

Top 7 Oracle Database Tuning Techniques and Case Studies

PROCASE Consulting
Feb 13, 2002

Lev Moltyaner
Managing Partner
lmoltyaner@ProcaseConsulting.com

Introduction

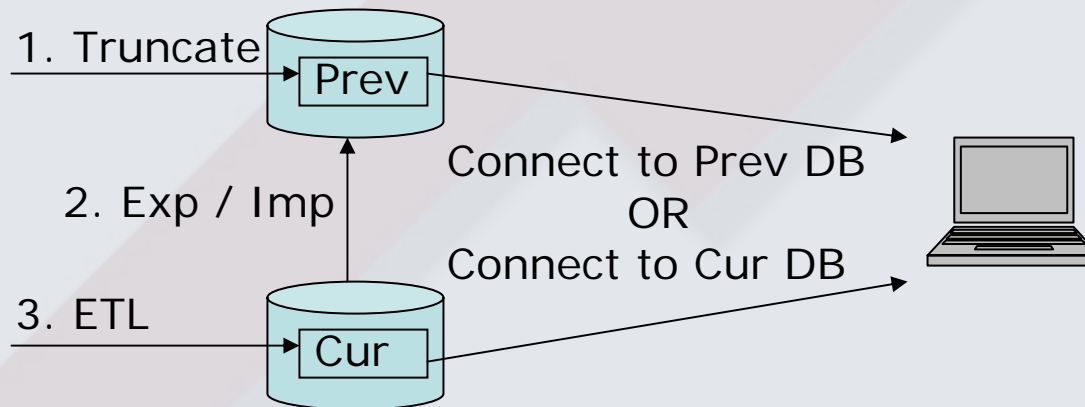
- Tuning experience is difficult to share because it is often application specific
- In this session:
 - We will define 7 practical tuning techniques
 - Present a case study of how each technique was applied
 - Quantify performance gains and offer recommendations
- We will also illustrate how these techniques represent a tuning methodology which can be applied to any database
- We hope that each attendee will be able to identify immediate benefits for their databases

1. Optimizing Process Flow

- First step in any tuning exercise is to optimize the process flow
 - Review the process design to identify steps which can be combined or eliminated
- Identify alternatives which were not considered during development
 - Possible because the original design may have inherited legacy process flow or did not have a complete picture

Case Study #1: Definition

- Application: A data mart to analyze company's order-inventory mapping for a rolling two day window
 - Access: current day 85%, previous day 15%
- Requirement: Tune the nightly ETL (extract, transform, load) process to populate current day mappings
- Original process architecture:



Case Study #1: Optimization

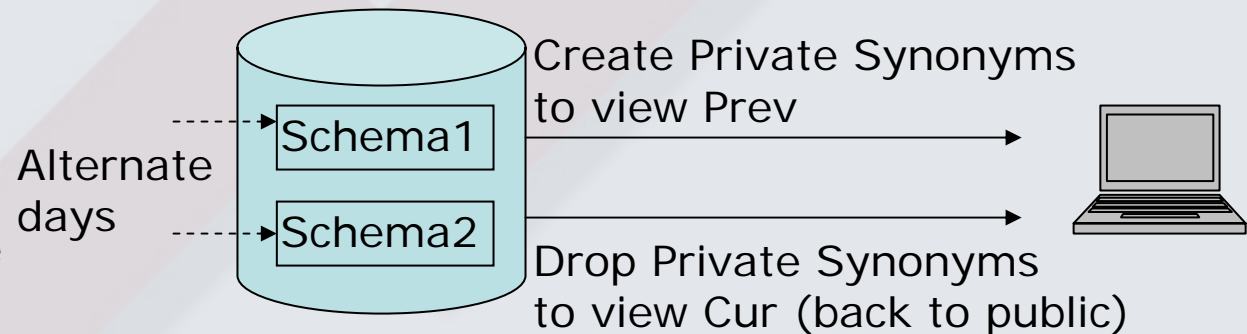
- First optimization is to combine two databases into one
 - Improved memory utilization, fewer background processes
- Must be achieved without changing the front-end code
 - Utilize a combination of private and public synonyms
- Implementation:
 - Each day, create public synonyms for current day
 - Front end has buttons to switch between current and previous day
 - Previous button creates private synonyms
 - Current button drops private synonyms

Case Study #1: Optimization

- Second optimization is to eliminate the Exp/Imp process
 - Table drive the schemas by creating a control table to re-define the current schema each day

New Processes

1. Truncate Prev
2. ETL into Prev
3. Switch Cur and Prev in Control Table
4. Switch public Synonyms to new Cur



Case Study #1: Results

- Improved performance of ETL process by 1.5 hours by eliminating the export/import step
- Improved performance of overall application due to increased memory utilization and elimination of background processes

2. Utilizing Intended Indexes

- Next tuning step is to check if all intended indexes are utilized
- Use TKPROF to examine statistics of slow processes
 - Sort TKPROF results by CPU time
 - Review access paths of slowest SQL statements

Case Study #2 : Definition

- Application: A mutual fund management system
- Requirement: Tune the end-of-day account balancing process which was taking 26 hours during the production rollout
- TKPROF results indicated that a simple SELECT used up more than 80% of total process CPU time:

```
SELECT ... FROM transaction_history  
WHERE member_no = v_member_no ...
```
- TKPROF also identified the access path as a full table scan
 - Yet, transaction_history has an index on member_no column
- Why was the index not used?

Case Study #2: Optimization

- The index was not used because of implicit conversion
 - Implicit conversion is always away from strings
 - VARCHAR2 or CHAR -> NUMBER or DATE
- member_no column was declared as VARCHAR2(10) in the database, while v_member_no was declared as NUMBER(10) in the program
 - TO_NUMBER function was applied on member_no column to convert away from a string
 - A function on an indexed column disables the index
- The process was tuned by fixing the declare statement
`v_member_no transaction_history.member_no%TYPE;`

Case Study #2: Results

- Performance improved from 26 hours to 4 hours
- Recommendations:
 - When constructing WHERE conditions, analyze impact of functions on indexes
 - Implicit functions, NVL, UPPER, etc
 - Create function based indexes if necessary
 - During data modeling, do not give VARCHAR columns names which may indicate that they are numbers
 - Use %TYPE to declare variables based on table columns
 - Use TKPROF to identify the problem quickly
 - 98% of performance problems is in 2% of the code

3. Creating Specialized Objects

- Next step is to check if the most appropriate Oracle performance objects are utilized
- Normally B-tree indexes are the best choice because they ensure good performance for a wide variety of queries
- However, other Oracle objects can significantly improve performance in specific situations
- One such object is a table cluster
 - Use only if majority of queries need to join the tables
 - Reduces performance of full table scans
 - In one case, clustering GL Transactions and Report Definitions tables, improved performance of a set of 7 financial reports by 30% (used for 300+ electric utilities)
- Another performance object is a bitmap index

Case Study #3 : Definition

- Application: Human Resource Management System
 - Main table is 19 years of payroll history
 - 54M records or 0.7M records / quarter
 - Average row length is 200 bytes for approx 10G
 - Indexes on date and on person ID are 1G each
- Requirement: Tune performance of queries on payroll history table
- First, analyze the queries that reference the payroll table
 - 98% of queries include a date range
 - Day, week, month, quarter, year, 3 years, 5 years
 - Other 2% read the entire history for a specific person

Case Study #3: Optimization

- An obvious conclusion was to implement partitioning
 - However, the client did not have a license and an upgrade was cost prohibitive
- Instead, we added a fiscal week and a fiscal quarter columns and created a bitmap index on each column
 - Each index only took 18M
 - Compared to 1G B-Tree index
 - The date index was dropped
 - Queries were enhanced to include the new columns

Case Study #3: Results

- Benchmark of a simple query (on average 3-5 times faster)

Period	Week/Quarter Bitmap Indexes	Date B-tree Index	Factor
1 Week	0:05	0:05	1.0
1 Month	0:15	0:35	2.3
1 Quarter	0:50	2:33	3.1
2 Quarters	1:20	4:10	3.1
1 Year	2:33	8:40	3.4
3 Years	6:10	17:27	2.8

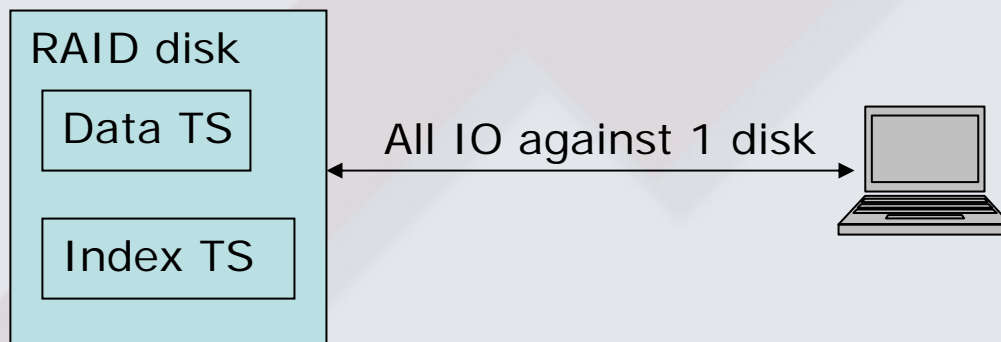
- Full table scan took 30 min
 - Bitmap index was faster for up to 75% of rows
 - B-tree indexes are normally faster for up to 5-25%

4. Reducing IO Contention

- Next critical area for tuning is IO
 - Improved IO throughput will improve performance of the entire application
- Most critical for applications which are IO bound
 - Applications which do a lot of IO
 - Machine has many CPUs but not enough disks

Case Study #4 : Definition

- Application: Human Resource Management System
 - All data on RAID to ensure data redundancy
 - Main table is payroll history
 - Populated once a week by a payroll interface
 - 6 CPUs, one RAID, 2 non-redundant Jamaica disks
- Requirement: Tune overall application performance

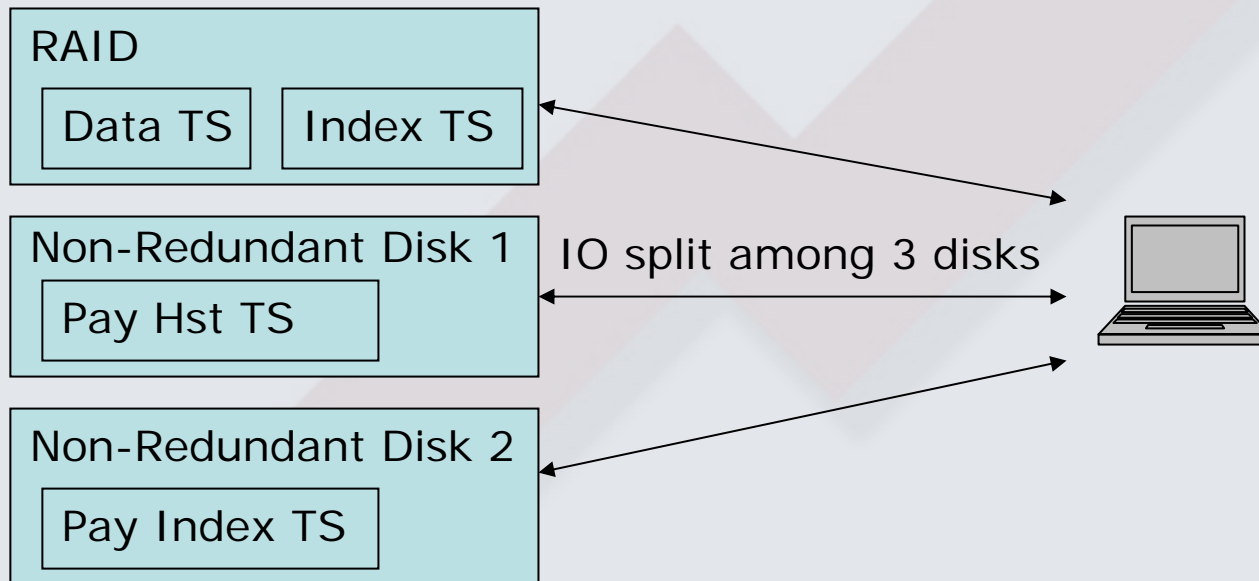


Case Study #4: Optimization

- No CPU intensive processing but a lot of queries
 - Machine has many CPUs: not likely CPU bound
 - All data is on one RAID: likely IO bound
 - One disk creates a bottleneck
- Key to tuning IO is to recognize special data characteristics and to take advantage of them
 - Many application queries are against payroll history
 - Moving it to a new disk will greatly reduce contention
 - Payroll history is not updateable and new records are inserted only once a week
 - Can be placed on a non-redundant disk while handling redundancy manually

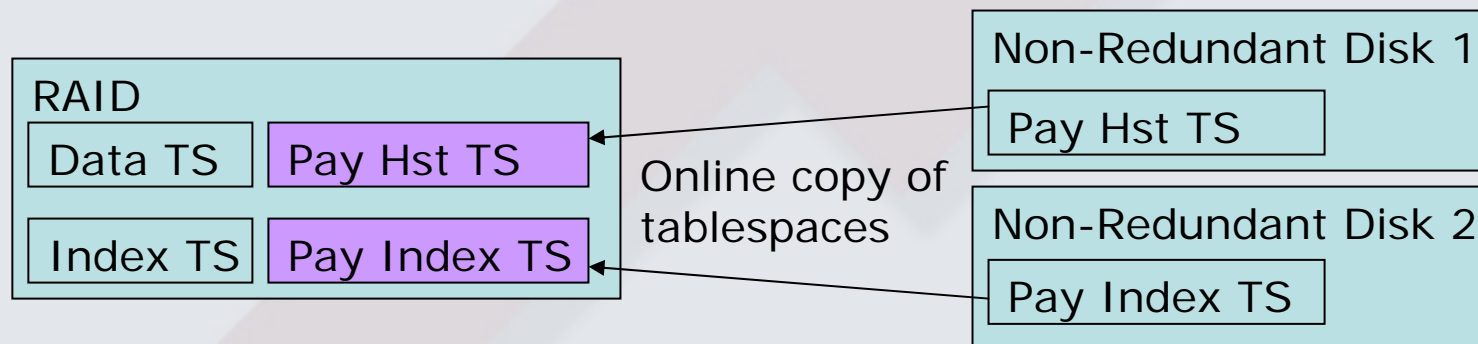
Case Study #4: Optimization

- Reduce contention by moving payroll history to one non-redundant disk and its indexes to another
 - Will split IO load between 3 disks



Case Study #4: Optimization

- Implement redundancy by copying payroll history and its index tablespaces to RAID
 - Payroll history is updated once a week, then tablespaces are set to READ ONLY and then copied to RAID
 - If a non-redundant disk fails, control file is changed to point to the tablespaces on RAID and the database is restarted within a few minutes



Case Study #4: Results

- Overall system response time improved by 10-20%, surpassing management and user expectations
 - Achieved because application was IO constrained while CPUs were under utilized
- Can often do a lot with existing hardware

5. Coding Complex SQL vs PL/SQL

- Next recommended area of tuning is to review processes which under utilize the power of SQL
- Many Oracle developers think procedurally
 - Simple queries retrieve data which is then manipulated within a process
- Even the most efficient process cannot perform as well as SQL most of the time
 - Context switches are very inefficient
 - Significantly increases network traffic if process is not on the database server

Case Study #5 : Definition

- Application: Oracle Payroll Implementation
 - Oracle Payroll tables are very normalized
 - Best way to support customization
 - However, this implies large data volumes
 - In our example, the main 2 tables have 130M and 550M records
 - Run results (21G) and run result values (44G)
- Requirement: Tune the process which extracts employee information from Oracle Payroll tables

Case Study #5 : Definition

- To simplify the example, consider extracting from 3 tables, each with person id (pid), start date (sd), end date (ed)
 - End dates are derived in triggers
 - Thus, there are no gaps or overlaps in dates
 - Primary key of each table is pid and sd
- The result should have a row for each period created by a record in any of the 3 tables
 - The result should only contain periods for which there is data in ALL 3 tables
- In fact, such merging of multiple tables is a common business requirement

Case Study #5: Optimization

1. The easiest solution is a "brute force" method
 - Procedure with a cursor loop to read each row from one table and 2 queries within the loop to join to the other tables
 - This is the easiest solution, but it is inefficient because the queries within the loop are executed many times
2. A more efficient solution is an algorithm which minimizes I/O by performing a sort-merge join within PL/SQL
 - It defines a cursor per table and orders the rows
 - Then the code synchronizes the rows from the three cursors by using the least and greatest functions on dates to control which cursor(s) to fetch next
 - This is very efficient because there are only 3 full table scans

Case Study #5: Optimization

3. The best solution is a single SQL statement

```
select t1.pid person_id
      ,greatest(t1.sd, t2.sd, t3.sd) start_date
      ,least(t1.ed, t2.ed, t3.ed) end_date
      ,t1.x, t2.y, t3.z
from t1, t2, t3
where t2.pid = t1.pid
and t3.pid = t1.pid
and greatest( t1.sd, t2.sd, t3.sd )
           <= least( t1.ed, t2.ed, t3.ed );
```

Case Study #5: Results

- Most people would not think of the SQL solution because they think of one row at a time versus sets of rows
- Once you make the paradigm shift to sets of data, SQL solution is easier to develop and maintain
 - Few lines of code versus hundreds
- Most importantly, it performs significantly better than the best PL/SQL solution:

Solution	Time
Original Brute Force PL/SQL	65 min
Most efficient PL/SQL	6 min
SQL	1.5 min

6. Reducing I/O with PL/SQL tables

- Next recommended area of tuning is to review processes which generate unnecessary I/O
- Check if a process
 - Reads the same data many times
 - Updates the same rows multiple times

Case Study #6 : Definition

- Application: Auto-insurance data mart
 - Source tables had a 4 year history of policies and claims
 - Ten tables 1-10M rows each
 - Transformation and summarization process categorized claims into 150 segments and for each segment further into sub-segments
 - Segments were derived based on source data such as vehicles, operators, and coverages
- Requirement: Tune the monthly transformation and summarization process which was taking 44 days
 - Service Level Agreement (SLA) was 15 days

Case Study #6 : Optimization

- Process had to be redesigned because of a number of significant design flaws
 - Re-read source data to derive each segment
 - Updated target data multiple times for each policy
 - Implemented in PowerBuilder which generated unnecessary network traffic
- First we created a view to union the source tables
 - Each source table was assigned a record type
 - Sorted by policy number, change date, and record type
- The main process looped through this view
 - One full table scan of each source table
 - Eliminated multiple reads of source data

Case Study #6 : Optimization

- The procedure included a set of PL/SQL tables to store the state of a policy
 - Policy, operators, vehicles, coverage, and claims
- Using the state tables, the procedure then derived which segments the policy belonged to during each period
 - Segments were stored in a PL/SQL table in a single string of 30 mutually exclusive characters
 - Multiple segments mapped to each character
- Once a new policy was encountered, one bulk insert moved the policy and its segments from PL/SQL tables into an actual table
 - Eliminated multiple updates of same records

Case Study #6 : Results

- The new process ran in 42 hours
 - Most of the gain came from reduction of reads and writes
 - Achieved by using PL/SQL tables

7. Multi-Threading Processes

- Our final tuning technique is to split processes into multiple concurrent sub-processes
 - Called multi-threading a process
 - Simplest, yet most effective
 - Most effective on machines with multiple CPUs
 - Performance improves linearly with each CPU
- Can be used on any process which can be divided into multiple processes without having to rewrite it
 - Particularly easy to do for programs with an outer loop to process records from a cursor
 - Achieved by adding a condition in the query to do a range of records rather than all

Case Study #7 : Definition

- Application: Oracle Payroll Implementation
 - 12 CPU machine
- Requirement: Tune the process which uses Oracle APIs to load weekly time cards into Oracle Payroll tables

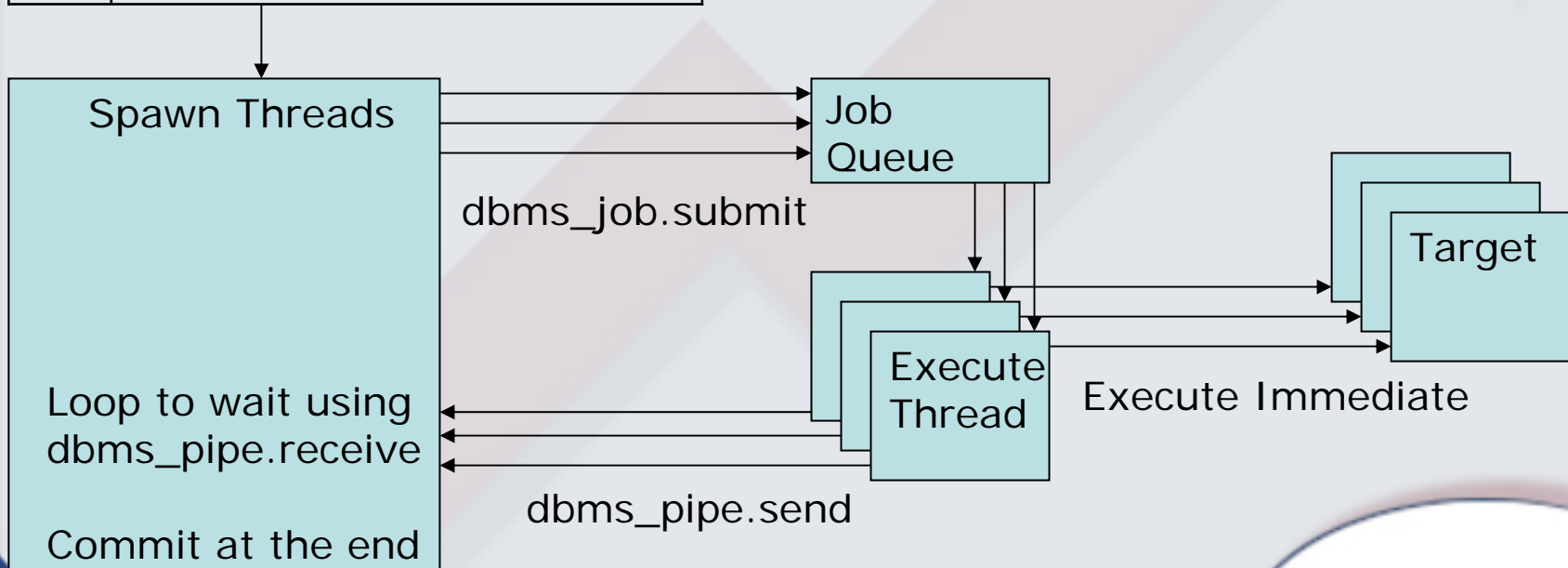
Case Study #7 : Optimization

- Created a generic multi-threading algorithm, made up of two procedures
 1. Spawn Threads
 - Input is a PL/SQL table with one record per thread
 - Contains thread number and a call to the target procedure with all the parameters
 - Uses `dbms_pipe` to open a pipe to monitor all threads
 - Uses `dbms_job.submit` to spawn each thread
 - Inserts Execute Thread procedure into job queue
 - Loops using `dbms_pipe.receive` to wait for all threads to finish
 2. Execute Thread
 - Calls the target procedure using Execute Immediate
 - Uses `dbms_pipe.send` to return success or failure

Case Study #7 : Optimization

Call Spawn Threads (pass PL/SQL table)

Thr	Call target with ranges
1	Target(p1, ..., px, 1, 100)
2	Target(p1, ..., px, 101,200)



Case Study #7 : Results

- The generic nature of this algorithm allows it to be used for any process
 - Minimum changes to the target process
- The performance gains are almost linear
 - By using 8 threads the load process was tuned from 70+ minutes to 10 minutes

Summary: Tuning Methodology

- These scenarios define a simple but effective methodology

Design	Eliminate inefficiencies in system architecture and process flow
SQL	Ensure that intended indexes are utilized Create most effective Oracle objects (bitmap indexes, clusters, index organized tables)
Database	Reduce IO contention (IO load balancing)
Processes	Decrease PL/SQL context switches (complex SQL) Increase memory utilization (PL/SQL tables) Increase CPU usage (multi-threading)

- We hope that you can apply this methodology to your databases!