Ask Us Anything™ Series

- July 20, 2011
- OMG! Identifying and Refactoring Common SQL Performance Anti-patterns
  - Jeff Jacobs, Jeffrey Jacobs & Associates, LLC
Welcome

Subject Matter Expert – Jeff Jacobs

Moderator – Faun deHenry

Technical Support – Lori Hayes
Webinar Guidelines

- No company specific information will be presented
- Time permitting we will address all inquiries in the webinar timeframe
- Questions not addressed during the webinar may be submitted to aua@fmtsi.com, we will respond within 48 hours
Little Bit About Jeff!

- 20+ years experience in Oracle and software engineering
- Consultant to numerous companies
- Board of Directors, IOUG Oracle Exadata SIG
- ODTUG Board of Directors for 16 years
  - President
  - Conference Chair
- Trained over 3000 students in Oracle products, database design, modeling and methodology
- Presented at Oracle Open World, ODTUG Kaleidoscope, Collaborate, NoCOUG and various other local user groups
The following is intended to outline Jeff Jacobs impressions on the SQL Anti-patterns and perhaps deliver a few techniques to assist your endeavors.

This presentation is not a commitment to deliver any material, code or functionality, and should not be relied upon in making purchasing decisions.

It is intended for information purposes only, and does not commit FMT Systems Incorporated of Texas, The Oracle Applications Users Group, Oracle Corporation or any other corporate entity.

*The views and opinions expressed in the presentation and webinar pages are those of the page authors only.*
OMG! Identifying and Refactoring Common SQL Performance Anti-patterns

Jeff Jacobs
Jeffrey Jacobs & Associates
jmjacobs@jeffreyjacobs.com
Qualifications

• 2 centuries of experience with Oracle, as consultant and trainer
Survey Says

• DBAs
• Developers
• Architects
• Heavily non-Oracle development shop
• Concerned with performance
• Access to production size database
• Easy access to running traces, Enterprise Manager
Introduction to OMG Method

• OMG Method focuses on
  – Refactoring SQL
  – Indexing
  – Refactoring application side code
• OMG Method targets performance problems created by Developers Inexperienced in Oracle technologies (DIO)
• OMG Method requires (almost) no DBA privileges other than indexing
  – No tracing, (almost) no init.ora parameter changes
Fair Warning

• No demos
• No “proofs”
• Quick fixes
Requirements for SQL Performance Heroes

• Good SQL fundamentals
• Able to read basic explain plans
• Understand basic performance statistics from autotrace
• Courage to make and test changes
  – *Don’t take my word for it!*
• Willingness to work with and educate DIOs
Why OMG Method

• Vast majority of performance problems are result of DIOs’
  – Lack of training in SQL and Oracle
  – Lack of interest in SQL and Oracle
  – Misinformation about SQL and Oracle performance
  – Resistance to PL/SQL
  – Focus on OO, procedural and functional programming techniques
    • Iterative thinking vs set thinking
Anti-Patterns

• **Definition**
  – *Common SQL or design practice that results in poor performance*

• OMG Method identifies common anti-patterns and techniques to fix them
  – *Always verify that OMG fixes actually improve performance*

• OMG Method does not address schema design problems
  – No changes to tables or columns
Understanding Common Design and DIOs

Anti-patterns

• Overly Generic Data Models
  – OBJECT, INSTANCE, ATTRIBUTE, ATTRIBUTE_VALUE structures

• Fat, Unnormalized Tables
  – Often with in-line CLOBs

• Fear of Joins
  – “Joins are to be avoided at all costs” mentality

• Failure to Understand SQL query cost in application code

• *Iterative vs Set World View*
Understanding Common Design and DIOs Anti-patterns

• Unmanaged Surrogate Primary Keys
  – (Nearly) all tables have surrogate primary keys
  – Values for same row is not consistent across environments, e.g., COMPANY_ID value for same company differs across production, development, test environments
  – Typically use additional lookup columns

• Widespread use of Dummy values instead of NULL
  – DIOs uncomfortable using NULL
  – Misunderstanding of performance issues with NULL
Understanding Common Design and DIOs Anti-patterns

- “Indexed searches are always better”
- Lack of documentation
Avoid Dynamic SQL

- Avoid/eliminate dynamic SQL, e.g. creation and execution of SQL queries created by concatenating strings
  - Particularly problematic when using literals for constants
- Use prepared statements with bind variable
- Dynamic SQL results in heavy parsing overhead and SGA memory usage
  - Child cursors may be created even if the only differences between SQL queries is literal values
  - Potential for SQL Injection
Inline Views

• In SQL code, an inline view is a subquery used in place of a table, e.g.,

```sql
SELECT ... FROM
  (SELECT ...)
...
```
Avoid/Replace Materialized Inline Views

• Inline views typically results in an “inline view” being created in the execution plan
  – Referred to as *materialized inline view* (MIV)
• Oracle may also *merge* the SQL inline view with the outer query
• MIVs produce a *result set*, e.g., a temporary table (not to be confused with *Global Temporary Table*)
  – MIVs are never indexed
  – Joins with a MIV effectively perform a Full Table Scan (FTS) against the MIV, e.g. *multiple FTS*!
    • Poor performance if result set is large
Avoid/Replace Materialized Inline Views

• DIOs frequently write inline views which can and should be replaced by joins
  – Generally can be done with little or no understanding of underlying schema semantics
  – Try /*+ MERGE */ hint first; generally doesn’t improve performance, but worth trying
    • May also help in rewrite
Merged Inline Views

• As the Cost Based Optimizer has evolved, it frequently *merges* SQL inline views with the outer query
• Frequently not a performance improvement!
  – Particularly with poorly written SQL inline views
  – 10G’s merging is much better than 9i’s
  – 11G’s should be even better
• Try /*+ NO_MERGE */ hint
Never Update Primary Key Columns

• Primary key (PK) columns should never be updated, even to current value
• Common DIO approach is to update all columns in a row
• Updating PK columns forces examination of referencing foreign key (FK) constraints on child tables
  – General performance issue, even if FK columns indexed
  – Results in FTS if FK columns not indexed
Avoid/Remove Unnecessary Outer Joins

• DIOs frequently add outer joins “just to be safe”
• Outer joins may be expensive, limiting CBO choices
  – Be sure join columns are indexed
• Work with developer or end user to determine if outer join is needed
EXISTS \textit{vs IN}

- Replacing \textit{IN} with \textit{EXISTS} often produces dramatic performance improvement
- \textit{IN} uses \textit{uncorrelated} subquery
- \texttt{SELECT} ...
  
  \begin{verbatim}
  FROM table_1 outer
  WHERE
  outer.col_1 IN
  (SELECT inner.col_1
  FROM table_2 inner
  [WHERE ...])
  \end{verbatim}
IN Performance Issues

• IN may perform poorly
  – Produces result set, effectively a materialized inline view
    • CBO may replace IN with EXIST; verify via execution plan
  – Result set is unindexed
  – Result set is scanned for every row in outer query
  – Large result set is well known performance killer

• IN should only be used when the result set is small

• Note that if the value of outer.col_1 is NULL, it will never match the result of the IN
  – Use NVL on both the inner and outer columns if NULL must be matched
**EXISTS vs IN**

- DIOs seldom know how to use EXISTS as it involves a *correlated subquery*, e.g., a join between column(s) in the outer and column(s) in the inner query.
- Replace the uncorrelated subquery with a subquery by joining the outer column from the IN clause with an appropriate column in the subquery.
**EXISTS Correlated Subquery**

- `SELECT ...`
  - `FROM table_1 outer`
  - `WHERE EXISTS (SELECT 'T' -- use a simple constant here FROM table_2 inner WHERE outer.col_1 = inner.col_1 [AND ...]) -- WHERE predicates from original query`
EXISTS Correlated Subquery

- The join columns (inner.col_1 in example) from the table in the correlated subquery should be indexed
  - Check to see if appropriate indexes exist; add them if needed
- Use a constant in the SELECT of the correlated subquery; do not select the value of an actual column
  - NULL works as “constant”, but is very confusing
- Note that SELECT DISTINCT is unnecessary for both IN and EXISTS
Subquery Factoring using WITH

- Very powerful (and virtually unknown)
- Many DIO written queries use *identical* subqueries/inline views repeatedly
- Often lengthy UNIONs
Often lengthy UNIONs

```sql
SELECT ...
FROM
  table_1,
  (SELECT ...
    FROM table_2, table_3, ...
    WHERE table_2.id = table_3.id)
  IV
WHERE ...
UNION
SELECT ...
FROM
  Table_4,
  (SELECT ...
    FROM table_2, table_3, ...
    WHERE table_2.id = table_3.id)
  IV
WHERE ...
UNION ...
```
Performance Issue

• Oracle’s CBO is not aware of identical nature of subqueries (unlike programming language optimizers)
  – Executes each subquery
  – Returns distinct result set for each subquery
  – Redundant, unnecessary work
Subquery Factoring

• Subquery factoring has two wonderful features
  – Generally results in improved performance
  – *Always* simplifies code
    • *Factored subquery* only appears once in the code as a *preamble*
      – Referenced by name in main query body
    • More readable, easier to maintain and modify
Syntax

/* Preamble, multiple subqueries may be defined */
WITH
    pseudo_table_name_1
    AS (SELECT ...)
    [, pseudo_table_name_2 ... AS (SELECT ...)]
    /* Main query body */
    SELECT ...
    FROM pseudo_table_name_1 ...

... -- typically UNIONS
Example

- Applying this to the example
  
  /* Preamble */
  WITH
    IV AS
      (SELECT ... FROM table_2, table_3, ... WHERE table_2.id = table_3.id
       /* Main query body */
      SELECT ...
      FROM
        table_1, IV -- IV is pseudo table name
      WHERE ...
      UNION
      SELECT ...
      FROM
        Table_4, IV
      WHERE ...
      UNION ...
Factoring Options

- Oracle may perform one of two operations on factored subqueries
  - **Inline** – performs textual substitution into main query body
    - Effectively same query as pre-factoring
    - No performance improvement due to factoring
    - Still more readable
  - **Materializing** factored subquery
    - Executes the factored subquery only once
    - Creates true temporary table (not Global Temporary Table)
    - Populates temporary table with direct load INSERT from factored subquery
Materialized Factored Subquery Issues

- Materialized Factored Subqueries (MFS) issues
- CREATE TABLE for temp table at least once (on 1st execution)
- Multiple tables may be created if query executions overlap
- Tables are reused if possible
- Recursive SQL performs INSERT
- Data is written to disk
- Doesn’t always result in performance improvement
Hints for Subquery Factoring

• /*+ Materialize */ will force materializing
  – Seldom, if ever, needed
  – Oracle only materializes when subquery used more than once (but verify)
• /*+ Inline */ will force textual substitution
  – Use when materializing does not improve performance
• Other hints may be used in factored subquery, e.g. INDEX, etc.
  – Note that MERGE and NO_MERGE may be combined with INLINE
• Hint follows SELECT in factored subquery
  – WITH (SELECT /*+ hint */ ..) AS …
INDEX Hints

- DIO often believe everything should use indexes
- Frequent use of *unqualified INDEX* hint, e.g., only table name specified, but no index
  - `SELECT /*+ INDEX (table_name) */`
  - Yes, this does work!
- Oracle will always use an index, no matter how bad
  - Unclear which index will be used; documentation says “best cost”, but unclear if true; experience suggests 1st in alphabetical order
  - Further complicated by poor indexing
- Fix by either
  - Qualifying hint by specifying index name(s)
  - Removing hint entirely
    - Removing the hint often improves performance
Constant Data Conversion Issues

• When comparing a VARCHAR2 (or CHAR) column to a constant or bind variable, be sure to use string data type or conversion function
• Oracle does not always do what you would expect
  – WHERE my_varchar2_col = 2
    does not convert 2 to a string!!!
  – Converts every rows’s my_varchar2_col to a number for the comparison
    • Generally results in FTS
    • Common cause of “I just can’t get rid of this FTS”
• Common problem with overloaded generic and OO models
• Be aware of other type implicit type conversion functions, e.g. DATE and TIMESTAMP!
• Even SQL Performance Heroes get bit!!!
Eliminate Unnecessary *Lookup* Joins

- Tables with unmanaged surrogate keys typically have *lookup/alternate key* column(s) with consistent data across environments
  - Very common with generic and OO models
- Typical code is:
  ```sql
  SELECT
    FROM child_table, reference_table
  WHERE
    child_table.reference_table_id = reference_table.reference_table_id
    and reference_table.lookup_column = 'constant'
  ...  
- Results in access to reference_table for every applicable row in child_table
Eliminate Unnecessary *Lookup* Joins

- Even worse when UPPER/LOWER function applied to lookup_column (unless appropriate functional index exists)
- Replace with scalar subquery
  ```sql
  SELECT
  FROM child_table
  WHERE
  child_table.reference_table_id =
  (SELECT reference_table_id
  FROM reference_table
  WHERE
    reference_table.lookup_column = 'constant')
  -- Only performs scalar subquery once
  ```
Improving Pagination

- *Pagination* refers to returning row *n* through *m* from an ordered result set using ROWNUM
  - Typically for data on a web page or screen
- Common, worst case code:
  ```sql
  SELECT t1.col_1, ...
  FROM
  (SELECT *
   FROM table_1
   WHERE ...
   ORDER BY ...) t1
  WHERE
  ROWNUM between *n* and *m*
  ```
Improvement Steps

1. Replace literals with bind variables
2. Replace "*" in innermost inline view with desired columns
   - Potentially reduces unnecessary I/O and sort processing
3. Refactor the query so that inline view only returns 1\textsuperscript{st} \textit{m} rows and use \texttt{/*+ FIRST_ROWS */} hint (per Tom Kyte’s \textit{Effective Oracle by Design} on \textit{Pagination with ROWNUM })
Improvement Step #3

SELECT *
FROM
(SELECT /*+ FIRST_ROWS */
  ROWNUM AS rnum, a.*,,
FROM
(SELECT t1.col_1,...
  FROM table_1
  WHERE ...
  ORDER BY ...) a
WHERE
  ROWNUM <= :m)
WHERE rnum > = :n
**Improvement Step #4**

- Replace the columns in innermost inline view with ROWID and join to table in outermost query
  - May provide substantial I/O performance improvements on fat tables, particularly those with inline CLOBs
Improvement Step #4

SELECT t1.col_1,...
    FROM
    table_1,
    (SELECT /*+ FIRST_ROWS */
        ROWNUM AS rnum, inner_row_id
    FROM
        (SELECT ROWID inner_row_id -- innermost query
            FROM table_1
            WHERE ... ORDER BY ...)
    WHERE
        ROWNUM <= :m)
WHERE rnum >= :n
AND table_1.ROWID = inner_row_id
UPDATE and DELETE Performance

- “I’m DELETEing/UPDATEing a few rows. It’s virtually instantaneous when I test it in my development environment, but takes a very long time in production!” – Joe the DIO

- Check for indexes on FK constraint columns of child tables.
  - Lack of indexes on FK constraints requires an FTS of each child table for each row to be DELETEEd/UPDATEEd in parent table
  - Common problem with history tables

- Add appropriate indexes
UPDATE and DELETE Performance

• Look for foreign key constraints using Cascade Delete
  – Hierarchy of cascade deletes can result in very poor performance
  – Unclear if circular references ever complete
• Beyond scope of OMG
  – Application code may depend on existence of Cascade Delete
  – Quick fix may be temporarily altering constraints
Eliminate *Bitmap Conversion to/from ROWIDs*

**Execution Plan Step**

- Known performance problem unrelated to existence/use of bitmap indexes
- May be *huge* CPU hog
- `alter session set "_b_tree_bitmap_plans" = false`
- Default in 9i and 10G is *true*
- Real fix requires setting `init.ora` parameter
  - `_b_tree_bitmap_plans = false`
  - Requires database restart; can’t be set dynamically
Add Indexes on Foreign Key Constraints

• FK constraints should always be indexed
  – Have not yet seen exception to this rule (but always interested)
• Primary performance gains
  – Improved join performance – fundamental feature of CBO
  – UPDATE and DELETE performance
  – Oracle apparently still performs table level locks, despite statements to contrary
Add Foreign Key Constraints

• “FK constraints hurt performance. We’ll enforce referential integrity (RI) in the application” – Flo the DIO
  – Translation: “We won’t make any mistakes in the application code”
  – Won’t really verify RI in the application
    • True verification would result in worse performance
• *It doesn’t matter how well the system performs if the data is corrupt!* 
  – Earned big $ as expert witness demonstrating issues with lack of FK constraints
• CBO uses existence of FK constraints
• Adds to effective documentation of system
Eliminate Redundant Indexes

- Redundant indexes, e.g., indexes with identical leading columns
  - Common DIO anti-pattern
- Impacts INSERT/UPDATE/DELETE performance
- Confuses CBO
  - Unclear how CBO selects index when two (or more) have needed leading columns, but different trailing columns
- Rules of thumb
  - Eliminate index with most trailing columns
  - Indexes with more than 3 columns are suspect
  - PK indexes with trailing columns should be reduced to PK only
Reduce Unnecessary and Redundant Queries

• Worst real world case
  – 80,000 individual queries from application takes 3+ hours
  – Single query took under 30 seconds
• Individual query is not performance problem
  – Total number of queries is problem
• Two general cases
  1. Iteration
     • DIO issues large number of SELECTs, typically performing join, calculations or sorts in application
     • Generally easy to replace with single query
  2. Redundant Queries
     • DIO issues same query repeatedly for unchanging data, typically refreshing page/screen, i.e., field label
     • Requires changes to application code structure
       – Not usually Hero’s domain
Add Appropriate Functional Indexes

• Functional indexes (FI) are great quick fixes for many anti-patterns
• Two most common anti-patterns
Mixed case string columns

- Column contains mixed case data used for both lookup/filtering and display
  - Good design would be two columns, one for lookup and one for display
- (Somewhat) knowledgeable DIO uses UPPER(column_name)
  - Less knowledgeable use LOWER(column_name)
- Add appropriate index(es)
  - If possible, standardize queries to use one function
  - May need to add both indexes :-{
Eliminating Dummy Values

• DIOs typically use dummy values in place of NULL, e.g., -99
• Queries use:
  WHERE column_name <> -99
  instead of
  WHERE column_name IS NOT NULL
• <> kills use of index on column_name
• If significant percentage of rows contain dummy value, add functional index to improve performance
  – NULLIF(column_name,-99)
• Queries need to be modified to use function
  • WHERE NULLIF(column_name,-99) IS NOT NULL
• Real world cases may involve multiple dummy values, e.g. -9, -99 and -999 (really!)
  – Use DECODE, CASE or other function
Use PL/SQL for Bulk Operations

- Use of BULK COLLECT and FORALL provides huge performance improvements over application side operations
Summary

• Many anti-patterns easily identifiable
• Many anti-patterns subject to easy, quick and safe fixes
  – OMG Tips won’t work for every query
• SQL Hero needs to be willing to modify queries and test results
• SQL Hero needs to understand why DIOs use anti-patterns and educate them
Questions?
Sources For More Info

- Jeffrey Jacobs & Associates  
  - http://www.jeffreyjacobs.com
- Oracle  
  - http://www.oracle.com/technology
- FMT Systems Inc.  
  - www.fmtsystems.com/
- Our Favorite Blogs Sites/Updates  
  - http://processconnectionsblog.com/ Faun deHenry’s Blog  
  - http://orclville.blogspot.com Floyd M. Teter’s Blog  
Ask Us Anything Webinar 2011 Series

**August**- Summer Break
**September**- Mike Swing of TruTek
**October**- Oracle ACEs Take On OOW
**November**- Faun deHenry
**December**- tbd

Are You Interested in Presenting?
Let us know at [Hope.Malone@fmtsi.com](mailto:Hope.Malone@fmtsi.com)
• Additional questions can be sent to aua@fmtsi.com

• Visit our website for more information at www.FMTSystems.com

• Register to receive our newsletters and webinar invitations-enter AUA711 in promo code for a chance to win a $25 gift card.
Thank You for Joining!

• Additional questions can be sent to aua@fmtsi.com

• Visit our website for more information at www.FMTSystems.com

• Register to receive our newsletters and webinar invitations-enter AUA711 in promo code for a chance to win a $25 gift card.