SQL PERFORMANCE HERO AND OMG METHOD PLAY THE ANTI-PATTERNS GREATEST HITS

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

INTRODUCTION

Most papers and presentations on SQL performance focus on tracing and other techniques, assuming that the query developer is highly experienced in SQL and Oracle.

Sadly, in today's procedural and object oriented coding world, knowledge of SQL and Oracle are not highly valued.

Many, if not most, SQL performance problems are the result of inexperience, misguidance, "heard it/read it somewhere" and other sources of mis-information.

This paper addresses common "anti-patterns" in database design and SQL query writing that I've encountered over the years. It focuses on fixing common performance problems without tracing, changing init.ora parameters or other techniques requiring DBA level privileges. The OMG Method uses techniques that are typically available to developers, primarily query and code refactoring, and indexing.

An *anti-pattern*, for the purposes of this paper, is a common design or SQL practice that results in poor performance.

I call this technique the *OMG Method*, textese for "Oh My God", which is part of a common response I have to many of the problems I've encountered. Discretion prevents me from completing the response, but it is seldom complimentary. But as a "method", it's "Oh my god, that's *sooo* easy to fix".

Once you learn the OMG Method, you too will be considered a SQL Performance Hero!

Note that this paper assumes that the SQL Performance Hero can refactor SQL, work with developers on the application side, and work with the DBA's to modify indexing, but is powerless to otherwise `ALTER` the tables or columns in the schema.

UNDERSTANDING THE DEVELOPER INEXPERIENCED IN ORACLE TECHNOLOGIES (DIO)

In order to be a true SQL Performance Hero, you need to understand why OMG is such a common reaction to so many performance problems.

Even though OO theory is in fact older than relational theory, originating with Simula-67 in 1967, versus the early '70s for Codd's and Date's works, OO didn't really become the dominant force until the '90s, resulting in the common view that RDBMS are old and outdated.

The majority of today's developers receive little, if any, training in RDBMS theory or practice in college. Their training is almost totally in procedural, object oriented (OO) languages, primarily Java, possibly with a smattering of *functional* languages and techniques.

Further, most of today's developers have had no real training in SQL, Oracle or good design practices. They typically pick up Oracle and SQL by a little bit of reading, but never delve deeply into the techniques. They pick up random bits of mis-information, which they take as gospel. We will refer to them as *Developers Inexperienced in Oracle technologies (DIO).*

This results in a common set of anti-patterns, which I've seen in many different environments. While this paper doesn't directly address design issues, the common design anti-patterns are often significant contributors to the performance problems which are addressed here.

Common anti-patterns include:

- Overly Generic Data Models – In particular, variants of the `OBJECT/INSTANCE` and `ATTRIBUTE/ATTRIBUTE_VALUE` table approach. Very often, this will involve poorly managed data, e.g., surrogate keys for the same meta-data that is not consistent across environments, overloaded columns requiring data conversion, lack of integrity constraints and inappropriate indexing.
- Fat, Unnormalized Tables – Normalization is seldom taught these days, and, when it is, it is all too often perceived as a purely theoretical exercise. Tables with dozens, and even 100s, of columns are all too common. Making

matters worse is the overuse of CLOBs, typically stored in-line. This is further exacerbated by making such tables generic, e.g., a table named DATA_OBJECT with 15 number columns named "NUMBER_COL_1" thru "NUMBER_COL_15", 15 such columns for various sizes of VARCHAR2, DATE, etc. In fact, this model was even worse! There were a 6 more tables, DATA_OBJECT_1 through DATA_OBJECT_6, all of which had 6 recursive foreign keys and 6 foreign keys back to DATA_OBJECT. The data model diagram looks like a tarantula or chip diagram.

- Fear of Joins – It's amazing, and a bit scary, how often I've heard the justification "I didn't want to do any joins" as the reason for poor design. Or "joins are bad", so "we avoid them at all costs". One of my favorite jokes, "A DBA asked a developer why he wrote Cartesian join that brought the system to its knees. The developer replied 'I wanted to be sure Oracle could handle something simple before I gave it anything more complex'"!
- SQL Query Cost – While it seems a bit contradictory, one of the most common causes of poor performance is the DIO's lack of understanding of the relative cost of a SQL query, typically resulting in issuing far more queries than are necessary. While the individual query may be very efficient, the sheer volume results in a severe performance problem.
- *Iterative* vs. *Set* World View – The procedural developer thinks in terms of iterating over a *collection*, e.g., performing the same operation on each individual element of the collection. The SQL Performance Hero thinks in terms of (result-) sets. The net effect is that the DIO all too often performs operations, such as de-facto joins, in application code that should be performed in the database, resulting in executing far more queries than are necessary. My favorite, real world example is the DIO that performed 80,000 SELECTs taking almost 4 hours. The same operation performed with a single query took approximately 30 seconds. All of the processing that was performed in the application was performed by a CASE clause in the SELECT. (Even sadder, the development manager wanted to schedule a meeting to determine which approach was better, as the CASE clause was "complicated".)
- Unmanaged Surrogate Keys – One of the fundamental tenets of relational design is that the PK value for the same "row" is the same everywhere, e.g. the value of COMPANY_ID for a company is the same in all environments. However, since so much data is generated by application code on the fly, this tenet is often violated, leading to a wide variety of performance and consistency issues. Imagine a BUSINESS_SERVICE reference table with 30 rows where the PK value is different across the development, QA, integration and production environments (except this is a real example, not imaginary)! Typically, there will be another lookup column(s) with a unique index/key that is supposed to be consistent across environments. Be aware that such lookup columns all too frequently use *mixed* case strings.
- Widespread use of *Dummy* Values instead of NULL – DIOs are typically very uncomfortable with 3-valued logic and unsure when it is appropriate to use NULL. They are also unsure about the interactions between NULL and indexes. As such, they typically use dummy values where NULL would be a better choice.
- Lack of Documentation

SQL TIPS

AVOID DYNAMIC SQL

Avoid "dynamic" SQL, e.g. creating a SQL string by concatenating strings together and then executing the string. This is very expensive in both memory and CPU usage. Whenever possible, use prepared statements with bind variables. This reduces parsing time and allows the resulting cursor to be reused. (A database "parse" is not the same thing as parsing text; it is potentially an expensive operation more akin to compiling.)

AVOID/REPLACE INLINE VIEWS

An *inline view* is a subquery that returns a result set that is used as a "temporary" table for the query, e.g., the inline view is not merged. See *Note about Inline View Merging* below.

Note that Oracle's definition of a "(Global) temporary table" is very different!!! You will find that many DIOs, particularly those with SQLServer backgrounds, will refer to inline views as "temporary tables". It helps to try to educate them on the proper Oracle terminology.

While there are certainly times where an inline view is the appropriate construct to use, they are all too frequently misused. The following example is not unusual, even for simple queries.

```
SELECT EMP.NAME, DEPT_INLINE.NAME
        FROM
        EMP,
        (SELECT NAME,DEPT_ID FROM DEPT) DEPT_INLINE
        WHERE
        DEPT_INLINE.DEPT_ID = EMP.DEPT_ID
```

The performance issue with an inline view when joined to another table is that all of the rows in the result set of the in-line view are scanned for each join row, e.g. the equivalent of a full table scan (FTS).

Joins between inline views and nested inline views are also common, resulting in even bigger performance issues.

The vast majority of inline views can, and should, be rewritten as joins between the tables involved, e.g.

```
SELECT EMP.NAME, DEPT.NAME
        FROM
        EMP, DEPT
        WHERE
        DEPT.DEPT_ID = EMP.DEPT_ID
```

Often the same inline view will be used more than once in a query; see Subquery Factoring below.

Note about Merging of Inline View

Note that this tip is only applicable if Oracle creates an inline view in the execution plan. Oracle may instead *merge* the inline view with the rest of the query, e.g., rewriting the query. However, this sometimes results in *worse* performance. You can determine whether letting Oracle merge the inline view is better by using the `/*+ NO_MERGE */` hint. Conversely, you can test merging using the `/*+ MERGE */` hint.

Never UPDATE Primary key (PK) Columns

Never UPDATE primary key (PK) columns, not even to their current value. Not only is it bad practice and bad design, it results in unnecessary checking for child rows if there are foreign keys (FK) referencing the primary key (and always fails if they exist).

Such UPDATEing is part of a common anti-pattern where all of the columns in a row are UPDATEd, even if the values of some columns aren't changed. This anti-pattern allegedly improves performance by reducing the number of distinct queries and resulting cursors in the Shared Global Area (SGA). This faulty logic is primarily a holdover from memory and tuning limitations in earlier versions of Oracle.

Avoid Unnecessary Outer Joins

Avoid unnecessary outer joins. Outer joins always impact performance, inhibiting appropriate use of indexes. "Just to be safe" is a common DIO's basis for performing an outer join. Determine if the outer join is in fact necessary. All too often, it is not needed.

Fetch Size

Be sure your "fetch size" is set appropriately. This is the number of rows that are returned with each fetch. The more fetches you perform, the more network traffic. The default value for Oracle ODBC/JDBC is generally too small.

EXISTS vs IN

One of the most common DIO mistakes, and one of the most common dramatic improvement in query performance, is replacing IN with EXISTS.

When coded properly, IN uses an uncorrelated subquery. The subquery is executed once, returning a result set. This result set is then scanned for every row that is to be joined in the outer query. If this result set is large, the performance will be very poor.

Use EXISTS with a correlated subquery instead of IN for such *semi-joins*. If you are performing a query of the form

```
SELECT …
        FROM table_1 outer
        WHERE outer.col_1
        IN
        (SELECT inner.col_1
                FROM table_2 inner
                [WHERE …])
```

You will generally get much better performance from

```
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
```

This assumes that `inner.column_1` is indexed. This is *usually* the case; *however, if it is not indexed, performance will generally be worse than using IN. Check to see if there should be an index on* `inner.column_1`*!!*

The main exception to this is when you want all, or the vast majority of, the values from the semi-join condition.

Note that when using a semi-join (or anti-join), IN (SELECT DISTINCT...) is not necessary, and is in fact redundant. The DISTINCT is exactly what distinguishes the EXISTS and IN from a regular equi-join.

SUBQUERY FACTORING USING WITH

One of the least known constructs in Oracle is "subquery factoring" using the WITH clause.

Unlike a procedural language compiler's optimizer, Oracle's optimizer does not unify identical subqueries within a query. A common example is the use of the same inline view in UNION queries, e.g.

```
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 …
```

The "IV" inline views are identical, but Oracle will execute and return a separate result set for each occurrence. If these inline views are poorly written and expensive, this can be a huge performance issue.

Inexperienced coders frequently use multiple UNIONs of this form because they are "simple" to code.

Subquery factoring allows you to both improve performance and simplify the query.

The syntax for using the WITH clause is:

```
WITH
(SELECT …)
AS pseudo_table_name_1
[, (SELECT …) AS pseudo_table_name_2 …] - multiple queries can be defined

SELECT …
        FROM pseudo_table_name …
… -- typically UNIONs
```

Applying this to the example:

```
WITH
        (SELECT … FROM table_2, table_3, … WHERE table_2.id = table_3.id
        AS IV
SELECT …
        FROM
        table_1, IV
        WHERE …
UNION
        SELECT …
        FROM
        Table_4,IV
        WHERE …
UNION …
```

Oracle executes the WITH query, e.g. IV, *only once* and store the results in a "temporary table". Note that this is a true table, and the results are written to disk. The table is populated by a direct load INSERT from the query.

I've used this technique as a quick fix on some incredibly ugly queries, achieving performance improvement of 10X and better.

Some caveats are in order. Oracle actually executes a CREATE TABLE as part of the recursive SQL, at least on the 1st execution of the query. The table, and the corresponding INSERT cursor, may be reused for subsequent executions of the query. However, it appears that the table can only be used for one query at a time. If query executions overlap, Oracle will create additional tables and INSERT statements, i.e. it does not create a "Global Temporary Table". Instead, it uses a naming convention and apparently keeps track of availability of the tables and corresponding cursors.

The creation of the temporary table is also referred to as *materializing*. Use of the temporary table is only a performance gain if the factored subquery is used more than once.

Also, the temporary table feature does not always result in a performance improvement; sometimes the additional overhead of the temporary table is a loser, even if the subquery is used more than once! *However*, simplifying code is always a winner. You can use the INLINE hint to force Oracle to bypass the creation of the temporary table and use the subquery in the same manner as it did in the original, unfactored query.

To force materializing, you can use the /*+ MATERIALIZE */ hint.

Note that these hints are not documented by Oracle.

INDEX HINTS

One of the most common DIO mis-conceptions is that full table scans (FTS) are *always* bad, and using indexes are *always* better, so the DIO sprinkles *unqualified* `INDEX` hints everywhere.

The SQL Hero knows that sometimes FTS are in fact better.

When using `/+ INDEX … */`, always specify the index to be used, not just the table name, e.g.

```
/+ INDEX(DEPT DEPT_PK_IDX)*/
```

instead of

```
/+ INDEX (DEPT) */.
```

The latter simply tells the Cost Based Optimizer (CBO) to use index(es) instead of FTS, but doesn't specify which index to use. As such, this method is unpredictable and unreliable; the execution plan can change when statistics are recomputed, indexes are added, etc. Further, if there isn't a *good* index, the CBO will still use an index, no matter how badly it may perform

Hints should only be added when you understand the execution plan and what specifically will improve performance

DATA CONVERSION ISSUES

When comparing a VARCHAR2 column to a constant or bind variable, be sure to use the appropriate data type for the constant/bind variable. In particular, if the constant is a number, enclose the constant in quotes, e.g.

```
WHERE MY_VARCHAR2_COL = '1'
```

Otherwise, Oracle does the *wrong* data conversion which can cause both performance problems and possible spurious errors; it converts the values from the VARCHAR2 column to a number instead of converting the numeric constant to a string!

Even SQL Heroes often forget or are unaware of this little quirk. If you ever wonder why you can't get rid of an FTS, don't forget this gotcha!

This is a very common performance issue where the IDs have created generic tables or object oriented style schema designs with a column that holds more than one logical attribute.

ELIMINATE UNNECESSARY *LOOKUP* JOINS

As described above, the use of unmanaged surrogate key values is typically coupled with a lookup column with a (hopefully) consistent value. The typical code used to join to such table is:

```
SELECT
        FROM child_table, reference_table
        WHERE
                child_table.reference_table_id = reference_table.reference_table_id
                and reference_table.lookup_column = '<CONSTANT>'
                …
```

This results in accessing `reference_table` for every row accessed in `child_table`.

Even worse is the use of an UPPER or LOWER function applied to `lookup_column` without a corresponding functional index on `lookup_column`, which will result in an FTS of `reference_table`.

The join should be replaced with a scalar subquery:

```
SELECT
        FROM child_table
        WHERE
                child_table.reference_table_id =
                (SELECT reference_table_id
                        FROM reference_table
                        WHERE
                                reference_table.lookup_column = '<CONSTANT>')
```

IMPROVING PAGINATION

Pagination generally refers to returning rows *n* through *m* from an ordered result set using ROWNUM, typically to display a subset of the data on a web page.

The most common, and worst performing, use of this technique is:

```
SELECT t1.col_1,…
        FROM
        (SELECT *
                FROM table_1
                WHERE …
                ORDER BY …) t1
        WHERE
                ROWNUM between :n and :m
```

There are several steps to improve this query's performance..

1. Replace any '*' in the innermost in-line view with the actual columns to be retrieved, as shown in Step 2. This avoids unnecessary I/O in the innermost in-line view processing.
2. Refactor the query so that the in-line view only returns the 1st *m* rows and uses the `/*+ FIRST_ROWS */` hint (see Tom Kyte's *Effective Oracle by Design* on *Pagination with ROWNUM*), e.g.

```
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
```

3. Consider returning ROWIDs from the innermost in-line view instead of the actual data and retrieving the actual data in the outer query. This can substantially reduce the amount of I/O and related processing in the innermost in-line view when the table is fat with CLOBs, e.g.,

```
SELECT t1.col_1,…
FROM
        table_1,
        (SELECT /*+ FIRST_ROWS */
                ROWNUM AS rnum, row_id
                FROM
                (SELECT ROWID row_id
                        FROM table_1
                        WHERE …
                        ORDER BY …)
                WHERE
                ROWNUM <= :m)
WHERE rnum > = :n
AND table_1.ROWID = row_id
```

DESIGN AND INDEX TIPS

UPDATE AND DELETE PERFORMANCE ISSUES

"I'm deleting/updating a few hundred rows. In my development environment, it's virtually instantaneous. But in other environments, even though it's still a small number of rows, it's taking forever"! – DIO

This is typically due to lack of an index on the referencing FK of a child table(s). This results in an FTS of the child table for *every* row in the parent table to be deleted/updated. Typically, the DIO will be focusing on the proper creation of rows in the parent table, but only test a few rows in the child table. My favorite real world example involves a child table that had only 30 rows in the development environment, but close to 3,000,000 in production.

ALWAYS INDEX COLUMNS ON FOREIGN KEYS

It's scary how often I encounter FK constraints that are not indexed properly. Proper indexing on FKs is an must! I've yet to encounter a situation where I would not index an FK constraint. In a recent engagement, I found and created 135 indexes on FKs in a schema with approximately 600 tables.

There are 3 main reasons to index the FK columns:

1. Improved JOIN performance. The CBO uses indexes and related statistics to determine its execution plan.
2. UPDATE and DELETE performance will be greatly improved.
3. In the absence of such indexes, Oracle appears to still perform table level locks, despite claims to the contrary.

Proper indexing of FK columns requires that the columns in the FK constraint be the leading columns in the index. The index may have additional columns, but they must follow the FK columns.

ADD FOREIGN KEY (FK) CONSTRAINTS

DIOs all too often avoid FK constraints to "improve performance", using the argument that "we'll enforce referential integrity (RI) in the application". However, what is really meant here is that "we'll try not to make any mistakes in the application code when creating/updating parent/child relationships". If referential integrity were fully enforced in the application code, performance would be worse not better, as such enforcement would require more traffic between the application and the RDBMS.

The CBO in fact makes use of the existence of FK constraints in its decision making.

It doesn't matter how well an application performs if the results are wrong! It would be interesting to calculate how many hours go into data fixes that could have been prevented by the simple use of FK constraints.

Further, the existence of FK constraints adds to the effective documentation of the system, eliminating guess work.

Reduce Unnecessary and Redundant Queries

Probably the single greatest performance killer is too many queries. This is also typically the SQL Performance Hero's greatest challenge, as it involves working with the DIO to change the application. There are two primary cases:

1. Iteration – In this case, the DIO issues a large number of SELECTs which can and should be combined into a single query. Often, the DIO is effectively performing joins in the application code that should be performed in the database.
2. Redundant Queries – In this case, the DIO is retrieving the same information repeatedly, even though the data has already been retrieved and should have been cached locally in some fashion. Of course, you need to be sure that the queries are in fact redundant!

Add Functional Indexes

Functional indexes (FI) are a tremendously powerful tool for quick performance improvements. While a comprehensive discussion of FIs is beyond the scope of this paper, following are several common situations where addition of an FI is appropriate.

- Use of `UPPER/LOWER` in `WHERE` clause – The application of a function in a WHERE clause inhibits the use a non-functional index. Many DIOs use mixed case data in their `VARCHAR2` lookup columns. Other developers who do not want to depend on exactly matching the case of the data will force case conversion using `UPPER/LOWER` function on the column, e.g. ,

  ```
  WHERE UPPER(LOOKUP_COLUMN) = '<CONSTANT>'
  ```

  Adding the appropriate FI will often result in a significant performance improvement.
- Eliminating Dummy Values – As described earlier, DIOs frequently use dummy values instead of NULL. While the use of dummy values, such as -99, may improve performance in the case of queries with:

  ```
  WHERE table_1.column_1 = -99 AND …
  ```

  which is the logical equivalent of

  ```
  WHERE table_1.column_1 IS NULL AND…
  ```

  and

  ```
  WHERE table_1.column_1 <> -99 AND …
  ```

  which is the logical equivalent of

  ```
  WHERE table_1.column_1 IS NOT NULL AND…
  ```

  The latter will typically result in a FTS of `table_1`.

If the table has a significant percentage of rows with the dummy value, the addition of a functional index, such as `NULLIF(column_1,-99)`, and its appropriate use in queries, will effectively treat the column as being a `NOT NULL` column. The resulting index will be smaller, and only rows where `column_1 <> -99` will be evaluated.

Eliminate Redundant Indexes

DIOs tend to create far too many indexes, in the mistaken belief that using indexes is *always* better. To compound the problem, the indexes are frequently redundant, e.g., the same leading columns are used for more than one index. I've seen tables where 3 multi-column indexes on the table had the same two leading columns (with different trailing columns), and another single column index that used the same leading column as the others.

Redundant indexes not only impact inserts, updates and delete performance, they effectively confuse the CBO. I must confess that in such situations, I've yet to determine how the CBO selects the index to use.

In general, indexes with more than two or three columns should be eliminated.

Indexes that have PK columns as leading columns followed by additional columns should be eliminated.

USE PL/SQL FOR BULK OPERATIONS

Most DIOs have little or no knowledge of PL/SQL. In particular, they will perform operations on large data sets in the application code. This can be incredibly expensive; I've seen developers attempt to update hundreds of thousands of rows in Java code. Even using array binding, this typically requires an excessive number of queries and network round trips.

Using PL/SQL and the bulk binding features, `BULK COLLECT` and `FORALL`, dramatically improves performance by eliminating round trips and queries between the application and the database. Further, the use of these features dramatically reduces context switching in the server.

SUMMARY

I hope the tips and techniques in this paper will help you become a SQL Performance Hero. All it takes is a willingness to dive in and fix the obvious. Realizing that the vast majority of SQL performance problems are the result of inexperience on the part of the developer is the first step; once you realize this fact, you should be able to safely and effectively apply the tips and techniques of the OMG Method.