

A



**Blueoak**  
Database Engineering

*WHITE PAPER*

# DYNAMIC SQL UPDATES

---

HOW TO OPTIMIZE UPDATE STATEMENTS  
IN STORED PROCEDURES

---

**STORED PROCEDURES AND DYNAMIC SQL**

---

Many developers have learned the importance of exploiting Oracle's PL/SQL for improving performance and abstracting database tasks from interface tasks. Almost all interactions with the database, including fetching data and transactions, can be wrapped completely within stored procedures. With the use of the Oracle native dynamic SQL interface as of 8i, many DBA tasks can also be wrapped in stored procedures, providing a consistent way for common tasks to be performed without embedding error prone commands in cron job scripts, etc.

For a more detailed discussion on stored procedures, please see the Blueoak's white paper, ["Stored Procedures: The Benefits of Using Stored Procedures."](#)

One of the strengths of the EXECUTE IMMEDIATE commands in PL/SQL is that you can submit any valid SQL statement. As most OCI programmers already know, SQL statements built and executed dynamically perform best when using bind variables. The EXECUTE IMMEDIATE ... USING provides an easy way to use bind variables within PL/SQL. In general, it always best to use static parameterized SQL wherever possible in a stored procedure. When you are forced to use dynamic SQL, bind variables should be the next alternative, with pure dynamic SQL being viewed as a last resort.

---

**NATIVE DYNAMIC SQL AND UPDATE STATEMENTS**

---

When looking at the three transaction types, inserts and deletes are usually very easily scripted using static SQL, while updates require dynamic SQL unless the update being performed is so trivial that it can be made static. Unfortunately, many developers must choose between updating too much data at once or using pure dynamic SQL. Consider a table with eleven columns where ten of the columns are updateable from a user interface. The number of possible update statements is  $2^{10} - 1$ , or 1023!! Bind variables on a simple dynamic SQL statement do not help because the columns being updated are not known in advance. It appears that there is no way to avoid pure dynamic SQL for updates other than updating all the columns of the table at once every time. However, there is a way around the problem of a static bind list.

When faced with the combinatorial issues of updates some developers would throw up their hands and claim that lower performance is the price of doing business. Consider a large scale OLTP system such as a web interface on a busy site. Even small improvements in performance can mean the difference between keeping such a system alive and thriving or dying the death of too few resources. Some simple benchmarks have shown that pure dynamic SQL runs around two to three times slower than static SQL unless the dynamic statement makes use of bind variables, and then the performance difference is usually less than 10%. Bind variables make up a performance gain that is too large to ignore.

One of the more useful features of native dynamic SQL is that the SQL string can be a PL/SQL anonymous block. This allows much greater flexibility in what you can do with dynamic SQL, and it also allows us to solve the update problem. Remember that the USING variable list of an

EXECUTE IMMEDIATE statement must be fixed. What this means is that every variable in the variable list must bind to some portion of the SQL string submitted for execution. The trick for update statements is to create a placeholder for every variable in the USING list whether or not the variable is used as part of the update. A running example will demonstrate this technique.

---

### A RUNNING EXAMPLE

---

Consider the EMPLOYEE table with the following structure:

Column Name	Type	Nullable	Updateable
ID	NUMBER	NOT NULL	NO
FIRST_NAME	VARCHAR2 (20)	NOT NULL	YES
LAST_NAME	VARCHAR2 (20)	NOT NULL	YES
MIDDLE_NAME	VARCHAR2 (20)	NULL	YES
DOB	DATE	NOT NULL	YES
TERMINATION	DATE	NULL	YES

There are five updateable columns in this table, meaning there are 31 possible update statements, which is far too many to code statically. However, it is easy to interrogate the values of the parameters of a stored procedure in order to build a dynamic update statement that would handle all 31 cases. There would be at most five IF/THEN statements.

The bind variable version of the EXECUTE IMMEDIATE statement would be something like:

```
EXECUTE IMMEDIATE   l_sql_string
USING               p_first_name,
                   p_last_name,
                   p_middle_name,
                   p_dob,
                   p_termination,
                   p_id;
```

Suppose that an employee had a name change due to a recent marriage, resulting in a change in her last name and middle name. The obvious string would be:

```
l_sql_string :=   'UPDATE employee SET last_name = :2, `
                  `middle_name = :3 WHERE id = :6';
```

Unfortunately, this statement would produce an error when plugged into the EXECUTE IMMEDIATE statement because not all the variables were bound. The way to ensure that this EXECUTE IMMEDIATE statement works every time is to construct an anonymous block:

## DYNAMIC SQL UPDATES

```
L_sql_string:= 'DECLARE `
                `L_FIRST_NAME  VARCHAR2(20) := :1;`||
                `L_LAST_NAME   VARCHAR2(20) := :2;`||
                `L_MIDDLE_NAME VARCHAR2(20) := :3;`||
                `L_DOB          DATE := :4;`||
                `L_TERMINATION DATE := :5;`||
                `BEGIN `||
                `UPDATE employee SET last_name = :2, `||
                `middle_name = :3 `||
                `WHERE id = :6; `||
                `END;`;
```

When this anonymous block is submitted to the execute immediate statement listed above, it will run correctly since every variable in the bind list is bound properly in the anonymous block. Rough benchmarks have shown this to result in only a 7% impact on performance when compared to statically coded updates. The only negative side effect is that the anonymous block does consume slightly more UGA.

---

### PERFORMANCE IMPACT

---

Dynamic SQL constructed in the way described above assumes a few things about the system and the application in which it runs. First, each form of the query will be executed at least twice. A dynamic query with bind variables can actually run a bit slower the first time it is executed when compared to pure dynamic SQL. However, once a query with bind variables is executed it resides in Oracle's cache until it is flushed. Secondly, it assumes that there are sufficient resources for the query to reside in memory. If an instance is so resource starved that it is flushing quickly or swapping to disk, then creating a query that should be cached won't help. Finally, it assumes that users will tend execute the same types of queries. If the likelihood of a dynamic query being executed in the same way twice is low, meaning it is unlikely that two users will update the same columns, then we are back to our original assumption, meaning bind variables won't help. Also, if you have a large table that is updated in many different ways, then you could be wasting memory with several cached versions of a query. Remember that a table with ten updateable columns produces 1023 different update statements. Regardless of whether or not you choose to use bind variables, there are still many benefits to having a stored procedure perform an update for an application – consistency, security, and precise transactional control. However, no method is always best in every circumstance, and the same applies to the anonymous block method I have described.

---

### INTEGRATING THE EXAMPLE WITH A STORED PROCEDURE

---

The stored procedure that would manage the bind variables would only be slightly more complicated than a stored procedure that used pure dynamic SQL for the same update. The code to update the employee table is listed below. As a general rule, it is always better to keep a stored procedure within a package. However, for convenience the code is listed as standalone procedure.

## DYNAMIC SQL UPDATES

Also, the code will show how to handle the problem of determining when a column is being updated to NULL:

```
CREATE OR REPLACE
PROCEDURE up_employee_proc(  po_return_code      OUT PLS_INTEGER,
                             po_return_string    OUT VARCHAR2,
                             p_id              IN   NUMBER,
                             p_first_name      IN   VARCHAR2 := NULL,
                             p_last_name       IN   VARCHAR2 := NULL,
                             p_middle_name     IN   VARCHAR2 := NULL,
                             p_middle_name_null IN   CHAR := 'N',
                             p_dob            IN   DATE := NULL,
                             p_termination     IN   DATE := NULL,
                             p_termination_null IN   CHAR := 'N') IS

l_sql_string  VARCHAR2(1000);

BEGIN

po_return_code := 0; -- Zero means okay
po_return_string := ''; -- No news is good news

l_sql_string:= 'DECLARE '
               '\L_FIRST_NAME  VARCHAR2(20) := :1; '||
               '\L_LAST_NAME   VARCHAR2(20) := :2; '||
               '\L_MIDDLE_NAME VARCHAR2(20) := :3; '||
               '\L_DOB         DATE := :4; '||
               '\L_TERMINATION DATE := :5; '||
               '\BEGIN '||
               '\UPDATE employee SET '

IF p_first_name IS NOT NULL THEN
    l_sql_string := l_sql_string||'first_name = :1,';
END IF;

IF p_last_name IS NOT NULL THEN
    l_sql_string := l_sql_string||'last_name = :2,';
END IF;

IF p_middle_name IS NOT NULL OR p_middle_name_null = 'Y' THEN
    l_sql_string := l_sql_string||'middle_name = :3,';
END IF;

IF p_dob IS NOT NULL THEN
    l_sql_string := l_sql_string||'dob = :4,';
END IF;

IF p_termination IS NOT NULL OR p_termination_null = 'Y' THEN
    l_sql_string := l_sql_string||'termination = :5,';
```

## DYNAMIC SQL UPDATES

```
END IF;

l_sql_string := SUBSTR(l_sql_string,1,(LENGTH(l_sql_string) -1))||'
WHERE id = :6; END;';

EXECUTE IMMEDIATE l_sql_string
USING p_first_name, p_last_name, p_middle_name, p_dob, p_termination,
p_id;

COMMIT;

EXCEPTION
  WHEN OTHERS THEN
    po_return_code := -1;
    po_return_string := 'UPDATING EMPLOYEE '||SUBSTR(SQLERRM,1,200);
END;
```

---

### EXTENSIONS

---

A very powerful extension of this method is to combine a code generator with a stored procedure interface. In previous engagements, Blueoak personnel have written code generators that automatically read the schema information from a database and some configuration meta-data for package organization to construct all of the stored procedures at once grouped into packages. The method described above is used for all table updates. The code generator reduces the impact of schema changes and has reduced overall coding efforts by 50% or more, but that is a topic for another paper. Blueoak personnel have successfully used the technique described above on a schema with over 400 tables, and it has proven itself to be stable, reliable, and much faster than using either pure dynamic SQL or client constructed queries.

---

### OTHER DATABASES

---

Similar dynamic SQL structures exist in Microsoft SQL Server, Sybase, and DB2. The methods described here, including the code generator, could be implemented in almost any major RDBMS system.