

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 9, September 2015

Overhauling PL/SQL Applications for Optimized Performance

Vamsi Krishna Myalapalli¹, Karthik Dussa²

Open Text Corporation, Mind Space IT Park, Hitec City, Hyderabad, India

ABSTRACT: In the contemporary world it is inevitable to have a commercial application without PL/SQL Code. If the commercial application should outdo in the performance perspective, then it has to undergo sundry performance engineering approaches. This will include reducing end user response time, writing optimizer friendly code etc. As such this paper proposes miscellaneous revamping techniques to enhance PL/SQL Code Units and few methods to tackle SQL Injection attacks. Experimental outcomes of our analysis indicate that performance and maintainability are increased.

KEYWORDS: SQL Tuning, PL/SQL Tuning; SQL Optimization; PL/SQL Optimization; Query Tuning; Query Optimization; Query Rewrite.

I. INTRODUCTION

Typically commercial databases will entail lots of PL/SQL code for interacting with database. Tuning and debugging are indispensable for any software application, regardless of the language and software used in developing the application. Applications coded using PL/SQL as the development language for the database and/or other tiers also mandate that these exercises be performed before and after deployment in production environments.

In fact, tuning and debugging are both critical to the performance of the application and directly affect its productivity in terms of performance, scalability, and throughput. Tuning and debugging should be initiated as the application is run in test environments, which should typically simulate a production scenario. This prepares the application to be performance-centric in terms of efficiency, productivity, and scalability.

II. BACKGROUND AND RELATED WORK

High Performance PL/SQL Programming [1] explained the model to minimize the Context Switching process (an overhead) among SQL Engine and PL/SQL Engine. It also explained the methodology to increase Soft parsing rate.

High Performance SQL [2] explained tuning SQL queries via Rule Base Optimization (RBO) and Cost Based Optimization (CBO).

An Appraisal to Optimize SQL Queries [3] explained the impact of raw query vs. tuned query on database via statistics. It explained efficient use of indexes defined on database objects and sundry query rewriting techniques to reduce the rate of parsing (to ensure soft parse) and request less data.

This paper explains optimization techniques in PL/SQL code for ensuring High Performance PL/SQL applications.

III. PROPOSED MODEL

This section presents fast and holistic techniques. We have tested our proposed techniques on a production database environment.

Before optimizing PL/SQL code, we have to recognize the candidates for tuning. The following are several candidates eligible for optimization.

- a) Older code that does not take advantage of new PL/SQL language features.
- b) Code that spends much time processing SQL statements.

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 9, September 2015

- c) Functions invoked in queries, which might run millions of times.
- d) Code that spends much time looping through query results.
- e) Code that does many numeric computations.
- f) Code that spends much time processing PL/SQL statements (as opposed to issuing database definition language (DDL) statements that PL/SQL passes directly to SQL).

The following are sundry tuning techniques for ensuring optimized application performance.

1) Deter Full Table Scan (FTS):

Full Table Scan (FTS) retrieves each and every bit of data from the underlying table. FTS triggers a lot of disk I/O. It will be a nightmare for production environment and data warehouse. FTS may arise due to

- a) WHERE clause absence.
- b) Stale statistics of Table or Index.
- c) Absence of filtering rows at WHERE clause.

2) Ensuring Precise Indexing for Performance:

The following flow diagram indicates when to leverage Indexes and their creation.

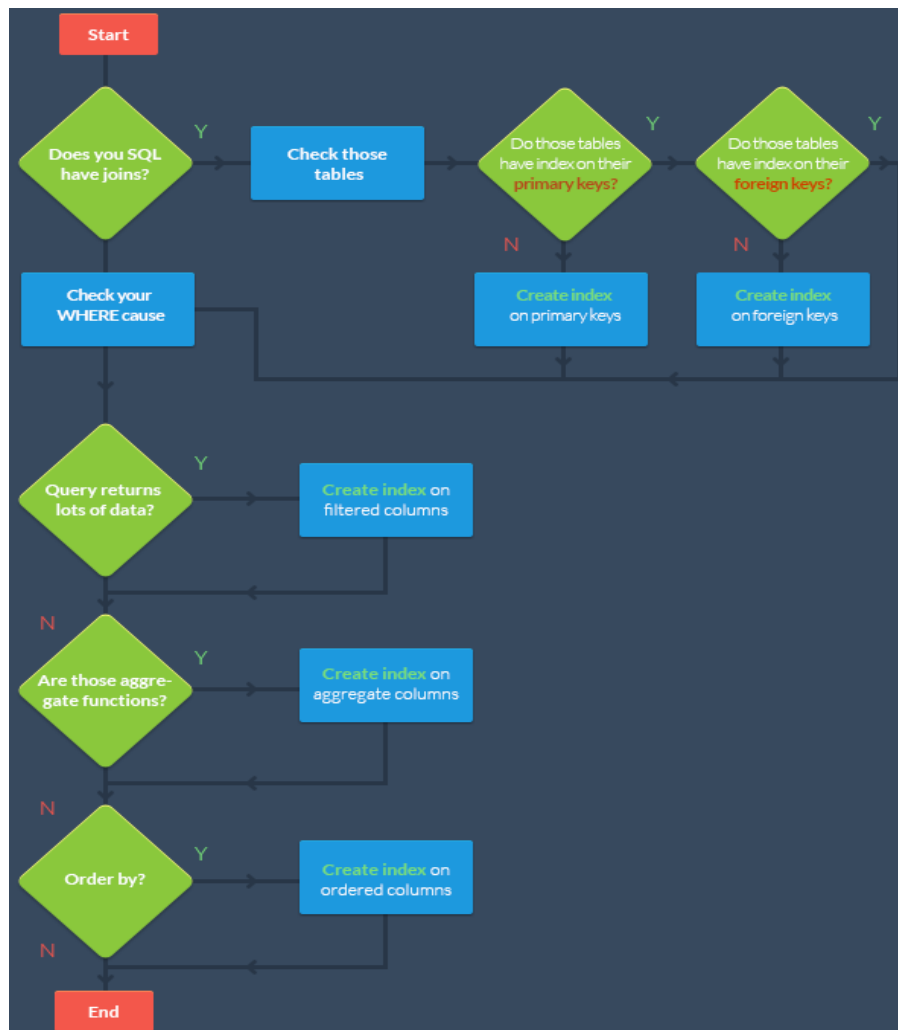


Figure 3.1: Indexing Flow Chart

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 9, September 2015

3) **Prefer Native SQL over DBMS_SQL:**

The journey from accessing a multitude of different large data sources to quickly answering key business questions is indeed technically complex. We should deem Native SQL over DBMS_SQL for the following reasons

- a) Native SQL requires much less code to implement and is much easier to use.
- b) Offers support for Objects and Collections.
- c) Offers better performance.

4) **LOB vs. LONG Data Type Columns:**

Prefer LOB over LONG column as LONG data type is obsolete. The following diagram signifies the importance of LOB by contrasting it with LONG data type.

Large Object Type (LOB)	LONG Column
Maximum Size is 4 GB	Maximum Size is 2GB
Allows Random Access	Allows only Sequential Access
Multiple LOB Column are allowed per table	Only one is allowed per table
Contemporarily used	Obsolète
LOB Data is stored separately from Table	LONG Data is stored inline with other Columns
Implicit Conversion among LOB & Varchar2 is allowed	No implicit conversions are allowed

Figure 3.1: Contrasting LOB and LONG Columns

5) **Minimizing Dependency Failure(s):**

We should strive to minimize Dependency failures as much as possible. The following practices would help in achieving this

- a) Making use of %ROWTYPE attribute when declaring records. %ROWTYPE attribute ensures that data types of variables that are declared with %ROWTYPE attribute change dynamically when underlying table is altered.
- b) Making use of the %TYPE attribute when declaring variables.
- c) Using the 'SELECT *' notation when querying tables.
- d) Providing a column list when using INSERT statements.

6) **Consider Using Stored Functions:** Stored functions can increase the efficiency of queries by performing functions in the query rather than in application.

7) **Watching Out for Side Effects:**

Be vigilant towards anticipated or unprecedented side effects. Example: When using a PL/SQL stored package, changes occur to database tables or packaged variables defined in a package specification.

8) **Efficient Exception Handling:**

- a) Handle named exceptions whenever possible, instead of using 'WHEN OTHERS' in the exception handler.
- b) Instead of adding exception to PL/SQL block, check for errors at every point where they may occur.

9) **Updating Rows Efficiently:**

When updating rows we should use SELECT FOR UPDATE or a similar mechanism to lock any rows that contain (Large Object) LOBs that will be updated by DBMS_LOB to provide concurrency.

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 9, September 2015

10) Dealing with Binary Data Types:

We need to be aware that BINARY_FLOAT and BINARY_DOUBLE have some side effects. For both of these data types floating point operations are indeed binary. Therefore the rounding results may not be what we expect since binary (and not decimal) rounding will be used. Also, because of machine arithmetic, we might have portability issues since the same operations may differ slightly on different hardware.

Data type SIMPLE_INTEGER is oriented towards running inside natively compiled PL/SQL.

11) Avoid Constrained Subtypes in Performance-Critical Code:

In performance-critical code, avoid constrained subtypes. Each assignment to a variable or parameter of a constrained subtype requires extra checking at run time to ensure that the value to be assigned does not violate the constraint.

12) Minimize Implicit Data Type Conversion:

At run time, PL/SQL converts between different data types implicitly (automatically) if necessary. Overload the subprograms with versions that accept parameters of different data types and optimize each version for its parameter types.

13) Use SQL Character Functions:

SQL has many highly optimized character functions, which use low-level code that is more efficient than PL/SQL code. Use these functions instead of writing PL/SQL code to do the same things.

14) Consider Autonomous Transactions:

An autonomous transaction is an independent transaction started by another transaction, the main transaction. Autonomous transactions do SQL operations and commit or roll back, without committing or rolling back the main transaction. Although an autonomous transaction is started by another transaction, it is not a nested transaction.

Properties of Autonomous Transaction(s):

- a) It does not share transactional resources (such as locks) with the main transaction.
- b) It does not depend on the main transaction. e.g., if main transaction rolls back, nested transactions roll back, but autonomous transactions do not.
- c) Its committed changes are visible to other transactions immediately. A nested transaction's committed changes are not visible to other transactions until the main transaction commits.
- d) Exceptions raised in an autonomous transaction cause a transaction-level rollback, not a statement-level rollback.

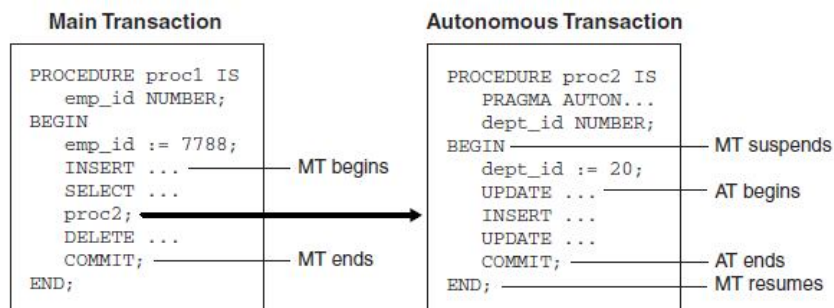


Figure 3.3: Transaction Control Flow

Advantages of Autonomous Transactions:

After starting, an autonomous transaction is fully independent. It shares no locks, resources, or commit-dependencies with the main transaction. We can log events, increment retry counters, and so on, even if the main transaction rolls back.

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 9, September 2015

Autonomous transactions help us build modular, reusable software components. We can encapsulate autonomous transactions in stored subprograms. An invoking application needs not know whether operations done by that stored subprogram succeeded or failed.

15) *Minimizing Subprogram Side Effects:*

A subprogram has side effects if it changes anything except the values of its own local variables. For example, a subprogram that changes any of the following has side effects:

- a) Its own OUT or IN OUT parameter
- b) A global variable
- c) A public variable in a package
- d) A database table
- e) The database
- f) The external state (by invoking DBMS_OUTPUT or sending e-mail, for example)

Minimizing side effects is especially important when defining a result-cached function or a stored function for SQL statements to invoke.

16) *Faster Row Update:*

Always access rows by ROWID rather than by primary key when we are using SELECT...FOR UPDATE to retrieve, process, and then update the same row. This is faster than using the primary key.

17) *Dynamic vs. Static SQL:*

Do not use dynamic SQL when we can use static SQL. Dynamic SQL is costlier than static SQL in terms of parsing and execution and also has the disadvantages of loss of dependency checking and compile-time error checking.

18) *Preventing SQL Injection:*

All SQL injection techniques exploit a single vulnerability: String input is not correctly validated and is concatenated into a dynamic SQL statement. The following are the SQL Injection techniques:

a) Statement Modification:

Statement modification means deliberately altering a dynamic SQL statement so that it runs in a way unintended by the application developer. Typically, the user retrieves unauthorized data by changing the WHERE clause of a SELECT statement or by inserting a UNION ALL clause. The classic example of this technique is bypassing password authentication by making a WHERE clause always TRUE.

b) Statement Injection:

Statement injection means that a user appends one or more SQL statements to a dynamic SQL statement. Anonymous PL/SQL blocks are vulnerable to this technique.

c) Data Type Conversion:

A less known SQL injection technique uses NLS session parameters to modify or inject SQL statements.

A date-time or numeric value that is concatenated into the text of a dynamic SQL statement must be converted to the VARCHAR2 data type. The conversion can be either implicit (when the value is an operand of the concatenation operator) or explicit (when the value is the argument of the TO_CHAR function). This data type conversion depends on the NLS settings of the database session that runs the dynamic SQL statement. The conversion of date-time values uses format models specified in the parameters NLS_DATE_FORMAT, NLS_TIMESTAMP_FORMAT, or NLS_TIMESTAMP_TZ_FORMAT, depending on the particular date-time data type. The conversion of numeric values applies decimal and group separators specified in the parameter NLS_NUMERIC_CHARACTERS.

Guarding Against SQL Injection:

If we use dynamic SQL in PL/SQL applications, we must check the input text to ensure that it is exactly what we expected. We can use the following techniques:

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 9, September 2015

a) Bind Variables:

The most effective way to make the PL/SQL code invulnerable to SQL injection attacks is to use bind variables. The database uses the values of bind variables exclusively and does not interpret their contents in any way. (Bind variables also improve performance.)

b) Validation Checks:

Always have the program validate user input to ensure that it is what is intended. For example, if the user is passing a department number for a DELETE statement, check the validity of this department number by selecting from the departments table. Similarly, if a user enters the name of a table to be deleted, check that this table exists by selecting from the static data dictionary view ALL_TABLES.

When checking the validity of a user name and its password, always return the same error regardless of which item is invalid. Otherwise, a malicious user who receives the error message "invalid password" but not "invalid user name" (or the reverse) can realize that he or she has guessed one of these correctly.

In validation-checking code, the subprograms in the DBMS_ASSERT package are often useful. For example, we can use the DBMS_ASSERT.ENQUOTE_LITERAL function to enclose a string literal in quotation marks. This prevents a malicious user from injecting text between an opening quotation mark and its corresponding closing quotation mark.

In validation-checking code, the subprograms in the DBMS_ASSERT package are often useful. E.g., we can use the DBMS_ASSERT.ENQUOTE_LITERAL function to enclose a string literal in quotation marks. This prevents a malicious user from injecting text between an opening quotation mark and its corresponding closing quotation mark.

c) Explicit Format Models:

If we use date-time and numeric values that are concatenated into the text of a SQL or PL/SQL statement, and we cannot pass them as bind variables, convert them to text using explicit format models that are independent from the values of the NLS parameters of the running session. Ensure that the converted values have the format of SQL date-time or numeric literals. Using explicit locale-independent format models to construct SQL is recommended not only from a security perspective, but also to ensure that the dynamic SQL statement runs correctly in any globalization environment.

IV. EXPERIMENTAL SETUP

Some of the Queries are explained in this section via queries, which correspond to technics explained in previous section.

```
10. SQL> create or replace procedure Test_Number IS
    myvar NUMBER:=1;
    BEGIN
    For i in 1..100000 LOOP
        myvar := myvar + 2 * myvar - 1 * myvar ;
    END LOOP;
END;
```

```
SQL> create or replace procedure Test_Binary_Integer IS
    myvar BINARY_INTEGER:=1;
    BEGIN
    For i in 1..100000 LOOP
        myvar := myvar + 2 * myvar - 1 * myvar ;
    END LOOP;
END;
```

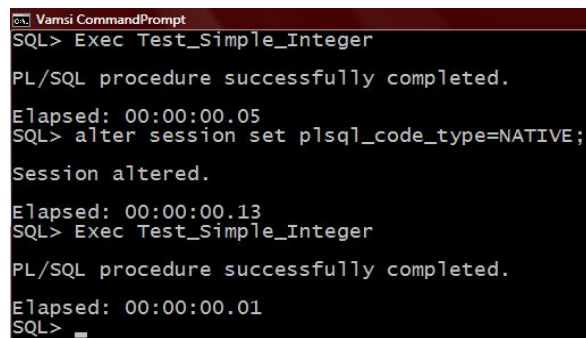

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 9, September 2015

```
SQL> create or replace procedure Test_Simple_Integer IS
  myvar SIMPLE_INTEGER:=1;
BEGIN
  For i in 1..100000 LOOP
    myvar := myvar + 2 * myvar - 1 * myvar ;
  END LOOP;
END;
```

Natively compiled SIMPLE_INTEGER cuts the total time spent.
For native compilation we have to issue the following command.
SQL> alter session set plsql_code_type=NATIVE;



```
Vamsi Command Prompt
SQL> Exec Test_Simple_Integer

PL/SQL procedure successfully completed.
Elapsed: 00:00:00.05
SQL> alter session set plsql_code_type=NATIVE;

Session altered.
Elapsed: 00:00:00.13
SQL> Exec Test_Simple_Integer

PL/SQL procedure successfully completed.
Elapsed: 00:00:00.01
SQL> _
```

Figure 4.1: Screen Shot Demonstrating the Effect of Native Compilation

If we know the number and data types of the input and output variables of a dynamic SQL statement at compile time, then we can rewrite the statement in native dynamic SQL, which runs noticeably faster compared to its equivalent code.

V. CONCLUSION

Optimization begins when an application's development is already finished. Performance is not merely optional, though; it is a key property of an application. Not only does poor performance jeopardize the acceptance of an application, it usually leads to a lower return on investment because of lower productivity of those using it.

We must code our program in PL/SQL by trying to offload work to the SQL engine as much as possible. Use single-row, loop-based processing only as a last resort. PL/SQL Tuning primarily heeds on minimizing Context switch among SQL Engine and PL/SQL Engine [1] and reducing the rate of Hard Parse [3].

In fact we should not try to optimize when there is need. The only thing we should do during programming is follow the best practices and optimize by recognizing the candidates for tuning. Apart from these approaches, we should try revamping our code according to our requirements and based on the underlying hard-ware.

This paper explained a sample of potential problems and corresponding overhauling methodologies which are recurrently experienced by developers.

REFERENCES

- [1] Vamsi Krishna Myalapalli and Bhupati Lohit Ravi Teja, "High Performance PL/SQL Programming", IEEE International Conference on Pervasive Computing, January 2015, Pune, India.
- [2] Vamsi Krishna Myalapalli and Pradeep Raj Savarapu, "High Performance SQL", 11th IEEE India Conference on Emerging Trends in Innovation and Technology, December 2014, Pune, India.
- [3] Vamsi Krishna Myalapalli and Muddu Butchi Shiva, "An Appraisal to Optimize SQL Queries", IEEE International Conference on Pervasive Computing, January 2015, Pune, India.

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 9, September 2015

- [4] Vamsi Krishna Myalapalli, Thirumala Padmakumar Totakura and Sunitha Geloth, "Augmenting Database Performance via SQL Tuning", IEEE International Conference on Energy System and Application, October 2015, Pune, India.
- [5] Vamsi Krishna Myalapalli, ASN Chakravarthy and Keesari Pratap Reddy, "Accelerating SQL Queries by Unravelling Performance Bottlenecks in DBMS Engine", IEEE International Conference on Energy Systems and Application, October 2015, Pune, India.
- [6] Vamsi Krishna Myalapalli, "Optimizing SQL Queries in OLAP Database Systems", (Unpublished).
- [7] Vamsi Krishna Myalapalli, "Wait Event Tuning in Database Engine", Springer International Journal, 2015.

BIOGRAPHY



Vamsi Krishna completed his Bachelor's in Computer Science from JNTUK University College of Engineering, Vizianagaram. He is Oracle Certified SQL Expert, PL/SQL Developer, Professional JAVA SE7 Programmer, Professional DBA Professional 11g, Performance Tuning Expert and Microsoft Certified Professional (Specialist in HTML5 with JAVA Script & CSS3). He has fervor towards SQL & Database Tuning.



Karthik completed his Bachelor's in Computer Science from Vidya Jyothi Institute of Technology, Hyderabad. He is OpenText Certified Content Server Solution Developer, Content Server System Administrator, Oracle Certified JAVA SE7 Programmer and Microsoft Certified Windows Application Developer. He has fervor towards Application Development, OScript, JAVA and SQL programming.