SQL

A STEP BY STEP GUIDE TO LEARN SQL FOR ABSOLUTE BEGINNERS

LILLY TRINITY

SQL

A STEP BY STEP GUIDE TO LEARN SQL FOR ABSOLUTE BEGINNER

By

Lilly Trinity

Copyright

Copyright © 2019 All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, without the prior written permission of the publisher.

Table of Contents

Copyright
Introduction
- Why Learn SQL?
- Understand Relational Database
- The Relational Model

What Is SQL?
- The SQL Evolution
- SQL Statement Classes
- Types of SQL Statements
- Types of Execution
- A Beginner's Guide
- A Nonprocedural Language

Creating and Populating a Database
- Creating a MySQL Database
- Working with MySQL Environment
- Understand the SQL Environment
- Component Type Description SQL Agent
- Understand SQL Catalogs

Create a Database
- MySQL Data
- Creating a Database and Schema
- Query Mechanics
- Query Clauses

Querying Multiple Tables
- What Is a Join?
- Cartesian Product
- Joining Three or More Tables
- Using Subqueries as Table

Working with Sets
- Set Theory Primer
- Set Operators
- The Union Operator
- The Intersect Operator
- Set Operation Rules

Data Generation, Conversion, and Manipulation
- Working with String Data
- String Generation
- String Manipulation
- Working with Numeric Data
- Controlling Number Precision
- Implicit Versus Explicit Groups
- Using Expressions

Subqueries
- What Is a Subquery?
- Subquery Types
- The exists Operator
- Data Manipulation Using Correlated Subqueries
- Transactions
- Lock Granularities
- What Is a Transaction?

- Starting a Transaction

Indexes and Constraints
- Indexes
- Types of Indexes
- How Indexes Are Used
- The Downside of Indexes
- Constraints
- Constraint Creation

Views
- What Are Views?
- Why Use Views?
- Data Aggregation

Conclusion

Disclaimer

Introduction

Programming languages come and go constantly, and very few languages in use today have roots going back more than a decade or so. Some examples are Cobol, which is still used quite heavily in mainframe environments, and C, which is still quite popular for operating system and server development and for embedded systems. In the database arena, we have SQL, whose roots go all the way back to the 1970s.

Relational databases have become the most common data storage mechanism for modern computer applications. Programming languages such as Java, C, and COBOL, and scripting languages such as Perl, VBScript, and JavaScript must often access a data source in order to retrieve or modify data. Many of these data sources are managed by a relational database management system (RDBMS), such as Oracle, Microsoft SQL Server, MySQL, and DB2, that relies on the Structured Query Language (SQL) to create and alter database objects, add data to and delete data from the database, modify data that has been added to that database, and of course, retrieve data stored in the database for display and processing.

SQL is the most widely implemented language for relational databases. Much as mathematics is the language of science, SQL is the language of relational databases. SQL not only allows you to manage the data within the database, but also manage the database itself.

SQL is the language for generating, manipulating, and retrieving data from a relational database. One of the reasons for the popularity of relational databases is that properly designed relational databases can handle huge amounts of data. When working with large data sets, SQL is akin to one of those snazzy digital cameras with the high-power zoom lens in that you can use SQL to look at large sets of data, or you can zoom

in on individual rows (or anywhere in between). Other database management systems tend to break down under heavy loads because their focus is too narrow (the zoom lens is stuck on maximum), which is why attempts to dethrone relational databases and SQL have largely failed. Therefore, even though SQL is an old language, it is going to be around for a lot longer and has a bright future in store.

By using SQL statements, you can access an SQL database directly by using an interactive client application or through an application programming language or scripting language.

Regardless of which method you use to access a data source, a foundation in how to write SQL statements is required in order to access relational data. SQL: A Beginner's Guide, provides you with such a foundation. It describes the types of statements that SQL supports and explains how they're used to manage databases and their data. By working through this book, you'll build a strong foundation in basic SQL and gain a comprehensive understanding of how to use SQL to access data in your relational database.

Why Learn SQL?

If you are going to work with a relational database, whether you are writing applications, performing administrative tasks, or generating reports, you will need to know how to interact with the data in your database. Even if you are using a tool that generates SQL for you, such as a reporting tool, there may be times when you need to bypass the automatic generation feature and write your own SQL statements.

Learning SQL has the added benefit of forcing you to confront and understand the data structures used to store information about your organization. As you become comfortable with the tables in your database, you may find yourself proposing modifications or additions to your database schema.

Understand Relational Database

Structured Query Language (SQL) supports the creation and maintenance of the relational database and the management of data within that database. However, before I go into a discussion about relational databases, I want to explain what I mean by the term database.

The term itself has been used to refer to anything from a collection of names and addresses to a complex system of data retrieval and storage that relies on user interfaces and a network of client computers and servers.

A database is nothing more than a set of related information. A telephone book, for example, is a database of the names, phone numbers, and addresses of all people living in a particular region. While a telephone book is certainly a ubiquitous and frequently used database, it suffers from the following:

- Finding a person's telephone number can be time-consuming, especially if the telephone book contains a large number of entries.
- A telephone book is indexed only by last/first names, so finding the names of the people living at a particular address, while possible in theory, is not a practical use for this database.
- From the moment the telephone book is printed, the information becomes less and
- less accurate as people move into or out of a region, change their telephone numbers, or move to another location within the same region.

The same drawbacks attributed to telephone books can also apply to any manual data

storage system, such as patient records stored in a filing cabinet. Because of the cumbersome nature of paper databases, some of the first computer applications developed were database systems, which are computerized data storage and retrieval mechanisms.

Because a database system stores data electronically rather than on paper, a database system is able to retrieve data more quickly, index data in multiple ways, and deliver up-to-the-minute information to its user community.

Early database systems managed data stored on magnetic tapes. Because there were generally far more tapes than tape readers, technicians were tasked with loading and unloading tapes as specific data was requested.

Because the computers of that era had very little memory, multiple requests for the same data generally required the data to be read from the tape multiple times. While these database systems were a significant improvement over paper databases, they are a far cry from what is possible with today's technology. (Modern database systems can manage terabytes of data spread across many fast-access disk drives, holding tens of gigabytes of that data in high-speed memory, but I'm getting a bit ahead of myself).

Over the years, a number of database models have been implemented to store and manage data. Several of the more common models include the following:

Hierarchical: This model has a parent–child structure that is similar to an inverted tree, which is what forms the hierarchy. Data is organized in nodes, the logical equivalent of tables in a relational database. A parent node can have many child nodes, but a child node can have only one parent node. Although the model has been highly implemented, it is often considered unsuitable for many applications because of its inflexible structure and lack of support for complex relationships. Still, some implementations such as IMS from IBM have introduced features that work around these limitations.

Network: This model addresses some of the limitations of the hierarchical model. Data is organized in record types, the logical equivalent of tables in a relational database. Like the hierarchical model, the network model uses an inverted tree structure, but record types are organized into a set structure that relates pairs of record types into owners and members. Any one record type can participate in any set with other record types in the database, which supports more complex queries and relationships than are possible in the hierarchical model. Still, the network model has its limitations, the most serious of which is complexity. In accessing the database, users must be very familiar with the structure and keep careful track of where they are and how they got there. It's also difficult to change the structure without affecting applications that interact with the database.

Relational: This model addresses many of the limitations of both the

hierarchical and network models. In a hierarchical or network database, the application relies on a defined implementation of that database, which is then hard-coded into the application. If you add a new attribute (data item) to the database, you must modify the application, even if it doesn't use the attribute. However, a relational database is independent of the application; you can make nondestructive modifications to the structure without impacting the application. In addition, the structure of the relational database is based on the relation, or table, along with the ability to define complex relationships between these relations. Each relation can be accessed directly, without the cumbersome limitations of a hierarchical or owner/member model that requires navigation of a complex data structure. In the following section, "The Relational Model," I'll discuss the model in more detail.

Although still used in many organizations, hierarchical and network databases are now considered legacy solutions. The relational model is the most extensively implemented model in modern business systems, and it is the relational model that provides the foundation for SQL.

The Relational Model

In 1970, Dr. E. F. Codd of IBM's research laboratory published a paper titled "A Relational Model of Data for Large Shared Data Banks" that proposed that data be represented as sets of tables. Rather than using pointers to navigate between related entities, redundant data is used to link records in different tables.

Codd defines a relational data structure that protects data and allows that data to be manipulated in a way that is predictable and resistant to error. The relational model, which is rooted primarily in the mathematical principles of set theory and predicate logic, supports easy data retrieval, enforces data integrity (data accuracy and consistency), and provides a database structure independent of the applications accessing the stored data.

At the core of the relational model is the relation. A relation is a set of columns and rows collected in a table-like structure that represents a single entity made up of related data. An entity is a person, place, thing, event, or concept about which data is collected, such as a recording

artist, a book, or a sales transaction. Each relation comprises one or more attributes (columns). An attribute is a unit fact that describes or characterizes an entity in some way. For example, the entity is a compact disc (CD) with attributes of CD_NAME (the title of the CD), ARTIST_NAME (the name of the recording artist), and COPYRIGHT_YEAR (the year the recording was copyrighted).

Each attribute has an associated domain. A domain defines the type of data that can be stored in a particular attribute; however, a domain is not the same thing as a data type. A data type, which is, is a specific kind of constraint (a control used to enforce data integrity) associated with a column, whereas a domain, as it is used in the relational model, has a much broader meaning and describes exactly what data can be included in an attribute associated with that domain. For example, the COPYRIGHT_YEAR attribute is associated with the Year domain.

It is common practice to include a class word that describes the domain in attribute names, but this is not at all mandatory. The domain can be defined so that the attribute includes only data whose values and format are limited to years, as opposed to days or months. The domain might also limit the data to a specific range of years. A data type, on the other hand, restricts the format of the data, such as allowing only numeric digits, but not the values, unless those values somehow violate the format.

NOTE

The logical terms relation, attribute, and tuple are used primarily when referring to the relational model. SQL uses the physical terms table, column, and row to describe these items. Because the relational model is based on mathematical principles (a logical model) and SQL is concerned more with the physical implementation of the model, the meanings for the model’s terms and the SQL language’s terms are slightly different, but the underlying principles are the same.

What Is SQL?

Along with Codd's definition of the relational model, he proposed a language called DSL/Alpha for manipulating the data in relational tables. Shortly after Codd's paper was released, IBM commissioned a group to build a prototype based on Codd's ideas.

This group created a simplified version of DSL/Alpha that they called SQUARE. Refinements to SQUARE led to a language called SEQUEL, which was, finally, renamed SQL.

Now that you have a fundamental understanding of the relational model, it's time to introduce you to SQL and its basic characteristics. As you might recall from the "Understand Relational Databases" section earlier in this chapter, SQL is based on the relational model, although it is not an exact implementation. While the relational model provides the theoretical underpinnings of the relational database, it is the SQL language that supports the physical implementation of that database SQL, a nearly universally implemented relational language, is different from other computer languages such as C, COBOL, and Java, which are procedural. A procedural language defines how an application's operations should be performed and the order in which they are performed. A nonprocedural language, on the other hand, is concerned more with the results of an operation; the underlying software environment determines how the operations will be processed. This is not to say that SQL supports no procedural functionality. For example, stored procedures, added to many RDBMS products a number of years ago, are part of the SQL:2006 standard and provide procedural-like capabilities.

SQL still lacks many of the basic programming capabilities of most other computer languages. For this reason, SQL is often referred to as a data

sublanguage because it is most often used in association with application programming languages such as C and Java, languages that are not designed for manipulating data stored in a database. As a result, SQL is used in conjunction with the application language to provide an efficient means of accessing that data, which is why SQL is considered a sublanguage.

The SQL Evolution

In the early 1970s, after E. F. Codd's groundbreaking paper had been published, IBM began to develop a language and a database system that could be used to implement that model. When it was first defined, the language was referred to as Structured English Query Language (SEQUEL). When it was discovered that SEQUEL was a trademark owned by Hawker-Siddeley Aircraft Company of the UK, the name was changed to SQL. As word got out that IBM was developing a relational database system based on SQL, other companies began to develop their own SQL-based products.

In fact, Relational Software, Inc., now the Oracle Corporation, released their database system before IBM got their product to market.

As more vendors released their products, SQL began to emerge as the standard relational database language.

In 1986, the American National Standards Institute (ANSI) released the first published standard for the language (SQL-86), which was adopted by the International Organization for Standardization (ISO) in 1987. The standard was updated in 1989, 1992, 2003, 2006, and work continues. It has grown over time the original standard was well under 1,000 pages, while the SQL:2006 version weighs in at more than 3,700 pages. The standard was written in parts to permit more timely publication of revisions and to facilitate parallel work by different committees. It provides an overview of the parts and the current status of each RDBMS vendors had products on the market before there was a standard, and some of the features in those products were implemented differently enough that the standard could not accommodate them all when it was developed. We often call these vendor extensions. This may explain why there is no standard for a database. And as each release of the SQL standard comes out, RDBMS vendors have to work to incorporate the

new standard into their products.

SQL Statement Classes

The SQL language is divided into several distinct parts: the parts that we explore in this book include SQL schema statements, which are used to define the data structures stored in the database; SQL data statements, which are used to manipulate the data structures previously defined using SQL schema statements; and SQL transaction statements, which are used to begin, end, and roll back transactions. For example, to create a new table in your database, you would use the SQL schema statement create table, whereas the process of populating your new table with data would require the SQL data statement insert.

To give you a taste of what these statements look like, here's an SQL schema statement that creates a table called corporation:

```
CREATE TABLE corporation

(corp_id SMALLINT,

name VARCHAR(30),

CONSTRAINT pk_corporation PRIMARY KEY (corp_id)

);
```

This statement creates a table with two columns, corp_id and name, with the corp_id column identified as the primary key for the table. We probe the finer details of this statement, such as the different data types available with MySQL. Next, here's an SQL data statement that inserts a row into the corporation table for Acme Paper Corporation:

```
INSERT INTO corporation (corp_id, name)

VALUES (27, 'Acme Paper Corporation');
```

This statement adds a row to the corporation table with a value of 27 for

the corp_id column and a value of Acme Paper Corporation for the name column.

Finally, here's a simple select statement to retrieve the data that was just created:

```
mysql< SELECT name

-> FROM corporation

-> WHERE corp_id = 27;
```

name
Acme Paper Corporation

All database elements created via SQL schema statements are stored in a special set of tables called the data dictionary. This "data about the database" is known collectively as metadata and is explored. Just like tables that you create yourself, data dictionary tables can be queried via a select statement, thereby allowing you to discover the current data structures deployed in the database at runtime. For example, if you are asked to write a report showing the new accounts created last month, you could either hardcode the names of the columns in the account table that were known to you when you wrote the report, or query the data dictionary to determine the current set of columns and dynamically generate the report each time it is executed

Types of SQL Statements

Although SQL is considered a sublanguage because of its nonprocedural nature, it is nonetheless a complete language in that it allows you to create and maintain database objects, secure those objects, and manipulate the data within the objects. One common method used to categorize SQL statements is to divide them according to the functions they perform. Based on this method, SQL can be separated into three types of statements:

Data Definition Language (DDL): DDL statements are used to

create, modify, or delete database objects such as tables, views, schemas, domains, triggers, and stored procedures.

The SQL keywords most often associated with DDL statements are CREATE, ALTER, and DROP. For example, you would use the CREATE TABLE statement to create a table, the ALTER TABLE statement to modify the table's properties, and the DROP TABLE statement to delete the table definition from the database.

Data Control Language (DCL): DCL statements allow you to control who or what (a database user can be a person or an application program) has access to specific objects in your database. With DCL, you can grant or restrict access by using the GRANT or REVOKE statements, the two primary DCL commands. The DCL statements also allow you to control the type of access each user has to database objects. For example, you can determine which users can view a specific set of data and which users can manipulate that data.

Data Manipulation Language (DML): DML statements are used to retrieve, add, modify, or delete data stored in your database objects. The primary keywords associated with DML statements are SELECT, INSERT, UPDATE, and DELETE, all of which represent the types of statements you'll probably be using the most. For example, you can use a SELECT statement to retrieve data from a table and an INSERT statement to add data to a table.

Most SQL statements that you'll be using fall neatly into one of these categories, and I'll be discussing a number of these statements throughout the remainder of the book.

Types of Execution

In addition to defining how the language can be used, the SQL: 2006 standard provides details on how SQL statements can be executed. These methods of execution, known as binding styles, not only affect the nature of the execution, but also determine which statements, at a minimum, must be supported by a particular binding style. The standard defines four methods of execution:

Direct Invocation: By using this method, you can communicate

directly from a front end application, such as iSQL*Plus in Oracle or Management Studio in Microsoft SQL Server, to the database. (The front-end application and the database can be on the same computer, but often are not.) You simply enter your query into the application window and execute your SQL statement. The results of your query are returned to you as immediately as processor power and database constraints permit. This is a quick way to check data, verify connections, and view database objects. However, the SQL standard's guidelines about direct invocation are fairly minimal, so the methods used and SQL statements supported can vary widely from product to product.

Embedded SQL: In this method, SQL statements are encoded (embedded) directly in the host programming language. For example, you can embed SQL statements within C application code. Before the code is compiled, a preprocessor analyzes the SQL statements and splits them out from the C code. The SQL code is converted to a form the RDBMS can understand, and the remaining C code is compiled as it would be normally.

Module Binding: This method allows you to create blocks of SQL statements (modules) that are separate from the host programming language. Once the module is created, it is combined into an application with a linker. A module contains, among other things, procedures, and it is the procedures that contain the actual SQL statements.

Call-level interface (CLI): A CLI allows you to invoke SQL statements through an interface by passing SQL statements as argument values to subroutines. The statements are not precompiled as they are in embedded SQL and module binding. Instead, they are executed directly by the RDBMS.

Direct invocation, although not the most common method used, is the one I'll be using primarily for the examples and exercises in this book because it supports the submission of ad hoc queries to the database and generates immediate results. However, embedded SQL is currently the method most commonly used in business applications

A Nonprocedural Language

If you have worked with programming languages in the past, you are used to defining variables and data structures, using conditional logic (i.e., if-then-else) and looping constructs (i.e., do while ... end), and breaking your code into small, reusable pieces (i.e., objects, functions, procedures). Your code is handed to a compiler, and the executable that results does exactly (well, not always exactly) what you programmed it to do. Whether you work with Java, C#, C, Visual Basic, or some other procedural language, you are in complete control of what the program does.

A procedural language defines both the desired results and the mechanism, or process, by which the results are generated. Nonprocedural languages also define the desired results, but the process by which the results are generated is left to an external agent.

With SQL, however, you will need to give up some of the control you are used to, because SQL statements define the necessary inputs and outputs, but the manner in which a statement is executed is left to a component of your database engine known as the optimizer. The optimizer's job is to look at your SQL statements and, taking into account how your tables are configured and what indexes are available, decide the most efficient execution path (well, not always the most efficient).

Most database engines will allow you to influence the optimizer's decisions by specifying optimizer hints, such as suggesting that a particular index be used; most SQL users, however, will never get to this level of sophistication and will leave such tweaking to their database administrator or performance expert. With SQL, therefore, you will not be able to write complete applications. Unless you are writing a simple script to manipulate certain data, you will need to integrate SQL with your favorite programming language. Some database vendors have done this for you, such as Oracle's PL/SQL language, MySQL's stored procedure language, and Microsoft's Transact-SQL language. With these languages, the SQL data statements are part of the language's grammar, allowing you to seamlessly integrate database queries with procedural commands. If you are using a non-database-specific language such as Java, however, you will need to use a toolkit/API to execute SQL statements from your code. Some of these toolkits are provided by your

database vendor, whereas others are created by third-party vendors or by open source providers. Table 1 shows some of the available options for integrating SQL into a specific language.

Table 1 SQL Integration Toolkits

Language	**Toolkit**
Java	JDBC (Java Database Connectivity; JavaSoft)
C++	Rogue Wave SourcePro DB (third-party tool to connect to Oracle, SQL Server, MySQL, Informix, DB2, Sybase, and PostgreSQL databases)
C/C++	Pro*C (Oracle), MySQL C API (open source), and DB2
	Call Level Interface (IBM) #
C#	ADO.NET (Microsoft)
Perl	Perl DBI
Python	Python DB
Visual Basic	ADO.NET (Microsoft)

Creating and Populating a Database

This chapter provides you with the information you need to create your first database and to create the tables and associated data used for the examples in this book. You will also learn about various data types and see how to create tables using them. Because the examples in this book are executed against a MySQL database, this chapter is somewhat skewed toward MySQL's features and syntax, but most concepts are applicable to any server.

Creating a MySQL Database

If you already have a MySQL database server available for your use, you can skip the installation instructions and start with the instructions. Keep in mind, however, that this book assumes that you are using MySQL version 6.0 or later, so you may want to consider upgrading your server or installing another server if you are using an earlier release. The following instructions show you the minimum steps required to install a MySQL 6.0 server on a Windows computer:

1. Go to the download page for the MySQL Database Server at http://dev.mysql.com/ downloads. If you are loading version 6.0, the full URL is http://dev.mysql.com/ downloads/mysql/6.0.html.

2. Download the Windows Essentials (x86) package, which includes only the commonly used tools.

3. When asked "Do you want to run or save this file?" click Run

4. The MySQL Server 6.0—Setup Wizard window appears. Click Next.

5. Activate the Typical Install radio button, and click Next.

6. Click Install.

7. A MySQL Enterprise window appears.

8. When the installation is complete, make sure the box is checked next to "Configure the MySQL Server now," and then click Finish. This launches the Configuration Wizard. 9. When the Configuration Wizard launches, activate the Standard Configuration radio button, and then select both the "Install as Windows Service" and "Include Bin Directory in Windows Path" checkboxes. Click Next.

10. Select the Modify Security Settings checkbox and enter a password for the root user (make sure you write down the password, because you will need it shortly!), and click Next.

11. Click Execute. At this point, if all went well, the MySQL server is installed and running. If not, I suggest you uninstall the server and read the "Troubleshooting a MySQL Installation Under Windows" guide

Next, you will need to open a Windows command window, launch the mysql tool, and create your database and database user.

Working with MySQL Environment

The SQL environment provides the structure in which SQL is implemented. Within this structure, you can use SQL statements to define database objects and store data in those objects. However, before you start writing SQL statements, you should have a basic understanding of the foundations on which the SQL environment is built so you can apply this information throughout the rest of the book. In fact, you might find it helpful to refer back to this chapter often to help gain a conceptual understanding of the SQL environment and how it relates to the SQL elements you'll learn about in subsequent chapters.

Understand the SQL Environment

The SQL environment is, quite simply, the sum of all the parts that make up that environment. Each distinct part, or component, works in conjunction with other components to support SQL operations such as creating and modifying objects, storing and querying data, or modifying and deleting that data. Taken together, these components form a model on which an RDBMS can be based. This does not imply, however, that RDBMS vendors adhere strictly to this model; which components they implement and how they implement them are left, for the most part, to the discretion of those vendors. Even so, I want to provide you with an overview of the way in which the SQL environment is defined, in terms of its distinct components, as they are described in the SQL:2006 standard.

The SQL environment is made up of six types of components, as shown in Figure 1. The SQL client and SQL servers are part of the SQL implementation and are therefore subtypes of that component. Notice that there is only one SQL agent and one SQL implementation, but there are multiple components for other types, such as catalogs and sites. According to SQL:2006, there must be exactly one SQL agent and SQL implementation and zero or more SQL client modules, authorization identifiers, and catalogs. The standard does not specify how many sites are supported, but implies multiple sites.

Each type of component performs a specific function within the SQL environment. Table 1 describes the eight types. For the most part, you need to have only a basic understanding of the components that make up an SQL environment (in terms of beginning SQL programming).

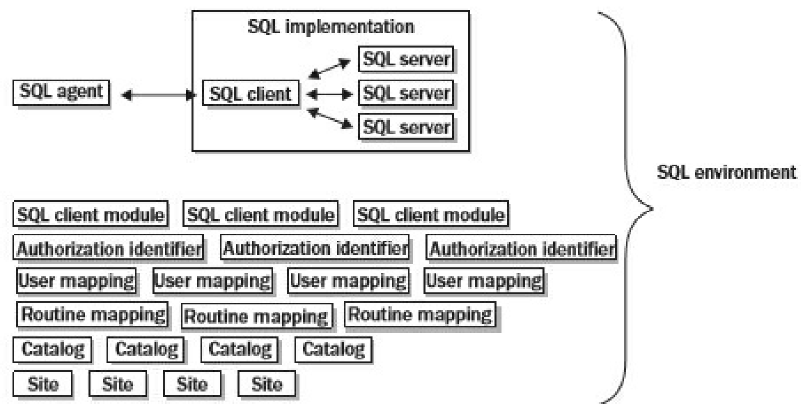


Figure 2-1 The components of the SQL environment

However, one of these components the catalog plays a more critical role than the others, with regard to what you'll be learning in this book. Therefore, I will cover this topic in more detail and explain how it relates to the management of data and the objects that hold that data.

Component Type Description SQL Agent

Any structure that causes SQL statements to be executed. The SQL agent is bound to the SQL client within the SQL implementation. SQL implementation a processor that executes SQL statements according to the requirements of the SQL agent. The SQL implementation includes one SQL client and one or more SQL servers. The SQL client establishes SQL connections with the SQL servers and maintains data related to interactions with the SQL agent and the SQL servers. An SQL server manages the SQL session that takes place over the SQL connection and executes SQL statements received from the SQL client.

Component Type Description SQL Client Module

A collection of SQL statements that are written separately from your programming application language but that can be called from within that language. An SQL client module contains zero or more externally invoked procedures, with each procedure consisting of a single SQL statement. SQL client modules reside within the SQL environment and are processed by the SQL implementation, unlike embedded SQL, which is written within the application programming language and precompiled before the programming language is compiled.

Authorization Identifier

An identifier that represents a user or role that is granted specific access privileges to objects and data within the SQL environment. A user is an individual security account that can represent an individual, an application, or a system service. A role is a set of predefined privileges that can be assigned to a user or to another role.

User Mapping: A user mapping pairs an authorization identifier with a foreign server descriptor.

Routine Mapping: A routine mapping pairs an SQL-invoked routine with a foreign server descriptor.

Catalog: A group of schemas collected together in a defined namespace. Each catalog contains the information schema, which includes descriptors of a number of schema objects. The catalog itself provides a hierarchical structure for organizing data within the schemas. (A schema is basically a container for objects such as tables, views, and domains, all of which I'll be discussing in greater detail in the next section, "Understand SQL Catalogs.").

Site: A group of base tables that contain SQL data, as described by the contents of the schemas. This data may be thought of as "the database," but keep in mind that the SQL standard does not include a definition of the term "database" because it has so many different meanings.

Understand SQL Catalogs

In the previous section, "Understand the SQL Environment," I state that an SQL environment is the sum of all parts that make up that environment. You can use the same logic to describe a catalog, in that a catalog is a collection of schemas and these schemas, taken together, define a namespace within the SQL environment.

NOTE: A namespace is a naming structure that identifies related components in a specified environment. A namespace is often depicted as an inverted tree configuration to represent the hierarchical relationship of objects. For example, suppose your namespace includes two objects: OBJECT_1 and OBJECT_2. If the namespace is called

NAME_1, the full object names will be NAME_1.OBJECT_1 and NAME_1.OBJECT_2 (or some such naming configuration), thus indicating that they share the same namespace.

Another way to look at a catalog is as a hierarchical structure with the catalog as the parent object and the schemas as the child objects, as shown in Figure 2. At the top of the hierarchy is the SQL environment, which can contain zero or more catalogs (although an environment with zero catalogs wouldn't do you much good because the catalog is where you'll find the data definitions and SQL data). The schemas are located at the third tier, beneath the catalog, and the schema objects are at the fourth tier.

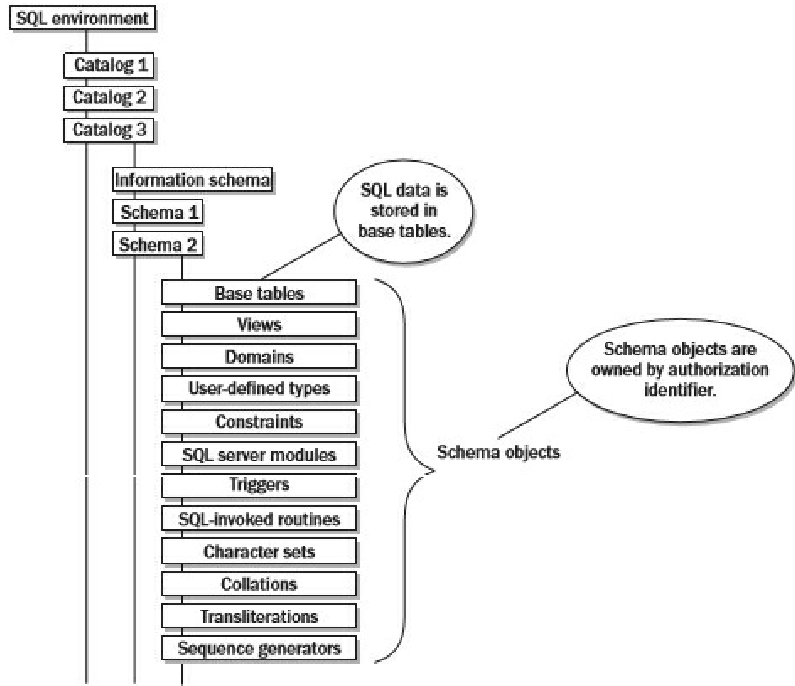


Figure 2-2 The components of a catalog

Create a Database

Despite the fact that the SQL standard does not define what a database is, let alone provide a statement to create any sort of database object, there is a good possibility that you'll be working with an RDBMS that not only supports the creation of a database object, but also uses that object as the foundation for its hierarchical structure in the management of data objects. Consequently, you might find that, in order to work through the examples and projects in this book, you will want to create a test database so you have an environment in which you can create, alter, or delete data objects or data as necessary, without risking the loss of data definitions or data from an actual database. (Ideally, you'll be working with an RDBMS that is a clean installation, without any existing databases, except preinstalled system and sample databases.) If you've already worked with an RDBMS, you might be familiar with how database objects are organized within that system. For example, you can see that SQL Server organizes the server's databases into a logical structure beneath the Databases node.

Each database node (for example, INVENTORY) contains child nodes that represent the different types of objects associated with that particular database. As you can see, the INVENTORY database currently lists eight categories of objects: Database Diagrams, Tables, Views, Synonyms, Programmability, Service Broker, Storage, and Security. And under the ARTIST_CDS table, the categories are Columns, Keys, Constraints, Triggers, Indexes, and Statistics. For a definition of how SQL Server defines each of these types of objects, you should view the product documentation, which you should do for any RDBMS.

Most products that support database objects also support language to create those objects. For example, Oracle, MySQL, and SQL Server all include the CREATE DATABASE statement in their SQL-based languages.

However, which parameters can be defined when building that statement, what permissions you need in order to execute that statement, and how a system implements the database object vary from product to product. Fortunately, most products use the same basic syntax to create a database object:

```
CREATE DATABASE <database name> <additional parameters>
```

Before creating a database in any system, make sure to first read the product documentation, and if appropriate, consult with a database administrator to be sure that it is safe for you to add a database object to the SQL environment. Once you create the database, you can create schemas, tables, views, and other objects within that database, and from there, populate the tables with the necessary data.

MySQL Data

Types In general, all the popular database servers have the capacity to store the same types of data, such as strings, dates, and numbers. Where they typically differ is in the specialty data types, such as XML documents or very large text or binary documents. Since this is an introductory book on SQL, and since 98% of the columns you encounter will be simple data types, this book covers only the character, date, and numeric data types.

Character Data

Character data can be stored as either fixed-length or variable-length strings; the difference is that fixed-length strings are right-padded with spaces and always consume the same number of bytes, and variable-length strings are not right-padded with spaces and don't always consume the same number of bytes. When defining a character column, you must specify the maximum size of any string to be stored in the column. For example, if you want to store strings up to 20 characters in length, you could use either of the following definitions:

```
char(20)    /* fixed-length */

varchar(20) /* variable-length */
```

The maximum length for char columns is currently 255 bytes, whereas

varchar columns can be up to 65,535 bytes. If you need to store longer strings (such as emails, XML documents, etc.), then you will want to use one of the text types (medium text and long text. In general, you should use the char type when all strings to be stored in the column are of the same length, such as state abbreviations, and the varchar type when strings to be stored in the column are of varying lengths. Both char and varchar are used in a similar fashion in all the major database servers.

Oracle Database is an exception when it comes to the use of varchar. Oracle users should use the varchar2 type when defining variable-length character columns.

Character Sets

For languages that use the Latin alphabet, such as English, there is a sufficiently small number of characters such that only a single byte is needed to store each character. Other languages, such as Japanese and Korean, contain large numbers of characters, thus requiring multiple bytes of storage for each character. Such character sets are therefore called multibyte character sets. MySQL can store data using various character sets, both single- and multibyte. To view the supported character sets in your server, you can use the show command, as in:

Numeric Data:

Although it might seem reasonable to have a single numeric data type called "numeric," there are actually several different numeric data types that reflect the various ways in which numbers are used, as illustrated here: A column indicating whether a customer order has been shipped This type of column, referred to as a Boolean, would contain a 0 to indicate false and a 1 to indicate true. A system-generated primary key for a transaction table This data would generally start at 1 and increase in increments of one up to a potentially very large number.

An item number for a customer's electronic shopping basket. The values for this type of column would be positive whole numbers between 1 and, at most, 200 (for shopaholics). Positional data for a circuit board drill machine High-precision scientific or manufacturing data often requires accuracy to eight decimal points. To handle these types of data (and more), MySQL has several different numeric data types. The most

commonly used numeric types are those used to store whole numbers. When specifying one of these types, you may also specify that the data is unsigned, which tells the server that all data stored in the column will be greater than or equal to zero.

Temporal Data:

Along with strings and numbers, you will almost certainly be working with information about dates and/or times. This type of data is referred to as temporal, and some examples of temporal data in a database include:

The future date that a particular event is expected to happen, such as shipping a customer's order

- The date that a customer's order was shipped
- The date and time that a user modified a particular row in a table
- An employee's birth date
- The year corresponding to a row in a yearly_sales fact table in a data warehouse
- The elapsed time needed to complete a wiring harness on an automobile assembly line.

While database servers store temporal data in various ways, the purpose of a format string is to show how the data will be represented when retrieved, along with how a date string should be constructed when inserting or updating a temporal column.

Creating a Database and Schema

Connecting to a Database," you established access to an RDBMS. In that project, you used a front-end application that allowed you to directly invoke SQL statements. You will be using that application for this project (and the rest of the projects in the book) to create a database and a schema, or whichever of these functions your system supports. Once you create the database, you should work within the context of that database for future examples and projects. If your system supports schema creation but not database creation, you should work within the context of that schema for the other projects. Step by Step 1.

Open the client application that allows you to directly invoke SQL statements. If applicable, check with the database administrator to make sure that you're logging in with the credentials necessary to create a database and schema. You might need special permissions to create these objects. Also verify whether there are any parameters you should include when creating the database (for example, log file size), restrictions on the name you can use, or restrictions of any other kind. Be sure to check the product documentation before going any further.

Create a database named INVENTORY (if your RDBMS supports this functionality in Oracle you'll want to create a user named INVENTORY, which will implicitly create a schema with the same name). Depending on the product you're using, you'll be executing a statement that's similar to the following:

CREATE DATABASE INVENTORY; If you're required to include any additional parameters in the statement, they would most likely be included in the lines following the CREATE DATABASE clause. Once you execute the statement, you should receive some sort of message telling you that the statement has been executed successfully. 3. Connect to the new database. The method for doing that will vary from product to product. In Oracle, you can connect to a database by entering the appropriate logon information in any of several tools such as SQL*Plus, iSQL*Plus, and SQL Developer. In SQL Server, it's simply a matter of selecting the appropriate database from the Connect drop-down list of databases in the SQL Server Management Studio toolbar, or you can execute the following statement (MySQL uses this same syntax).

Create a schema named CD_INVENTORY (if your RDBMS supports this functionality). Create the schema under your current authorization identifier. Do not include any schema elements at this time. In most cases, you will be executing a statement that looks similar to the following:

```
CREATE SCHEMA CD_INVENTORY;
```

Query Mechanics

Before dissecting the select statement, it might be interesting to look at how queries are executed by the MySQL server (or, for that matter, any

database server). If you are using the mysql command-line tool (which I assume you are), then you have already logged in to the MySQL server by providing your username and password (and possibly a hostname if the MySQL server is running on a different computer). Once the server has verified that your username and password are correct, a database connection is generated for you to use. This connection is held by the application that requested it (which, in this case, is the mysql tool) until the application releases the connection (i.e., as a result of your typing quit) or the server closes the connection (i.e., when the server is shut down). Each connection to the MySQL server is assigned an identifier, which is shown to you when you first log in:

This information might be useful to your database administrator if something goes awry, such as a malformed query that runs for hours, so you might want to jot it down. Once the server has verified your username and password and issued you a connection, you are ready to execute queries (along with other SQL statements). Each time a query is sent to the server, the server checks the following things prior to statement execution:

- Do you have permission to execute the statement?
- Do you have permission to access the desired data?
- Is your statement syntax correct?

If your statement passes these three tests, then your query is handed to the query optimizer, whose job it is to determine the most efficient way to execute your query. The optimizer will look at such things as the order in which to join the tables named in your from clause and what indexes are available, and then picks an execution plan, which the server uses to execute your query.

Query Clauses

Several components or clauses make up the select statement. While only one of them is mandatory when using MySQL (the select clause), you will usually include at least two or three of the six available clauses. Query clauses, Clause name, Purpose Select Determines which columns to include in the query's result set From Identifies the tables from which to draw data and how the tables should be joined where Filters out

unwanted data Group by Used to group rows together by common column values Having Filters out unwanted groups Order by Sorts the rows of the final result set by one or more columns .

The select Clause Even though the select clause is the first clause of a select statement, it is one of the last clauses that the database server evaluates. The reason for this is that before you can determine what to include in the final result set, you need to know all of the possible columns that could be included in the final result set. In order to fully understand the role of the select clause, therefore, you will need to understand a bit about the from clause. Here's a query to get started:

```
mysql> SELECT *

-> FROM department;
```

dept_id	name
1	Operations
2	Loans
3	Administration

3 rows in set (0.04 sec)

In this query, the from clause lists a single table (department), and the select clause indicates that all columns (designated by *) in the department table should be included in the result set. This query could be described in English as follows: Show me all the columns and all the rows in the department table. In addition to specifying all the columns via the asterisk character, you can explicitly name the columns you are interested in, such as:

```
mysql> SELECT dept_id, name

  -> FROM department;
```

dept_id	name
1	Operations
2	Loans
3	Administration

3 rows in set (0.01 sec)

The results are identical to the first query, since all the columns in the department table (dept_id and name) are named in the select clause. You can choose to include only a subset of the columns in the department table as well:

```
mysql> SELECT name

-> FROM department;
```

name
Operations
Loans
Administration

3 rows in set (0.00 sec)

The job of the select clause, therefore, is the following: The select clause determines which of all possible columns should be included in the query's result set. If you were limited to including only columns from the table or tables named in the from clause, things would be rather dull. However, you can spice things up by including in your select clause such things as:

- Literals, such as numbers or strings
- Expressions, such as transaction.amount * −1
- Built-in function calls, such as ROUND(transaction.amount

User-defined Function Calls

The next query demonstrates the use of a table column, a literal, an expression, and a built-in function call in a single query against the employee table:

```
mysql> SELECT emp_id,

         -> 'ACTIVE',

-> emp_id * 3.14159,

-> UPPER(lname)

-> FROM employee;
```

Column Aliases Although the mysql tool will generate labels for the columns returned by your queries, you may want to assign your own labels. While you might want to assign a new label to a column from a table (if it is poorly or ambiguously named), you will almost certainly want to assign your own labels to those columns in your result set that are generated by expressions or built-in function calls. You can do so by adding a column alias after each element of your select clause. Here's the previous query against the employee table with column aliases applied to three of the columns:

```
mysql> SELECT emp_id,

->  'ACTIVE' status,

 ->  emp_id * 3.14159 empid_x_pi,

->  UPPER(lname) last_name_upper

  -> FROM employee;
```

emp_id	status	empid_x_pi	last_name_upper
1	ACTIVE	3.14159	SMITH
2	ACTIVE	6.28318	BARKER
3	ACTIVE	9.42477	TYLER
4	ACTIVE	12.56636	HAWTHORNE
5	ACTIVE	15.70795	GOODING
6	ACTIVE	18.84954	FLEMING
7	ACTIVE	21.99113	TUCKER
8	ACTIVE	25.13272	PARKER
9	ACTIVE	28.27431	GROSSMAN
10	ACTIVE	31.41590	ROBERTS
11	ACTIVE	34.55749	ZIEGLER
12	ACTIVE	37.69908	JAMESON
13	ACTIVE	40.84067	BLAKE
14	ACTIVE	43.98226	MASON
15	ACTIVE	47.12385	PORTMAN
16	ACTIVE	50.26544	MARKHAM
17	ACTIVE	53.40703	FOWLER
18	ACTIVE	56.54862	TULMAN

If you look at the column headers, you can see that the second, third, and

fourth columns now have reasonable names instead of simply being labeled with the function or expression that generated the column. If you look at the select clause, you can see how the column aliases status, empid_x_pi, and last_name_upper are added after the second, third, and fourth columns. I think you will agree that the output is easier to understand with column aliases in place, and it would be easier to work with programmatically if you were issuing the query from within Java or C# rather than interactively via the mysql tool. In order to make your column aliases stand out even more, you also have the option of using the as keyword before the alias name, as in:

```
mysql> SELECT emp_id,

-> 'ACTIVE' AS status,

  -> emp_id * 3.14159

AS empid_x_pi,

  -> UPPER(lname)

 AS last_name_upper

-> FROM employee;
```

Removing Duplicates

In some cases, a query might return duplicate rows of data. For example, if you were to retrieve the IDs of all customers that have accounts, you would see the following:

```
mysql> SELECT cust_id

-> FROM account;
```

cust_id

1
1
1
2
2
3
3
4
4
4
5
6
6
7
8
8
9
9
9

10
10

24 rows in set (0.00 sec)

Since some customers have more than one account, you will see the same customer ID once for each account owned by that customer. What you probably want in this case is the distinct set of customers that have accounts, instead of seeing the customer ID for each row in the account table. You can achieve this by adding the keyword distinct directly after the select keyword, as demonstrated by the following:

The result set now contains 13 rows, one for each distinct customer, rather than 24 rows, one for each account. If you do not want the server to remove duplicate data, or you are sure there will be no duplicates in your result set, you can specify the ALL keyword instead of specifying DISTINCT. However, the ALL keyword is the default and never needs to be explicitly named, so most programmers do not include ALL in their queries.

Keep in mind that generating a distinct set of results requires the data to be sorted, which can be time-consuming for large result sets. Don't fall into the trap of using DISTINCT just to be sure there are no duplicates; instead, take the time to understand the data you are working with so that you will know whether duplicates are possible.

If all conditions in the where clause are separated by the or operator, however, only one of the conditions must evaluate to true for the row to be included in the result set. Consider the following two conditions: WHERE title = 'Teller' OR start_date < '2007-01-01' There are now various ways for a given employee row to be included in the result set:

- The employee is a teller and was employed prior to 2007.
- The employee is a teller and was employed after January 1, 2007.
- The employee is something other than a teller but was employed prior to 2007.

Intermediate result Final result WHERE true OR true True WHERE true

OR false True WHERE false OR true True WHERE false OR false False

In the case of the preceding example, the only way for a row to be excluded from the result set is if the employee is not a teller and was employed on or after January 1, 2007.

Building a Condition

Now that you have seen how the server evaluates multiple conditions, let’s take a step back and look at what comprises a single condition. A condition is made up of one or more expressions coupled with one or more operators. An expression can be any of the following:

- A number
- A column in a table or view
- A string literal, such as 'Teller'
- A built-in function, such as concat('Learning', ' ', 'SQL')
- A subquery
- A list of expressions, such as ('Teller', 'Head Teller', 'Operations Manager') The operators used within conditions include:
- Comparison operators, such as =, !=, <, >, <>, LIKE, IN, and BETWEEN
- Arithmetic operators, such as +, −, *, and / The following section demonstrates how you can combine these expressions and operators to manufacture the various types of conditions.

Condition Types

There are many different ways to filter out unwanted data. You can look for specific values, sets of values, or ranges of values to include or exclude, or you can use various pattern-searching techniques to look for partial matches when dealing with string data. The next four subsections explore each of these condition types in detail.

Equality Conditions

A large percentage of the filter conditions that you write or come across will be of the form 'column = expression' as in: title = 'Teller' fed_id = '111-11-1111' amount = 375.25 dept_id = (SELECT dept_id FROM

department WHERE name = 'Loans') Conditions such as these are called equality conditions because they equate one expression to another. The first three examples equate a column to a literal (two strings and a number), and the fourth example equates a column to the value returned from a subquery. The following query uses two equality conditions; one in the on clause (a join condition), and the other in the where clause (a filter condition):

```
mysql> SELECT pt.name product_type, p.name product
    -> FROM product p INNER JOIN product_type pt
  ->   ON p.product_type_cd = pt.product_type_cd
   -> WHERE pt.name = 'Customer Accounts';
```

product_type	product
Customer Accounts	certificate of deposit
Customer Accounts	checking account
Customer Accounts	money market account
Customer Accounts	savings account

4 rows in set (0.08 sec)

This query shows all products that are customer account types.

Range Conditions

Along with checking that an expression is equal to (or not equal to) another expression, you can build conditions that check whether an expression falls within a certain range. This type of condition is common when working with numeric or temporal data. Consider the following query:

```
mysql> SELECT emp_id, fname, lname, start_date

-> FROM employee

-> WHERE start_date

< '2007-01-01';
```

emp_id	fname	lname	start_date
1	Michael	Smith	2005-06-22
2	Susan	Barker	2006-09-12
3	Robert	Tyler	2005-02-09
4	Susan	Hawthorne	2006-04-24
8	Sarah	Parker	2006-12-02
9	Jane	Grossman	2006-05-03
10	Paula	Roberts	2006-07-27
11	Thomas	Ziegler	2004-10-23
13	John	Blake	2004-05-11
14	Cindy	Mason	2006-08-09
16	Theresa	Markham	2005-03-15
17	Beth	Fowler	2006-06-29
18	Rick	Tulman	2006-12-12

13 rows in set (0.15 sec)

This query finds all employees hired prior to 2007. Along with specifying an upper limit for the start date, you may also want to specify a lower range for the start date:

```
mysql> SELECT emp_id, fname, lname, start_date

-> FROM employee

-> WHERE start_date < '2007-01-01'

->   AND start_date >= '2005-01-01';
```

emp_id	fname	lname	start_date
1	Michael	Smith	2005-06-22
2	Susan	Barker	2006-09-12
3	Robert	Tyler	2005-02-09
4	Susan	Hawthorne	2006-04-24
8	Sarah	Parker	2006-12-02
9	Jane	Grossman	2006-05-03
10	Paula	Roberts	2006-07-27
11	Thomas	Ziegler	2004-10-23
13	John	Blake	2004-05-11
14	Cindy	Mason	2006-08-09
16	Theresa	Markham	2005-03-15
17	Beth	Fowler	2006-06-29

18	Rick	Tulman	2006-12-12

11 rows in set (0.00 sec)

This version of the query retrieves all employees hired in 2005 or 2006.

The between operator When you have both an upper and lower limit for your range, you may choose to use a single condition that utilizes the between operator rather than using two separate conditions, as in:

```
mysql> SELECT emp_id, fname, lname, start_date

-> FROM employee

-> WHERE start_date BETWEEN '2005-01-01' AND '2007-01-01';
```

emp_id	fname	lname	start_date
1	Michael	Smith	2005-06-22
2	Susan	Barker	2006-09-12
3	Robert	Tyler	2005-02-09
4	Susan	Hawthorne	2006-04-24
8	Sarah	Parker	2006-12-02
9	Jane	Grossman	2006-05-03
10	Paula	Roberts	2006-07-27
11	Thomas	Ziegler	2004-10-23
13	John	Blake	2004-05-11
14	Cindy	Mason	2006-08-09
16	Theresa	Markham	2005-03-15

17	Beth	Fowler	2006-06-29
18	Rick	Tulman	2006-12-12

11 rows in set (0.03 sec)

When using the between operator, there are a couple of things to keep in mind. You should always specify the lower limit of the range first (after between) and the upper limit of the range second (after and). Here's what happens if you mistakenly specify the upper limit first: mysql> SELECT emp_id, fname, lname, start_date -> FROM employee -> WHERE start_date BETWEEN '2007-01-01' AND '2005-01-01'; Empty set (0.00 sec)

Membership Conditions

In some cases, you will not be restricting an expression to a single value or range of values, but rather to a finite set of values. For example, you might want to locate all accounts whose product code is either 'CHK', 'SAV', 'CD', or 'MM':

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance

-> FROM account

-> WHERE product_cd = 'CHK' OR product_cd = 'SAV'

->   OR product_cd = 'CD' OR product_cd = 'MM';
```

account_id	product_cd	cust_id	avail_balance
1	CHK	1	1057.75
2	SAV	1	500.00
3	CD	1	3000.00
4	CHK	2	2258.02

5	SAV	2	200.00
7	CHK	3	1057.75
8	MM	3	2212.50
10	CHK	4	534.12
11	SAV	4	767.77
12	MM	4	5487.09
13	CHK	5	2237.97
14	CHK	6	122.37
15	CD	6	10000.00
17	CD	7	5000.00
18	CHK	8	3487.19
19	SAV	8	387.99
21	CHK	9	125.67
22	MM	9	9345.55
23	CD	9	1500.00
24	CHK	10	23575.12
28	CHK	12	38552.05

21 rows in set (0.28 sec)

While this where clause (four conditions or'd together) wasn't too tedious to generate, imagine if the set of expressions contained 10 or 20 members. For these situations, you can use the in operator instead:

SELECT account_id, product_cd, cust_id, avail_balance FROM account WHERE product_cd IN ('CHK','SAV','CD','MM');

With the in operator, you can write a single condition no matter how many expressions are in the set.

Matching Conditions

So far, you have been introduced to conditions that identify an exact string, a range of strings, or a set of strings; the final condition type deals with partial string matches. You may, for example, want to find all employees whose last name begins with T. You could use a built-in function to strip off the first letter of the lname column, as in:

```
mysql> SELECT emp_id, fname, lname

-> FROM employee

-> WHERE LEFT(lname, 1) = 'T';
```

emp_id	fname	lname
3	Robert	Tyler
7	Chris	Tucker
18	Rick	Tulman

3 rows in set (0.01 sec)

While the built-in function left() does the job, it doesn't give you much flexibility. Instead, you can use wildcard characters to build search expressions, as demonstrated in the next section.

Using wildcards When searching for partial string matches, you might be interested in:

- Strings beginning/ending with a certain character
- Strings beginning/ending with a substring
- Strings containing a certain character anywhere within the

string

- Strings containing a substring anywhere within the string
- Strings with a specific format, regardless of individual characters

Querying Multiple Tables

I demonstrated how related concepts are broken into separate pieces through a process known as normalization. The end result of this exercise was two tables: person and favorite_food. If, however, you want to generate a single report showing a person's name, address, and favorite foods, you will need a mechanism to bring the data from these two tables back together again; this mechanism is known as a join, and this chapter concentrates on the simplest and most common join, the inner join.

What Is a Join?

Queries against a single table are certainly not rare, but you will find that most of your queries will require two, three, or even more tables. To illustrate, let's look at the definitions for the employee and department tables and then define a query that retrieves data from both tables:

```
mysql> DESC employee;
```

Field	Type	Null	Key	Default
emp_id	smallint(5) unsigned	NO	PRI	NULL
fname	varchar(20)	NO	NULL	NULL
lname	varchar(20)	NO	NULL	NULL
start_date	Date	NO	NULL	NULL
end_date	Date	YES	NULL	NULL

superior_emp_id	smallint(5) unsigned	YES	NULL	NULL
dept_id	smallint(5) unsigned	YES	NULL	NULL
title		YES	NULL	NULL
assigned_branch_id	smallint(5) unsigned	YES	NULL	NULL

9 rows in set (0.11 sec)

```
mysql> DESC department;
```

Field	Type	Null	Key	Default
dept_id	smallint(5) unsigned	NO	PRI	NULL
name	varchar(20)	NO	NULL	NULL

2 rows in set (0.03 sec)

Let's say you want to retrieve the first and last names of each employee along with the name of the department to which each employee is assigned. Your query will therefore need to retrieve the employee.fname, employee.lname, and department.name columns. But how can you retrieve data from both tables in the same query? The answer lies in the employee.dept_id column, which holds the ID of the department to which each employee is assigned (in more formal terms, the employee.dept_id column is the foreign key to the department table).

The query, which you will see shortly, instructs the server to use the employee.dept_id column as the bridge between the employee and department tables, thereby allowing columns from both tables to be included in the query's result set. This type of operation is known as a join.

Cartesian Product

The easiest way to start is to put the employee and department tables into the from clause of a query and see what happens. Here's a query that retrieves the employee's first and last names along with the department name, with a from clause naming both tables separated by the join keyword:

```
mysql> SELECT e.fname, e.lname, d.name

-> FROM employee e JOIN department d;
```

fname	lname	name
Michael	Smith	Operations
Michael	Smith	Loans
Michael	Smith	Administration
Susan	Barker	Operations
Susan	Barker	Loans
Susan	Barker	Administration
Robert	Tyler	Operations
Robert	Tyler	Loans
Robert	Tyler	Administration
Susan	Hawthorne	Operations
Susan	Hawthorne	Loans
Susan	Hawthorne	Administration
John	Gooding	Operations

John	Gooding	Loans
John	Gooding	Administration
Helen	Gooding	Operations
Helen	Gooding	Loans
Helen	Gooding	Administration
Chris	Tucker	Operations
Chris	Tucker	Loans
Chris	Tucker	Administration
Sarah	Parker	Operations
Sarah	Parker	Loans
Sarah	Parker	Administration
Jane	Grossman	Operations
Jane	Grossman	Loans
Jane	Grossman	Administration
Paula	Roberts	Operations
Paula	Roberts	Loans
Paula	Roberts	Administration
Thomas	Ziegler	Operations
Thomas	Ziegler	Loans
Thomas	Ziegler	Administration

Samantha	Jameson	Operations
Samantha	Jameson	Loans
Samantha	Jameson	Administration
John	Blake	Operations
John	Blake	Loans
John	Blake	Administration
Cindy	Mason	Operations
Cindy	Mason	Loans
Cindy	Mason	Administration
Frank	Portman	Operations
Frank	Portman	Loans
Frank	Portman	Administration
Theresa	Markham	Operations
Theresa	Markham	Loans
Theresa	Markham	Administration
Beth	Fowler	Operations
Beth	Fowler	Loans
Beth	Fowler	Administration
Rick	Tulman	Operations

Rick	Tulman	Loans
Rick	Tulman	Administration

54 rows in set (0.23 sec)

Hmmm…there are only 18 employees and 3 different departments, so how did the result set end up with 54 rows? Looking more closely, you can see that the set of 18 employees is repeated three times, with all the data identical except for the department name. Because the query didn't specify how the two tables should be joined, the database server generated the Cartesian product, which is every permutation of the two tables (18 employees × 3 departments = 54 permutations). This type of join is known as a cross join, and it is rarely used (on purpose, at least).

Inner Joins

To modify the previous query so that only 18 rows are included in the result set (one for each employee), you need to describe how the two tables are related. Earlier, I showed that the employee.dept_id column serves as the link between the two tables, so this information needs to be added to the on subclause of the from clause:

```
mysql> SELECT e.fname, e.lname, d.name

-> FROM employee e JOIN department d

->   ON e.dept_id = d.dept_id;
```

fname	lname	name
Michael	Smith	Administration
Susan	Barker	Administration
Robert	Tyler	Administration
Susan	Hawthorne	Operations

John	Gooding	Loans
Helen	Fleming	Operations
Chris	Tucker	Operations
Sarah	Parker	Operations
Jane	Grossman	Operations
Paula	Roberts	Operations
Thomas	Ziegler	Operations
Samantha	Jameson	Operations
John	Blake	Operations
Cindy	Mason	Operations
Frank	Portman	Operations
Theresa	Markham	Operations
Beth	Fowler	Operations
Rick	Tulman	Operations

18 rows in set (0.00 sec)

Instead of 54 rows, you now have the expected 18 rows due to the addition of the on subclause, which instructs the server to join the employee and department tables by using the dept_id column to traverse from one table to the other. For example, Susan Hawthorne's row in the employee table contains a value of 1 in the dept_id column (not shown in the example). The server uses this value to look up the row in the department table having a value of 1 in its dept_id column and then retrieves the value 'Operations' from the name column in that row.

If a value exists for the dept_id column in one table but not the other,

then the join fails for the rows containing that value and those rows are excluded from the result set. This type of join is known as an inner join, and it is the most commonly used type of join. To clarify, if the department table contains a fourth row for the marketing department, but no employees have been assigned to that department, then the marketing department would not be included in the result set.

Likewise, if some of the employees had been assigned to department ID 99, which doesn't exist in the department table, then these employees would be left out of the result set. If you want to include all rows from one table or the other regardless of whether a match exists, you need to specify an outer join, but we cover this later in the book. In the previous example, I did not specify in the from clause which type of join to use. However, when you wish to join two tables using an inner join, you should explicitly specify this in your from clause; here's the same example, with the addition of the join type (note the keyword INNER):

```
mysql> SELECT e.fname, e.lname, d.name
  -> FROM employee e INNER JOIN department d
```

The ANSI Join Syntax

The notation used throughout this book for joining tables was introduced in the SQL92 version of the ANSI SQL standard. All the major databases (Oracle Database, Microsoft SQL Server, MySQL, IBM DB2 Universal Database, and Sybase Adaptive Server) have adopted the SQL92 join syntax. Because most of these servers have been around since before the release of the SQL92 specification, they all include an older join syntax as well. For example, all these servers would understand the following variation of the previous query:

```
mysql> SELECT e.fname, e.lname, d.name

-> FROM employee e, department d

-> WHERE e.dept_id = d.dept_id;
```

fname	lname	name

Michael	Smith	Administration
Susan	Barker	Administration
Robert	Tyler	Administration
Susan	Hawthorne	Operations
John	Gooding	Loans
Helen	Fleming	Operations
Chris	Tucker	Operations
Sarah	Parker	Operations
Jane	Grossman	Operations
Paula	Roberts	Operations
Thomas	Ziegler	Operations
Samantha	Jameson	Operations
John	Blake	Operations
Cindy	Mason	Operations
Frank	Portman	Operations
Theresa	Markham	Operations
Beth	Fowler	Operations
Rick	Tulman	Operations

18 rows in set (0.01 sec)

This older method of specifying joins does not include the on subclause;

instead, tables are named in the from clause separated by commas, and join conditions are included in the where clause. While you may decide to ignore the SQL92 syntax in favor of the older join syntax, the ANSI join syntax has the following advantages:

Join conditions and filter conditions are separated into two different clauses (the on subclause and the where clause, respectively), making a query easier to understand.

The join conditions for each pair of tables are contained in their own on clause, making it less likely that part of a join will be mistakenly omitted.

Queries that use the SQL92 join syntax are portable across database servers, whereas the older syntax is slightly different across the different servers. The benefits of the SQL92 join syntax are easier to identify for complex queries that include both join and filter conditions. Consider the following query, which returns all accounts opened by experienced tellers (hired prior to 2007) currently assigned to the Woburn branch:

```
mysql> SELECT a.account_id, a.cust_id, a.open_date, a.product_cd

-> FROM account a, branch b, employee e

-> WHERE a.open_emp_id = e.emp_id

->   AND e.start_date < '2007-01-01'

->   AND e.assigned_branch_id = b.branch_id

->   AND (e.title = 'Teller' OR e.title = 'Head Teller')

->   AND b.name = 'Woburn Branch';
```

account_id	cust_id	open_date	product_cd
1	1	2000-01-15	CHK
2	1	2000-01-15	SAV

3	1	2004-06-30	CD
4	2	2001-03-12	CHK
5	2	2001-03-12	SAV
17	7	2004-01-12	CD
27	11	2004-03-22	BUS

7 rows in set (0.00 sec)

Joining Three or More Tables

Joining three tables is similar to joining two tables, but with one slight wrinkle. With a two-table join, there are two tables and one join type in the from clause, and a single on subclause to define how the tables are joined. With a three-table join, there are three tables and two join types in the from clause, and two on subclauses. Here's another example of a query with a two-table join:

```
mysql> SELECT a.account_id, c.fed_id

-> FROM account a INNER JOIN customer c

->   ON a.cust_id = c.cust_id

-> WHERE c.cust_type_cd = 'B';
```

account_id	fed_id
24	04-1111111
25	04-1111111
27	04-2222222
28	04-3333333

29	04-4444444

5 rows in set (0.15 sec)

This query, which returns the account ID and federal tax number for all business accounts, should look fairly straightforward by now. If, however, you add the employee table to the query to also retrieve the name of the teller who opened each account, it looks as follows:

```
mysql> SELECT a.account_id, c.fed_id, e.fname, e.lname

-> FROM account a INNER JOIN customer c

->   ON a.cust_id = c.cust_id

->   INNER JOIN employee e

->   ON a.open_emp_id = e.emp_id

-> WHERE c.cust_type_cd = 'B';
```

account_id	fed_id	fname	lname
24	04-1111111	Theresa	Markham
25	04-1111111	Theresa	Markham
27	04-2222222	Paula	Roberts
28	04-3333333	Theresa	Markham
29	04-4444444	John	Blake

Now three tables, two join types, and two on subclauses are listed in the from clause, so things have gotten quite a bit busier. At first glance, the order in which the tables are named might cause you to think that the employee table is being joined to the customer table, since the account

table is named first, followed by the customer table, and then the employee table. If you switch the order in which the first two tables appear, however, you will get the exact same results:

```
mysql> SELECT a.account_id, c.fed_id, e.fname, e.lname

-> FROM customer c INNER JOIN account a

->   ON a.cust_id = c.cust_id

->   INNER JOIN employee e

->   ON a.open_emp_id = e.emp_id

-> WHERE c.cust_type_cd = 'B';
```

account_id	fed_id	fname	lname
24	04-1111111	Theresa	Markham
25	04-1111111	Theresa	Markham
27	04-2222222	Paula	Roberts
28	04-3333333	Theresa	Markham
29	04-4444444	John	Blake

5 rows in set (0.09 sec)

Using Subqueries as Table

You have already seen several examples of queries that use three tables, but there is one variation worth mentioning: what to do if some of the data sets are generated by subqueries. Here’s another version of an earlier query (find all accounts opened by experienced tellers currently assigned to the Woburn branch) that joins the account table to subqueries against the branch and employee tables:

```
SELECT a.account_id, a.cust_id, a.open_date, a.product_cd

FROM account a INNER JOIN

(SELECT emp_id, assigned_branch_id FROM employee

WHERE start_date < '2007-01-01'

AND (title = 'Teller' OR title = 'Head Teller')) e

ON a.open_emp_id = e.emp_id

INNER JOIN 9    (SELECT branch_id

FROM branch

WHERE name = 'Woburn Branch') b

ON e.assigned_branch_id = b.branch_id;
```

The first subquery, which starts on line 3 and is given the alias e, finds all experienced tellers. The second subquery, which starts on line 9 and is given the alias b, finds the ID of the Woburn branch. First, the account table is joined to the experienced-teller subquery using the employee ID and then the table that results is joined to the Woburn branch subquery using the branch ID. The results are the same as those of the previous version of the query (try it and see for yourself), but the queries look very different from one another. There isn't really anything shocking here, but it might take a minute to figure out what's going on.

Notice, for example, the lack of a where clause in the main query; since all the filter conditions are against the employee and branch tables, the filter conditions are all inside the subqueries, so there is no need for any filter conditions in the main query. One way to visualize what is going on is to run the subqueries and look at the result sets. Here are the results of the first subquery against the employee table:

Using the Same Table Twice

If you are joining multiple tables, you might find that you need to join the same table more than once. In the sample database, for example, there are foreign keys to the branch table from both the account table (the branch at which the account was opened) and the employee table (the branch at which the employee works). If you want to include both branches in your result set, you can include the branch table twice in the from clause, joined once to the employee table and once to the account table.

For this to work, you will need to give each instance of the branch table a different alias so that the server knows which one you are referring to in the various clauses, as in:

```
mysql> SELECT a.account_id, e.emp_id,

-> b_a.name open_branch, b_e.name emp_branch

-> FROM account a INNER JOIN branch b_a

-> ON a.open_branch_id = b_a.branch_id

-> INNER JOIN employee e

-> ON a.open_emp_id = e.emp_id

-> INNER JOIN branch b_e -> ON e.assigned_branch_id = b_e.branch_id

-> WHERE a.product_cd = 'CHK';
```

account_id	emp_id	open_branch	emp_branch
10	1	Headquarters	Headquarters
14	1	Headquarters	Headquarters

21	1	Headquarters	Headquarters
1	10	Woburn Branch	Woburn Branch
4	10	Woburn Branch	Woburn Branch
7	13	Quincy Branch	Quincy Branch
13	16	So. NH Branch	So. NH Branch
18	16	So. NH Branch	So. NH Branch
24	16	So. NH Branch	So. NH Branch
28	16	So. NH Branch	So. NH Branch

10 rows in set (0.16 sec)

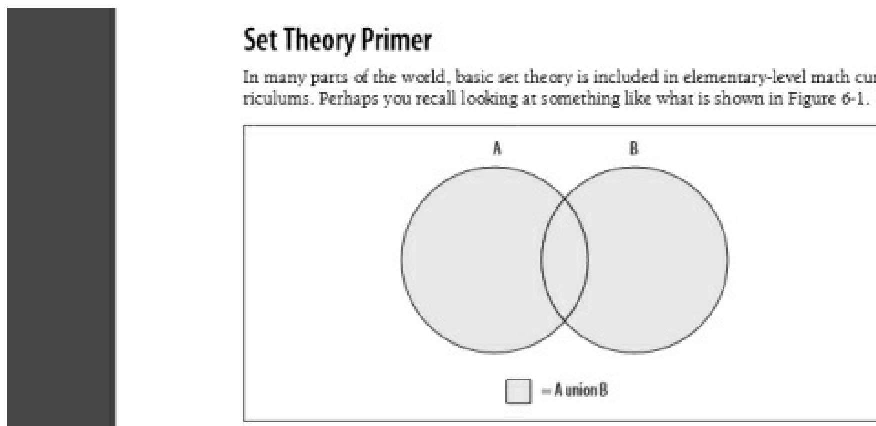
This query shows who opened each checking account, what branch it was opened at, and to which branch the employee who opened the account is currently assigned. The branch table is included twice, with aliases b_a and b_e. By assigning different aliases

Working with Sets

Although you can interact with the data in a database one row at a time, relational databases are really all about sets. You have seen how you can create tables via queries or subqueries, make them persistent via insert statements, and bring them together via joins; this chapter explores how you can combine multiple tables using various set operators.

Set Theory Primer

In many parts of the world, basic set theory is included in elementary-level math curriculums.


Set Theory Primer

In many parts of the world, basic set theory is included in elementary-level math curriculums. Perhaps you recall looking at something like what is shown in Figure 6-1.

The shaded area in the diagram below represents the union of sets A and B, which is the combination of the two sets (with any overlapping regions included only once). Is this starting to look familiar? If so, then you'll finally get a chance to put that knowledge to use; if not, don't worry, because it's easy to visualize using a couple of diagrams.

Using circles to represent two data sets (A and B), imagine a subset of data that is common to both sets; this common data is represented by the overlapping area shown. Since set theory is rather uninteresting without an overlap between data sets, I use the same diagram to illustrate each

set operation. There is another set operation that is concerned only with the overlap between two data sets; this operation is known as the intersection and is demonstrated.

The data set generated by the intersection of sets A and B is just the area of overlap between the two sets. If the two sets have no overlap, then the intersection operation yields the empty set.

Set Theory in Practice

The circles used in the previous section's diagrams to represent data sets don't convey anything about what the data sets comprise. When dealing with actual data, however,there is a need to describe the composition of the data sets involved if they are to be combined. Imagine, for example, what would happen if you tried to generate the union of the product table and the customer table, whose table definitions are as follows:

```
mysql> DESC product;
```

Field	Type	Null	Key	Default	Extra
product_cd	varchar(10)	NO	PRI		
name	varchar(50)	NO	NULL		
product_type_cd	varchar(10)	NO	NULL		
date_offered	date	YES	NULL		
date_retired	date	YES	NULL		

5 rows in set (0.23 sec)

```
mysql> DESC customer;
```

Field	Type	Null	Key	Default	Extra
cust_id	int(10) unsigned	No	PRI	NULL	auto_increment
fed_id	varchar(12)	No		NULL	

cust_type_cd	enum('I','B')	No		NULL	
address	varchar(30)	YES		NULL	
city	varchar(20)	YES		NULL	
state	varchar(20)	YES		NULL	
postal_code	varchar(10)	YES		NULL	

7 rows in set (0.04 sec)

When combined, the first column in the table that results would be the combination of the product.product_cd and customer.cust_id columns, the second column would be the combination of the product.name and customer.fed_id columns, and so forth. While some of the column pairs are easy to combine (e.g., two numeric columns), it is unclear how other column pairs should be combined, such as a numeric column with a string column or a string column with a date column.

Additionally, the sixth and seventh columns of the combined tables would include data from only the customer table's sixth and seventh columns, since the product table has only five columns. Clearly, there needs to be some commonality between two tables that you wish to combine. Therefore, when performing set operations on two data sets, the following guidelines must apply:

Both data sets must have the same number of columns.

The data types of each column across the two data sets must be the same (or the server must be able to convert one to the other). With these rules in place, it is easier to envision what "overlapping data" means in practice; each column pair from the two sets being combined must contain the same string, number, or date for rows in the two tables to be considered the same.

Set Operators

The SQL language includes three set operators that allow you to perform each of the various set operations described earlier in the chapter.

Additionally, each set operator has two flavors, one that includes duplicates and another that removes duplicates (but not necessarily all of the duplicates). The following subsections define each operator and demonstrate how they are used.

The Union Operator

The union and union all operators allow you to combine multiple data sets. The difference between the two is that union sorts the combined set and removes duplicates, whereas union all does not. With union all, the number of rows in the final data set will always equal the sum of the number of rows in the sets being combined. This operation is the simplest set operation to perform (from the server's point of view), since there is no need for the server to check for overlapping data. The following example demonstrates how you can use the union all operator to generate a full set of customer data from the two customer subtype tables:

```
mysql> SELECT 'IND' type_cd, cust_id, lname name

-> FROM individual

-> UNION ALL

-> SELECT 'BUS' type_cd, cust_id, name

-> FROM business;
```

type_cd	cust_id	name
IND	1	Hadley
IND	2	Tingley
IND	3	Tucker
IND	4	Hayward

IND	5	Frasier
IND	6	Spencer
IND	7	Young
IND	8	Blake
IND	9	Farley
BUS	10	Chilton Engineering
BUS	11	Northeast Cooling Inc.
BUS	12	Superior Auto Body
BUS	13	AAA Insurance Inc.

13 rows in set (0.04 sec)

The query returns all 13 customers, with nine rows coming from the individual table and the other four coming from the business table. While the business table includes a single column to hold the company name, the individual table includes two name columns, one each for the person's first and last names. In this case, I chose to include only the last name from the individual table. Just to drive home the point that the union all operator doesn't remove duplicates, here's the same query as the previous example but with an additional query against the business table:

```
mysql> SELECT 'IND' type_cd, cust_id, lname name

-> FROM individual

-> UNION ALL

-> SELECT 'BUS' type_cd, cust_id, name
```

```
-> FROM business

-> UNION ALL

-> SELECT 'BUS' type_cd, cust_id, name

-> FROM business;
```

type_cd	cust_id	name
IND	1	Hadley
IND	2	Tingley
IND	3	Tucker
IND	4	Hayward
IND	5	Frasier
IND	6	Spencer
IND	7	Young
IND	8	Blake
IND	9	Farley
BUS	10	Chilton Engineering
BUS	11	Northeast Cooling Inc.
BUS	12	Superior Auto Body
BUS	13	AAA Insurance Inc.
BUS	10	Chilton Engineering

BUS	11	Northeast Cooling Inc.
BUS	12	Superior Auto Body
BUS	13	AAA Insurance Inc.Auto Body

17 rows in set (0.01 sec)

The Intersect Operator

The ANSI SQL specification includes the intersect operator for performing intersections. Unfortunately, version 6.0 of MySQL does not implement the intersect operator. If you are using Oracle or SQL Server 2008, you will be able to use intersect; since I am using MySQL for all examples in this book, however, the result sets for the example queries in this section are fabricated and cannot be executed with any versions up to and including version 6.0.

I also refrain from showing the MySQL prompt (mysql>), since the statements are not being executed by the MySQL server. If the two queries in a compound query return nonoverlapping data sets, then the intersection will be an empty set. Consider the following query:

```
SELECT emp_id, fname, lname

FROM employee INTERSECT

SELECT cust_id, fname, lname

FROM individual;
```

Empty set (0.04 sec)

The first query returns the ID and name of each employee, while the second query returns the ID and name of each customer. These sets are completely nonoverlapping, so the intersection of the two sets yields the empty set. The next step is to identify two queries that do have overlapping data and then apply the intersect operator. For this purpose, I use the same query used to demonstrate the difference between union

and union all, except this time using intersect:

```
SELECT emp_id FROM employee

WHERE assigned_branch_id = 2  AND (title = 'Teller' OR title = 'Head Teller') INTERSECT SELECT DISTINCT open_emp_id FROM account

WHERE open_branch_id = 2;

+--------+ | emp_id | +--------+ |     10 | +--------+
```

1 row in set (0.01 sec)

The intersection of these two queries yields employee ID 10, which is the only value found in both queries' result sets. Along with the intersect operator, which removes any duplicate rows found in the overlapping region, the ANSI SQL specification calls for an intersect all operator, which does not remove duplicates. The only database server that currently implements the intersect all operator is IBM's DB2 Universal Server.

The Except Operator

The ANSI SQL specification includes the except operator for performing the except operation. Once again, unfortunately, version 6.0 of MySQL does not implement the except operator, so the same rules apply for this section as for the previous section.

If you are using Oracle Database, you will need to use the non-ANSIcompliant minus operator instead.

The except operator returns the first table minus any overlap with the second table. Here's the example from the previous section, but using except instead of intersect:

```
SELECT emp_id FROM employee

WHERE assigned_branch_id = 2  AND (title = 'Teller' OR title = 'Head Teller')
```

```
EXCEPT SELECT DISTINCT open_emp_id
FROM account
WHERE open_branch_id = 2;
+--------+
| emp_id |
+--------+
|     11 |
|     12 |
+--------+
```

2 rows in set (0.01 sec)

In this version of the query, the result set consists of the three rows from the first query minus employee ID 10, which is found in the result sets from both queries. There is also an except all operator specified in the ANSI SQL specification, but once again, only IBM's DB2 Universal Server has implemented the except all operator.

The except all operator is a bit tricky, so here's an example to demonstrate how duplicate data is handled. Let's say you have two data sets that look as follows:

```
Set A
+--------+
| emp_id |
+--------+
```

```
|    10 |

|    11 |

|    12 |

|    10 |

|    10 |

+--------+
```

Set Operation Rules

The following sections outline some rules that you must follow when working with compound queries.

Sorting Compound Query Results

If you want the results of your compound query to be sorted, you can add an order by clause after the last query. When specifying column names in the order by clause, you will need to choose from the column names in the first query of the compound query. Frequently, the column names are the same for both queries in a compound query, but this does not need to be the case, as demonstrated by the following:

```
mysql> SELECT emp_id, assigned_branch_id

-> FROM employee

-> WHERE title = 'Teller'

-> UNION

-> SELECT open_emp_id, open_branch_id
```

```
-> FROM account

-> WHERE product_cd = 'SAV'

-> ORDER BY emp_id;
```

emp_id	assigned_branch_id
1	1
7	1
8	1
9	1
10	2
11	2
12	2
14	3
15	3
16	4
17	4
18	4

12 rows in set (0.04 sec)

The column names specified in the two queries are different in this example. If you specify a column name from the second query in your order by clause, you will see the following error:

```
mysql> SELECT emp_id, assigned_branch_id

-> FROM employee

-> WHERE title = 'Teller'

-> UNION

-> SELECT open_emp_id, open_branch_id

-> FROM account

-> WHERE product_cd = 'SAV'

-> ORDER BY open_emp_id;
```

ERROR 1054 (42S22):

Unknown column 'open_emp_id' in 'order clause' I recommend giving the columns in both queries identical column aliases in order to avoid this issue.

Set Operation Precedence

If your compound query contains more than two queries using different set operators, you need to think about the order in which to place the queries in your compound statement to achieve the desired results. Consider the following three-query compound statement:

```
mysql> SELECT cust_id

-> FROM account

-> WHERE product_cd IN ('SAV', 'MM')

-> UNION ALL
```

```
-> SELECT a.cust_id

-> FROM account a INNER JOIN branch b

->   ON a.open_branch_id = b.branch_id

-> WHERE b.name = 'Woburn Branch'

-> UNION

-> SELECT cust_id

-> FROM account

-> WHERE avail_balance BETWEEN 500 AND 2500;
```

Data Generation, Conversion, and Manipulation

As I mentioned in the above, this book strives to teach generic SQL techniques that can be applied across multiple database servers. This chapter, however, deals with the generation, conversion, and manipulation of string, numeric, and temporal data, and the SQL language does not include commands covering this functionality. Rather, builtin functions are used to facilitate data generation, conversion, and manipulation, and while the SQL standard does specify some functions, the database vendors often do not comply with the function specifications.

Therefore, my approach for this chapter is to show you some of the common ways in which data is manipulated within SQL statements, and then demonstrate some of the built-in functions implemented by Microsoft SQL Server, Oracle Database, and MySQL. Along with reading this chapter, I strongly recommend you purchase a reference guide covering all the functions implemented by your server.

Working with String Data

When working with string data, you will be using one of the following character data types: CHAR Holds fixed-length, blank-padded strings. MySQL allows CHAR values up to 255 characters in length, Oracle Database permits up to 2,000 characters, and SQL Server allows up to 8,000 characters.

Varchar Holds variable-length strings. MySQL permits up to 65,535 characters in a varchar column, Oracle Database (via the varchar2 type) allows up to 4,000 characters, and SQL Server allows up to 8,000 characters. text (MySQL and SQL Server) or CLOB (Character Large

Object; Oracle Database) Holds very large variable-length strings (generally referred to as documents in this context). MySQL has multiple text types (tinytext, text, mediumtext, and long text) for documents up to 4 GB in size. SQL Server has a single text type for documents up to 2 GB in size, and Oracle Database includes the CLOB data type, which can hold documents up to a whopping 128 TB. SQL Server 2005 also includes the varchar (max) data type and recommends its use instead of the text type, which will be removed from the server in some future release. To demonstrate how you can use these various types, I use the following table for some of the examples in this section: CREATE TABLE string_tbl (char_fld CHAR (30), vchar_fld VARCHAR(30), text_fld TEXT); The next two subsections show how you can generate and manipulate string data.

String Generation

The simplest way to populate a character column is to enclose a string in quotes, as in:

```
mysql> INSERT INTO string_tbl (char_fld, vchar_fld, text_fld)

-> VALUES ('This is char data',

->   'This is varchar data',

->   'This is text data');
```

Query OK, 1 row affected (0.00 sec)

When inserting string data into a table, remember that if the length of the string exceeds the maximum size for the character column (either the designated maximum or the maximum allowed for the data type), the server will throw an exception. Although this is the default behavior for all three servers, you can configure MySQL and SQL Server to silently truncate the string instead of throwing an exception.

To demonstrate how MySQL handles this situation, the following update statement attempts to modify the vchar_fld column, whose maximum length is defined as 30, with a string that is 46 characters in length:

```
mysql> UPDATE string_tbl

-> SET vchar_fld =
```

'This is a piece of extremely long varchar data'; ERROR 1406 (22001): Data too long for column 'vchar_fld' at row 1 With MySQL 6.0, the default behavior is now “strict” mode, which means that exceptions are thrown when problems arise, whereas in older versions of the server the string would have been truncated and a warning issued. If you would rather have the engine truncate the string and issue a warning instead of raising an exception, you can opt to be in ANSI mode. The following example shows how to check which mode you are in, and then how to change the mode using the SET command:

```
mysql> SELECT @@session.sql_mode;

+-----------------------------------------------------------+

| @@session.sql_mode                                        |

+-----------------------------------------------------------+

|
STRICT_TRANS_TABLES,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTIO
|

+-----------------------------------------------------------+

1 row in set (0.00 sec)
mysql> SET sql_mode='ansi';
Query OK, 0 rows affected (0.08 sec)
mysql> SELECT @@session.sql_mode;

+-------------------------------------------------------+
```

```
| @@session.sql_mode                                       |
+----------------------------------------------------------+
| REAL_AS_FLOAT,PIPES_AS_CONCAT,ANSI_QUOTES,IGNORE_SPACE,ANSI |
+----------------------------------------------------------+
```

1 row in set (0.00 sec)

If you rerun the previous UPDATE statement, you will find that the column has been modified, but the following warning is generated:

```
mysql> SHOW WARNINGS;
+---------+------+---------------------------------------------+
| Level   | Code | Message                                     |
+---------+------+---------------------------------------------+
| Warning | 1265 | Data truncated for column 'vchar_fld' at row 1
| +---------+------+---------------------------------------------+
1 row in set (0.00 sec)
```

If you retrieve the vchar_fld column, you will see that the string has indeed been truncated:

```
mysql> SELECT vchar_fld
-> FROM string_tbl;
+------------------------------+
```

```
| vchar_fld                    |

+--------------------------------+

| This is a piece of extremely l

| +--------------------------------+
```

1 row in set (0.05 sec)

As you can see, only the first 30 characters of the 46-character string made it into the vchar_fld column. The best way to avoid string truncation (or exceptions, in the case of Oracle Database or MySQL in strict mode) when working with varchar columns is to set the upper limit of a column to a high enough value to handle the longest strings that might be stored in the column (keeping in mind that the server allocates only enough space to store the string, so it is not wasteful to set a high upper limit for varchar columns).

String Manipulation

Each database server includes many built-in functions for manipulating strings. This section explores two types of string functions: those that return numbers and those that return strings. Before I begin, however, I reset the data in the string_tbl table to the following:

```
mysql> DELETE FROM string_tbl;

Query OK, 1 row affected (0.02 sec)

mysql> INSERT INTO string_tbl (char_fld, vchar_fld, text_fld)

-> VALUES ('This string is 28 characters',

->   'This string is 28 characters',

->   'This string is 28 characters');
```

Query OK, 1 row affected (0.00 sec)

String functions that return numbers Of the string functions that return numbers, one of the most commonly used is the length() function, which returns the number of characters in the string (SQL Server users will need to use the len() function). The following query applies the length() function to each column in the string_tbl table:

```
mysql> SELECT LENGTH(char_fld) char_length,

->   LENGTH(vchar_fld) varchar_length,

->   LENGTH(text_fld) text_length

-> FROM string_tbl;

+-------------+----------------+-------------+

| char_length | varchar_length | text_length |

+-------------+----------------+-------------+

|          28 |             28 |          28 |

+-------------+----------------+-------------+
```

1 row in set (0.00 sec)

While the lengths of the varchar and text columns are as expected, you might have expected the length of the char column to be 30, since I told you that strings stored in char columns are right-padded with spaces. The MySQL server removes trailing spaces from char data when it is retrieved, however, so you will see the same results from all string functions regardless of the type of column in which the strings are stored. Along with finding the length of a string, you might want to find the location of a substring within a string. For example, if you want to find the position at which the string 'characters' appears in the vchar_fld column, you could use the position() function, as demonstrated by the

following:

```
mysql> SELECT POSITION('characters' IN vchar_fld)

-> FROM string_tbl;

+------------------------------------+

| POSITION('characters' IN vchar_fld) |

+------------------------------------+

|                         19 |

+------------------------------------+
```

1 row in set (0.12 sec)

If the substring cannot be found, the position() function returns 0.

For those of you who program in a language such as C or C++, where the first element of an array is at position 0, remember when working with databases that the first character in a string is at position 1.

A return value of 0 from position() indicates that the substring could not be found, not that the substring was found at the first position in the string.

If you want to start your search at something other than the first character of your target string, you will need to use the locate() function, which is similar to the position() function except that it allows an optional third parameter, which is used to define the search's start position. The locate() function is also proprietary, whereas the position() function is part of the SQL:2003 standard. Here's an example asking for the position of the string 'is' starting at the fifth character in the vchar_fld column:

```
mysql> SELECT LOCATE('is', vchar_fld, 5)
```

```
-> FROM string_tbl;
+---------------------------+
| LOCATE('is', vchar_fld, 5) |
+---------------------------+
|                        13 |
+---------------------------+
```

1 row in set (0.02 sec)

Oracle Database does not include the position() or locate() function, but it does include the instr() function, which mimics the position() function when provided with two arguments and mimics the locate() function when provided with three arguments. SQL Server also doesn't include a position() or locate() function, but it does include the charindx() function, which also accepts either two or three arguments similar to Oracle's instr() function.

Another function that takes strings as arguments and returns numbers is the string comparison function strcmp(). Strcmp(), which is implemented only by MySQL and has no analog in Oracle Database or SQL Server, takes two strings as arguments, and returns one of the following:

- −1 if the first string comes before the second string in sort order
- 0 if the strings are identical
- 1 if the first string comes after the second string in sort order

To illustrate how the function works, I first show the sort order of five strings using a query, and then show how the strings compare to one another using strcmp(). Here are the five strings that I insert into the string_tbl table:

```
mysql> DELETE FROM string_tbl;
```

Query OK, 1 row affected (0.00 sec)

```
mysql> INSERT INTO string_tbl(vchar_fld) VALUES ('abcd');
```

Query OK, 1 row affected (0.03 sec)

```
mysql> INSERT INTO string_tbl(vchar_fld) VALUES ('xyz');
```

Query OK, 1 row affected (0.00 sec)

```
mysql> INSERT INTO string_tbl(vchar_fld) VALUES ('QRSTUV');
```

Query OK, 1 row affected (0.00 sec)

```
mysql> INSERT INTO string_tbl(vchar_fld) VALUES ('qrstuv');
```

Query OK, 1 row affected (0.00 sec)

```
mysql> INSERT INTO string_tbl(vchar_fld) VALUES ('12345');
```

Query OK, 1 row affected (0.00 sec)

Here are the five strings in their sort order:

```
mysql> SELECT vchar_fld

-> FROM string_tbl

-> ORDER BY vchar_fld;

+-----------+

| vchar_fld |

+-----------+ | 12345     |

| abcd      |

| QRSTUV    |
```

```
| qrstuv    |

| xyz       |

+-----------+
```

5 rows in set (0.00 sec)

The next query makes six comparisons among the five different strings:

```
mysql> SELECT STRCMP('12345','12345') 12345_12345,

->   STRCMP('abcd','xyz') abcd_xyz,

->   STRCMP('abcd','QRSTUV') abcd_QRSTUV,

->   STRCMP('qrstuv','QRSTUV') qrstuv_QRSTUV,

->   STRCMP('12345','xyz') 12345_xyz,

->   STRCMP('xyz','qrstuv') xyz_qrstuv;
```

12345_12345	abcd_xyz	abcd_QRSTUV	qrstuv_QRSTUV	12345_xyz	xyz_qrstuv
0	-1	-1	0	-1	1

1 row in set (0.00 sec)

Working with Numeric Data

Unlike string data (and temporal data, as you will see shortly), numeric data generation is quite straightforward. You can type a number, retrieve it from another column, or generate it via a calculation. All the usual arithmetic operators (+, -, *, /) are available for performing calculations, and parentheses may be used to dictate precedence, as in: mysql> SELECT (37 * 59) / (78 - (8 * 6));

```
+---------------------------+
```

```
| (37 * 59) / (78 - (8 * 6)) |

+----------------------------+

|                      72.77 |

+----------------------------+
```

1 row in set (0.00 sec)

As I mentioned above, the main concern when storing numeric data is that numbers might be rounded if they are larger than the specified size for a numeric column. For example, the number 9.96 will be rounded to 10.0 if stored in a column defined as float(3,1).

```
mysql> SELECT MOD(10,4);

+-----------+

| MOD(10,4) |

+-----------+

|         2 |

+-----------+
```

1 row in set (0.02 sec)

While the mod() function is typically used with integer arguments, with MySQL you can also use real numbers, as in:

```
mysql> SELECT MOD(22.75, 5);

+---------------+

| MOD(22.75, 5) |
```

```
+---------------+

|        2.75 |

+---------------+
```

1 row in set (0.02 sec)

SQL Server does not have a mod() function. Instead, the operator % is used for finding remainders. The expression 10 % 4 will therefore yield the value 2.

Controlling Number Precision

When working with floating-point numbers, you may not always want to interact with or display a number with its full precision. For example, you may store monetary transaction data with a precision to six decimal places, but you might want to round to the nearest hundredth for display purposes. Four functions are useful when limiting the precision of floating-point numbers:

- ceil()
- floor()
- round()
- truncate().

All three servers include these functions, although Oracle Database includes trunc() instead of truncate(), and SQL Server includes ceiling() instead of ceil(). The ceil() and floor() functions are used to round either up or down to the closest integer, as demonstrated by the following:

```
mysql> SELECT CEIL(72.445),

FLOOR(72.445);

+--------------+---------------+

| CEIL(72.445) | FLOOR(72.445) |
```

```
+-------------+--------------+
|        73 |         72 |
+-------------+--------------+
```

1 row in set (0.06 sec)

Thus, any number between 72 and 73 will be evaluated as 73 by the ceil() function and 72 by the floor() function. Remember that ceil() will round up even if the decimal portion of a number is very small, and floor() will round down even if the decimal portion is quite significant, as in:

```
mysql> SELECT CEIL(72.000000001),
FLOOR(72.999999999);
+-------------------+-------------------+
| CEIL(72.000000001) | FLOOR(72.999999999) |
+-------------------+-------------------+
|            73 |             72 |
+-------------------+-------------------+
```

1 row in set (0.00 sec)

If this is a bit too severe for your application, you can use the round() function to round up or down from the midpoint between two integers, as in:

```
mysql> SELECT ROUND(72.49999), ROUND(72.5), ROUND(72.50001);
```

ROUND(72.49999)	ROUND(72.5)	ROUND(72.50001)

72	73	73

1 row in set (0.00 sec)

Using round(), any number whose decimal portion is halfway or more between two integers will be rounded up, whereas the number will be rounded down if the decimal portion is anything less than halfway between the two integers.

Most of the time, you will want to keep at least some part of the decimal portion of a number rather than rounding to the nearest integer; the round() function allows an optional second argument to specify how many digits to the right of the decimal place to round to. The next example shows how you can use the second argument to round the number 72.0909 to one, two, and three decimal places:

```
mysql> SELECT ROUND(72.0909, 1), ROUND(72.0909, 2), ROUND(72.0909, 3);
```

ROUND(72.0909, 1)	ROUND(72.0909, 2)	ROUND(72.0909, 3)
72.1	72.09	72.091

1 row in set (0.00 sec)

Like the round() function, the truncate() function allows an optional second argument to specify the number of digits to the right of the decimal, but truncate() simply discards the unwanted digits without rounding. The next example shows how the number 72.0909 would be truncated to one, two, and three decimal places:

```
mysql> SELECT TRUNCATE(72.0909, 1), TRUNCATE(72.0909, 2),

-> TRUNCATE(72.0909, 3);
```

TRUNCATE(72.0909, 1)	TRUNCATE(72.0909, 2)	TRUNCATE(72.0909, 3)
72.0	72.09	72.090

1 row in set (0.00 sec)

SQL Server does not include a truncate() function. Instead, the round()

function allows for an optional third argument which, if present and nonzero, calls for the number to be truncated rather than rounded.

Handling Signed Data

If you are working with numeric columns that allow negative values showed how a numeric column may be labeled unsigned, meaning that only positive numbers are allowed), several numeric functions might be of use. Let's say, for example, that you are asked to generate a report showing the current status of each bank account. The following query returns three columns useful for generating the report:

```
mysql> SELECT account_id, SIGN(avail_balance), ABS(avail_balance)

-> FROM account;
```

account_id	SIGN(avail_balance)	ABS(avail_balance)
1	1	1057.75
2	1	500.00
3	1	3000.00
4	1	2258.02
5	1	00.00
19	1	1500.00
20	1	23575.12
21	0	0.00
22	1	9345.55
23	1	38552.05
24	1	50000.00

24 rows in set (0.00 sec)

The second column uses the sign() function to return −1 if the account balance is negative, 0 if the account balance is zero, and 1 if the account balance is positive. The third column returns the absolute value of the account balance via the abs() function.

Working with Temporal Data Of the three types of data discussed in this chapter (character, numeric, and temporal), temporal data is the most involved when it comes to data generation and manipulation. Some of the complexity of temporal data is caused by the myriad ways in which a single date and time can be described. For example, the date on which I wrote this paragraph can be described in all the following ways:

- Wednesday, September 17, 2008
- 9/17/2008 2:14:56 P.M. EST
- 9/17/2008 19:14:56 GMT
- 2612008 (Julian format)
- Star date [−4] 85712.03 14:14:56 (Star Trek format)

Aggregate Functions

Aggregate functions perform a specific operation over all rows in a group. Although every database server has its own set of specialty aggregate functions, the common aggregate functions implemented by all major servers include:

Max() Returns the maximum value within a set Min() Returns the minimum value within a set Avg() Returns the average value across a set

Sum() Returns the sum of the values across a set Count() Returns the number of values in a set Here's a query that uses all of the common aggregate functions to analyze the available balances for all checking accounts:

```
mysql> SELECT MAX(avail_balance) max_balance,

->   MIN(avail_balance) min_balance,
```

```
->   AVG(avail_balance) avg_balance,

->   SUM(avail_balance) tot_balance,

->   COUNT(*) num_accounts

-> FROM account

-> WHERE product_cd = 'CHK';
```

max_balance	min_balance	avg_balance	tot_balance	num_accounts
38552.05	122.37	7300.800985	73008.01	10

1 row in set (0.09 sec)

The results from this query tell you that, across the 10 checking accounts in the account table, there is a maximum balance of $38,552.05, a minimum balance of $122.37, an average balance of $7,300.80, and a total balance across all 10 accounts of $73,008.01. Hopefully, this gives you an appreciation for the role of these aggregate functions; the next subsections further clarify how you can utilize these functions.

Implicit Versus Explicit Groups

In the previous example, every value returned by the query is generated by an aggregate function, and the aggregate functions are applied across the group of rows specified by the filter condition product_cd = 'CHK'. Since there is no group by clause, there is a single, implicit group (all rows returned by the query). In most cases, however, you will want to retrieve additional columns along with columns generated by aggregate functions. What if, for example, you wanted to extend the previous query to execute the same five aggregate functions for each product type, instead of just for checking accounts? For this query, you would want to retrieve the product_cd column along with the five aggregate functions, as in:

```
SELECT product_cd,  MAX(avail_balance) max_balance,
```

```
MIN(avail_balance) min_balance,

AVG(avail_balance) avg_balance,

SUM(avail_balance) tot_balance,

COUNT(*) num_accounts
```

FROM account; However, if you try to execute the query, you will receive the following error:

ERROR 1140 (42000):

Mixing of GROUP columns (MIN(),MAX(),COUNT(),...) with no GROUP columns is illegal if there is no GROUP BY clause While it may be obvious to you that you want the aggregate functions applied to each set of products found in the account table, this query fails because you have not explicitly specified how the data should be grouped. Therefore, you will need to add a group by clause to specify over which group of rows the aggregate functions should be applied:

```
mysql> SELECT product_cd,

->   MAX(avail_balance) max_balance,

->   MIN(avail_balance) min_balance,

->   AVG(avail_balance) avg_balance,

->   SUM(avail_balance) tot_balance,

->   COUNT(*) num_accts

-> FROM account

-> GROUP BY product_cd;
```


product_cd	max_balance	min_balance	avg_balance	tot_balance	num_accts
BUS	9345.55	0.00	4672.774902	9345.55	1
CD	10000.00	1500.00	4875.000000	19500.00	4
CHK	38552.05	122.37	7300.800985	73008.01	10
MM	9345.55	2212.50	5681.713216	17045.14	3
SAV	767.77	200.00	463.940002	1855.76	4
SBL	50000.00	50000.00	50000.00	50000.00	1

6 rows in set (0.00 sec)

With the inclusion of the group by clause, the server knows to group together rows having the same value in the product_cd column first and then to apply the five aggregate functions to each of the six groups.

Counting Distinct Values

When using the count() function to determine the number of members in each group, you have your choice of counting all members in the group, or counting only the distinct values for a column across all members of the group. For example, consider the following data, which shows the employee responsible for opening each account:

```
mysql> SELECT account_id, open_emp_id

-> FROM account

-> ORDER BY open_emp_id;
```

account_id	open_emp_id
8	1
9	1

10	1
12	1
13	1
17	1
18	1
19	1
1	10
2	10
3	10
4	10
5	10
14	10
22	10
6	13
7	13
24	13
11	16
15	16
16	16
20	16

21	16
23	16

24 rows in set (0.00 sec)

As you can see, multiple accounts were opened by four different employees (employee IDs 1, 10, 13, and 16). Let's say that, instead of performing a manual count, you want to create a query that counts the number of employees who have opened accounts. If you apply the count() function to the open_emp_id column, you will see the following results:

```
mysql> SELECT COUNT(open_emp_id)

-> FROM account;
```

COUNT(open_emp_id)
24

1 row in set (0.00 sec)

In this case, specifying the open_emp_id column as the column to be counted generates the same results as specifying count(*). If you want to count distinct values in the group rather than just counting the number of rows in the group, you need to specify the distinct keyword, as in:

```
mysql> SELECT COUNT(DISTINCT open_emp_id)

-> FROM account;
```

COUNT(DISTINCT open_emp_id)
4

1 row in set (0.00 sec)

By specifying distinct, therefore, the count() function examines the

values of a column for each member of the group in order to find and remove duplicates, rather than simply counting the number of values in the group.

Using Expressions

Along with using columns as arguments to aggregate functions, you can build expressions to use as arguments. For example, you may want to find the maximum value of pending deposits across all accounts, which is calculated by subtracting the available balance from the pending balance. You can achieve this via the following query:

```
mysql> SELECT MAX(pending_balance - avail_balance) max_uncleared

-> FROM account;
```

max_uncleared
660.00

1 row in set (0.00 sec)

While this example uses a fairly simple expression, expressions used as arguments to aggregate functions can be as complex as needed, as long as they return a number, string, or date. I show you how you can use case expressions with aggregate functions to determine whether a particular row should or should not be included in an aggregation.

How Nulls Are Handled

When performing aggregations, or, indeed, any type of numeric calculation, you should always consider how null values might affect the outcome of your calculation. To illustrate, I will build a simple table to hold numeric data and populate it with the set {1, 3, 5}:

```
mysql> CREATE TABLE number_tbl

->  (val SMALLINT); Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO number_tbl VALUES (1);
```

```
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO number_tbl VALUES (3);
```

Query OK, 1 row affected (0.00 sec)

```
mysql> INSERT INTO number_tbl VALUES (5);
```

Query OK, 1 row affected (0.00 sec) Consider the following query, which performs five aggregate functions on the set of numbers:

```
mysql> SELECT COUNT(*) num_rows,

->   COUNT(val) num_vals,

->   SUM(val) total,

->   MAX(val) max_val,

->   AVG(val) avg_val

-> FROM number_tbl;
```

num_rows	num_vals	total	max_val	avg_val
3	3	9	5	3.0000

1 row in set (0.08 sec)

The results are as you would expect: both count(*) and count(val) return the value 3, sum(val) returns the value 9, max(val) returns 5, and avg(val) returns 3. Next, I will add a null value to the number_tbl table and run the query again.

Subqueries

Subqueries are a powerful tool that you can use in all four SQL data statements. This explores in great detail the many uses of the subquery.

What Is a Subquery?

A subquery is a query contained within another SQL statement (which I refer to as the containing statement for the rest of this discussion). A subquery is always enclosed within parentheses, and it is usually executed prior to the containing statement. Like any query, a subquery returns a result set that may consist of:

- A single row with a single column
- Multiple rows with a single column
- Multiple rows and columns

The type of result set the subquery returns determines how it may be used and which operators the containing statement may use to interact with the data the subquery returns. When the containing statement has finished executing, the data returned by any subqueries is discarded, making a subquery act like a temporary table with statement scope (meaning that the server frees up any memory allocated to the subquery results after the SQL statement has finished execution). You already saw several examples of subqueries in earlier chapters, but here's a simple example to get started:

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance

-> FROM account

-> WHERE account_id = (SELECT MAX(account_id) FROM account);
```

```
+------------+------------+---------+---------------+
| account_id | product_cd | cust_id | avail_balance |
+------------+------------+---------+---------------+
|         29 | SBL        |      13 |      50000.00 |
+------------+------------+---------+---------------+
```

1 row in set (0.65 sec)

In this example, the subquery returns the maximum value found in the account_id column in the account table, and the containing statement then returns data about that account. If you are ever confused about what a subquery is doing, you can run the subquery by itself (without the parentheses) to see what it returns. Here's the subquery from the previous example:

```
mysql> SELECT MAX(account_id) FROM account;
+-----------------+
| MAX(account_id) |
+-----------------+
|              29 |
+-----------------+
```

1 row in set (0.00 sec)

So, the subquery returns a single row with a single column, which allows it to be used as one of the expressions in an equality condition (if the subquery returned two or more rows, it could be compared to something but could not be equal to anything, but more on this later). In this case,

you can take the value the subquery returned and substitute it into the righthand expression of the filter condition in the containing query, as in:

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance

-> FROM account

-> WHERE account_id = 29;

+------------+------------+---------+---------------+

| account_id | product_cd | cust_id | avail_balance |

+------------+------------+---------+---------------+

|         29 | SBL        |      13 |      50000.00 |

+------------+------------+---------+---------------+
```

1 row in set (0.02 sec)

The subquery is useful in this case because it allows you to retrieve information about the highest numbered account in a single query, rather than retrieving the maximum account_id using one query and then writing a second query to retrieve the desired data from the account table. As you will see, subqueries are useful in many other situations as well, and may become one of the most powerful tools in your SQL toolkit.

Subquery Types

Along with the differences noted previously regarding the type of result set a subquery returns (single row/column, single row/multicolumn, or multiple columns), you can use another factor to differentiate subqueries; some subqueries are completely selfcontained (called noncorrelated subqueries), while others reference columns from the containing statement (called correlated subqueries). The next several sections explore these two subquery types and show the different

operators that you can employ to interact with them.

Noncorrelated Subqueries

The example from earlier in the chapter is a noncorrelated subquery; it may be executed alone and does not reference anything from the containing statement. Most subqueries that you encounter will be of this type unless you are writing update or delete statements, which frequently make use of correlated subqueries (more on this later). Along with being noncorrelated, the example from earlier in the chapter also returns a table comprising a single row and column. This type of subquery is known as a scalar subquery and can appear on either side of a condition using the usual operators (=, <>, <, >, <=, >=). The next example shows how you can use a scalar subquery in an inequality condition:

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance

-> FROM account

-> WHERE open_emp_id <> (SELECT e.emp_id

->   FROM employee e INNER JOIN branch b

->     ON e.assigned_branch_id = b.branch_id

->   WHERE e.title = 'Head Teller' AND b.city = 'Woburn');

+------------+------------+---------+---------------+

| account_id | product_cd | cust_id | avail_balance |

+------------+------------+---------+---------------+

|          7 | CHK        |       3 |       1057.75 |

|          8 | MM         |       3 |       2212.50 |
```

```
|         10 | CHK        |       4 |       534.12 |
|         11 | SAV        |       4 |       767.77 |
|         12 | MM         |       4 |      5487.09 |
|         13 | CHK        |       5 |      2237.97 |
|         14 | CHK        |       6 |       122.37 |
|         15 | CD         |       6 |     10000.00 |
|         18 | CHK        |       8 |      3487.19 |
|         19 | SAV        |       8 |       387.99 |
|         21 | CHK        |       9 |       125.67 |
|         22 | MM         |       9 |      9345.55 |
|         23 | CD         |       9 |      1500.00 |
|         24 | CHK        |      10 |     23575.12 |
|         25 | BUS        |      10 |         0.00 |
|         28 | CHK        |      12 |     38552.05 |
|         29 | SBL        |      13 |     50000.00 |
+------------+------------+---------+--------------+
```

17 rows in set (0.86 sec)

This query returns data concerning all accounts that were not opened by the head teller at the Woburn branch (the subquery is written using the

assumption that there is only a single head teller at each branch). The subquery in this example is a bit more complex than in the previous example, in that it joins two tables and includes two filter conditions. Subqueries may be as simple or as complex as you need them to be, and they may utilize any and all the available query clauses (select, from, where, group by, having, and order by).

Multiple-Row, Single-Column Subqueries:

If your subquery returns more than one row, you will not be able to use it on one side of an equality condition, as the previous example demonstrated. However, there are four additional operators that you can use to build conditions with these types of subqueries.

The in and not in operators While you can't equate a single value to a set of values, you can check to see whether a single value can be found within a set of values. The next example, while it doesn't use a subquery, demonstrates how to build a condition that uses the in operator to search for a value within a set of values:

```
mysql> SELECT branch_id, name, city

-> FROM branch

-> WHERE name IN ('Headquarters', 'Quincy Branch');

+-----------+---------------+---------+

| branch_id | name          | city    |

+-----------+---------------+---------+

|         1 | Headquarters  | Waltham |

|         3 | Quincy Branch | Quincy  |

+-----------+---------------+---------+
```

2 rows in set (0.03 sec)

The expression on the lefthand side of the condition is the name column, while the righthand side of the condition is a set of strings. The in operator checks to see whether either of the strings can be found in the name column; if so, the condition is met and the row is added to the result set. You could achieve the same results using two equality conditions, as in:

```
mysql> SELECT branch_id, name, city

-> FROM branch

-> WHERE name = 'Headquarters' OR name = 'Quincy Branch';

+-----------+---------------+---------+

| branch_id | name          | city    |

+-----------+---------------+---------+

|         1 | Headquarters  | Waltham |

|         3 | Quincy Branch | Quincy  |

+-----------+---------------+---------+
```

2 rows in set (0.01 sec)

While this approach seems reasonable when the set contains only two expressions, it is easy to see why a single condition using the in operator would be preferable if the set contained dozens (or hundreds, thousands, etc.) of values.

Multicolumn Subqueries

So far, all of the subquery examples in this chapter have returned a single column and one or more rows. In certain situations, however, you can use subqueries that return two or more columns. To show the utility of

multiple-column subqueries, it might help to look first at an example that uses multiple, single-column subqueries:

```
mysql> SELECT account_id, product_cd, cust_id

-> FROM account

-> WHERE open_branch_id = (SELECT branch_id

->   FROM branch

->   WHERE name = 'Woburn Branch')

->   AND open_emp_id IN (SELECT emp_id

->   FROM employee

->   WHERE title = 'Teller' OR title = 'Head Teller');

+------------+------------+---------+

| account_id | product_cd | cust_id |

+------------+------------+---------+

|          1 | CHK        |       1 |

|          2 | SAV        |       1 |

|          3 | CD         |       1 |

|          4 | CHK        |       2 |

|          5 | SAV        |       2 |
```

```
|        17 | CD         |       7 |

|        27 | BUS        |      11 |

+------------+------------+---------+
```

7 rows in set (0.09 sec)

This query uses two subqueries to identify the ID of the Woburn branch and the IDs of all bank tellers, and the containing query then uses this information to retrieve all checking accounts opened by a teller at the Woburn branch. However, since the employee table includes information about which branch each employee is assigned to, you can achieve the same results by comparing both the account.open_branch_id and account.open_emp_id columns to a single subquery against the employee and branch tables. To do so, your filter condition must name both columns from the account table surrounded by parentheses and in the same order as returned by the subquery, as in:

```
mysql> SELECT account_id, product_cd, cust_id

-> FROM account

-> WHERE (open_branch_id, open_emp_id) IN

->  (SELECT b.branch_id, e.emp_id

->   FROM branch b INNER JOIN employee e

->     ON b.branch_id = e.assigned_branch_id

->   WHERE b.name = 'Woburn Branch'

->     AND (e.title = 'Teller' OR e.title = 'Head Teller'));

+------------+------------+---------+
```

```
| account_id | product_cd | cust_id |
+------------+------------+---------+
|          1 | CHK        |       1 |
|          2 | SAV        |       1 |
|          3 | CD         |       1 |
|          4 | CHK        |       2 |
|          5 | SAV        |       2 |
|         17 | CD         |       7 |
|         27 | BUS        |      11 |
+------------+------------+---------+
```

7 rows in set (0.00 sec)

Correlated Subqueries

All of the subqueries shown thus far have been independent of their containing statements, meaning that you can execute them by themselves and inspect the results. A correlated subquery, on the other hand, is dependent on its containing statement from which it references one or more columns. Unlike a noncorrelated subquery, a correlated subquery is not executed once prior to execution of the containing statement; instead, the correlated subquery is executed once for each candidate row (rows that might be included in the final results). For example, the following query uses a correlated subquery to count the number of accounts for each customer, and the containing query then retrieves those customers having exactly two accounts:

```
mysql> SELECT c.cust_id, c.cust_type_cd, c.city
```

```
-> FROM customer c

-> WHERE 2 = (SELECT COUNT(*)

->   FROM account a

->   WHERE a.cust_id = c.cust_id);
```

cust_id	cust_type_cd	city
2	I	Woburn
3	I	Quincy
6	I	Waltham
8	I	Salem
10	B	Salem

5 rows in set (0.01 sec)

The reference to c.cust_id at the very end of the subquery is what makes the subquery correlated; the containing query must supply values for c.cust_id for the subquery to execute. In this case, the containing query retrieves all 13 rows from the customer table and executes the subquery once for each customer, passing in the appropriate customer ID for each execution. If the subquery returns the value 2, then the filter condition is met and the row is added to the result set.

The exists Operator

While you will often see correlated subqueries used in equality and range conditions, the most common operator used to build conditions that utilize correlated subqueries is the exists operator. You use the exists operator when you want to identify that a relationship exists without regard for the quantity; for example, the following query finds all the accounts for which a transaction was posted on a particular day, without regard for how many transactions were posted:

```
SELECT a.account_id, a.product_cd, a.cust_id, a.avail_balance

FROM account a

WHERE EXISTS (SELECT 1  FROM transaction t  WHERE t.account_id = a.account_id   AND t.txn_date = '2008-09-22');
```

Using the exists operator, your subquery can return zero, one, or many rows, and the condition simply checks whether the subquery returned any rows. If you look at the select clause of the subquery, you will see that it consists of a single literal (1); since the condition in the containing query only needs to know how many rows have been returned, the actual data the subquery returned is irrelevant. Your subquery can return whatever strikes your fancy, as demonstrated next:

```
SELECT a.account_id, a.product_cd, a.cust_id, a.avail_balance FROM
account a WHERE EXISTS (SELECT t.txn_id, 'hello', 3.1415927  FROM
transaction t  WHERE t.account_id = a.account_id    AND t.txn_date =
'2008-09-22');
```

However, the convention is to specify either select 1 or select * when using exists

Data Manipulation Using Correlated Subqueries

All of the examples thus far in the chapter have been select statements, but don't think that means that subqueries aren't useful in other SQL statements. Subqueries are used heavily in update, delete, and insert statements as well, with correlated subqueries appearing frequently in update and delete statements. Here's an example of a correlated subquery used to modify the last_activity_date column in the account table:

```
UPDATE account a SET a.last_activity_date = (SELECT MAX(t.txn_date)

FROM transaction t  WHERE t.account_id = a.account_id);
```

This statement modifies every row in the account table (since there is no where clause) by finding the latest transaction date for each account. While it seems reasonable to expect that every account will have at least one transaction linked to it, it would be best to check whether an account has any transactions before attempting to update the last_activity_date column; otherwise, the column will be set to null, since the subquery would return no rows. Here's another version of the update statement, this time employing a where clause with a second correlated subquery:

```
UPDATE account a SET a.last_activity_date = (SELECT MAX(t.txn_date)

FROM transaction t  WHERE t.account_id = a.account_id)

WHERE EXISTS (SELECT 1  FROM transaction t  WHERE t.account_id =
a.account_id);
```

The two correlated subqueries are identical except for the select clauses. The subquery in the set clause, however, executes only if the condition in the update statement's where clause evaluates to true (meaning that at least one transaction was found for the account), thus protecting the data in the last_activity_date column from being overwritten with a null.

When to Use Subqueries

Now that you have learned about the different types of subqueries and the different operators that you can employ to interact with the data returned by subqueries, it's time to explore the many ways in which you can use subqueries to build powerful SQL statements. The next three sections demonstrate how you may use subqueries to construct custom tables, to build conditions, and to generate column values in result sets.

Subqueries as Data Sources

Since a subquery generates a result set containing rows and columns of data, it is perfectly valid to include subqueries in your from clause along with tables. Although it might, at first glance, seem like an interesting feature without much practical merit, using subqueries alongside tables is one of the most powerful tools available when writing queries. Here's a

simple example:

```
mysql> SELECT d.dept_id, d.name, e_cnt.how_many num_employees

-> FROM department d INNER JOIN

->  (SELECT dept_id, COUNT(*) how_many

->   FROM employee    ->   GROUP BY dept_id) e_cnt

->   ON d.dept_id = e_cnt.dept_id;

+---------+----------------+---------------+

| dept_id | name           | num_employees |

+---------+----------------+---------------+

|       1 | Operations     |            14 |

|       2 | Loans          |             1 |

|       3 | Administration |             3 |

+---------+----------------+---------------+
```

3 rows in set (0.04 sec)

In this example, a subquery generates a list of department IDs along with the number of employees assigned to each department. Here's the result set generated by the subquery:

```
mysql> SELECT dept_id, COUNT(*) how_many

-> FROM employee

-> GROUP BY dept_id;
```

```
+---------+----------+
| dept_id | how_many |
+---------+----------+
|       1 |       14 |
|       2 |        1 |
|       3 |        3 |
+---------+----------+
```

3 rows in set (0.00 sec)

The subquery is given the name e_cnt and is joined to the department table via the dept_id column. The containing query then retrieves the department ID and name from the department table, along with the employee count from the e_cnt subquery. Subqueries used in the from clause must be noncorrelated; they are executed first, and the data is held in memory until the containing query finishes execution. Subqueries offer immense flexibility when writing queries, because you can go far beyond the set of available tables to create virtually any view of the data that you desire, and then join the results to other tables or subqueries. If you are writing reports or generating data feeds to external systems, you may be able to do things with a single query that used to demand multiple queries or a procedural language to accomplish.

Transactions

All of the examples thus far in this book have been individual, independent SQL statements. While this may be the norm for ad hoc reporting or data maintenance scripts, application logic will frequently include multiple SQL statements that need to execute together as a logical unit of work. This chapter explores the need and the infrastructure necessary to execute multiple SQL statements concurrently.

Multiuser Databases

Database management systems allow not only a single user to query and modify data, but multiple people to do so simultaneously. If every user is only executing queries, such as might be the case with a data warehouse during normal business hours, then there are very few issues for the database server to deal with. If some of the users are adding and/or modifying data, however, the server must handle quite a bit more bookkeeping.

Let's say, for example, that you are running a report that shows the available balance for all the checking accounts opened at your branch. At the same time you are running the report, however, the following activities are occurring:

- A teller at your branch is handling a deposit for one of your customers.
- A customer is finishing a withdrawal at the ATM in the front lobby.
- The bank's month-end application is applying interest to the accounts.

While your report is running, therefore, multiple users are modifying the underlying data, so what numbers should appear on the report? The answer depends somewhat on how your server handles locking, which is described in the next section.

Locking Locks are the mechanism the database server uses to control simultaneous use of data resources. When some portion of the database is locked, any other users wishing to modify (or possibly read) that data must wait until the lock has been released. Most database servers use one of two locking strategies:

Database writers must request and receive from the server a write lock to modify data, and database readers must request and receive from the server a read lock to query data. While multiple users can read data simultaneously, only one write lock is given out at a time for each table (or portion thereof), and read requests are blocked until the write lock is released.

Database writers must request and receive from the server a write lock to modify data, but readers do not need any type of lock to query data. Instead, the server ensures that a reader sees a consistent view of the data (the data seems the same even though other users may be making modifications) from the time her query begins until her query has finished. This approach is known as versioning.

There are pros and cons to both approaches. The first approach can lead to long wait times if there are many concurrent read and write requests, and the second approach can be problematic if there are long-running queries while data is being modified. Of the three servers discussed in this book, Microsoft SQL Server uses the first approach, Oracle Database uses the second approach, and MySQL uses both approaches (depending on your choice of storage engine, which we'll discuss a bit later in the chapter).

Lock Granularities

There are also a number of different strategies that you may employ when deciding how to lock a resource. The server may apply a lock at one of three different levels, or granularities:

Table locks Keep multiple users from modifying data in the same table simultaneously Page locks Keep multiple users from modifying data on the same page (a page is a segment of memory generally in the range of 2 KB to 16 KB) of a table simultaneously Row locks Keep multiple users from modifying the same row in a table simultaneously Again, there are pros and cons to these approaches. It takes very little bookkeeping to lock entire tables, but this approach quickly yields unacceptable wait times as the number of users increases.

On the other hand, row locking takes quite a bit more bookkeeping, but it allows many users to modify the same table as long as they are interested in different rows. Of the three servers discussed in this book, Microsoft SQL Server uses page, row, and table locking, Oracle Database uses only row locking, and MySQL uses table, page, or row locking (depending, again, on your choice of storage engine). SQL Server will, under certain circumstances, escalate locks from row to page, and from page to table, whereas Oracle Database will never escalate locks.

What Is a Transaction?

If database servers enjoyed 100% uptime, if users always allowed programs to finish executing, and if applications always completed without encountering fatal errors that halt execution, then there would be nothing left to discuss regarding concurrent database access. However, we can rely on none of these things, so one more element is necessary to allow multiple users to access the same data.

This extra piece of the concurrency puzzle is the transaction, which is a device for grouping together multiple SQL statements such that either all or none of the statements succeed (a property known as atomicity). If you attempt to transfer $500 from your savings account to your checking account, you would be a bit upset if the money were successfully withdrawn from your savings account but never made it to your checking account.

Whatever the reason for the failure (the server was shut down for maintenance, the request for a page lock on the account table timed out, etc.), you want your $500 back. To protect against this kind of error, the program that handles your transfer request would first begin a transaction, then issue the SQL statements needed to move the money from your savings to your checking account, and, if everything succeeds, end the transaction by issuing the commit command. If something unexpected happens, however, the program would issue a rollback command, which instructs the server to undo all changes made since the transaction began. The entire process might look something like the following:

```
START TRANSACTION; /* withdraw money from first account, making
sure balance is sufficient */ UPDATE account SET avail_balance =
avail_balance - 500 WHERE account_id = 9988  AND avail_balance > 500;
```

```
   IF <exactly one row was updated by the previous statement> THEN   /*
deposit money into second account */  UPDATE account SET avail_balance =
avail_balance + 500    WHERE account_id = 9989;
```

```
   IF <exactly one row was updated by the previous statement> THEN    /*
```

everything worked, make the changes permanent */ COMMIT; ELSE /* something went wrong, undo all changes in this transaction */ ROLLBACK; END IF; ELSE /* insufficient funds, or error encountered during update */ ROLLBACK; END IF;

Starting a Transaction

Database servers handle transaction creation in one of two ways:

An active transaction is always associated with a database session, so there is no need or method to explicitly begin a transaction. When the current transaction ends, the server automatically begins a new transaction for your session.

Unless you explicitly begin a transaction, individual SQL statements are automatically committed independently of one another. To begin a transaction, you must first issue a command. Of the three servers, Oracle Database takes the first approach, while Microsoft SQL Server and MySQL take the second approach. One of the advantages of Oracle's approach to transactions is that, even if you are issuing only a single SQL command, you have the ability to roll back the changes if you don't like the outcome or if you change your mind.

Thus, if you forget to add a where clause to your delete statement, you will have the opportunity to undo the damage (assuming you've had your morning coffee and realize that you didn't mean to delete all 125,000 rows in your table). With MySQL and SQL Server, however, once you press the Enter key, the changes brought about by your SQL statement will be permanent (unless your DBA can retrieve the original data from a backup or from some other means).

Ending a Transaction

Once a transaction has begun, whether explicitly via the start transaction command or implicitly by the database server, you must explicitly end your transaction for your changes to become permanent. You do this by way of the commit command, which instructs the server to mark the changes as permanent and release any resources (i.e., page or row locks) used during the transaction. If you decide that you want to undo all the changes made since starting the transaction, you must issue the rollback command, which instructs the server to return the data to its

pretransaction state. After the rollback has been completed, any resources used by your session are released.

Along with issuing either the commit or rollback command, there are several other scenarios by which your transaction can end, either as an indirect result of your actions or as a result of something outside your control:

The server shuts down, in which case, your transaction will be rolled back automatically when the server is restarted.

You issue an SQL schema statement, such as alter table, which will cause the current transaction to be committed and a new transaction to be started.

You issue another start transaction command, which will cause the previous transaction to be committed.

The server prematurely ends your transaction because the server detects a deadlock and decides that your transaction is the culprit. In this case, the transaction will be rolled back and you will receive an error message.

Indexes and Constraints

Because the focus of this book is on programming techniques, i concentrated on elements of the SQL language that you can use to craft powerful select, insert, update, and delete statements. However, other database features indirectly affect the code you write.

Indexes

When you insert a row into a table, the database server does not attempt to put the data in any particular location within the table. For example, if you add a row to the department table, the server doesn't place the row in numeric order via the dept_id column or in alphabetical order via the name column. Instead, the server simply places the data in the next available location within the file (the server maintains a list of free space for each table). When you query the department table, therefore, the server will need to inspect every row of the table to answer the query. For example, let's say that you issue the following query:

```
mysql> SELECT dept_id, name

-> FROM department

-> WHERE name LIKE 'A%';

+---------+-------------    ---+

| dept_id | name           |

+---------+---------------+
```

```
|      3 | Administration |

+---------+---------------+
```

1 row in set (0.03 sec)

To find all departments whose name begins with A, the server must visit each row in the department table and inspect the contents of the name column; if the department name begins with A, then the row is added to the result set. This type of access is known as a table scan.

While this method works fine for a table with only three rows, imagine how long it might take to answer the query if the table contains 3 million rows. At some number of rows larger than three and smaller than 3 million, a line is crossed where the server cannot answer the query within a reasonable amount of time without additional help. This help comes in the form of one or more indexes on the department table. Even if you have never heard of a database index, you are certainly aware of what an index is (e.g., this book has one).

An index is simply a mechanism for finding a specific item within a resource. Each technical publication, for example, includes an index at the end that allows you to locate a specific word or phrase within the publication. The index lists these words and phrases in alphabetical order, allowing the reader to move quickly to a particular letter within the index, find the desired entry, and then find the page or pages on which the word or phrase may be found.

Index Creation

Returning to the department table, you might decide to add an index on the name column to speed up any queries that specify a full or partial department name, as well as any update or delete operations that specify a department name. Here's how you can add such an index to a MySQL database:

```
mysql> ALTER TABLE department

-> ADD INDEX dept_name_idx (name);
```

Query OK, 3 rows affected (0.08 sec) Records: 3 Duplicates: 0 Warnings: 0

This statement creates an index (a B-tree index to be precise, but more on this shortly) on the department.name column; furthermore, the index is given the name dept_name_idx. With the index in place, the query optimizer can choose to use the index if it is deemed beneficial to do so (with only three rows in the department table, for example, the optimizer might very well choose to ignore the index and read the entire table). If there is more than one index on a table, the optimizer must decide which index will be the most beneficial for a particular SQL statement.

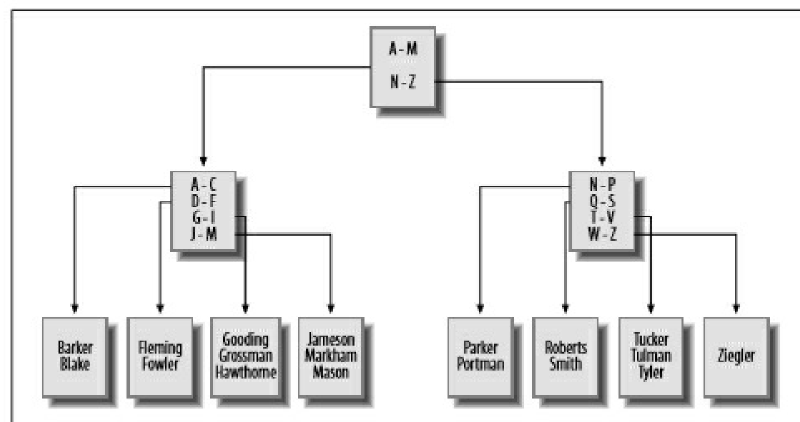
Types of Indexes

Indexing is a powerful tool, but since there are many different types of data, a single indexing strategy doesn't always do the job. The following sections illustrate the different types of indexing available from various servers.

B-tree Indexes

All the indexes shown thus far are balanced-tree indexes, which are more commonly known as B-tree indexes. MySQL, Oracle Database, and SQL Server all default to Btree indexing, so you will get a B-tree index unless you explicitly ask for another type. As you might expect, B-tree indexes are organized as trees, with one or more levels of branch nodes leading to a single level of leaf nodes. Branch nodes are used for navigating the tree, while leaf nodes hold the actual values and location information..

ure 13-1.



Bitmap Indexes

Although B-tree indexes are great at handling columns that contain many different values, such as a customer's first/last names, they can become unwieldy when built on a column that allows only a small number of values. For example, you may decide to generate an index on the account.product_cd column so that you can quickly retrieve all accounts of a specific type (e.g., checking, savings). Because there are only eight different products, however, and because some products are far more popular than others, it can be difficult to maintain a balanced B-tree index as the number of accounts grows. For columns that contain only a small number of values across a large number of rows (known as low-cardinality data), a different indexing strategy is needed. To handle this situation more efficiently, Oracle Database includes bitmap indexes, which generate a bitmap for each value stored in the column.

Value/row	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
BUS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
CD	0	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	0	0	0	0	0
CHK	1	0	0	1	0	1	0	1	0	0	1	1	0	0	1	0	1	0	0	1	0	0	1	0
MM	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0
SAV	0	1	0	0	1	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
SBL	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

How Indexes Are Used

Indexes are generally used by the server to quickly locate rows in a particular table, after which the server visits the associated table to extract the additional information requested by the user. Consider the following query:

```
mysql> SELECT emp_id, fname, lname

-> FROM employee

-> WHERE emp_id IN (1, 3, 9, 15);
```

emp_id	fname	lname
1	Michael	Smith
3	Robert	Tyler
9	Jane	Grossman
15	Frank	Portman

4 rows in set (0.00 sec)

For this query, the server can use the primary key index on the emp_id column to locate employee IDs 1, 3, 9, and 15 in the employee table, and then visit each of the four rows to retrieve the first and last name columns.

If the index contains everything needed to satisfy the query, however, the server doesn't need to visit the associated table. To illustrate, let's look at how the query optimizer approaches the same query with different indexes in place. The query, which aggregates account balances for specific customers, looks as follows:

```
mysql> SELECT cust_id, SUM(avail_balance) tot_bal

-> FROM account

-> WHERE cust_id IN (1, 5, 9, 11)

-> GROUP BY cust_id;
```

cust_id	tot_bal
1	4557.75
5	2237.97
9	10971.22

11	9345.55

4 rows in set (0.00 sec)

The Downside of Indexes

If indexes are so great, why not index everything? Well, the key to understanding why more indexes are not necessarily a good thing is to keep in mind that every index is a table (a special type of table, but still a table). Therefore, every time a row is added to or removed from a table, all indexes on that table must be modified. When a row is updated, any indexes on the column or columns that were affected need to be modified as well. Therefore, the more indexes you have, the more work the server needs to do to keep all schema objects up-to-date, which tends to slow things down.

Indexes also require disk space as well as some amount of care from your administrators, so the best strategy is to add an index when a clear need arises. If you need an index for only special purposes, such as a monthly maintenance routine, you can always add the index, run the routine, and then drop the index until you need it again. In the case of data warehouses, where indexes are crucial during business hours as users run reports and ad hoc queries but are problematic when data is being loaded into the warehouse overnight, it is a common practice to drop the indexes before data is loaded and then re-create them before the warehouse opens for business.

In general, you should strive to have neither too many indexes nor too few. If you aren't sure how many indexes you should have, you can use this strategy as a default:

Make sure all primary key columns are indexed (most servers automatically create unique indexes when you create primary key constraints). For multicolumn primary keys, consider building additional indexes on a subset of the primary key columns, or on all the primary key columns but in a different order than the primary key constraint definition.

Build indexes on all columns that are referenced in foreign key constraints. Keep in mind that the server checks to make sure there are

no child rows when a parent is deleted, so it must issue a query to search for a particular value in the column. If there's no index on the column, the entire table must be scanned.

Index any columns that will frequently be used to retrieve data. Most date columns are good candidates, along with short (3- to 50-character) string columns. After you have built your initial set of indexes, try to capture actual queries against your tables, and modify your indexing strategy to fit the most-common access paths.

Constraints

A constraint is simply a restriction placed on one or more columns of a table. There are several different types of constraints, including: Primary key constraints Identify the column or columns that guarantee uniqueness within a table Foreign key constraints Restrict one or more columns to contain only values found in another table's primary key columns, and may also restrict the allowable values in other tables if update cascade or delete cascade rules are established unique constraints.

Restrict one or more columns to contain unique values within a table (primary key constraints are a special type of unique constraint) Check constraints Restrict the allowable values for a column Without constraints, a database's consistency is suspect. For example, if the server allows you to change a customer's ID in the customer table without changing the same customer ID in the account table, then you will end up with accounts that no longer point to valid customer records (known as orphaned rows). With primary and foreign key constraints in place, however, the server will either raise an error if an attempt is made to modify or delete data that is referenced by other tables, or propagate the changes to other tables for you (more on this shortly).

Constraint Creation

Constraints are generally created at the same time as the associated table via the create table statement. To illustrate, here's an example from the schema generation script for this book's example database:

```
CREATE TABLE product (product_cd VARCHAR(10) NOT NULL,  name
```

```
VARCHAR(50) NOT NULL, product_type_cd VARCHAR (10) NOT NULL, date_offered DATE, date_retired DATE,

CONSTRAINT fk_product_type_cd FOREIGN KEY (product_type_cd) REFERENCES product_type (product_type_cd), CONSTRAINT pk_product PRIMARY KEY (product_cd) );
```

Views

Well-designed applications generally expose a public interface while keeping implementation details private, thereby enabling future design changes without impacting end users. When designing your database, you can achieve a similar result by keeping your tables private and allowing your users to access data only through a set of views. This chapter strives to define what views are, how they are created, and when and how you might want to use them.

What Are Views?

A view is simply a mechanism for querying data. Unlike tables, views do not involve data storage; you won't need to worry about views filling up your disk space. You create a view by assigning a name to a select statement, and then storing the query for others to use. Other users can then use your view to access data just as though they were querying tables directly (in fact, they may not even know they are using a view). As a simple example, let's say that you want to partially obscure the federal IDs (Social Security numbers and corporate identifiers) in the customer table.

The customer service department, for example, may need access to just the last portion of the federal ID in order to verify the identity of a caller, but exposing the entire number would violate the company's privacy policy. Therefore, instead of allowing direct access to the customer table, you define a view called customer_vw and mandate that all bank personnel use it to access customer data. Here's the view definition: CREATE VIEW customer_vw (cust_id, fed_id, cust_type_cd, address, city, state, zipcode) AS SELECT cust_id, concat('ends in ', substr(fed_id, 8, 4)) fed_id, cust_type_cd, address, city, state, postal_code FROM customer;

The first part of the statement lists the view's column names, which may be different from those of the underlying table (e.g., the customer_vw view has a column named zipcode which maps to the customer.postal_code column). The second part of the statement is a select statement, which must contain one expression for each column in the view. When the create view statement is executed, the database server simply stores the view definition for future use; the query is not executed, and no data is retrieved or stored. Once the view has been created, users can query it just like they would a table, as in:

```
mysql> SELECT cust_id, fed_id, cust_type_cd

-> FROM customer_vw;

+---------+-------------+-------------+

| cust_id | fed_id      | cust_type_cd |

+---------+-------------+------------+

|       1 | ends in 1111 | I          |

|       2 | ends in 2222 | I          |

|       3 | ends in 3333 | I          |

|       4 | ends in 4444 | I          |

|       5 | ends in 5555 | I          |

|       6 | ends in 6666 | I          |

|       7 | ends in 7777 | I          |

|       8 | ends in 8888 | I          |
```

```
|       9 | ends in 9999 | I           |
|      10 | ends in 111  | B           |
|      11 | ends in 222  | B           |
|      12 | ends in 333  | B           |
|      13 | ends in 444  | B           |
+---------+--------------+-------------+
```

13 rows in set (0.02 sec)

The actual query that the server executes is neither the one submitted by the user nor the query attached to the view definition. Instead, the server merges the two together to create another statement, which in this case looks as follows:

```
SELECT cust_id, concat('ends in ', substr(fed_id, 8, 4)) fed_id, cust_type_cd
```

FROM customer; Even though the customer_vw view definition includes seven columns of the customer table, the query executed by the server retrieves only three of the seven. As you'll see later in the chapter, this is an important distinction if some of the columns in your view are attached to functions or subqueries.

Why Use Views?

In the previous section, I demonstrated a simple view whose sole purpose was to mask the contents of the customer.fed_id column. While views are often employed for this purpose, there are many reasons for using views, as I demonstrate in the following subsections.

Data Security

If you create a table and allow users to query it, they will be able to access every column and every row in the table. As I pointed out earlier, however, your table may include some columns that contain sensitive

data, such as identification numbers or credit card numbers; not only is it a bad idea to expose such data to all users, but also it might violate your company's privacy policies, or even state or federal laws, to do so.

The best approach for these situations is to keep the table private (i.e., don't grant select permission to any users) and then to create one or more views that either omit or obscure (such as the 'ends in ' approach taken with the customer_vw.fed_id column) the sensitive columns. You may also constrain which rows a set of users may access by adding a where clause to your view definition. For example, the next view definition allows only business customers to be queried:

```
CREATE VIEW business_customer_vw
(cust_id, fed_id, cust_type_cd, address, city, state, zipcode )

AS SELECT cust_id, concat('ends in ', substr(fed_id, 8, 4))
fed_id, cust_type_cd, address, city, state, postal_code

FROM customer WHERE cust_type_cd = 'B'
```

If you provide this view to your corporate banking department, they will be able to access only business accounts because the condition in the view's where clause will always be included in their queries.

Data Aggregation

Reporting applications generally require aggregated data, and views are a great way to make it appear as though data is being pre-aggregated and stored in the database. As an example, let's say that an application generates a report each month showing the number of accounts and total deposits for every customer. Rather than allowing the application developers to write queries against the base tables, you could provide them with the following view:

```
CREATE VIEW customer_totals_vw
(cust_id, cust_type_cd, cust_name, num_accounts, tot_deposits )

AS SELECT cst.cust_id, cst.cust_type_cd,
```

```
CASE    WHEN cst.cust_type_cd = 'B' THEN

(SELECT bus.name FROM business bus

WHERE bus.cust_id = cst.cust_id)

ELSE    (SELECT concat(ind.fname, ' ', ind.lname)

FROM individual ind

WHERE ind.cust_id = cst.cust_id)

END cust_name,

sum(CASE WHEN act.status = 'ACTIVE' THEN 1 ELSE 0 END) tot_active_accounts,     sum(CASE WHEN act.status = 'ACTIVE' THEN act.avail_balance ELSE 0 END) tot_balance FROM customer cst

INNER JOIN account act  ON act.cust_id = cst.cust_id

GROUP BY cst.cust_id, cst.cust_type_cd;
```

Using this approach gives you a great deal of flexibility as a database designer. If you decide at some point in the future that query performance would improve dramatically if the data were preaggregated in a table rather than summed using a view, you can create a customer_totals table and modify the customer_totals_vw view definition to retrieve data from this table. Before modifying the view definition, you can use it to populate the new table. Here are the necessary SQL statements for this scenario:

```
mysql> CREATE TABLE customer_totals

-> AS
```

```
-> SELECT * FROM customer_totals_vw;
```

Query OK, 13 rows affected (3.33 sec)

Records: 13 Duplicates: 0 Warnings: 0

```
mysql> CREATE OR REPLACE VIEW customer_totals_vw

-> (cust_id,

->  cust_type_cd,

->  cust_name,

->  num_accounts,

->  tot_deposits

-> )

-> AS

-> SELECT cust_id, cust_type_cd, cust_name, num_accounts, tot_deposits

-> FROM customer_totals;
```

Query OK, 0 rows affected (0.02 sec)

From now on, all queries that use the customer_totals_vw view will pull data from the new customer_totals table, meaning that users will see a performance improvement without needing to modify their queries.

Hiding Complexity

One of the most common reasons for deploying views is to shield end users from complexity. For example, let's say that a report is created each month showing the number of employees, the total number of

active accounts, and the total number of transactions for each branch. Rather than expecting the report designer to navigate four different tables to gather the necessary data, you could provide a view that looks as follows:

```
CREATE VIEW branch_activity_vw
(branch_name, city, state, num_employees, num_active_accounts, tot_tra
)

AS SELECT br.name, br.city, br.state,

(SELECT count(*)  FROM employee emp

WHERE emp.assigned_branch_id = br.branch_id) num_emps,

(SELECT count(*)  FROM account acnt

WHERE acnt.status = 'ACTIVE' AND acnt.open_branch_id = br.branch_id)
num_accounts, (SELECT count(*)

FROM transaction txn

WHERE txn.execution_branch_id = br.branch_id) num_txns

FROM branch br;
```

This view definition is interesting because three of the six column values are generated using scalar subqueries. If someone uses this view but does not reference the num_employees, num_active_accounts, or tot_transactions column, then none of the subqueries will be executed.

Conclusion

If you enjoy playing with, manipulating, and analyzing data, a career as a SQL programmer could be right for you. Learning SQL for beginners will set you up with the skills and knowledge that you need to work with databases of any complexity or size.

Start by enrolling in an online course and learning the basics of SQL. Make sure that you understand how SQL works and what it's used for, and get familiar with some of the most common types of database. Get in the habit of using some of the other resources that are available to you – reference guides and videos are just a couple of my favorites – and make sure that you practice writing code as often as possible.

Above all, make sure that you're always following best practices, that you're continually working to build your skills, and that you're writing code as often as you can. Get active on SQL forums, try and meet other coders in your area, and make sure that you have fun on your journey from SQL beginner to pro database manipulator.

The breadth and scope of the SQL commands provide the capability to create and manipulate a wide variety of database objects using the various CREATE, ALTER, and DROP commands. Those database objects then can be loaded with data using commands such as INSERT. The data can be manipulated using a wide variety of commands, such as SELECT, DELETE, and TRUNCATE, as well as the cursor commands, DECLARE, OPEN, FETCH, and CLOSE. Transactions to manipulate the data are controlled through the SET command, plus the COMMIT and ROLLBACK commands. And finally, other commands covered in this chapter include those that control a user's access to database resources through commands such as GRANT and REVOKE.

Disclaimer

Disclaimer All the material contained in this book is provided for educational and informational purposes only. No responsibility can be taken for any results or outcomes resulting from the use of this material. While every attempt has been made to provide information that is both accurate and effective, the author does not assume any responsibility for the accuracy or use/misuse of this information