Chapter 1: Tuning the Database

In This Chapter

- ✔ **Examining the work to be done**
- ✔ **Contemplating physical design considerations**
- ✔ **Choosing and clustering indexes**
- ✔ **Co-clustering two relations**
- ✔ **Indexing on multiple attributes**
- ✔ **Tuning indexes, queries, and transactions**
- ✔ **Query tuning in a high-concurrency environment**
- ✔ **Benchmarking**
- ✔ **Keeping user interactions separate from transactions**
- ✔ **Holding down traffic between application and server**
- ✔ **Creating a precompiled list of frequently used queries**

The word *tuning* is generally taken to mean optimizing an existing system that isn't operating at top capacity. Tuning doesn't do you much good, however, if your initial design isn't at least close to optimal in the first place. Tuning can take you only so far from your starting point. It's a lot easier to tune a slightly off-pitch B string on your guitar to a perfect B than it is to tune a G string up to a perfect B. (Also, you're a lot less likely to break the string.) Tuning for optimal performance should start in the initial design stage of a database, not at some later time when design decisions have been cast in concrete.

The performance of a database management system (DBMS) is generally judged by how fast it executes queries. Two types of operations are important: the retrieval of data from a database and the updating of records in a database. The speed at which records can be accessed is key to both types of operations, because you must locate a record before you can retrieve or update the data in it. The users' data model on which you'll base your database design is almost certainly structured in a way that isn't the best from a performance standpoint. The users are primarily concerned with functionality and may have little or no idea of how the design of a database affects how well it performs. You must transform the users' data into a conceptual schema that you actualize in the form of an Entity-Relationship (ER) model diagram. Recall that the Entity-Relationship data model and its associated diagrams are extensively covered in Book II.

Analyzing the Workload

Optimal design of a database depends largely on how the database will be used. What kinds of queries will it be subjected to? How often will updates be made, compared with how often queries are posed? These kinds of questions try to get at what the workload will be. The answers to such questions have great bearing on how the database should be structured. In effect, the design of the database is tuned based on how it will typically be used.

To give you a sound foundation for designing your database to best handle the workload to which it will be subjected, draft a workload description. The workload description should include the following elements:

- A list of all the queries you expect to be run against the database, along with an estimate of the expected frequency of each query compared with the frequencies of all the other queries and update operations
- A list of all the update operations you expect to perform, along with an estimate of the expected frequency of each operation compared with the frequencies of all the other updates and queries
- Your goal for the performance of each type of query and update

Queries can vary tremendously in complexity, so it's important to determine in advance how complex each query is and how that complexity will affect the overall workload. You can determine query complexity by answering a few questions:

- How many relations (tables) are accessed by this query?
- Which attributes (columns) are selected?
- Which attributes appear in the `WHERE` clause, and how selective are the `WHERE` clause conditions likely to be?

Just as queries can vary a great deal, so can update operations. Questions regarding updates should include the following:

- Which attributes appear in the `WHERE` clause, and how selective are the `WHERE` clause conditions likely to be?
- What type of update is it: `INSERT`, `DELETE`, or `UPDATE`?
- In `UPDATE` statements, which fields will be modified?

Considering the Physical Design

Among the factors that have a major impact on performance, few, if any, have a greater effect than indexes. On the plus side, indexes point directly to the desired record in a table, thereby bypassing the need to scan down through the table until you come upon the record you want. This feature can be a tremendous time-saver for a query. On the minus side, every time an insertion update or a deletion update is made to a table, the indexes on that table must be updated too, costing time.

When chosen properly, indexes can be a great help. When chosen poorly, indexes can waste resources and slow processing substantially.

Regarding indexes, you need to answer several questions:

- Which tables should have indexes, and which should not?
- For the tables that should have indexes, which columns should be indexed?
- For each index, should it be clustered or unclustered? Recall that a table can have only one clustered index, and that it will give the greatest performance boost. The column that is used most often as a retrieval key should be the one with a clustered index. Other columns used as retrieval keys less frequently would get unclustered indexes.

I address all these questions in this chapter.

After you arrive at a conceptual schema and determine that you need to make changes to improve performance, what kinds of modifications can you make? For one thing, you could change the way you divide up your data among the tables in your design. For another, you could alter the level of normalization of your tables.

- Often, you have more than one way to normalize a schema, and one such way may deliver better performance than others. You may want to change the way tables are defined to take advantage of a schema that gives you better performance than your current schema does.
- Although this method may sound somewhat heretical, sometimes it pays to denormalize your schema and accept a risk of modification anomalies in exchange for a significant performance boost.
- Contrary to the preceding point, sometimes it makes sense to take normalization a step further than you otherwise would — in effect, to overnormalize. This method can improve the performance of queries that involve only a few attributes. When you give those attributes a table of their own, sometimes you can speed retrievals.

You should carefully examine queries and updates that are run frequently to see whether rewriting them would enable them to execute faster. There's probably not much advantage to applying such scrutiny to queries that are rarely run, but after you have some history and notice the ones that are being run continually, it may pay to give those queries an extra look to see whether they can be improved.

Choosing the Right Indexes

Indexes can improve the performance of database retrievals dramatically, for several reasons. One reason is that an index tends to be small compared with the table that it's indexing. This fact means that the index is likely to be in the cache, which is accessible at semiconductor-memory speed rather than on disk — a million-to-one performance advantage right there. Other reasons depend on the type of query being performed and on whether the index is clustered. I discuss clustering in the next section.

Avoiding unnecessary indexes

Because maintaining indexes carries an overhead cost, you don't want to create any indexes that won't improve the performance of any of your retrieval or update queries. To decide which database tables shouldn't be indexed, consult the workload description you created as the first step in the design process (refer to "Analyzing the Workload," earlier in this chapter). This description contains a list of queries and their frequencies.

Here's a no-brainer: If a table has only a small number of rows, there's no point in indexing it. A sequential scan through relatively few rows executes quickly.

For larger tables, the best candidates for indexes are columns that appear in the query's `WHERE` clause. The `WHERE` clause determines which table rows are to be selected.

It's likely — particularly in a system in which a large number of different queries are run — that some queries are more important than others. Those queries are run more often, or they're run against more and larger tables, or getting results quickly is critical for some reason. Whatever the case, prioritize your queries, with the most important coming first. For the most important query, create indexes that give the best performance. Then move down the line, adding indexes that help the progressively less-important queries. Your DBMS's query optimizer chooses the best execution plan available to it based on the indexes that are present.

Different kinds of indexes exist, each with its own structure. One kind of index is better for some retrievals; another kind is better for others. The most common index types are B+ tree, hash, and ISAM (see "Choosing an index type," later in this chapter). Theoretically, for any given query, the query optimizer chooses the best index type available. Most of the time, practice follows theory.

Choosing a column to index

Any column that appears in a query's `WHERE` clause is a candidate for indexing. If the `WHERE` clause contains an exact-match selection, such as `EMPLOYEE.DepartmentID = DEPARTMENT.DepartmentID`, a hash index on EMPLOYEE.DepartmentID usually performs best. The number of rows in the EMPLOYEE table is sure to be larger than the number of rows in the DEPARTMENT table, so the index is of more use applied to EMPLOYEE than it is applied to DEPARTMENT.

A hash index stores pairs of keys and values based on a pseudo-randomizing function called a hash function.

If the WHERE clause contains a range selection, such as EMPLOYEE.Age BETWEEN 55 AND 65, a B+ tree index on EMPLOYEE.Age will probably be the best performer. (A B+ tree is a balanced tree data structure whose leaves contain a sequence of key/pointer pairs.) If the table is rarely updated, an ISAM index may be competitive with the B+ tree index.

ISAM indexes are small and can be searched quickly. However, if insertions or deletions are frequent, a table with ISAM indexing can quickly lose its efficiency advantage.

Using multicolumn indexes

If a WHERE clause imposes conditions on more than one attribute, such as EMPLOYEE.Age BETWEEN 55 AND 65 AND EMPLOYEE.DeptName = Shipping, you should consider using a multicolumn index. If the index includes all the columns that the query retrieves (an index-only query), the query could be completed without touching the data table at all. This method could speed the query dramatically and may be sufficient motivation to include in the index a column that you otherwise wouldn't include.

Clustering indexes

A *clustered index* is one that determines the sort order of the table that it's indexing, as opposed to an *unclustered index, which* has no relationship to the sort order of the table.

Suppose that several queries of the EMPLOYEE table have a WHERE clause similar to WHERE EMPLOYEE.LastName = 'Smith'. In such a case, it would be beneficial to have a clustered index on EMPLOYEE.LastName. All the employees named Smith would be clustered in the index, and they'd be retrieved very quickly. Quick retrieval is possible because after you've found the index to the first Smith, you've found them all. Access to the desired records is almost instantaneous.

Any given table can have only one clustered index. All other indexes on that table must be unclustered. Unclustered indexes can be helpful, but not as helpful as clustered indexes. For that reason, if you're going to choose one index to be the clustered index for a table, choose the one that will be used by the most important queries in the list of queries in the workload description (refer to "Analyzing the Workload," earlier in this chapter).

Consider the following example:

```
SELECT DeptNo
FROM EMPLOYEE
WHERE EMPLOYEE.Age > 29 ;
```

You can use a B+ tree index on Age to retrieve only the rows in which employee age is greater than 29. Whether this method is worthwhile depends on the age distribution of the employees. If most employees are 30 or older, the indexed retrieval won't do much better than a sequential scan.

Suppose that only 10 percent of the employees are more than 29 years old. If the index on Age is clustered, you gain a substantial improvement over a sequential scan. If the index is unclustered, however — as it's likely to be — it could require a buffer-page swap for every qualifying employee and will likely be more expensive than a sequential scan. I say that an index on Age is likely to be unclustered based on the assumption that at least one column in the EMPLOYEE table is more deserving of a clustered index than the Age column.

You can see from this example that choosing whether to create an index for a table column isn't a simple matter. Doing an effective job of choosing requires detailed knowledge of the data as well as of the queries that are run on it.

Figure 1-1 compares the costs of using a clustered index, an unclustered index, and a sequential scan to retrieve rows from a table.

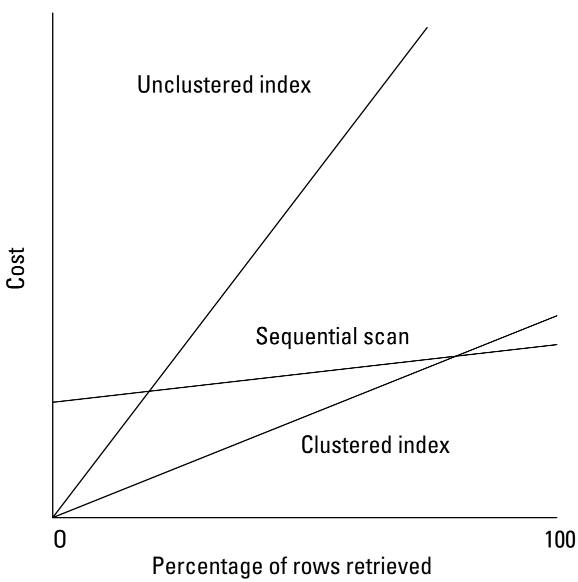

Figure 1-1: The cost of retrievals with and without an index.

Figure 1-1 reveals a few things about the cost of indexes:

- A clustered index always performs better than an unclustered index.
- A clustered index performs better than a sequential scan unless practically all the rows are retrieved.
- When one record is being retrieved, or a very few records are being retrieved, a clustered index performs much better than a sequential scan.
- When one record is being retrieved, or a very few records are being retrieved, an unclustered index performs better than a sequential scan.
- When more than about 10 percent of the records in a table are retrieved, a sequential scan performs better than an unclustered index.

That last point disproves the myth that indexing a table column that is used as a retrieval key always improves performance compared with the performance of a sequential scan.

Choosing an index type

In most cases, a B+ tree index is preferred because it does a good job on range queries as well as equality queries. Hash indexes are slightly better than B+ tree indexes in equality queries but not nearly as good in range queries, so overall, B+ tree indexes are preferred.

In some cases where a retrieval is made of data contained in multiple tables, however, a hash index will do better. One such case involves a nested loop join, in which the inner table is the indexed table and the index includes the join columns. (This situation is called a *hash join*.) Because an equality selection is generated for each row in the outer table, the advantage of the hash index over the B+ tree index is multiplied. Another case in which the hash join comes out ahead is when there is an important equality query and there are no range queries on a table.

TIP

You don't need to lose a lot of sleep over choosing an index type. Most database engines make the choice for you, and that choice usually is the best one.

Weighing the cost of index maintenance

Indexes slow update operations because every time a table is updated with an insertion or a deletion, all its indexes must be updated as well. Balance this situation against the speed gained by accessing table rows faster than would be possible with a sequential table scan. Even updates are potentially speeded because a row must be located before it can be updated. Nevertheless, you may find that the net benefit of some indexes doesn't justify their inclusion in the database, and you're better off dropping them.

If you suspect that an index might be doing you more harm than good, run some test queries with the index both present and absent. Use the results to guide your decision.

Using composite indexes

Composite indexes are indexes on more than one column. They can give superior performance to queries that have more than one condition in the WHERE clause. Here's an example:

```
SELECT EmployeeID
FROM EMPLOYEES
WHERE Age BETWEEN 55 AND 65
   AND Salary BETWEEN 4000 and 7000 ;
```

Both conditions in the WHERE clause are range conditions. An index based on <Age, Salary> performs about as well as an index based on <Salary, Age>. Either one performs better than an index based only on <Age> or only on <Salary>.

Now consider the following example:

```
SELECT EmployeeID
FROM EMPLOYEES
WHERE Age = 57
   AND Salary BETWEEN 4000 and 7000 ;
```

In this case, an index based on <Age, Salary> performs better than an index based on <Salary, Age> because the equality condition on <Age> means that all the records that have Age = 57 are clustered by the time the salary evaluation is done.

Tuning Indexes

After the database you've designed has been in operation for a while, you should reevaluate the decisions you made about indexing. When you created the system, you chose indexes based on what you expected usage to be. Now, after several weeks or months of operation, you have actual usage statistics. Perhaps some of the queries that you thought would be important aren't run very often after all. Perhaps you made assumptions about what indexes would be used by the query optimizer, but now you find that limitations of the optimizer prevent it from using those indexes, to the detriment of performance.

Based on the actual performance data that you have now, you can tune your indexes. This tuning may entail dropping indexes that are doing you no good and merely consuming resources, or it may mean adding new indexes to speed queries that turned out to be more important than they first appeared.

For best results, tuning indexes must be an ongoing activity. As time goes on, the nature of the workload is bound to evolve. As it does, the best indexes to support the current workload need to evolve, too. The database administrator must keep track of performance and respond when it starts to trend downward.

Another problem, which appears after a database has been in operation for an extended period of time, might be called the tired index. A *tired index* is one that no longer delivers the performance advantage that it did when it was first applied to the database. When an index is fresh and new — whether it's a B+ tree index, an ISAM index, or some other kind — it has an optimal structure. As time goes on, insertions, deletions, and updates are made to the table that the index is associated with, and the index must adjust to these changes. In the process of making those adjustments, the structure of the index changes and moves away from optimality. Eventually, performance is affected enough to be noticeable. The best solution to this problem is to drop the index and then rebuild it. The rebuilt index once again has an optimal structure.

The only downside to this solution is that the database table must be out of service while its index is being rebuilt. The amount of time it takes to rebuild an index depends on several things, including the speed of the processor and the size of the table being indexed. For some databases, you may not even experience any downside. The database engine will rebuild indexes automatically as needed.

Tuning Queries

After your system has been running for a while, you may find that a query is running slower than you expect. Several possible causes exist, and you have several ways to fix the problem. Because you generally have several ways to code a query, all producing the same result, perhaps you could recode it, along with an appropriate change of indexes.

Sometimes, a query doesn't run as you expect because the query optimizer isn't executing the plan that you expect it to. You can check on this situation in most DBMSes by having the optimizer display the plan that it generated. It's quite possible that the optimizer isn't finding the best plan. Here are some possible causes:

- Some query optimizers don't handle `NULL` values well. If the table you're querying contains `NULL` values in a field that appears in the `WHERE` clause, this situation could be the problem.
- Some query optimizers don't handle arithmetic or string expressions well. If one of these expressions appears in the `WHERE` clause, the optimizer may not handle it correctly.

- An `OR` connective in the `WHERE` clause could cause a problem.
- If you expect the optimizer to select a fast but sophisticated, plan, you could be disappointed. Sometimes, the best plan is beyond the capability of even high-end optimizers to find.

Some DBMSes give you some help in overcoming optimizer deficiencies. They enable you to force the optimizer to use an index that you know will be helpful or to join tables in the order that you know is best. For best results, a thorough knowledge of the capabilities and the deficiencies of your DBMS is essential, as is a good grasp of optimization principles.

Two possible culprits in performance problems are nested queries and correlated queries. Many optimizers don't handle these queries well. If a nested or correlated query isn't performing up to expectations, recoding it without nesting or correlation is a good thing to try.

Tuning Transactions

In an environment in which many users are using a database concurrently, contention for a popular resource can slow performance for everyone. The problem arises because a user locks a resource before using it and releases the lock when she is finished with it. As long as the resource is locked, no one else can access it.

Here are several things you can do to minimize the performance impact of locking:

- **Minimize the amount of time that you hold a lock.** If you're performing a series of operations with a transaction, obtain your locks as late as possible and release them as soon as possible.
- **Put indexes on a different disk from the one that holds the data files.** This practice prevents accesses to indexes from interfering with accesses to data.
- **Switch to a hash index.** If a table is updated frequently, B+ tree indexes on its columns lose much of their advantage, because the root of the tree and the pages just below it must be traversed by every update. They become hot spots, meaning that they're locked frequently, becoming bottlenecks. Making the switch to a hash index may help.

Separating User Interactions from Transactions

Because computer instructions operate in the nanosecond realm and humans operate in the second or even minute realm, one thing that can really slow a database transaction is any interaction with a human. If that transaction happens to hold a lock on a critical resource, the application with which the user is interacting isn't the only one to suffer a delay. Every other application that needs that resource is brought to a screeching halt for an interval of time that could be billions of times longer than necessary.

The obvious solution is to separate user interactions from transactions. Never hold a lock on anything while waiting for a human to do something.

Minimizing Traffic between Application and Server

If you have a lot of applications running on a lot of client machines, all depending on data that resides on a server, overall performance is limited by the server's capacity to send and receive messages. The fewer messages that need to travel between client and server, the better. The smaller the messages that need to travel between client and server, the better.

One approach to this problem is to use *stored procedures* — precompiled application modules that run on the server rather than on the client. Their primary purpose is to filter result sets rather than send a big chunk of the database, so that only the needed data is transmitted to the client. This method can reduce traffic between the server and client machines dramatically.

Precompiling Frequently Used Queries

If you execute the same query repeatedly — say, daily or even hourly — you can save time by compiling it in advance. At runtime, executing the query is the only thing that needs to be done. The compilation is done only once and never needs to be repeated. The time saving due to this forethought adds up and becomes significant over the course of weeks and months.