Chapter 9

J2ME Database Concepts
J2ME application saves and retrieves data using the Record Management System (RMS), which is discussed in the previous chapter. Applications running on both Connected Limited Device Configuration (CLDC) and Connected Device Configuration (CDC) devices use RMS for local data management. J2ME applications that run on CDC devices are also capable of utilizing a relational database management system (DBMS), which provides industrial-strength database management service to the application. The DBMS is typically located on a server connected to the device over a network, although some CDC devices might have the DBMS stored locally on a hard drive.

As you’ll recall from Chapter 1, CLDC devices usually have between 160KB and 512KB of available memory and are battery powered. They also use an inconsistent, small-bandwidth network wireless connection and may not have a user interface. CLDC devices use the KJava Virtual Machine (KVM) implementation, which is a stripped-down version of the JVM. CLDC devices include pagers, personal digital assistants, cell phones, dedicated terminals, and handheld consumer devices with between 128KB and 512KB of memory.

CDC devices use a 32-bit architecture, have at least 2MB of memory available, and implement a complete functional JVM. CDC devices include digital set-top boxes, home appliances, navigation systems, point-of-sale terminals, and smart phones.

A database is a collection of data managed by a DBMS. Many corporations use one of several commercially available DBMSs, such as Oracle, DB2, Sybase, and Microsoft Access (used for small data collections).

A CDC-based J2ME application interacts with commercial DBMSs by using a combination of Java data objects that are defined in the Java Database Connection (JDBC) specification and by using the Structured Query Language (SQL). The JDBC interface forms a communications link with a DBMS, while SQL is the language used to construct the message (called a query) that is sent to the DBMS to request, update, delete, and otherwise manipulate data in the DBMS.

This is the first of three chapters that show how to incorporate database interactions into a J2ME application. In this chapter you learn how to transform data elements of a business system into a relational database, which is a common task in development of J2ME applications. You can skip this chapter if you are familiar with this technique. The next chapter focuses on Java data objects—what they are, how they work, and how to use them to communicate with a DBMS. Chapter 11 shows how to write and execute queries, and then interact with the results returned by the DBMS.

Data

You are probably familiar with the term data because you use data in everyday life, such as when you dial a telephone number or log into a computer network using a user ID and password. The telephone number, user ID, and password are types of data. Yet the term data is commonly confused with the term information. Although these terms have a similar meaning in the vernacular, they are different when related to a DBMS. Information consists of one or more words that collectively infer a meaning, such as a person’s
address. Data refers to an atomic unit that is stored in a DBMS and is sometimes reassembled into information.

Examples of data are a person’s street address, city, state, and zip code. Each of these is an atomic unit that is commonly found in a DBMS. A J2ME application can access one or multiple atomic units as required by the application. Data is organized in a database so that a J2ME application can quickly find, retrieve, update, or delete one or more data elements.

**Databases**

As mentioned previously in this chapter, a database in the purest sense is a collection of data. While you can use Java and Java’s IO classes to create your own database, you’ll probably interact with a commercially available DBMS. DBMSs use proprietary and public domain algorithms to assure fast and secure interaction with data stored in the database. Most DBMSs adhere to a widely accepted relational database model. A database model is a description of how data is organized in a database. In a relational database model, data is grouped into tables using a technique called normalization, which you’ll learn about later in this chapter.

Once a database and at least one table are created, a J2ME application can send SQL statements to the DBMS to perform the following:

- Save data
- Retrieve data
- Update data
- Manipulate data
- Delete data

**Tables**

A table is the component of a database that contains data in the form of rows and columns, very similar to a spreadsheet. A row contains related data such as clients’ names and addresses. A column contains like data such as clients’ first names. Each column is identified by a unique name, called a column name, that describes the data contained in the column. For example, “client first name” is a likely choice as the name of a column that contains clients’ first names. In addition, columns are defined by attributes. An attribute describes the characteristic of data that can be stored in the column. Attributes include size, data type, and format. You’ll learn about these and other attributes later in this chapter.

Database name, table name, column name, column attributes, and other information that describe database components are known as metadata. Metadata is data about data. For example, the size of the client first name column describes the data contained within the column and therefore is referred to as metadata. Metadata is used by J2ME
applications to identify database components without needing to know details of a column, table, or the database. For example, a J2ME application can request from the DBMS the data type of a specific column. The column type is used by a J2ME application to copy data retrieved from the DBMS into a Java collection.

**Database Schema**

A *database schema* (see Figure 9-1) is a document that defines all components of a database, such as tables, columns, and indexes. A database schema also shows relationships between tables; the relationships are used to join rows of two tables. You’ll learn about indexes and relating tables later in this chapter.

---

**Figure 9-1.** A database schema diagrams the relationships among all components of a database.
To create a database schema, you must perform six steps:

1. Identify information used in the existing system or legacy system that is being replaced by the J2ME application.
2. Decompose this information into data.
3. Define data.
4. Normalize data into logical groups.
5. Create primary and foreign keys.
6. Group data together into logical groups.

**Identifying Information**

The initial step in defining a database schema is to identify all information used by the system that is being converted to J2ME technology. Information is associated with objects—also known as *entities*—of the system. An entity is an order form, a product, a customer, a sales representative, for example. Figure 9-2 illustrates the entities for an order system. Each entity is defined by attributes. An attribute is information that describes an entity, such as a customer name for a customer entity.

Don’t confuse an entity attribute with data attributes because an entity attribute can be different from data attributes. An entity attribute provides general information about an entity, while a data attribute provides information about data that is used by the entity. Data, as you’ll recall, is the atomic level of information. An attribute of an entity is at a more general level than the atomic level. For example, a customer name is an entity attribute, and a customer first name and customer last name are data attributes.
Identifying attributes is intuitive most times because an attribute is information commonly used to describe an entity. For example, a customer name and address are information normally used to describe a customer. Therefore, customer name and address are easily recognizable as attributes of a customer entity. Figure 9-3 contains attributes for entities in the order system. Notice how attributes of an entity uniquely identify the entity.

The best way to identify attributes of an entity is by analyzing instances of the entity. An entity is like an empty order form and an instance is an order form that contains order information. Looking at instances of an entity helps to identify attributes because

![Diagram of entities and attributes](image-url)
you are viewing a real entity. That is, instead of looking at a blank order form, you are
looking at an order form that represents a real order. You’ll find instances of an entity
in the existing system.

Once attributes are identified, you must describe the characteristics of each attribute
(see Figure 9-4). Here are common characteristics found in many attributes:

- **Attribute name** The name of the attribute uniquely distinguishes the attribute
  from other attributes of the same entity. “First name” is an attribute name.
  Duplicate attribute names within the same entity are prohibited. However, two
  entities can use the same attribute name. That is, the customer entity and the
  sales representative entity can both have an attribute called first name.

![Figure 9-4. Attributes are defined by their characteristics.](image-url)
Attribute type  An attribute type is nearly identical to the data type of a column in a table. Common attribute types include numeric, character, alphanumeric, date, time, Boolean, integer, float, and double, among other attribute types. However, unlike a data type for a column, you do not have to use precise terminology to describe an attribute type. That is, you might call an attribute type “sales in dollars,” which is not a valid Java data type but is sufficient to convey the type of information assigned to the entity. It is also advisable to include sample values that are assigned to the entity. This enables you to match the attribute with the most advantageous data type for the DBMS you are using when you create a table for the system.

Attribute size  The attribute size describes the number of characters used to store values of the attribute. This is similar to the size of a column in a table.

Attribute range  An attribute range contains minimum and maximum values that can be assigned to an attribute. For example, the value of the “total amount” attribute of an order entity is likely to be greater than zero and less than 10,000, assuming that no order has ever been received that had a total amount of more than 9,999. This range is then used to throw an error should an order be received with a total amount outside this range.

Attribute default value  An attribute default value is the value that is automatically assigned to the attribute if the attribute isn’t assigned a value by the J2ME application. For example, the J2ME application uses the default system date for the date of an order if a sales representative fails to date the order. The system date is the attribute default value.

Acceptable values  An acceptable value for an attribute is one of a set of values established by the business unit and includes zip codes, country codes, methods of delivery, and simply “yes” or “no.”

Required value  An attribute may require a value before the attribute is saved to a table. For example, an order entity has an order number attribute that must be assigned an order number.

Attribute format  The attribute format consists of the way an attribute appears in the existing system, such as the format of data.

Attribute source  The attribute source identifies the origin of the attribute value. Common sources are from data entry and J2ME applications (such as using the system date as the value of the attribute).

Comments  A comment is free-form text used to describe an attribute.

Decomposing Attributes to Data
Once attributes of entities are identified, they must be reduced to data elements. This process is called decomposing. For the most part, decomposing is an intuitive process because you can easily recognize whether an attribute is already at an atomic level, as
illustrated by the customer name and customer address attributes. Both attributes are not atomic, but the atomic level is obvious—first name, last name, city, state, and zip code.

However, decomposing other attributes might be less intuitive. For example, should a customer number attribute be decomposed? At first glance, the response is no because typically, a customer number is already atomic. Consider the following customer number that consists of three numbered segments: 12-24-1001. The first segment (12) represents the sales region where the customer is located. The second segment (24) is the branch in the sales region that handles the relationship with the customer. And the final segment (1001) is the number that identifies the customer within the branch and region.

Since the customer number is in three segments, should the customer number attribute be decomposed into each segment? This isn’t an easy question to answer since in some systems it makes sense to further decompose the customer number attribute, and in other systems no further decomposition of the customer number is necessary. The nature of the system will determine whether or not additional decomposition is required for an attribute. For example, in this scenario the customer number probably should be further decomposed if the system references each segment of the customer number, such as using the first segment to select customers of a particular region. Otherwise, the customer number should be treated as atomic.

**How to Decompose Attributes**

The process of decomposing attributes begins by analyzing the list of entities and their attributes. The list of attributes represents all the information used by the existing system. The objective is to reduce each attribute to a list of data that represents the atomic level of the attribute. Here’s how to do this:

1. Look at each attribute and ask yourself if the attribute is atomic.
2. If the attribute isn’t atomic, it must be further decomposed. Create a list of data derived from the attribute, as illustrated in Figure 9-5.
3. If the attribute is atomic, no further decomposition is necessary for that attribute.
4. Place the name of the attribute on the data list.
5. Review the data list developed in step 2 and repeat the decomposition process until all attributes are atomic.

**Tools for Analysis**

Computer-aided software engineering (CASE) tools are designed to automate the process of analyzing a system. Oracle Designer/2000, Rational Rose, and Together J are three of the better CASE tools on the market.

A CASE tool transforms basic information that is entered into the tool into entities, attributes, data, and relationships among entities. CASE tools are beyond the scope of this book. Visit the manufacturers’ web sites for more information about CASE tools.
Decomposing by Example

Let’s work through an example of decomposing attributes using the customer address attribute of a customer entity. The following are attributes for the customer entity:

- Customer number
- Customer name

![Diagram of data elements for the customer name attribute]
Chapter 9: J2ME Database Concepts

- Customer address
- Customer telephone number

Once the list of attributes is assembled, each attribute on the list must be decomposed. The customer address attribute is decomposed in this example. The same process can be applied to the other attributes.

Review the customer address attribute to determine whether the attribute is atomic, which it isn’t. Therefore, the customer address attribute must be decomposed into the following data elements:

- Street address 1
- Street address 2
- City
- State
- Country
- Country code
- Postal code
- Address type (home or business)

Defining Data

Decomposing attributes results in the identification of data elements used by the existing system. Each data element must be defined using techniques similar to those used to describe an attribute. Here are common ways to define a data element:

- **Data name**  The unique name given to the data element, which should reflect the kind of data (see “The Art of Choosing a Name”).
- **Data type**  A data type describes the kind of values associated with the data (see the next section, “Data Types”).
- **Data size**  The size of text data is the maximum number of characters required to represent values of the data. The size of numeric data is usually either the number of digits or the number of bytes for binary representation (for example, smallint in DB2 is 2 bytes).

Data Types

As previously mentioned in this chapter, a data type describes the characteristics of data associated with a data element. For example, a street address is likely to be an alphanumeric data type because a street address has a mixture of characters and numbers. It is important to use care when selecting the data type of a data element at this stage in the analysis because the data type that you choose typically becomes the data type of the column in the table that contains the data.
**The Art of Choosing a Name**

Picking a name for a data element might seem intuitive, but can easily become tricky when you realize the name must describe the data, it must be unique, and it may have size and character limitations, depending on the DBMS. The nature of the data provides a hint to the data name, such as “first name” being used as the name for data associated with a person’s first name. However, sometimes the obvious choice isn’t the best choice. For example, intuitively you might select “telephone number” as the name of data associated with a telephone number. But an entity such as a customer might have more than one telephone number, a home number and business number. This means the name of the data element that contains the telephone number must reflect the kind of telephone number too.

Another twist to selecting a name comes when more than one entity contains similar data. In this example, there could be a customer entity and a sales representative entity, both of which have home and business telephone numbers. The solution is to use the name of the entity as part of the data element name, such as “customer home telephone number” and “customer business telephone number,” since this clearly identifies the data.

Keep the length of the name reasonable. The name of a data element typically becomes a column name in a table. Some DBMSs restrict the length of column names and the types of characters that can be used as a column name. Names of data should have as few characters as possible to identify the data. There are two reasons for this. First, the data name will probably conform to any restrictions imposed by the DBMS. Second, a programmer won’t need to type long column names when interacting with the database.

A data name can be abbreviated using components of the name. For example, “customer home telephone number” can be shortened to “cust home phone.” When abbreviating data names, make sure the same style of abbreviation is used for naming data. This means that cust is the abbreviation for customer and should be used in other data of the customer entity (for example, cust bus phone).

Be aware that some DBMSs prohibit spaces in column names. You can avoid this problem by removing spaces in the data name. Doing so might make the name unreadable. Readability can be enhanced by capitalizing the first letter of each word in the name, or separating these words with hyphens or underlines as shown here:

- CustHomePhone
- cust-home-Phone
- cust_home_phone

Where possible, limit the choice of data types to those that are common to commercial DBMSs. Not all DBMSs use the same data types; some enhance the standard data type offering. However, if you are unsure of the data type to describe a data element, describe...
the type of data in your own words and include an example of data values that are associated with the data element. You can refine your choice to available data types in the DBMS when you create the table used to store the data element.

Many commercially available DBMSs have adopted a common set of data types based on the SQL set of data types, which you’ll learn about in detail in the next chapter. Some of these are listed here:

- **Character, also referred to as text**  Stores alphabetical characters and punctuation
- **Alpha**  Stores only alphabetical characters
- **Alphanumeric**  Stores alphabetical characters, punctuation, and numbers
- **Numeric**  Stores numbers only
- **Date/Time**  Stores dates and time values
- **Logical (Boolean)**  Stores one of two values: true or false, 0 or 1, or yes or no
- **LOB (large object)**  Stores large text fields, images, and other binary data

### Normalizing Data

Normalization is the process of organizing data elements into related groups to minimize redundant data and to assure data integrity. Redundant data elements occur naturally since multiple entities have the same data elements. For example, an order form and invoice are both entities that contain a customer name and address. Therefore, customer name and address are redundant.

For transactional databases, redundant data makes a database complex, inefficient, and exposes the database to problems referred to as *anomalies* when the DBMS maintains the database. Anomalies occur whenever new data is inserted into the database and when existing data is either modified or deleted, and can lead to a breach in referential integrity of the database (see “Referential Integrity” later in this chapter). However, for reporting data, the redundancy rules are bent a little because redundant data is more efficient. It minimizes the number of joins and allows data to be summarized into logical groups.

Errors caused by redundant data are greatly reduced and possibly eliminated by applying the normalization process to the list of data elements that describe all the entities in a system. This is called normalizing the logical data model of a system. The normalization process consists of applying a series of rules called *normal forms* to the list of data elements to:

- Remove redundant data elements.
- Reorganize data elements into groups.
- Define one data element of the group (called a *primary key*) to uniquely identify the group. Often, two or more data elements make up the primary key, which is referred to as a *composite key*.
Make other data elements of the group (called *non-key* data elements) functionally
dependent on the primary key.

Relate one group to another using the primary key.

For example, a customer number is the primary key of a group that contains customer
information. Other data contained in the group such as the customer first name and last
name are referred to as non-key data elements. Non-key data elements are functionally
dependent on the primary key. That is, a customer name, address, and related information
cannot exist in the customer group without being assigned a customer number.

### The Normalization Process

There are five normal forms. However, many industry leaders have concluded that the
fourth and fifth normal forms are difficult to implement and unnecessary. Therefore,
we’ll confine ourselves to the first three normal forms.

- First normal form (1NF) requires that information is atomic, as discussed
  previously in this chapter.
- Second normal form (2NF) requires data to be in the first normal form. In
  addition, data elements are organized into groups eliminating redundant
data. Each group contains a primary key and non-key data, and non-key
  data must be functionally dependent on a primary key.
- Third normal form (3NF) requires that data elements be in the second normal
  form, and non-key data must not contain transitive dependencies.

### Grouping Data

A common way to organize data elements into groups is to first assemble a list of all
data elements, as discussed previously in this chapter. When this is done, you’ll notice
that some data elements are duplicated because they are used by more than one entity.
Duplicate data elements must be removed from the list. Although this is an intuitive
process, you must be careful because not all data elements with similar sounding names
are duplicates.

Let’s say there are two data elements, zip code and postal code. At first glance
these appear to have the same meaning. A zip code is another term for postal code—
or is it? A zip code is a specific kind of postal code used in the United States. Postal
code is a general term that also applies to postal codes used by countries other than
the United States.

The difference is subtle, but could have an impact if you assume zip code and
postal code are the same. For example, many relational databases contain a table of
zip codes, where only valid zip codes are stored. The table also contains the city and
state that corresponds to a zip code. Postal code data may or may not be contained in
a relational database table since it could be difficult to identify all the postal codes from
every country and maintain this list as postal codes are modified.
Figure 9-6 contains a list of all data elements before redundant data elements are removed. Figure 9-7 shows the same list after redundant data is removed. Notice that the number of data elements on the list has been dramatically reduced. Figure 9-8 organizes related data into groups.
Creating Primary Keys

As discussed previously in this chapter, a primary key is a data element that uniquely identifies a row of data elements within a group. The data selected to become the primary key may or may not exist in the data list generated as the result of analyzing entities. Sometimes a data element, such as an order number, is used as the primary key. Other times, the DBMS can be requested to automatically generate a primary key whenever a column in the group isn’t suitable to be designated the primary key. Figure 9-8 indicates the primary key of each group with an asterisk.

Let’s use a customer entity as an illustration. In the purest sense, a customer has a name and address as attributes. These attributes decompose to first name, last name,
street, city, state, and zip code. However, none of these data elements are suited to become a primary key because individually and collectively none uniquely identify a customer.

Intuitively, the customer first name and last name seem to uniquely identify a customer, but upon closer analysis you’ll see that more than one customer might have the same first name and last name. Likewise, two people at the same address might have the same name, although it’s somewhat unlikely. If neither a single data element nor a combination of data elements uniquely identify a row, then you must create another data element to serve as the primary key of the table, which is what is required in the previous example. Alternatively, you can request the DBMS to generate a primary key automatically.
Nearly all commercial DBMSs can generate primary keys to make the database thread safe and reliable. In contrast, a J2ME application that generates a key must contain the logic to be sure that none of the components running on different servers accidentally generate the same key.

**Functional Dependency**

A functional dependency occurs when data depends on other data, such as when non-key data is dependent on a primary key. This means that all non-key data has a functional dependency on the primary key within its group.

For example, order product quantity in the orders group cannot exist unless there is an order number. The order number is the primary key of the group, and the order product quantity is non-key data. You can say that order product quantity is functionally dependent on order number. This is noted by the expression

\[
\text{order product quantity} \rightarrow \text{order number}
\]

**Transitive Dependencies**

A transitive dependency is a functional dependency between two or more non-key data elements. This is an elusive concept at first, but an example will clearly illustrate transitive dependency. The first grouping in Figure 9-9 shows the Order entity with the two data elements Sales Rep Number and Sales Region, which is the region to which the sales representative is assigned. Salesperson and region have a transitive dependency. The region is functionally dependent on the salesperson, and the salesperson is functionally dependent on the order number. Both salesperson and region are non-key data and are therefore functionally mutually dependent.

The problem lies with the fact that a salesperson cannot be relocated to a different region without having to modify the region data element in the order information group. Therefore, data elements must be regrouped to conform to the third normal form and eliminate transitive dependency.

The second grouping in Figure 9-9 illustrates the regrouping of the Order entity to address the transitive dependency problem. Notice that a new group is formed that contains the Sales Rep Number and the Sales Region. The Sales Rep Number and the Sales Region can be used as a composite key. A composite key, as you recall from an earlier discussion in this chapter, is a primary key that consists of two data elements. In this example, the Sales Rep Number and the Sales Region can be joined to form a composite key. If a sales representative is permitted to make sales in multiple regions, the composite key is used to associate the sales representative with one of many regions and a region to one of many sales representatives.

Identifying transitive dependencies is tricky. You have to carefully analyze the data elements once the list of data elements is in the second normal form to spot transitive dependencies.
Foreign Keys

As you recall, a foreign key is a primary key of another group used to draw a relationship between two groups of data elements (see ** in Figure 9-8). Relationships between two groups are made using the value of a foreign key and not necessarily the name of a foreign key.

Let’s say there are two groups, one contains customer information, and the other contains order information. The primary key in the customer information group is the customer number, and the primary key in the order information group is the order number. Each row in the order group contains the customer number of the customer who placed the order. The customer number in the order group is a foreign key. That is, the customer number in the order group is the primary key of the customer information group.

The DBMS is able to join information about a customer along with information about orders placed by that customer by joining together the customer number in both the customer information group and the order group.
Referential Integrity

The success of a relational database is based on the existence of primary keys and foreign keys of data groups to create relationships among groups. The existence of this relationship is called referential integrity and is illustrated in Figure 9-10.

Referential integrity is enforced by imposing database constraints. This means the DBMS assures referential integrity by preventing primary and foreign keys from being modified or deleted. Likewise, database constraints prevent new rows from being inserted without maintaining referential integrity.

![Diagram of Referential Integrity](image)

**Figure 9-10.** Groups of data elements are related to other groups of data elements by using a combination of foreign keys and primary keys.
The Art of Indexing

An index is used to quickly locate information in a table, similar to the way information is located in a book. However, instead of page numbers, an index references the row number in the table that contains the search criteria. Conceptually, an index is a table that has two columns. One column is the key to the index, and the other column is the number of the row in the corresponding table that contains the value of the key.

Let’s say that you create a table of customer information and use customer number as the primary key to the table. Figure 9-11 contains such a table. An index is created using the primary key as the index key. Notice that each row in the index corresponds to a row in the customer table. However, an index always has two columns regardless of the number of columns in the associated table. The number of columns plays a critical role in finding information quickly in a table because an index has less information for a DBMS to search than a table that contains all the data.

Indexed keys are sorted in alphabetical or numerical order depending on the value of the key. The DBMS begins at the center row of the index when searching for a particular key value. The search criteria is either an exact match to the key value of the center row of the index, is greater than the key value, or less than the key value.

If there is a match, the row that corresponds to the key is retrieved. If the value is greater than the key, the DBMS begins the next search at the center row of the lower

<table>
<thead>
<tr>
<th>Customer number</th>
<th>Row number</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345345</td>
<td>1</td>
</tr>
<tr>
<td>345684899</td>
<td>3</td>
</tr>
<tr>
<td>86766787</td>
<td>2</td>
</tr>
</tbody>
</table>

Table

<table>
<thead>
<tr>
<th>Row number</th>
<th>Customer number</th>
<th>Customer first name</th>
<th>Customer last name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12345345</td>
<td>Bob</td>
<td>Smith</td>
</tr>
<tr>
<td>2</td>
<td>86766787</td>
<td>Mary</td>
<td>Jones</td>
</tr>
<tr>
<td>3</td>
<td>345684899</td>
<td>Tim</td>
<td>Jones</td>
</tr>
</tbody>
</table>

Figure 9-11. The index key is associated with the row number of the row in the table that contains other information related to the index key.
half of the index. Likewise, if the value is less than the key, the search continues at the center row of the upper half of the index. The process is repeated until either a match is found or the search criteria cannot be found.

**An Index in Motion**

Unlike an index of a book, a table can be associated with multiple indexes, each of which contains a different key. For example, a customer information table might have an index based on zip code and another on customer number. The designer of the database determines the number of indexes that are associated with a table, although some DBMSs create their own indexes to speed searches if an appropriate index doesn’t exist.

A J2ME application sends the DBMS a query that contains search criteria for information required by the component. Instead of searching the table that contains the search criteria, the DBMS compares the search criteria to keys of indexes, looking for an index to use in the search.

Let’s say that a J2ME application needs to retrieve information about a customer. The request for information (called a query) is sent to the DBMS along with the customer number. The DBMS recognizes that the request contains a customer number, then searches a catalog of indexes for an index that uses customer number as its key.

The DBMS always uses an index, if one exists, to locate search criteria. If an index doesn’t match the search criteria, the DBMS either creates a temporary index as part of the search or sequentially searches the table. The method used with your DBMS depends on how the manufacturer designed the DBMS.

Once an index is located or a temporary index is created, the DBMS compares the search criteria contained in the request to the index key. When a match is found, the DBMS notes the row number in the second column of the index, then opens the associated table and moves directly to the row that corresponds to the row number. The DBMS then selects columns from the row that contains customer information requested by the J2ME application.

**Drawbacks of Using an Index**

An index offers an unparalleled advantage for finding information in a table quickly. However, there is a drawback when too many indexes are used with one table. An unacceptable delay can occur whenever a row is inserted into or deleted from the table. Once an index is built, the DBMS is responsible for automatically maintaining the index whenever a row is inserted or deleted from the table. This means that each index associated with a table must be modified whenever a row is inserted or deleted from the table, which can cause performance degradation if a table is associated with many indexes. The trade-off for using multiple indexes is the time necessary for the DBMS to maintain each index.
Performance degradation can be minimized by using a publisher-consumer database design for applications where rows are frequently inserted, deleted, or modified, as in an order processing system. The publisher-consumer database design consists of two or more databases that contain the same information. One database, called the publisher, receives requests from the J2ME application to insert a new row, modify, or delete an existing row. J2ME applications don’t use the publisher to retrieve information, therefore the publisher database isn’t indexed. A consumer database receives instructions to insert a new row, modify, or delete an existing row from the publisher. However, the consumer receives requests for information from a J2ME application. Therefore, the consumer database is indexed.

Once the publisher passes data to the consumer, the publisher is free to process the next incoming data, during which time the DBMS is updating the consumer’s indexes. There are two major benefits to this design. First, a bottleneck is avoided when many requests are received by the DBMS to insert new information or update existing information. This is because the publisher has two tasks to perform: insert or modify the database, then pass along those changes to the consumer.

The other benefit is that the database becomes scalable. This means that the database can be adjusted to handle an increased volume of requests. Let’s say a company begins with a publisher and one consumer. Over time there is an increase in the number of requests for information, which can overwhelm the DBMS and cause performance degradation. This can be remedied by creating a second consumer and routing requests to one or the other consumer, reducing response time by half.

**Clustered Keys**

A clustered key is an index key whose value represents data contained in multiple columns of the corresponding table. Although there is only one column in the index to store the index key, that value can be a combination of data from multiple columns of the table.

Let’s say a customer name is used as the index key. Customer name is composed of two data elements: the customer first name and the customer last name. Since there is only one column for the index key, the customer first name and the customer last name are concatenated into one value. That is, the DBMS takes one data element and places it behind the other data element to create a new data element that becomes the index key.

Let’s say that the customer first name is Tom and last name is Jones. Here’s what they look like when they are concatenated: TomJones. Notice there isn’t a space between the first and last name. This is because concatenating places the character of the second value immediately following the last character of the first value.

Clustered keys add overhead to a DBMS because columns are delimited, and the delimiter must be escaped if it appears in the data. This extra step might impede performance of the DBMS. The DBMS treats the index key of a clustered index as it does any index key and uses a combination of trees and hashing algorithms to locate search criteria in the index.
Use concatenation of data elements to:

- Create an index key that uses two or more columns to uniquely identify rows in a table
- Facilitate searching for values of multiple columns, such as a customer name, using one index

**Derived Keys**

A derived key consists of a value that represents part of the value of a column rather than the entire value of the column. Let’s say that an order number comprises three components: the first component represents the sales region where the order was placed, the second is the sales representative’s number, and the third is a unique number that identifies the order. Although the order number appears in one column of the order table, a component of the order number can be used as an index key. For example, the DBMS can be instructed to derive an index key from the order number by using the first component. This means the index can be used to search for all orders placed within a region that is specified in the search criteria.

**Selective Rows**

Typically, all the rows in a table are represented in an index associated with that table. However, an index can be created that references a subset of rows in a table. The subset is determined when the index is created.

Let’s return to the order number example used in the previous section to illustrate this feature. There might be tens of thousands of orders, each having a row in the order table and in indexes associated with the order table. Suppose a J2ME application is used only to search for orders within a specified region. There will never be an occasion for the component to search other than in the region. Indexes used for searches by the component can be limited to only rows of the table that contain orders placed within the region as identified by the first component of the order number. This means that the index does not contain any references to rows in the order table that are outside the specified region.

A performance benefit is realized by using an index that contains a subset of all the rows of the associated table. This is because rows that will never be searched are excluded from the index, thereby reducing the number of rows of the index that must be searched by the DBMS. The boost in performance, however, is only realized in databases that contain huge amounts of rows. Little if any increase in performance is realized by creating a subset of rows in a typical database because many DBMSs have been optimized to search volumes of data without having to use a subset of rows in an index.

Using a conditional statement when creating an index, which you’ll learn to do in the next two chapters, creates a subset. In the order number example, the conditional statement directs the DBMS to include rows in the index where the first component of the order number matches the conditional statement. All other rows are excluded from the index.
Exact Matches and Partial Matches

A DBMS can be instructed to use an index to find an exact or partial match to the search criteria. By default, the DBMS searches for exact matches whenever a query is received from a J2ME application. However, the programmer can construct the query to direct the DBMS to find partial matches.

An exact match requires that all the characters of the search criteria match the index key. If a single character is mismatched, the DBMS does not return data in the corresponding row of the table. A partial match requires that some—not all—of the characters of the index key match the search criteria. That is, if the first character of the search criteria and the index key are the same and the remaining characters are different, the DBMS still considers it a match and returns data in the corresponding row of the table. Exact matches are used whenever a particular value, and only that value, is required, such as a specific customer number. Partial matches are used whenever someone is unsure of the exact value.

Let’s say that a sales representative is looking for information about a customer, but she isn’t sure if the customer’s last name is Johnson or Johnston. She can search using an index of customer names that looks for any customers with the first five letters of “Johns” and any other characters as their last name.

In this example, the customer name index is a concatenated index. The order in which values are concatenated plays a critical role when a DBMS searches for a partial match. This is because the DBMS begins matches left to right. Therefore, the most significant value must be placed first in the concatenated key. The most significant value in the customer name is the customer last name because there are more people with the same first name than the same last name. Therefore, the index key is last name, then first, such as JohnstonMike.

Searching for Phonetic Matches

Some DBMSs feature phonetic searches in which the DBMS returns rows containing index keys that “sound” like the search value. This means that the DBMS stores both exact spelling and phonetic spelling of the index key. Phonetic searches are a valuable feature to look for in a DBMS, especially for use with customer service databases. Customer service typically must take bits and pieces of information provided by a customer, and then assemble those pieces into meaningful search criteria.

Many interactions with customers take place over the telephone, and customer service representatives may not have printed materials (such as invoices) that the customer has at hand. This situation typically leads to miscommunication, which aggravates a customer relationship. Although a phonetic search won’t guarantee better communication between the customer service representative and the customer, it does give the customer service representative a tool for locating information necessary to respond properly to a customer.

Phonetic searches are made possible by an algorithm built into the DBMS. The phonetic algorithm used by the DBMS defines each index key phonetically. Likewise, the DBMS converts the search criteria into its phonetic spelling before comparing the phonetic spelling of the search criteria to the phonetic spelling of the index key.