PL/SQL and Bind Variable: the two ways to increase the efficiency of Network Databases

Hitesh KUMAR SHARMA
Assistant Professor, ITM University
Ranjit BISWAS
Associate Director Faculty, Manav Rachna International University Faridabad
Aditya SHASTRI
Vice-Chancellor, Banasthali University,Rajasthan-304022, India
hiteshsharma@itmindia.edu, adityashastri@yahoo.com, ranjitbiswas@yahoo.com

Modern data analysis applications are driven by the Network databases. They are pushing traditional database and data warehousing technologies beyond their limits due to their massively increasing data volumes and demands for low latency. There are three major challenges in working with network databases: interoperability due to heterogeneous data repositories, proactively due to autonomy of data sources and high efficiency to meet the application demand. This paper provides the two ways to meet the third challenge of network databases. This goal can be achieved by network database administrator with the usage of PL/SQL blocks and bind variable. The paper will explain the effect of PL/SQL block and bind variable on Network database efficiency to meet the modern data analysis application demand.

***Key Words**: Network Database, Web Application, Bind Variable, PL/SQL, Middleware.*

1 Introduction

Most Web applications of any size involve the use of a database. They are pushing traditional database and data warehousing technologies beyond their limits. [1]. Typically, a web application allows the addition or creation of new records (for example, when a new user registers on the site), and the reading and searching of many records in a database. The most common bottleneck when developing a Web application is in the reading of a large number of records from a database, or executing a particularly complex SELECT statement against the database. Writing to or updating a database usually is performed on a small number of records at a time. This is often much less of an issue than cases that involve reading thousands of records at a time.

By eliminating unused fields from SELECT statements, we can reduce the complexity of the query and reduce the amount of data sent over the network (or at least between the database server and the web script). The net affect of making such changes is a reduced database read time.

PL/SQL is the language of choice for data-centric application development in Network databases. In most programming languages, database work involves connecting to the server, mapping datatypes and manually preparing and processing result sets. PL/SQL is a procedural language that is so tightly integrated with the SQL language that most of these tasks are either eliminated completely or incredibly simple.

The two major costs decide the performance of a network database.

1.1. Cost 1: Round Trips

The first basic cost of retrieving data is the "round trip". Database programmers speak of a "round trip" as occurring whenever you send a request the server and retrieve some results. Each round trip to the server carries some overhead, as the server must do some basic work to allocate and release resources at the start and end of the request. This overhead is added to the base

cost the server must pay to actually go out to disk to find and retrieve your data.
If your application makes more round trips than are necessary, then the program will be slower than it could be.

1.2. Cost 2: Retrieval Size

An application runs code (such as a Java servlet) that makes queries to a backend database to customize the content, typically based on the user's request or a stored user profile. Overloading the work of the application server (e.g., executing the Java servlets) to proxy nodes is not difficult, but the central database server remains a performance bottle- neck [2].
Every byte that the application retrieves from the server carries a cost at several points. The server must go to disk and read it, the wire must carry the load from the db server to the web server, and the web server must hold the result in memory. If your web code regularly retrieves more information than it needs, then the program will be slower than it could be.

2. Related work

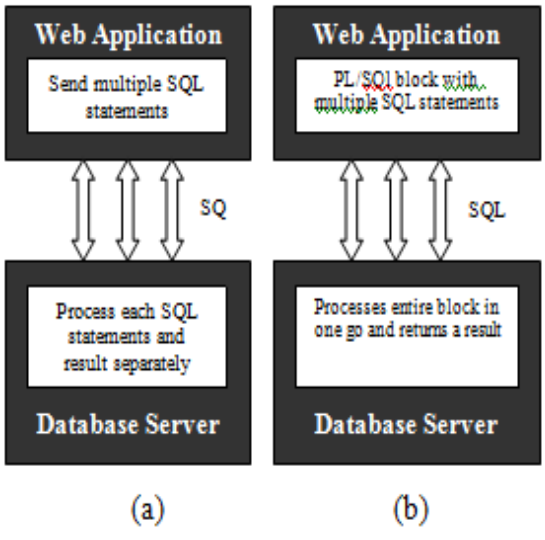
Companies across all industries are seeing very steep increases in the amount of data they must process. For example, one recent study [9] has estimated that the amount of data stored in data warehouses has been growing by an average of 173% per year across all industries. This rate of growth is substantially faster than the typical 12 to 18-month doubling of hardware capacity as dictated by Moore's law, Shuggart's law and others. As a result, for data analytics workloads hardware continues to become slower relative to the demands being placed on it.
The internet is a collection of millions upon millions of local, regional, national, and global networks. It commenced in 1969 with four supercomputers across the U.S. networked to the Pentagon.

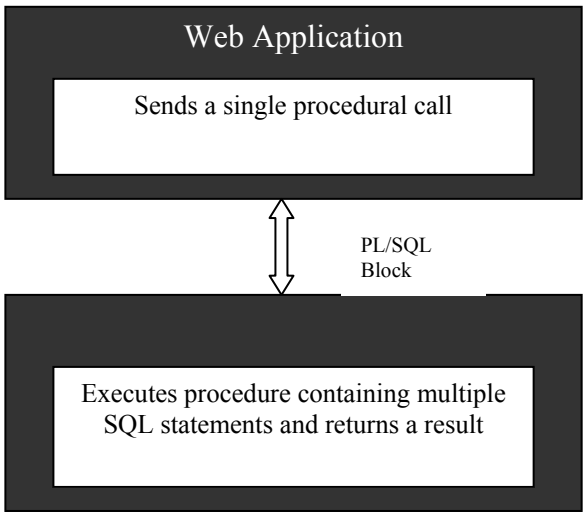
Very few people used the Internet for the first 20 years. Explosive growth took place after 1992 when the Netscape Navigator web browser incorporated HTML protocols to read HTML codes invented by particle physicists in Switzerland in 1990. This was the beginning of the "first generation" of network computing on what became known as the world wide web (WWW) or simply the "web." The first generation was mainly one way flows of information from web server computers to client user computers on the web. At this same time, the first generation of database interactive computing was confined to local and wide area networks (LANs and WANs). Although data files could be transmitted across the Internet using FTP and other protocols, databases could not interact on the WWW or the Internet as a whole. Most web applications are still in the first HTML generation.
The second generation of networking and databases followed quickly when web servers commenced to interact in a more formal way with remote clients on the Internet. With special types of **middleware** software, database servers could process data transmitted back from remote Internet client computers. For example, customer orders and market surveys could be processed and server-side databases could be updated without human intervention. Middleware CGI scripting and later ActiveX and Java software enabled web servers, database servers, and remote clients on the Internet to become more interactive. The second generation is relatively new and growing in popularity at this time.
The third generation is only just emerging and is hard to put into words. It is best described as distributed network computing. In the second generation, middleware links to" front ends" of database servers on the server side when clients transmit signals. In the third generation, databases can be distributed globally and can communicate with each other with "back-end" distributed network computing. There is virtually no difference between having all databases on one computer with one operator versus having

databases on 100 computers with 100 operators residing anywhere in the world.
The third generation of network database is using three-tier architecture, Client/Server with Middleware layer. When business logic is processed in a client application, it is often necessary to pass a succession of statements between the application and the database server. Storing application code in the database takes this a step further as application logic is removed from the client layer and precompiled in the database, allowing modification without the need for a redeployment of client software. Each request and response involves network traffic, which can greatly affect overall performance. Running application logic on the database server can increase efficiency by reducing network traffic.
This paper explains the two ways to increase the efficiency of network databases by reducing network traffic.

3. The First way: Placing of PL/SQL block on Database server

PL/SQL is the language for data-centric application development [4] in Network databases. In most programming languages, database work involves connecting to the server, mapping datatypes and manually preparing and processing result sets. PL/SQL is a procedural language that is so tightly integrated with the SQL language that most of these tasks are either eliminated completely or incredibly simple.
The datatypes available in PL/SQL are a superset of those available in SQL, so datatype conversions between SQL and PL/SQL are rarely needed. As a result PL/SQL allows interaction with both the data and metadata of database objects with greater ease and efficiency than is possible with most other languages. In addition PL/SQL supports dynamic SQL allowing statements to be created at runtime for greater flexibility.
Running application logic as PL/SQL on the database server can increase efficiency by reducing network traffic. When business logic is processed in a client application, it is often necessary to pass a succession of statements between the application and the database server. Each request and response involves network traffic, which can greatly affect overall performance.
Passing a PL/SQL block containing multiple statements to the server can reduce network round trips, thereby improving performance. Storing application code in the database takes this a step further as application logic is removed from the client layer and precompiled in the database, allowing modification without the need for a redeployment of client software.
The PL/SQL language is available on all platforms, making it significantly more portable than many programming languages. When application logic is located within the database, changes in client programming models have a reduced impact, as only presentation of the data is controlled at that level. (See Fig 1.1)





(c)

Figure 1: PL/SQL to improve performance , Fig 1(a) shows that heavy network usage to communicate with the server with separate SQL statements, Fig 1(b) shows that low network usage to communicate with the server usig PL/SQL, Fig 1(c) shows that very low network usage to communicate with the server by placing PL/SQL processing logic on Server,

Centralizing application logic enables a higher degree of security and productivity. The use of Application Program Interfaces (APIs) can abstract complex data structures and security implementations from client application developers, leaving them free to do what they do best.

3.1 PL/SQL Architecture

The PL/SQL language is made up of both procedural code and SQL statements. When valid PL/SQL code is executed, the PL/SQL engine executes all procedural code and sends SQL statements to the SQL engine of the database server. Figure 1.2 represents this process in action for a PL/SQL block.

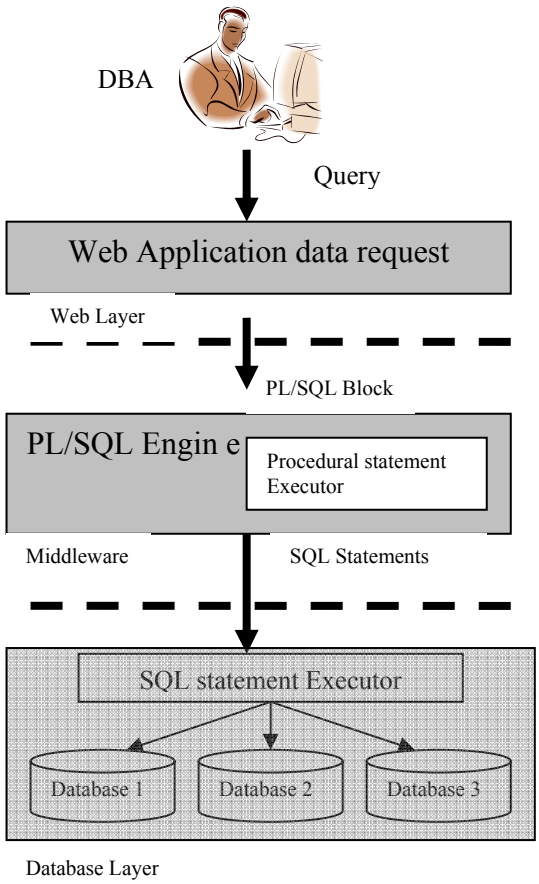


Figure 1.2 – PL/SQL Architecture.

The database contains a PL/SQL engine, which is used to execute all stored procedures, functions, packages, objects and triggers. This allows application logic to be processed entirely within the database layer.

Recently, a number of systems have been proposed with a similar architecture for scaling the delivery of database-backed dynamic content [5, 7, 8, 9]. In each of these systems users interact with proxy servers that mimic a traditional three-tiered architecture (containing a web server to handle user requests, an application server to generate dynamic content, and a database server as a backend data repository).

Some application development tools, such as Oracle Forms and Oracle Reports, have their own PL/SQL engine, allowing procedural logic to be processed with no reference to the database server.

3.2 Overview of PL/SQL Elements

Blocks in PL/SQL

Blocks are the organizational unit for all PL/SQL code, whether it is in the form of an anonymous block, procedure, function, trigger or type. A PL/SQL block is made up of three sections: declaration, executable and exception. Only the executable section is mandatory.

```
[DECLARE
 -- delarations]
BEGIN
 -- statements
[EXCEPTION
 -- handlers
END;
```

Based on this definition, the simplest valid block is shown below, but it does not do anything.

```
BEGIN
 NULL;
END;
```

The optional declaration section allows variables, types, procedures and functions do be defined for use within the block. The scope of these declarations is limited to the code within the block itself, or any nested blocks or procedure calls. The limited scope of variable declarations is shown by the following two examples. In the first, a variable is declared in the outer block and is referenced successfully in a nested block. In the second, a variable is declared in a nested block and referenced from the outer block, resulting in an error as the variable is out of scope.

```
DECLARE
 l_number  NUMBER;
BEGIN
 l_number := 1;
 BEGIN
  l_number := 2;
 END;
END;
/
PL/SQL procedure successfully completed.
BEGIN
 DECLARE
  l_number  NUMBER;
 BEGIN
  l_number := 1;
 END;  l_number := 2;
END;
/
 l_number := 2;
 *
ERROR at line 8:
ORA-06550: line 8, column 3:
PLS-00201: identifier 'L_NUMBER' must be
declared
ORA-06550: line 8, column 3:
PL/SQL: Statement ignored
SQL>
```

The main work is done in the mandatory executable section of the block, while the optional exception section is where all error processing is placed. The following two examples demonstrate the usage of exception handlers for trapping error messages. In the first, there is no exception handler so a query returning no rows results in an error. In the second, the same error is trapped by the exception handler, allowing the code to complete successfully.

```
DECLARE
 l_date  DATE;
BEGIN
 SELECT SYSDATE
 INTO  l_date
 FROM  dual
 WHERE  1=2; -- For zero rows
END;
/
DECLARE
*
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at line 4
DECLARE
 l_date  DATE;
BEGIN
 SELECT SYSDATE
 INTO  l_date
 FROM  dual
 WHERE  1=2; -- For zero rows
EXCEPTION
 WHEN NO_DATA_FOUND THEN
  NULL;
END;
/
PL/SQL procedure successfully completed.
```

4. The Second way: Using Bind Variables

For every statement issued against the server, Oracle searches the shared pool to see if the statement has already been parsed. If an exact text match of the statement is already present in the shared pool a soft parse is performed as the execution plan for the statement has already been created and can be reused. If the statement is not found in the shared pool a hard parse must be performed to determine the optimal execution path.

The important thing to remember from the previous paragraph is the term "exact text match", as different numbers of spaces, literal values and case will result in a failure to find a text match, such that the following statements are considered different.

SELECT 1 FROM dual WHERE dummy = 'X';
SELECT 1 FROM dual WHERE dummy = 'Y';
SELECT 1 FROM DUAL WHERE dummy = 'X';
SELECT 1 FROM dual WHERE dummy = 'X';

The first two statements only differ by the value of the search criteria, specified using a literal. In these situations exact text matches can be achieved by replacing the literal values with bind variables that have the correct values bound to them. Using the previous example the statement passed to the server might look like this.

```
SELECT 1 FROM dual WHERE dummy = :B1;
```

For every execution the bind variable may have a different value, but the text sent to the server is the same allowing for an exact text, which results in a soft parse.

There are two main problems associated with applications that do not use bind variables:

- Parsing SQL statements is a CPU intensive process, so reparsing similar statements constantly represents a waste of CPU cycles.
- Parsed statements are stored in the shared pool until they are aged out. By not using bind variables the shared pool can rapidly become filled with similar statements, which waste memory and make the instance less efficient.

The Script_bind.sql script illustrates the problems associated with not using bind variables by using dynamic SQL to simulate an application sending insert statements to the server.

Script_bind.sql

```
CREATE TABLE bind_variables (
 code  VARCHAR2(10)
);
BEGIN
  -- Perform insert without bind variables.
  FOR i IN 1 .. 10 LOOP
    BEGIN
      EXECUTE IMMEDIATE
       'INSERT INTO bind_variables (code)
VALUES ('' || i || '')';
    EXCEPTION
      WHEN NO_DATA_FOUND THEN
        NULL;
    END;
  END LOOP;
  -- Perform insert with bind variables.
  FOR i IN 1 .. 10 LOOP
    BEGIN
      EXECUTE IMMEDIATE
       'INSERT INTO bind_variables (code)
VALUES (:B1)' USING TO_CHAR(i);
    EXCEPTION
      WHEN NO_DATA_FOUND THEN
        NULL;
    END;
  END LOOP;
  COMMIT;
END;
/
-- Display the associated SQL text.
COLUMN sql_text FORMAT A60
COLUMN executions FORMAT 9999
SELECT sql_text,
       executions
FROM   v$sql
WHERE  INSTR(sql_text, 'INSERT INTO
bind_variables') > 0
AND    INSTR(sql_text, 'EXECUTE') = 0
ORDER BY sql_text;
DROP TABLE bind_variables;
```

The script starts by creating a test table and executing a simple insert statement 10 times, where the insert statement concatenates a value into the string rather than using a bind variable. Next it repeats this process but this time uses a bind

variable rather than concatenating the value into the string. Finally it displays the SQL text parsed by the server and stored in the shared pool, which requires query access on the v$sql view. The results from the script are displayed below

```
* SQL> @ Script_bind.sql
Table created.
PL/SQL procedure successfully completed.
SQL_TEXT
 EXECUTIONS
---------------------------------------------------
----- ----------
insert into bind_variables (code) values
('1')     1
insert into bind_variables (code) values
('10')    1
insert into bind_variables (code) values
('2')     1
insert into bind_variables (code) values
('3')     1
insert into bind_variables (code) values
('4')     1
insert into bind_variables (code) values
('5')     1
insert into bind_variables (code) values
('6')     1
insert into bind_variables (code) values
('7')     1
insert into bind_variables (code) values
('8')     1
insert into bind_variables (code) values
('9')     1
insert into bind_variables (code) values
(:b1)     10
11 rows selected.
Table dropped.
```

From this we can see that when bind variables were not used the server parsed and executed each query as a unique statement, whereas the bind variable statement was parsed once and executed 10 times. This clearly demonstrates how applications that do not use bind variables can result in wasted memory in the shared pool, along with increased CPU usage.

5. Conclusion

It is not difficult to create database applications that perform well. The basic rules of thumb are to make a minimum number of round trips to the server and to retrieve precisely the values that you need and no more. These ideas work well because they minimize your most expensive operation, which is disk access. the procedures/functions are stored in the database and are, therefore, executed on the database server which is likely to me more powerful than the clients which in turn means that stored procedures should run faster;
the code is stored in a pre-compiled form which means that it is syntactically valid and does not need to be compiled at run-time, thereby saving resources; each user of the stored procedure/function will use exactly the same form of queries which means the queries are reused thereby reducing the parsing overhead and improving the scalability of applications; as the procedures/functions are stored in the database there is no need to transfer the code from the clients to the database server or to transfer intermediate results from the server to the clients. This results in much less network traffic and again improves scalability when using PL/SQL packages, as soon as one object in the package is accessed, the whole package is loaded into memory which makes subsequent access to objects in the package much faster stored procedures/functions can be compiled into “native” machine code making them even faster. there is an overhead involved in switching from SQL to PL/SQL, this may be significant in terms of performance but usually this overhead is outweighed by performance advantages of using PL/SQL more memory may be required when using packages as the whole package is loaded into memory as soon as any object in the package is accessed native compilation can take twice as long as normal compilation

References

[1] Agrawal, R., et al. "The Claremont Report on Database Research", http://db.cs.berkeley.edu/claremont/, May 2008.
[2] C. Olston, A. Manjhi, C. Garrod, A. Ailamaki, B. Maggs, and T. Mowry. A scalability service for dynamic web applications. In Proc. Conference on Innovative Data Systems Research (CIDR), 2005.
[3] K. Burleson, Donald "Creating a self-tuning Oracle database"
[4] John Garmany "Easy Oracle PL/SQL Programming:" Get Started Fast with Working PL/SQL Code .
[5] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A dynamic data cache for Web applications. In Proc. International Conference on Data Engineering, 2003.
[6] M. Altinel, C. Bornhovd, S.Krishnamurthy, C. Mohan, H. Pirahesh, and B. Reinwald. Cache tables: Paving the way for an adaptive database cache. In Proc. International Conference on Very Large Data Bases, 2003.
[7] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, and J. F. Naughton. Middle-tier database caching for e-business. In Proc. ACM SIGMOD International Conference on Management of Data, 2002.
[8] Winter, R, "Why Are Data Warehouses Growing So Fast?",B-eye Network, http://www.b-eye-network.com/view/7188,April 2008

Hitesh KUMAR SHARMA, The author is An Assistant Professor in ITM University. He has published 8 research papers in National Journals and 1 research paper in International Journal. Currently He is pursuing his Ph.D. in the area of database tuning**.**

Ranjit BISWAS, Associate Director, MIRU, Faridabad, Published about 100 research papers in International journals/bulletins of USA/Europe, out of which more than 40 papers are independently published papers and the rest are published jointly with other authors (with Ph.D. scholars). Haryana, INDIA.

Aditya SHASTRI, Ph.D. MIT, Published about 200 research papers in international journals on Graph Theory with applications in Communication, Computer Graphics and Parallel Processing ,Vice Chancellor, Director, Banasthali University, Banasthali, INDIA