Database Fundamentals

You have just been assigned a project. You must create and maintain a list of all the movies produced by your employer—Orange Whip Studios.

What do you use to maintain this list? Your first thought might be to use a word processor. You could create the list, one movie per line, and manually insert each movie’s name so the list is alphabetical and usable. Your word processor provides you with sophisticated document-editing capabilities, so adding, removing, or updating movies is no more complicated than editing any other document.

Initially, you might think you have found the perfect solution—that is, until someone asks you to sort the list by release date and then alphabetically for each date. Now you must re-create the entire list, again sorting the movies manually and inserting them in the correct sequence. You end up with two lists to maintain. You must add new movies to both lists and possibly remove movies from both lists as well. You also discover that correcting mistakes or simply making changes to your list has become more complicated because you must make every change twice. Still, the list is manageable. You have only the two word-processed documents to be concerned with, and you can even open them both at the same time and make edits simultaneously.

Okay, the word processor is not the perfect solution, but it is still a manageable solution—that is, until someone else asks for the list sorted by director. As you fire up your word processor yet again, you review the entire list-management process in your mind. New movies must now be added to three lists. Likewise, any deletions must be made to all three lists. If a movie tag line changes, you must change just the multiple lists.

And then, just as you think you have the entire process worked out, your face pales and you freeze. What if someone else wants the list sorted by rating? And then, what if yet another department needs the list sorted in some other way? You panic, break out in a sweat, and tell yourself, “There must be a better way!”

This example is a bit extreme, but the truth is that a better way really does exist. You need to use a database.
Databases: A Definition

Let’s start with a definition. A database is simply a structured collection of similar data. The important words here are structured and similar, and the movie list is a perfect example of both.

Imagine the movie list as a two-dimensional grid or table, similar to that shown in Figure 2.1. Each horizontal row in the table contains information about a single movie. The rows are broken up by vertical columns. Each column contains a single part of the movie record. The Movie Title column contains movie titles, and so on.

The movie list contains similar data for all movies. Every movie record, or row, contains the same type of information. Each has a title, tag line, budget amount, and so on. The data is also structured in that the data can be broken into logical columns, or fields, that contain a single part of the movie record.

Here’s the rule of thumb: Any list of information that can be broken into similar records of structured fields should probably be maintained in a database. Product prices, phone directories, invoices, invoice line items, vacation schedules, and lists of actors and directors are all database candidates.

Where Are Databases Used?

You probably use databases all the time, often without knowing it. If you use a software-based accounting program, you are using a database. All accounts payable, accounts receivable, vendor, and customer information is stored in databases. Scheduling programs use databases to store appointments and to-do lists. Even email programs use databases for directory lists and folders.

These databases are designed to be hidden from you, the end user. You never add accounts receivable invoice records into a database yourself. Rather, you enter information into your accounting program, and it adds records to the database.

Clarification of Database-Related Terms

Now that you understand what a database is, I must clarify some important database terms for you. In the SQL world (you learn about SQL in depth in Chapter 5, “Introduction to SQL”), this collection of data is called a table. The individual records in a table are called rows, and the fields that make up the rows are called columns. A collection of tables is called a database.
Picture a filing cabinet. The cabinet houses drawers, each of which contains groups of data. The cabinet is a means of keeping related but dissimilar information in one place. Each cabinet drawer contains a set of records. One drawer might contain employee records, whereas another drawer might contain sales records. The individual records within each drawer are different, but they all contain the same type of data, in fields.

The filing cabinet shown in Figure 2.2 is the database—a collection of drawers or tables containing related but dissimilar information. Each drawer contains one or more records, or rows, made up of different fields, or columns.

![Figure 2.2](image)

**Data Types**

Each row in a database table is made up of one or more columns. Each column contains a single piece of data, part of the complete record stored in the row. When a table is created, each of its columns needs to be defined. Defining columns involves specifying the column’s name, size, and data type. The data type specifies what data can be stored in a column.

Data types specify the characteristics of a column and instruct the database as to what kind of data can be entered into it. Some data types allow the entry of free-form alphanumeric data. Others restrict data entry to specific data, such as numbers, dates, or true or false flags. A list of common data types is shown in Table 2.1.

<table>
<thead>
<tr>
<th>DATA TYPE</th>
<th>RESTRICTIONS</th>
<th>TYPICAL USE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>Upper and lowercase text, numbers, symbols</td>
<td>Names, addresses, descriptions</td>
</tr>
<tr>
<td>Numeric</td>
<td>Positive and negative numbers, decimal points</td>
<td>Quantities, numbers</td>
</tr>
<tr>
<td>Date</td>
<td>Dates, times</td>
<td>Dates, times</td>
</tr>
<tr>
<td>Money</td>
<td>Positive and negative numbers, decimal points</td>
<td>Prices, billing amounts, invoice line items</td>
</tr>
<tr>
<td>Boolean</td>
<td>Yes and No or True and False</td>
<td>On/off flags, switches</td>
</tr>
<tr>
<td>Binary</td>
<td>Non-text data</td>
<td>Pictures, sound, and video data</td>
</tr>
</tbody>
</table>
Most database applications provide a graphic interface to database creation, enabling you to select data types from a list. Microsoft Access uses a drop-down list box, as shown in Figure 2.3, and provides a description of each data type.

You use data types for several reasons, instead of just entering all data into simple text fields. One of the main reasons is to control or restrict the data a user can enter into that field. A field that has to contain a person’s age, for example, could be specified as a numeric field. This way, the user cannot enter characters into it—only the digits 0–9 would be allowed. This restriction helps ensure that no invalid data is entered into your database.

Various data types are also used to control how data is sorted. Data entered in a text field is sorted one character at a time, as if it were left-justified. The digit 0 comes before 1, which comes before 9, which comes before a, and so on. Because each character is evaluated individually, a text field containing the number 10 is listed after 1 but before 2 because 10 is greater than 1 but less than 2, just as a 0 is greater than a but less than b. If the value being stored in this column is a person’s age, correctly sorting the table by that column would be impossible. Data entered into a numeric field, however, is evaluated by looking at the complete value rather than a character at a time; 10 is considered greater than 2 instead of less than 2. Figure 2.4 shows how data is sorted if numbers are entered into a text field.

The same is true for date fields. Dates in these fields are evaluated one character at a time, from left to right. The date 02/05/01 is considered less than the date 10/12/99 because the first character of the date 02/05/01—the digit 0—is less than the first character of the date 10/12/99—the digit 1. If the same data is entered in a date field, the database evaluates the date as a complete entity and therefore sorts the dates correctly.
The final reason for using various data types is the storage space that plain-text fields take up. A text field that is large enough to accommodate up to 10 characters takes up 10 bytes of storage. Even if only 2 characters are entered into the field, 10 bytes are still stored. The extra space is reserved for possible future updates to that field. Some types of data can be stored more efficiently when not treated as text. For example, a 4-byte numeric field can store numeric values from 0 to over 4,000,000,000! Storing 4,000,000,000 in a text field requires 10 bytes of storage. Similarly, a 4-byte date/time field can store the date and time with accuracy to the minute. Storing that same information in a text field would take a minimum of 14 bytes or as many as 20 bytes, depending on how the data is formatted.

TIP
In addition to all that has already been said about picking the appropriate data types, it is also important to note that picking the wrong type can have a significant impact on performance.

NOTE
Different database applications use different terms to describe the same data type. For example, Microsoft Access uses the term text to describe a data type that allows the entry of all alphanumeric data. Microsoft SQL Server calls this same data type char and uses text to describe variable-length text fields. After you determine the type of data you want a column to contain, refer to your database application’s manuals to ensure that you use the correct term when making data type selections.

When you’re designing a database, you should give careful consideration to data types. You usually cannot easily change the type of a field after the table is created. If you do have to change the type, you might have to create a new table and write routines to convert the data from one table to the new one.

Planning the size of fields is equally important. With most databases, you can’t change the size of a field after the table is created. Getting the size right the first time and allowing some room for growth can save you much aggravation later.

CAUTION
When you’re determining the size of data fields, always try to anticipate future growth. If you’re defining a field for phone numbers, for example, realize that not all phone numbers follow the three-digit area code plus seven-digit phone number convention used in the United States and Canada. Paris, France, for example, has eight-digit phone numbers, and area codes in small towns in England can contain four or five digits.
CHAPTER 2 Building the Databases

Using a Database

Back to the example. At this point, you have determined that a database will make your job easier and might even help preserve your sanity. You create a table with columns for movie title, tag line (or pitch text), release date, and the rest of the required data. You enter your movie list into the table, one row at a time, and are careful to put the correct data in each column.

Next, you instruct the database application to sort the list by movie title. The list is sorted in a second or less, and you print it out. Impressed, you try additional sorts—by rating and by budgeted amount. The results of these sorts are shown in Figures 2.5, 2.6, and 2.7.

Figure 2.5
Data entered once in a Microsoft Access table can be sorted any way you want.

Figure 2.6
Data sorted by rating.

Figure 2.7
Data sorted by budgeted amount.
You now have two or more lists, but you had to enter the information only once; because you were careful to break the records into multiple columns, you can sort or search the list in any way necessary. You just need to reprint the lists whenever your records are added, edited, or deleted. And the new or changed data is automatically sorted for you.

“Yes,” you think to yourself, “this really is a better way.”

A Database Primer

You have just seen a practical use for a database. The movie list is a simple database that involves a single table and a small set of columns. Most well designed database applications require many tables and ways to link them. You’ll revisit the movie list when relational databases are introduced.

Your first table was a hit. You have been able to accommodate any list request, sorted any way anyone could need. But just as you are beginning to wonder what you’re going to do with all your newfound spare time, your boss informs you that he’ll need reports sorted by the director name.

“No problem,” you say. You open your database application and modify your table. You add two new columns, one for the director’s first name and one for the last name. Now, every movie record can contain the name of the director, and you even create a report of all movies including director information. Once again, you and your database have saved the day, and all is well.

Or so you think. Just when things are looking good, you receive a memo asking you to include movie expenses in your database so as to be able to run reports containing this information.

You think for a few moments and come up with two solutions to this new problem. The first solution is simply to add lots more columns to the table, three for each expenses item (date, description, and amount).
This, you realize, is not a long-term solution at all. How many expenses should you allow space for? Every movie can, and likely will, have a different set of expenses, and you have no way of knowing how many expenses you should accommodate for. Inevitably, whatever number you pick will not be enough at some point. In addition, adding all these extra columns, which will not be used by most records, is a tremendous waste of disk space. Furthermore, data manipulation becomes extremely complicated if data is stored in more than one column. If you need to search for specific expenses, you’d have to search multiple columns. This situation greatly increases the chance of incorrect results. It also makes sorting data impossible because databases sort data one column at a time, and you have data that must be sorted together spread over multiple columns.

**NOTE**

An important rule in database design is that if columns are seldom used by most rows, they probably don’t belong in the table.

Your second solution is to create additional rows in the table, one for each expense for each movie. With this solution, you can add as many expenses as necessary without creating extra columns. This solution, though, is not workable. Although it does indeed solve the problem of handling more than a predetermined number of expenses, doing so introduces a far greater problem. Adding additional rows requires repeating the basic movie information—things such as title and tag line—over and over, for each new row.

Not only does reentering this information waste storage space, it also greatly increases the likelihood of your being faced with conflicting data. If a movie title changes, for example, you must be sure to change every row that contains that movie’s data. Failing to update all rows would result in queries and searches returning conflicting results. If you do a search for a movie and find two rows, both of which have different ratings, how would you know which is correct?

This problem is probably not overly serious if the conflicting data is the spelling of a name—but imagine that the data is customer-billing information. If you reenter a customer’s address with each order and then the customer moves, you could end up shipping orders to an incorrect address.

You should avoid maintaining multiple live copies of the same data whenever possible.

**NOTE**

Another important rule in database design is that data should never be repeated unnecessarily. As you multiply the number of copies you have of the same data, the chance of data-entry errors also multiplies.

**TIP**

One point worth mentioning here is that the “never duplicate data” rule does not apply to backups of your data. Backing up data is incredibly important, and you can never have too many backup plans. The rule of never duplicating data applies only to live data—data to be used in a production environment on an ongoing basis.

And while you are thinking about it, you realize that even your earlier solution for including director names is dangerous. After all, what will happen when a movie has two directors? You’ve allocated room for only one name.
Understanding Relational Databases

The solution to your problem is to break the movie list into multiple tables. Let's start with the movie expenses.

The first table, the movie list, remains just that—a movie list. To link movies to other records, you add one new column to the list, a column containing a unique identifier for each movie. It might be an assigned movie number or a sequential value that is incremented as each new movie is added to the list. The important thing is that no two movies have the same ID.

TIP
Never reusing record-unique identifiers is generally a good idea. If the movie with ID number 105 is deleted, for example, that number should never be reassigned to a new movie. This policy guarantees that there is no chance of the new movie record getting linked to data that belonged to the old movie.

Next, you create a new table with several columns: movie ID, expense date, expense description, and expense amount. As long as a movie has no associated expenses, the second table—the expenses table—remains empty. When an expense is incurred, a row is added to the expenses table. The row contains the movie that uniquely identifies this specific movie and the expense information.

The point here is that no movie information is stored in the expenses table except for that movie ID, which is the same movie ID assigned in the movie list table. How do you know which movie the record is referring to when expenses are reported? The movie information is retrieved from the movie list table. When displaying rows from the expenses table, the database relates the row back to the movie list table and grabs the movie information from there. This relationship is shown later in this chapter, in Figure 2.8.

This database design is called a relational database. With it you can store data in various tables and then define links, or relationships, to find associated data stored in other tables in the database. In this example, a movie with two expenses would have two rows in the expenses table. Both of these rows contain the same movie ID, and therefore both refer to the same movie record in the movie table.

NOTE
The process of breaking up data into multiple tables to ensure that data is never duplicated is called normalization.

Primary and Foreign Keys

Primary key is the database term for the column(s) that contains values that uniquely identify each row. A primary key is usually a single column, but doesn’t have to be.

There are only two requirements for primary keys:

- Every row must have a value in the primary key—Empty fields, sometimes called null fields, are not allowed.
- Primary key values can never be duplicated—If two movies were to have the same ID, all relationships would fail. In fact, most database applications prevent you from entering duplicate values in primary key fields.
When you are asked for a list of all expenses sorted by movie, you can instruct the database to build the relationship and retrieve the required data. The movie table is scanned in alphabetical order, and as each movie is retrieved, the database application checks the expenses table for any rows that have a movie ID matching the current primary key. You can even instruct the database to ignore the movies that have no associated expenses and retrieve only those that have related rows in the expenses table.

**TIP**

Many database applications support a feature that can be used to auto-generate primary key values. Microsoft Access refers to this as an Auto Number field, SQL Server uses the term Identity, and other databases use other terms (for essentially the same thing). Using this feature, a correct and safe primary key is automatically generated every time a new row is added to the table.

**NOTE**

Not all data types can be used as primary keys. You cannot use columns with data types for storing binary data, such as sounds, images, variable-length records, or OLE links, as primary keys.

The movie ID column in the expenses table is not a primary key. The values in that column are not unique if any movie has more than one expense listed. All records of a specific movie’s expenses contain the same movie ID. The movie ID is a primary key in a different table—the movie table. This is a foreign key. A foreign key is a nonunique key whose values are contained within a primary key in another table.

To see how the foreign key is used, assume that you have been asked to run a report to see which movies incurred expenses on a specific date. To do so, you instruct the database application to scan the expenses table for all rows with expenses listed on that date. The database application uses the value in the expenses table’s movie ID foreign key field to find the name of the movie; it does so by using the movie table’s primary key. This relationship is shown in Figure 2.8.

The relational database model helps overcome scalability problems. A database that can handle an ever-increasing amount of data without having to be redesigned is said to scale well. You should always take scalability into consideration when designing databases.
Now you’ve made a significant change to your original database, but what you’ve created is a manageable and scalable solution. Your boss is happy once again, and your database management skills save the day.

**Different Kinds of Relationships**

The type of relationship discussed up to this point is called a *one-to-many* relationship. This kind of relationship allows an association between a single row in one table and multiple rows in another table. In the example, a single row in the movie list table can be associated with many rows in the expenses table. The one-to-many relationship is the most common type of relationship in a relational database.

Two other types of relational database relationships exist: the *one-to-one* relationship and the *many-to-many* relationship.

The one-to-one relationship allows a single row in one table to be associated with no more than one row in another table. This type of relationship is used infrequently. In practice, if you run into a situation in which a one-to-one relationship is called for, you should probably revisit the design. Most tables that are linked with one-to-one relationships can simply be combined into one large table.

The many-to-many relationship is also used infrequently. The many-to-many relationship allows one or more rows in one table to be associated with one or more rows in another table. This type of relationship is usually the result of bad design. Most many-to-many relationships can be more efficiently managed with multiple one-to-many relationships.

**Multitable Relationships**

Now that you understand relational databases, let’s look at the directors problem again. You will recall that the initial solution was to add the directors directly into the movie table, but that was not a viable solution because it would not allow for multiple directors in a single movie.

Actually, an even bigger problem exists with the suggested solution. As I said earlier, relational database design dictates that data never be repeated. If the director’s name was listed with the movie, any director who directed more than one movie would be listed more than once.

In other words, unlike the expenses—which are always associated with a single movie—directors can be associated with multiple movies, and movies can be associated with multiple directors. Two tables will not help here.

The solution to this type of relationship problem is to use three database tables:

- Movies are listed in their own table, and each movie has a unique ID.
- Directors are listed in their own table, and each director has a unique ID.
- A new third table is added, which relates the two previous tables.
For example, if movie number 105 was directed by director number 3, a single row would be added to the third table. It would contain two foreign keys, the primary keys of each of the movie and director tables. To find out who directed movie number 105, all you’d have to do is look at that third table for movie number 105 and you’d find that director 3 was the director. Then, you’d look at the directors table to find out who director 3 is.

That might sound overly complex for a simple mapping, but bear with me—this is all about to make a lot of sense.

If movie number 105 had a second director (perhaps director ID 5), all you would need to do is add a second row to that third table. This new row would also contain 105 in the movie ID column, but it would contain a different director ID in the director column. Now you can associate two, three, or more directors with each movie. Each director is associated with a movie by simply adding one more record to that third table.

And if you wanted to find all movies directed by a specific director, you could do that too. First, you’d find the ID of the director in the directors table. Then, you’d search that third table for all movie IDs associated with the director. And then you’d scan the movies table for the names of those movies.

This type of multitable relationship frequently is necessary in larger applications, and you’ll be using it later in this chapter. Figure 2.9 summarizes the relationships used.

To summarize, two tables are used if the rows in one table might be related to multiple rows in a second table and when rows in the second table are only related to single rows in the first table. If, however, rows in both tables might be related to multiple rows, three tables must be used.

**Indexes**

Database applications make extensive use of a table’s primary key whenever relationships are used. It is therefore vital that accessing a specific row by primary key value be a fast operation. When data is added to a table, you have no guarantee that the rows are stored in any specific order. A row with a higher primary key value could be stored before a row with a lower value. You should make no assumptions about the actual physical location of any rows within your table.
Now take another look at the relationship between the movie list table and the expenses table. You have the database scan the expenses table to learn which movies have incurred expenses on specific dates; only rows containing that date are selected. This operation, however, returns only the movie IDs—the foreign key values. To determine to which movies these rows are referring, you have the database check the movie list table. Specific rows are selected—the rows that have this movie ID as their primary key values.

To find a specific row by primary key value, you could have the database application sequentially read through the entire table. If the first row stored is the one needed, the sequential read is terminated. If not, the next row is read, and then the next, until the desired primary key value is retrieved.

This process might work for small sets of data. Sequentially scanning hundreds, or even thousands, of rows is a relatively fast operation, particularly for a fast computer with plenty of available system memory. As the number of rows increases, however, so does the time it takes to find a specific row.

The problem of finding specific data quickly in an unsorted list is not limited to databases. Suppose you’re reading a book on mammals and are looking for information on cats. You could start on the first page of the book and read everything, looking for the word cat. This approach might work if you have just a few pages to search through, but as the number of pages grows, so does the difficulty of locating specific words and the likelihood that you will make mistakes and miss references.

To solve this problem, books have indexes. An index allows rapid access to specific words or topics spread throughout the book. Although the words or topics referred to in the index are not in any sorted order, the index itself is. Cat is guaranteed to appear in the index somewhere after bison, but before cow. To find all references to cat, you would first search the index. Searching the index is a quick process because the list is sorted. You don’t have to read as far as dog if the word you’re looking for is cat. When you find cat in the index list, you also find the page numbers where cats are discussed.

Databases use indexes in much the same way. Database indexes serve the same purpose as book indexes—allowing rapid access to unsorted data. Just as book indexes list words or topics alphabetically to facilitate the rapid location of data, so do database table indexes list the values indexed in a sorted order. Just as book indexes list page numbers for each index listing, database table indexes list the physical location of the matching rows, as shown in Figure 2.10. After the database application knows the physical location of a specific row, it can retrieve that row without having to scan every row in the table.

However, two important differences exist between an index at the back of a book and an index to a database table. First, an index to a database table is dynamic. This means that every time a row is added to a table, the index is automatically modified to reflect this change. Likewise, if a row is updated or deleted, the index is updated to reflect this change. As a result, the index is always up-to-date and always useful. Second, unlike a book index, the table index is never explicitly browsed by the end user. Instead, when the database application is instructed to retrieve data, it uses the index to determine how to complete the request quickly and efficiently.
The index is maintained by the database application and is used only by the database application. You never actually see the index in your database, and in fact, most modern database applications hide the actual physical storage location of the index altogether.

When you create a primary key for a table, it is automatically indexed. The database assumes the primary key will be used constantly for lookups and relationships and therefore does you the favor of creating that first index automatically.

When you run a report against the expenses table to find particular entries, the following process occurs. First, the database application scans the expenses table to find any rows that match the desired date. This process returns the IDs of any matching expenses. Next, the database application retrieves the matching movie for each expense row it has retrieved. It searches the primary key index to find the matching movie record in the movie list table. The index contains all movie IDs in order and, for each ID, lists the physical location of the required row. After the database application finds the correct index value, it obtains a row location from the index and then jumps directly to that location in the table. Although this process might look involved on paper, it actually happens very quickly and in less time than any sequential search would take.

### Using Indexes

Now revisit your movies database. Movie production is up, and the number of movies in your movies table has grown, too. Lately, you’ve noticed that operations are taking longer than they used to. The alphabetical movie list report takes considerably longer to run, and the performance drops even further as more movies are added to the table. The database design was supposed to be a scalable solution, so why is the additional data bringing the system to its knees?

The solution here is the introduction of additional indexes. The database application automatically creates an index for the primary key. Any additional indexes have to be explicitly defined. To improve sorting and searching by rating, you just need an index on the rating column. With this index, the database application can instantly find the rows it is looking for without having to sequentially read through the entire table.

The maximum number of indexes a table can have varies from one database application to another. Some databases have no limit at all and allow every column to be indexed. That way, all searches or sorts can benefit from the faster response time.

**CAUTION**

Some database applications limit the number of indexes any table can have. Before you create dozens of indexes, check to see whether you should be aware of any limitations.
Before you run off and create indexes for every column in your table, you have to realize the trade-off. As explained earlier, unlike an index at the end of a book, a database table index is dynamic. As data changes, so do the indexes, and updating indexes takes time. The more indexes a table has, the longer write operations take. Furthermore, each index takes up additional storage space, so unnecessary indexes waste valuable disk space.

When, then, should you create an index? The answer is entirely up to you. Adding indexes to a table makes read operations faster and write operations slower. You have to decide the number of indexes to create and which columns to index for each application. Applications that are used primarily for data entry have less need for indexes. Applications that are used heavily for searching and reporting can definitely benefit from additional indexes.

For example, you should probably index the movie list table by rating because you often will be sorting and searching by movie rating. You will seldom need to sort by movie summary, so you don’t have any justification for indexing the summary column. You still can search or sort by summary if the need arises, but the search will take longer than a rating search. Likewise, the release date column might be a candidate for indexing. Whether you add indexes is up to you and your determination of how the application will be used.

**TIP**

With many database applications, you can create and drop indexes as needed. You might decide that you want to create additional temporary indexes before running a batch of infrequently used reports. They enable you to run your reports more quickly. You can drop the new indexes after you finish running the reports, which restores the table to its previous state. The only downside to doing so is that write operations are slower while the additional indexes are present. This slowdown might or might not be a problem; again, the decision is entirely up to you.

**Indexing on More than One Column**

Often, you might find yourself sorting data on more than one column; an example is indexing on last name plus first name. Your directors table might have more than one director with the same last name. To correctly display the names, you need to sort on last name plus first name. This way, Jack Smith always appears before Jane Smith, who always appears before John Smith.

Indexing on two columns—such as last name plus first name—is not the same as creating two separate indexes (one for last name and one for first name). You have not created an index for the first name column itself. The index is of use only when you’re searching or sorting the last name column, or both the last name and first name.

As with all indexes, indexing more than one column often can be beneficial, but this benefit comes with a cost. Indexes that span multiple columns take longer to maintain and take up more disk space. Here, too, you should be careful to create only indexes that are necessary and justifiable.

**Understanding the Various Types of Database Applications**

All the information described to this point applies equally to all databases. The basic fundamentals of databases, tables, keys, and indexes are supported by all database applications. At some point, however, databases start to differ. They can differ in price, performance, features, security, scalability, and more.