Chapter 11

Database Performance Tuning and Query Optimization

Discussion Focus

This chapter focuses on the factors that directly affect database performance. Because performance-tuning techniques can be DBMS-specific, the material in this chapter may not be applicable under all circumstances, nor will it necessarily pertain to all DBMS types.

This chapter is designed to build a foundation for the general understanding of database performance-tuning issues and to help you choose appropriate performance-tuning strategies. (For the most current information about tuning your database, consult the vendor's documentation.)

- Start by covering the basic database performance-tuning concepts. Encourage students to use the web to search for information about the internal architecture (internal process and database storage formats) of various database systems. Focus on the similarities to lay a common foundation.
- Explain how a DBMS processes SQL queries in general terms and stress the importance of indexes in query processing. Emphasize the generation of database statistics for optimum query processing.
- Step through the query processing example in section 11-4, Optimizer Choices.
- Discuss the common practices used to write more efficient SQL code. Emphasize that some practices are DBMS-specific. As technology advances, the query optimization logic becomes increasingly sophisticated and effective. Therefore, some of the SQL practices illustrated in this chapter may not improve query performance as dramatically as it does in older systems.
- Finally, illustrate the chapter material using the query optimization example in section 11-8.

Answers to Review Questions

1. What is SQL performance tuning?

SQL performance tuning describes a process – on the client side – that will generate an SQL query to return the correct answer in the least amount of time, using the minimum amount of resources at the server end.

2. What is database performance tuning?

DBMS performance tuning describes a process – on the server side – that will properly configure the DBMS environment to respond to clients' requests in the fastest way possible, while making optimum use of existing resources.

3. What is the focus of most performance tuning activities, and why does that focus exist?

Most performance-tuning activities focus on minimizing the number of I/O operations, because the I/O operations are much slower than reading data from the data cache.

4. What are database statistics, and why are they important?

The term *database statistics* refers to a number of measurements gathered by the DBMS to describe a snapshot of the database objects' characteristics. The DBMS gathers statistics about objects such as tables, indexes, and available resources -- such as number of processors used, processor speed, temporary space available, and so on. Such statistics are used to make critical decisions about improving the query processing efficiency.

5. How are database statistics obtained?

Database statistics can be gathered manually by the DBA or automatically by the DBMS. For example, many DBMS vendors support the SQL's ANALYZE command to gather statistics. In addition, many vendors have their own routines to gather statistics. For example, IBM's DB2 uses the RUNSTATS procedure, while Microsoft's SQL Server uses the UPDATE STATISTICS procedure and provides the Auto-Update and Auto-Create Statistics options in its initialization parameters.

6. What database statistics measurements are typical of tables, indexes, and resources?

For tables, typical measurements include the number of rows, the number of disk blocks used, row length, the number of columns in each row, the number of distinct values in each column, the maximum value in each column, the minimum value in each column, what columns have indexes, and so on.

For indexes, typical measurements include the number and name of columns in the index key, the number of key values in the index, the number of distinct key values in the index key, histogram of key values in an index, etc.

For resources, typical measurements include the logical and physical disk block size, the location and size of data files, the number of extends per data file, and so on.

7. **How is the processing of SQL DDL statements (such as CREATE TABLE) different from the processing required by DML statements?**

   A DDL statement actually updates the data dictionary tables or system catalog, while a DML statement (SELECT, INSERT, UPDATE and DELETE) mostly manipulates end user data.

8. **In simple terms, the DBMS processes queries in three phases. What are those phases, and what is accomplished in each phase?**

   The three phases are:

   1. *Parsing*. The DBMS parses the SQL query and chooses the most efficient access/execution plan.
   2. *Execution*. The DBMS executes the SQL query using the chosen execution plan.
   3. *Fetching*. The DBMS fetches the data and sends the result set back to the client.

   Parsing involves breaking the query into smaller units and transforming the original SQL query into a slightly different version of the original SQL code -- but one that is "fully equivalent" and more efficient. *Fully equivalent* means that the optimized query results are always the same as the original query. More efficient means that the optimized query will, almost always, execute faster than the original query. (Note that we say *almost* always because many factors affect the performance of a database. These factors include the network, the client's computer resources, and even other queries running concurrently in the same database.)

   After the parsing and execution phases are completed, all rows that match the specified condition(s) have been retrieved, sorted, grouped, and/or – if required – aggregated. During the fetching phase, the rows of the resulting query result set are returned to the client. During this phase, the DBMS may use temporary table space to store temporary data.

9. **If indexes are so important, why not index every column in every table? (Include a brief discussion of the role played by data sparsity.)**

   Indexing every column in every table will tax the DBMS too much in terms of index-maintenance processing, especially if the table has many attributes, many rows, and/or requires many inserts, updates, and/or deletes.

One measure to determine the need for an index is the data *sparsity* of the column you want to index. Data sparsity refers to the number of different values a column could possibly have. For example, a STU_SEX column in a STUDENT table can have only two possible values, "M" or "F"; therefore this column is said to have low sparsity. In contrast, the STU_DOB column that stores the student date of birth can have many different date values; therefore, this column is said to have high sparsity. Knowing the sparsity helps you decide whether or not the use of an index is appropriate. For example, when you perform a search in a column with low sparsity, you are very likely to read a high percentage of the table rows anyway; therefore index processing may be unnecessary work.

10. What is the difference between a rule-based optimizer and a cost-based optimizer?

A rule-based optimizer uses a set of preset rules and points to determine the best approach to execute a query. The rules assign a "cost" to each SQL operation; the costs are then added to yield the cost of the execution plan.

A cost-based optimizer uses sophisticated algorithms based on the statistics about the objects being accessed to determine the best approach to execute a query. In this case, the optimizer process adds up the processing cost, the I/O costs and the resource costs (RAM and temporary space) to come up with the total cost of a given execution plan.

11. What are optimizer hints and how are they used?

Hints are special instructions for the optimizer that are embedded inside the SQL command text. Although the optimizer generally performs very well under most circumstances, there are some circumstances in which the optimizer may not choose the best execution plan. Remember, the optimizer makes decisions based on the existing statistics. If the statistics are old, the optimizer may not do a good job in selecting the best execution plan. Even with the current statistics, the optimizer choice may not be the most efficient one. There are some occasions when the end-user would like to change the optimizer mode for the current SQL statement. In order to accomplish this task, you have to use hints.

12. What are some general guidelines for creating and using indexes?

Create indexes for each single attribute used in a WHERE, HAVING, ORDER BY, or GROUP BY clause. If you create indexes in all single attributes *used in search conditions*, the DBMS will access the table using an index scan, instead of a full table scan. For example, if you have an index for P_PRICE, the condition P_PRICE > 10.00 can be solved by accessing the index, instead of sequentially scanning all table rows and evaluating P_PRICE for each row. Indexes are also used in join expressions, such as in CUSTOMER.CUS_CODE = INVOICE.CUS_CODE.

Do not use indexes in small tables or tables with low sparsity. Remember, small tables and low sparsity tables are not the same thing. A search condition in a table with low sparsity may return a high percentage of table rows anyway, making the index operation too costly and making the full table scan a viable option. Using the same logic, do not create indexes for tables with few rows and few attributes—*unless you must ensure the existence of unique values in a column*.

Declare primary and foreign keys so the optimizer can use the indexes in join operations. All natural joins and old-style joins will benefit if you declare primary keys and foreign keys because the optimizer will use the available indexes at join time. (The declaration of a PK or FK will automatically create an index for the declared column. Also, for the same reason, it is better to write joins using the SQL JOIN syntax. (See Chapter 8, "Advanced SQL.")

Declare indexes in join columns other than PK/FK. If you do join operations on columns other than the primary and foreign key, you may be better off declaring indexes in such columns.

13. Most query optimization techniques are designed to make the optimizer's work easier. What factors should you keep in mind if you intend to write conditional expressions in SQL code?

Use simple columns or literals as operands in a conditional expression—avoid the use of conditional expressions with functions whenever possible. Comparing the contents of a single column to a literal is faster than comparing to expressions.

Numeric field comparisons are faster than character, date, and NULL comparisons. In search conditions, comparing a numeric attribute to a numeric literal is faster than comparing a character attribute to a character literal. In general, numeric comparisons (integer, decimal) are handled faster by the CPU than character and date comparisons. Because indexes do not store references to null values, NULL conditions involve additional processing and therefore tend to be the slowest of all conditional operands.

Equality comparisons are faster than inequality comparisons. As a general rule, equality comparisons are processed faster than inequality comparisons. For example, P_PRICE = 10.00 is processed faster because the DBMS can do a direct search using the index in the column. If there are no exact matches, the condition is evaluated as false. However, if you use an inequality symbol (>, >=, <, <=) the DBMS must perform additional processing to complete the request. This is because there would almost always be more "greater than" or "less than" values and perhaps only a few exactly "equal" values in the index. The slowest (with the exception of NULL) of all comparison operators is LIKE with wildcard symbols, such as in V_CONTACT LIKE "%glo%". Also, using the "not equal" symbol (<>) yields slower searches, especially if the sparsity of the data is high; that is, if there are many more different values than there are equal values.

Whenever possible, transform conditional expressions to use literals. For example, if your condition is P_PRICE -10 = 7, change it to read P_PRICE = 17. Also, if you have a composite condition such as:
P_QOH < P_MIN AND P_MIN = P_REORDER AND P_QOH = 10
change it to read:
P_QOH = 10 AND P_MIN = P_REORDER AND P_MIN > 10

When using multiple conditional expressions, write the equality conditions first. (Note that we did this in the previous example.) Remember, equality conditions are faster to process than inequality conditions. Although most RDBMSs will automatically do this for you, paying attention to this detail lightens the load for the query optimizer. (The optimizer won't have to do what you have already done.)

If you use multiple AND conditions, write the condition most likely to be false first. If you use this technique, the DBMS will stop evaluating the rest of the conditions as soon as it finds a conditional expression that is evaluated to be false. Remember, for multiple AND conditions to be found true, all conditions must be evaluated as true. If one of the conditions evaluates to false, everything else is evaluated as false. Therefore, if you use this technique, the DBMS won't waste time unnecessarily evaluating additional conditions. Naturally, the use of this technique implies an implicit knowledge of the sparsity of the data set.

Whenever possible, try to avoid the use of the NOT logical operator. It is best to transform a SQL expression containing a NOT logical operator into an equivalent expression. For example:
NOT (P_PRICE > 10.00) can be written as P_PRICE <= 10.00.
Also, NOT (EMP_SEX = 'M') can be written as EMP_SEX = 'F'.

14. What recommendations would you make for managing the data files in a DBMS with many tables and indexes?

First, create independent data files for the system, indexes and user data table spaces. Put the data files on separate disks or RAID volumes. This ensures that index operations will not conflict with end-user data or data dictionary table access operations.

Second, put high-usage end-user tables in their own table spaces. By doing this, the database minimizes conflicts with other tables and maximizes storage utilization.

Third, evaluate the creation of indexes based on the access patterns. Identify common search criteria and isolate the most frequently used columns in search conditions. Create indexes on high usage columns with high sparsity.

Fourth, evaluate the usage of aggregate queries in your database. Identify columns used in aggregate functions and determine if the creation of indexes on such columns will improve response time.

Finally, identify columns used in ORDER BY statements and make sure there are indexes on such columns.

15. What does RAID stand for, and what are some commonly used RAID levels?

RAID is the acronym for **R**edundant **A**rray of **I**ndependent **D**isks. RAID is used to provide balance between performance and fault tolerance. RAID systems use multiple disks to create virtual disks (storage volumes) formed by several individual disks. RAID systems provide performance improvement and fault tolerance. Table 11.7 in the text shows the commonly used RAID levels. (We have reproduced the table for your convenience.)

TABLE 11.7 Common RAID Configurations

RAID Level	Description
0	The data blocks are spread over separate drives. Also known as striped array. Provides increased performance but no fault tolerance. Fault tolerance means that in case of failure, data could be reconstructed and retrieved. Requires a minimum of two drives.
1	The same data blocks are written (duplicated) to separate drives. Also referred to as mirroring or duplexing. Provides increased read performance and fault tolerance via data redundancy. Requires a minimum of two drives.
3	The data are striped across separate drives, and parity data are computed and stored in a dedicated drive. Parity data are specially generated data that permit the reconstruction of corrupted or missing data. Provides good read performance and fault tolerance via parity data. Requires a minimum of three drives.
5	The data and the parity are striped across separate drives. Provides good read performance and fault tolerance via parity data. Requires a minimum of three drives.