CHAPTER 8

Database Design and Performance Tuning

Abstract

This chapter looks at practical design choices that can impact the performance of a database. Topics include indexing, clustering, horizontal partitioning, and vertical partitioning.

Keywords

database indexes
database clustering
database partitioning
horizontal partitioning
vertical partitioning

How long are you willing to wait for a computer to respond to your request for information? 30 seconds? 10 seconds? 5 seconds? In truth, we humans are not very patient at all. Even five seconds can seem like an eternity, when you are waiting for something to appear on the screen. A database that has a slow response time to user queries usually means that you will have dissatisfied users.

Slow response times can be the result of any number of problems. You might be dealing with a client workstation that is not properly configured, a poorly written application program, a query involving multiple join operations, a query that requires reading large amounts of data from disk, a congested network, or even a DBMS that is not robust enough to handle to the volume of queries submitted to it.

One of the duties of a *database administrator* (DBA) is to optimize database performance (also known as *performance tuning*). This includes modifying the design—where possible, without compromising data integrity—to avoid performance bottlenecks, especially involving queries.

For the most part, a DBMS takes care of storing and retrieving data based on a user's commands without human intervention. The strategy used to process a data manipulation request is handled by the DBMS's *query optimizer*, a portion of the program that determines the most efficient sequence of relational algebra operations to perform a query.

Although most of the query optimizer's choices are out of the hands of a database designer or application developer, you can influence the behavior of the query optimizer and also optimize database performance to some extent with database design elements. In this chapter, you will be introduced to several such techniques.

Note: In Chapters 6 and 7 we discussed the impact of joins on queries as part of the theory of relational database design. You will learn about SQL syntax to avoid joins in Chapter 17.

Indexing

Indexing is a way of providing a fast access path to the values of a column or a concatenation of columns. New rows are typically added to the bottom of a table, resulting in a relatively random order of the values in any given column. Without some way of ordering the data, the only way the DBMS can search a column is by sequentially scanning each row from top to bottom. The larger a table becomes, the slower a sequential search will be.

Note: On average, in a table of N rows, a sequential search will need to examine N/2 rows to find a row that matches a query predicate. However, the only way for the DBMS to determine that no rows match the predicate is to examine all N rows. A table with 1000 rows requires, on average, looking at 500 rows; an unsuccessful search requires consulting all 1000 rows. However, the fast searching techniques provided by some indexes require looking at about 6 rows to find a matching row in that table of 1000 rows; an unsuccessful search requires consulting about 10 rows.

The alternative to indexing for ordering the rows in a table is sorting. A *sort* physically alters the position of rows in a table, placing the rows in order, starting with the first row in the table. Most SQL implementations do sort the virtual tables that are created as the result of queries when directed to do so by the SQL query. However, SQL provides no way to sort base tables … and there is good reason for this. Regardless of the sorting method used, as a table grows large (hundreds of thousands to millions of rows) sorting takes an enormously long time. Keeping a table in sorted order also means that, on average, half the rows in the table will need to be moved to make room for a new row. In addition, searching a sorted base table takes longer than searching an index, primarily because the index search requires less disk access. The overhead in maintaining indexes is far less than that required to sort base tables whenever a specific data order is needed.

The conceptual operation of an index is diagrammed in Figure 8.1. (The different weights of the lines have no significance other than to make it easier for you to follow the crossed lines.) In this illustration, you are looking at *Antique Opticals' item* relation and an index that provides fast access to rows in the table, based on the item's title. The index itself contains an ordered list of keys (the titles), along with the locations of the associated rows in the item table. The rows in the *item* table are in relatively random order. However, because the index is in alphabetical order by title, it can be searched quickly to locate a specific title. Then, the DBMS can use the information in the index to go directly to the correct row or rows in the *item* table, thus avoiding a slow sequential search of the base table's rows.

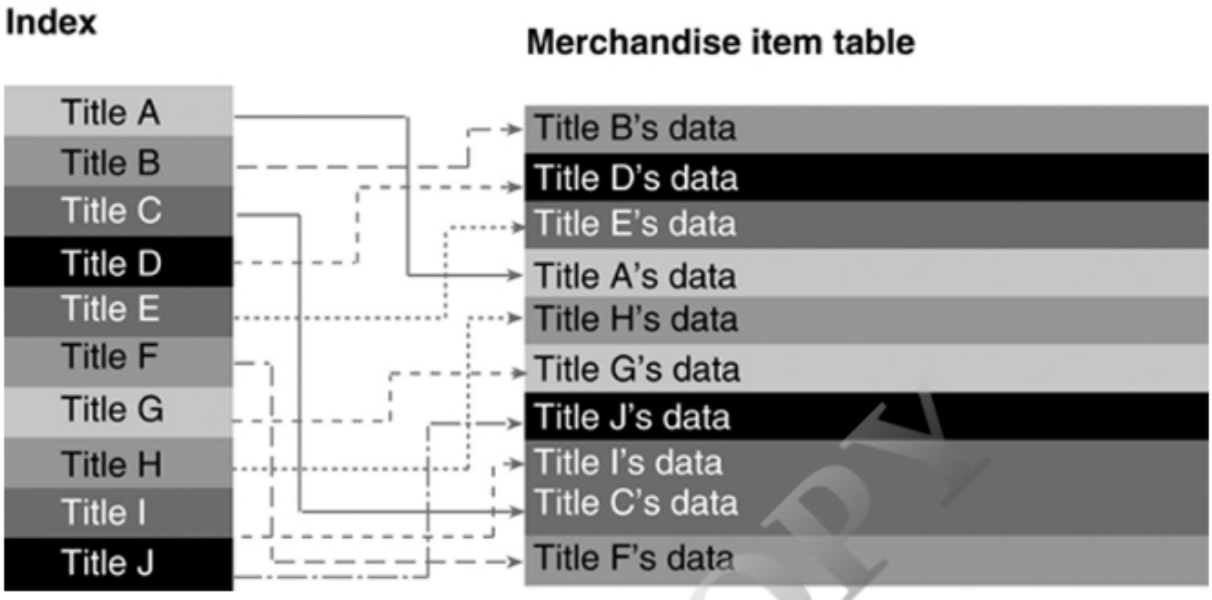

FIGURE 8.1 How an index works.

Once you have created an index, the DBMS's query optimizer will use the index whenever it determines that using the index will speed up data retrieval. You never need to access the index again yourself, unless you want to delete it.

When you create a primary key for a table, the DBMS automatically creates an index for that table using the primary key column or columns in the primary key as the index key. The first step in inserting a new row into a table is therefore verification that the index key (the primary key of the table) is unique in the index. In fact, uniqueness is enforced by requiring the index entries to be unique, rather than by actually searching the base table. This is much faster than attempting to verify uniqueness directly on the base table because the ordered index can be searched much more rapidly than the unordered base table.

Deciding Which Indexes to Create

You have no choice as to whether the DBMS creates indexes for your primary keys; you get them regardless of whether you want them. In addition, you can create indexes on any column or combination of columns you want. However, before you jump headfirst into creating indexes on every column in every table, there are some trade-offs to consider:

- Indexes take up space in the database. Given that disk space is relatively inexpensive today, this is usually not a major drawback.
- When you insert, modify, or delete data in indexed columns, the DBMS must update the index as well as the base table. This may slow down data modification operations, especially if the tables have a lot of rows.
- Indexes can significantly speed up access to data.

The trade-off is therefore generally between update speed and retrieval speed. A good rule of thumb is to create indexes for foreign keys and for other columns that are used frequently for queries that apply logical criteria to data. If you find that update speed is severely affected, you may choose at a later time to delete some of the indexes you created.

You should also try to avoid indexes on columns that contain *nondiscriminatory data*. Nondiscriminatory columns have only a few values throughout the entire table, such as Boolean columns that contain only true and false. Gender (male or female) is also nondiscriminatory. Although you may search on a column containing nondiscriminatory data—for example, a search for all open orders—an index will not provide much performance enhancement, because the DBMS must examine so many keys to complete the query.

Clustering

The slowest part of a DBMS's actions is retrieving data from or writing data to a disk. If you can cut down on the number of times the DBMS must read from or write to a disk, you can speed up overall database performance.

The trick to doing this is understanding that a database must retrieve an entire disk page of data at one time. The size of a page varies from one computing platform to another—it can be anywhere from 512 bytes to 4 K, with 1 K being typical on a PC—but data always travel to and from disk in page-sized units. Therefore, if you can get data that are often accessed together stored on the same disk page (or pages that are physically close together), you can speed up data access. This process is known as *clustering*, and is available with many large DBMSs (for example, Oracle).

Note: The term "clustering" has another meaning in the SQL standard. It refers to a group of catalogs (which in turn are groups of schemas) manipulated by the same DBMS. It has yet again another meaning when we talk about distributed databases, where it refers to a networked group of commodity PCs maintaining the same database. The use of the term in this section, however, is totally distinct from the SQL and distributed database meanings.

In practice, a cluster is designed to keep together rows related by matching primary and foreign keys. To define the cluster, you specify a column or columns on which the DBMS should form the cluster and the tables that should be included. Then, all rows that share the same value of the column or columns on which the cluster is based are stored as physically close together as possible. As a result, the rows in a table may be scattered across several disk pages, but matching primary and foreign keys are usually on the same disk page.

Clustering can significantly speed up join performance. However, just as with indexes, there are some trade-offs to consider when contemplating creating clusters:

- Because clustering involves physical placement of data in a file, a table can be clustered on only one column or combination of columns.

- Because clustering involves physical placement of data in a file, a table can be clustered on only one column or combination of columns.
- Clustering can slow down performance of operations that require a scan of the entire table because clustering may mean that the rows of any given table are scattered throughout many disk pages.
- Clustering can slow down insertion of data.
- Clustering can slow down modifying data in the columns on which the clustering is based.

Partitioning

Partitioning is the opposite of clustering. It involves the splitting of large tables into smaller ones so that the DBMS does not need to retrieve as much data at any one time. Consider, for example, what happens to *Antique Opticals*' *order* and *order item* tables over time. Assuming that the business is reasonably successful, those tables (especially *order item*) will become very large. Retrieval of data from those tables will therefore begin to slow down. It would speed up retrieval of open orders if filled orders and their items could be separated from open orders and their items.

There are two ways to partition a table, horizontally and vertically. *Horizontal partitioning* involves splitting the rows of a table between two or more tables with identical structures. *Vertical partitioning* involves splitting the columns of a table, placing them into two or more tables linked by the original table's primary key. As you might expect, there are advantages and disadvantages to both.

Horizontal Partitioning

Horizontal partitioning involves creating two or more tables with exactly the same structure, and splitting rows between those tables. *Antique Opticals* might use this technique to solve the problem with the *order* and *order items* tables becoming increasingly large. The database design might be modified as follows:

```
open_order (order_numb, customer_numb, order_date)

open_order_item (order_numb, item_numb, quantity, shipped?)

filled_order (order_numb, customer_numb, order_date)

filled_order_item (order_numb, item_numb, quantity, shipped?)
```

Whenever all items in an open order have shipped, an application program deletes rows from the *open order* and *open order item* table and inserts them into the *filled order* and f*illed order item* tables. The *open order* and *open order item* tables remain relatively small, speeding up both retrieval and modification performance. Although retrieval from *filled order* and *filled order line* will be slower, *Antique Opticals* uses those tables much less frequently.

The drawback to this solution occurs when *Antique Opticals* needs to access all of the orders and/or order items, at the same time. A query whose result table includes data from both sets of open and filled tables must actually be two queries connected by the union operator. (Remember that the union operation creates one table by merging the rows of two tables with the same structure.) Performance of such a query will be worse than that of a query of either set of tables individually. Nonetheless, if an analysis of *Antique Opticals*' data access patterns reveals that such queries occur rarely and that most retrieval involves the set of tables for open orders, then the horizontal partitioning is worth doing.

The only way you can determine whether horizontal partitioning will increase performance is to examine the ways in which your database applications access data. If there is a group of rows that are accessed together significantly more frequently than the rest of the rows in a table, then horizontal partitioning may make sense.

Vertical Partitioning

Vertical partitioning involves creating two or more tables with selected columns and all rows of a table. For example, if *Antique Opticals* accesses the titles and prices of their merchandise items more frequently than the other columns in the *item* table, the *item* table might be partitioned as follows:

```
item_title (item_numb, title, price)

item_detail (item_numb, distributor, release_date, …)
```

The benefit of this design is that the rows in the smaller *item title* table will be physically closer together; the smaller table will take up fewer disk pages and thus support faster retrieval.

Queries that require data from both tables must join the tables over the item number. Like most joins, this will be a relatively slow operation. Therefore, vertical partitioning makes sense only when there is a highly skewed access pattern of the columns of a table. The more often a small, specific group of columns is accessed together, the more vertical partitioning will help.

For Further Reading

Alapati S, Kuhn D, Padfield B. *Oracle database 12c performance tuning recipes: a problem-solution approach*. Apress; 2013.

Alapati S, Kuhn D, Padfield B. *Oracle database 12c performance tuning recipes: a problem-solution approach*. Apress; 2013.
Andrews T. *DB2 SQL tuning tips for z/OS developers*. IBM Press; 2012.
Atonini C. *Troubleshooting oracle performance*. Apress; 2008.
Gulutzan P, Pelzer T. *SQL performance tuning*. Addison-Wesley Professional; 2002.
IBM Redbooks *A deep blue view of DB2 performance*. IBM.com/Redbooks; 2006.
Meade K. *Oracle SQL performance tuning and optimization: it's all about the cardinalities*. CreateSpace Independent Publishing Platform; 2014.
Mittra SS. *Database performance tuning and optimization*. Springer; 2002.
Nevarez B. *Microsoft SQL server 2014 query tuning & optimization*. McGraw-Hill Education; 2014.
Shasha D, Bonnet P. *Database tuning: principles, experiments, and troubleshooting techniques*. Morgan Kaufmann; 2002.
Schwartz B, Zaitsev P, Tkachenko V, Zawony JD, Lentz A, Balling DJ. *High performance MySQL: optimization, backups replication, and more*. O'Reilly; 2008.
Tow D. *SQL tuning*. O'Reilly; 2003.
Winand M. *SQL performance explained: everything developers need to know about SQL performance*. Markus Winand; 2012.