

11

DATABASE PERFORMANCE TUNING AND
QUERY OPTIMIZATION

ELEVEN

In this chapter, you will learn:

- Basic database performance-tuning concepts
- How a DBMS processes SQL queries
- About the importance of indexes in query processing
- About the types of decisions the query optimizer has to make
- Some common practices used to write efficient SQL code
- How to formulate queries and tune the DBMS for optimal performance

Database performance tuning is a critical topic, yet it usually receives minimal coverage in the database curriculum. Most databases used in classrooms have only a few records per table. As a result, the focus often is on making SQL queries perform an intended task, without considering the efficiency of the query process. In fact, even the most efficient query environment yields no visible performance improvements over the least efficient query environment when only 20 or 30 table rows (records) are queried. Unfortunately, that lack of attention to query efficiency can yield unacceptably slow results, when in the real world, queries are executed over tens of millions of records. In this chapter, you learn what it takes to create a more efficient query environment.

A circular graphic with a yellow background and a black border. The word "Preview" is written in a serif font, with the "P" being significantly larger and positioned to the left of the "review". The background of the page features a purple-tinted image of a computer keyboard and a mouse, with a grid pattern overlaid.

Preview

NOTE

Because this book focuses on databases, this chapter covers only those factors directly affecting *database* performance. Also, because performance-tuning techniques can be DBMS-specific, the material in this chapter might 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.)

11.1 DATABASE PERFORMANCE-TUNING CONCEPTS

One of the main functions of a database system is to provide timely answers to end users. End users interact with the DBMS through the use of queries to generate information, using the following sequence:

1. The end-user (client-end) application generates a query.
2. The query is sent to the DBMS (server end).
3. The DBMS (server end) executes the query.
4. The DBMS sends the resulting data set to the end-user (client-end) application.

End users expect their queries to return results as quickly as possible. How do you know that the performance of a database is good? Good database performance is hard to evaluate. How do you know if a 1.06-second query response time is good enough? It's easier to identify bad database performance than good database performance—all it takes is end-user complaints about slow query results. Unfortunately, the same query might perform well one day and not so well two months later. Regardless of end-user perceptions, *the goal of database performance is to execute queries as fast as possible*. Therefore, database performance must be closely monitored and regularly tuned. **Database performance tuning** refers to a set of activities and procedures designed to reduce the response time of the database system—that is, to ensure that an end-user query is processed by the DBMS in the minimum amount of time.

The time required by a query to return a *result set* depends on many factors. Those factors tend to be wide-ranging and to vary from environment to environment and from vendor to vendor. The performance of a typical DBMS is constrained by three main factors: CPU processing power, available primary memory (RAM), and input/output (hard disk and network) throughput. Table 11.1 lists some system components and summarizes general guidelines for achieving better query performance.

TABLE 11.1 General Guidelines for Better System Performance

	SYSTEM RESOURCES	CLIENT	SERVER
Hardware	CPU	The fastest possible Dual-core CPU or higher	The fastest possible Multiple processors (Quad-core technology)
	RAM	The maximum possible	The maximum possible
	Hard Disk	Fast SATA/EIDE hard disk with sufficient free hard disk space	Multiple high-speed, high-capacity hard disks (SCSI / SATA / Firewire / Fibre Channel) in RAID configuration
	Network	High-speed connection	High-speed connection
Software	Operating System	Fine-tuned for best client application performance	Fine-tuned for best server application performance
	Network	Fine-tuned for best throughput	Fine-tuned for best throughput
	Application	Optimize SQL in client application	Optimize DBMS server for best performance

Naturally, the system will perform best when its hardware and software resources are optimized. However, in the real world, unlimited resources are not the norm; internal and external constraints always exist. Therefore, the system components should be optimized to obtain the best throughput possible with existing (and often limited) resources, which is why database performance tuning is important.

Fine-tuning the performance of a system requires a holistic approach. That is, *all* factors must be checked to ensure that each one operates at its optimum level and has sufficient resources to minimize the occurrence of bottlenecks. Because database design is such an important factor in determining the database system's performance efficiency, it is worth repeating this book's mantra:

Good database performance starts with good database design. *No amount of fine tuning will make a poorly designed database perform as well as a well-designed database.* This is particularly true in the case of redesigning existing databases, where the end user expects unrealistic performance gains from older databases.

What constitutes a good, efficient database design? From the performance tuning point of view, the database designer must ensure that the design makes use of the database features available in the DBMS that guarantee the integrity and optimal performance of the database. This chapter provides you with fundamental knowledge that will help you to optimize database performance by selecting the appropriate database server configuration, utilizing indexes, understanding table storage organization and data locations, and implementing the most efficient SQL query syntax.

11.1.1 PERFORMANCE TUNING: CLIENT AND SERVER

In general, database performance-tuning activities can be divided into those taking place on the client side and those taking place on the server side.

- On the client side, the objective is to generate a SQL query that returns the correct answer in the least amount of time, using the minimum amount of resources at the server end. The activities required to achieve that goal are commonly referred to as **SQL performance tuning**.
- On the server side, the DBMS environment must be properly configured to respond to clients' requests in the fastest way possible, while making optimum use of existing resources. The activities required to achieve that goal are commonly referred to as **DBMS performance tuning**.



ONLINE CONTENT

If you want to learn more about clients and servers, check **Appendix F, Client/Server Systems**, located in the Student Online Companion for this book.

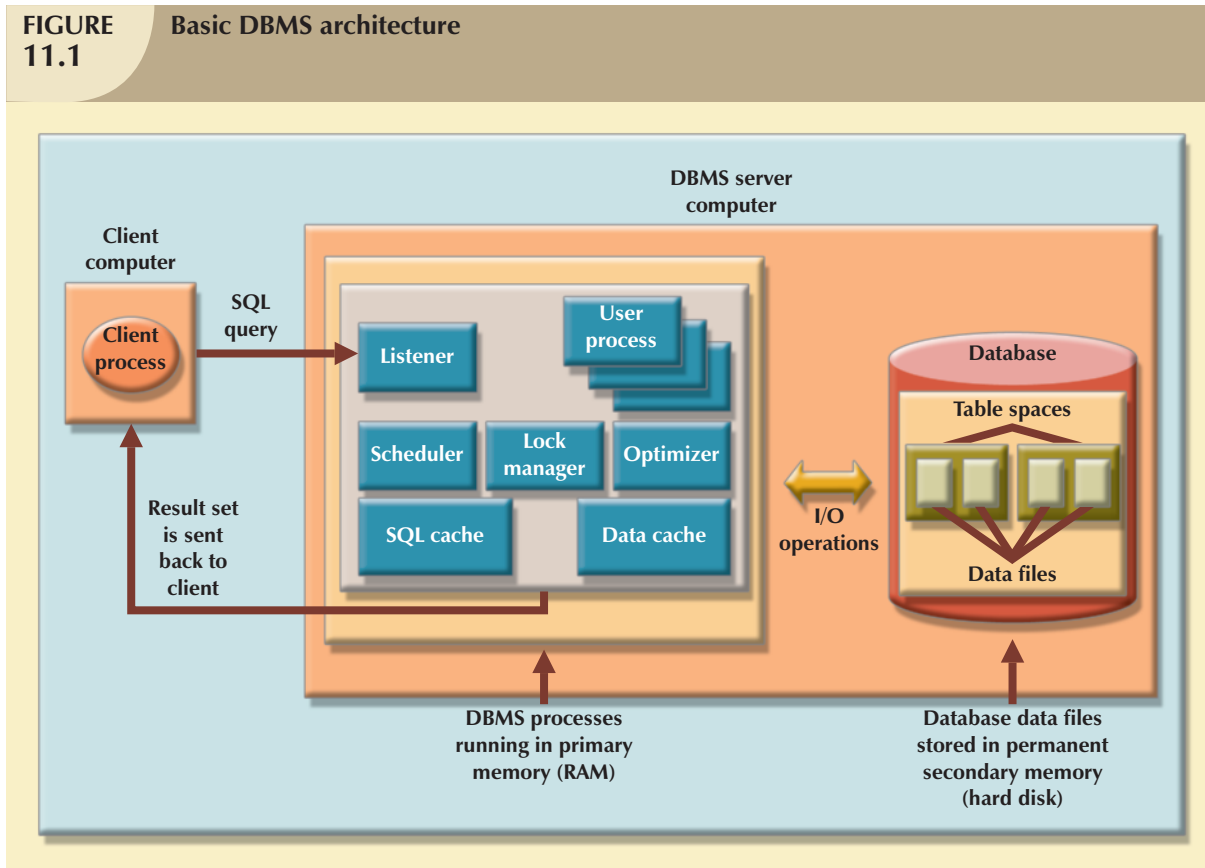
Keep in mind that DBMS implementations are typically more complex than just a two-tier client/server configuration. However, even in multi-tier (client front-end, application middleware, and database server back-end) client/server environments, performance-tuning activities are frequently divided into subtasks to ensure the fastest possible response time between any two component points.

This chapter covers SQL performance-tuning practices on the client side and DBMS performance-tuning practices on the server side. But before you can start learning about the tuning processes, you must first learn more about the DBMS architectural components and processes and how those processes interact to respond to end-users requests.

11.1.2 DBMS ARCHITECTURE

The architecture of a DBMS is represented by the processes and structures (in memory and in permanent storage) used to manage a database. Such processes collaborate with one another to perform specific functions. Figure 11.1 illustrates the basic DBMS architecture.

FIGURE 11.1 Basic DBMS architecture



Note the following components and functions in Figure 11.1:

- All data in a database are stored in **data files**. A typical enterprise database is normally composed of several data files. A data file can contain rows from one single table, or it can contain rows from many different tables. A database administrator (DBA) determines the initial size of the data files that make up the database; however, as required, the data files can automatically expand in predefined increments known as **extends**. For example, if more space is required, the DBA can define that each new extend will be in 10 KB or 10 MB increments.
- Data files are generally grouped in file groups or table spaces. A **table space** or **file group** is a logical grouping of several data files that store data with similar characteristics. For example, you might have a *system* table space where the data dictionary table data are stored; a *user data* table space to store the user-created tables; an *index* table space to hold all indexes; and a *temporary* table space to do temporary sorts, grouping, and so on. Each time you create a new database, the DBMS automatically creates a minimum set of table spaces.
- The **data cache** or **buffer cache** is a shared, reserved memory area that stores the most recently accessed data blocks in RAM. The data cache is where the data read from the database data files are stored after the data have been read or before the data are written to the database data files. The data cache also caches system catalog data and the contents of the indexes.
- The **SQL cache** or **procedure cache** is a shared, reserved memory area that stores the most recently executed SQL statements or PL/SQL procedures, including triggers and functions. (To learn more about PL/SQL procedures, triggers, and SQL functions, study Chapter 8, *Advanced SQL*.) The SQL cache does not store the end-user written SQL. Rather, the SQL cache stores a “processed” version of the SQL that is ready for execution by the DBMS.
- To work with the data, the DBMS must retrieve the data from permanent storage (data files in which the data are stored) and place it in RAM (data cache).

- To move data from the permanent storage (data files) to the RAM (data cache), the DBMS issues I/O requests and waits for the replies. An **input/output (I/O) request** is a low-level (read or write) data access operation to and from computer devices, such as memory, hard disks, video, and printers. The purpose of the I/O operation is to move data to and from various computer components and devices. Note that an I/O disk read operation retrieves an entire physical disk block, generally containing multiple rows, from permanent storage to the data cache, even if you will be using only one attribute from only one row. The physical disk block size depends on the operating system and could be 4K, 8K, 16K, 32K, 64K, or even larger. Furthermore, depending on the circumstances, a DBMS might issue a single-block read request or a multiblock read request.
- Working with data in the data cache is many times faster than working with data in the data files because the DBMS doesn't have to wait for the hard disk to retrieve the data. This is because no hard disk I/O operations are needed to work within the data cache.
- The majority of performance-tuning activities focus on minimizing the number of I/O operations because using I/O operations is many times slower than reading data from the data cache. For example, as of this writing, RAM access times range from 5 to 70 ns (nanoseconds), while hard disk access times range from 5 to 15 ms (milliseconds). This means that hard disks are about six orders of magnitude (a million times) slower than RAM.¹

Also illustrated in Figure 11.1 are some typical DBMS processes. Although the number of processes and their names vary from vendor to vendor, the functionality is similar. The following processes are represented in Figure 11.1:

- *Listener*. The listener process listens for clients' requests and handles the processing of the SQL requests to other DBMS processes. Once a request is received, the listener passes the request to the appropriate user process.
- *User*. The DBMS creates a user process to manage each client session. Therefore, when you log on to the DBMS, you are assigned a user process. This process handles all requests you submit to the server. There are many user processes—at least one per each logged-in client.
- *Scheduler*. The scheduler process organizes the concurrent execution of SQL requests. (See Chapter 10, Transaction Management and Concurrency Control.)
- *Lock manager*. This process manages all locks placed on database objects, including disk pages. (See Chapter 10.)
- *Optimizer*. The optimizer process analyzes SQL queries and finds the most efficient way to access the data. You will learn more about this process later in the chapter.

11.1.3 DATABASE STATISTICS

Another DBMS process that plays an important role in query optimization is gathering database statistics. The term **database statistics** refers to a number of measurements about database objects, such as number of processors used, processor speed, and temporary space available. Such statistics give a snapshot of database characteristics.

As you will learn later in this chapter, the DBMS uses these statistics to make critical decisions about improving query processing efficiency. Database statistics can be gathered manually by the DBA or automatically by the DBMS. For example, many DBMS vendors support the ANALYZE command in SQL 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. A sample of measurements that the DBMS may gather about various database objects is shown in Table 11.2.

¹Low Latency, Eliminating Application Jitters with Solaris, White Paper, May 2007, Sun Microsystems, http://www.sun.com/solutions/documents/white-papers/fn_lowlatency_solaris.pdf.

TABLE 11.2 Sample Database Statistics Measurements

DATABASE OBJECT	SAMPLE MEASUREMENTS
Tables	Number of rows, number of disk blocks used, row length, number of columns in each row, number of distinct values in each column, maximum value in each column, minimum value in each column, and columns that have indexes
Indexes	Number and name of columns in the index key, number of key values in the index, number of distinct key values in the index key, histogram of key values in an index, and number of disk pages used by the index
Environment Resources	Logical and physical disk block size, location and size of data files, and number of extends per data file

If the object statistics exist, the DBMS will use them in query processing. Although some of the newer DBMSs (such as Oracle, SQL Server, and DB2) automatically gather statistics, others require the DBA to gather statistics manually. To generate the database object statistics manually, you could use the following syntax:

```
ANALYZE <TABLE/INDEX> object_name COMPUTE STATISTICS;
```

(In SQL Server, use UPDATE STATISTICS <object_name>, where the object_name refers to a table or a view.)

For example, to generate statistics for the VENDOR table, you would use the following command:

```
ANALYZE TABLE VENDOR COMPUTE STATISTICS;
```

(In SQL Server, use UPDATE STATISTICS VENDOR;.)

When you generate statistics for a table, all related indexes are also analyzed. However, you could generate statistics for a single index by using the following command:

```
ANALYZE INDEX VEND_NDX COMPUTE STATISTICS;
```

In the above example, VEND_NDX is the name of the index.

(In SQL Server, use UPDATE STATISTICS <table_name> <index_name>. For example: UPDATE STATISTICS VENDOR VEND_NDX;.)

Database statistics are stored in the system catalog in specially designated tables. It is common to periodically regenerate the statistics for database objects, especially those database objects that are subject to frequent change. For example, if you are the owner of a video store and you have a video rental DBMS, your system will likely use a RENTAL table to store the daily video rentals. That RENTAL table (and its associated indexes) would be subject to constant inserts and updates as you record your daily rentals and returns. Therefore, the RENTAL table statistics you generated last week do not depict an accurate picture of the table as it exists today. The more current the statistics, the better the chances are for the DBMS to properly select the fastest way to execute a given query.

Now that you know the basic architecture of DBMS processes and memory structures, and the importance and timing of the database statistics gathered by the DBMS, you are ready to learn how the DBMS processes a SQL query request.

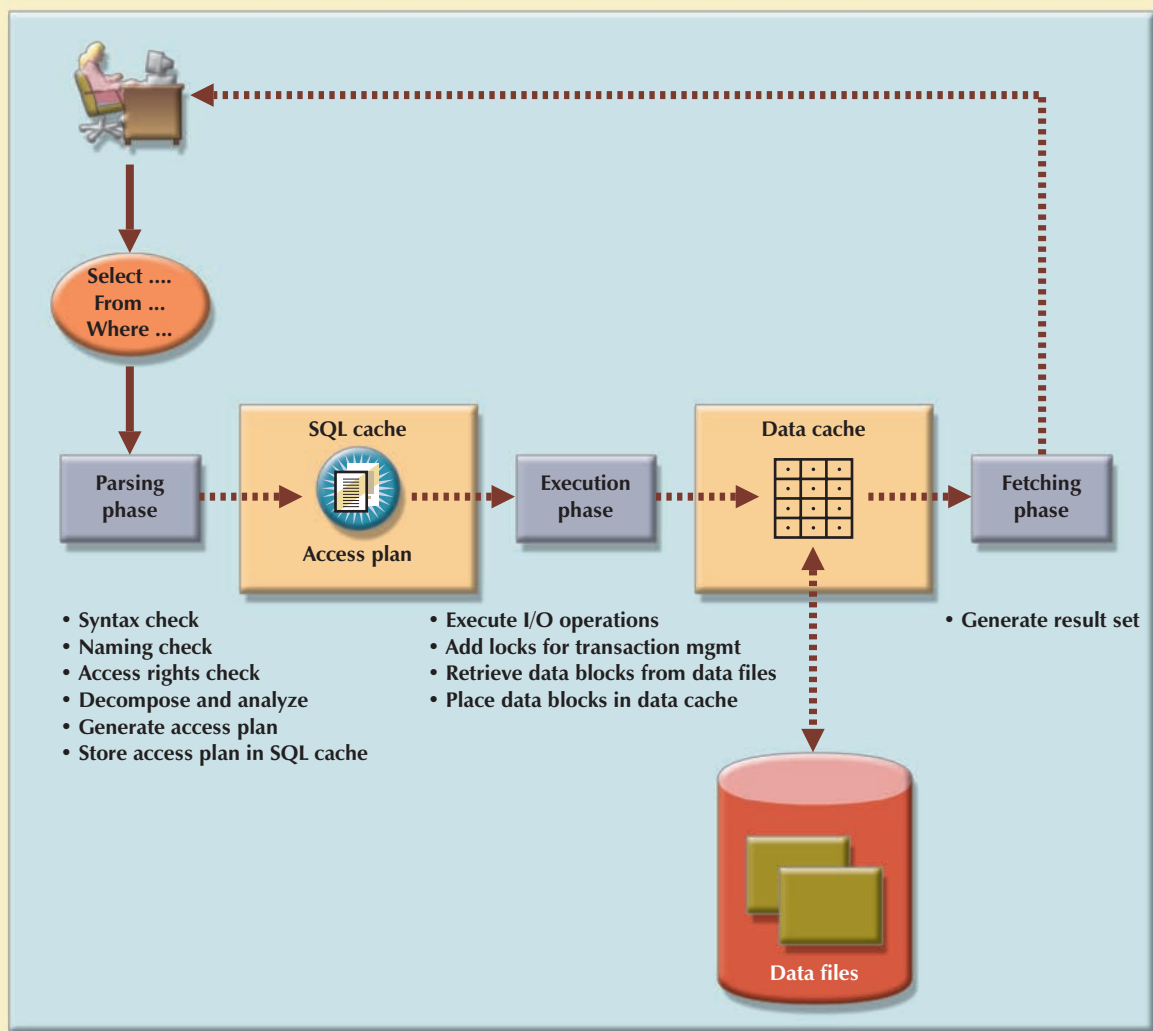
11.2 QUERY PROCESSING

What happens at the DBMS server end when the client's SQL statement is received? In simple terms, the DBMS processes a query in three phases:

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.

The processing of SQL DDL statements (such as CREATE TABLE) is different from the processing required by DML statements. The difference is that 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. Figure 11.2 shows the general steps required for query processing. Each of the steps will be discussed in the following sections.

FIGURE 11.2 Query processing



11.2.1 SQL PARSING PHASE

The optimization process includes breaking down—parsing—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 it *almost* always executes faster because, as explained earlier, many factors affect the performance of a database. Those factors include the network, the client computer's resources, and other queries running concurrently in the same database.) To determine the most efficient way to execute the query, the DBMS may use the database statistics you learned about earlier.

The SQL parsing activities are performed by the **query optimizer**, which analyzes the SQL query and finds the most efficient way to access the data. This process is the most time-consuming phase in query processing. Parsing a SQL query requires several steps, in which the SQL query is:

- Validated for syntax compliance.
- Validated against the data dictionary to ensure that tables and column names are correct.
- Validated against the data dictionary to ensure that the user has proper access rights.
- Analyzed and decomposed into more atomic components.
- Optimized through transformation into a fully equivalent but more efficient SQL query.
- Prepared for execution by determining the most efficient execution or access plan.

Once the SQL statement is transformed, the DBMS creates what is commonly known as an access or execution plan. An **access plan** is the result of parsing an SQL statement; it contains the series of steps a DBMS will use to execute the query and to return the result set in the most efficient way. First, the DBMS checks to see if an access plan already exists for the query in the SQL cache. If it does, the DBMS reuses the access plan to save time. If it doesn't, the optimizer evaluates various plans and makes decisions about what indexes to use and how to best perform join operations. The chosen access plan for the query is then placed in the SQL cache and made available for use and future reuse.

Access plans are DBMS-specific and translate the client's SQL query into the series of complex I/O operations required to read the data from the physical data files and generate the result set. Access plans are DBMS-specific; some commonly found I/O operations are illustrated in Table 11.3.

TABLE 11.3 Sample DBMS Access Plan I/O Operations

OPERATION	DESCRIPTION
Table Scan (Full)	Reads the entire table sequentially, from the first row to the last row, one row at a time (slowest)
Table Access (Row ID)	Reads a table row directly, using the row ID value (fastest)
Index Scan (Range)	Reads the index first to obtain the row IDs and then accesses the table rows directly (faster than a full table scan)
Index Access (Unique)	Used when a table has a unique index in a column
Nested Loop	Reads and compares a set of values to another set of values, using a nested loop style (slow)
Merge	Merges two data sets (slow)
Sort	Sorts a data set (slow)

Table 11.3 shows just a few database access I/O operations. (This illustration is based on an Oracle RDBMS.) However, Table 11.3 does illustrate the type of I/O operations that most DBMSs perform when accessing and manipulating data sets.

In Table 11.3, note that a table access using a row ID is the fastest method. A row ID is a unique identification for every row saved in permanent storage; it can be used to access the row directly. Conceptually, a row ID is similar to a parking

slip you get when you park your car in an airport parking lot. The parking slip contains the section number and lot number. Using that information, you can go directly to your car without having to go through every section and lot.

11.2.2 SQL EXECUTION PHASE

In this phase, all I/O operations indicated in the access plan are executed. When the execution plan is run, the proper locks—if needed—are acquired for the data to be accessed, and the data are retrieved from the data files and placed in the DBMSs data cache. All transaction management commands are processed during the parsing and execution phases of query processing.

11.2.3 SQL FETCHING PHASE

After the parsing and execution phases are completed, all rows that match the specified condition(s) are retrieved, sorted, grouped, and/or aggregated (if required). During the fetching phase, the rows of the resulting query result set are returned to the client. The DBMS might use temporary table space to store temporary data. In this stage, the database server coordinates the movement of the result set rows from the server cache to the client cache. For example, a given query result set might contain 9,000 rows; the server would send the first 100 rows to the client and then wait for the client to request the next set of rows, until the entire result set is sent to the client.

11.2.4 QUERY PROCESSING BOTTLENECKS

The main objective of query processing is to execute a given query in the fastest way possible with the least amount of resources. As you have seen, the execution of a query requires the DBMS to break down the query into a series of interdependent I/O operations to be executed in a collaborative manner. The more complex a query is, the more complex the operations are, and the more likely it is that there will be bottlenecks. A **query processing bottleneck** is a delay introduced in the processing of an I/O operation that causes the overall system to slow down. In the same way, the more components a system has, the more interfacing among the components is required, and the more likely it is that there will be bottlenecks. Within a DBMS, there are five components that typically cause bottlenecks:

- **CPU.** The CPU processing power of the DBMS should match the system's expected work load. A high CPU utilization might indicate that the processor speed is too slow for the amount of work performed. However, heavy CPU utilization can be caused by other factors, such as a defective component, not enough RAM (the CPU spends too much time swapping memory blocks), a badly written device driver, or a rogue process. A CPU bottleneck will affect not only the DBMS but all processes running in the system.
- **RAM.** The DBMS allocates memory for specific usage, such as data cache and SQL cache. RAM must be shared among all running processes (operating system, DBMS, and all other running processes). If there is not enough RAM available, moving data among components that are competing for scarce RAM can create a bottleneck.
- **Hard disk.** Another common cause of bottlenecks is hard disk speed and data transfer rates. Current hard disk storage technology allows for greater storage capacity than in the past; however, hard disk space is used for more than just storing end user data. Current operating systems also use the hard disk for *virtual memory*, which refers to copying areas of RAM to the hard disk as needed to make room in RAM for more urgent tasks. Therefore, the greater the hard disk storage space is and the faster the data transfer rates are, the lesser likelihood of bottlenecks.
- **Network.** In a database environment, the database server and the clients are connected via a network. All networks have a limited amount of bandwidth that is shared among all clients. When many network nodes access the network at the same time, bottlenecks are likely.
- **Application code.** Not all bottlenecks are caused by limited hardware resources. One of the most common sources of bottlenecks is badly written application code. No amount of coding will make a poorly designed database perform better. We should also add: you can throw unlimited resources at a badly written application and it will still perform as a badly written application!

Learning how to avoid these bottlenecks and thus optimize database performance is the main focus of this chapter.

11.3 INDEXES AND QUERY OPTIMIZATION

Indexes are crucial in speeding up data access because they facilitate searching, sorting, and using aggregate functions and even join operations. The improvement in data access speed occurs because an index is an ordered set of values that contains the index key and pointers. The pointers are the row IDs for the actual table rows. Conceptually, a data index is similar to a book index. When you use a book index, you look up the word, similar to the index key, which is accompanied by the page number(s), similar to the pointer(s), which direct you to the appropriate page(s).

An index scan is more efficient than a full table scan because the index data are preordered and the amount of data is usually much smaller. Therefore, when performing searches, it is almost always better for the DBMS to use the index to access a table than to scan all rows in a table sequentially. For example, Figure 11.3 shows the index representation of a CUSTOMER table with 14,786 rows and the index STATE_NDX on the CUS_STATE attribute.

FIGURE 11.3 Index representation for the CUSTOMER table

STATE_NDX INDEX		CUSTOMER TABLE (14,786 rows)								
Key	Row	Row ID	CUS_CODE	CUS_LNAME	CUS_FNAME	CUS_INITIAL	CUS_AREACODE	CUS_PHONE	CUS_STATE	CUS_BALANCE
AZ	2	1	10010	Ramas	Alfred	A	615	844-2573	FL	\$0.00
....	2	10011	Dunne	Leona	K	713	894-1238	AZ	\$0.00
....	3	10012	Smith	Kathy	W	615	894-2285	TX	\$345.86
....	4	10013	Olowski	Paul	F	615	894-2180	AZ	\$536.75
FL	1	5	10014	Orlando	Myron		615	222-1672	NY	\$0.00
FL	7	6	10015	O'Brian	Amy	B	713	442-3381	NY	\$0.00
FL	8	7	10016	Brown	James	G	615	297-1228	FL	\$221.19
FL	13245	8	10017	Williams	George		615	290-2556	FL	\$768.93
FL	14786	9	10018	Farriss	Anne	G	713	382-7185	TX	\$216.55
....	10	10019	Smith	Olette	K	615	297-3809	AZ	\$0.00
....
....
....	13245	23120	Veron	George	D	415	231-9872	FL	\$675.00
....
....
....	14786	24560	Suarez	Victor		435	342-9876	FL	\$342.00

Suppose you submit the following query:

```
SELECT    CUS_NAME, CUS_STATE
FROM      CUSTOMER
WHERE     CUS_STATE = 'FL';
```

If there is no index, the DBMS will perform a full table scan, thus reading all 14,786 customer rows. Assuming that the index STATE_NDX is created (and ANALYZED), the DBMS will automatically use the index to locate the first customer with a state equal to 'FL' and then proceed to read all subsequent CUSTOMER rows, using the row IDs in the index as a guide. Assuming that only five rows meet the condition CUS_STATE = 'FL' then, there are 5 accesses to the index and 5 accesses to the data, for a total of 10 I/O accesses. The DBMS would save approximately 14,776 I/O requests for customer rows that do not meet the criteria. That's a lot of CPU cycles!

If indexes are so important, why not index every column in every table? It's not practical to do so. Indexing every column in every table taxes the DBMS too much in terms of index-maintenance processing, especially if the table has many attributes; has many rows; and/or requires many inserts, updates, and/or deletes.

One measure that determines 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 that 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, that column is said to have high sparsity. Knowing the sparsity helps you decide whether the use of an index is appropriate. For example, when you perform a search in a column with low sparsity, you are likely to read a high percentage of the table rows anyway; therefore, index processing might be unnecessary work. In Section 11.5, you learn how to determine when an index is recommended.

Most DBMSs implement indexes using one of the following data structures:

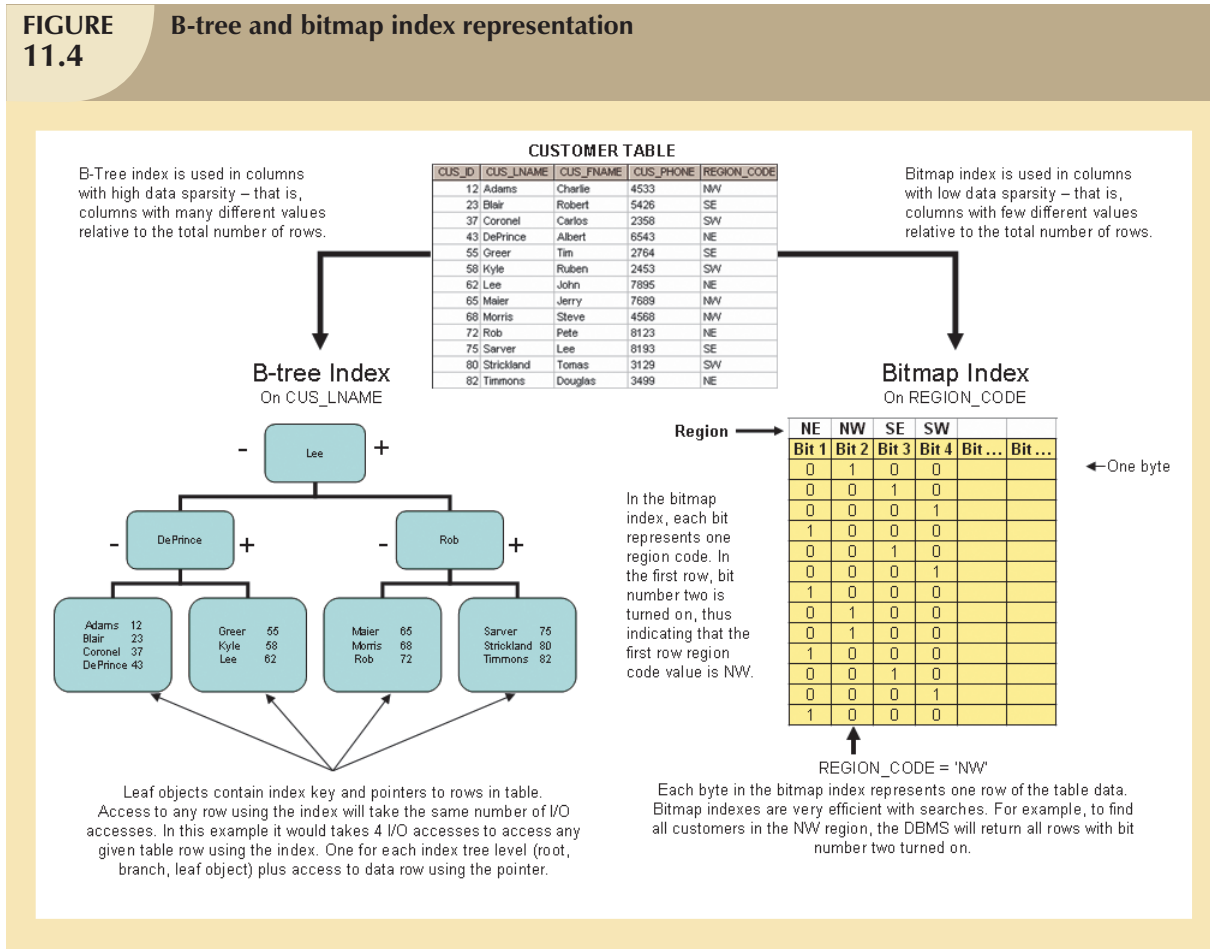
- *Hash indexes.* A hash algorithm is used to create a hash value from a key column. This value points to an entry in a hash table which in turn points to the actual location of the data row. This type of index is good for simple and fast lookup operations.
- *B-tree indexes.* This is the default and most common type of index used in databases. The B-tree index is used mainly in tables in which column values repeat a relative smaller number of times. The B-tree index is an ordered data structure organized as an upside down tree. The index tree is stored separate from the data. The lower-level leaves of the B-tree index contain the pointers to the actual data rows. B-tree indexes are “self-balanced,” which means that it takes the same amount of access to find any given row in the index.
- *Bitmap indexes.* Used in data warehouse applications in tables with large number of rows in which a small number of column values repeat many times. Bitmap indexes tend to use less space than B-tree indexes because they use bits (instead of bytes) to store their data.

Using the above index characteristics, a database designer can determine the best type of index to use. For example, assume a CUSTOMER table with several thousand rows. The CUSTOMER table has two columns that are used extensively for query purposes: CUS_LNAME that represents a customer last name and REGION_CODE that could have one of four values (NE, NW, SW, and, SE). Based on this information, you could conclude that:

- Because the CUS_LNAME column contains many different values that repeat a relatively small number of times (compared to the total number of rows in the table), a B-tree index will be used.
- Because the REGION_CODE column contains fewer different values that repeat a relatively large number of times (compared to the total number of rows in the table), a bitmap index will be used. Figure 11.4 shows the B-tree and bitmap representations for a CUSTOMER table used in the previous discussion.

Current generation DBMSs are intelligent enough to determine the best type of index to use under certain circumstances (provided the DBMS has updated database statistics). Whatever the index chosen, the DBMS determines the best plan to execute a given query. The next section guides you through a simplified example of the type of choices that the query optimizer must perform.

FIGURE 11.4 B-tree and bitmap index representation



11.4 OPTIMIZER CHOICES

Query optimization is the central activity during the parsing phase in query processing. In this phase, the DBMS must choose what indexes to use, how to perform join operations, what table to use first, and so on. Each DBMS has its own algorithms for determining the most efficient way to access the data. The query optimizer can operate in one of two modes:

- A **rule-based optimizer** uses preset rules and points to determine the best approach to execute a query. The rules assign a “fixed cost” to each SQL operation; the costs are then added to yield the cost of the execution plan. For example, a full table scan has a set cost of 10, while a table access by row ID has a set cost of 3.
- 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.

The optimizer objective is to find alternative ways to execute a query—to evaluate the “cost” of each alternative and then to choose the one with the lowest cost. To understand the function of the query optimizer, let’s use a simple example. Assume that you want to list all products provided by a vendor based in Florida. To acquire that information, you could write the following query:

```
SELECT    P_CODE, P_DESCRIPT, P_PRICE, V_NAME, V_STATE
FROM      PRODUCT, VENDOR
WHERE     PRODUCT.V_CODE = VENDOR.V_CODE
          AND VENDOR.V_STATE = 'FL';
```

Furthermore, let’s assume that the database statistics indicate that:

- The PRODUCT table has 7,000 rows.
- The VENDOR table has 300 rows.
- Ten vendors are located in Florida.
- One thousand products come from vendors in Florida.

It’s important to point out that only the first two items are available to the optimizer. The second two items are assumed to illustrate the choices that the optimizer must make. Armed with the information in the first two items, the optimizer would try to find the most efficient way to access the data. The primary factor in determining the most efficient access plan is the I/O cost. (Remember, the DBMS always tries to minimize I/O operations.) Table 11.4 shows two sample access plans for the previous query and their respective I/O costs.

TABLE 11.4 Comparing Access Plans and I/O Costs

PLAN	STEP	OPERATION	I/O OPERATIONS	I/O COST	RESULTING SET ROWS	TOTAL I/O COST
A	A1	Cartesian product (PRODUCT, VENDOR)	7,000 + 300	7,300	2,100,000	7,300
	A2	Select rows in A1 with matching vendor codes	2,100,000	2,100,000	7,000	2,107,300
	A3	Select rows in A2 with V_STATE = 'FL'	7,000	7,000	1,000	2,114,300
B	B1	Select rows in VENDOR with V_STATE = 'FL'	300	300	10	300
	B2	Cartesian Product (PRODUCT, B1)	7,000 + 10	7,010	70,000	7,310
	B3	Select rows in B2 with matching vendor codes	70,000	70,000	1,000	77,310

To make the example easier to understand, the I/O Operations and I/O Cost columns in Table 11.4 estimate only the number of I/O disk reads the DBMS must perform. For simplicity’s sake, it is assumed that there are no indexes and that each row read has an I/O cost of 1. For example, in step A1, the DBMS must perform a Cartesian product of PRODUCT and VENDOR. To do that, the DBMS must read all rows from PRODUCT (7,000) and all rows from VENDOR (300), yielding a total of 7,300 I/O operations. The same computation is done in all steps. In Table 11.4, you can see how plan A has a total I/O cost that is almost 30 times higher than plan B. In this case, the optimizer will choose plan B to execute the SQL.

NOTE

Not all DBMSs optimize SQL queries the same way. As a matter of fact, Oracle parses queries differently than what is described in several sections in this chapter. Always read the documentation to examine the optimization requirements for your DBMS implementation.

Given the right conditions, some queries could be answered entirely by using only an index. For example, assume the `PRODUCT` table and the index `P_QOH_NDX` in the `P_QOH` attribute. Then a query such as `SELECT MIN(P_QOH) FROM PRODUCT` could be resolved by reading only the first entry in the `P_QOH_NDX` index, without the need to access any of the data blocks for the `PRODUCT` table. (Remember that the index defaults to ascending order.)

You learned in Section 11.3 that columns with low sparsity are not good candidates for index creation. However, there are cases where an index in a low sparsity column would be helpful. For example, assume that the `EMPLOYEE` table has 122,483 rows. If you want to find out how many female employees are in the company, you would write a query such as:

```
SELECT COUNT(EMP_SEX) FROM EMPLOYEE WHERE EMP_SEX = 'F';
```

If you do not have an index for the `EMP_SEX` column, the query would have to perform a full table scan to read all `EMPLOYEE` rows—and each full row includes attributes you do not need. However, if you have an index on `EMP_SEX`, the query could be answered by reading only the index data, without the need to access the employee data at all.

11.4.1 USING HINTS TO AFFECT OPTIMIZER CHOICES

Although the optimizer generally performs very well under most circumstances, in some instances the optimizer might not choose the best execution plan. Remember, the optimizer makes decisions based on the existing statistics. If the statistics are old, the optimizer might not do a good job in selecting the best execution plan. Even with current statistics, the optimizer choice might 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 do that, you need to use hints. **Optimizer hints** are special instructions for the optimizer that are embedded inside the SQL command text. Table 11.5 summarizes a few of the most common optimizer hints used in standard SQL.

TABLE 11.5 Optimizer Hints

HINT	USAGE
ALL_ROWS	Instructs the optimizer to minimize the overall execution time, that is, to minimize the time it takes to return all rows in the query result set. This hint is generally used for batch mode processes. For example: <pre>SELECT /*+ ALL_ROWS */ * FROM PRODUCT WHERE P_QOH < 10;</pre>
FIRST_ROWS	Instructs the optimizer to minimize the time it takes to process the first set of rows, that is, to minimize the time it takes to return only the first set of rows in the query result set. This hint is generally used for interactive mode processes. For example: <pre>SELECT /*+ FIRST_ROWS */ * FROM PRODUCT WHERE P_QOH < 10;</pre>
INDEX(name)	Forces the optimizer to use the <code>P_QOH_NDX</code> index to process this query. For example: <pre>SELECT /*+ INDEX(P_QOH_NDX) */ * FROM PRODUCT WHERE P_QOH < 10;</pre>

Now that you are familiar with the way the DBMS processes SQL queries, let's turn our attention to some general SQL coding recommendations to facilitate the work of the query optimizer.

11.5 SQL PERFORMANCE TUNING

SQL performance tuning is evaluated from the client perspective. Therefore, the goal is to illustrate some common practices used to write efficient SQL code. A few words of caution are appropriate:

1. Most current-generation relational DBMSs perform automatic query optimization at the server end.
2. Most SQL performance optimization techniques are DBMS-specific, and therefore, are rarely portable, even across different versions of the same DBMS. Part of the reason for this behavior is the constant advancement in database technologies.

Does this mean that you should not worry about how a SQL query is written because the DBMS will always optimize it? No, because there is considerable room for improvement. (The DBMS uses *general* optimization techniques, rather than focusing on specific techniques dictated by the special circumstances of the query execution.) A poorly written SQL query can, *and usually will*, bring the database system to its knees from a performance point of view. The majority of current database performance problems are related to poorly written SQL code. Therefore, although a DBMS provides general optimizing services, a carefully written query almost always outperforms a poorly written one.

Although SQL data manipulation statements include many different commands (such as INSERT, UPDATE, DELETE, and SELECT), most recommendations in this section are related to the use of the SELECT statement, and in particular, the use of indexes and how to write conditional expressions.

11.5.1 INDEX SELECTIVITY

Indexes are the most important technique used in SQL performance optimization. The key is to know when an index is used. As a general rule, indexes are likely to be used:

- When an indexed column appears by itself in a search criteria of a WHERE or HAVING clause.
- When an indexed column appears by itself in a GROUP BY or ORDER BY clause.
- When a MAX or MIN function is applied to an indexed column.
- When the data sparsity on the indexed column is high.

Indexes are very useful when you want to select a small subset of rows from a large table based on a given condition. If an index exists for the column used in the selection, the DBMS may choose to use it. The objective is to create indexes with high selectivity. **Index selectivity** is a measure of how likely an index will be used in query processing. Here 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 or FK.* If you do join operations on columns other than the primary and foreign keys, you might be better off declaring indexes in those columns.

You cannot always use an index to improve performance. For example, using the data shown in Table 11.6 in the next section, the creation of an index for P_MIN will not help the search condition $P_QOH > P_MIN * 1.10$. The reason is because in some DBMSs, *indexes are ignored when you use functions in the table attributes*. However, major databases (such as Oracle, SQL Server, and DB2) now support function-based indexes. A **function-based index** is an index based on a specific SQL function or expression. For example, you could create an index on YEAR(INV_DATE). Function-based indexes are especially useful when dealing with derived attributes. For example, you could create an index on $EMP_SALARY + EMP_COMMISSION$.

How many indexes should you create? It bears repeating that you should not create an index for every column in a table. Too many indexes will slow down INSERT, UPDATE, and DELETE operations, especially if the table contains many thousands of rows. Furthermore, some query optimizers will choose only one index to be the driving index for a query, even if your query uses conditions in many different indexed columns. Which index does the optimizer use? If you use the cost-based optimizer, the answer will change with time as new rows are added or deleted from the tables. In any case, you should create indexes in all search columns and then let the optimizer choose. It's important to constantly evaluate the index usage—monitor, test, evaluate, and improve it if performance is not adequate.

11.5.2 CONDITIONAL EXPRESSIONS

A conditional expression is normally expressed within the WHERE or HAVING clauses of a SQL statement. Also known as conditional criteria, a conditional expression restricts the output of a query to only the rows matching the conditional criteria. Generally, the conditional criteria have the form shown in Table 11.6.

TABLE 11.6 Conditional Criteria

OPERAND1	CONDITIONAL OPERATOR	OPERAND2
P_PRICE	>	10.00
V_STATE	=	FL
V_CONTACT	LIKE	Smith%
P_QOH	>	P_MIN * 1.10

In Table 11.6, note that an operand can be:

- A simple column name such as P_PRICE or V_STATE.
- A literal or a constant such as the value 10.00 or the text 'FL'.
- An expression such as $P_MIN * 1.10$.

Most of the query optimization techniques mentioned next are designed to make the optimizer's work easier. Let's examine some common practices used to write efficient 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. For example, $P_PRICE > 10.00$ is faster than $P_QOH > P_MIN * 1.10$ because the DBMS must evaluate the $P_MIN * 1.10$ expression first. The use of functions in expressions also adds to the total query execution time. For example, if your condition is $UPPER(V_NAME) = 'JIM'$, try to use $V_NAME = 'Jim'$ if all names in the V_NAME column are stored with proper capitalization.
- *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, the CPU handles numeric comparisons (integer and decimal) faster 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. The reason is because there will almost always be more “greater than” or “less than” values than exactly “equal” values in the index. With the exception of NULL, the slowest 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 when the sparsity of the data is high, that is, when 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 this was done 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, the whole set of conditions will be evaluated as false. 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. For example, look at the following condition list:

```
P_PRICE > 10 AND V_STATE = 'FL'
```

If you know that only a few vendors are located in Florida, you could rewrite this condition as:

```
V_STATE = 'FL' AND P_PRICE > 10
```

- *When using multiple OR conditions, put the condition most likely to be true first.* By doing this, the DBMS will stop evaluating the remaining conditions as soon as it finds a conditional expression that is evaluated to be true. Remember, for multiple OR conditions to evaluate to true, only one of the conditions must be evaluated to true.

NOTE

Oracle does not evaluate queries as described here. Instead, Oracle evaluates conditions from last to first.

- *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'`.

11.6 QUERY FORMULATION

Queries are usually written to answer questions. For example, if an end user gives you a sample output and tells you to match that output format, you must write the corresponding SQL. To get the job done, you must carefully evaluate what columns, tables, and computations are required to generate the desired output. And to do that, you must have a good understanding of the database environment and of the database that will be the focus of your SQL code.

This section focuses on SELECT queries because they are the queries you will find in most applications. To formulate a query, you would normally follow the steps outlined below.

1. *Identify what columns and computations are required.* The first step is to clearly determine what data values you want to return. Do you want to return just the names and addresses, or do you also want to include some computations? Remember that all columns in the SELECT statement should return single values.
 - a. Do you need simple expressions? That is, do you need to multiply the price times the quantity on hand to generate the total inventory cost? You might need some single attribute functions such as DATE(), SYSDATE(), or ROUND().
 - b. Do you need aggregate functions? If you need to compute the total sales by product, you should use a GROUP BY clause. In some cases, you might need to use a subquery.
 - c. Determine the granularity of the raw data required for your output. Sometimes, you might need to summarize data that are not readily available on any table. In such cases, you might consider breaking the query into multiple subqueries and storing those subqueries as views. Then you could create a top-level query that joins those views and generates the final output.
2. *Identify the source tables.* Once you know what columns are required, you can determine the source tables used in the query. Some attributes appear in more than one table. In those cases, try to use the least number of tables in your query to minimize the number of join operations.
3. *Determine how to join the tables.* Once you know what tables you need in your query statement, you must properly identify how to join the tables. In most cases, you will use some type of natural join, but in some instances, you might need to use an outer join.
4. *Determine what selection criteria is needed.* Most queries involve some type of selection criteria. In this case, you must determine what operands and operators are needed in your criteria. Ensure that the data type and granularity of the data in the comparison criteria are correct.
 - a. *Simple comparison.* In most cases, you will be comparing single values. For example, P_PRICE > 10.
 - b. *Single value to multiple values.* If you are comparing a single value to multiple values, you might need to use an IN comparison operator. For example, V_STATE IN ('FL', 'TN', 'GA').
 - c. *Nested comparisons.* In other cases, you might need to have some nested selection criteria involving subqueries. For example: P_PRICE > = (SELECT AVG(P_PRICE) FROM PRODUCT).
 - d. *Grouped data selection.* On other occasions, the selection criteria might apply not to the raw data, but to the aggregate data. In those cases, you need to use the HAVING clause.
5. *Determine in what order to display the output.* Finally, the required output might be ordered by one or more columns. In those cases, you need to use the ORDER BY clause. Remember that the ORDER BY clause is one of the most resource-intensive operations for the DBMS.

11.7 DBMS PERFORMANCE TUNING

DBMS performance tuning includes global tasks such as managing the DBMS processes in primary memory (allocating memory for caching purposes) and managing the structures in physical storage (allocating space for the data files).

Fine-tuning the performance of the DBMS also includes applying several practices examined in the previous section. For example, the DBA must work with developers to ensure that the queries perform as expected—creating the indexes to speed up query response time and generating the database statistics required by cost-based optimizers.

DBMS performance tuning at the server end focuses on setting the parameters used for:

- *Data cache.* The data cache must be set large enough to permit as many data requests as possible to be serviced from the cache. Each DBMS has settings that control the size of the data cache; some DBMSs might require a restart. This cache is shared among all database users. The majority of primary memory resources will be allocated to the data cache.
- *SQL cache.* The SQL cache stores the most recently executed SQL statements (after the SQL statements have been parsed by the optimizer). Generally, if you have an application with multiple users accessing a database, the *same* query will likely be submitted by many different users. In those cases, the DBMS will parse the query only once and execute it many times, using the same access plan. In that way, the second and subsequent SQL requests for the same query are served from the SQL cache, skipping the parsing phase.
- *Sort cache.* The sort cache is used as a temporary storage area for ORDER BY or GROUP BY operations, as well as for index-creation functions.
- *Optimizer mode.* Most DBMSs operate in one of two optimization modes: cost-based or rule-based. Others automatically determine the optimization mode based on whether database statistics are available. For example, the DBA is responsible for generating the database statistics that are used by the cost-based optimizer. If the statistics are not available, the DBMS uses a rule-based optimizer.

Managing the physical storage details of the data files also plays an important role in DBMS performance tuning. Following are some general recommendations for the creation of databases.

- Use **RAID** (redundant array of independent disks) 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 shows the most common RAID configurations.

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.

- Minimize disk contention. Use multiple, independent storage volumes with independent spindles (a spindle is a rotating disk) to minimize hard disk cycles. Remember, a database is composed of many table spaces, each with a particular function. In turn, each table space is composed of several data files in which the data are actually stored. A database should have at least the following table spaces:
 - *System table space*. This is used to store the data dictionary tables. It is the most frequently accessed table space and should be stored in its own volume.
 - *User data table space*. This is used to store end-user data. You should create as many user data table spaces and data files as are required to balance performance and usability. For example, you can create and assign a different user data table space for each application and/or for each distinct group of users; but not necessary for each user.
 - *Index table space*. This is used to store indexes. You can create and assign a different index table space for each application and/or for each group of users. The index table space data files should be stored on a storage volume that is separate from user data files or system data files.
 - *Temporary table space*. This is used as a temporary storage area for merge, sort, or set aggregate operations. You can create and assign a different temporary table space for each application and/or for each group of users.
 - *Rollback segment table space*. This is used for transaction-recovery purposes.
- Put high-usage tables in their own table spaces. By doing this, the database minimizes conflict with other tables.
- Assign separate data files in separate storage volumes for the indexes, system, and high-usage tables. This ensures that index operations will not conflict with end-user data or data dictionary table access operations. Another advantage of this approach is that you can use different disk block sizes in different volumes. For example, the data volume can use a 16K block size, while the index volume can use an 8K block size. Remember that the index record size is generally smaller, and by changing the block size you will be reducing contention and/or minimizing I/O operations. This is very important; many database administrators overlook indexes as a source of contention. By using separate storage volumes and different block sizes, the I/O operations on data and indexes will happen asynchronously (at different times), and more importantly, the likelihood of write operations blocking read operations is reduced (as page locks tend to lock less records).
- Take advantage of the various table storage organizations available in the database. For example, in Oracle consider the use of index organized tables (IOT); in SQL Server consider clustered index tables. An **index organized table** (or **clustered index table**) is a table that stores the end user data and the index data in consecutive locations on permanent storage. This type of storage organization provides a performance advantage to tables that are commonly accessed by a given index order. This is due to the fact that the index contains the index key as well as the data rows, and therefore the DBMS tends to perform fewer I/O operations.
- Partition tables based on usage. Some RDBMSs support horizontal partitioning of tables based on attributes. (See Chapter 12, Distributed Database Management Systems.) By doing so, a single SQL request could be processed by multiple data processors. Put the table partitions closest to where they are used the most.
- Use denormalized tables where appropriate. Another performance-improving technique involves taking a table from a higher normal form to a lower normal form—typically, from third to second normal form. This technique adds data duplication, but it minimizes join operations. (Denormalization was discussed in Chapter 5, Normalization of Database Tables.)
- Store computed and aggregate attributes in tables. In short, use derived attributes in your tables. For example, you might add the invoice subtotal, the amount of tax, and the total in the INVOICE table. Using derived attributes minimizes computations in queries and join operations.

11.8 QUERY OPTIMIZATION EXAMPLE

Now that you have learned the basis of query optimization, you are ready to test your new knowledge. Let's use a simple example to illustrate how the query optimizer works and how you can help it do its work. The example is based on the QOVENDOR and QOPRODUCT tables. Those tables are similar to the ones you used in previous chapters. However, the QO prefix is used for the table name to ensure that you do not overwrite previous tables.



ONLINE CONTENT

The databases and scripts used in this chapter can be found in the Student Online Companion for this book.

To perform this query optimization illustration, you will be using the Oracle SQL*Plus interface. Some preliminary work must be done before you can start testing query optimization. The following steps will guide you through this preliminary work:

1. Log in to Oracle SQL Plus using the username and password provided by your instructor.
2. Create a fresh set of tables using the QRYOPTDATA.SQL script file located on the Student Online Companion for this book. This step is necessary so Oracle has a new set of tables and the new tables contain no statistics. At the SQL> prompt, type:
`@path\ QRYOPTDATA.SQL`
where *path* is the location of the file in your computer.
3. Create the PLAN_TABLE. The PLAN_TABLE is a special table used by Oracle to store the access plan information for a given query. End users can then query the PLAN_TABLE to see how Oracle will execute the query. To create the PLAN_TABLE, run the UTLXPLAN.SQL script file located in the RDBMS\ADMIN folder of your Oracle RDBMS installation. The UTLXPLAN.SQL script file is also found in the Student Online Companion for this book. At the SQL prompt, type:
`@path\UTLXPLAN.SQL`

You use the EXPLAIN PLAN command to store the execution plan of a SQL query in the PLAN_TABLE. Then, you would use the SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY) command to display the access plan for a given SQL statement.

NOTE

Oracle 10g automatically defaults to cost-based optimization without giving you a choice. All previous versions of Oracle default to the Choose optimization mode, which implies that the DBMS will choose either rule-based or cost-based optimization, depending on the availability of table statistics.

To see the access plan used by the DBMS to execute your query, use the EXPLAIN PLAN and SELECT statements as shown in Figure 11.5. Note that the first SQL statement in Figure 11.5 generates the statistics for the QOVENDOR table. Also note that the initial access plan in Figure 11.5 uses a full table scan on the QOVENDOR table and that the cost of the plan is 4.

FIGURE 11.5 Initial explain plan

```

Oracle SQL*Plus
File Edit Search Options Help
SQL> ANALYZE TABLE QOVENDOR COMPUTE STATISTICS;

Table analyzed.

SQL> EXPLAIN PLAN FOR SELECT * FROM QOVENDOR WHERE U_NAME LIKE 'B%' ORDER BY U_AREACODE;

Explained.

SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY);

PLAN_TABLE_OUTPUT
-----
Plan hash value: 1837703589

-----
| Id | Operation          | Name          | Rows | Bytes | Cost (%CPU)| Time     |
-----|-----|-----|-----|-----|-----|-----|
|  0 | SELECT STATEMENT   |               |      |      |      |      |
|  1 |   SORT ORDER BY   |               |      |      |      |      |
|* 2 |    TABLE ACCESS FULL| QOVENDOR     |      |      |      |      |
-----

Predicate Information (identified by operation id):

PLAN_TABLE_OUTPUT
-----

      2 - filter("U_NAME" LIKE 'B%')

14 rows selected.

SQL>

```

Let's now create an index on V_AREACODE (note that V_AREACODE is used in the ORDER BY clause) and see how that affects the access plan generated by the cost-based optimizer. The results are shown in Figure 11.6.

FIGURE 11.6 Explain plan after index on V_AREACODE

```

Oracle SQL*Plus
File Edit Search Options Help
SQL> CREATE INDEX QOV_NDX1 ON QOVENDOR(V_AREACODE);
Index created.
SQL> ANALYZE TABLE QOVENDOR COMPUTE STATISTICS;
Table analyzed.
SQL> EXPLAIN PLAN FOR SELECT * FROM QOVENDOR WHERE U_NAME LIKE 'B%' ORDER BY V_AREACODE;
Explained.
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY);
PLAN_TABLE_OUTPUT
-----
Plan hash value: 2305289760

-----
| Id | Operation                                | Name      | Rows | Bytes | Cost (%CPU)| Time     |
-----+-----+-----+-----+-----+-----+-----+
|  0 | SELECT STATEMENT                          |           |     1 |    38 |      2 (0)| 00:00:01 |
|*  1 |  TABLE ACCESS BY INDEX ROWID            | QOVENDOR  |     1 |    38 |      2 (0)| 00:00:01 |
|  2 |    INDEX FULL SCAN                        | QOV_NDX1  |    15 |           |      1 (0)| 00:00:01 |
-----+-----+-----+-----+-----+

Predicate Information (identified by operation id):
-----
   1 - filter("U_NAME" LIKE 'B%')

14 rows selected.

SQL>

```

In Figure 11.6, note that the new access plan cuts the cost of executing the query by half! Also note that this new plan scans the QOV_NDX1 index and accesses the QOVENDOR rows, using the index row ID. (Remember that access by row ID is one of the fastest access methods.) In this case, the creation of the QOV_NDX1 index had a positive impact on overall query optimization results.

At other times, indexes do not necessarily help in query optimization. This is the case when you have indexes on small tables or when the query accesses a great percentage of table rows anyway. Let's see what happens when you create an index on V_NAME. The new access plan is shown in Figure 11.7. (Note that V_NAME is used on the WHERE clause as a conditional expression operand.)

FIGURE 11.7 Explain plan after index on V_NAME

```

Oracle SQL*Plus
File Edit Search Options Help
SQL> CREATE INDEX QOV_NDX2 ON QOVENDOR(V_NAME);

Index created.

SQL> ANALYZE TABLE QOVENDOR COMPUTE STATISTICS;

Table analyzed.

SQL> EXPLAIN PLAN FOR SELECT * FROM QOVENDOR WHERE V_NAME LIKE 'B%' ORDER BY V_AREACODE;

Explained.

SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY);

PLAN_TABLE_OUTPUT
-----
Plan hash value: 2305289760

-----
| Id | Operation                | Name      | Rows | Bytes | Cost (%CPU)| Time     |
-----+-----+-----+-----+-----+-----+-----+
|  0 | SELECT STATEMENT         |           |     1 |    38 |      2 (0)| 00:00:01 |
|*  1 | TABLE ACCESS BY INDEX ROWID| QOVENDOR |     1 |    38 |      2 (0)| 00:00:01 |
|  2 |   INDEX FULL SCAN        | QOV_NDX1 |    15 |           |      1 (0)| 00:00:01 |
-----

Predicate Information (identified by operation id):
-----
   1 - filter("V_NAME" LIKE 'B%')

14 rows selected.

SQL>

```


As you can see in Figure 11.7, creation of the second index did not help the query optimization. However, there are occasions when an index might be used by the optimizer, but it is not executed because of the way in which the query is written. For example, Figure 11.8 shows the access plan for a different query using the V_NAME column.

FIGURE 11.8 Access plan using index on V_NAME

```

Oracle SQL*Plus
File Edit Search Options Help
SQL> EXPLAIN PLAN FOR SELECT U_NAME, P_CODE FROM QOVENDOR U, QOPRODUCT P
2 WHERE U.U_CODE = P.U_CODE AND U_NAME = 'ORDVA, Inc.';

Explained.
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY);

PLAN_TABLE_OUTPUT
-----
Plan hash value: 3956542569

-----
| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |
-----
| 0 | SELECT STATEMENT | | 2 | 74 | 6 (17)| 00:00:01 |
|* 1 | HASH JOIN | | 2 | 74 | 6 (17)| 00:00:01 |
| 2 | TABLE ACCESS BY INDEX ROWID | QOVENDOR | 1 | 17 | 2 (0)| 00:00:01 |
|* 3 | INDEX RANGE SCAN | QOU_NDX2 | 1 | 1 | 1 (0)| 00:00:01 |
| 4 | TABLE ACCESS FULL | QOPRODUCT | 16 | 320 | 3 (0)| 00:00:01 |
-----

Predicate Information (identified by operation id):
-----
 1 - access("U"."U_CODE"="P"."U_CODE")
 3 - access("U_NAME"='ORDVA, Inc.')
```

Note

- dynamic sampling used for this statement

21 rows selected.

SQL> |

In Figure 11.8, note that the access plan for this new query uses the QOV_NDX2 index on the V_NAME column. What would happen if you wrote the same query, using the UPPER function on V_NAME? The results of that action are illustrated in Figure 11.9.

FIGURE 11.9 Access plan using functions on indexed columns

```

Oracle SQL*Plus
File Edit Search Options Help
SQL> EXPLAIN PLAN FOR SELECT U_NAME, P_CODE FROM QOVENDOR U, QOPRODUCT P
2 WHERE U.U_CODE = P.U_CODE AND UPPER(U_NAME) = 'ORDVA, INC.';

Explained.
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY);

PLAN_TABLE_OUTPUT
-----
Plan hash value: 4061476548

-----
| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |
-----
| 0 | SELECT STATEMENT | | 1 | 37 | 7 (15)| 00:00:01 |
|* 1 | HASH JOIN | | 1 | 37 | 7 (15)| 00:00:01 |
|* 2 | VIEW | index$\_join$\_001 | 1 | 17 | 3 (0)| 00:00:01 |
|* 3 | HASH JOIN | | 1 | 17 | 1 (0)| 00:00:01 |
|* 4 | INDEX FAST FULL SCAN| QOV_NDX2 | 1 | 17 | 1 (0)| 00:00:01 |
| 5 | INDEX FAST FULL SCAN| SYS_C005802 | 1 | 17 | 1 (0)| 00:00:01 |
| 6 | TABLE ACCESS FULL | QOPRODUCT | 16 | 320 | 3 (0)| 00:00:01 |
-----

Predicate Information (identified by operation id):
-----
 1 - access("U"."U_CODE"="P"."U_CODE")
 2 - filter(UPPER("U_NAME")='ORDVA, INC.')
 3 - access(ROWID=ROWID)
 4 - filter(UPPER("U_NAME")='ORDVA, INC.')

Note
-----
 - dynamic sampling used for this statement

25 rows selected.

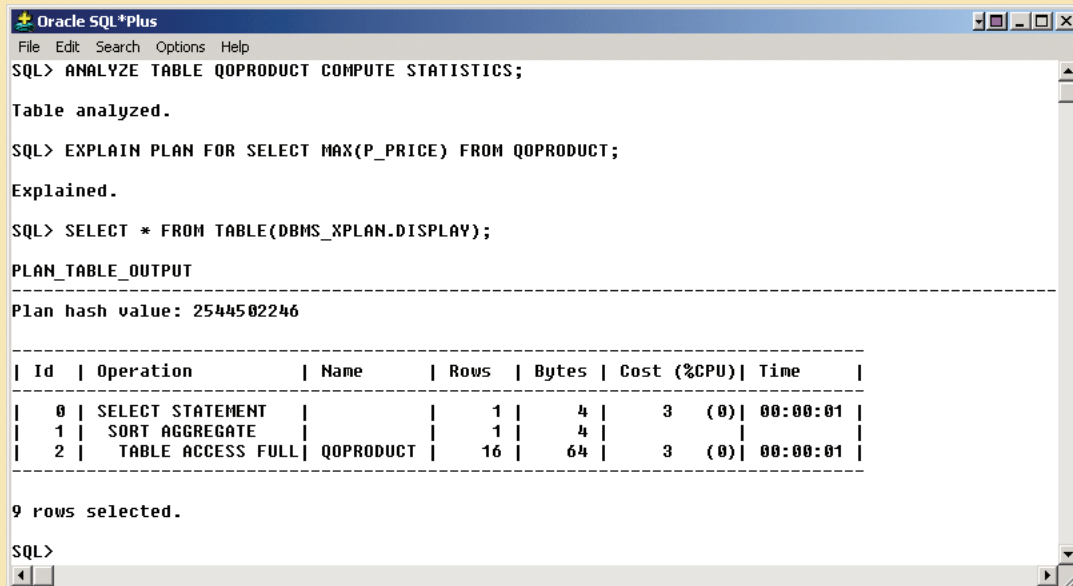
SQL>

```

As Figure 11.9 shows, the use of a function on an indexed column caused the DBMS to perform additional operations that increased the cost of the query. Note that the same query might produce different costs if your tables contain many more rows and if the index sparsity is different.

Let's now use the table QOPRODUCT to demonstrate how an index can help when aggregate function queries are being run. For example, Figure 11.10 shows the access plan for a SELECT statement using the MAX(P_PRICE) aggregate function. Note that this plan uses a full table scan with a total cost of 3.

FIGURE 11.10 First explain plan: aggregate function on a non-indexed column



```
Oracle SQL*Plus
File Edit Search Options Help
SQL> ANALYZE TABLE QOPRODUCT COMPUTE STATISTICS;

Table analyzed.

SQL> EXPLAIN PLAN FOR SELECT MAX(P_PRICE) FROM QOPRODUCT;

Explained.

SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY);

PLAN_TABLE_OUTPUT
-----
Plan hash value: 2544502246

-----
| Id | Operation          | Name      | Rows | Bytes | Cost (%CPU)| Time     |
-----|-----|-----|-----|-----|-----|-----|
|  0 | SELECT STATEMENT   |           |     1 |     4 |     3 (0)| 00:00:01 |
|  1 |  SORT AGGREGATE    |           |     1 |     4 |           |          |
|  2 |   TABLE ACCESS FULL| QOPRODUCT |    16 |    64 |     3 (0)| 00:00:01 |
-----

9 rows selected.

SQL>
```

A cost of 3 is very low already, but could you improve it? Yes, you could improve the previous query performance by creating an index on P_PRICE. Figure 11.11 shows how the plan cost is reduced by two-thirds after the index is created and the QOPRODUCT table is analyzed. Also note that the second version of the access plan uses only the index QOP_NDX2 to answer the query; *the QOPRODUCT table is never accessed*.

FIGURE 11.11 Second explain plan: aggregate function on an indexed column

```

Oracle SQL*Plus
File Edit Search Options Help
SQL> CREATE INDEX QOP_NDX2 ON QOPRODUCT(P_PRICE);

Index created.

SQL> ANALYZE TABLE QOPRODUCT COMPUTE STATISTICS;

Table analyzed.

SQL> EXPLAIN PLAN FOR SELECT MAX(P_PRICE) FROM QOPRODUCT;

Explained.

SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY);

PLAN_TABLE_OUTPUT
-----
Plan hash value: 3423609809

-----
| Id | Operation                      | Name      | Rows | Bytes | Cost (%CPU)| Time     |
-----|-----|-----|-----|-----|-----|-----|
|  0 | SELECT STATEMENT                |           |     1 |     4 |     1 (0)| 00:00:01 |
|  1 |   SORT AGGREGATE                |           |     1 |     4 |     1 (0)| 00:00:01 |
|  2 |    INDEX FULL SCAN (MIN/MAX)    | QOP_NDX2 |    16 |    64 |     1 (0)| 00:00:01 |
-----

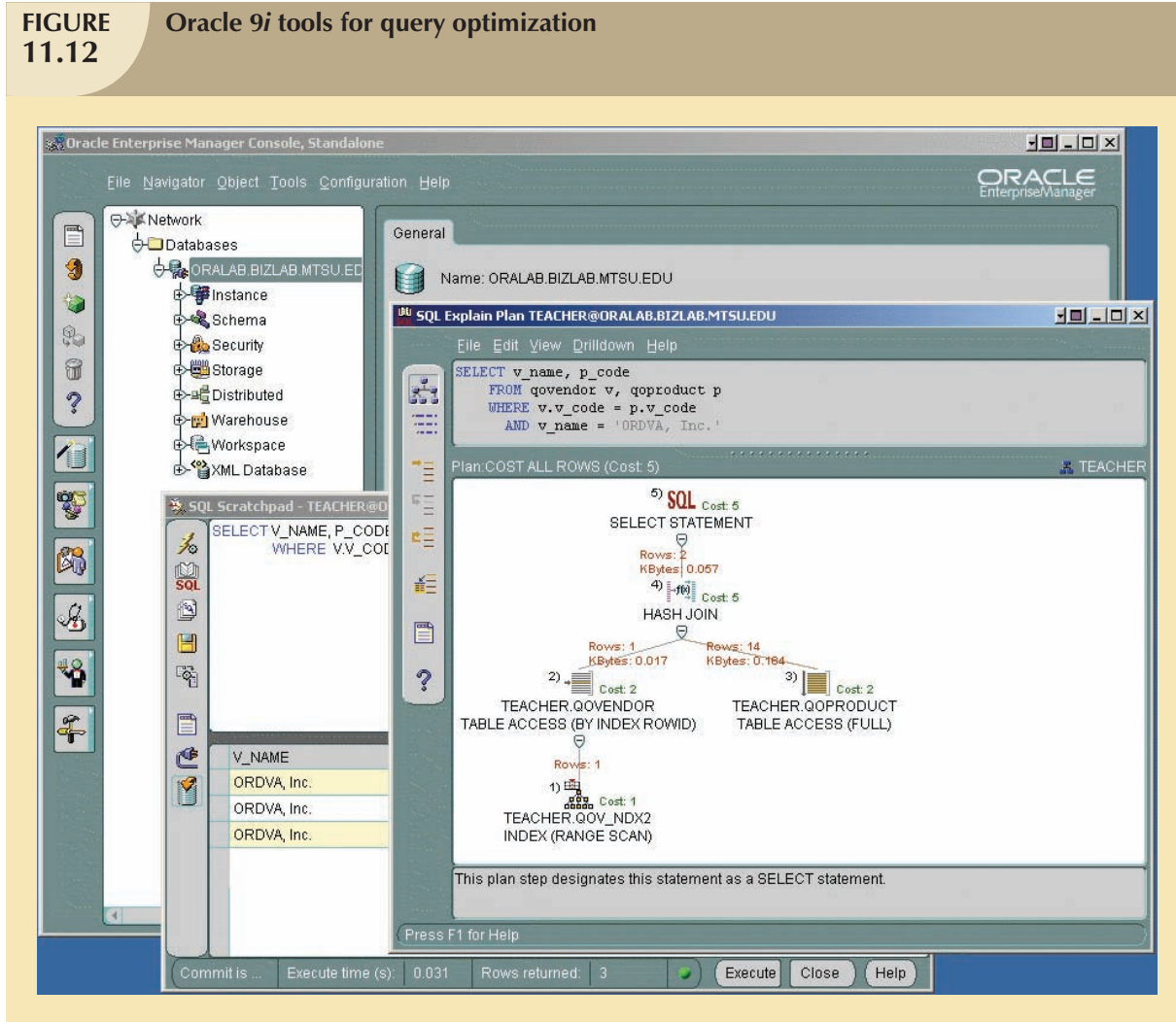
9 rows selected.

SQL>

```

Although the few examples in this section show how important proper index selection is for query optimization, you also saw examples in which index creation does not improve query performance. As a DBA, you should be aware that the main goal is to optimize overall database performance—not just for a single query, but for all requests and query types. Most database systems provide advanced graphical tools for performance monitoring and testing. For example, Figure 11.12 shows the graphical representation of the access plan using the Oracle 9i graphical tools. (Oracle 10g does not include this interface.)

FIGURE 11.12 Oracle 9i tools for query optimization



S U M M A R Y

- Database performance tuning refers to a set of activities and procedures designed to ensure that an end-user query is processed by the DBMS in the minimum amount of time.
- SQL performance tuning refers to the activities on the client side that are designed to generate SQL code that returns the correct answer in the least amount of time, using the minimum amount of resources at the server end.
- DBMS performance tuning refers to activities on the server side that are oriented to ensure that the DBMS is properly configured to respond to clients' requests in the fastest way possible while making optimum use of existing resources.
- The DBMS architecture is represented by the many processes and structures (in memory and in permanent storage) used to manage a database.
- Database statistics refers to a number of measurements gathered by the DBMS that 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, and temporary space available. The DBMS uses the statistics to make critical decisions about improving the query processing efficiency.
- DBMSs process queries in three phases:
 - *Parsing*. The DBMS parses the SQL query and chooses the most efficient access/execution plan.
 - *Execution*. The DBMS executes the SQL query, using the chosen execution plan.
 - *Fetching*. The DBMS fetches the data and sends the result set back to the client.
- Indexes are crucial in the process that speeds up data access. Indexes facilitate searching, sorting, and using aggregate functions and join operations. The improvement in data access speed occurs because an index is an ordered set of values that contains the index key and pointers. Data sparsity refers to the number of different values a column could possibly have. Indexes are recommended in high-sparsity columns used in search conditions.
- During query optimization, the DBMS must choose what indexes to use, how to perform join operations, which table to use first, and so on. Each DBMS has its own algorithms for determining the most efficient way to access the data. The two most common approaches are rule-based optimization and cost-based optimization.
 - A rule-based optimizer uses preset rules and points to determine the best approach to execute a query. The rules assign a "fixed 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.
- Hints are used to change the optimizer mode for the current SQL statement. Hints are special instructions for the optimizer that are embedded inside the SQL command text.
- SQL performance tuning deals with writing queries that make good use of the statistics. In particular, queries should make good use of indexes. Indexes are very useful when you want to select a small subset of rows from a large table based on a condition. When an index exists for the column used in the selection, the DBMS may choose to use it. The objective is to create indexes with high selectivity. Index selectivity is a measure of how likely an index will be used in query processing. It is also important to write conditional statements using some common principles.

- Query formulation deals with how to translate business questions into specific SQL code to generate the required results. To do this, you must carefully evaluate what columns, tables, and computations are required to generate the desired output.
- DBMS performance tuning includes tasks such as managing the DBMS processes in primary memory (allocating memory for caching purposes) and managing the structures in physical storage (allocating space for the data files).

KEY TERMS

access plan, 449	extends, 445	query optimizer, 449
cost-based optimizer, 453	function-based index, 457	query processing bottleneck, 450
database performance tuning, 443	index organized table or cluster indexed table, 461	RAID, 460
database statistics, 446	index selectivity, 456	rule-based optimizer, 453
data cache or buffer cache, 445	input/output (I/O) request, 446	SQL cache or procedure cache, 445
data files, 445	optimizer hints, 455	SQL performance tuning, 444
data sparsity, 452		table space or file group, 445
DBMS performance tuning, 444		



ONLINE CONTENT

Answers to selected Review Questions and Problems for this chapter are contained in the Student Online Companion for this book.

REVIEW QUESTIONS

1. What is SQL performance tuning?
2. What is database performance tuning?
3. What is the focus of most performance-tuning activities, and why does that focus exist?
4. What are database statistics, and why are they important?
5. How are database statistics obtained?
6. What database statistics measurements are typical of tables, indexes, and resources?
7. How is the processing of SQL DDL statements (such as CREATE TABLE) different from the processing required by DML statements?
8. In simple terms, the DBMS processes queries in three phases. What are those phases, and what is accomplished in each phase?
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.)
10. What is the difference between a rule-based optimizer and a cost-based optimizer?
11. What are optimizer hints, and how are they used?
12. What are some general guidelines for creating and using indexes?
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?
14. What recommendations would you make for managing the data files in a DBMS with many tables and indexes?
15. What does RAID stand for, and what are some commonly used RAID levels?

P R O B L E M S

Problems 1 and 2 are based on the following query:

```
SELECT    EMP_LNAME, EMP_FNAME, EMP_AREACODE, EMP_SEX
FROM      EMPLOYEE
WHERE     EMP_SEX = 'F' AND EMP_AREACODE = '615'
ORDER BY  EMP_LNAME, EMP_FNAME;
```

1. What is the likely data sparsity of the EMP_SEX column?
2. What indexes should you create? Write the required SQL commands.
3. Using Table 11.4 as an example, create two alternative access plans. Use the following assumptions:
 - a. There are 8,000 employees.
 - b. There are 4,150 female employees.
 - c. There are 370 employees in area code 615.
 - d. There are 190 female employees in area code 615.

Problems 4–6 are based on the following query:

```
SELECT    EMP_LNAME, EMP_FNAME, EMP_DOB, YEAR(EMP_DOB) AS YEAR
FROM      EMPLOYEE
WHERE     YEAR(EMP_DOB) = 1966;
```

4. What is the likely data sparsity of the EMP_DOB column?
5. Should you create an index on EMP_DOB? Why or why not?
6. What type of database I/O operations will likely be used by the query? (See Table 11.3.)

Problems 7–10 are based on the ER model shown in Figure P11.7 and on the query shown after the figure. Given the following query:

```
SELECT    P_CODE, P_PRICE
FROM      PRODUCT
WHERE     P_PRICE >= (SELECT AVG(P_PRICE) FROM PRODUCT);
```

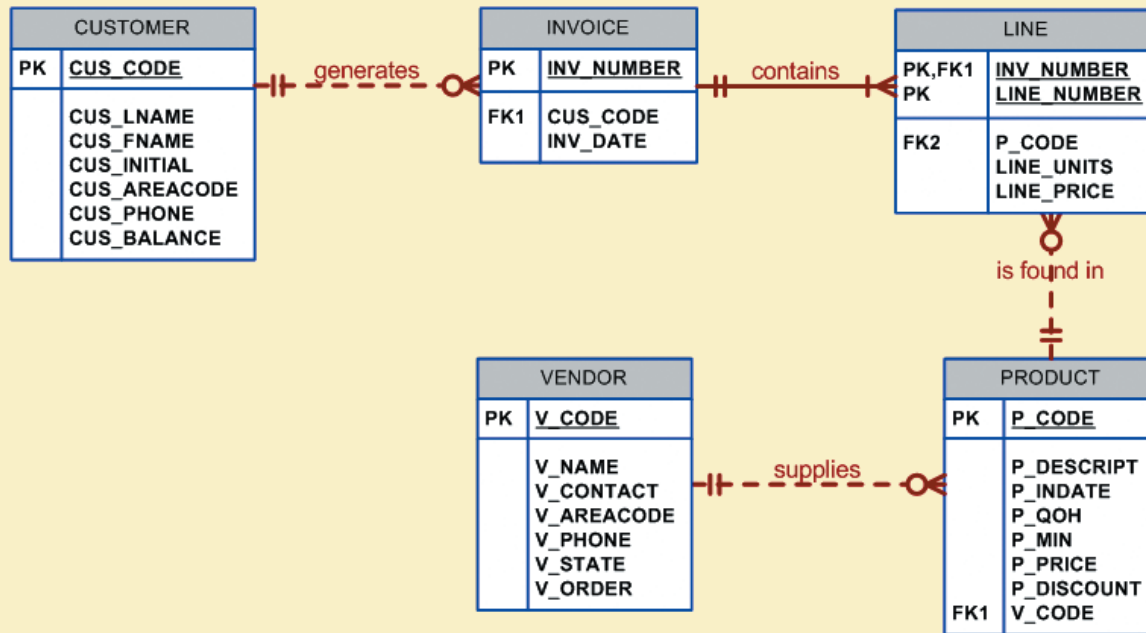
7. Assuming that there are no table statistics, what type of optimization will the DBMS use?
8. What type of database I/O operations will likely be used by the query? (See Table 11.3.)
9. What is the likely data sparsity of the P_PRICE column?
10. Should you create an index? Why or why not?

Problems 11–14 are based on the following query:

```
SELECT    P_CODE, SUM(LINE_UNITS)
FROM      LINE
GROUP BY  P_CODE
HAVING    SUM(LINE_UNITS) > (SELECT MAX(LINE_UNITS) FROM LINE);
```

11. What is the likely data sparsity of the LINE_UNITS column?
12. Should you create an index? If so, what would the index column(s) be, and why would you create that index? If not, explain your reasoning.
13. Should you create an index on P_CODE? If so, write the SQL command to create that index. If not, explain your reasoning.
14. Write the command to create statistics for this table.

FIGURE P11.7 The Ch11_SaleCo ER model



Problems 15 and 16 are based on the following query:

```
SELECT    P_CODE, P_QOH*P_PRICE
FROM      PRODUCT
WHERE     P_QOH*P_PRICE > (SELECT AVG(P_QOH*P_PRICE) FROM PRODUCT)
```

15. What is the likely data sparsity of the P_QOH and P_PRICE columns?
16. Should you create an index, what would the index column(s) be, and why should you create that index?

Problems 17–21 are based on the following query:

```
SELECT    V_CODE, V_NAME, V_CONTACT, V_STATE
FROM      VENDOR
WHERE     V_STATE = 'TN'
ORDER BY  V_NAME;
```

17. What indexes should you create and why? Write the SQL command to create the indexes.
18. Assume that 10,000 vendors are distributed as shown in Table P11.18. What percentage of rows will be returned by the query?
19. What type of I/O database operations would most likely be used to execute that query?
20. Using Table 11.4 as an example, create two alternative access plans.
21. Assume that you have 10,000 different products stored in the PRODUCT table and that you are writing a Web-based interface to list all products with a quantity on hand (P_QOH) that is less than or equal to the minimum quantity, P_MIN. What optimizer hint would you use to ensure that your query returns the result set to the Web interface in the least time possible? Write the SQL code.

**TABLE
P11.18**

STATE	NUMBER OF VENDORS	STATE	NUMBER OF VENDORS
AK	15	MS	47
AL	55	NC	358
AZ	100	NH	25
CA	3244	NJ	645
CO	345	NV	16
FL	995	OH	821
GA	75	OK	62
HI	68	PA	425
IL	89	RI	12
IN	12	SC	65
KS	19	SD	74
KY	45	TN	113
LA	29	TX	589
MD	208	UT	36
MI	745	VA	375
MO	35	WA	258

Problems 22–24 are based on the following query:

```

SELECT    P_CODE, P_DESCRIPT, P_PRICE, P.V_CODE, V_STATE
FROM      PRODUCT P, VENDOR V
WHERE     P.V_CODE = V.V_CODE
          AND V_STATE = 'NY'
          AND V_AREACODE = '212'
ORDER BY  P_PRICE;

```

22. What indexes would you recommend?
23. Write the commands required to create the indexes you recommended in Problem 22.
24. Write the command(s) used to generate the statistics for the PRODUCT and VENDOR tables.

Problems 25 and 26 are based on the following query:

```

SELECT    P_CODE, P_DESCRIPT, P_QOH, P_PRICE, V_CODE
FROM      PRODUCT
WHERE     V_CODE = '21344'
ORDER BY  P_CODE;

```

25. What index would you recommend, and what command would you use?
26. How should you rewrite the query to ensure that it uses the index you created in your solution to Problem 25?

Problems 27 and 28 are based on the following query:

```
SELECT    P_CODE, P_DESCRIPT, P_QOH, P_PRICE, V_CODE
FROM      PRODUCT
WHERE     P_QOH < P_MIN
          AND P_MIN = P_REORDER
          AND P_REORDER = 50
ORDER BY  P_QOH;
```

27. Use the recommendations given in Section 11.5.2 to rewrite the query to produce the required results more efficiently.
28. What indexes would you recommend? Write the commands to create those indexes.

Problems 29–32 are based on the following query:

```
SELECT    CUS_CODE, MAX(LINE_UNITS*LINE_PRICE)
FROM      CUSTOMER NATURAL JOIN INVOICE NATURAL JOIN LINE
WHERE     CUS_AREACODE = '615'
GROUP BY  CUS_CODE;
```

29. Assuming that you generate 15,000 invoices per month, what recommendation would you give the designer about the use of derived attributes?
30. Assuming that you follow the recommendations you gave in Problem 29, how would you rewrite the query?
31. What indexes would you recommend for the query you wrote in Problem 30, and what SQL commands would you use?
32. How would you rewrite the query to ensure that the index you created in Problem 31 is used?