View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by CERN Document Server

Basic database performance tuning - developer's perspective

Michal Kwiatek
CERN, Geneva, Switzerland

Abstract
This lecture discusses selected database performance issues from the developer's point of view: connection overhead, bind variables and SQL injection, making most of the optimizer with up-to-date statistics, reading execution plans. Prior knowledge of SQL is expected.

1 Introduction

Application developers tend to perceive the problem of database performance as the responsibility of the database administrators (DBAs). This is a misconception, because with a badly designed application there is little a DBA can do.

It is true that the DBA often has the intimate knowledge of the database. However, the developer has the intimate knowledge of the application which allows him or her to design and implement the application so that it fulfils the requirements in the fastest possible way.

This lecture explains selected elements of the Relational Database Management System (RDBMS). Oracle RDBMS is used in the examples, but similar mechanisms exist in other database systems. Understanding these mechanisms will enable the developers to increase the efficiency of their applications.

Indexes, which are of paramount importance to database performance, are not discussed in this lecture. The reader should therefore turn to other sources to learn how to effectively use database indexes.

2 Connection overhead

Problems with database performance can occur as soon as one connects to the database. In the most common scenario, when a client first connects to an Oracle database, a dedicated server process is created to serve this and all the following requests for this client.

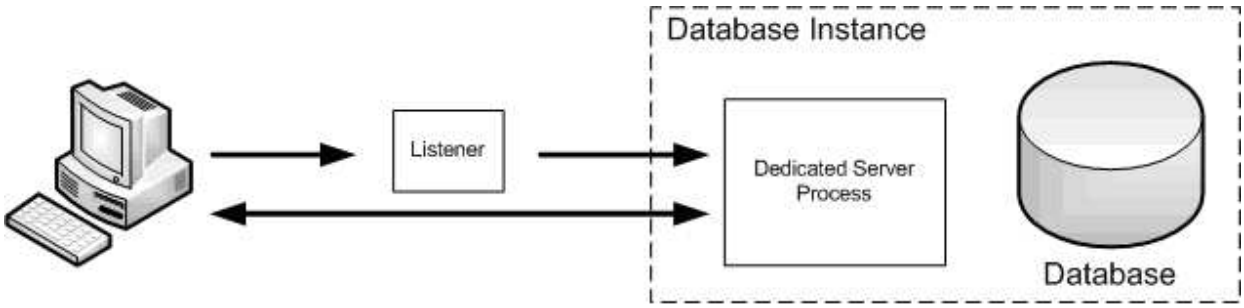

Fig. 1: Oracle dedicated server architecture, based on [1]

Creating a new process is a time consuming operation. Indeed, the time needed to open a connection is the matter of seconds, not milliseconds. This is not a problem for a single-threaded application where the connection is opened once and then used for many subsequent queries. However, in a web application serving hundreds of requests per minute where each request needs only one short query to complete, opening a separate connection for every request can create a non-negligible overhead.

To solve his problem, using connection pooling should be considered. Create a pool of connections that are open only once and shared among the threads of the application according to needs. When a

database connection is needed, it is taken from the pool. When it is no longer needed, it is returned to the pool. When the number of threads that need a connection is larger than the number of available connections some threads just wait.

This is a standard solution and no new code has to be written for this purpose: ready-to-use implementations exist for most of today's programming languages.

3 Performance cost of risk

Once connected to the database, one probably wants to query some data. Let's have a quick look at what happens when an SQL query is run.

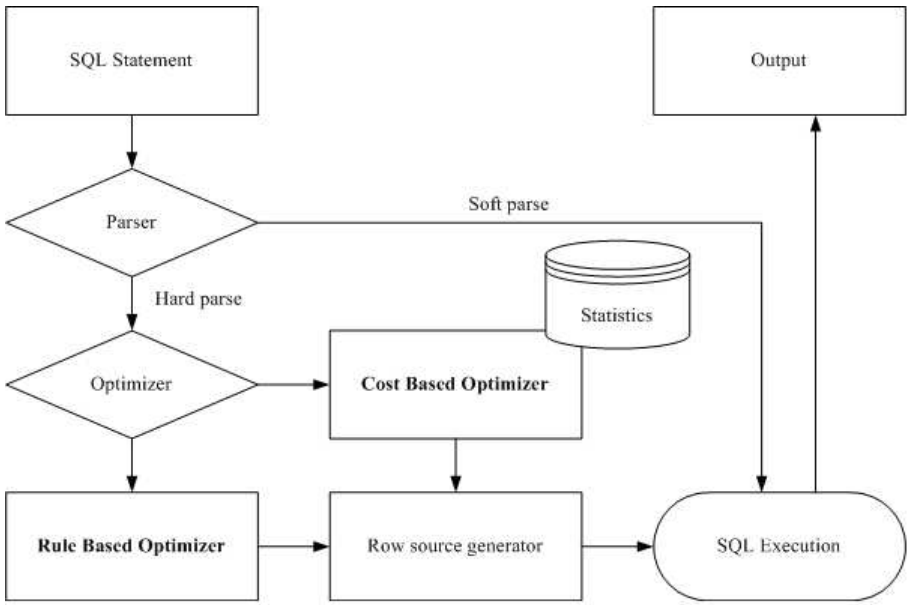

Fig. 2: Running a SQL query

The query first has to be checked, both syntactically and semantically. It is done by a parser. Then it is transformed into row-level operations that will actually be run. This is called an execution plan. For a given SQL query, there are many semantically correct execution plans - many different ways to achieve the same result. The optimizer analyses the query to create the best execution plan possible. But the optimization itself is time consuming, so every time an execution plan is created, it is put into a special memory structure, called Library Cache, together with the digest of the query. Next time the same query is executed, optimization is skipped. This is called a soft parse, as opposed to the hard parse, which includes the optimization.

Let us now consider the example listing in Fig. 3

```
1: String myName = "Kwiatek"; //value from user input
2: String sql = "select sal from emp where ename = '"+myName+"'";
3: Statement stmt = conn.createStatement(sql);
4: ResultSet rs = stmt.executeQuery(sql);
```

Fig. 3: SQL query with a hardcoded literal value

It shows an SQL query invoked from the Java programming language. When line 3 is executed, the query "select sal from emp where ename = 'Kwiatek'" is submitted to the database. If such a query is run for

many different employees, and a different name is submitted every time - will there be a hard or a soft parse every time this query is run?

From the point of view of the database, the query "select sal from emp where ename = 'Kwiatek'" is different from the query "select sal from emp where ename = 'Smith'", because their digests are different. So every time such a query is run, it will be hard- parsed. In consequence, the database will be unnecessarily loaded. This can be observed when a large number of such queries are run in parallel.

But the code shown on the listing in Fig. 3 has a more important problem. Imagine variable myName is assigned a value of "O'Brian". In this case, the query, as constructed in line 2, will be: ""select sal from emp where ename = 'O'Brian'". This query will fail: it is syntactically incorrect, because it contains 3 single-quotes characters. So, using the code as presented in Fig. 3 , one runs the risk of having errors in the application!

Very often the data, used as parameters to queries, is actually taken from user input. So depending on user input, the application will work fine or crash. This is very bad programming style. What is worse, control over the query is lost: a malevolent user could try to hack it by submitting a value that closes the apostrophe, adds an unauthorised clause (for example a union with a table the user should not be allowed to see) and the second apostrophe to make such crafted query syntactically correct. In short, the application becomes vulnerable to SQL injection. This is insecure.

One could of course change the code in such a way, that all special characters in user input are replaced, as shown on the listing in Fig. 4. This will protect the application from SQL injection, provided that all special characters are effectively encoded. But it will not help against a hard parse every time such a query is executed.

```
1: String myName = "Kwiatek"; //value from user input
2: String sql =
"select sal from emp where ename='"+myName.replaceAll("'","''")+"'";
3: Statement stmt = conn.createStatement(sql);
4: ResultSet rs = stmt.executeQuery(sql);
```

Fig. 4: Manually encoding special characters in an SQL query

There is an easy solution to both these issues: bind variables. Using placeholders for values, instead of the literal values in the SQL query, makes the query secure. What is more, the digest of such a query is the same no matter what values are submitted, so the existing query plan can be retrieved from the library pool and reused. The listing in Fig. 5 shows the syntax for using bind variables in the Java language. Again, similar syntax exists in most of today's computer languages.

```
1: String myName = "Kwiatek"; //value from user input
2: String sql = "select sal from emp where ename = ?";
3: PreparedStatement stmt = conn.prepareStatement(sql);
4: stmt.setString(1, myName);
5: ResultSet rs = stmt.executeQuery();
```

Fig. 5: SQL query with bind variables

4 Can a blindfolded optimizer be effective?

Looking again at Fig. 2, two types of optimizer can be seen: the Rule Based Optimizer (RBO) and the Cost Based Optimizer (CBO).

RBO prepares the execution plan based on a predefined set of rules. No matter what the data actually is, RBO will always come up with the same execution plan for the given query. This is bad, because the distribution of the data has important impact on the time needed to query it.

CBO, on the other hand, uses table and index statistics to estimate the cost of running the query. This is much more accurate but recent statistics are needed.

The listing in Fig. 6 shows an Oracle PL/SQL procedure call that will gather statistics for all objects in the current schema. See [2] for an explanation of parameters used. Whenever an investigation of query performance is needed, it is recommended to start by running this procedure to make sure that the optimizer is at its best.

```
BEGIN
DBMS_STATS.GATHER_SCHEMA_STATS(
  ownname=>null,
  estimate_percent=>DBMS_STATS.AUTO_SAMPLE_SIZE,
  method_opt=>'FOR ALL COLUMNS SKEW ONLY',
  cascade=>TRUE
);
END;
```

Fig. 6: Gathering statistics for all objects in the current schema

CBO is so much more effective that in Oracle 10g RBO disappears altogether. Statistics are automatically gathered on regular basis. Still, in case of performance problems, it is worth refreshing the statistics.

5 The black box syndrome

SQL is a declarative language: its syntax lets us specify what results we want but not how to get them. This is very convenient, because the complexity of the database operation is hidden from us as if the database was a black box that produces the results without revealing how.

It works very well in most cases: RDBMS takes care of the tedious technical details. However, when performance problems arise, it is a signal that the automatic operation of the database is not as effective as it is supposed to be. Clearly, a human investigation is needed. The first step is to understand what is actually going on within the black box.

Oracle provides mechanisms that let us investigate with different levels of complexity. Some mechanisms, like tkprof (see [3] to learn how to use tkprof at CERN), require shell access to the database server, so are reserved to the DBAs. Other, like the EXPLAIN PLAN command, can be very well used by the developers.

The EXPLAIN PLAN command lets us read the execution plan of our query. To use it, we first need to create the PLAN_TABLE where the plan will be stored for us to see. We just need to run the $ ORACLE_HOME/rdbms/admin/utlxplan.sql script (it can be found on the Internet) and we can start using EXPLAIN PLAN for our queries. After the query has been explained, we can see it using the dbms_xplan.display function, as shown on the listing in Fig. 7.

The table on the listing shown in 7 contains the tree of operations that constitute the execution plan. Next to each operation we can see the number of rows, the number of bytes and the relative CPU cost of this operation. To have a complete picture, we need to add the predicate information presented below the table. The complete tree is presented in Fig. 8.

The last piece of information we need is the meaning of the access paths and join methods used in the execution plan (see [4] for a complete list). The nested loops predicate at the root of the tree means that for every row returned by the first (upper) branch, the second (lower) branch is executed and their results are joined. In our example, the table EMP will be searched row-by-row (this is the meaning of

```
Connected to:
Oracle9i Enterprise Edition Release 9.2.0.6.0 - Production
With the Partitioning, OLAP and Oracle Data Mining options
JServer Release 9.2.0.6.0 - Production

DEVDB:SQL> explain plan for select e.ename emp, m.ename mgr
  2  from emp e, emp m
  3  where e.mgr = m.empno
  4  and e.deptno = 10;

Explained.

DEVDB:SQL> select * from table(dbms_xplan.display);

----------------------------------------------------------------------------------
| Id  | Operation                     |  Name         | Rows  | Bytes | Cost (%CPU)|
----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |               |     3 |    69 |    12   (9)|
|   1 |  NESTED LOOPS                 |               |     3 |    69 |    12   (9)|
|*  2 |   TABLE ACCESS FULL           | EMP           |     3 |    39 |     9  (12)|
|   3 |   TABLE ACCESS BY INDEX ROWID| EMP           |     1 |    10 |     2  (50)|
|*  4 |    INDEX UNIQUE SCAN          | EMP_EMPNO_PK  |     1 |       |            |
----------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("E"."DEPTNO"=10 AND "E"."MGR" IS NOT NULL)
   4 - access("E"."MGR"="M"."EMPNO")
```

Fig. 7: Reading the execution plan of your query

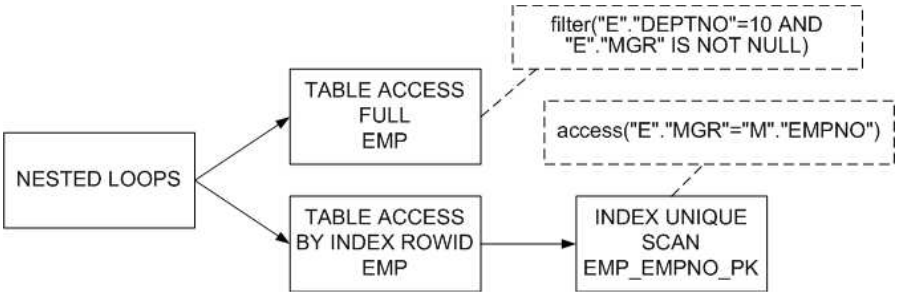

Fig. 8: The execution plan presented in a tree

the "table access full" access path) but only the rows which satisfy the filter criteria will be returned. For each such row, the lower branch will be executed: first index EMP_EMPNO_PK will be scanned with the unique condition to find the ROWID needed to get the full row by ROWID in the second step. ROWIDs in Oracle are basically physical addresses of data (see also [5]). In a semi-formal imperative language, it would look like the listing in Fig. 9.

```
For each row r1 in
      (select * from emp where deptno=10 and mgr is not null)
Loop
      Find rowid of row r2 using index emp_empno_pk;
      Get row r2 by rowid;
      Output r1.ename, r2.ename;
End loop
```

Fig. 9: The execution plan in a semi-formal imperative language

In conclusion, Oracle, as well as other RDBMS vendors, gives us the tool to look inside the black box and understand the execution plan of SQL queries. This is the first thing to do for a developer who investigates performance problems. See also [6] for a wider explanation of how to read execution plans.

6 Summary: the importance of being earnest

To sum up, the developer of a database application has even more potential to tune for performance than the DBA. This is because the developer knows the application: its structure, its use cases and its potential performance bottlenecks.

Achieving higher performance requires following best practices (some of which are discussed in this lecture) and thinking about performance before problems actually arise. To investigate performance issues, one needs to earnestly try to understand what the application does on the database side. It takes some work to get this understanding, but this work is most easily done by the developer.

Understanding certain points about the RDBMS is required, and although this lecture touches some areas of RDBMS, it is far from being exhaustive. In particular, indexes, which are substantial to database performance, are not discussed in this lecture at all. The reader should therefore turn to other sources to further enhance his or her knowledge of the database. A very good source of information is "AskTom", the question and answer service hosted by Oracle (see [7]).

References

[1] Oracle9i Net Services Administrator's Guide, Dedicated Server Processes.
`http://download-uk.oracle.com/docs/cd/B10501_01/network.920/a96580/architec.htm#1048705`

[2] Oracle9i Supplied PL/SQL Packages and Types Reference, GATHER_DATABASE_STATS Procedure.
`http://download-uk.oracle.com/docs/cd/B10501_01/appdev.920/a96612/d_stats2.htm`

[3] Michal Kwiatek, Performance Optimization and Tuning, iCSC 2005.
`http://indico.cern.ch/getFile.py/access?contribId=s5t4&sessionId=5&resId=1&materialId=0&confId=a05859`

[4] Oracle9i Database Performance Tuning Guide and Reference, Understanding Access Paths for CBO.
`http://download-uk.oracle.com/docs/cd/B10501_01/server.920/a96533/optimops.htm#35891`

[5] Thomas Kyte, Effective Oracle by Design, Oracle Press, ISBN 0-07-223065-7, p. 454-455.

[6] Ibid. p. 84-87.

[7] Ask Tom Oracle, the Question and Answer service edited by Thomas Kyte.
`http://asktom.oracle.com/`