is faster than the client.
- It has more memory, and more powerful CPUs.
- There is no network traffic between the client and server to slow things down.

PRINCIPLE NUMBER TWO

- For performance, put the heavy work onto the database server.
- Avoid dragging lots of data into the client and doing the processing there.
- This principle still applies to java.
- Object layer population times.

PRINCIPLE NUMBER TWO

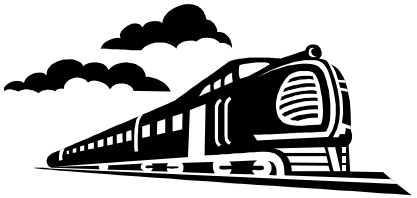
- Note: One area where you -should- use the client CPU power is in painting the client's GUI.

- Don't use X windows for the GUI.
- On one system, in addition to the server doing the processing, it painted the screen of EVERY client using the system.

- With less than 10 people on the Dev system, bringing up a new screen took 5 - 10 seconds!

Principle Three:
Read the Data Once

- Avoid nested loops retrieving small amounts of data.
- The faster method is to read the data en masse.
- Retrieve everything in one big SQL query or cursor.



Coding Using Temp Tables

- Temp tables have been very useful for me.
- Often used for reporting.
- But can also be used for batch processing, and data input.

Coding Using Temp Tables

- Reduces up to millions of rows into small manageable sets, say thousands of rows.

- Now, it's not unusual to have Materialized views, which can do much the same thing.

Coding Using Temp Tables

- The task: a report on four different time periods.
- Co-worker's 128 decode statements in one query. Aborted after an hour.
- Use Temp tables.

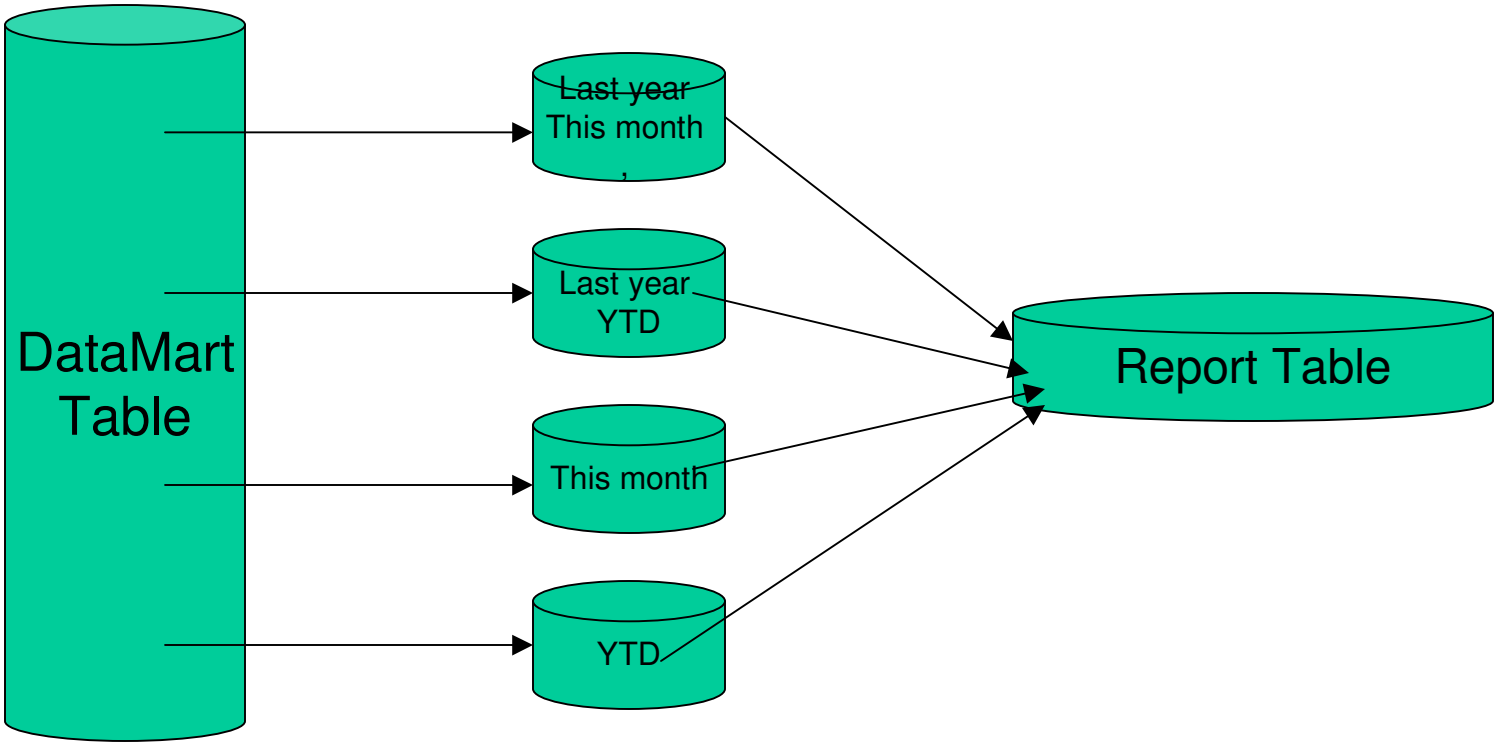
Coding Using Temp Tables

```
describe mart

 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 YEAR                                      NOT NULL NUMBER(4)
 MONTH                                     NOT NULL NUMBER(2)
 SUPPLIER_NO                               NOT NULL NUMBER(1)
 CUST_NO                                   NOT NULL NUMBER(2)
 SOURCE_NO                                 NOT NULL NUMBER(2)
 DEST_NO                                   NOT NULL NUMBER(2)
 CASH                                               NUMBER(6)
 CREDIT                                             NUMBER(6)
 TOTAL                                              NUMBER(7)

INDEX_NAME                     COLUMN_NAME           COLUMN_POSITION
------------------------------ --------------------- ---------------
PK_MART                        YEAR                             1
                               MONTH                            2
                               SUPPLIER_NO                      3
                               CUST_NO                          4
                               SOURCE_NO                        5
                               DEST_NO                          6
```

Coding Using Temp Tables

```
describe report_table
 Name                                                  Null?    Type
 ----------------------------------------- -------- ----------------------------
 YEAR                                      NOT NULL NUMBER(4)
 SUPPLIER_NO                               NOT NULL NUMBER(1)
 CUST_NO                                   NOT NULL NUMBER(2)
 THIS_MONTH_CASH                                    NUMBER(20)
 THIS_MONTH_CREDIT                                  NUMBER(20)
 THIS_MONTH_TOTAL                                   NUMBER(20)
 YTD_CASH                                           NUMBER(20)
 YTD_CREDIT                                         NUMBER(20)
 YTD_TOTAL                                          NUMBER(20)
 LAST_YEAR_THIS_MONTH_CREDIT                        NUMBER(20)
 LAST_YEAR_THIS_MONTH_CASH                          NUMBER(20)
 LAST_YEAR_THIS_MONTH_TOTAL                         NUMBER(20)
 LAST_YTD_CREDIT                                    NUMBER(20)
 LAST_YTD_CASH                                      NUMBER(20)
 LAST_YTD_TOTAL                                     NUMBER(20)

PRIMARY KEY:        YEAR
                    SUPPLIER_NO
                    CUST_NO
```

Temp Tables - Method 1



Temp tables

Populating Temp Tables - Method 1

- Method 1: Five temp tables.
- Four temp tables have the set of data for each time period (This_month, YTD, ...).
- One final summary table with all the sets of data.

Populating Temp Tables - Method 1

- Insert as select, . four times.

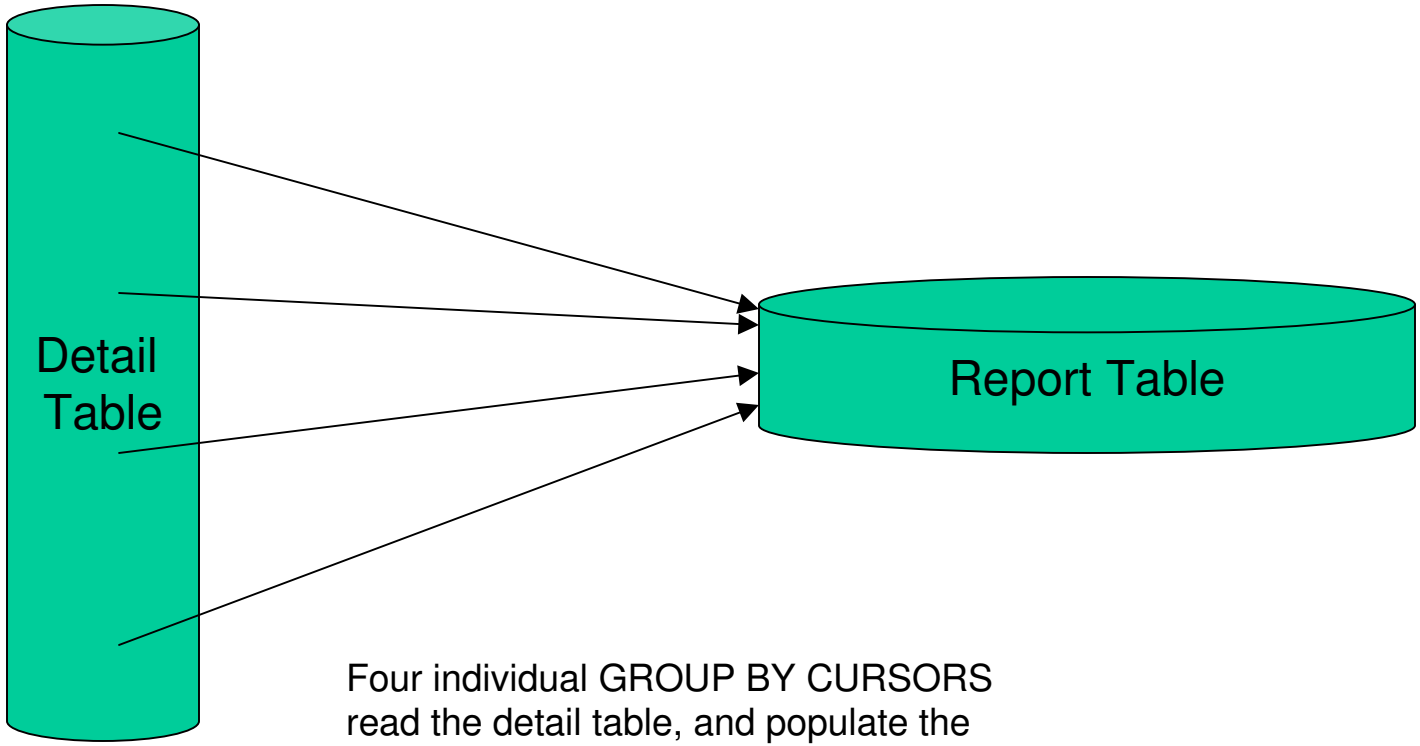
- INSERT into temp_YTD (F1, F2 ...)
- (Select ...
- FROM DETAIL_TABLE
- WHERE date_range = YTD
- Group by)

Populating Temp Tables - Method 1

- Then, INSERT into final temp table, that had all the date ranges. Join all 4 temp tables to insert.

- After populated, retrieve from the 4 tables.
- Response time: about 3 minutes.

Temp Tables - Method 2



Four individual GROUP BY CURSORS read the detail table, and populate the summary table.

Populating Temp Tables - Method 2

- Very similar to method 1.
- But, no intermediate tables.
- Only one destination table.

- Instead of four INSERT statements,
- Use four GROUP BY cursors to insert into the summary table.

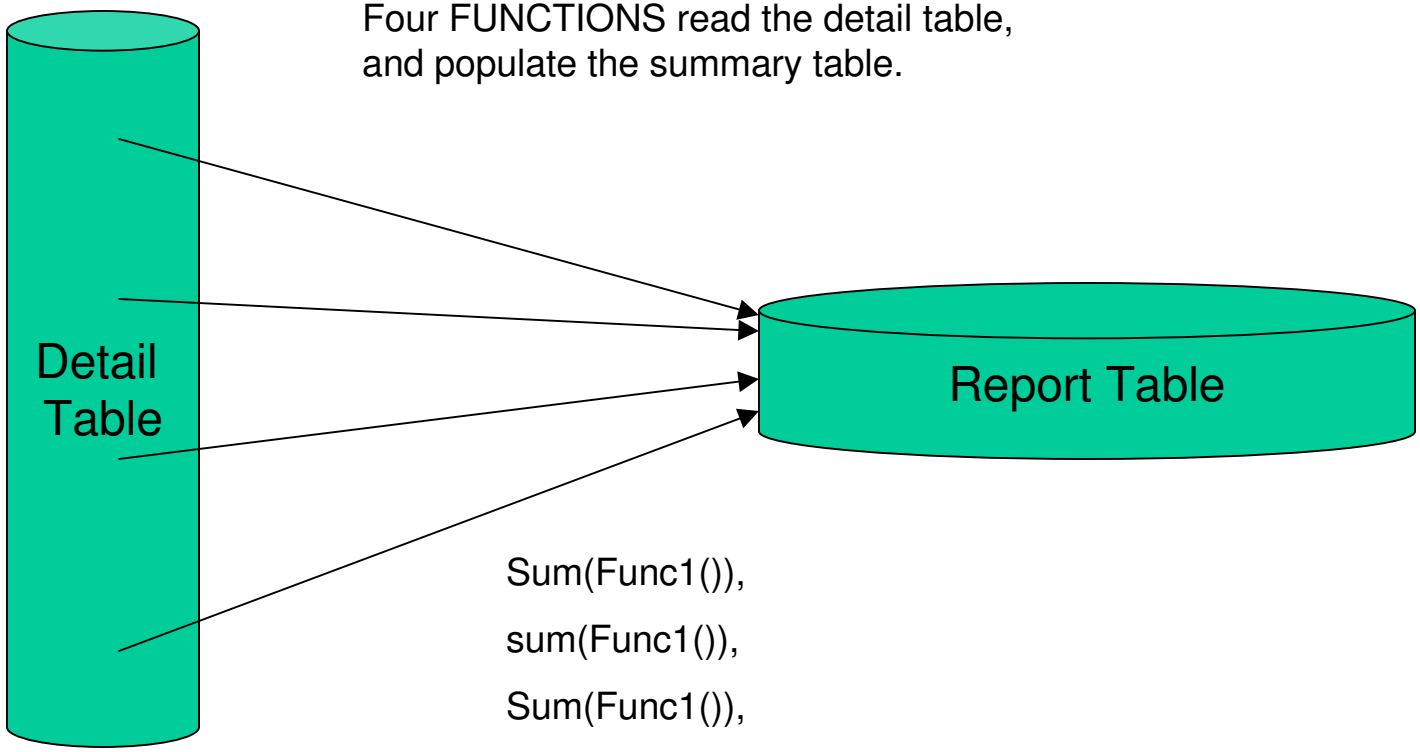
Populating Temp Tables - Method 2

- For c1 in This_YTD loop
  - insert into Temp_table (c1.field1, c1.field2 ..)
  - if dupe key, update appropriate columns
- end loop

- For c2 in This_month loop ...
- For c3 in Last_year_YTD loop ...
- For c4 in Last_year_This_Month loop ...

- Result: About 2 minutes.

Temp Tables - Method 3



Populating Temp Tables - Method 3

- It was an interesting concept my coworker had: do everything in one cursor. I decided to try similar, but with fewer functions.
- I made a PLSQL function that would take as parameters:
- a reference date
- an actual date
- numeric fields
- and return the data if it was in the date range.

Populating Temp Tables - Method 3

```
Create or Replace Function Func1
(p_ref_date ... , p_actual_date ..., p_numeric_info ...)
return number
(
if (p_ref_date = p_actual_date) then
        return p_numeric_info;
else
        return 0;
end if;
);
```

Populating Temp Tables - Method 3

```
Cursor          Pop_Report_Table  is
Select          tab1.report_key
sum(func1(this_month, tab1.datefield, tab1.data)),  /* this month */
sum(func1(this_month, tab1.datefield, tab1.data)),   /* this year */
/*  etc. ...  */
FROM            Detail_table      tab1
WHERE           Tab1.datefield in range
GROUP BY        Tab1. report_key
```

Populating Temp Tables - Method 3

```
For     C1     IN   Pop_Report_Table   Loop
          Insert into report_table
                    (Year, month, ...)
          Values
                    (c1.field1, ...)
End loop;
```

Populating Temp Tables - Method 3

- Response time was between 12 and 13 minutes.
- About 4 to 6 times longer than using temp tables.
- But much faster than using 128 decode statements in the single cursor.

- Conclusions:
- 1) Doing -all- the work in functions is slower than temp tables.
- 2) But, the fewer functions that you use, the faster the response time.

TEMP Tables: Conclusion

- The only independent variable in these examples, is how I coded. Everything else was the same.

- Max time: 12 minutes. (Co-worker's did not finish.)
- Min time: 2 seconds.

- How you code, can make a big difference in response times.

Set Theory

- Think in terms of SETS of information.
- Don' t think in terms of row by row processing. That is like COBOL. (Claim you are too young for Cobol.)

- Any query returns a set of rows. Zero, one, or more.

- An RDBMS is based on the set theory that you learned in junior high school.
- Remember: UNION, INTERSECT, MINUS ???

Basic Hints - Joins

- Everyone knows Inner Join (AKA Equi-join)
- Less popular, but very useful joins are:

- UNION
- INTERSECT
- MINUS
- OUTER JOIN

- These joins are both very efficient and very useful.

Basic Hints - Joins

- IN, and NOT IN
- EXISTS, and NOT EXISTS

- These can be fast, or slow, depending on the query.
- My rule of thumb for IN: If there are only a few values to compare to, say 5, don't worry about the performance hit.
- But, if there are many, say 1000, then I will avoid the IN clause, and try to use an Outer Join or another method.

Basic Hints - Outer Joins

- Outer joins are very efficient, and very useful.
- I've had great success with outer joins.

- An outer join has a built in implicit IF statement.

- If in this set, but NOT in this set. (IS NULL)
- or
- If in this set, AND this set. (IS NOT NULL)

Basic Hints - Outer Joins

Rewriting a query from IN to use an outer join. Using IN:

```
Select    Deptno
from      Dept
where     Deptno not IN
          (select deptno from emp)

Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=4 Card=1 Bytes=3)
   1    0   FILTER
   2    1     TABLE ACCESS (FULL) OF 'DEPT' (Cost=2 Card=1 Bytes=3)
   3    1     TABLE ACCESS (FULL) OF 'EMP' (Cost=2 Card=1 Bytes=3)

Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
         16  consistent gets  ...

Proof of concept from:  Tom Kyte.  Effective Oracle By Design
```

Basic Hints - Outer Joins

Using an outer join:

```
Select    Dept.Deptno
from      Dept,
          Emp
Where     Dept.Deptno = Emp.Deptno (+)
And       Emp.Deptno is null

Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=5 Card=14 Bytes=84)
   1    0   FILTER
   2    1     HASH JOIN (OUTER)
   3    2       TABLE ACCESS (FULL) OF 'DEPT' (Cost=2 Card=4 Bytes=12)
   4    2       TABLE ACCESS (FULL) OF 'EMP' (Cost=2 Card=14 Bytes=42)

Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          6  consistent gets  ...

Proof of concept from:  Tom Kyte.  Effective Oracle by Design.
```

Basic Hints - Explain Plan

- EXPLAIN PLAN is very useful.
- Naturally, look for full table scans.
- Then look where you can add indexes on the full table scans.

Basic Hints - Explain Plan

- But!

- Very Important: Don’t assume that a full table scan will be slow.
- Full table scans can actually be faster than using indexes. (Example to follow.)

- Don’t speculate.
- Try both and Test!

Basic Hints - Explain Plan

- One day's processing, was taking about 20 hours.
- By looking at the data dictionary during processing, I was able to figure out which query was running most of the time.
- I recommended some indexes to be created on the fields involved in full table scans.

Basic Hints - Explain Plan

- By adding indexes, the queries sped up significantly.
- Result: the processing finished in about 3 hours.
- About 7 times faster.

Explain Plan - Total Rows Returned

- One aspect that is not talked about much with Explain Plan is the total number of rows returned in the Plan.

- An Explain Plan with many rows returned, is usually much slower than one with just a few rows returned.

- An Explain Plan doing full table scans, but with fewer rows in the Plan, can actually be much faster than the inverse.

Explain Plan - Total Rows Returned

- Example: One co-worker's query was running for about 15 minutes, and then not finishing. It would run out of rollback room.
- It was joining 4 indexed tables, and had complex search patterns.

- I ran Explain Plan. It returned 81 rows!
- So, for every row, or set of rows returned, Oracle had to do 81 operations!

Explain Plan - Total Rows Returned

- Solution: Count the rows in each of the four tables in the query.
- Note the table that had the most operations being done to it. In this case, it was the largest table.
- Add a hint to this largest table to do a full table scan.

Explain Plan - Total Rows Returned

- Result: Explain plan then showed only about 25 rows returned. About 1/3rd of the original number of rows.
- The query finished in about 1 minute, 12 seconds.
- At least a dozen times faster (assuming that the first query would have finished in 15 minutes).

Explain Plan - Total Rows Returned

- Conclusion:

- 1) The more rows returned in Explain Plan, the query will probably be slower, and more resources required. (However: test it.)

- 2) Full table scans can sometimes be faster than using indexes.

Basic Hints - Functions

User defined PLSQL functions can be used in the SELECT and WHERE clauses.

```
SELECT      Field1,  MY_FUNC1(field1)
FROM        TABLE_1
WHERE       Field2 = MY_FUNC1(field1)
```

Basic Hints - Functions

- User defined functions are not particularly efficient.
- However, user defined functions in SELECT or WHERE clause can be very useful with poorly designed data models, requiring complex logic to make it work.

Basic Hints - Functions

- Example: Is the order completed?

- Actually was not designed into the system. The answer was found indirectly in a number of tables.
- If the data was found in some tables, but not others, then, incomplete. Else complete.
- The Project Manager actually had us looking at the data manually! Row by row!

Basic Hints - Functions

- IF statements were required for the answer.
- But how to code that into a single query?
- I wrote a function to determine whether or not work was complete. And used it as such:

- Select Cust_ID, F1(cust_ID)
- FROM Customer_table
- Where F1(Cust_ID) = 'COMPLETE'

Basic Hints - Functions

- Result: for about 30,000 rows, it finished in about 3 minutes.

- Not particularly efficient. About 166 rows per second.
- But much more efficient than a number of other complex procedures that were being tried.

- The main benefit was that it freed up the people time, which was -much- more expensive that the CPU time.

Basic Hints - Analytic Functions

- Oracle now has a lot of analytic functions, that can eliminate lots of coding.
- What previously had to be coded in the program, can now be done right in the query.
- No looping required.

Basic Hints - Analytic Functions

```
Select    deptno,   ename,    sal,
          sum(sal) over (partition by deptno order by sal)  cum_dept_tot,
          sum(sal) over (partition by deptno)  tot_dept_sal,
          sum(sal) over (order by deptno, sal)  cum_tot,
          sum(sal) over ()  co_tot
from      emp
order by  deptno, sal
```

DEPTNO	ENAME	SAL	CUM_DEPT_TOT	TOT_DEPT_SAL	CUM_TOT	CO_TOT
10	MILLER	1300	1300	8750	1300	29025
10	CLARK	2450	3750	8750	3750	29025
10	KING	5000	8750	8750	8750	29025
20	SMITH	800	800	10875	9550	29025
20	ADAMS	1100	1900	10875	10650	29025
20	JONES	2975	4875	10875	13625	29025
20	SCOTT	3000	10875	10875	19625	29025
20	FORD	3000	10875	10875	19625	29025
30	JAMES	950	950	9400	20575	29025
30	WARD	1250	3450	9400	23075	29025
30	MARTIN	1250	3450	9400	23075	29025
30	TURNER	1500	4950	9400	24575	29025
30	ALLEN	1600	6550	9400	26175	29025
30	BLAKE	2850	9400	9400	29025	29025

Source of example: Tom Kyte Effective Oracle by Design.

Use Metrics

- Always be asking the question: “how much?”
- Try to answer with actual metrics.
- How long does this technique take?
- How much faster (or slower) is it?
- Don't speculate. Test it.

Use Metrics

- Ie. Some still stick to an explicit cursor because “it is faster” than an implicit cursor (FOR loop).
- Perhaps true. But I ask: just how much faster is it?
- 10 Seconds? Or a micro second?

- SET TIMING ON. Test it.
- If there is no measurable difference, I tend to write the code so that it is easier to read, saving people and maintenance time.

Stay Close to the Table

Try to avoid creating objects many layers deep. A view is built on top of the view, etc. This is not unusual in DWH environments where rights are given only to views.

Table

View (Select * from tab_A)

View (where field1 = x)

View (and field2 = y)

Stay Close to the Table

Instead, try to stay close to the table:

Table

View (Select N from tab_A)

View (Select N from tab_A
where field1 = x)

View (Select N from tab_A
where field1 = x
and field2 = y)

Other benefit: much easier to maintain and debug.

Avoid the Loop De Loop

- Instead of opening up a number of nested loops, try to retrieve all the data in one big query.
- Outer joins can be very useful here.
- You can also use UNION, MINUS, INTERSECT.

Avoid the Loop De Loop

At times, I've seen nested cursors over database links, or between tables:

```
For C1 in          Data_From_DBLink_1   Loop
    For C2 in      Data_From_DBLink_2   Loop
              If (something) then
                   If (something_else) then
                              some_processing();
                   End if;
              End if;
    End loop;      /* C2 */
End Loop:          /*  C1 */
```

Avoid the Loop De Loop

- Such a method reads one row from one remote database. Then, it reads another row from a second remote database.

- For each row retrieved, there is both database, and network overhead.

Avoid the Loop De Loop

Try an Outer Join instead:

```
Cursor      All_The_Data is
Select      Tab1.PK,
            Tab2.field1,   ...
From        Table1@DB_LINK1        Tab1,
            Table2@DB_LINK2        Tab2
WHERE       Tab1.PK = Tab2.PK (+)
And         ...
```

Avoid the Loop De Loop

- The outer join will use the SGA to do the joins.
- It eliminates hard coding how the join is being handled.
  - Inner loop.
  - Often, if statements.

Avoid the Loop De Loop

- The other benefit: the body of the code is much smaller and much easier to read.

- For C1 in All_The_Data Loop
  - Insert
  - Update
  - etc.
- End Loop;

Avoid the Loop De Loop

If the task is an insert, you can sometimes eliminate the loop altogether with an INSERT as SELECT ...

```
Insert    into      Tab3

(field1, field2 ...)

(Select             Tab1.PK,

                    Tab2.field1,                ...

From                Table1@DB_LINK1             Tab1,

                    Table2@DB_LINK2             Tab2

WHERE               Tab1.PK = Tab2.PK (+)       ...   )
```

Avoid the Loop De Loop

- Example:
- In one environment, an existing procedure used nested loops to read two different databases. It finished in 30-40 minutes on average.

- I wrote a companion procedure, using the outer join. It typically finished in 1.5 to 2 minutes.

- About 20 times faster.

Avoid Loops By Using Calculations

- Between the CPU and the hard drives, use the CPU.
- One system that I was maintaining, was constantly working on one particular SQL statement. This I noticed by checking the active statements in the data dictionary during processing.
- The statement was in a WHERE clause:
- Business_Days_Between (:day1, :day2)) = ???

Avoid Loops By Using Calculations

- Looking at the code, to begin with, it had deep dependencies, view from view ...
- It also had a lot of loop de loops.
- There was a Holiday table, that had the list of holiday days (no Saturday, or Sunday dates).

Avoid Loops By Using Calculations

The pseudo-code for the function looked something like:

Count the days between the two dates.

While J <= V_days_between loop

    Count if is this a Saturday.

    Count if is this a Sunday.

    Read the Holiday table, count if this is a holiday.

    J := J + 1;

End loop;

Return (Days_between - Saturdays - Sundays - Holidays);

Avoid Loops By Using Calculations

- It was simple to write, but awful in execution.
- For only one calculation, there were many loops, and many reads to the Holiday table. And this happened for EVERY row of processing.
- For only 10 days apart: 20 IF statements, and 10 reads to the holiday table.

- When this procedure ran, the single query took up most of the CPU.
- One day's processing took hours each night.

Avoid Loops By Using Calculations

- I developed a function using calculations instead.
- For any week, there are at two standard non-business days: Saturday and Sunday.
- And we could count all the statutory holidays in a single statement.
- No loops required.

Avoid Loops By Using Calculations

The pseudo-code looked like:

Calculate the number of weeks between the two days.

Multiply by two to get the number of weekend days.

Make adjustments depending if you are starting or ending on a weekend day.

Avoid Loops By Using Calculations

Count the holidays from the holiday table in a single read,
And return the result:

```
Select count(*)
Into Holiday_days
From Holiday
Where Holiday_Date Between Start_Date and End_Date;

Return (total_days - weekend_days - holiday_days) ;
```

Avoid Loops By Using Calculations

Results:

Querying the view using the old method:	minutes
Querying the view using the new method:	seconds

Dozens of times faster.

Avoid Both Loops and Calculations

But wait. Since then, I have noticed Tom Kyte was able to do this in a single query!

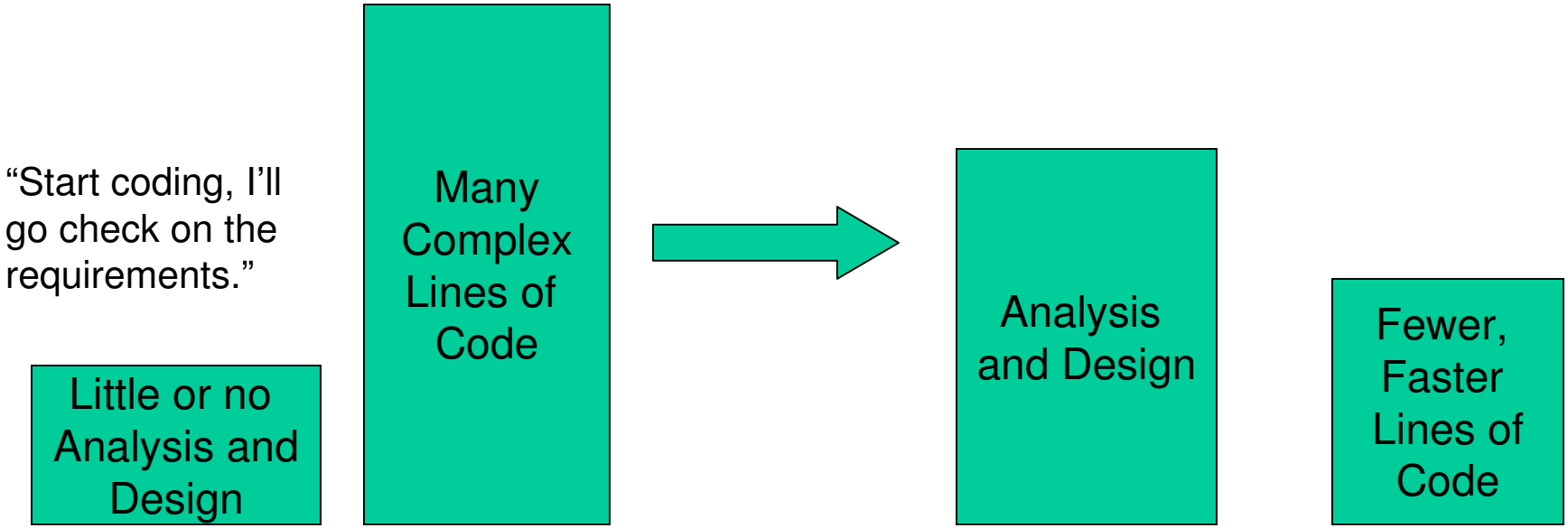
```
select count(*)
from ( select rownum rnum
          from all_objects
          where rownum <= to_date('&1') - to_date('&2')+1 )
where to_char( to_date('&2')+rnum-1, 'DY' )
             not in ( 'SAT', 'SUN' )
and not exists
       ( select null from exclude_dates
            where no_work = trunc(to_date('&2')+rnum-1) )
```

Source: asktom.oracle.com Brilliant!

Analysis and Design

Conclusion: For performance, do more analysis and design.

Write smaller, faster amounts of code.



Re-engineer the Code

- For performance reasons:
- Try to replace complexity with simplicity.
- Don’t throw more and more complexity at already complex processes.
- The added complexity will slow the processing down even more.

Analysis and Design

- Read and write to the database less.
- Try to have the goal: One SQL statement.

- Ask yourself:
  - Can I do all this work, in one big query?
  - How can I make this go faster?

- Alternatively, what might make this go slow?

Use Fewer Functions

- The same system, had another function:
- prev_business_day(P_DAY IN DATE)
- that was used in a view.

- Often, while checking the active processes during processing, I also noticed this function running for a long time:

- prev_business_day(prev_business_day(:b1))

- Nesting the function slowed down the processing.

Use Fewer Functions

- The same function could have been rewritten to take two parameters.

- Prev_Business_Day (P_Day, P_No_of_Days)

- where P_No_Of_Days would indicate how many number days previous. Default: 1 day.
- Or, pass in 2, 3, or more days.

Use Fewer Functions

- And used as:

- Prev_Business_Day (:v_today, 2)

- to return the second previous business day.
- This would avoid running the function twice (remember previous.)

- Code reuse.

Go For the Big Increases

- Go for the large increases in speed, not miniscule.
- By tuning the worst performing statements, you will usually speed up the whole system dramatically.

- The idea 20% gives 80% comes into play.

- 80% of the delays, can be found in less than 20% of the system.

Avoid Hard Parses to Memory

- Parse once, use many times.
- Avoid repeating hard parses to memory.

- Use bind variables instead of strings.
- In PLSQL, declare a variable, and use that, rather than using a new string each time.

- Very useful when developing in java!!!

Avoid Hard Parses to Memory

If the same statement is used more than once, use bind variables:

```
Var1      := func1(c1.field2);
Var2      := func2(c1.field3);
update    table_2
set       field_a = :var1
where     field_b = :var2;
...
update    table_2
set       field_a = :var1
where     field_b = :var2;
```

Avoid Hard Parses to Memory

- It must be exactly the same statement, otherwise, reparsed.

- The other method is to use a procedure, or function, and pass the values to it with bind variables.
- proc1 (:var1, var2);
- v_rtnval := func1 (:var1, :var2);

- But, don't go overboard, or many layers deep.

Inline Views

- I have had a lot of success with Inline Views.
- Inline Views are very useful to get data into SETS of information.

Inline Views

- Inline Views are a query, in the FROM clause of the SQL statement. (I used to call them virtual tables.)

```
Select ...
FROM
(SELECT A, MAX (A), COUNT(*)
FROM TABLE_X
GROUP BY A)    VIRTUAL,
TABLE_Y
where TABLE_Y.A = VIRTUAL.A
```

Inline Views - Uses

- Inline Views read all the data in a single query. Using them, you can:
- Use SETS, and eliminate loops.
- Filter large tables to a more manageable size.
- Right in a single query, combine different kinds of logic. Ie. Detail, with a GROUP BY.
- Inline views are VERY useful with poorly designed schemas.

Using Inline Views, Union, Intersect, Outer Joins Altogether

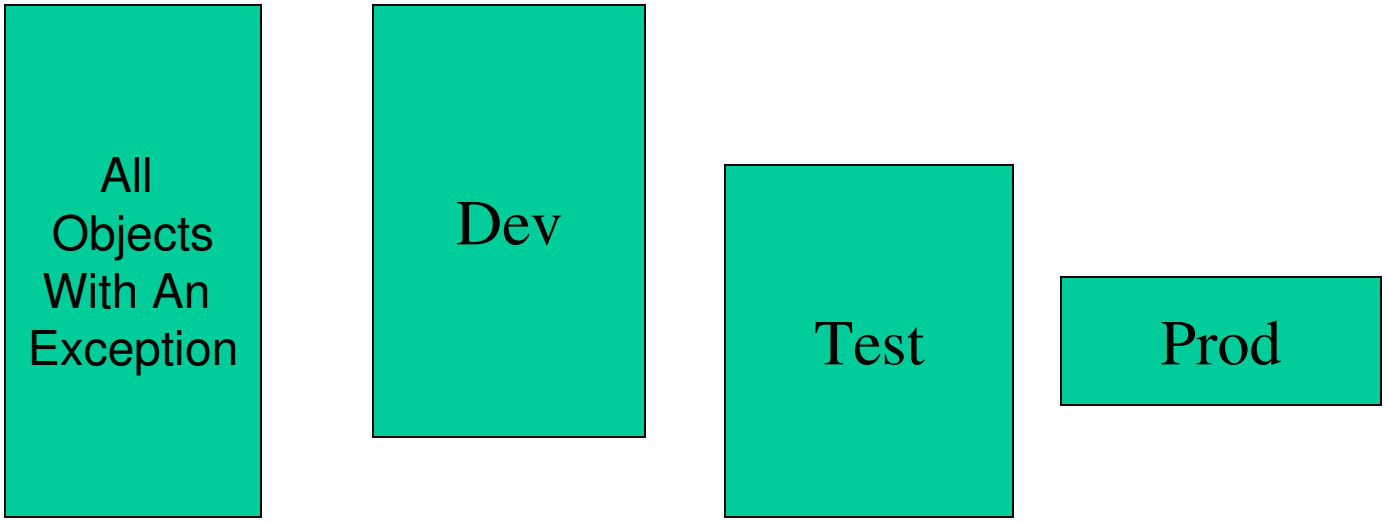
- Three databases. Dev, Test, and Production.

- Which objects have been created in one database, but not the other?
- In one query. Just the exceptions please.

- The tools I've seen for database comparison, only compare two databases. Not three.
- Use database links.

Using Inline Views, Union, Intersect, Outer Joins Altogether

Exceptions only. Missing in one or two databases.



Using Inline Views, Union, Intersect, Outer Joins Altogether

Get the SET of any and all objects from each database:
Use UNION. Create an inline view.

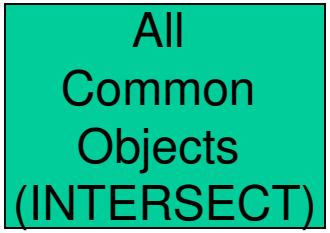
```
(
select    owner, object_name
from      all_objects@dev
where     ...  /* not the SYS, SYSTEM, etc. objects */
UNION
select    owner, object_name
from      all_objects  /* test */
where     ...
UNION
select    object_name
from      owner, all_objects@prod
where     ...
)         ALL_POSS_OBJ
```

Using Inline Views, Union, Intersect, Outer Joins Altogether

Get the SET of ALL COMMON objects from all three databases.
Use INTERSECT. Create an inline view.

```
(
select    owner, object_name
from      all_objects@dev
where     ... /* not the SYS, SYSTEM, etc. objects */
INTERSECT
select    owner, object_name
from      all_objects  /* test */
where     ...
INTERSECT
select    object_name
from      owner, all_objects@prod
where     ...
)         ALL_COMMON_OBJ
```

Using Inline Views, Union, Intersect, Outer Joins Altogether



(UNION)

MINUS

(INTERSECT)

gives the set of objects that are not found in at least one database.

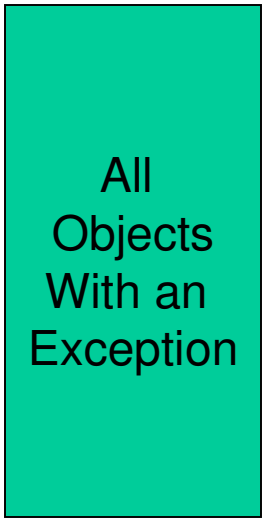
Using Inline Views, Union, Intersect, Outer Joins Altogether

All Objects With an Exception

Or, Outer join the two inline views together.

```
(
Select    poss.owner,  poss.object_name
From      (...)  All_Poss_Obj          poss,
          (...) All_Common_Obj     comm
Where     Poss.owner = Comm.owner (+)
and       Poss.object_name = Comm.object_name (+)
and       Comm.object_name IS NULL
)         ALL_EXCEPTIONS
```

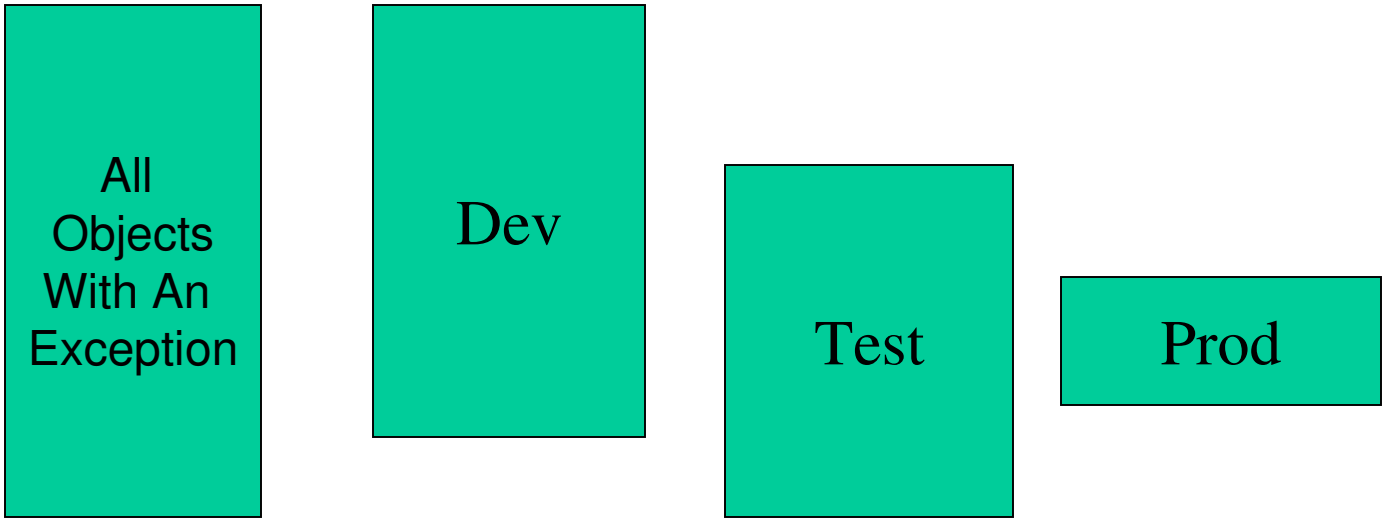
Using Inline Views, Union, Intersect, Outer Joins Altogether



Then, Outer join each database to this set of information.

(Actually, we could eliminate the INTERSECT, but it's kept for demonstration.)

Using Inline Views, Union, Intersect, Outer Joins Altogether



Using Inline Views, Union, Intersect, Outer Joins Altogether

```
- Outer join DEV, TEST, and PROD objects to this set of data.
Select    All_exceptions.owner || '.' || All_exceptions.object_name,
          Dev.owner || '.' || dev.object_name, test ..., prod ...
from      ((...UNION ...) MINUS (... INTERSECT ...))  All_exceptions,
          all_objects@dev              dev,
          all_objects                  test,
          all_objects@prod             prod
Where     all_exceptions.owner         = dev.owner (+)
and       all_exceptions.object_name   = dev.object_name (+)
and       all_exceptions.owner         = test.owner (+)
and       all_exceptions.object_name   = test. object_name (+)
and       ( dev.object_name  is null
or        test.object_name  is null
or        prod.object_name  is null )
```

Using Inline Views, Union, Intersect, Outer Joins Altogether

- It’s just one big query.
- About 70 lines long.

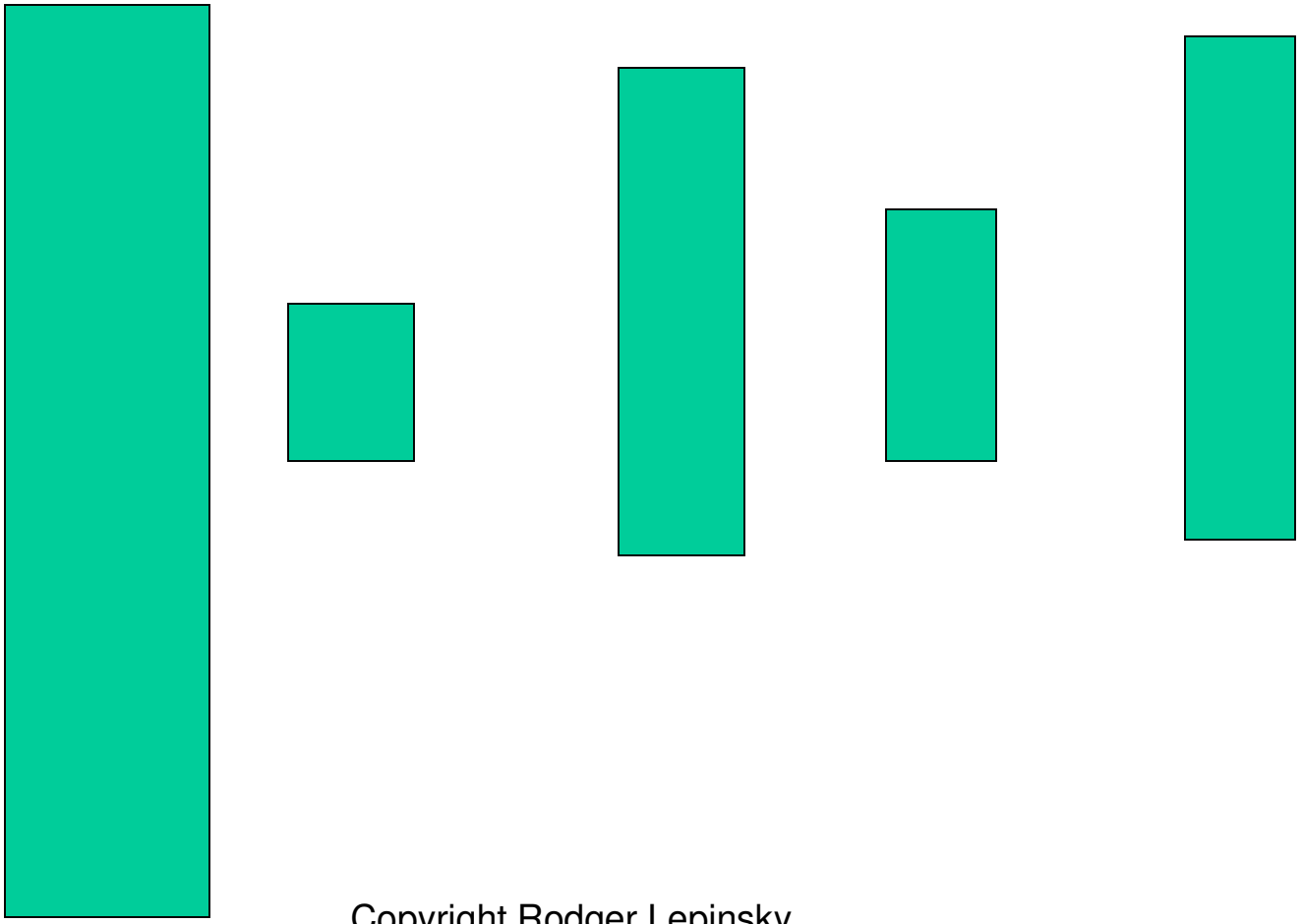
- Gives us exactly what we need.
- No loop de loop required.

Inline Views - Filtering Large Tables

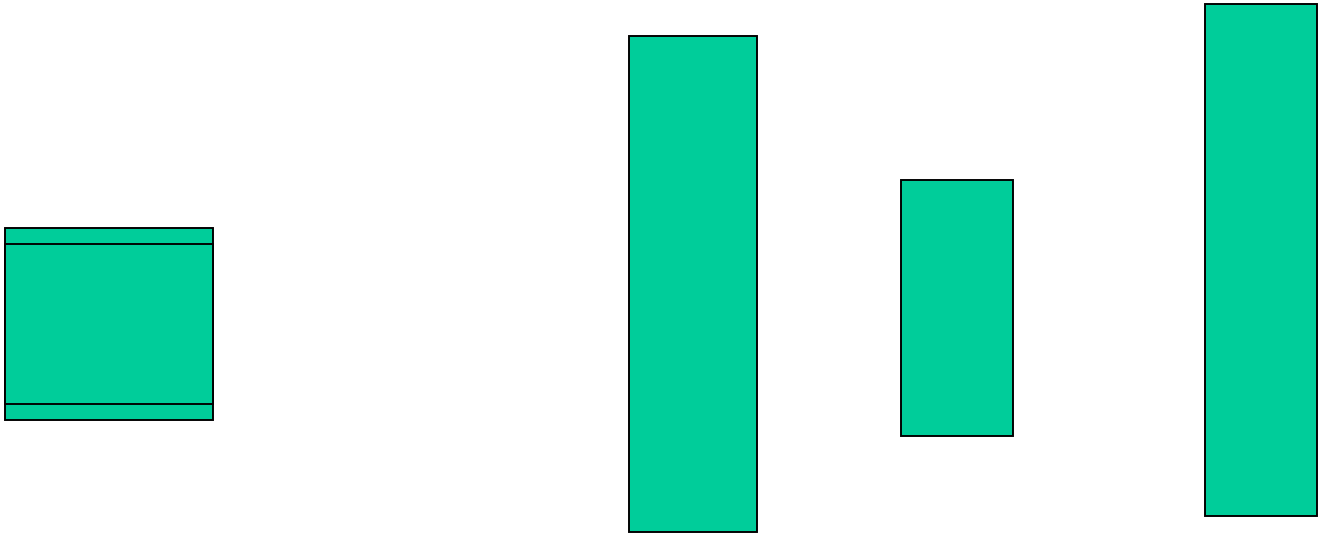
- Inline Views can be used when one table in the query, slows the query down substantially, and it runs out of rollback space.

- Example. Large query. A number of tables. One table is very large. And, a number of tables join to it. Many search conditions.
- But only a small subset of that big table is needed after the where clause is added.
- Create a Inline view with that table, and the filtering table.
- This will force the large table to be filtered by the smallest table first.

Inline Views - Filtering Large Tables - Before



Inline Views - Filtering Large Tables - After



Inline Views - Query Before

Once a co-worker's very complex query ran for 38 minutes, and then crashed. It ran out of rollback room. Later, after more rollback was added, it ran out of TEMP tablespace. It looked something like this:

```
SELECT    COUNT(*)  /* I often use count while debugging. */
FROM      ABC ,
          DEF,
          GHI,
          C,
          JKL,
          MNO,
          LARGE                 /*  <- LARGE TABLE  */
WHERE     LARGE.C_ID          = DEF.C_ID
AND       LARGE.ABC_ID        = ABC.C_ID
AND       LARGE.LID           = GHI.LID /* join conditions on LARGE. Non-PK */
AND       LARGE.STU_ID        = MNO.STU_ID
AND       C.ID                = ABC.C_ID
AND       JKL.XYZ_ID          = DEF.XYZ_ID
AND       LARGE.field_A = ... /* many filter conditions on LARGE. Non Indexed */
```

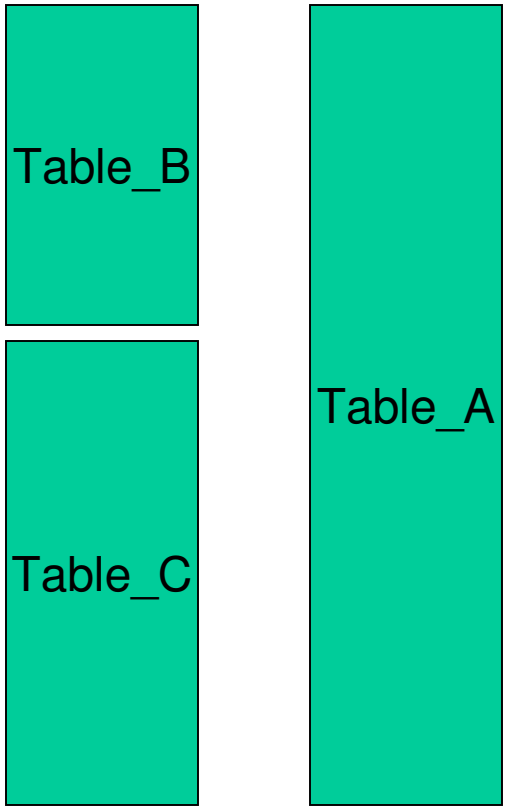
Inline Views - Query After

I rewrote the query using an Inline View much like this:

```
Select      count(*)
FROM        JKL, ABC, DEF, GHI, C,
            (Select                 /* Start Inline view here */
            LARGE.White_id,
            LARGE.C_ID,             /* Select all the fields needed for the join */
            LARGE.STU_ID,
            LARGE.abc_id,
            LARGE.LID
            from                    mno,  /* Join to small table */
                                    LARGE
            where       LARGE.STU_ID   =  mno.STU_ID
            and         LARGE.White_id is null          /* Filter */
            )           VIRT        /*  <- give the Inline view an ALIAS */
WHERE                   VIRT.lid                = GHI.lid
AND                     VIRT.C_ID               = DEF.C_ID
AND                     VIRT.ABC_ID             = abc.C_ID
AND                     c.ID                    = abc.C_ID
AND                     JKL.XYZ_ID              = def.XYZ_ID
and                     ...  /* many filter conditions */
```

This ran in less than 6 seconds; about 380 times faster (assuming that the first query finished, which it hadn't).

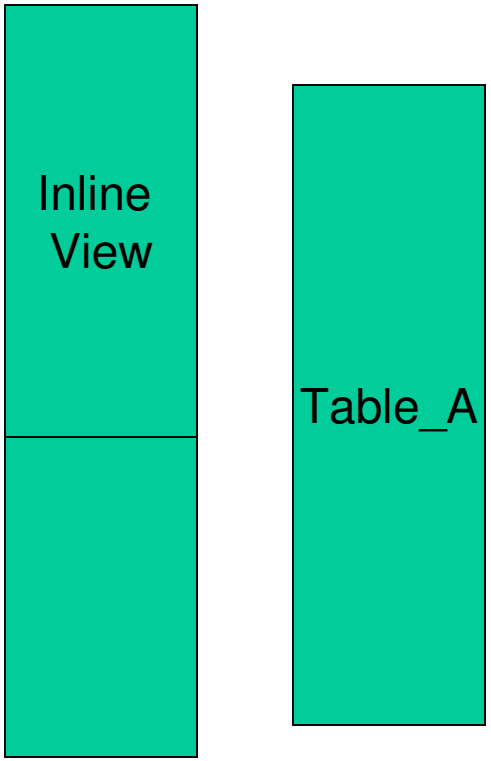
Inline Views, with Outer Joins



You cannot outer join the same table, to more than one table.

```
SELECT     COUNT(1)
FROM       TABLE_A,
           TABLE_B,
           TABLE_C
WHERE      TABLE_B.A_ID = TABLE_A.A_ID (+)
AND        TABLE_C.A_ID = TABLE_A.A_ID (+)

ERROR at line 5:
ORA-01417: a table may be outer joined to at
most one other table
```

Inline Views, with Outer Joins



Solution: Join the two “big”, or left hand side tables together into an inline view, then outer join the third table to that.

```
Select count(1)
from
(Select    TABLE_b.a_id
From       TABLE_b,
           TABLE_c
Where      TABLE_b.a_id = TABLE_c.a_id
/* can add more logic here */
)                    virtual,
                     Table_a
Where      virtual.a_id = table_a.a_id (+)

- works
```

Inline Views

- This way, you can still get one big SET of data.
- I have used inline views with outer joins on a number of poorly designed data models.

- Result: four to dozens of times faster.

- Note: Before using this technique, you really have to understand your data, the data model, and what you are trying to do. Analysis is required.

Scalar SubQueries

- Scalar SubQueries are similar to an Inline View, but are not in the FROM clause.
- A SQL statement, inside the SELECT clause. Almost like a function.
- Must return zero or one row. Otherwise, error.
- I personally haven't used this much, so can't comment on its performance.

Scalar SubQueries

```
Select  username,
        (select count(*)
        from all_constraints   /* <- Scalar SubQuery */
        where owner = username)   cons
from    all_users

USERNAME                          CONS
------------------------------ ----------
WKSYS                              57
WKPROXY                             0
ODM                                 0
ODM_MTR                             0
LBACSYS                             2
OLAPSYS                            93
...

Efficiency?
```

Finding Problem SQL

- How to find the problem SQL?

- As a developer, you may or may not have the rights to query these parts of the data dictionary.
- VSQL, VSQLAREA, V$SESSION

- As a DBA, I use the following queries to find problem SQL statements. Then, tune the statements.

Finding Problem SQL

To find the current active processes. Run this frequently during periods of high activity. Note the SQL statements running most often.

```
Select
sid,
serial#,
username,
command,
sql_text
from          v$session    sess,
              v$sqlarea    sql
where         sess.sql_address    = sql.address
and           command > 0
and           status = 'ACTIVE'
```

Finding Problem SQL

To find the SQL statements with lots of activity:

```
Select
  SQL_TEXT,
  EXECUTIONS,
  SORTS,
  DISK_READS,
  BUFFER_GETS,
  ROWS_PROCESSED,
  OPTIMIZER_COST,
  (BUFFER_GETS/ROWS_PROCESSED)  BUFFERS_PER_ROW
from          v$sql
where         optimizer_cost is not null
AND           ROWS_PROCESSED > 0
Order by      8
```

Order by the critical factor you want to see.

Summary:
Techniques For Queries

- Summary of my frequently used techniques for speeding up queries and processing.
- Outer joins
- Inline views
- Functions
- Temp tables

- Also used, but less frequently:
  - Indexes
  - Hints

Summary of Principles

- Hard drives are the slowest part of any system.
- The server is faster than the client.
- Read the data once. Avoid loops. Attempt to do everything in one SQL statement.

Updates

- Updating row by row is very slow.
- Avoid this at every opportunity.
- Just like queries, try to write all the data in one SQL statement.

- Update Table_A
- Set field_1 = something
- Where field_2 = x;

Updating Many Rows

- With the single Where clause, the updated field will get the SAME value in all the rows updated, whether one, or many rows.
- To update the same field with DIFFERENT values, developers often use a loop.
- This can be avoided by using a subquery:

Updating Many Rows With Different Values

```
update    emp1
set       ename = null

14 rows updated.

Update    emp1
Set       (ename) =
          (Select    ename
          From      emp2                          /* Another table */
          Where     emp1.empno = emp2.empno)   /* Join condition here */

14 rows updated.

SQL> Select ename
  2  from emp1

ENAME
----------
SMITH
ALLEN
WARD
...
```

Updating Many Rows

- Advantages:
- Less coding.
- No loops.

- Disadvantage:
- For this technique to work, you need to be able to produce a subquery with the correct data first. If complex logic is required, this may not be possible.

Inserts And Updates - FORALL

- Forall / Bulk Collect
- The FORALL can be much faster than regular FOR loops.
- It uses a similar principle: read or write to the database -once-, rather than many times.

- “This technique improves performance by minimizing the number of context switches between the PL/SQL and SQL engines. With bulk binds, entire collections, not just individual elements, are passed back and forth.”
- Oracle 9i documentation. PL/SQL User's Guide and Reference

FORALL - Advantages

- Advantage of using FORALL: Speed
- In tests (2001), I was trying hard to get to 11,000 rows per second. With a single cursor, basic insert speed was about 6,000 rows per second.
- Using FORALL, I was able to get the inserts to 25,000 rows per second.
- Over 4 times as fast.

FORALL - Disadvantages

- Disadvantage of using FORALL: More coding.

- A PLSQL table has only one field in it, other than it’s index.
- For each field that you want to INSERT/UPDATE, you have to declare a PLSQL table of the same datatype and populate each table first.

- It's not like using %rowtype,

FORALL - Disadvantages

- The FORALL cannot be used as a regular loop.
- You can only use one INSERT, UPDATE, or DELETE statement with each FORALL statement.

```
FORALL  J in Lower .. Upper
        Insert into table_a  (field1)   values  (pltabvar (j);
```

FORALL - Disadvantages

- Error handling is more complex, and cannot be done in the same manner, as with a regular FOR loop. Ie. Row by row, if statement, procedural logic.

- Dupe data can be problematic.

- The speed advantage seems to be lost if the table has a number of concatenated fields in the primary key, or other indexes.

Inserts and Updates

Often, I've seen PLSQL code that will count first to see if data exists, before doing an insert or update.

```
Select    Count(*)
Into      ln_count
from      table_a
Where     field1 = somevalue;

If (ln_count >= 1) then

          update table_a  ...

Else

          insert  into table_a  ...

end if;
```

Inserts and Updates

- This method reads the database twice.

- Once for the count.
- The second time for the insert or update.

- You can eliminate one read as follows:

Inserts and Updates

```
Begin
        Insert into table_a  ...
        Exception
                When DUP_VAL_ON_INDEX then
                        Begin
                        Update table_a ...
                                Exception ....
                        End;     /* update */
End:    /* Insert */
```

Inserts and Updates

Alternatively, you can do this:

```
Begin
        Update Table_A  ...
        If ( sql%rowcount = 0 ) then
                begin
                Insert into Table_A ...
                        Exception ....
                End;    /* insert */
        End if;
End:    /* update*/
```

Inserts and Updates

- If you expect to do more INSERTs, use the one that inserts first.

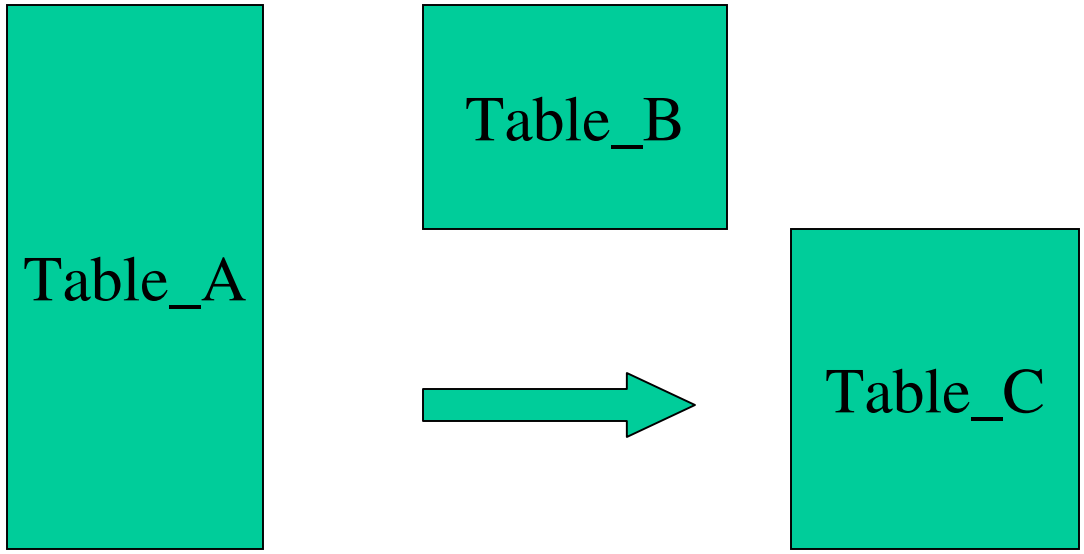
- If you expect the code to do more UPDATEs, use the one that updates first.

Lots Wrong to Much Better

- I was once asked to tune up some very slow processing.
- The task was to find data that was in Table_A, but not Table_B, and insert it into Table_C.
- Table_C was a collection of “bad” records, that would then be used for clean up.

Lots Wrong to Much Better

- Can you see one technique already?



Lots Wrong to Much Better

- Complicating the issue was that these tables used large objects.

- In one table, the data was found in a regular fields.
- In the other table, data was found inside the CLOB.

- Extracting the data, required the use of a number of functions.

Lots Wrong to Much Better

- Methods used that slowed down the processing included:

- Nested loops, instead of one big cursor.
- Row by row processing. All the functions used to parse the data were done in each row.
- Used cursors, instead of a count.
- Used the DBMS_SQL statement for SELECT statements (not DDL). Caused hard parses in the SGA, every row!

- Some of these methods were actually used for “performance”.

Lots Wrong to Much Better

- The code had an explicit cursor to find the min() of table.
- And then a second explicit cursor to find the max() of the SAME TABLE, with the same where clause.
- One explicit cursor could have been used.

- But the same logic could have been done in one Select statement:
- Select MIN(field1), MAX(field2)
- Into var1, var2
- from Table_A ...

Lots Wrong to Much Better

One cursor was opened on the first table.
Then, a second cursor was opened on the second.
Nested loops.

```
For        C1 in  Table_A_cursor loop

           For        C2  in  Table_B_cursor ( C1.field1 ) loop
                     ...

           end loop;

end loop;
```

Lots Wrong to Much Better

Inside the nested loops, there was a lot of processing to extract the data out of the CLOB.

var1 = func1 (func2 (func3 (c2.CLOB_FIELD)))
var2 = func4 (func5 (func6 (c2.CLOB_FIELD)))

Lots Wrong to Much Better

Then, the DBMS_SQL statement was used, along with exception handling to do the INSERT.

V_String := ‘Select ‘’OK’’ from Table_C where || to_char (var1) ... || to_char (var2) .... ‘

Execute Immediate V_STRING ....

```
Exception
        WHEN NO_DATA_FOUND THEN
                INSERT  INTO TABLE_C ....
        END;
```

This parsed each and every row in the SGA!

Lots Wrong to Much Better

To use the same logic, a simple count(*) statement could have been used instead.

```
Select          Count(*)
into            LN_TEMP
from            Table_C
where           field1 =  :bind_var

If      (LN_TEMP = 0 )          then
        INSERT into TABLE_C ....
end if;
```

It would have avoided the hard parse and would be much easier to code.

Lots Wrong to Much Better

Alternatively, they could have just inserted the data, and done nothing if it already existed:

```
Insert into Table_C
        (field1, field2 ...)
Values
        (:var1, :var2 ... );
Exception
        When DUP_VAL_ON_INDEX
                NULL;
        END;
```

This would have eliminated one more read.

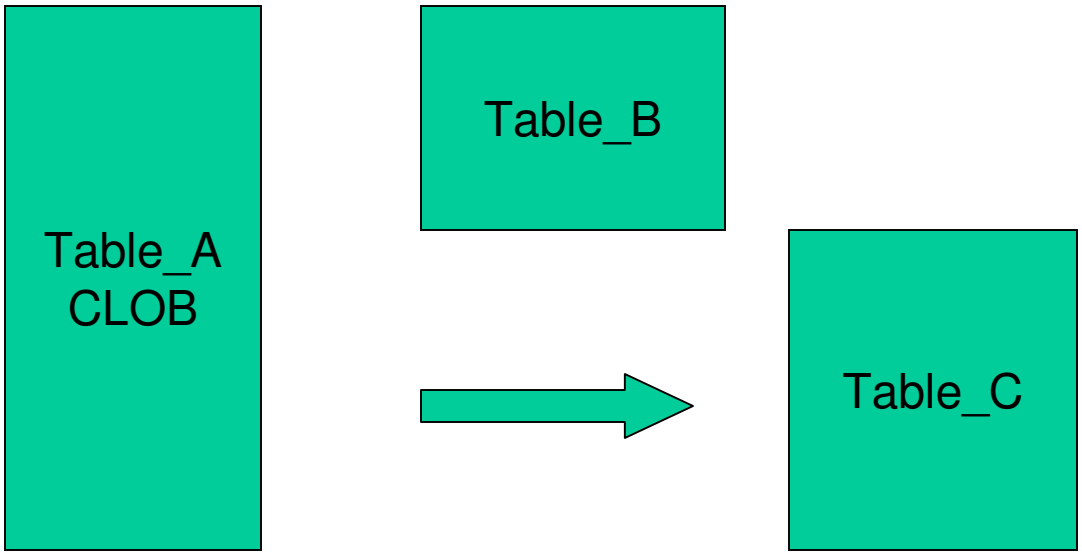
Lots Wrong to Much Better

- I was asked to take a look at this code, because it was running very slowly.

- Very soon, I gave up trying to tweak it.
- I asked, what is this code supposed to do?
- Once I understood what all the functions were extracting, I came up with a different strategy.

Lots Wrong to Much Better

- Techniques:
- Outer join.
- Run all the functions in the cursor, not the body.



Lots Wrong to Much Better

```
Select      A.Field1 ,
            Func1(func2(func3(A.CLOB_field2))) ...
From        TABLE_A     A,
            TABLE_B     B
WHERE       Func4(func5(func6(A.CLOB_field2 )))
                  = B.fieldA (+)
and         ...
```

Lots Wrong to Much Better

The body of the procedure was just a case of opening the cursor, and doing the insert:

```
For     C1      in      Big_cursor      loop
        Insert into Table_C
        (field1, field2 ... )
        values
        (c1.field1, c1.field2 ...);
end  loop;  /* error handling not shown */
```

Lots Wrong to Much Better

- Response times.
- In the test environment, the number of rows was in the hundreds of thousands, about 200,000.

- Testing showed the original method took well over 30 minutes to run. This was just to SELECT the data.
- My streamlined method ran in about 2 minutes.
- At least 15 times faster.

The Multi-Million Dollar Mistake

- Very slow database. Built (before I arrived) with no real database expertise.
- Single update taking two hours.
- New hardware purchased. Top of the line. Still slow.
- Their conclusion: "Big iron didn't help." Even more CPU power is needed.

The Multi-Million Dollar Mistake

- So, the next new system was designed with custom built cluster computing using 50 Linux machines.
- The middleware was very expensive to build, and was tied into the application.
- The new product would not work on Windows, or Unix. The customer would have to implement the 50 Linux boxes of middleware, in addition to the app. Difficult sell.

The Multi-Million Dollar Mistake

- Underlying cause. Rollback problem. The update kept running out of rollback space.
- They brought in a (poor) database consultant who increased the rollback room to 20 gigs. But the problem persisted.
- Make it work anyway. Updates done in loops, row by row. With respect, the programmer did know that this was slower and incorrect. He was no dummy.
- (Note: this solution did require DBA knowledge, but it is mostly a developer issue.)

The Multi-Million Dollar Mistake

- Cause: MAXEXTENTS parameter not set right.
- I fixed this parameter, created an identical test table, and updated it with a single update statement.

- Result:
- Old row by row update: Over 2 hours.
- New single update statement: 4 seconds.

- About 2000 times faster.

The Multi-Million Dollar Mistake

- Result of the new company.
- Money was not spent on initially hiring database expertise.
- But much money was spent on hardware, consultants, developing middleware (eventually scrapped), and later re-architecting the system.
- Long story, but the company is now out of business.

The Multi-Million Dollar Mistake

- Technical Conclusions:

- Avoid row by row updates.
- A single update statement is faster.

- Reads and writes to the hard drive are the slowest part of any computer system.
- Always be concerned with IO to the hard drive first. Be concerned with the CPU second. Extra CPU will not help if you are waiting on disk reads and writes.

The Multi-Million Dollar Mistake

- Other Conclusions:
- Extrapolating the logic, might make logical sense, but it's not always correct . ("It's the CPU." Not.)
- Database expertise is still important. Especially in database centric operations.

Questions

- ????

- Would a seminar on data design for speed be useful?
- Data Design for High Speed Systems

END

- Thank you!