Performance Tuning with reference to Oracle

Hardeep Singh
http://blog.hardeep.name

Performance tuning activity can be applied to:

1. An application
2. Process tuning
3. A SQL query

Whenever an application is requires tuning, the first step is to go look at the code. If the code is not optimal, no amount of Oracle tuning will speed up the application. Once that step is taken care of, we can come to Oracle level tuning.

Let's start with **SQL query tuning**.

A **filter** is something that restricts certain rows from appearing in the output. For example

```
where employee.emp_type='P'
```

However, the following is not a filter, it's a **join**:

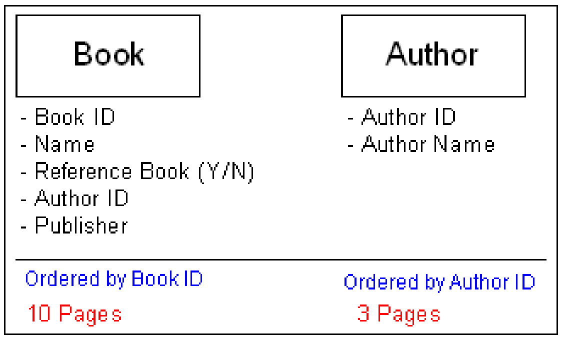
```
where a.emp_type = b.emp_type
```

Some of the common query pitfalls are:

- Using NOT EQUAL TO prevents the use of indexes.
  ```
       a.emp_type <> 'P'
  ```
  Replace with EQUAL TO, LESS THAN or GREATER THAN wherever possible
  ```
       a.emp_type = 'T'
  ```

- Large IN lists also prevent use of indexes
- Functions (e.g. substring) prevent the use of indexes
- UNION operation involves an internal DISTINCT operation to eliminate all the duplicate rows, and as such is expensive. If the distinctness is not that important for your query, use UNION ALL instead of UNION

Book-Author example



Consider the two tables above: the book and the author tables. If the information had to be printed - **the book table would print on ten pages and the author table on three.**

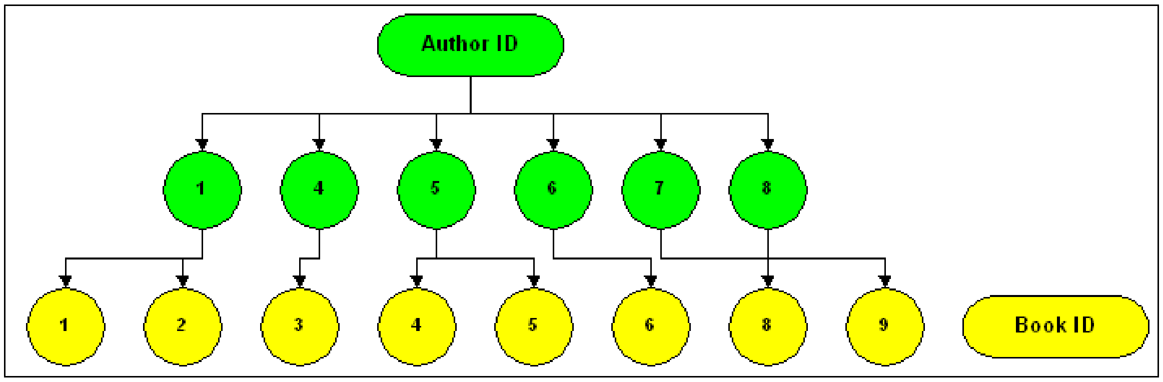
In order to tune queries involving these tables, it's important to imagine how you would search for the information if you were given the table contents printed on paper.

Consider the query:

"I need all the author names who have authored reference books"

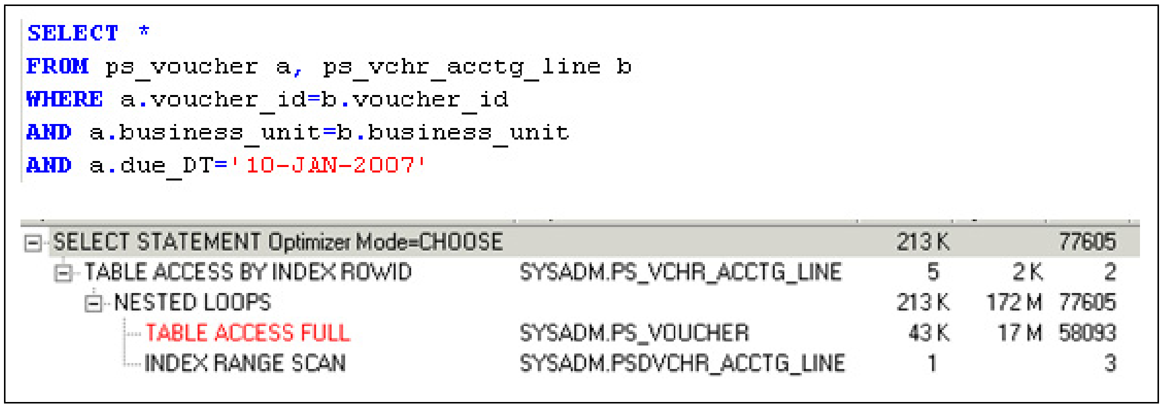
We can either start at the book table, or at the author table. If we start with the author table, we can lookup the first author ID, and then look through the entire book table if he has any reference books. The other way is to start with the book table, and when we find a reference book we can check the author name against that. Which one is faster? The second – starting with the book table. Why? – because the author table is order by `Author ID`.

Now let's have a look at how Oracle interprets indexes.



If we had an index on the Book table, on the fields `Author ID` and `Book ID`, this is how it would logically look. It can be used if we know both the `Book ID` and the `Author ID` (have both the fields in the where clause of the query). It can also be used if we know just the `Author ID`, but will be slightly slower (see RANGE SCAN below). However, it cannot be used if we know just the `Book ID`.

Before executing a query, Oracle creates a **plan** - the order in which it will search the tables, the indexes it will use, the way it will use the indexes etc. Step 1 in tuning is to check the plan for the SQL in question:



```
SELECT *
FROM ps_voucher a, ps_vchr_acctg_line b
WHERE a.voucher_id=b.voucher_id
AND a.business_unit=b.business_unit
AND a.due_DT='10-JAN-2007'
```

```
⊟ SELECT STATEMENT Optimizer Mode=CHOOSE                                213 K          77605
  ⊟ TABLE ACCESS BY INDEX ROWID          SYSADM.PS_VCHR_ACCTG_LINE        5      2 K       2
    ⊟ NESTED LOOPS                                                      213 K  172 M   77605
        TABLE ACCESS FULL                SYSADM.PS_VOUCHER               43 K   17 M   58093
        INDEX RANGE SCAN                 SYSADM.PSDVCHR_ACCTG_LINE        1                3
```

In order to generate the plan, on SQL Plus use:

```
set autotrace on;
```

or if we want to just see the trace without executing the query use:

```
set autotrace traceonly explain;
```

With Toad, you can press **Control – E** to see the plan.

Now looking at the explain plan shown above, what does it mean? The point to note is that explain plan is actually executed from more indented to less indented (inside to outside) and from bottom to top.

Table access full means that the complete table needs to be searched in order to find the information. **Index range scan** means that index will be used, however the filters provided in the WHERE clause may not be 'complete'.

Index full scan means the entire index will be searched.

Table access by index rowid means the rows are being located using an index.

Hash, **Nested loops** and **Merge** are all different types of joins. These are the most commonly seen operations in an explain plan.

More information on the plan: http://www.orafaq.com/node/1420

Now that we know how Oracle is executing our query, the next step is to influence the way Oracle runs the query. How to do that?

- ❖ By creating indexes and optimizing the join order of tables
- ❖ By creating **statistics**
- ❖ By adding **hints** to the query

I have already discussed indexes. An example of how to create an index:

```
CREATE INDEX book_author_index ON book (book_id,
author_id);
```

Statistics refers to the information that Oracle can store about tables and fields on your request that will help it to make more informed decisions regarding the plan. For example, let us say that purchase order table P has a field STATUS which can be 'O' (Open) or 'C' (Close). If you give Oracle a query with this field in the WHERE clause – it will assume that O & C values are equally distributed: 50% each. In truth this may not be the case: O values maybe present only in 10% of the rows.
Such information being calculated offline, stored and used during plan making is called statistics.
To create statistics use something like:

```
EXEC DBMS_STATS.gather_table_stats('SYSADM', 'BOOK');
```

More information on statistics: http://www.oradev.com/create_statistics.jsp

A **hint** is written as part of the query, but doesn't affect the output. It only affects the plan.
For example:

```
SELECT /*+ INDEX(A IDX_PURCH_ORD) */ *
FROM PURCH_ORD A
WHERE STATUS='O'
```

The hint is written within comments, but with a plus sign to indicate that it's a hint, not a comment. This hint tells Oracle to use the index `IDX_PURCH_ORD` to lookup the table `PURCH_ORD`. Most of the time Oracle automatically uses the correct index, however is certain 'tough' situations it needs help.

One of the most useful hints that I have found is the LEADING hint. This hint tells Oracle to start with a particular table from the tables used in the query. For example:

```
SELECT /*+ LEADING(i) */ po_id, status
FROM PURCH_ORD p, INVOICE i
WHERE invoice='123456' AND
i.po_id=p.po_id
```

In this query, the LEADING hint tells Oracle to start with the INVOICE table, find the PO_ID for the invoice, and use that to lookup the STATUS in the p table.

A good list of all indexes is available here: http://www.adp-gmbh.ch/ora/sql/hints/index.html

Now coming to **process tuning**. We already know how to tune an SQL query. The problem that remains now is to find out which query is doing badly, from among all the queries being executed by the process. If the process is small / simple – a simple inspection of the code will give you an idea of the problem SQL. In a slightly larger SQL you may be able to add some code to display the current time before and after each query. However, if that is not possible, or the SQL is very large – we need some automated tracing to find the SQL. One good way to do this is through the **10046 trace**.

To perform this trace the process should execute the following SQL statement as the first step:

```
ALTER SESSION SET EVENTS '10046 trace name context forever,
level 12'
```

Once the process completes, the DBA should be able to check the trace output, and run it through **tkprof** to give you an HTML report. This report gives you a list of the worst queries, the corresponding plans and the waiting times for those queries. It's a wealth of information to pinpoint the problem SQLs and start tuning them.

If the DBA gives you a trace file (from the udump folder on the database server) and not a report, you can use the trace file to generate the report yourself as below:

```
tkprof <trace_file> <output_file> sort=fchela
```

Lets move on now to **application tuning**. This is similar to process tuning except that we need to find the worst performers in the entire application, not just a given process. One way to do that is through the STATPACK analysis. The DBA can help in generating a STATPACK report for a period (say one day) and would give you a report of the worst SQLs running during the period. More information on STATPACK is here: http://docs.oracle.com/cd/B10500_01/server.920/a96533/statspac.htm

A quick cheat sheet:

When you get a query to tune, what are the things we can quickly look at to get going? Here is a quick cheat sheet:

- Try to limit the output. For example, if we want a list of open orders, we may not be interested in orders from more than 2 years back. Hence, if the `order_date` is an indexed column, we can add:
  `order_date >= SYSDATE-700`
- Have a look at the indexes being used. The index usage may not be optimal.
- Have a look at the join order of the tables. As a rule, Oracle finds it hard to come up with a good join order where the number of tables is 4 or more. Use the LEADING hint if needed, or the USE_NL hint.
- If the query uses multiple business units such as:

```
business_unit IN (‘NL100’,’NL200’,’NL300’)
```
it may be faster to run the query separately, once for each business unit. If it must run as a single query, explore connecting the three separate queries through UNION ALL

- ❖ If its being run for multiple FISCAL_YEARs, the same applies as business unit above
- ❖ It we are joining the tables (or self join), it is important to use all the keys in the join. For example

<u>Case study 1:</u>

One of the queries in REQSOURCE (requisition sourcing application engine program) was the **worst performing query of the system.**
The query used to take more than 15 minutes to run, and would cause a lot of IO operations, throttling other processes. We had to keep this process on hold whenever we noticed performance issues, and release it later when the performance became better.
The reason this query was such a problem was that a lot of Canadian requisitions remained open in the system. This was more due to business processes, and a process had been created to close the requisitions. However the new process did not receive business approval because of the deemed risk.

Hence we decided to think of something out of the box.

The query was this:

```
INSERT INTO PS_REQ_DIST_SUM_WK
(PROCESS_INSTANCE,  BUSINESS_UNIT,  REQ_ID,   LINE_NBR,  SCHED_NBR,
QTY_REQ,  MERCHANDISE_AMT,  QTY_OPEN,  QTY_OPEN_STD,  AMT_OPEN,
AMT_OPEN_BSE,  DISTRIB_PCT )
SELECT
 &BIND(PROCESS_INSTANCE)
,  DIST.BUSINESS_UNIT
,  DIST.REQ_ID
,  DIST.LINE_NBR
,  DIST.SCHED_NBR
,  SUM(DIST.QTY_REQ)
,  SUM(DIST.MERCHANDISE_AMT)
,  SUM(DIST.QTY_OPEN)
,  SUM(DIST.QTY_OPEN_STD)
,  SUM(DIST.AMT_OPEN)
,  SUM(DIST.AMT_OPEN_BSE)
,  SUM(DIST.DISTRIB_PCT)
FROM     PS_REQ_LN_DISTRIB DIST
WHERE  (DIST.REQ_ID, DIST.BUSINESS_UNIT, DIST.LINE_NBR, DIST.SCHED_NBR)
  IN (SELECT /*+ LEADING(HDR) USE_NL (HDR SHIP) */
            SHIP.REQ_ID, SHIP.BUSINESS_UNIT, SHIP.LINE_NBR, SHIP.SCHED_NBR
      FROM PS_REQ_HDR HDR
          ,PS_REQ_LINE_SHIP SHIP
      WHERE  &BIND(SELECT_PARM1,NOQUOTES)
         HDR.REQ_STATUS = 'A'
      AND     HDR.HOLD_STATUS = 'N'
      AND     HDR.RFQ_IND = 'N'
      AND     HDR.IN_PROCESS_FLG = 'N'
      AND     HDR.BCM_HDR_STATUS = 'V'
      AND     SHIP.BUSINESS_UNIT = HDR.BUSINESS_UNIT
      AND     SHIP.REQ_ID = HDR.REQ_ID
      AND     NVL(SHIP.BAL_STATUS,' ') = 'I'
      MINUS
      SELECT S.PO_STG_ID, S.BUSINESS_UNIT, S.LINE_NBR, S.SCHED_NBR
      FROM PS_PO_ITM_STG S
      WHERE  S.PO_STG_TYPE = 'R'
     )
GROUP BY     DIST.BUSINESS_UNIT
            ,DIST.REQ_ID
            ,DIST.LINE_NBR
            ,DIST.SCHED_NBR
HAVING (SUM(DIST.QTY_OPEN_STD)>0) OR (SUM(DIST.AMT_OPEN)>0)
```

The problem with this query was that the filter that would cause the maximum reduction of rows in the output (the main 'exclusion' clause) was in the HAVING part, due to which it was executed at the very end. As a result, a large number of requisitions had to undergo all the calculations, only to be dropped at the end because they did not satisfy HAVING criteria.

Since the conditions were based on SUM aggregator, not on individual row values, they could neither be improved through indexes, nor moved to the WHERE clause.

We came up with this key idea: since neither quantity nor amount can be negative, hence for the SUM to be more than zero, at least one of the individual rows have to be more than zero.

As a result we created an index on BUSINESS_UNIT, QTY_OPEN_STD and AMT_OPEN. Using this index we rewrote the SELECT part of the query as below:

```
SELECT &BIND(PROCESS_INSTANCE)
,  DIST.BUSINESS_UNIT
,  DIST.REQ_ID
,  DIST.LINE_NBR
,  DIST.SCHED_NBR
,  SUM(DIST.QTY_REQ)
,  SUM(DIST.MERCHANDISE_AMT)
,  SUM(DIST.QTY_OPEN)
,  SUM(DIST.QTY_OPEN_STD)
,  SUM(DIST.AMT_OPEN)
,  SUM(DIST.AMT_OPEN_BSE)
,  SUM(DIST.DISTRIB_PCT)
FROM     PS_REQ_LN_DISTRIB DIST
WHERE  EXISTS(SELECT /*+ INDEX(X PSDREQ_LN_DISTRIB) */  1 FROM
PS_REQ_LN_DISTRIB X WHERE
         X.BUSINESS_UNIT=DIST.BUSINESS_UNIT AND X.REQ_ID=DIST.REQ_ID AND
X.LINE_NBR=DIST.LINE_NBR
         AND X.SCHED_NBR=DIST.SCHED_NBR AND (X.QTY_OPEN_STD>0 OR X.AMT_OPEN>0))
AND
       (DIST.REQ_ID, DIST.BUSINESS_UNIT, DIST.LINE_NBR, DIST.SCHED_NBR)
  IN (SELECT /*+ LEADING(HDR) USE_NL (HDR SHIP) */
             SHIP.REQ_ID, SHIP.BUSINESS_UNIT, SHIP.LINE_NBR, SHIP.SCHED_NBR
      FROM PS_REQ_HDR HDR
          ,PS_REQ_LINE_SHIP SHIP
      WHERE  &BIND(SELECT_PARM1,NOQUOTES) DIST.BUSINESS_UNIT=HDR.BUSINESS_UNIT
      AND     HDR.REQ_STATUS = 'A'
      AND     HDR.HOLD_STATUS = 'N'
      AND     HDR.RFQ_IND = 'N'
      AND     HDR.IN_PROCESS_FLG = 'N'
      AND     HDR.BCM_HDR_STATUS = 'V'
      AND     SHIP.BUSINESS_UNIT = HDR.BUSINESS_UNIT
      AND     SHIP.REQ_ID = HDR.REQ_ID
      AND     NVL(SHIP.BAL_STATUS,' ') = 'I'
      MINUS
      SELECT S.PO_STG_ID, S.BUSINESS_UNIT, S.LINE_NBR, S.SCHED_NBR
      FROM PS_PO_ITM_STG S
      WHERE  S.PO_STG_TYPE = 'R'
     )
GROUP BY     DIST.BUSINESS_UNIT
            ,DIST.REQ_ID
            ,DIST.LINE_NBR
            ,DIST.SCHED_NBR
HAVING (SUM(DIST.QTY_OPEN_STD)>0) OR (SUM(DIST.AMT_OPEN)>0)
```

In other words, we added the EXISTS clause in the beginning and added a comparison of BUSINSS_UNIT (as per a tuning idea stated above that all primary keys must be compared in the join to make efficient use of indexes). In addition, we let the HAVING clause at the end be, to be double sure that correct data results.

In addition to this main step of performance tuning, we undertook the following:

- since the explain plan didn't show the newly created index as being used, we entered a HINT:
  ```
  SELECT /*+ INDEX(DIST PSBREQ_LN_DISTRIB) */
   &BIND(PROCESS_INSTANCE)
  ```
- The table `PS_REQ_DIST_SUM_WK` was cleaned up of its historic data
- Rebuilt statistics on `PS_REQ_LN_DIST` table.

As a result of these changes, the average processing time came down from **22 min to 5 min.** In addition the positive impact on the system performance due to reduced IO was tremendous.

<u>Case study 2:</u>

We had created a process to forecast the amount money Rockwell may expect from its customers in the upcoming period (7 days, 14 days etc). This report had to run once per business unit. The stream took a long time to run and needed tuning. We determined that it was not possible to tune the process just from a SQL tuning perspective: the SQR process would need to be looked at.

The SQR was based on a big SELECT query which was the driver:

```
SELECT …
FROM PS_ITEM B,PS_ITEM_ACTIVITY F,PS_CUSTOMER U,PS_SET_CNTRL_REC V,PS_CUST_ADDR_EF_VW K,
PS_CUST_OPTION A, PS_RKA_CUST_CR_INF II
WHERE
  [$WHEREBU1]
  AND B.CUST_ID = U.CUST_ID
  AND U.SETID = V.SETID
  AND U.CUST_ID = K.CUST_ID
  AND U.SETID = K.SETID
  AND U.ADDRESS_SEQ_NUM = K.ADDRESS_SEQ_NUM
```

```
  AND V.SETCNTRLVALUE = B.BUSINESS_UNIT
  AND B.CUST_ID = II.CUST_ID
  AND U.SETID = II.SETID
  AND F.BUSINESS_UNIT = B.BUSINESS_UNIT
  AND F.CUST_ID = B.CUST_ID
  AND F.ITEM = B.ITEM
  AND F.ITEM_LINE = B.ITEM_LINE
  AND (F.ENTRY_AMT <> 0 AND F.ENTRY_AMT_BASE <> 0)
  AND U.CUST_STATUS = 'A'
  AND A.PYMNT_TERMS_CD <> '0000'
  AND B.BAL_CURRENCY NOT IN ('NLG','DEM','FRF','BEF','ESP','ITL','ATS','IEP','PTE')
  AND A.SETID = U.SETID
     AND A.CUST_ID = U.CUST_ID
     AND A.EFFDT =
        (SELECT MAX(B_ED.EFFDT) FROM PS_CUST_OPTION B_ED
        WHERE A.SETID = B_ED.SETID
          AND A.CUST_ID = B_ED.CUST_ID
          AND B_ED.EFFDT <= $rpt_asofdate)
  AND B.ASOF_DT <= $rpt_asofdate
  [$RKA_customer]
GROUP  BY  B.BUSINESS_UNIT,  B.CUST_ID,  B.PO_REF,  K.STATE,  U.NAME1,  K.CITY,  B.ITEM,
B.ITEM_LINE,
B.ASOF_DT, B.BAL_AMT, B.BAL_AMT_BASE, B.BAL_CURRENCY, B.CURRENCY_CD, B.POST_DT, B.DUE_DT,
B.ACCOUNTING_DT,   B.DISPUTE_STATUS,   B.DISPUTE_AMOUNT,   B.ENTRY_TYPE,   U.CUSTOMER_TYPE,
B.DUE_DAYS,
U.SETID, U.CUST_ID, U.CNTCT_SEQ_NUM, U.BILL_TO_FLG, B.ITEM_STATUS, II.RKA_CUST_SIZE,
II.RKA_CUST_PROFILE, II.RKA_CUST_RISK
ORDER BY [$order-by]
```

They key bottlenecks here were the NOT clauses: as already stated in this tutorial, NOT clauses suppress the use of indexes.
The following actions were indentified:

- The clause:
  ```
  AND (F.ENTRY_AMT <> 0 AND F.ENTRY_AMT_BASE <> 0)
  ```
  was the key row reducing element. Hence the query should start with the `ITEM_ACTIVITY` table (`F`). This was added as a `LEADING` hint.
- A small table containing all the currency codes `('NLG','DEM','FRF','BEF','ESP','ITL','ATS','IEP','PTE')` was created, called `PS_RKA_ARFOR_TMP1`
- All the customer related clauses (including `A.PYMNT_TERMS_CD <> '0000')` were separated into another query which inserted data into another temporary table called `PS_RKA_ARFOR_TMP`:
  ```
        insert into PS_RKA_ARFOR_TMP
                      select distinct cs.setid, cs.cust_id
                      from ps_customer cs, ps_cust_option cso
                      where cs.setid=cso.setid
                      and cs.cust_id=cso.cust_id
                      AND cso.PYMNT_TERMS_CD <> '0000'
                      and cs.setid <> 'CA100'
                      and cs.customer_type = 1
                      AND cs.CUST_STATUS = 'A'
                      and cso.effdt = (select max(B_ED.EFFDT)
                                        FROM PS_CUST_OPTION B_ED
                                    where cso.setid = B_ED.setid
                                    and cso.cust_id = B_ED.cust_id
                                    and B_ED.eff_status = 'A'
                                    and B_ED.effdt <= $rpt_asofdate);
  ```
- The final query then converted to:
  ```
  SELECT …
  FROM    PS_ITEM    B,PS_ITEM_ACTIVITY    F,PS_CUSTOMER    U,PS_SET_CNTRL_REC
  V,PS_CUST_ADDR_EF_VW K, PS_RKA_CUST_CR_INF II
  WHERE
  ```

```
  [$WHEREBU1]
  AND B.CUST_ID = U.CUST_ID
  AND U.SETID = V.SETID
  AND U.CUST_ID = K.CUST_ID
  AND U.SETID = K.SETID
  AND U.ADDRESS_SEQ_NUM = K.ADDRESS_SEQ_NUM
  AND V.SETCNTRLVALUE = B.BUSINESS_UNIT
  AND B.CUST_ID = II.CUST_ID
  AND U.SETID = II.SETID
  AND F.BUSINESS_UNIT = B.BUSINESS_UNIT
  AND F.CUST_ID = B.CUST_ID
  AND F.ITEM = B.ITEM
  AND F.ITEM_LINE = B.ITEM_LINE
  AND (F.ENTRY_AMT <> 0 AND F.ENTRY_AMT_BASE <> 0)
  AND B.ASOF_DT <= $rpt_asofdate
  and EXISTS (select 1 from ps_rka_arfor_tmp1 t1 where t1.bal_currency =
b.bal_currency)
  and EXISTS (select 1 from ps_rka_arfor_tmp t where t.setid = u.setid
and t.cust_id = u.cust_id)
  [$RKA_customer]
GROUP BY B.BUSINESS_UNIT, B.CUST_ID, B.PO_REF, K.STATE, U.NAME1, K.CITY,
B.ITEM, B.ITEM_LINE,
B.ASOF_DT,  B.BAL_AMT,  B.BAL_AMT_BASE,  B.BAL_CURRENCY,  B.CURRENCY_CD,
B.POST_DT, B.DUE_DT,
B.ACCOUNTING_DT,   B.DISPUTE_STATUS,   B.DISPUTE_AMOUNT,   B.ENTRY_TYPE,
U.CUSTOMER_TYPE, B.DUE_DAYS, …
```

The idea is to identify the bottlenecks by looking at the plan, then tweaking (building statistics, indexes, adding hints). Afterwards checking the plan once again, or running the query / process and checking if the tweaks took effect. If not, checking the plan once again and trying to work backwards why the changes have not taken place, and making more tweaks to 'force' Oracle to accept the suggestions.

As a result of this tuning, the processing time came down from 20 minutes to 5 minutes. Since there were a total of around 20 processes executed daily for difference business units, **the total saving was nearly 5 hours per day.**

Lessons learnt:

At the end, we come to the lessons that we learnt, performing performance tuning activities at Rockwell Automation for more than 3 years:

- **Start out with low hanging fruit**
  When we started out tuning efforts with Rockwell, we looked at processes that were crunching more data than needed and added date clauses to the critical processes. For example, REQSORC process (requisition sourcing) need not look at ALL requisitions – more than 2 years old requisitions can simply be ignored.

  In addition, queries can be modified as under:

  Original
  ```
  UPDATE PS_CIP_CTRLQ
   SET CIP_MSG_STAT = '02' WHERE PROCESS_INSTANCE = #process_id;
  ```

Modified
```
  UPDATE PS_CIP_CTRLQ
  SET CIP_MSG_STAT = '02' WHERE PROCESS_INSTANCE = #process_id and
  CIP_MSG_STAT = '01';
```

With the first query Oracle feels the need to modify ALL the rows in the table, while with the second, only those having value ‘01’ are modified. Hence with the second version, rows already having value ‘02’ are spared.

- **Go after the FULL SCANs**
  If you have a system that doesn’t perform very well, and the previous step has already been completed, then go after the FULL SCANs. Pick up queries that do FULL SCAN one by one, and tune them. **The timing different may not be much, even after removing the FULL SCAN, but the overall impact on the system will be huge, due to the much reduced IO operations.** The disk is the slowest part of the chain, remember. To check the FULL SCANS going on now, run the SQL

```
SELECT target, a.sid, time_remaining, b.username, b.program, opname
FROM v$session_longops a, active b
WHERE time_remaining>120 AND a.sid=b.sid(+)
```

- If tape backups are happening then the tape drive can be a bottleneck. With the RA system, there is a single tape drive shared across PROD and QA. Hence, if both the backups are running together, then PROD performance would go down.

- **Reschedule activity**
  This is a workaround in cases where performance of the query itself is difficult to improve. If a job with a bad impact on performance can wait till the weekend, better go that route. If it can run during a time when the system is underutilized go for it. If two IO intensive jobs can run sequentially rather than parallely go for it.

  On those same lines, if there are two jobs that run almost simultaneously, it may be prudent to move one of them five minutes above or below the other.

- **Deadlocks**
  Deadlocks are the last bane of performance tuning in a production system. Oracle takes some time in detecting deadlocks, and killing the offending process. During this timeframe your database queue can spiral.
  These are quite easy to fix: the offending queries are present in the database log. Hence, pick up the queries, search the processes to find where these are running and redesign suitably. Deadlock will go away if the processes **do not try to process the same set of rows at the same time.**

“Performance tuning is like cleaning your room: you have too keep doing it every now and then in a production system.”

http://blog.hardeep.name

Please refer to the following link for more information on performance tuning. There is a video tutorial also available:

http://blog.hardeep.name/general/20090711/app-tuning/

INDEX