COM230 SQL and Database Design

8 Units: 60 Hours Lecture + 40 Hours Lab

* Student Manual

Lesson #1

|  |  |  |
| --- | --- | --- |
| Topic | Introduction to MySQL and Relational Databases | Points |
| Chapter | One |  |
| Activity | Practical Application: creating my own database | 3 |

**Main Topics:**

* What is a Database?
* Why is MySQL so Popular?
* Elements of MySQL and Its Environment
* The Relational Database Model

A database system is essentially a high-powered way to manage lists of information. The information can come from a variety of sources. It might be research data, business records, customer requests, sports statistics, sales reports, personal hobby information, personnel records, or student grades. However, although database systems can deal with a wide range of information, you don’t use such a system for its own sake. If a job is easy to do already, there’s no reason to drag a database into it just to use one. A grocery list is a good example: You write down the items to get, cross them off as you do your shopping, and then throw the list away. It’s highly unlikely that you would use a database for this. Even if you have a palmtop computer, you would probably keep track of a grocery list using its notepad function rather than its database capabilities.

The power of a database system comes into play when the information you want to organize and manage becomes so voluminous or complex that you records become more burdensome than you care to deal with by hand. Clearly this is the case for large corporations processing millions of transactions a day; a database is a necessity under such circumstances. But even small-scale operations involving a single person maintaining information of personal interest might require a database. Consider the following situations:

* Your carpentry business has several employees. You need to maintain employee and payroll records so that you know whom you have paid and when, and you must summarize those records so that you can report earnings statements to the government for tax purposes. You also need to keep track of the jobs your company as been hired to do and which employees you’ve scheduled to work on each job.
* You run a network of automobile parts warehouses and need to be able to tell which ones have any given part in their inventory so that you can fill customer orders.
* That file of research data that you have been collecting over the course of many years needs to be analyzed for publication. You want to boil down large amounts of raw data to generate summary information, and to pull out selected subsets of observations for more detailed statistical analysis.
* You are a teacher who needs to keep track of grades and attendance. Each time you give a quiz or a test, you record every student’s grade. It’s easy enough to write down scores in a grade-book, but using the scores later is a tedious chore. You would rather avoid sorting the scores to determine the grading curve, and you’d really rather not add up each student’s scores when you determine final grades. Counting each student’s absences is no fun, either.

These scenarios range from situations involving small amounts to large amounts of information. Their common characteristic is that they involve tasks that can be performed manually but that could be performed more efficiently by a database system.

What specific benefits should you expect to see from using a database system such as MySQL? It depends on you particular needs and requirements. A database is like a big filing cabinet in some ways, but one with a sophisticated built-in filing system. There are some important advantages of electronically maintained records over records maintained by hand:

* Reduced record filing time
* Reduced record retrieval time
* Flexible retrieval order
* Flexible output format
* Simultaneous multiple-user access to records
* Remote access to and electronic transmission of records

**Basic Database Terminology**

Within the database world, MySQL is classified as a relational database management system (RDBMS). That phrase breaks down as follows:

* The Database (the "DB" in RDBMS) is the repository for the information you want to store, structured in a simple, regular fashion:
  + The collection of data in a database is organized into tables
  + Each table is organized into rows and columns
  + Each row in a table is a record
  + Records can contain several pieces of information; each column in a table corresponds to one of those pieces
* The management system (the "MS") is the software that lets you use your data by allowing you to insert, retrieve, modify, or delete records.
* The word "relational" (the "R") indicates a particular kind of DBMS. The power of a relational DBMS lies in its capability to pull data from related tables using common columns.

**HOMEWORK**

**Review Terms: 5 points**

The 65 terms listed in lesson 12 with definitions/explanations will be turned in word format when the student reaches lesson 12. It is recommended that the student fills out these definitions/ explanations as they occur.

**Practical Application** (3points): Select a data-dense topic of interest to you (ensure that the topic is not offensive in an academic setting). Map a single table schema, and create the table in your database instance.Lesson #2

|  |  |  |
| --- | --- | --- |
| Topic | Relational Database Concepts | Points |
|  | Lecture |  |

**Main Topics**

* Entities
* Attributes
* Relationships
* Constraints
* Tables
* SQL
* Primary Key
* Foreign Key
* Referential Integrity

The basic building blocks of all data models are entities, attributes, relationships, and constraints. An entity is anything (a person, a place, a thing, or an event) about which data are to be collected and stored. An entity represents a particular type of object in the real world. Entities may be physical objects, such as customers or products, but entities may also be abstractions, such as flight routes or musical concerts.

An attribute is a characteristic of an entity. For example, a CUSTOMER entity would be described by attributes such as last name, first name, phone, address, and credit limit. Attributes are the equivalent of fields in file systems.

A relationship describes an association among entities. For example, a relationship exists between customers and agents that can be described as follows: an agent can serve many customers, and each customer may be served by one agent. Data models use three types of relationships: one-to-many, many-to-many, and one-to-one. Database designers usually use the shorthand notations 1:M, M:N, and 1:1. The following examples illustrate the distinctions among the three.

* One-to-many relationship. A painter paints many different paintings, but each one of them is painted by only one painter. The painter (the "one") is related to the paintings (the "many").
* Many-to-many relationship. An employee may learn many job skills, and each job skill may be learned by many employees.
* One-to-one relationship. A retail company’s structure may require that each of its stores be managed by a single employee. In turn, each store manager, who is an employee, manages only a single store.

A constraint is a restriction placed on the data. Constraints are important because they help to ensure data integrity. Constraints are normally expressed in the form of rules; for example:

* The employee’s salary must have values that are between 6,000 and 350,000.
* A student’s GPA must be between 0.00 and 4.00.
* Each class must have one and only one teacher.

The relational data model is implemented through a relational database management system (RDBMS). The RDBMS performs the same basic function provided by the hierarchical and network DBMS systems, in addition to a host of other functions that make the relational data model easier to understand and implement.

The most important advantage of the RDBMS is its ability to hide the complexities of the relational model from the user. The RDBMS manages all of the physical details, while the user sees the relational database as a collection of tables in which data are stored and can manipulate and query data in a way that seems intuitive and logical.

Each table is a matrix consisting of a series of row/column intersections. Tables, also called relations, are related to each other through the sharing of a common entity characteristic (value in a column).

Another reason for the relational data model’s rise to dominance is its powerful and flexible query language. For most relational database software, the query language is Structured Query Language (SQL). SQL is a 4GL that allows the user to specify what must be done without specifying how it must be done. The RDBMS uses SQL to translate user queries into instructions for retrieving the requested data. SQL makes it possible to retrieve data with far less effort than any other database environment.

The logical view of the relational database is facilitated by the creation of data relationships based on a logical construct known as a table. A table is perceived as a two-dimensional structure composed of rows and columns. As far as the table’s user is concerned, a table contains a group of related entities, that is, an entity set; for that reason, the terms entity set and table are often used interchangeably. A table is also called a relation because the relational model’s creator, E. F. Codd, used the term relation as a synonym for table.

**Characteristics of a Relational Table**

* A table is perceived as a two-dimensional structure composed of rows and columns
* Each table row (tuple) represents a single entity occurrence within the entity set
* Each table column represents an attribute, and each column has a distinct name
* Each row/column intersection represents a single data value
* All values in a column must conform to the same data format
* Each column has a specific range of values known as the attribute domain
* The order of the rows and columns is immaterial to the DBMS
* Each table must have an attribute or a combination of attributes that uniquely identifies each row.

Each table must have a primary key, the primary key is an attribute (or combination of attributes) that uniquely identifies any given row. A null value is never permitted in the primary key.

A foreign key is an attribute whose values match the primary key values in a related table. If the foreign key contains either matching values or nulls, the table that makes use of that foreign key is said to exhibit referential integrity, In other words, referential integrity means that if the foreign key contains a value, that value refers to an existing valid row in the related table.

Lesson #3

|  |  |  |
| --- | --- | --- |
| Topic | MySQL Tools | Points |
| Chapter | Three |  |

**Main Topics:**

* MySql Monitor
* PHPMyAdmin

MySql Monitor is a command line interface to a MySql database. On our network it is accessible on linuxsandbox.coleman.edu. You will use the Putty application to gain Remote Access to the Sandbox:

Enter the server name in the Host Name textbox as shown in Figure 2.1. Click Open

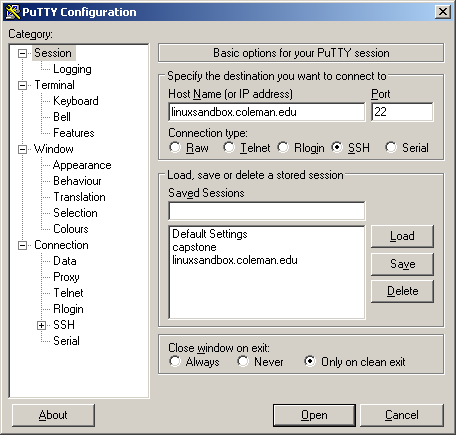


Figure 2.1

Enter your username and password as shown in Figure 2.2. Press the Enter key.

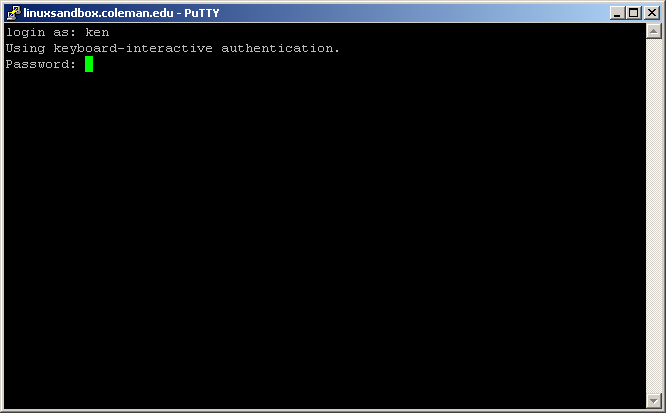


Figure 2.2

You will now see the screen shown in Figure 2.3

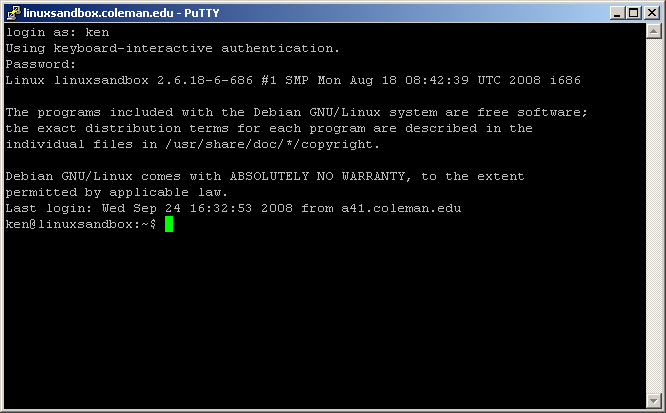
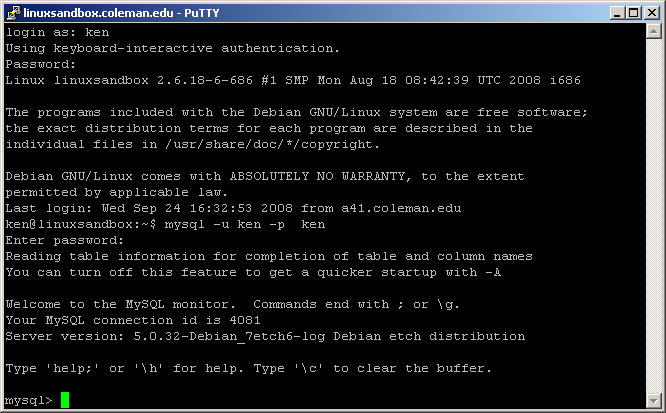


Figure 2.3

Enter the following command:

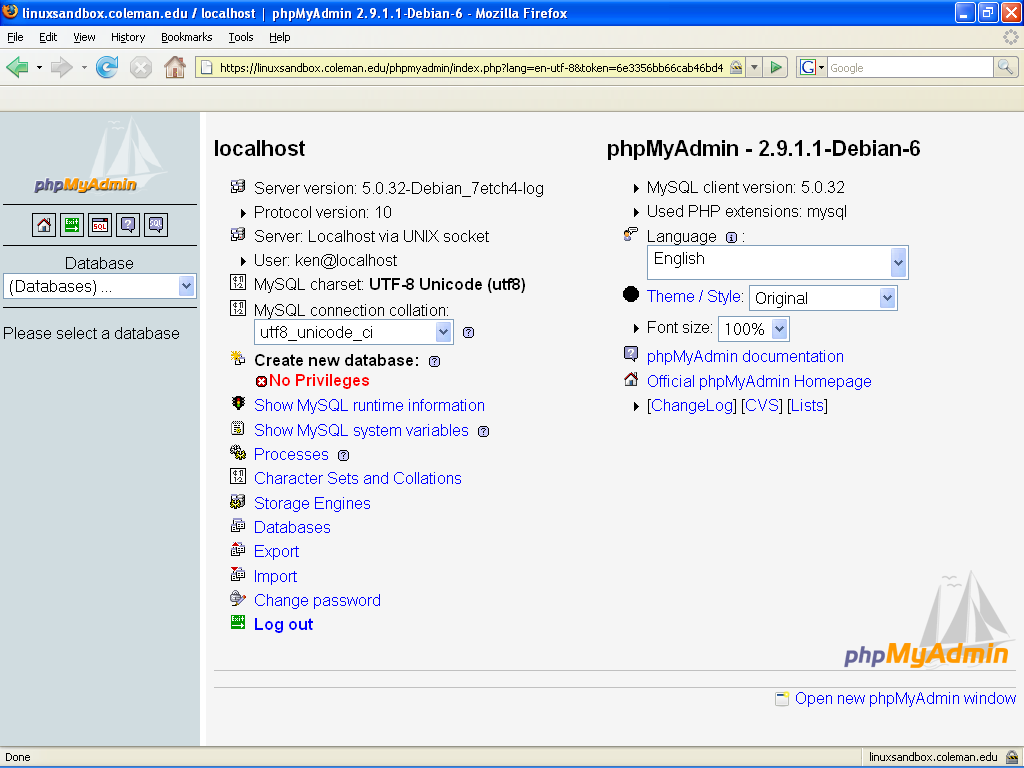
mysql -u username –p databasename

The –u tells MySQL that you are entering a user name and –p tells MySQL to ask for a password.

Figure 2.4

After you enter your password you will see the screen as shown in Figure 2.4 and you can start entering SQL commands.

PHPMyAdmin is a graphical interface program written in PHP, which gives you the ability to perform numerous operations in MySQL. Your instructor will cover the details of this interface in class.



PHPMyAdmin – Main Screen

Lesson #4

|  |  |  |
| --- | --- | --- |
| Topic | Database Design | Points |
| Chapter | Four |  |
| Project | One: Northwoods University Database Design | 5 |

**Main Topics:**

* How Not to Design a Database
* The Database Design Process
* Data Abstraction
* The Entity Relationship Model
  + Chen Model
  + Crow’s Foot Model
* Entities
* Mapping Entities and Relationships to Database Tables

When implementing a new database, it’s easy to fall into the trap of trying to quickly get something up and running without dedicating adequate time and effort to the design. This carelessness frequently leads to costly redesigns and re-implementations down the line. Designing a database is similar to drafting the blueprints for a house; it’s silly to start building without detailed plans. A good design allows you to extend the original building without having to pull everything down and start from scratch.

**The Database Design Process**

There are three major stages in database design, each producing a progressively lower-level description:

* Requirements analysis – First, we determine and write down what exactly the database is needed for, what data will be stored, and how the data items relate to each other. In practice, this might involve detailed study of the application requirements and talking to people in various roles that will interact with the database and the application.
* Conceptual design – Once we know what the database requirements are, we distill them into a formal description of the database design. In this lesson, we’ll see how to use modeling to produce the conceptual design.
* Logical design – Finally, we map the database design onto an actual database management system and database tables.

**Data Abstraction**

The levels of abstraction can be divided into three models:

* Conceptual - consists of External models, which define the end users’ view of the data. The term ‘end users’ refers to people who use the application programs to manipulate the data and generate information. End users usually operate in an environment in which an application has a specific business unit focus. The external models are then combined to make the conceptual model that represents a global view of the database. The conceptual model is independent of both software and hardware.
* Internal - Once a specific DBMS has been selected, the internal model maps the conceptual model to the DBMS. The internal model is the representation of the database as "seen" by the DBMS. In other words, the internal model requires the designer to match the conceptual model to the selected database. The internal model is said to be software-dependent.
* Physical - The physical model operates at the lowest level of abstraction, describing the way data are saved on storage media such as disks or tapes. The physical model is software- and hardware-dependent. The relational model does not require the designer to be concerned about the data’s physical storage.

**The Entity Relationship Model**

At the basic level, databases store information about distinct objects, or entities, and the associations, or relationships, between these entities. For example, a university might store information about students, courses, and enrollment. A student and a course are entities, while an enrollment is a relationship between them. An inventory and sales database might store information about products, customers, and sales. A product and a customer are entities, while a sale is a relationship between a customer and a product.

A popular approach to conceptual design uses the Entity Relationship (ER) model, which helps transform the requirements into a formal description of the entities and relationships that appear in the database. We’ll start by looking at how the Entity Relationship modeling process works, then apply it to some sample databases.

**Representing Entities**

To help visualize the design, the Entity Relationship Modeling approach involves drawing an Entity Relationship (ER) diagram. In the ER diagram, an entity set is represented by a rectangle containing the entity name.

We typically use the database to store certain characteristics, or attributes, of the entities. In a sales database, we could store the name, email address, postal address, and telephone number for each customer. Attributes describe the entity they belong to.

An attribute may be formed from smaller parts; for example, a postal address is composed of a street number, city, state, zip code and country. We classify attributes as composite if they are composed of smaller parts in this way, and as simple otherwise.

Some attributes can have multiple values for a given entity. For example, a customer could provide several phone numbers, so the telephone number attribute is multi-valued.

In the ER diagram, attributes are represented as labeled ovals and are connected to their owning entity. Attributes comprising the primary key are underlined. The parts of any composite attributes are drawn connected to the oval of the composite attribute, and multi-valued attributes are shown as double-lined ovals.

Attributes can be empty, for example, some customers may not provide their telephone numbers. The primary key of an entity may never be unknown.

You should think carefully when classifying an attribute as multi-valued: are all the values equivalent, or do they in fact represent different things? For example, a customer’s business phone, home phone, or cell phone. Would they be more useful if labeled separately?

**Representing Relationships**

Entities can participate in relationships with other entities. For example, a customer can buy a product, a student can take a course, an artist can record a CD, and so on.

Relationships can be determined by the number of entities that appear on each side of a relationship.

* One-to-Many
* Many-to-Many
* One-to-One

In an ER diagram, a relationship set is represented with a named diamond. The cardinality of the relationship is often indicated alongside the relationship diamond.

It is often possible to simplify many-to-many relationships by replacing the many-to-many relationship with a linking entity and connecting the original entities through two one-to-many relationships.

**Entity or Attribute?**

From time to time, you will encounter cases where you wonder whether an item should be an attribute or an entity on its own. For example, an email address could be modeled as an entity in its own right. When in doubt, consider these rules of thumb:

* Is the item of direct interest to the database?
  + Object of direct interest should be entities, and information that describes them should be stored in attributes. An inventory and sales database is really interested in customers, and not their email addresses, so the email address would best be modeled as an attribute of the customer entity.
* Does the item have components of its own?
  + If so, we must find a way of representing these components; a separate entity might be the best solution. In a student grades example the course name, year and semester could be stored for each course that a student takes. It would be more compact to treat the course as a separate entity and to create a class ID number to identify each time a course is offered to students.
* Can the object have multiple instances?
  + If so, we must find a way to store data on each instance. The cleanest way to do this is to represent the object as a separate entity. In the sales example, we must ask whether customers are allowed to have more than one email address, if they are, we should model the email address as a separate entity.
* Is the object often nonexistent or unknown?
  + If so, it is effectively an attribute of only some of the entities, and it would be better to model it as a separate entity rather than as an attribute that is often empty.

**Weak and Strong Entities**

In database design, we can omit some key information for entities that are dependent on other entities. If we wanted to store the names of our customers’ children, we could create a child entity and store only enough key information to identify it in the context of its parent. We could simply list a child’s first name on the assumption that a customer will never have several children with the same name. Here, the child entity is a weak entity, and its relationship with the customer entity is called an identifying relationship. Weak entities participate totally in the identifying relationship, since they can’t exist in the database independently of their owning entity.

In the ER diagram, we show weak entities and identifying relationships with double lines, and the partial key of a weak entity with a dashed underline. A weak entity is uniquely identified in the context of its regular (or strong) entity, and so the full key for a weak entity is the combination of its own partial key with the key of its owning entity.

**Entity or Relationship**

An easy way to decide whether an object should be an entity or a relationship is to map nouns in the requirements to entities, and to map the verbs to relationships.

**Mapping Entities and Relationships to Database tables**

When converting an ER model to a database schema, we work through each entity and then through each relationship according to the following rules to end up with a set of database tables.

* Map the entities to database tables
  + For each strong entity, create a table comprising its attributes and designate the primary key. The parts of any composite attributes are also included here.
  + For each weak entity, create a table comprising its attributes and including the primary key of its owing entity. The primary key of the owing entity is known as a foreign key here, because it’s not of this table, but of another table. The primary key of the table for the weak entity is the combination of the foreign key and the partial key of the weak entity. If the relationship with the owning entity has any attributes, add them to this table.
  + For each multi-valued attribute of an entity, create a table comprising the entities primary key and the attribute.
* Map the relationship to database tables
  + For each one-to-one relationship between two entities, include the primary key of one entity as a foreign key in the table belonging to the other. If one entity participates totally in the relationship, place the foreign in its table. If both participate totally in the relationship, consider merging them into a single table.
  + For each non-identifying one-to-many relationship between two entities, include the primary key of the entity on the "1" side as a foreign key in the table for the entity on the "N" side. Add any attributes of the relationship in the table alongside the foreign key.
  + For each many-to-many relationship between two entities, create a new table containing the primary key of each entity as the primary key, and add any attributes of the relationship. This step helps to identify linking entities.
  + For each relationship involving more than two entities, create a table with the primary keys of all the participating entities, and add any attributes of the relationship.

**Project One - Northwoods University Design - 5 Points**

The Northwoods University Student Registration Database

Northwoods University has decided to replace its aging mainframe-based student registration system with a more modern client/server database system. School officials want students to be able to retrieve course availability information, register for courses, and print transcripts using personal computers located in the student computer labs. In addition, faculty members must be able to retrieve student course lists, drop and add students, and record course grades. Faculty members must also be able to view records for the students they advise. Security is prime concern, so student and course records must be protected by password access.

The data items for the Northwoods database are:

* Student name, address, telephone number, class (freshman, sophomore, junior, or senior), date of birth, PIN (personal identification number), and advisor
* Course call number (such as COM340), course name, credits, location, duration, maximum enrollment, instructor, and term offered
* Instructor name, office location, telephone number, rank (Professor, Instructor, etc.), and PIN
* Student enrollment and grade information

The database must be able to allow multiple sections of the same course to be taught by different instructors and on different days and times.

You have been asked to design a relational database to support the required tasks. You will provide two diagrams for this project.

* An Entity Relationship Diagram (ERD) using the Chen model (this will be considered a conceptual diagram).
* A Table Diagram using the Crow’s Foot Model (this will be considered an internal diagram)

The tables will need to be in Third Normal form, which will be covered in Lesson 5.

Lesson #5

|  |  |  |
| --- | --- | --- |
| Topic | Database Normalization | Points |
|  | Lecture |  |
| Activity | Practical Application: Inserting and retrieving data | 3 |

**Main Topics:**

* The Normalization Process
* Normal Forms
* The purpose of Normalization

The table is a basic building block in the database design process. Consequently, the table’s structure is of great interest. Ideally, the database design process produces good table structures. Yet it is possible to create poor table structures even in a good database design. So how do you recognize a poor table structure, and how do you produce a good table? The answer to both questions is based on normalization. Normalization is a process for evaluating and correcting table structure to minimize data redundancies, thereby reducing the likelihood of data anomalies. The normalization process involves assigning attributes to tables.

Normalization works through a series of stages called normal forms. The first three stages are described as first normal form (1NF), second normal form (2NF), and third normal form (3NF). For most business database design purposes, 3NF is as high as you need to go in the normalization process.

Although normalization is a very important database design ingredient, you should not assume that the highest level of normalization is always the most desirable. Generally, the higher the normal form, the more relational join operations are required to produce a specified output and more resources are required by the database system to respond to end-user queries. A successful design must also consider end-user demand for fast performance. The process of de-normalization (going from a higher normal form to a lower normal form) may sometimes be required. However the price you pay for the increased performance through de-normalization is greater data redundancy.

The objective of normalization is to produce a set of normalized tables to store the data that will be used to generate the required information. The objective is to create tables that have the following characteristics:

* Each table represents a single entity. For example, a course table will only contain data that directly pertains to courses. Similarly, a student table will only contain student data.
* No data item will be unnecessarily stored in more than one table. This is to ensure that the data are updated in only one place.
* All attributes in a table are dependent on the primary key – the entire primary key and nothing but the primary key.

To accomplish the objective, the normalization process takes you through the steps that lead to successively higher normal forms. The most common normal forms and their basic characteristics and shown in the table below.

|  |  |
| --- | --- |
| Normal Form | Characteristics |
| First normal form (1NF) | Table format, no repeating groups and PK identified |
| Second normal form (2NF) | 1NF and no partial dependencies |
| Third normal form (3NF) | 2NF and no transitive dependencies |
| Boyce-Codd normal form | Every determinant is a candidate key (special case of 3NF) |
| Fourth normal form | 3NF and no independent multi-valued dependencies |

Even higher-level normal forms exist. However, normal forms such as the fifth normal form (5NF) and domain-key normal form (DKNF) are not likely to be encountered in a business environment.

**Conversion to First Normal Form**

A repeating group derives its name from the fact that a group of multiple entries of the same type can exist for any single key attribute occurrence. A relational table must not contain repeating groups. Identify the primary key.

The term first normal (1NF) describes the tabular format in which:

* All of the key attributes are defined
* There are no repeating groups
* All attributes are dependent on the primary key

**Conversion to Second Normal Form**

Second normal form (2NF) only applies to tables that have composite primary keys (a primary key that is composed of more than one column). For a table to be in 2NF you must eliminate partial dependencies. Non-key columns in the table must be dependent on all parts of the primary key.

A table is in second normal form (2NF) when:

* It is in 1NF
* It contains no partial dependencies

**Conversion to Third Normal Form**

In third normal you must eliminate transitive dependencies. In other word, every non-key column in the table must be dependent on the primary key and only the primary key.

A table is in third normal form (3NF) when:

* It is in 2NF
* It contains no transitive dependencies

**Practical Application** (3points): Collect and enter 30 records with complete data, related to your topic of interest. Perform SELECT statements on the data in a manner that follows you lecture and lab activities, capture and submit screenshots of the resulting tables.

Lesson #6

|  |  |  |
| --- | --- | --- |
| Topic | Creating and Altering Tables | Points |
| Chapters | Six |  |
| Activity | Practical Application: altering tables | 3 |

**Main Topics:**

* Creating tables
* Data types
* Keys and Indexes
* The Auto-Increment feature
* Altering the table structure
  + Adding, removing, and changing columns
  + Adding, removing, and changing indexes
  + Renaming tables
* Dropping tables
* Dropping databases

This section covers topics on tables. You will learn how to:

* Create tables, through introductory examples
* Choose names for tables and table-related structures
* Understand and choose column types
* Understand and choose keys and indexes
* Use the MySQL AUTO\_INCREMENT feature

We will start by showing you how to create a table to hold artist details. Here is the command that we use:

CREATE TABLE artist

( artist\_id SMALLINT(5) NOT NULL,

artist\_name CHAR(128),

PRIMARY KEY (artist\_id));

To verify that a table was created type the following command: SHOW TABLES;

The CREATE TABLE statement has three major sections:

* The CREATE TABLE statement, which is followed by the table name to create. In this example, it’s artist.
* A list of one or more columns to add to the table. In this example we have added two: artist\_id and artist\_name.
* Optional key definitions. In this example, we have defined the artist\_id column as the Primary Key.

Notice that the CREATE TABLE component is followed by an opening parenthesis that’s matched by a closing parenthesis at the end of the statement. Notice also that the other components are separated by commas.

Normally, attempts to create a table with a name that already exists result in an error. If the IF NOT EXISTS clause is specified, the table is not created and no error occurs.

CREATE TABLE IF NOT EXISTS artist

( artist\_id SMALLINT(5) NOT NULL,

artist\_name CHAR(128),

PRIMARY KEY (artist\_id));

Additional features include:

* The AUTO\_INCREMENT feature for numeric columns. This feature allows you to automatically create unique identifiers for a table.
* Foreign key constraints. You can tell MySQL to check whether data in one or more columns matches data in another table. For example, you might to prevent a CD from being added to the music database unless there is a matching artist table entry, this feature is currently supported only by the InnoDB table type.
* Column comments. You can add a comment to a column.

**Column Types**

This section describes the column types you can use in MySQL. The following are commonly used column types in MySQL tables:

* INT [(width)] [UNSIGNED] [ZEROFILL]
  + The most commonly used numeric type. Stores integer values in the range 2,147,483,648 to 2,147,483, 647. If the optional UNSIGNED keyword is added, the range is 0 to 4.294,967,295. The keyword is short for INTEGER. An INT column requires four bytes of storage space. You can also include the optional width and ZEROFILL arguments to left-pad the values with zeros up to the specified length. The maximum width is 255. The width parameter has no effect on what is stored. If you store a value wider than the width, the width value is ignored.
* DECIMAL[(width[,decimals])] [UNSIGNED] [ZEROFILL]
  + DECIMAL stores a fixed-point number, with a total of width digits of which some small number are decimals that follow the decimal point. The width is optional and a value of 10 is assumed when this is omitted. The maximum value of width is 255.
* DATE
  + Stores and displays a date in the format YYYY-MM-DD
* TIME
  + Stores a time in the format HH:MM:SS
* TIMESTAMP
  + Stores and displays a date and time pair in the format YYYY-MM-DD HH:MM:SS
* CHAR[(width)]
  + CHAR stores a fixed-length string. The maximum width is 255.
* VARCHAR(width)
  + Stores variable-length strings up to a maximum width. The maximum value of width is 65,535 characters.
* TEXT
  + A commonly used type for storing large string data objects.

**Keys and Indexes**

A primary key uniquely identifies each row in a table. When you declare one to MySQL, it creates a new file on disk that stores information about where the data from each row in the table is stored. This information is called an index, and its purpose is to speed up searches that use the primary key.

You can create other indexes on the data in a table. You do this so that other searches-on other columns or combinations of columns-are very fast.

CREATE TABLE artist

( artist\_id SMALLINT(5) NOT NULL DEFAULT 0,

artist\_name CHAR(128) DEFAULT NULL,

PRIMARY KEY (artist\_id),

KEY artist\_name (artist\_name));

**The AUTO\_INCREMENT Feature**

MySQL’s AUTO\_INCREMENT feature allows you to create a unique identifier for a row:

CREATE TABLE artist

( artist\_id SMALLINT(5) NOT NULL AUTO\_INCREMENT,

artist\_name CHAR(128) DEFAULT NULL,

PRIMARY KEY (artist\_id));

You can now insert rows, without providing an artist\_id.

The AUTO\_INCREMENT feature has the following requirements:

* The column it is used on must be indexed
* The column that it is used on cannot have a DEFAULT value
* There can only be one AUTO\_INCREMENT column per table

**Altering Structures**

You can use the ALTER TABLE statement to add new columns to a table, remove existing columns, and change column names, types, and lengths.

**Adding, Removing, and Changing Columns**

You can use the ALTER TABLE statement to add new columns to a table, remove existing columns, and change column names, types, and lengths.

The following example will rename a table column. We will change the name of the title column to book\_title in the Title table.

ALTER TABLE Title CHANGE title book\_title CHAR(60);

In the previous example, you can see that we provided four parameters to the ALTER TABLE statement with the CHANGE keyword:

* The table name, Title
* The original column name, title
* The new column name, book\_title
* The column type, CHAR(60)

You must provide all four; that means you need to respecify the type and any clauses that go with it. If you want to modify the type and clauses of a column, but not its name, you can use the MODIFY keyword:

ALTER TABLE artist MODIFY artist\_name CHAR(64) Default "Unknown";

You can also do this with the CHANGE keyword, but by specifying the same column name twice:

ALTER TABLE artist CHANGE artist\_name artist\_name CHAR(64) Default "Unknown";

Suppose you want to add an extra column to an existing table. The following example will accomplish this:

ALTER TABLE artist ADD formed YEAR;

You must supply the ADD keyword, the new column name, and the column type and clauses.

To remove a column, use the DROP keyword followed by the column name. This will get rid of the newly added formed column:

ALTER TABLE artist DROP formed;

**Practical application:** Based upon the lecture and lab information, alter your tables to improve organization. Correct inputted data where appropriate. Describe the changes and your justification for making those changes in a document. Include screenshots of the resulting tables (query the table description) in the document.

Lesson #7

|  |  |  |
| --- | --- | --- |
| Topics | Constraints | Points |
| Chapter | Six |  |
|  |  |  |

**Main Topics:**

* Keys
* Default Values
* Referential Integrity
* Entity Integrity

Constraints are rules that control the data that can be inserted into a table.

Primary Keys

* A primary key is an attribute or combination of attributes that uniquely identifies each row
* Entity Integrity - A primary key is present in a table

Foreign Keys

* A foreign key is a column in a table that refers back to the primary key of another table
* Referential Integrity
* A foreign key must relate to an existing primary key value
* A foreign key can have a NULL value

Default Values

* A value that will be placed in a column if no data is provided

Lesson #8

|  |  |  |
| --- | --- | --- |
| Topic | Storage Engines | Points |
| Chapters | Six and Seven |  |

**Main Topics:**

* Table Types
* Storage Engine Configuration

**MyISAM**

This is the default table type. It is an all-round performer that’s designed for typical applications. It supports very fast querying and has very low overhead for changes to data. It’s also very flexible and it uses whole-table locks.

**Memory or Heap**

Prior to MySQL 4.1, the Memory table type was known as the Heap table type. Both key words are supported. The Memory table type is useful when you want to force data to be in main memory and not on disk. There are serious disadvantages to the Memory type. The most serious is that when you stop and restart a MySQL server, the data stored in a memory table is lost. You must restore it each time you start the MySQL server.

**InnoDB**

The InnoDB is the heavyweight, reliable, high-performance choice for large-scale applications.

Lesson #9

|  |  |  |
| --- | --- | --- |
| Topic | Basic SQL | Points |
| Chapters | Five and Seven |  |
| Activity | Practical Application: querying my data | 3 |
| Project | Two: Simple Queries | 10 |

**Main Topics:**

* The SELECT Statement
* Basic Querying Techniques
* The WHERE clause
* The ORDER BY clause
* The LIMIT clause
* The INSERT Statement
* The DELETE Statement
* The UPDATE Statement
* Using LIKE
* Regular Expressions
* Null values
* The DESCRIBE and SHOW commands
* Using expressions in the SELECT clause

The examples that follow were not done in PHPMyAdmin so your displays will look different. This is normal. The important thing is what the queries are going to produce, not how they display, as this will depend on the tool that you use to execute the query. In order to do this lab you will need to create the Northwoods University and River’s End tables in your MySQL database. Your instructor will show you how to do this.

The following is an example of a simple query. If you use PHPMyAdmin your results will look very different. If you use MySql monitor your results will look very similar.

SELECT \*

FROM course;

COURSE\_ COURSE\_NAME CREDITS

------- ------------------------- ----------

MIS 101 Intro. to Info. Systems 3

MIS 301 Systems Analysis 3

MIS 441 Database Management 3

CS 155 Programming in C++ 3

MIS 451 Web-Based Systems 3

This query selects rows from the course table. The asterisk (\*) tells MySQL to display all the columns from the course table.

The following query produces an identical result table by listing all the columns from the course table.

SELECT course\_no, course\_name, credits

FROM course;

You can list columns individually separated by commas. The SELECT statement also specifies the table name in the FROM clause. The result table will display the columns in the order specified in the SELECT clause.

Although SQL does not require it, a new line should be started for each clause in a SELECT command. The following keywords are your signal to start a new line:

* SELECT
* FROM
* WHERE
* GROUP BY
  + HAVING
* ORDER BY

Using the asterisk (\*) is an easy way to list all the columns in a table. However, in most queries you will not need to display all the columns. If you want to display just the student’s last name, first name and class, you could use the following query:

SELECT s\_last, s\_first, s\_class

FROM student;

S\_LAST S\_FIRST S\_

------------------------------ ------------------------------ --

Jones Tammy SR

Perez Jorge SR

Marsh John JR

Smith Mike SO

Johnson Lisa SO

Nguyen Ni FR

6 rows selected.

The rules for writing a simple select are:

* Specify the column names you want to display by typing the exact, complete column names.
* Separate each column name with a comma (,).
* Specify the table name after the FROM clause.
* Terminate the query with a semicolon (;).

**The WHERE clause**

Specific rows can be selected by adding a WHERE clause to a query. The WHERE clause will limit the rows that are selected for display.

SELECT \*

FROM location

WHERE bldg\_code = ‘BUS’;

LOC\_ID BLDG\_CODE ROOM CAPACITY

---------- ---------- ------ ----------

5 BUS 105 42

6 BUS 404 35

7 BUS 421 35

8 BUS 211 55

9 BUS 424 1

10 BUS 402 1

11 BUS 433 1

7 rows selected.

The result table contains only rows where the condition is met.

**Common MySQL Comparison Operators:**

|  |  |
| --- | --- |
| = | equal to |
| < | less than |
| > | greater than |
| >= | greater than or equal to |
| <= | less than or equal to |
| <> | not equal to |
| != | not equal to |

When you use a WHERE clause the column used in the condition does not have to be included in the select clause. You can use the WHERE clause to select any subset of the rows in a table, from no rows selected to all rows selected.

**Logical Operators (AND, OR, and NOT)**

The logical operators AND, OR, and NOT add power to your queries. The additional power comes from combining more than one condition in the WHERE clause. The NOT operator permits the creation of simple queries that otherwise would be unnecessarily bulky or complex.

* AND: joins two or more conditions, and returns results only when all of the conditions are true.
* OR: joins two or more conditions, and returns results when any of the conditions are true.
* NOT: negates the expression that follows it.

**The AND Operator**

The AND operator links two or more conditions in a WHERE clause. Suppose you want a list of locations in the Business building that have a capacity of more than 35 and you want the list sorted by room number.

SELECT room, capacity

FROM location

WHERE bldg\_code = ‘BUS’ and capacity > 35

ORDER BY room;

ROOM CAPACITY

------ ---------

105 42

211 55

**The OR Operator**

Suppose that you need a list of locations that are in either the Library or the Computer Resource building.

SELECT bldg\_code, room, capacity

FROM location

WHERE bldg\_code = ‘CR’ OR bldg\_code = ‘LIB’;

BLDG\_CODE ROOM CAPACITY

---------- ------ ---------

CR 101 150

CR 202 40

CR 103 35

CR 105 35

LIB 217 2

LIB 222 1

6 rows selected.

**The NOT Operator**

Now let’s say you need a list of locations in a specified building, such as the Library. The WHERE clause is straightforward:

WHERE bldg\_code = ‘LIB’

However, what if you need a list of locations not in the Library, the OR operator can be used to specify all locations except the library.

WHERE bldg\_code = ‘CR’ or bldg\_code = ‘BUS’

The WHERE clause above could get out of control if there were many other buildings besides the Library. Let’s use the NOT operator instead:

SELECT bldg\_code, room, capacity

FROM location

WHERE NOT bldg\_code = ‘LIB’;

BLDG\_CODE ROOM CAPACITY

---------- ------ ----------

CR 101 150

CR 202 40

CR 103 35

CR 105 35

BUS 105 42

BUS 404 35

BUS 421 35

BUS 211 55

BUS 424 1

BUS 402 1

BUS 433 1

11 rows selected.

**Combining OR and AND Operators**

You can combine OR and AND operators to create complex queries, but you must exercise caution. There are pitfalls when combining the AND and OR operators.

You need a list of locations that are in the Library or the Business building and have a capacity greater than 35.

SELECT bldg\_code, room, capacity

FROM location

WHERE capacity > 35 AND bldg\_code = ‘BUS’ OR bldg\_code = ‘LIB’;

BLDG\_CODE ROOM CAPACITY

---------- ------ ----------

BUS 105 42

BUS 211 55

LIB 217 2

LIB 222 1

The SELECT statement executes, but is it correct? The answer is no, the result table should not display the locations from the Library because they do not have a capacity greater than 35.

Where does this query go wrong? The answer is that there is an order of precedence for the logical operators. When the AND operator is combined with the OR operator, MySQL evaluates the conditions connected by the AND operator first. Following this MySQL next evaluates the conditions connected by the OR operator. Let’s break the WHERE clause into pieces in order to see how the conditions are evaluated.

There are three conditions to be evaluated. MySQL begins by examining the two conditions connected by the AND operator. With this WHERE clause, a table row will be included in the result table if both conditions are true. MySQL returns a TRUE or FALSE from the AND operation.

Condition 1 AND Condition 2

capacity > 35 AND bldg\_code = ‘BUS’

Next, MySQL combines the result of the AND operation with the remaining condition in an OR operation. Let’s suppose that the result of the AND operation was TRUE for a given row. The resulting OR operation looks like the following to MySQL.

TRUE OR Condition 3

TRUE OR bldg\_code = ‘LIB’

You can force a change in the order of evaluation for a complex condition by using parentheses to force the order of operation.

The following is the correct rewritten query:

SELECT bldg\_code, room, capacity

FROM location

WHERE capacity > 35 AND (bldg\_code = ‘BUS’ OR bldg\_code = ‘LIB’);

BLDG\_CODE ROOM CAPACITY

---------- ------ ---------

BUS 105 42

BUS 211 55

MySQL evaluates the complex condition beginning with the inner most set of parentheses. Here, MySQL will first test a row to see if bldg\_code is ‘BUS’ OR ‘LIB’. If a row passes that test, the query then tests to see if the capacity is greater than 35.

**The ORDER BY clause**

When rows are added to a table, they are normally appended to the end of the table. This produces a table with unordered rows.

When displaying many rows from a table it may be necessary to sort the output. Output from a SELECT statement can be sorted by using the optional ORDER BY clause. When you use the ORDER BY clause, the column name on which you are ordering must be included in the SELECT clause.

SELECT \*

FROM location

ORDER BY bldg\_code;

LOC\_ID BLDG\_CODE ROOM CAPACITY

---------- ---------- ------ ----------

7 BUS 421 35

11 BUS 433 1

10 BUS 402 1

9 BUS 424 1

8 BUS 211 55

6 BUS 404 35

5 BUS 105 42

4 CR 105 35

3 CR 103 35

2 CR 202 40

1 CR 101 150

12 LIB 217 2

13 LIB 222 1

13 rows selected.

You can also sort data based on a numeric column value.

SELECT \*

FROM location

ORDER BY capacity;

LOC\_ID BLDG\_CODE ROOM CAPACITY

---------- ---------- ------ ----------

9 BUS 424 1

13 LIB 222 1

11 BUS 433 1

10 BUS 402 1

12 LIB 217 2

6 BUS 404 35

4 CR 105 35

3 CR 103 35

7 BUS 421 35

2 CR 202 40

5 BUS 105 42

8 BUS 211 55

1 CR 101 150

13 rows selected.

By default, the ORDER BY clause will sort output rows in ascending order. However, there are times when you will need to display the results in descending order. Let’s rewrite the two previous examples to sort in descending order.

We will use the DESC keyword to change to descending order.

SELECT \*

FROM location

ORDER BY bldg\_code DESC;

LOC\_ID BLDG\_CODE ROOM CAPACITY

---------- ---------- ------ ----------

12 LIB 217 2

13 LIB 222 1

2 CR 202 40

1 CR 101 150

4 CR 105 35

3 CR 103 35

11 BUS 433 1

10 BUS 402 1

9 BUS 424 1

8 BUS 211 55

6 BUS 404 35

5 BUS 105 42

7 BUS 421 35

13 rows selected.

SELECT \*

FROM location

ORDER BY capacity DESC;

LOC\_ID BLDG\_CODE ROOM CAPACITY

---------- ---------- ------ ----------

1 CR 101 150

8 BUS 211 55

5 BUS 105 42

2 CR 202 40

6 BUS 404 35

4 CR 105 35

3 CR 103 35

7 BUS 421 35

12 LIB 217 2

9 BUS 424 1

10 BUS 402 1

11 BUS 433 1

13 LIB 222 1

13 rows selected.

When the ASC (Ascending) or DESC (Descending) optional keywords are used, they must follow the column name on which you are sorting. Since the default is ascending order, the ASC keyword is rarely used.

There are situations where it will be necessary to sort on more than one column.

SELECT \*

FROM location

ORDER BY bldg\_code, room;

LOC\_ID BLDG\_CODE ROOM CAPACITY

---------- ---------- ------ ----------

5 BUS 105 42

8 BUS 211 55

10 BUS 402 1

6 BUS 404 35

7 BUS 421 35

9 BUS 424 1

11 BUS 433 1

1 CR 101 150

3 CR 103 35

4 CR 105 35

2 CR 202 40

12 LIB 217 2

13 LIB 222 1

13 rows selected.

In the example, the major sort key is the bldg\_code and within each bldg\_code, the locations are sorted by room (the minor key).

**Character Matching (LIKE and NOT LIKE)**

LIKE and NOT LIKE operators can be used to search for data rows containing incomplete or partial character strings within a data column.

The following example searches the Title table for titles that begin with the characters ‘Th’

SELECT title, retail\_price

FROM title

WHERE title LIKE ‘Th%’;

TITLE RETAIL\_PRICE

--------------------------------------------- ------------

The C Programming Language 39.95

The Sword of Shannara Trilogy 24.95

The Ivy Tree 6.99

The Legend of the Seventh Virgin 15.98

Thyme of Death 6.95

The English Breakfast Murder 6.95

The Jasmine Moon Murder 16.95

The Last Camel Died at Noon 6.75

The Cleansing 32.5

9 rows selected.

The percent (%) sign is a wild card symbol used to represent zero or more characters. The underscore (\_) is also a wild card symbol, but it represents a single character.

There is a notable limitation when using wild card characters. You cannot use the comparison operators (=, >, <, etc.). Wild card characters used without the LIKE operator are interpreted as characters that you want to search for.

**Regular Expressions:**

MySQL also uses regular expressions to do pattern matching. The operator is REGEXP rather than LIKE. Your Instructor will show you some of the most common pattern characters.

**Unknown Values (IS NULL and IS NOT NULL)**

The term NULL is a keyword meaning the absence of any stored value. If a column in a data row is NULL, then there is no value stored in that column. The NULL or NOT NULL condition cannot be tested using comparison operators. You must use the IS NULL or IS NOT NULL operators.

**Example WHERE clauses:**

WHERE wage IS NULL

WHERE title IS NOT NULL

**Using Expressions in SELECT Clauses**

Thus far, we have used the SELECT clause of a SELECT statement to retrieve data and display result tables for data columns from a specific table. However, a SELECT clause can also contain expressions, or computed columns. Expressions, or computed columns can also be used in a WHERE clause to manipulate column data. An expression is formed by combining a column name or constant with an arithmetic operator. The SQL arithmetic operators are shown below:

|  |  |
| --- | --- |
| \* | Multiplication |
| / | Division |
| % | Modulus |
| + | Addition |
| - | Subtraction |

Example:

SELECT order\_id, isbn, qty \* price

FROM order\_detail;

ORDER\_ID ISBN QTY\*PRICE

---------- ---------- ----------

1 0870541811 32.5

1 0870541781 32.5

2 0449242463 6.75

3 0425198138 30.52

3 0380820765 5.39

4 0140296301 5.95

5 0345470389 5.56

6 0345345991 21.39

6 0345453751 23.7

7 0688010377 7.95

8 0670032786 11.9

8 0812574990 7.5

9 0131103628 35.96

9 0619016620 23.4

10 1878252402 32.78

11 0785269606 19.96

12 0449214222 14.95

12 0688010377 15.9

18 rows selected.

You will notice that the column header for the computed column just displays the expression used to calculate the column. We will assign an alias to the computed column to improve its appearance. A column alias is a string that replaces the column name when that column is displayed in a result table.

SELECT order\_id, isbn, qty \* price As "Extended Price"

FROM order\_detail;

ORDER\_ID ISBN Extended Price

---------- ---------- --------------

1 0870541811 32.5

1 0870541781 32.5

2 0449242463 6.75

3 0425198138 30.52

3 0380820765 5.39

4 0140296301 5.95

5 0345470389 5.56

6 0345345991 21.39

6 0345453751 23.7

7 0688010377 7.95

8 0670032786 11.9

8 0812574990 7.5

9 0131103628 35.96

9 0619016620 23.4

10 1878252402 32.78

11 0785269606 19.96

12 0449214222 14.95

12 0688010377 15.9

18 rows selected.

Parentheses are optional for a computed column. If you want to make the calculation clearer, you can enclose the expression in parentheses. As show below:

SELECT order\_id, isbn, (qty \* price) As "Extended Price"

FROM order\_detail;

**Practical application:** Based upon the lecture and lab information, perform several queries on your table. Submit the query statements and screenshots of the resulting outputs. You must include compound query statements, and specify columns. **Project Two: Simple Queries - 10 Points**

You will need to write SELECT commands to answer the following questions. Print the SELECT command and your results using PHPMyAdmin. Your instructor will give show you how to create the tables. The table descriptions start on Page 111.

1. Select all the rows from the Item table. Display Description and Category
2. Select inventory items that have a price of less than 100 dollars. Display Inventory\_id, Size, Price and QOH
3. List the inventory items that have a QOH of more than 30. Display Inventory\_id, QOH and Price
4. List the customers in ‘Washburn ’ and ‘Silver Lake’. Display first\_name, last\_name, mi and city
5. Select the prices that occur in the Inventory table. A specific price should only appear once. Display the Price
6. Select the inventory items that are in stock. Display Inventory\_id, Price and QOH
7. Select the customer orders placed before November 1, 2007. Display the Order\_id and Order\_date
8. List the inventory items that are ‘Coral’ or ‘Olive’ and have a QOH of less than 105. Display Inventory\_id and QOH
9. List the items that contain the word ‘Fleece’ in the item description. Display Item\_id, description, and Category
10. List all the inventory items that do not have a size or color assigned. Display the Inventory\_id and Price
11. Determine the number of orders placed on 10 October 2007. Display Number of Orders

Hint: Use the COUNT function

1. Determine the extended price for each row in the Orderline table. Display Order\_id, Inventory\_id, and Extended Price
2. Determine the number of different items on each order. Display Order\_ID and Number of Items Hint: Determine the number of different products ordered, not the total quantity ordered. This query requires a GROUP BY clause.
3. Determine the number of orders placed by each customer. Only display the data for customers who have placed more than one order. Display Cust\_id and Number of Orders Hint: This query requires a GROUP BY clause and a HAVING clause.
4. Determine the order total for each order that has an order total greater than 100. Display ‘*Order Id*’ and ‘*Order Total’.* Make sure the results are in ascending order total sequence.
5. Determine what is the most expensive price, the least expensive price, and the average price in the inventory table.
6. Now that you know the average price in the inventory table, display all of the information for inventory items whose price is greater than the average price.

Lesson #10

|  |  |  |
| --- | --- | --- |
| Topic | Advanced SQL | Points |
| Chapters | Five and Seven |  |

Main Topics:

* Joining tables
* Aliases
* Data Grouping
* The DISTINCT clause
* GROUP BY and HAVING
* The INNER JOIN clause
* The UNION Statement
* Left and Right Joins
* The Natural Join

You will often need information that requires the retrieval of data from related rows that are stored in more than one table. These queries are called Join operations.

The related tables of a large database are linked by foreign keys or what are often referred to as common columns. The ability to join tables will enable you to add more meaning to the result table that is produced by your query. The learning objectives for this lesson are:

* Write a join query using the WHERE clause.
* Learn the basic join operation rules.
* Write complex join queries with more than two tables.

MySQL uses the WHERE clause to join tables. We want to see all the book titles and the names of the publishers who published them. Look at the example below:

SELECT title, name As "Publisher Name"

FROM title, publisher

WHERE title.pub\_id = publisher.pub\_id

TITLE Publisher Name

--------------------------------------------- ------------------------

The C Programming Language Prentice Hall

Aunt Dimity's Christmas Viking Press

Crewel Lye Del Rey

The Sword of Shannara Trilogy Del Rey

On the Night of the Seventh Moon Fawcett Books

Back Roads and Hidden Places Sunset Books

The Ivy Tree Harper Torch

Nine Coaches Waiting Harper Torch

The Legend of the Seventh Virgin Fawcett Books

Thyme of Death Berkley Publisher Group

Witches' Bane Berkley Publisher Group

The English Breakfast Murder Prime Crime

A Dilly of a Death Berkley Publisher Group

The Jasmine Moon Murder Prime Crime

Crocodile on the Sandbank Mysterious Press

The Last Camel Died at Noon Mysterious Press

A Walk in Wolf Wood Harper Torch

Snowfire Fawcett Books

Programming with Visual Basic .Net Thomson Learning

Aunt Dimity Snowbound Viking Press

Airs Above the Ground Harper Torch

TITLE Publisher Name

--------------------------------------------- ------------------------

Brrr! Greenwillow Books

A City Under the Sea Atheneum Books

A Village Christmas Thomas Nelson Publishers

Winter Wheat Henry Holt and Company

Up in a Heaval Tor Fantasy

Dragonfly Arkham House Publishers

In The Stone House Arkham House Publishers

The Cleansing Arkham House Publishers

Shadows over Innsmouth Arkham House Publishers

Whispers in the Night Arkham House Publishers

31 rows selected.

Now that you have seen a basic join, it’s time to learn what SQL is doing for you behind the scenes. When two or more tables are joined, SQL creates a Cartesian product for the tables. A Cartesian product consists of all possible combinations of the rows from each of the tables. Therefore, when a table with 10 rows is joined with a table with 20 rows, the Cartesian product is 200 rows (10 x 20 = 200). Let’s say we have the following tables:

testcust:

CUST\_ID NAME

--------- ------

1 Joe

2 Tim

3 Fred

2 rows selected.

and testorder:

ORDER\_ID CUST\_ID

--------- ---------

1 1

2 2

3 3

3 rows selected.

We want to know the name of the customer that placed each order, so we can select all the columns from both tables. For example:

SELECT \*

FROM testcust, testorder;

CUST\_ID NAME ORDER\_ID CUST\_ID

-------- ------ --------- ---------

1 Joe 1 1

2 Tim 1 1

3 Fred 1 1

1 Joe 2 2

2 Tim 2 2

3 Fred 2 2

1 Joe 3 3

2 Tim 3 3

3 Fred 3 3

9 rows selected.

A Cartesian product of these tables yields a result table with nine rows. A Cartesian product result table is normally not very useful, In fact, it can be very misleading, the result table implies that every customer has a relationship with every order, and we know that this is simply not the case. The previous example requires a WHERE clause to specify the nature of the relationship between the two tables. The following is the revised query with the correct result table:

SELECT \*

FROM testcust, testorder

WHERE testcust.cust\_id = testorder.cust\_id

CUST\_ID NAME ORDER\_ID CUST\_ID

--------- ------ --------- ---------

1 Joe 1 1

2 Tim 2 2

3 Fred 3 3

3 rows selected.

The WHERE clause filters out the rows that are not related. The join condition only displays the record if the customer ids are the same.

**Qualifying Column Names and Table Aliases**

What if we just want to show orders that were placed by customer #1. Look at the following:

SELECT \*

FROM testcust, testorder

WHERE testcust.cust\_id = testorder.cust\_id

AND testcust.cust\_id = 1;

CUST\_ID NAME ORDER\_ID CUST\_ID

------- ------- --------- ---------

1 Joe 1 1

1 row selected.

The important point to remember from this query is that the WHERE clause can be used to join tables and filter rows in the same query.

Normally you do not need to create alias names for tables, this is optional. However, when column names are ambiguous, you must qualify them. A column name is ambiguous when the same column name is used in different tables.

The ambiguous column name is qualified by using the dot (.) connector to connect the table name with the column name. Sometimes, it is easier to qualify column names by using table alias names. Often, a single letter is used as the alias.

SELECT \*

FROM testcust cu, testorder ord

WHERE cu.cust\_id = ord.cust\_id

AND cu.cust\_id = 1;

CUST\_ID NAME ORDER\_ID CUST\_ID

--------- ------ --------- ---------

1 Joe 1 1

1 row selected.

The alias name must follow a table name. Use a space to separate the table and its alias name. The alias name must be unique within the SELECT statement. Once you have defined an alias you must use it. The original table name will not work.

**More Than Two Tables**

Often when more than two tables are involved in a join you are trying to implement a many-to-many relationship. You will need to include the linking table. The following query displays the book title and the author that wrote it:

SELECT title, full\_name

FROM title t, author a, author\_title at

WHERE t.isbn = at.isbn

AND a.author\_id = at.author\_id;

TITLE FULL\_NAME

--------------------------------------------- -------------------

The C Programming Language Dennis Ritchie

A City Under the Sea Norbert Wu

Crocodile on the Sandbank Elizabeth Peters

The Last Camel Died at Noon Elizabeth Peters

Brrr! James Stevenson

A Village Christmas Thomas Kinkade

TITLE FULL\_NAME

--------------------------------------------- -------------------

Winter Wheat Brenda Guiberson

Back Roads and Hidden Places Barbara Braasch

Whispers in the Night Basil Cooper

Shadows over Innsmouth Ramsey Campbell

The Cleansing John Harvey

Crewel Lye Piers Anthony

Up in a Heaval Piers Anthony

The Sword of Shannara Trilogy Terry Brooks

Thyme of Death Susan Albert

Witches' Bane Susan Albert

A Dilly of a Death Susan Albert

In The Stone House Barry Malzberg

Dragonfly Federic Durbin

Aunt Dimity's Christmas Nancy Atherton

Aunt Dimity Snowbound Nancy Atherton

The English Breakfast Murder Laura Childs

The Jasmine Moon Murder Laura Childs

The Ivy Tree Mary Stewart

Nine Coaches Waiting Mary Stewart

A Walk in Wolf Wood Mary Stewart

Airs Above the Ground Mary Stewart

Snowfire Phyllis Whitney

On the Night of the Seventh Moon Victoria Holt

The Legend of the Seventh Virgin Victoria Holt

Programming with Visual Basic .Net Diane Zak

31 rows selected.

The following is a methodology that may help you when you need to join tables together:

* Determine the columns that need to be displayed
* Determine the tables that the above columns are in
* Determine the number of join conditions required (Number of Tables – 1)
* Write the join conditions using common columns
* Make sure that every table is part of at least one join condition

**The Distinct Clause**

If you select a column in a table other than the primary key, you may get duplicate values. The following produces a list of building codes.

SELECT bldg\_code

FROM location;

BLDG\_CODE

----------

CR

CR

CR

CR

BUS

BUS

BUS

BUS

BUS

BUS

BUS

LIB

LIB

13 rows selected.

When row output is duplicated, it is possible to obscure relevant data. The query can be re-written using the DISTINCT keyword to eliminate duplicate rows.

SELECT DISTINCT bldg\_code

FROM location;

BLDG\_CODE

----------

BUS

CR

LIB

The DISTINCT keyword must immediately follow the SELECT keyword and is not separated from the first column name with a comma.

When more than one column is selected in a query with a DISTINCT clause, you may not get the results you expect.

SELECT DISTINCT bldg\_code, room

FROM location;

BLDG\_CODE ROOM

---------- ------

BUS 105

LIB 101

BUS 421

LIB 217

CR 101

BUS 433

LIB 222

CR 103

CR 105

BUS 402

BUS 201

BUS 211

CR 202

13 rows selected.

Notice that when you run this query you get all 13 rows in the location table. This is because the DISTINCT clause affects all the columns in the SELECT clause. All the rows are displayed because even though the building codes are the same, the room numbers are different

Queries are often needed to provide group information to support business decisions. For a university typical questions include:

* How many credit hours does a student have?
* How many classes has a student taken?
* How many classes is a student taking now?
* What is a student’s GPA?

SQL includes numerous predefined group functions to produce this kind of information.

* In this section, you will learn to use many of SQL’s group functions to write specialized queries. You will also learn the GROUP BY and HAVING clauses. The GROUP BY clause specifies how to group rows from a data table when grouping information. The HAVING clause filters out groups that do not meet specified conditions.

**Implementing Group Functions**

Group functions perform a variety of actions such as counting all the rows in a table, averaging a column’s data, and summing numeric data. Group functions can also search a table to find the highest "MAX" or lowest "MIN" values in a column. As with other types of queries, you can restrict, or filter out the rows these functions act on with the WHERE clause. If a school dean needs to know how many students attend Northwoods University, the group function COUNT(\*) can be used to produce the information. The COUNT(\*) function shown below counts all rows in a table. The (\*) is used as the parameter to the function; it means count all the rows:

SELECT COUNT(\*)

FROM student;

COUNT(\*)

---------

6

The result table for the COUNT(\*) function is a single value. Notice that the result table has a column heading that corresponds to the name of the group function used in the SELECT clause. The output column can be assigned a more meaningful name as shown below. Here an alias is assigned to the group function. If this alias is more than one word, it must be enclosed in double-quotes.

SELECT COUNT(\*) As "Number of Students"

FROM student;

Number of Students

------------------

6

The table below lists some of the commonly used group functions including their syntax and use:

| Function Syntax | Function Use |
| --- | --- |
| SUM([ALL | DISTINCT] expression) | The total of the (distinct) values in a numeric column/expression |
| AVG([ALL | DISTINCT] expression) | The average of the (distinct) values in a numeric column/expression |
| COUNT([ALL | DISTINCT] expression) | The number of (distinct) non-NULL values in a column/expression |
| COUNT(\*) | The number of selected rows |
| MAX(expression) | The highest value in a column/expression |
| MIN(expression) | The lowest value in a column/expression |

The ALL and DISTINCT keywords are optional. The ALL keyword is the default. The expression listed can be a constant, a function, or any combination of column names, constants, and functions connected by arithmetic operators. However, group functions are most often used with a column name. Group functions can be used in the SELECT, HAVING, and ORDER BY clauses.

Group functions cannot be used in a WHERE clause.

**Using the AVG Function**

The AVG function averages the values in a specified numeric column. You can use the AVG function to compute the average value for the retail\_price column in the Title table. The query below returns the average of retail prices. The output column in the result table is renamed "Average Retail Price" and formatted.

SELECT AVG(retail\_price) As "Average Retail Price"

FROM title;

Average Retail Price

--------------------

$14.78

Using the SUM Function

The SUM function adds the values in a specified numeric column together.

Using the SUM function, we can determine the total number of books in stock:

SELECT SUM(qoh) As "Total Books"

FROM title;

Total Books

-----------

192

**Using the Minimum (MIN) and Maximum (MAX) Functions**

The example below uses the MIN and MAX functions to determine the smallest and largest qoh for the Title table:

SELECT MIN(qoh) As "Smallest QOH", MAX(qoh) As "Largest QOH"

FROM title;

Smallest QOH Largest QOH

------------ -----------

0 23

**Using the COUNT Function**

At the beginning of this lab, we used the COUNT(\*) function to count the number of rows in a table. The COUNT(column\_name) function does almost the same thing, the difference is that you define a specific column to be counted. When the COUNT function processes a specific column, rows containing a NULL value in the specified column are omitted from the count. The query below counts the number of publishers that have a contact assigned:

SELECT COUNT(contact)

FROM publisher;

COUNT(CONTACT)

--------------

13

In contrast, the next query counts each publisher regardless of NULL values.

SELECT COUNT(\*)

FROM publisher;

COUNT(\*)

---------

16

**Using GROUP BY with Group Functions**

Now that you are familiar with group functions, you’re ready to add power to your queries. The power of group functions is increased when combined with the GROUP BY clause. In fact the GROUP BY clause is rarely used without a group function.

When properly used, the GROUP BY clause enables you to use group functions to answer complex questions such as:

* What is the average retail price for each type of book?
* How many books of each type are in stock?
* How many books were written by each author?

The following query determines how many books of each type are in our database.

SELECT type, COUNT(\*) As "Number of Books"

FROM title

GROUP BY type;

TYPE Number of Books

--------------- ---------------

Art 1

Children 3

Fantasy 4

Gothic 4

Horror 3

Mystery 9

Programming 2

Romance 3

Science Fiction 1

Travel 1

10 rows selected.

If your SELECT clause contains both column names and group functions then you must have a GROUP BY in your query. The column name(s) in the GROUP BY clause must match the column name(s) listed in the SELECT clause. A simple way to remember what goes in a GROUP BY clause is that everything in the SELECT clause except for group functions must be in the GROUP BY clause.

**Using GROUP BY with a WHERE clause**

You can combine the WHERE and GROUP BY clauses in a SELECT statement. The WHERE clause works to eliminate data table rows before the data grouping takes place.

SELECT type, AVG(retail\_price) As "Average Retail Price"

FROM title

WHERE type IN (‘Mystery’, ‘Gothic’, ‘Horror’)

GROUP BY type;

TYPE Average Retail Price

--------------- --------------------

Gothic 8.97

Horror 32.316667

Mystery 8.9055556

In the preceding query only the groups designated by the IN clause are returned.

**Using GROUP BY with an ORDER BY**

MySQL automatically sorts the result table by the column(s) that are included in the GROUP BY clause, if you want to change the sort order, you can specify an ORDER BY clause after the GROUP BY clause.

SELECT type, AVG(retail\_price) As "Average Retail Price"

FROM title

WHERE type IN (‘Mystery’, ‘Gothic’, ‘Horror’)

GROUP BY type

ORDER BY AVG(retail\_price);

TYPE Average Retail Price

--------------- --------------------

Mystery 8.9055556

Gothic 8.97

Horror 32.316667

**Using GROUP BY with a HAVING clause**

The HAVING clause is used for group functions in the same way that a WHERE clause is used for column names and expressions.

The difference between the WHERE and HAVING clauses is:

* A WHERE clause is used to filter rows before the grouping action.
* A HAVING clause filters groups after the grouping action

SELECT type, AVG(retail\_price) As "Average Retail Price"

FROM title

WHERE type IN (‘Mystery’, ‘Gothic’, ‘Horror’)

GROUP BY type

HAVING AVG(retail\_price) > 9.00

ORDER BY AVG(retail\_price);

TYPE Average Retail Price

--------------- --------------------

Horror 32.316667

A HAVING clause cannot be used without a GROUP BY clause.

**Inner Joins:**

An inner join is also known as a simple join or a equi-join. This is accomplished using the INNER JOIN operator. The INNER JOIN results in rows being retrieved when an exact match between key columns is made.

INNER JOIN requires the use of either the USING or ON clause to identify the joined columns.

INNER JOIN with USING Example:

The USING clause explicitly identifies the common column between the two tables for the INNER JOIN. This may be a recommended self-documenting standard. For example:

SELECT pub\_id, name, title

FROM title

INNER JOIN publisher

USING (pub\_id)

**Inner Join with On Example:**

This also might be a recommended method of coding, as this style is self-documenting. The ON clause specifically identifies the join column, thus potentially leaving other selection criteria for a subsequent WHERE clause if desired.

Note the use of table name as column prefixes to resolve any possible ambiguity with the column pub\_id.

SELECT publisher.pub\_id, name, title

FROM title

INNER JOIN publisher

ON publisher.pub\_id = title.pub\_id

**Outer Joins:**

The following tables are used to demonstrate outer joins:

DEMO\_TITLE

Description

|  |  |  |
| --- | --- | --- |
| Name | Null? | Type |
| TITLE |  | VARCHAR(10) |
| TYPE\_ID |  | INT |

Data

|  |  |
| --- | --- |
| TITLE | TYPE\_ID |
| A | 1 |
| B | 2 |
| C | 3 |
| D |  |

DEMO\_TYPE

Description

|  |  |  |
| --- | --- | --- |
| Name | Null? | Type |
| TYPE\_ID |  | INT |
| TYPE |  | VARCHAR(10) |

Data

|  |  |
| --- | --- |
| TYPE\_ID | TYPE |
| 1 | Gothic |
| 2 | Mystery |
| 3 | Horror |
| 4 | Children |

**Note:** In the DEMO\_TITLE table, title "D" does not have a type\_id assigned and also in the DEMO\_TYPE table, type "Children" is not used in the DEMO\_TITLE table. If you do an INNER JOIN on these tables the record in the DEMO\_TITLE table that does not have a type\_id and the record in the type table that is not used will not show up in the result table. For example:

SELECT title, type

FROM demo\_title NATURAL JOIN demo\_type

|  |  |
| --- | --- |
| TITLE | TYPE |
| A | Gothic |
| B | Mystery |
| C | Horror |

The following command is an example of a LEFT OUTER JOIN, which returns rows from the DEMO\_TITLE table (the one on the left of the join operation) will be returned even if there is no matching row in the DEMO\_TYPE table (which is the table on the right side of the join). A NULL row is added to the DEMO\_TYPE table to match any title not assigned a type\_id.

SELECT title, type

FROM demo\_title

LEFT OUTER JOIN demo\_type

ON demo\_title.type\_id = demo\_type.type\_id

|  |  |
| --- | --- |
| TITLE | TYPE |
| A | Gothic |
| B | Mystery |
| C | Horror |
| D |  |

The example shown below illustrates a RIGHT OUTER JOIN. Rows from the RIGHT table of the join operation (DEMO\_TYPE) will be returned, even if there is no matching row in the DEMO\_TITLE table. A NULL row is added to the DEMO\_TITLE to match any types that are not used.

SELECT title, type

FROM demo\_title

RIGHT OUTER JOIN demo\_type

ON demo\_title.type\_id = demo\_type.type\_id

|  |  |
| --- | --- |
| TITLE | TYPE |
| A | Gothic |
| B | Mystery |
| C | Horror |
|  | Children |

**The Natural Join:**

A natural join means that you tell MySQL what tables you want to join, and it figures out how to do it and gives you an INNER JOIN result set.

SELECT title, name

FROM title NATURAL JOIN publisher;

All MySQL does is look for columns with the same name and, behind the scenes, adds these silently into an inner join with a USING clause. So, the above query is translated into:

SELECT title, name

FROM title INNER JOIN publisher USING (pub\_id);

The mystery makes natural joins worth avoiding; spell out queries using an INNER JOIN or a WHERE clause instead.

Lesson #11

|  |  |  |
| --- | --- | --- |
| Topic | Adding More to Queries | Points |
| Chapter | Seven |  |
| Project | Three: Advanced Queries | 15 |

**Main Topics:**

* The IN operator
* Nested Queries (Subqueries)
* The BETWEEN operator
* ANY and ALL
* The EXISTS operator

**The IN Operator**

Until this point, all queries have required you to compare the value stored in a single column of a table with another single value.

In order to compare a column against several values, it is necessary to use the OR operator to combine multiple conditions.

You want a listing of faculty members who have a rank of ‘Assistant’, ‘Full’, or ‘Associate’. The query can be written as follows:

SELECT f\_first, f\_last, f\_rank

FROM faculty

WHERE f\_rank = ‘Assistant’ OR f\_rank = ‘Full’ OR f\_rank = ‘Associate’;

F\_FIRST F\_LAST F\_RANK

------------------------- ------------------------- ---------

Teresa Marx Associate

Mark Zhulin Full

Colin Langley Assistant

Jonnel Brown Full

James Sealy Associate

The use of the OR is unnecessarily complex. The IN operator can simplify the query. The revised query tests to see if the frank column matches any of the values in the list that is enclosed in parentheses.

SELECT f\_first, f\_last, f\_rank

FROM faculty

WHERE f\_rank IN (‘Assistant’,‘Full’,‘Associate’);

Because the WHERE clause has less code, the query is easier to read. Notice that each value is separated by a comma.

**The NOT IN Operator**

The NOT IN operator returns rows that have values that do not match the values in the list.

We will revise the previous query, so that it will select rows in the table that do not match the values in the list.

SELECT f\_first, f\_last, f\_rank

FROM faculty

WHERE f\_rank NOT IN (‘Full’,‘Associate’);

F\_FIRST F\_LAST F\_RANK

--------------------------- --------------------------- ---------

Colin Langley Assistant

Thus far, you have learned to write queries where all the information needed to specify retrieval criteria is known at design time. The term design time simply means that you are in the process of writing a query. This contrasts with run time, which refers to the actual execution and processing of a query. In this section you will learn about subqueries. A subquery is a query within a query. Subqueries enable you to write queries that select data rows from criteria that are actually developed while the query is executing.

Suppose you need to write a query where the criteria values to be used are unknown at design time. For example, consider a requirement to list the titles of books that have a price equal to the minimum retail price. The problem is that at design time, you do not know what the minimum retail price is! Further, over time, the minimum retail price will surely change. You could break this query into two parts by first writing a query to determine the minimum retail price:

SELECT MIN(retail\_price)

FROM title;

MIN(RETAIL\_PRICE)

-----------------

4.95

Next, you could substitute the 4.95 for the minimum retail price in the WHERE clause. As shown below:

SELECT title

FROM title

WHERE retail\_price = 4.95;

TITLE

---------------------------------------------

A City Under the Sea

However, the subquery approach allows you to combine these two queries into one.

SELECT title

FROM title

WHERE retail\_price = (SELECT MIN(retail\_price)

FROM title);

TITLE

---------------------------------------------

A City Under the Sea

This example is an example of a subquery inside a WHERE clause. This is termed a nested subquery or inner query. The term outer query is sometimes used to refer to the select statement that contains a subquery.

Generally speaking, the SELECT clause of a subquery must contain only one expression, only one group function, or only one column name. The column returned by the subquery must match the column used in the WHERE clause.

**Subqueries and the IN Operator**

In earlier tutorials you used the IN operator that defined row selection criteria based on the use of lists of data enclosed in parentheses. The only difference in the use of the IN operator with subqueries is that the list does not consist of hard-coded values.

Suppose that we want to display the order\_id, and order\_date for orders that contain the following isbn: 0688010377. For example:

SELECT order\_id, order\_date

FROM orders

WHERE order\_id IN (SELECT order\_id

FROM order\_detail

WHERE isbn = ‘0688010377’);

ORDER\_ID ORDER\_DAT

--------- ---------

7 02-MAR-04

12 20-MAY-04

**Subqueries and Comparison Operators**

The most important point to remember when using a subquery with a comparison operator is that the subquery can only return a single value. A comparison operator can only have a single value on

it’s right hand side. So, if the comparison operator introduces a subquery, the subquery can only return one value.

**Group Functions and Comparison**

Group functions (AVG, SUM, MAX, MIN, and COUNT) always return a single value. Thus, a subquery with a group function as the object of a comparison operator will always execute provided you have written the subquery properly.

SELECT title

FROM title

WHERE retail\_price > (SELECT AVG(retail\_price) FROM title);

TITLE

---------------------------------------------

The C Programming Language

The Sword of Shannara Trilogy

The Legend of the Seventh Virgin

A Dilly of a Death

The Jasmine Moon Murder

A Walk in Wolf Wood

Programming with Visual Basic .Net

A Village Christmas

Dragonfly

In The Stone House

The Cleansing

Shadows over Innsmouth

Whispers in the Night

13 rows selected.

**Ranges (BETWEEN and NOT BETWEEN)**

The BETWEEN operator determines if a value is in a specified range. You can use the BETWEEN operator to specify an inclusive range of values.

SELECT bldg\_code, room, capacity

FROM location

WHERE capacity BETWEEN 10 AND 50;

BLDG\_CODE ROOM CAPACITY

---------- ------ ----------

CR 202 40

CR 103 35

CR 105 35

BUS 105 42

BUS 404 35

BUS 421 35

6 rows selected.

NOT BETWEEN simply excludes values in the range.

SELECT bldg\_code, room, capacity

FROM location

WHERE capacity NOT BETWEEN 10 AND 50;

BLDG\_CODE ROOM CAPACITY

---------- ------ ----------

CR 101 150

BUS 211 55

BUS 424 1

BUS 402 1

BUS 433 1

LIB 217 2

LIB 222 1

7 rows selected.

**Project Three: Advanced Queries - 15 Points**

You will need to write SELECT commands to answer the following questions. Print the SELECT command and your results using PHPMyAdmin. You will use the same tables that you used in Project 2.

1. List the inventory items that are the same color as inventory item 23. Display the Inventory\_ID and color (You must use a subquery in this query. You cannot hard-code the color.)
2. List the inventory items that have an average price greater than the average price of all the inventory items.

Display the Inventory\_ID and the Price (You must use a subquery in this query)

1. Select the following for each inventory item: Inventory\_ID, Item Description, item\_size, color, and Price
2. Select all orders and the names of the customers who placed the orders. Display Order\_ID, Order\_Date and the Customer Name
3. Display orders that contain the item Men’s Expedition Parka. Display Order\_ID, Order Date
4. List the items ordered for Order Id 6. Display Inventory\_ID, Extended Price and Qty
5. Display the in-stock quantity for each item. Display Item\_ID and Number of items (Order the list by the Item Id in ascending order. This query requires a GROUP BY clause)
6. List the quantity of each inventory item sold. Display Inventory ID and Number Sold. (Order the list by the number of items sold in descending order. This query requires a GROUP BY clause and the SUM function.)
7. Determine the order total (dollar amount) for each order and the customer who placed the order. Display the Order\_ID, Customer Name and the Order Total.
8. Display all of the inventory information for inv\_ids that do not have shipping records.
9. Display all of the inventory information and backorder information for inv\_ids that are on backorder
10. Display the customer first and last name and order total for order number 5
11. Display the inv\_id, description, price and color of the least expensive inventory item that we have in the inventory table. Use the join keyword to join the inventory table and item table. You need a subquery to determine the least expensive price then use an outer query to find all inventory items at that price.
12. Display the first name, last name, email address and order total for customers that have placed an order at Alpine Adventures.
13. Modify the above query to display ALL customers, whether they have placed an order or not.

Lesson #12

|  |  |  |
| --- | --- | --- |
| Topic | Review Terms, Quiz 1 and 2, Test 1 | Points |
| Lessons | 1 - 11 | 25 |

**Review Terms: 5 points**

The following 65 terms with definitions/explanations will be turned in word format.

What is a Database?

Why is MySQL so Popular?

Elements of MySQL and Its Environment

The Relational Database Model

MySQL Monitor

PHPMyAdmin

Entities

Attributes

Relationships

Constraints

Tables

SQL

Primary Key

Foreign Key

Referential Integrity

The Database Design Process

Data Abstraction

The Entity Relationship Model

Chen Model

Crow’s Foot Model

Entities

Mapping Entities and relationships to Database tables

The Normalization Process

Normal forms

The purpose of Normalization

Creating tables

Data types

Keys and indexes

The Auto\_Increment feature

Altering the table structure

Dropping tables and databases

Keys

Default values

Entity Integrity

Table types

Storage engine configuration

The SELECT Statement

Basic Querying Techniques

The WHERE clause

The ORDER BY clause

The Limit clause

The INSERT Statement

The DELETE Statement

The UPDATE Statement

Using LIKE

Regular Expressions

Null values

The DESCRIBE and SHOW commands

Using expressions in the SELECT clause

Joining tables

Aliases

Data Grouping

The DISTINCT clause

GROUP BY and HAVING

The INNER JOIN clause

The UNION Statement

Left and Right Joins

The Natural Join

The IN operator

Nested Queries (Subqueries)

The BETWEEN operator

ANY and ALL

The EXISTS operator

**HOMEWORK**

**Review Terms 2: 5 points**

The 29 terms listed in lesson 23 with definitions/explanations will be turned in word format when the student reaches lesson 23. It is recommended that the student fills out these definitions/ explanations as they occur.

Lesson #13

|  |  |  |
| --- | --- | --- |
| Topic | Views | Points |

**Main Topics:**

* Views

A view is a virtual table. That is, it acts like a table but actually contains no data. Instead, it is defined in terms of tables and provides alternative ways to look at table data.

A simple view can be nothing more than a way to select a subset of a table’s columns. Suppose you often want to select only the last\_name, first\_name, city, and state from the customer table, but you don’t want to write out all the columns like this:

SELECT last\_name, first\_name, city, state FROM customer;

Nor do you want to use SELECT \*. That is easier to write, but \* retrieves columns that you don’t want. The solution is to define a view that retrieves only the desired columns:

CREATE VIEW vcust AS

SELECT last\_name, first\_name, city, state FROM customer;

Now the view acts as a window into just those columns that you want to see. This means that you can use SELECT \* FROM vcust and get back only the columns named in the view definition.

A view can include a WHERE clause and can also refer to multiple tables.

Let’s say that in the example above we want to rename the column names that the view will create. The following example will do just that.

CREATE VIEW vcust (Last, First, City, State)

AS

SELECT last\_name, first\_name, city, state FROM customer;

Since the syntax uses the position of the new column name to determine what column the new name goes with, if you want to change even just one column name, you must provide a name for all the columns even if the name is same.

Lesson #14

|  |  |  |
| --- | --- | --- |
| Topic | Stored Procedures and Triggers | Points |
| Chapter | Lecture |  |
| Activity | Practical Application: adding triggers and stored procedures | 3 |

#### Procedure Tutorial

Microsoft SQL Server allows you to create stored procedures to simplify the database development process by grouping Transact-SQL statements into blocks of code that are stored on the server. Data can be passed into the procedure as parameters and values can be passed back to where the procedure was called.

#### Stored Procedure Advantages

You can do most of the things a stored procedure can do with simple ad hoc Transact-SQL queries. However, stored procedures have a number of advantages over ad hoc queries:

* They are parsed and stored on the server and can be run more than once
* They run faster and more efficiently than straight queries
* They encapsulate business rules and policies
* They incorporate modularization into your programs
* They can be shared between applications
* Access to the database is more secure and uniform

The following code will create a procedure that will display student information for the sid passed to the procedure when it is called. It is a common practice to start a procedure name with sp\_ so you know it is a procedure. Type in the following in SQL Server and execute it:

**CREATE PROCEDURE sp\_GetStudentInfo**

**@sid int**

**AS**

**SELECT \***

**FROM student**

**WHERE sid = @sid**

**GO**

To execute the procedure type the following code:

##### Exec sp\_GetStudentInfo 103

This should display the information for student 103. Amanda Mobley is student ID 103 and her name should be displayed. Type the following:

##### Exec sp\_GetStudentInfo 104

This should display the information for student 104. You can execute a procedure as many times as needed passing different parameters each time it is executed.

sp\_helptext lists the source for a stored procedure. Type the following:

**EXEC sp\_helptext ‘sp\_GetStudentInfo’**

You should see the text for the procedure that is stored on the server.

To drop a procedure type the following:

**drop procedure sp\_GetStudentInfo**

The following procedure will be stored on the server with the name ‘CountStudentsByZip’ and can be executed when needed. The OBJECT\_ID is a unique number in the server associated with each different procedure object. If the OBJECT\_ID(‘CountStudentsByZip’) value is NOT NULL, then that means the procedure already exists on the system. You cannot create the same procedure twice so you must drop the procedure before you try to create it again. This is done as a precaution so you do not accidentally try to create the same procedure twice. If you do, you will get an error stating the procedure already exists on the server. The ‘CountStudentsByZip’ is a procedure that DOES NOT receive any parameters. Type the following:

**IF OBJECT\_ID('CountStudentsByZip') IS NOT NULL**

**DROP PROC CountStudentsByZip**

**CREATE PROCEDURE CountStudentsByZip**

**AS**

**SELECT szip, COUNT(\*) AS NumberOfStudents**

**FROM Student**

###### GROUP BY sZip

**EXEC CountStudentsByZip**

The following code will create a procedure that updates a value in a table. The procedure is passed a cust\_id, a last name and a first name. The procedure will update the last name and first name in the customer table for the cust\_id passed into the procedure. Type the following code:

**IF OBJECT\_ID(sp\_UpdateCustName) IS NOT NULL**

**DROP PROC sp\_UpdateCustName**

**CREATE PROCEDURE sp\_UpdateCustName**

**(@cust\_id as int,**

**@lastName varchar(15),**

**@firstName varchar(15))**

**AS**

**UPDATE customer**

**SET last = @lastName,**

**first = @firstName**

**WHERE cust\_id = @cust\_id**

Check what value is in the customer table before you run the procedure.

**Select \* from customer where cust\_id = 2**

Run the procedure

###### EXEC sp\_UpdateCustName 2, 'Jones', 'Sally'

Check the value in the customer table after running the procedure.

The following is an example of a procedure that returns a value. You must declare a variable with the same data type as the OUTPUT variable from the procedure before you call the procedure. That is so you have a place in memory to hold the OUTPUT variable being returned. This procedure is passed a csecid and it determines the number of open seats for the given course section.

###### IF OBJECT\_ID('sp\_DetermineOpenSeats') IS NOT NULL

**DROP PROC sp\_DetermineOpenSeats**

**GO**

**CREATE PROCEDURE sp\_DetermineOpenSeats**

**(@csecid as int,**

**@OpenSeats as int OUTPUT )**

**AS**

**SELECT maxenrl - currenrl**

**FROM course\_section**

**WHERE csecid = @csecid**

To run this procedure you need to declare a variable to hold the open seats returned from the procedure. Try typing the following:

**DECLARE @OpenSeats int**

EXEC sp\_DetermineOpenSeats 1000,@OpenSeats = @OpenSeats OUTPUT

Type in the following procedure:

**CREATE Procedure sp\_DisplayInventoryDetails**

**(@CategoryName varchar(50) = NULL)**

**AS**

**-- Append a % so our users don't have to know**

**--exact subcategory names. Use the LIKE operator for the SELECT**

**if @CategoryName is null**

**select @CategoryName = '%'**

**else**

**select @CategoryName = '%' + @CategoryName + '%'**

###### SELECT inv\_id, description, category, price

**FROM inventory**

**INNER JOIN item**

**ON inventory.item\_id = item.item\_id**

**WHERE category like @CategoryName**

**ORDER BY category**

Run the procedure with different parameter values

**exec sp\_DisplayInventoryDetails**

**exec sp\_DisplayInventoryDetails 'Outdoor Gear'**

**exec sp\_DisplayInventoryDetails ‘Cloth’**

Database Triggers Tutorial

What is a database trigger?

* **Code that executes automatically in response to a specific event on a table**
* **Triggers are similar to procedures in that they contain procedural code that executes. Triggers are sometimes referred to as “special procedures”.**
* **However, the difference is a trigger executes automatically based on an event, you do not use the EXEC statement to execute them.**
* **The three events that can be set to “fire” a trigger are INSERT, UPDATE, and DELETE**
* **When you create a trigger it is associated with a single table.**
* **A table can have multiple triggers associated with it; each trigger will “fire” as required**
* **You may not know that a trigger has executed because they do not necessarily display anything. Triggers often operate “behind the scene” helping to make sure data in the database is valid.**
* **Common reasons to use a trigger:**
  + **to log changes**
  + **to notify of updates and changes**
  + **to enforce business rules**

**If necessary, you can handle more than one type of query in one trigger. For example, to handle both "INSERT" and "UPDATE", you would use "FOR INSERT, UPDATE". Any code after the "AS" keyword is actually executed when the trigger is called. This part of the trigger can contain any code that a standard stored procedure could contain. You can also call stored procedures using the "EXEC" command from within the body of the trigger.**

**When SQL server processes an "INSERT" command, it creates a new virtual (or temporary) table, which contains all of the fields that are being inserted into the table. This table is named "**inserted**", and is passed to the trigger. You can SELECT values from the inserted table to set parameter values.**

**If you create a trigger that is activated on a DELETE command, a virtual table named “**deleted**” contains all of the fields and values from the deleted record(s).**

**If you create a trigger that is activated for an UPDATE, then both the "inserted" and "deleted" virtual tables would be created and available from within the trigger. The "deleted" table would contain all of the fields and values for the row(s) before they were updated, and the "inserted" table would contain the new row(s) with the updated fields and values.**

**Let’s look at an example:**

**Create a trigger that will execute after an insert is done on the customer table to display the new customer name.**

Type the following in SQL Server:  
  
CREATE TRIGGER tr\_addCustomer

**ON customer**

**FOR INSERT**

**AS**

**-- Declare a variable to hold the name being inserted**

**DECLARE @newName VARCHAR(30)**

**--assign first and last name into @newName from inserted table**

**SET @newName = (SELECT first + ' ' + last**

**FROM inserted)**

**-- Print the name of the new customer.**

**PRINT 'New customer " ' + @newName + ' " added.'**

**GO**

This code will create a trigger that will “fire” whenever an INSERT statement is executed on the customer table. To test the trigger you must perform an INSERT statement. Type the following in SQL Server:

**INSERT INTO customer**

**VALUES( 'Rogers', 'Rebecca', 'E', '1156 Water Street Apt. 3', 'Osseo', 'WI', '54705', '7155558943', 'cjones@hotmail.com');**

You should see: New customer "Rebecca Rogers" added..

**Here is another example of using a trigger. When a shipment is received at Alpine Adventures we need to make sure that the quantity\_on\_hand value in the inventory table is updated to reflect the shipment quantity\_received amount for a specific inventory inv\_id value. We could create a trigger to automatically do this whenever the shipping table is updated with a received shipment. Type the following code to create a trigger on the shipping table for an UPDATE statement.**

create trigger tr\_shipUpdate

on shipping

for update

as

-- declare variables to hold inserted data

declare @quantity\_received int

declare @inv\_id int

-- set variables to inserted values

set @quantity\_received = (select quantity\_received

from inserted)

set @inv\_id = (select inv\_id from inserted)

update inventory set quantity\_on\_hand =

quantity\_on\_hand + @quantity\_received

where inv\_id = @inv\_id

GO

To test the trigger you must perform an UPDATE on the shipping table. But first, you would want to see the values in the database before the trigger then look at the values after the trigger.

Type both of the following statments in SQL Server and run them together:

Select \* from shipping

Select \* from inventory

Look at the shipping table for ship\_id 6 and note that the date\_received and qty\_received values are NULL. This shipment is for inv\_id 17. Look in the inventory table for inv\_id 17 and note the quantity\_on\_hand value.

Type the following to “fire” the update trigger.

**update shipping**

**set date\_received = ‘2009-10-10’,**

**quantity\_received = 50 where**

**ship\_id = 6**

**Now look at the two tables again to see the new values. The inventory should be updated with 50 additional quantity\_on\_hand for inv\_id 17.**

Select \* from shipping

**Select \* from inventory**

**Let’s look at an example of using a trigger to enforce a business rule to ensure the database has valid data. At Northwoods University we have a location table that includes information about room locations and the capacity of the room. We would want to be sure we do not allow a course\_section row to be inserted or updated with a maximum enrollment value (maxenrl) that is greater than the capacity of the location the class is being taught. We could create a trigger associated with an INSERT or UPDATE to the course\_section table that looks at capacity in the location table for the locid being inserted to make sure the room can accommodate the maxenrl value. Type the following:**

create trigger tr\_course\_section

on course\_section

for insert, update

as

declare @capacity int

declare @locid int

declare @maxenrl int

set @locid = (select locid from inserted)

set @capacity = (select capacity from location where locid = @locid)

set @maxenrl = (select maxenrl from inserted)

if (@maxenrl > @capacity)

BEGIN

PRINT 'cannot insert course\_section with maxenrl > capacity'

PRINT 'transaction has been cancelled'

ROLLBACK

END

GO

select \* from course\_section

insert into course\_section values

**(1013, 3, 6, 1, 1, 'MWF', '10:00AM', 45, 160, 100)**

**update course\_section set maxenrl = 160 where csecid = 1000**

Create a trigger that will automatically create a shipping record if an inventory item is update with a quantity\_on\_hand < 5. You can hard code the date\_expected or research on the Internet how to get the current date from the system and add 7 days to it for the date expected.

create trigger tr\_updateInventory

on inventory

for update

as

-- declare variables to hold inserted data

declare @quantity\_on\_hand int

declare @inv\_id int

declare @currentDate datetime

-- set variables to inserted values

set @quantity\_on\_hand = (select quantity\_on\_hand

from inserted)

set @inv\_id = (select inv\_id from inserted)

set @currentDate = dateadd (day, +10, GETDATE())

if (@quantity\_on\_hand < 5)

insert into shipping

values

(12, @inv\_id, @currentDate, 10, null, null)

print 'Shipping Record Generated'

update inventory set quantity\_on\_hand = 4 where inv\_id = 2

Create a trigger that will print the inv\_id and an informative message for any item that is added to the database that costs more than 500 dollars.

CREATE TRIGGER expensiveInventoryMsg

ON inventory

FOR INSERT

AS

DECLARE @price money

DECLARE @inv\_id int

SET @price = (SELECT price FROM inserted)

SET @inv\_id = (SELECT inv\_id FROM inserted)

IF @price >= 500

BEGIN

DECLARE @msg varchar(50)

SET @msg = 'Expensive item "' + @inv\_id + '" entered into inventory at $' + CAST(@price as varchar(10)) + '.'

PRINT @msg

/\*EXEC msdb.dbo.sp\_send\_dbmail

@profile\_name = 'ManagerProfile',

@recipients='manager@AlpineAdventures.com',

@body= @msg,

@subject = 'SQL Server Trigger Mail' \*/

END

GO

INSERT INTO inventory

VALUES( 5, null, 'Sienna', 274.99, 14);

**Practical application**: define a trigger for your database that logs updates to existing records. Also define stored procedures, one to select a set of data, one to delete, one to insert and one to update a record.

**Project Four:** Procedures and Triggers

In **this** project you will develop five procedures for Alpine Adventures that will be stored on the server. A Procedure is a block code that accomplishes a task. You will follow the steps that were outlined in the Procedure Tutorial. Be sure to test if the procedure exists so you can drop it if necessary and test that each procedure works correctly.

**Procedure 1:**

Alpine Adventures needs to be able to update the inventory table with a new quantity when an inventory shipment has arrived. Write a procedure that will update the *quantity\_on\_hand* column in the inventory table. It will accept two arguments (inv\_id, qty). It does not return any value. The qty passed into the procedure will be added to the quantity\_on\_hand in the table. You will need to test the procedure to verify it is working correctly. To test your procedure, check the quantity\_of\_hand for inv\_id 5. Then run your procedure using inv\_id 5 and a qty of 25. Check the quantity\_on\_hand after running your procedure and it should be 25 more. (Name the procedure sp\_UpdateInventory)

**Procedure 2:**

When a customer is placing a new order, an order record must be inserted into the order table. Write a procedure that will insert a row into the Orders table. It will accept arguments for each of the columns (except the order\_id because that is generated by SQL Server). Name the procedure sp\_InsertOrder

**Procedure 3:**

Alpine Adventures wants to be able to easily change a color in the inventory table. For example, instead of the color ‘Coral’ they want to use the color ‘Pink’. Write a procedure that will allow a specified color in the inventory table to be changed to a different color (Hint: you will use the UPDATE command). This procedure will be passed two values: the old color and the new color. Test the procedure by changing the color ‘Coral’ to the color ‘Pink’. Name the procedure sp\_UpdateColor

**Procedure 4**

Write a procedure that will allow a user to cancel an order. The procedure will be passed an order\_id and the necessary information will be deleted to cancel the order. Name the procedure sp\_CancelOrder.

**Procedure 5:**

Write a procedure that will calculate the total for a specified order\_id. The procedure will receive one input parameter (order\_id) and return one parameter (order\_total). Name the procedure sp\_CalcOrderTotal.

**Trigger 1:**

Create a trigger that will automatically create a shipping record if an inventory item is update with a quantity\_on\_hand < 5. You can hard code the date\_expected or research on the Internet how to get the current date from the system and add 7 days to it for the date expected. Name the trigger: tr\_updateInventory.

**Trigger 2:**

Create a trigger that will print the inv\_id and a message for any item that is added to the database that costs more than 500 dollars. Name the trigger: tr\_expensiveInventory.

Print the following:

Expensive item "23" entered into inventory table at $550.

Lesson #15

|  |  |  |
| --- | --- | --- |
| Topic | Introduction to PHP | Points |
| Chapter | Lecture |  |

**Main Topics:**

* What does PHP Do?
* Language Basics
  + Statements
  + Comments
  + Identifiers
  + Data Types
  + Functions
  + Arrays
  + Variables
  + Expressions and Operators
  + Flow-Control Statements
  + Embedding PHP in Web Pages
  + Strings
  + Include files

**What does PHP do?**

PHP is a simple yet powerful language designed for creating HTML content. PHP pages are HTML pages with PHP commands embedded in them. This is in contrast to many other dynamic web-pages solutions, which are scripts that generate HTML. The web server processes the PHP commands and sends their output (and any HTML from the file) to the browser.

A statement is a collection of PHP code that does something. The following is a sample of PHP statements:

echo "Hello, world";

myfunc(42, "Apress");

$number = 1;

$name = "John";

$c = $a / 10;

if ($a == $b)

{

echo "This test worked";

}

PHP uses semicolons to separate simple statements. A compound statement that uses curly braces to mark a block of code, such as a conditional test or loop, does not need a semicolon after a closing brace.

In general, whitespace doesn’t matter in a PHP program.

Several different types of comments are allowed in PHP.

* // comment text (one line comments)
* /\* multi

line comment \*/

An identifier is simply a name. In PHP, identifiers are used to name variables, functions, constants, and classes. The first character must be either a letter or an underscore.

Variable names always begin with a dollar sign($) and are case-sensitive.

Function names are not case-sensitive and do not begin with a dollar sign.

PHP provides eight types of values, or data types. Four are scalar (single-value) types:

* integers
* floating-point numbers
* strings
* Booleans

Two are compound types:

* arrays
* objects

The last two are special types:

* resource
* NULL

**Variables** in PHP are identifiers preceded by a dollar sign.

A variable may hold a value of any type. PHP is a weakly typed language, in other words, to create a variable just assign a value to it.

**Operators** in PHP are almost identical to Java operators, except for the concatenation operator, which is a period ( . ).

The basic assignment operator is the = sign.

**Functions**

A function is a named block of code that performs a specific task, possibly acting upon a set of values given to it, or parameters, and possibly returning a single value. Functions improve reliability by allowing you to fix any bugs in one place, rather than everywhere you perform a task.

Here is an example function:

// strlen( ) is a built-in function that returns the length of a string

$length = strlen("PHP"); // $length is 3

The following example shows how to define a function:

function strcat($left, $right)

{

$combined\_string = $left . $right;

return $combined\_string;

}

The function takes two arguments, $left and $right. Using the concatenation operator, the function creates a combined string in the variable $combined\_string. Finally, $combined\_string is returned.

There are two different ways to pass parameters to a function. The first, and more common, is by value. The other is by reference.

In most cases, you pass parameters by value. The argument is any valid expression. That expression is evaluated, and the resulting value is assigned to the appropriate variable in the function.

Passing by reference gives a function direct access to a variable. To be passed by reference, the argument must be a variable. You indicate that a particular argument of a function will be passed by reference by preceding the variable name in the parameter list with an ampersand (&).

function square(&$value)

{

$value = $value \* $value;

}

$a = 4;

square($a);

echo $a;

Because the function’s $value is passed by reference, the actual value of $a, rather than a copy of that value, is modified by the function. In this case the new value of $a is 16.

**Arrays**

There are two kinds of arrays in PHP: indexed and associative. The keys of an indexed array are integers, beginning at 0. Indexed arrays are used when you identify things by their position. Associative arrays have strings as keys. PHP internally stores all arrays as associative arrays, so the only difference is what the keys happen to be.

You can access specific values from an array using the array variable’s name, followed by the element’s key (sometimes called the index) within square brackets:

$age[‘Fred’]

$shows[2]

The key can be either a string or an integer.

Using simple assignment to initialize an array looks like this:

$names[ 0 ] = ‘Fred’;

$names[ 1 ] = ‘Joe’;

$names[ 2 ] = ‘Tim’;

That’s an indexed array, with integer indexes beginning at 0. The following is an associative array:

$price[‘Gasket’] = 15.29;

$price[‘Wheel’] = 75.25;

$price[‘Tire’] = 50.00;

An easier way to initialize an array is to use the array( ) construct, which builds an array from its arguments. This builds an indexed array, an the index values (starting at 0) are created automatically:

$names = array(‘Fred’, ‘Joe’, ‘Tim’);

To create an associative array with array( ), use the => symbol to separate indexes from values:

$price = array(‘Gasket’ => 15.29,

‘Wheel’ => 75.25,

‘Tire’ => 50.00);

**Flow-Control Statements**

if

if ($user\_validated)

{

echo ‘Welcome’;

}

else

{

echo ‘Access Forbidden’;

exit;

}

switch

switch($name)

{

case ‘Joe’:

// do something

break;

case ‘Tom’:

// do something

break;

default:

// do something else

break;

}

while loop

$total = 0;

$i = 1;

while ($I <= 10)

{

$total += $I;

$I++;

}

do/while loop - this loop will be executed at least once

$total = 0;

$I = 1;

do

{

$total += $++;

} while ($I <= 10);

for loop

$counter = 0;

for ($counter = 0; $counter < 10; $counter++)

{

echo "Counter is $counter\n’;

}

foreach

The foreach statement allows you to iterate over elements in an array.

$names = array(‘Joe’, ‘Tom’);

foreach ($names as $value)

{

echo "Processing $value\n";

}

Processing Joe

Processing Tom

PHP executes the body of the loop (the echo statement) once for each element of $names in turn, with $value set to the current element. Elements are processed by their internal order. The following example gives you access to the current key:

$person = array(‘name’ => ‘Fred’, ‘age’ => 35, ‘wife => ‘Wilma’);

foreach ($person as $key => $value)

{

echo "Fred’s $key is $value\n";

}

Fred’s name is Fred

Fred’s age is 35

Fred’s wife is Wilma

In this case, the key for each element is placed in $key and the corresponding value is placed in $value.

Since PHP is designed to be embedded in HTML files, we need a way to identify the regions of PHP code to be executed. The currently preferred method is as follows:

<?php

echo "Hi";

?>

**Strings**

Most data you encounter as your program runs will be sequences of characters, or strings. Strings hold people’s names, passwords, addresses, credit card numbers, purchase histories, and more. PHP has an extensive selection of functions for working with strings.

This lesson shows the many ways to write strings in your programs, including the sometimes-tricky subject of interpolation (placing a variable’s value into a string), then covers functions for changing, quoting, and searching strings.

There are three ways to write a literal string in your program: using single quotes, double quotes, and the ‘here document’ format. These methods differ in whether they recognize special escape sequences that let you encode other characters or interpolate variables.

The general rule is to use the least powerful quoting mechanism necessary. This means that you should use single-quoted strings unless you need to include escape sequences or interpolate variables, in which case you should use double-quoted strings. If you want a string that spans many lines, use a

‘heredoc’.

When you define a string literal using double quotes or a ‘heredoc’, the string is subject to variable interpolation. Interpolation is the process of replacing variable names in the string with the values of those variables. There are two ways to interpolate variables into string—the simple way and the complex way.

The simple way is to put the variable in double-quoted string or a ‘heredoc’

$who = ‘Kilroy’;

$where = ‘here’;

echo "$who was $where";

Kilroy was here

The complex way is to surround the variable with curly braces. This method can be used for array lookups. The classic use of curly braces is to separate the variable name from surrounding text:

$n = 12;

echo "You are the {$n}th person";

You are the 12th person

Without the curly braces, PHP would try to print the value of the $nth variable.

Single-quoted strings do not interpolate variables. The variable name in the following string is not expanded because the string literal in which it occurs is single-quoted:

$name = ‘Fred’;

$str = ‘Hello, $name’;

echo $str;

Hello, $name

The only escape sequences that work in single-quoted strings are \’, which puts a single quote in the string, and \\, which puts a backslash in a single-quoted string. Any other occurrence of a backslash is interpreted as a backslash:

$name = ‘Tim O\’Reilly’;

echo $name;

$path = ‘C:\\WINDOWS’;

echo $path;

$nope = ‘\n’;

echo nope;

Tim O’Reilly

C:\WINDOWS

\n

Double-quoted strings interpolate variables and expand the many PHP escape sequences, which are listed in the following table:

|  |  |
| --- | --- |
| Escape sequence | Character represented |
| \" | Double quote |
| \n | Newline |
| \r | Carriage return |
| \t | Tab |
| \\ | Backslash |
| \$ | Dollar sign |
| \{ | Left brace |
| \} | Right brace |
| \[ | Left bracket |
| \] | Right bracket |

You can easily put multiline strings into a program with a ‘heredoc, as shown below:

$test = <<< End\_of\_Quote

Line One.

Line Two.

Line Three.

End\_of\_Quote;

echo $test;

Line One.

Line Two.

Line Three.

The <<< Identifier tells PHP that you are using a ‘heredoc’. There must be a space after the <<< and before the identifier. You get to choose the identifier. The next line starts the text being quoted by the ‘herdoc’, which continues until it reaches a line that consists of nothing but the identifier.

* Single and double quotes in a ‘herdoc’ are passed through.
* Whitespace is also preserved.
* The Newline before the trailing terminator is removed.
* The final identifier must be on a line by itself and it must start in Column 1.

**echo**

To put a string into the HTML of a PHP-generated page, use echo. While it looks and behaves like a function, echo is a language contruct. It does not require parentheses.

echo "PrintMe";

**Accessing Individual Characters**

The strlen( ) function returns the number of characters in a string:

$string = ‘Hello, world’;

$length = strlen($string);

You can use the string offset syntax on a string to address individual characters:

$string = ‘Hello’;

for ($i=0; $i < strlen($string); $i++)

{

printf("The %dth character is %s\n", $i, $string{$i});

}

The 0th character is H

The 1th character is e

The 2th character is l

The 3th character is l

The 4th character is o

If you know where the data that you are interested in lies in a larger string, you can copy it out with the substr( ) function:

$piece = substr(string, start [, length]);

The start argument is the position in string at which to begin copying, with 0 meaning the start of the string. The length argument is the number of characters to copy (the default is to copy until the end of the string).

**Include files**

PHP provides two constructs to load code and HTML from another module: require and include. They both load a file as the PHP script runs, work in conditionals and loops, and complain if the file being loaded cannot be found. The main difference is that attempting to require a nonexistent file is a fatal error, while attempting to include such a file produces a warning but does not stop script execution.

A common use of include is to separate page-specific content from general site design. Common elements such as headers and footers go in separate HTML files, and each page then looks like:

<?php include ‘header.html’; ?>

content

<?php include ‘footer.html’; ?>

The require is more suited to loading code libraries, where the page cannot be displayed if the libraries do not load. For example:

require ‘codelib.inc’;

mysub( ); //defined in codelib.inc

A more efficient way to handle headers and footers is to load a single file and then call functions to generate the standardized elements:

<?php require ‘design.inc’;

header( );

?>

content

<?php footer( ); ?>

If a program uses include or require to include the same file twice, the file is loaded and the code is run or the HTML is printed twice. This can result in errors about the redefinition of functions or multiple copies of headers or HTML being sent. To prevent these errors from occurring, use the include\_once and require\_once construts. They behave the same as include and require the first time a file is loaded, but ignore subsequent attempts to load the same file.

Lesson #16

|  |  |  |
| --- | --- | --- |
| Topic | Database Access using PHP | Points |
| Chapter | Lecture |  |

**Main Topics:**

* Connecting to a MySQL server
* Selecting the correct database
* Submitting a query to a MySQL database
* Using and displaying the data

The following code demonstrates how to connect to a PHP database and display data from a MySQL table:

<html>

<body>

<h1>Display Items</h1>

<?php

// Create database connection

$conn = mysql\_connect("localhost", "UserName", "Password");

mysql\_select\_db("DatabaseName", $conn);

// Create a query

$sql = "SELECT Item\_Id, Description, Category FROM item";

// Submit query to database

$result = mysql\_query($sql, $conn);

// Start HTML table creation

echo "<table border='1'>\n";

// Get column names

echo "<tr>\n";

while ($field = mysql\_fetch\_field($result))

{

echo " <th>$field->name</th>\n";

}

echo "</tr>\n\n";

// Get row data as an associative array

while ($row = mysql\_fetch\_assoc($result))

{

echo "<tr>\n";

// Display each column

foreach ($row as $col=>$val)

{

echo " <td>$val</td>\n";

}

echo "</tr>\n\n";

}

// Close HTML table

echo "</table>\n";

?>

</body>

</html>

The first task is to get a connection between your PHP program and a MySQL server. The mysql\_connect( ) function arranges the communication link between MySQL and PHP. The following code from Example One creates the database connection and selects the default database to use:

// Create database connection

$conn = mysql\_connect("localhost", "UserName", "Password");

mysql\_select\_db("DatabaseName", $conn);

The mysql\_connect( ) function returns an integer referring to the database connection. In this case the data connection is stored in a variable called $conn so the connection can be accessed by other database functions.

A data connection can have a number of databases connected to it. The mysql\_select\_db( ) function lets you choose a database. This function requires the database name and a data connection. It returns FALSE if it is unable to connect to the specified database.

Creating a query is very easy. Look at the following code from Example One:

// Create a query

$sql = "SELECT Item\_Id, Description, Category FROM item";

// Submit query to database

$result = mysql\_query($sql, $conn);

Begin by placing SQL code in a variable.

The mysql\_query( ) function allows you to pass an SQL command through a connection to a database. You can send any SQL command to the database with mysql\_query( ), including table creation statements, inserts, updates, deletes, and queries. The database returns a special element called a result set. If the SQL command was a query, the result variable holds a pointer to the data, which is taken apart in the next step.

The data will be printed in an HTML table. You could create the table headings by hand, because you already know what all the columns are, but it is better to get the column information directly from the query. You won’t always know which columns are being returned by a particular query. The next code segment manages this task:

// Start HTML table creation

echo "<table border='1'>\n";

// Get column names

echo "<tr>\n";

while ($field = mysql\_fetch\_field($result))

{

echo " <th>$field->name</th>\n";

}

echo "</tr>\n\n";

The mysql\_fetch\_field( ) function expects a query result as its one parameter. It then fetches the next column and stores it in the $field variable. If no columns are left in the result, the functions returns FALSE. This allows this function to also be used as a conditional statement.

The $field variable is actually an object. The $field object is much like an associative array. It has a number of properties. The field object as the following properties:

max\_length - Column length

name - The column name

primary\_key - TRUE if the column is a primary key

table - Name of the table this column belongs to

type - This column’s data type

You use object-oriented syntax to refer to an object’s properties. The code echos $field->name to the HTML table. This syntax simply refers to the name property of the field object.

The rest of the code examines the result set.

// Get row data as an associative array

while ($row = mysql\_fetch\_assoc($result))

{

echo "<tr>\n";

// Display each column

foreach ($row as $col=>$val)

{

echo " <td>$val</td>\n";

}

echo "</tr>\n\n";

}

// Close HTML table

echo "</table>\n";

The mysql\_fetch\_assoc( ) function fetches the next row from a result set. It requires a result pointer as its parameter, and it returns an associative array.

If no rows are left in the result set, mysql\_fetch\_assoc( ) returns a FALSE. This function is often used as a condition in a while loop. Each row represents a row of the HTML table, so we print the HTML code to start a new row inside the while loop.

Once you have gotten a row, it’s stored as an associative array. You can manipulate this array using a foreach loop. We assign each element to $col and $val variables. We actually don’t need $col in this case, but it can be handy to have. Inside the foreach loop we placed code to print the current column in a table cell.

Lesson #17

|  |  |  |
| --- | --- | --- |
| Topic | Introduction to The Famous Quotes Content Management System | Points |
| Chapter | Lecture |  |
| Activity | Practical application: Get you data via the web | 3 |
| Project | Four: Famous Quotes CMS | 15 |

#### Project 4 - Famous Quotes CMS - 15 Points

To make the leap from a Web page that displays information stored in a database to a completely database-driven Website, we need to add a **Content Management System (CMS)**. Such a system usually takes the form of a series of Web pages, access to which is restricted to users who are authorized to make changes to the Website. These pages provide a database administration interface that allows a user to view and change the information that’s stored in the database without bothering with the mundane details of SQL syntax.

The following tables will need to be created in your MySQL database using the code below:

# Code to create a simple quote table that stores an author ID

CREATE TABLE fq\_quotes

(

quote\_id INT NOT NULL AUTO\_INCREMENT PRIMARY KEY,

quotetext TEXT,

quotedate DATE NOT NULL,

author\_id INT

)ENGINE = InnoDB;

# Code to create a simple author table

CREATE TABLE fq\_author

(

author\_id INT NOT NULL AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(255),

email VARCHAR(255)

)ENGINE = InnoDB;

# Code to create a lookup table for quote-category relationship

CREATE TABLE fq\_quotecategory

(

quote\_id INT NOT NULL,

category\_id INT NOT NULL,

PRIMARY KEY (quote\_id, category\_id)

)ENGINE = InnoDB;

# Code to create a simple category table

CREATE TABLE fq\_category

(

category\_id INT NOT NULL AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(255)

)ENGINE = InnoDB;

After you have created the tables in your MySQL database, you will begin to enter the code for each of the individual Web pages. The logic involved in each of these pages will be covered in detail by your instructor.

The front page will be constructed in this lesson. Lesson 17 will include the code for managing the authors. Lesson 18 will present the code to manage the categories and Lesson 19 will contain code to manage the quotes.

The front page of the Content Management System will contain links to pages that manage the three entities (quotes, authors, and categories). The following simple HTML code produces the index page shown in Figure 1-1:

*Important Note: Throughout the Famous Quotes project the code is presented in small fragments. These fragments are designed to make it easier to explain the code.*

*Whenever you need to start entering code the file name will be given and the code will be presented all at once.*

**\*File Name: index.html**

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

<title>Quote CMS</title>

<meta http-equiv="content-type" content="text/html; charset=iso-8859-1" />

</head>

<body>

<h1>Quote Management System</h1>

<ul>

<li><a href="quotes.php">Manage Quotes</a></li>

<li><a href="authors.php">Manage Authors</a></li>

<li><a href="categories.php">Manage Quote Categories</a></li>

</ul>

</body>

</html>

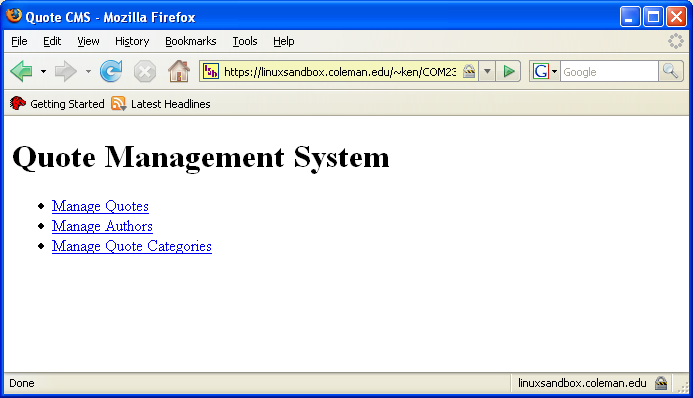


Figure 1-1

**Alternate Syntax**

This project uses an alternate for the *if* statement which may seem a little strange at first. Instead of enclosing the block of statements in curly braces, end the *if* and *else* lines with a colon (:) and use endif; to end the block. For example:

if ($user\_validated) :

echo “Welcome!”;

$greeted = 1;

else :

echo “Access Forbidden!”;

exit;

endif;

This syntax can be very useful if you have large blocks of HTML inside your statements. For example:

<?php if($user\_validated) : ?>

<table>

<tr>

<td>First Name:</td><td>Fred</td>

</tr>

<tr>

<td>Last Name:</td><td>Jones</td>

</tr>

</table>

<?php else : ?>

Please log in.

<?php endif; ?>

**Practical application**: define a trigger for your database that logs updates to existing records. Also define stored procedures, one to select a set of data, one to delete, one to insert and one to update a record.

Lesson #18

|  |  |  |
| --- | --- | --- |
| Topic | Famous Quotes CMS: Managing Authors | Points |
| Chapter | Lecture |  |

***Managing Authors***

Let’s begin with *authors.php*, the file that allows administrators to add new authors, delete and edit existing ones. The first thing we’ll present to an administrator who needs to manage authors is a list of all authors currently stored in the database. Code-wise, this is the same as listing the quotes in the database. As we’ll want to allow administrators to delete and edit existing authors, you should include links for these features next to each author’s name. These links will have the *author\_id* of the author attached to them, so that the target document knows which author’s details the user wishes to edit or delete. Finally, we’ll provide an Add new author link that leads to a form that will allow the user to create a record for a new author.

**\*File Name: authors.php**

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

<html>

<head>

<title>Quote CMS: Manage Authors</title>

<meta http-equiv="content-type"

content="text/html; charset=iso-8859-1" />

</head>

<body>

<h1>Manage Authors</h1>

<ul>

<?php

$dbcnx = @mysql\_connect('localhost', 'username', 'password');

if (!$dbcnx)

{

exit('<p>Unable to connect to the ' .

'database server at this time.</p>');

}

if (!@mysql\_select\_db('your\_databaseName'))

{

exit('<p>Unable to locate the ' .

'database at this time.</p>');

}

$authors = @mysql\_query('SELECT author\_id, name FROM fq\_author');

if (!$authors)

{

exit('<p>Error retrieving authors from database!<br />'.

'Error: ' . mysql\_error() . '</p>');

}

while ($author = mysql\_fetch\_array($authors))

{

$aid = $author['author\_id'];

$aname = htmlspecialchars($author['name']);

echo "<li>$aname ".

"<a href='editauthor.php?aid=$aid'>Edit</a> ".

"<a href='deleteauthor.php?aid=$aid'>Delete</a></li>";

}

?>

</ul>

<p><a href="newauthor.php">Add new author</a></p>

<p><a href="index.html">Return to front page</a></p>

</body>

</html>

The *htmlspecialchars* function used within the while loop in the code above may be a little worrisome to you. For the moment, you can simply ignore it. I’ll explain exactly what it does in the section called “Editing Authors”. The interface produced by this script is shown in Figure 2-1.



Figure 2-1

***Deleting Authors***

*deleteauthor.php* will allow us to remove an author from the database, given that author’s ID. This is easy to do, but there is added complexity here. Remember that our quote table has an author\_id column that indicates the author responsible for any given quote. When we remove an author from the database, we must also get rid of any references to that author in other tables. If we didn’t, our database might contain quotes associated with nonexistent authors.

We have three possible ways to handle this situation:

* Don’t allow users to delete authors that are associated with quotes in the database.
* When we delete an author, also delete any quotes attributed to the author.
* When we delete an author, set the *author\_id* of any quotes attributed to the author to NULL, to indicate that they have no author.

When we take measures like these to preserve the relationships in our database, we are said to be protecting the database’s **referential integrity**. MySQL, like most database servers, supports a feature called **foreign key constraints** that can do this automatically. By setting up these constraints, you can instruct MySQL to take any of the steps listed above, in order to keep your data properly related.

To take advantage of this feature, however, you must create your database using the more advanced **InnoDB table format**, rather than the simple **MyISAM table format** that MySQL creates by default. While more feature-rich, InnoDB tables work more slowly because of the added overhead of those features. In simple applications like this, the best result is usually achieved by letting the application code (in this case, the PHP script) take care of maintaining referential integrity. For more information on foreign key constraints, see the MySQL Reference Manual.

Since most authors would not like us to use their quotes without giving them credit, we’ll choose the second option above. This also saves us from having to handle quotes with NULL values in their *author\_id* column when we display our library of quotes.

Since we’ll be deleting quotes, there is yet another layer of complexity to consider. Quotes may be assigned to categories by means of entries in the *fq\_quotecategory* table. When we delete quotes, we must also make sure that such entries are removed from the database. In summary, our script will delete an author, any quotes belonging to that author, and any category assignments that pertain to those quotes.

**\*File Name deleteauthor.php**

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

<title>Quote CMS: Delete Author</title>

<meta http-equiv="content-type"

content="text/html; charset=iso-8859-1" />

</head>

<body>

<?php

$dbcnx = @mysql\_connect('localhost', 'username', 'password');

if (!$dbcnx)

{

exit('<p>Unable to connect to the ' .

'database server at this time.</p>');

}

if (!@mysql\_select\_db('your\_databaseName')) {

exit('<p>Unable to locate the ' .

'database at this time.</p>');

}

// Delete all quotes belonging to the author

// along with the entry for the author.

$aid = $\_GET['aid'];

$ok1 = @mysql\_query(

"DELETE fq\_quotes, fq\_quotecategory

FROM fq\_quotes, fq\_quotecategory

WHERE fq\_quotes.quote\_id=fq\_quotecategory.quote\_id AND author\_id='$aid'");

$ok2 = @mysql\_query("DELETE FROM fq\_quotes WHERE author\_id='$aid'");

$ok3 = @mysql\_query("DELETE FROM fq\_author WHERE author\_id='$aid'");

if ($ok1 and $ok2 and $ok3)

{

echo '<p>Author deleted successfully!</p>';

}

else

{

echo '<p>Error deleting author from database!<br />'.

'Error: ' . mysql\_error() . '</p>';

}

?>

<p><a href="authors.php">Return to authors list</a></p>

</body>

</html>

To delete everything it needs to delete, this script performs three separate SQL queries. The first of these uses a type of DELETE query that you have not yet seen: a multi-table DELETE:

$ok1 = @mysql\_query(

"DELETE fq\_quotes, fq\_quotecategory

FROM fq\_quotes, fq\_quotecategory

WHERE fq\_quotes.quote\_id=fq\_quotecategory.quote\_id AND author\_id='$aid'");

In brief, it works just like a SELECT join, giving a list of tables and matching up their entries in the WHERE clause. The first list of tables (before FROM) indicates which tables you actually want to delete entries from. The second list of tables (after FROM) lists the full set of tables involved in the query. In this case, the two lists happen to be the same. The query will delete any quotes that have been assigned to one or more categories, along with their category assignments.

Since this first query will miss any quotes that have no corresponding entries in the fq\_quotecategory table, the more familiar second query will clean up any lingering quotes that belong to the author in question:

$ok2 = @mysql\_query("DELETE FROM fq\_quotes WHERE author\_id ='$aid'");

The third query finally deletes the author:

$ok3 = @mysql\_query("DELETE FROM fq\_author WHERE author\_id ='$aid'");

Just like that, we can delete an author—and all his or her quotes—with a single

click, as shown in Figure 2-2.

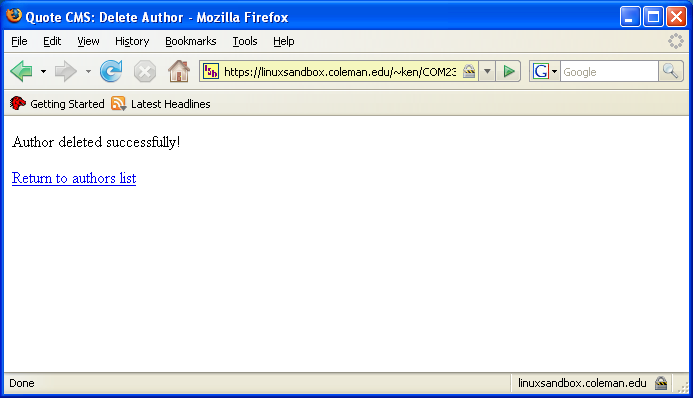


Figure 2-2

***Adding Authors***

Next comes newauthor.php, which allows administrators to add new authors to the database.

**\*File Name: newauthor.php**

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

<title>Quote CMS: Add New Author</title>

<meta http-equiv="content-type"

content="text/html; charset=iso-8859-1" />

</head>

<body>

<?php if (isset($\_POST['aname'])):

// A new author has been entered

// using the form below.

$dbcnx = @mysql\_connect('localhost', 'username', 'password');

if (!$dbcnx)

{

exit('<p>Unable to connect to the ' .

'database server at this time.</p>');

}

if (!@mysql\_select\_db('your\_databaseName'))

{

exit('<p>Unable to locate the ' .

'database at this time.</p>');

}

$aname = $\_POST['aname'];

$email = $\_POST['email'];

$sql = "INSERT INTO fq\_author SET

name='$aname',

email='$email'";

if (@mysql\_query($sql))

{

echo '<p>New author added</p>';

}

else

{

echo '<p>Error adding new author: ' .

mysql\_error() . '</p>';

}

?>

<p><a href="<?php echo $\_SERVER['PHP\_SELF']; ?>">Add another author</a></p>

<p><a href="authors.php">Return to authors list</a></p>

<?php else: // Allow the user to enter a new author ?>

<form action="<?php echo $\_SERVER['PHP\_SELF']; ?>" method="post">

<p>Enter the new author:</p>

<label>Name: <input type="text" name="aname" /></label><br />

<label>Email: <input type="text" name="email" /></label><br />

<input type="submit" value="SUBMIT" />

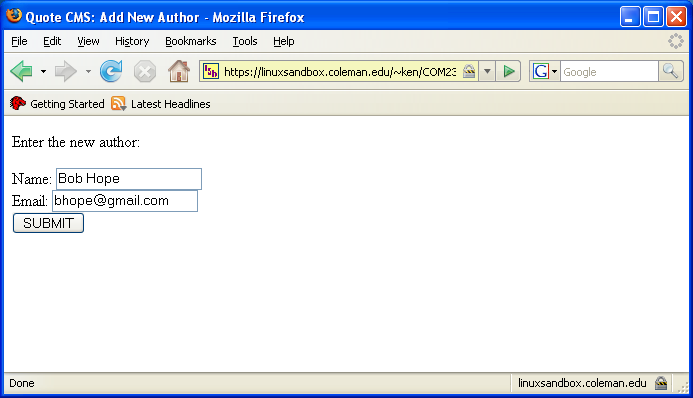
</form>

<?php endif; ?>

</body>

</html>

We now have a simple form that can be used to create new authors in the database, as shown in Figure 2-3.

Figure 2-3

***Editing Authors***

All that’s left is *editauthor.php*, which must provide an interface through which we can edit existing authors’ details. This page will actually be very similar to *newauthor.php*, except that the form fields will initially contain the values stored

in the database, and an UPDATE query will be used instead of an INSERT query when the form is submitted.

One minor complication comes into play here. To initialize the form fields with the values stored in the database, this page needs to retrieve these values and store them in PHP variables ($aname and $email). It will use the $aid variable passed from *authors.php* to do this. The code for our form should then look like this:

<form action="<?php echo $\_SERVER['PHP\_SELF']; ?>" method="post">

<p>Edit the author:</p>

<label>Name: <input type="text" name="aname"

value="<?php echo $aname; ?>" /></label><br />

<label>Email: <input type="text" name="email"

value="<?php echo $email; ?>" /></label><br />

<input type="hidden" name="aid" value="<?php echo $aid; ?>" />

<input type="submit" value="SUBMIT" />

</form>

As an aside, notice the hidden form field; we use this to pass the author’s ID along with the updated values when the form is submitted.

But consider what would happen if the author’s name were "The Quote Master" (*with* the quotes). The input tag produced by the PHP script would look like this:

<input type="text" name="name" value=""The Quote Master"" />

Obviously, this is invalid HTML. We need to replace the quotes in the name with their HTML **character entity** equivalents. Specifically, any double quotes in the name should be converted to the character code &quot; as follows:

<input type="text" name="name" value="&quot;The Quote Master&quot;" />

PHP provides a function called *htmlspecialchars* that converts special HTML characters such as <, > and quotes (among others) into their respective character codes automatically. Consider the following basic example:

$text = htmlspecialchars('<HTML> can be dangerous!');

echo $text; // output: &lt;HTML&gt; can be dangerous!

To avoid problems with quotes and angled brackets in your text strings, you should use this function whenever you output a non-HTML text string—especially when you output variables that, as they’re retrieved from a database, or are submitted by users, can have unpredictable values.

// Convert special characters for safe use

// as HTML attributes.

$aname = htmlspecialchars($aname);

$email = htmlspecialchars($email);

With this issue in mind, we can create an author editing page.

**\*File Name: editauthor.php**

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

<title>Quote CMS: Edit Author</title>

<meta http-equiv="content-type"

content="text/html; charset=iso-8859-1" />

</head>

<body>

<?php

$dbcnx = @mysql\_connect('localhost', 'username', 'password');

if (!$dbcnx)

{

exit('<p>Unable to connect to the ' .

'database server at this time.</p>');

}

if (!@mysql\_select\_db('your\_databaseName'))

{

exit('<p>Unable to locate the ' .

'database at this time.</p>');

}

if (isset($\_POST['aname'])):

// The author's details have been updated.

$aname = $\_POST['aname'];

$email = $\_POST['email'];

$aid = $\_POST['aid'];

$sql = "UPDATE fq\_author SET

name='$aname',

email='$email'

WHERE author\_id='$aid'";

if (@mysql\_query($sql)) {

echo '<p>Author details updated.</p>';

} else {

echo '<p>Error updating author details: ' .

mysql\_error() . '</p>';

}

?>

<p><a href="authors.php">Return to authors list</a></p>

<?php

else: // Allow the user to edit the author

$aid = $\_GET['aid'];

$author = @mysql\_query(

"SELECT name, email FROM fq\_author WHERE author\_id='$aid'");

if (!$author)

{

exit('<p>Error fetching author details: ' .

mysql\_error() . '</p>');

}

$author = mysql\_fetch\_array($author);

$aname = $author['name'];

$email = $author['email'];

// Convert special characters for safe use

// as HTML attributes.

$aname = htmlspecialchars($aname);

$email = htmlspecialchars($email);

?>

<form action="<?php echo $\_SERVER['PHP\_SELF']; ?>" method="post">

<p>Edit the author:</p>

<label>Name: <input type="text" name="aname" value="<?php echo $aname; ?>" /></label><br />

<label>Email: <input type="text" name="email" value="<?php echo $email; ?>" /></label><br />

<input type="hidden" name="aid" value="<?php echo $aid; ?>" />

<input type="submit" value="SUBMIT" />

</form>

<?php endif; ?>

</body>

</html>

Though there are a couple of complicated things going on under the hood, the user interface shown in Figure 2-4 is a model of simplicity.

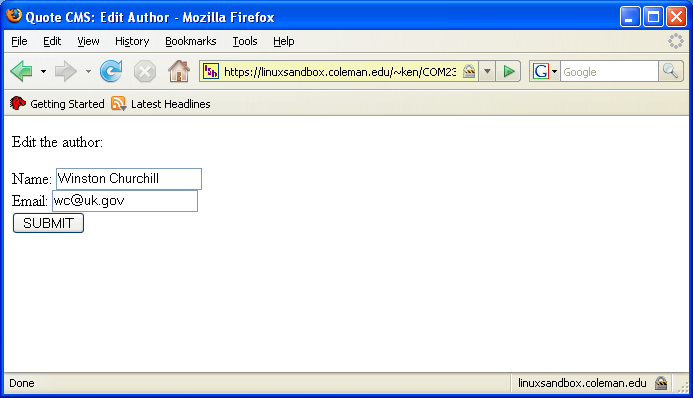


Figure 2-4

**\*\*\*\*Show your Instructor your project up to this point and demonstrate that you can manage the authors.\*\*\*\***

***Magic Quotes***

While we’re on the subject of troublesome special characters, there is another situation in which particular characters in a string can cause problems. Consider the following SQL query:

mysql>**INSERT INTO author SET**

->**name='Molly O'Reilly',**

->**email='molly@hotmail.com';**

Obviously, the apostrophe in the author’s last name will cause problems here, as MySQL can no longer figure out where the author’s name ends. The solution in this case would be to use another function provided by PHP: *addslashes*. This function, like *htmlspecialchars*, converts a string’s unsafe characters so that they’re safe. The difference is that addslashes is used to **escape** special characters by putting backslashes before them, as follows:

mysql>**INSERT INTO author SET**

->**name='Molly O\'Reilly',**

->**email='molly@hotmail.com';**

A backslash tells MySQL to treat the next character (the apostrophe, in this case) as a character in the string, ignoring any special meaning it might normally have. Thus, the above code will correctly insert the name “Molly O’Reilly” into the author table.

Why haven’t we worried about this problem before now? PHP has a nifty little feature called **magic quotes**, which is enabled by default with the following setting in your *php.ini* file:

magic\_quotes\_gpc = On

This setting tells PHP to use the addslashes function automatically on any values that are passed with the request for the page. The “gpc” in the name stands for “get, post, cookies,” which are the three methods by which information may be passed with a request for a Web page. As all the values we’ve inserted into our database until now have been passed as part of a form submission, the magic quotes feature of PHP has automatically added slashes to them every time. Values retrieved from a MySQL database, however, do not benefit from the magic quotes feature, so we must add slashes before we can use them in any situation where quotes, apostrophes, and other special characters might be a problem.

In some cases, you may not actually *want* to add backslashes to submitted values. For example, if you’re going to print out a value that was submitted with a form, and not insert it into a database, then those backslashes could turn out to be quite an eyesore. To undo the work of either the *addslashes* function or the magic quotes feature, you can use yet another function called *stripslashes*.

Complete information about these functions may be found in the PHP Manual. All the scripts in this book are written with the default setting, magic\_quotes\_gpc = On in mind.

Lesson #19

|  |  |  |
| --- | --- | --- |
| Topic | Famous Quotes CMS: Managing Categories | Points |
| Chapter | Lecture |  |
| Activity | Practical application: filtering queries over the web | 3 |

**Managing Categories**

The roles of the authors and quote categories in the database really are very similar. They both reside in tables of their own, and they both serve to group quotes together in some way. As a result, categories can be handled with the same code we’ve developed for authors, with one important exception.

When we delete a category, we can’t simultaneously delete any quotes that belong to that category, because those quotes may also belong to other categories. We could check each quote to see if it belonged to any other categories, and only delete those that did not, but rather than engage in such a time-consuming process, let’s allow for the possibility of including quotes in our database that don’t belong to any category at all. These quotes would be invisible to our site visitors, but would remain in the database in case we wanted to assign them to a category later on.

Thus, to delete a category, we also need to delete any entries in the *fq\_quotecategory* table that refer to that category:

// Delete all quote look-up entries for the

// category along with the entry for the category.

$ok1 = @mysql\_query(

"DELETE FROM fq\_quotecategory WHERE category\_id ='$cid'");

$ok2 = @mysql\_query("DELETE FROM fq\_category WHERE category\_id='$cid'");

Other than this one detail, category management is functionally identical to author management. The complete code for the four files involved follows.

**\*File Name: categories.php**

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

<title>Quote CMS: Manage Categories</title>

<meta http-equiv="content-type"

content="text/html; charset=iso-8859-1" />

</head>

<body>

<h1>Manage Categories</h1>

<ul>

<?php

$dbcnx = @mysql\_connect('localhost', 'username', 'password');

if (!$dbcnx)

{

exit('<p>Unable to connect to the ' .

'database server at this time.</p>');

}

if (!@mysql\_select\_db('your­\_databaseName'))

{

exit('<p>Unable to locate the ' .

'database at this time.</p>');

}

$cats = @mysql\_query('SELECT category\_id, name FROM fq\_category');

if (!$cats)

{

exit('<p>Error retrieving categories from database!<br />'.

'Error: ' . mysql\_error(). '</p>');

}

while ($cat = mysql\_fetch\_array($cats))

{

$cid = $cat['category\_id'];

$cname = htmlspecialchars($cat['name']);

echo "<li>$cname ".

"<a href='editcat.php?cid=$cid'>Edit</a> ".

"<a href='deletecat.php?cid=$cid'>Delete</a></li>";

}

?>

</ul>

<p><a href="newcat.php">Add a new category</a></p>

<p><a href="index.html">Return to front page</a></p>

</body>

</html>

**\*File Name: newcat.php**

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

<title>Quote CMS: Add New Category</title>

</head>

<body>

<?php if (isset($\_POST['cname'])):

// A new category has been entered

// using the form below.

$dbcnx = @mysql\_connect('localhost', 'username', 'password');

if (!$dbcnx)

{

exit('<p>Unable to connect to the ' .

'database server at this time.</p>');

}

if (!@mysql\_select\_db('your\_databaseName'))

{

exit('<p>Unable to locate the ' .

'database at this time.</p>');

}

$cname = $\_POST['cname'];

$sql = "INSERT INTO fq\_category SET name='$cname'";

if (@mysql\_query($sql))

{

echo '<p>New category added</p>';

}

else

{

echo '<p>Error adding new category: ' .

mysql\_error() . '</p>';

}

?>

<p><a href="<?php echo $\_SERVER['PHP\_SELF']; ?>">Add another category</a></p>

<p><a href="categories.php">Return to category list</a></p>

<?php else: // Allow the user to enter a new category ?>

<form action="<?php echo $\_SERVER['PHP\_SELF']; ?>" method="post">

<p>Enter the new category:</p>

<label>Name: <input type="text" name="cname" /></label><br />

<input type="submit" value="SUBMIT" />

</form>

<?php endif; ?>

</body>

</html>

**\*File Name: editcat.php**

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

<title>Quote CMS: Edit Category</title>

</head>

<body>

<?php

$dbcnx = @mysql\_connect('localhost', 'username', 'password');

if (!$dbcnx)

{

exit('<p>Unable to connect to the ' .

'database server at this time.</p>');

}

if (!@mysql\_select\_db('your\_databaseName'))

{

exit('<p>Unable to locate the ' .

'database at this time.</p>');

}

if (isset($\_POST['cname'])):

// The category's details have

// been updated.

$cname = $\_POST['cname'];

$cid = $\_POST['cid'];

$sql = "UPDATE fq\_category SET

name='$cname'

WHERE category\_id ='$cid'";

if (@mysql\_query($sql))

{

echo '<p>Category details updated.</p>';

}

else

{

echo '<p>Error updating category details: ' .

mysql\_error() . '</p>';

}

?>

<p><a href="categories.php">Return to category list</a></p>

<?php else: // Allow the user to edit the category

$cid = $\_GET['cid'];

$cats = @mysql\_query("SELECT name FROM fq\_category WHERE category\_id = '$cid'");

if (!$cats) {

exit('<p>Error fetching category details: ' .

mysql\_error() . '</p>');

}

$cat = mysql\_fetch\_array($cats);

$cname = $cat["name"];

// Convert special characters for safe use

// as an HTML attribute.

$cname = htmlspecialchars($cname);

?>

<form action="<?php echo $\_SERVER['PHP\_SELF']; ?>" method="post">

<p>Edit the category:</p>

<label>Name: <input type="text" name="cname" value="<?php echo $cname; ?>" /></label><br />

<input type="hidden" name="cid" value="<?php echo $cid; ?>" />

<input type="submit" value="SUBMIT" /></p>

</form>

<?php endif; ?>

</body>

</html>

**\*File Name: deletecat.php**

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

<title>Quote CMS: Delete Category</title>

<meta http-equiv="content-type"

content="text/html; charset=iso-8859-1" />

</head>

<body>

<?php

$dbcnx = @mysql\_connect('localhost', 'username', 'password');

if (!$dbcnx)

{

exit('<p>Unable to connect to the ' .

'database server at this time.</p>');

}

if (!@mysql\_select\_db('your\_databaseName'))

{

exit('<p>Unable to locate the ' .

'database at this time.</p>');

}

// Delete all quote lookup entries for the

// category along with the entry for the category.

$cid = $\_GET['cid'];

$ok1 = @mysql\_query("DELETE FROM fq\_quotecategory WHERE category\_id='$cid'");

$ok2 = @mysql\_query("DELETE FROM fq\_category WHERE category\_id='$cid'");

if ($ok1 and $ok2) {

echo '<p>Category deleted successfully!</p>';

} else {

echo '<p>Error deleting category from database!<br />'.

'Error: ' . mysql\_error() . '</p>';

}

?>

<p><a href="categories.php">Return to category list</a></p>

</body>

</html>

**\*\*\*\* Show your Instructor your project up to this point and demonstrate that you can manage the categories. \*\*\*\***

**Practical application:** Create a web form that will take a value that will be used to filter returning records. Submit the source code for the form. Perform a query that filters data results. Submit the source code and a screen shot of the filtered data. Lesson #20

|  |  |  |
| --- | --- | --- |
| Topic | Famous Quotes CMS: Managing Quotes | Points |
| Chapter | Lecture |  |
| Project | Five: Enhancements to Project Four | 5 |

***Managing Quotes***

Along with the addition, deletion, and modification of quotes in our database, we

also need to be able to assign categories and authors to our quotes. Furthermore, we’re likely to have many more quotes than authors or categories. As a result, to try to display a complete list of quotes, as we did for the authors and categories, could result in an unmanageably long list with no easy way to spot the quote we’re after. We need to create a more intelligent method of browsing our library of quotes.

***Searching for Quotes***

Sometimes, we may know the category, author, or some of the text in a quote with which we want to work, so let’s support all of these methods for finding quotes in our database. When we’re done, it should work like a simple search engine. The form that prompts the administrator for information about the desired quote must present lists of categories and authors. The code for the search form is as follows:

**\*File Name: quotes.php**

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

<title>Quote CMS: Manage Quotes</title>

<meta http-equiv="content-type"

content="text/html; charset=iso-8859-1" />

</head>

<body>

<h1>Manage Quotes</h1>

<p><a href="newquote.php">Create New Quote</a></p>

<?php

$dbcnx = @mysql\_connect('localhost', 'username', 'password');

if (!$dbcnx)

{

exit('<p>Unable to connect to the ' .

'database server at this time.</p>');

}

if (!@ mysql\_select\_db('your\_databaseName'))

{

exit('<p>Unable to locate the ' .

'database at this time.</p>');

}

$authors = @mysql\_query('SELECT author\_id, name FROM fq\_author');

if (!$authors)

{

exit('<p>Unable to obtain author list from the database.</p>');

}

$cats = @mysql\_query('SELECT category\_id, name FROM fq\_category');

if (!$cats)

{

exit('<p>Unable to obtain category list from the database.</p>');

}

?>

<form action="quotelist.php" method="post">

<p>View quotes satisfying the following criteria:</p>

<label>By author:

<select name="aid" size="1">

<option selected value="">Any Author</option>

<?php

while ($author = mysql\_fetch\_array($authors))

{

$aid = $author['author\_id'];

$aname = htmlspecialchars($author['name']);

echo "<option value='$aid'>$aname</option>\n";

}

?>

</select></label><br />

<label>By category:

<select name="cid" size="1">

<option selected value="">Any Category</option>

<?php

while ($cat = mysql\_fetch\_array($cats))

{

$cid = $cat['category\_id'];

$cname = htmlspecialchars($cat['name']);

echo "<option value='$cid'>$cname</option>\n";

}

?>

</select></label><br />

<label>Containing text: <input type="text" name="searchtext" /></label><br />

<input type="submit" value="Search" />

</form>

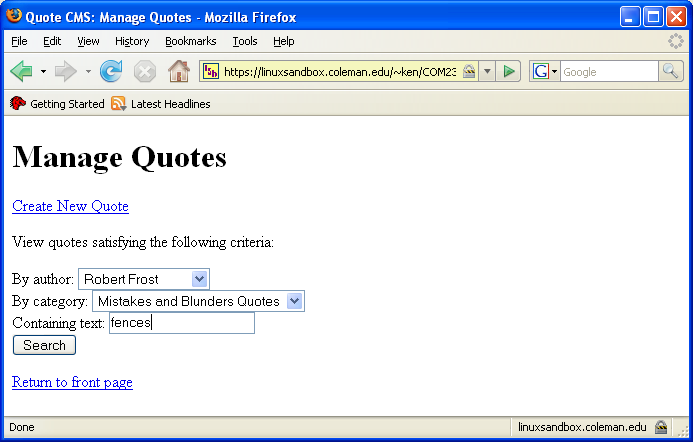
<p><a href="index.html">Return to front page</a></p>

</body>

</html>

Note the \n that appears at the end of the strings output by the echo statements. This is the special code for a new line, and serves to make the HTML code output by this script more readable. Also, note the use of *htmlspecialchars* to ensure that author and category names don’t contain any troublesome characters when they’re displayed.

The finished form appears in Figure 4-1.

Figure 4-1

It’s up to *quotelist.php* to use the values submitted through the above form to build a list of quotes that satisfies the criteria specified. Obviously, this will be done with a SELECT query, but the exact nature of that query will depend on what was entered through the form we defined above. Because the building of this SELECT statement is a fairly complicated process, let’s work through *quotelist.php* a little at a time. After this section the file will be shown in its entirety.

First, we get the preliminaries out of the way:

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

<title>Quote CMS: Manage Quotes</title>

<meta http-equiv="content-type"

content="text/html; charset=iso-8859-1" />

</head>

<body>

<h1>Manage Quotes</h1>

<?php

$dbcnx = @mysql\_connect('localhost', 'username', 'password');

if (!$dbcnx) {

exit('<p>Unable to connect to the ' .

'database server at this time.</p>');

}

if (!@mysql\_select\_db('your\_databaseName')) {

exit('<p>Unable to locate the ' .

'database at this time.</p>');

}

Now, to start, we define a few strings that, when strung together, form the SELECT query we’d need if no constraints had been selected in the form:

// The basic SELECT statement

$select = 'SELECT DISTINCT fq\_quotes.quote\_id, quotetext';

$from = ' FROM fq\_quotes';

$where = ' WHERE 1=1';

You might find the WHERE clause in the above code somewhat confusing. The idea here is for us to be able to build on this basic SELECT statement, depending on the constraints selected in the form. These constraints will require us to add to the FROM and WHERE clauses (portions) of the SELECT statement. But, if no constraints were specified (i.e. the administrator wanted a list of all quotes in the database), there would be no need for a WHERE clause at all! Because it’s difficult to add to a WHERE clause that doesn’t exist, we needed to come up with a “do nothing” WHERE clause that will always be true. Thus, we have introduced the requirement that 1 equals 1, which fits the bill nicely.

Our next task is to check each of the possible constraints (author, category, and search text) that may have been set in the form, and adjust the SQL accordingly.

First, we deal with the possibility that an author was specified. The Any Author option in the form was given a value of "" (the empty string), so, if the value of that form field (stored in $\_POST['aid']) is not equal to "", then an author has been specified, and we must adjust our query:

$aid = $\_POST['aid'];

if ($aid != '')

{

// An author is selected

$where .= " AND author\_id = '$aid'";

}

**.=**, the **append operator** is used to tack a new string onto the end of an existing one. In this case, we add to the WHERE clause the condition that the author\_id in the quote table must match the author ID selected in the form ($aid).

Next, we handle the specification of a quote category:

$cid = $\_POST['cid'];

if ($cid != '')

{

// A category is selected

$from .= ', fq\_quotecategory';

$where .= " AND fq\_quotes.quote\_id = fq\_quotecategory.quote\_id

AND category\_id = '$cid'";

}

As the categories associated with a particular quote are stored in the *quotecategory* table, we need to add this table to the query to create a join. To do this, we simply tack the name of the table onto the end of the $from variable. To complete the join, we need also to specify that the id column (in the quote table) must match the quote\_id column (in *quotecategory*), so we add this condition to the $where variable. Finally, we require the category\_id column (in quotecategory) to match the category ID selected in the form ($cid).

Handling search text is fairly simple, and uses the LIKE SQL operator.

$searchtext = $\_POST['searchtext'];

if ($searchtext != '')

{

// Some search text was specified

$where .= " AND quotetext LIKE '%$searchtext%'";

}

Now that we’ve built our SQL query, we can use it to retrieve and display our quotes along with links that allow us to edit and delete them, just like we did for the authors and quote categories. For readability, we display our quotes in an HTML table:

?>

<table>

<tr><th>Quote Text</th><th>Options</th></tr>

<?php

$quotes = @mysql\_query($select . $from . $where);

if (!$quotes) {

echo '</table>';

exit('<p>Error retrieving quotes from database!<br />'.

'Error: ' . mysql\_error() . '</p>');

}

while ($quote = mysql\_fetch\_array($quotes))

{

echo "<tr valign='top'>\n";

$qid = $quote['quote\_id'];

$quotetext = htmlspecialchars($quote['quotetext']);

echo "<td>$quotetext</td>\n";

echo "<td><a href='editquote.php?qid=$qid'>Edit</a> | " .

"<a href='deletequote.php?qid=$qid'>Delete</a></td>\n";

echo "</tr>\n";

}

?>

</table>

<p><a href="quotes.php">New search</a></p>

</body>

</html>

**\*File Name: quotelist.php**

!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

<title>Quote CMS: Manage Quotes</title>

<meta http-equiv="content-type"

content="text/html; charset=iso-8859-1" />

</head>

<body>

<h1>Manage Quotes</h1>

<?php

$dbcnx = @mysql\_connect('localhost', 'username', 'password');

if (!$dbcnx)

{

exit('<p>Unable to connect to the ' .

'database server at this time.</p>');

}

if (!@mysql\_select\_db('your\_databaseName'))

{

exit('<p>Unable to locate the ' .

'database at this time.</p>');

}

// The basic SELECT statement

$select = 'SELECT DISTINCT fq\_quotes.quote\_id, quotetext';

$from = ' FROM fq\_quotes';

$where = ' WHERE 1=1';

$aid = $\_POST['aid'];

if ($aid != '')

{

// An author is selected

$where .= " AND author\_id = '$aid'";

}

$cid = $\_POST['cid'];

if ($cid != '')

{

// A category is selected

$from .= ', fq\_quotecategory';

$where .= " AND fq\_quotes.quote\_id = fq\_quotecategory.quote\_id AND category\_id='$cid'";

}

$searchtext = $\_POST['searchtext'];

if ($searchtext != '')

{

// Some search text was specified

$where .= " AND quotetext LIKE '%$searchtext%'";

}

?>

<table>

<tr><th>Quote Text</th><th>Options</th></tr>

<?php

$quotes = @mysql\_query($select . $from . $where);

if (!$quotes)

{

echo '</table>';

exit('<p>Error retrieving quotes from database!<br />'.

'Error: ' . mysql\_error() . '</p>');

}

while ($quote = mysql\_fetch\_array($quotes))

{

echo "<tr valign='top'>\n";

$qid = $quote['quote\_id'];

$quotetext = htmlspecialchars($quote['quotetext']);

echo "<td>$quotetext</td>\n";

echo "<td><a href='editquote.php?qid=$qid'>Edit</a> | " .

"<a href='deletequote.php?qid=$qid'>Delete</a></td>\n";

echo "</tr>\n";

}

?>

</table>

<p><a href="quotes.php">New search</a></p>

</body>

</html>

The search results will display as shown in Figure 4-2.

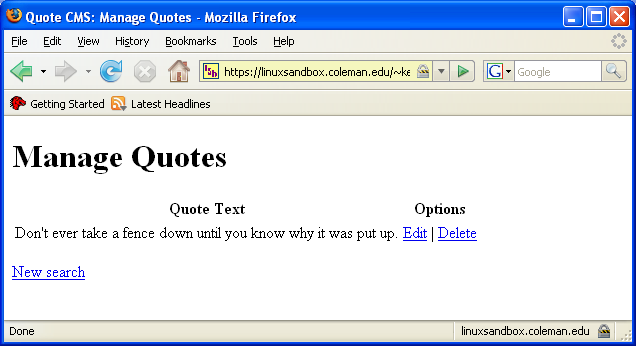


Figure 4-2

***Adding Quotes***

With *quotelist.php* out of the way, let’s tackle *newquote.php*, which is linked to from the top of *quotes.php*. This page will be very similar to *newauthor.php* and *newcat*.php. However, in addition to specifying the quote text, the page must allow an administrator to assign an author and categories to a quote. These features make the code of this file worth some examination. The complete file will be printed after the discussion.

We know from viewing the code of *newauthor.php* that the PHP code that processes the form submission comes before the form code itself. It doesn’t have to, but this is the layout we’ve used so far. Let’s begin by looking at the form code, so that the code for handling form submissions makes more sense.

First, we fetch lists of all the authors and categories in the database:

<?php

else: // Allow the user to enter a new quote

$authors = @mysql\_query('SELECT author\_id, name FROM author');

if (!$authors)

{

exit('<p>Unable to obtain author list from the

database.</p>');

}

$cats = @mysql\_query('SELECT category\_id, name FROM category');

if (!$cats)

{

exit('<p>Unable to obtain category list from the

database.</p>');

}

?>

Next, we create our form. We begin with a standard text area into which we can type the text of the quote:

<form action="<?php echo $\_SERVER['PHP\_SELF']; ?>" method="post">

<p>Enter the new quote:<br />

<textarea name="quotetext" rows="5" cols="45">

</textarea></p>

We’ll prompt the administrator to select an author from a drop-down list of the authors in the database:

<p>Author:

<select name="aid" size="1">

<option selected value="">Select One</option>

<option value="">---------</option>

<?php

while ($author = mysql\_fetch\_array($authors)) {

$aid = $author['author\_id'];

$aname = htmlspecialchars($author['name']);

echo "<option value='$aid'>$aname</option>\n";

}

?>

</select></p>

A drop-down list won’t suffice for the selection of categories, though, because we want the administrator to be able to select multiple categories. Thus, we’ll use a series of check boxes—one for each category. Since we have no way to know in advance the number of check boxes we’ll need, the matter of naming them becomes an interesting challenge.

What we’ll actually do is use a *single* variable for all the check boxes; thus, all the check boxes will have the same name. To be able to receive multiple values from a single variable name, we must make that variable an **array**. To submit a form element as part of an array variable, we simply add a pair of square brackets to the end of the variable name (making it cats[] in this case). With all of our check boxes named the same, we’ll need a way to identify which particular check boxes have been selected. To this end, we assign a different value to each check box—the ID of the corresponding category in the database. Thus, the form submits an array that contains the IDs of all the categories to which the new quote should be added.

<p>Place in categories:<br />

<?php

while ($cat = mysql\_fetch\_array($cats))

{

$cid = $cat['category\_id'];

$cname = htmlspecialchars($cat['name']);

echo "<label><input type='checkbox' name='cats[]'

value='$cid' />$cname</label><br />\n";

}

?>

</p>

And we finish off our form as usual:

<input type="submit" value="SUBMIT" />

</form>

<?php endif; ?>

</body>

</html>

Figure 4-3 shows what this form will look like.

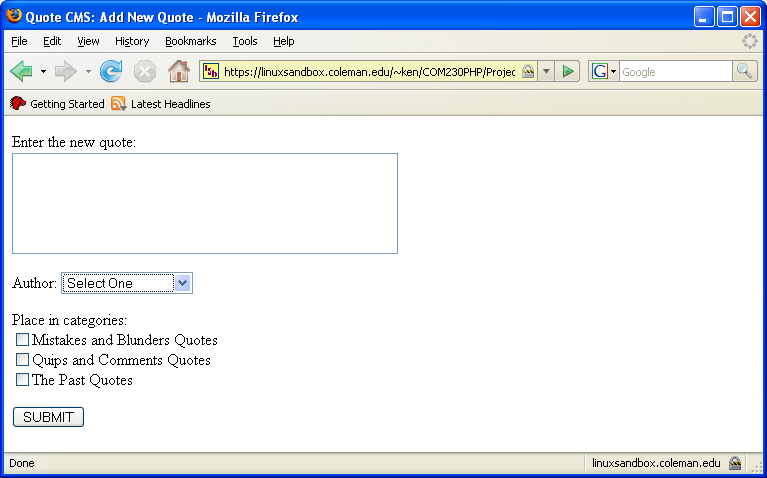


Figure 4-3

Since we’re submitting an array for the first time, the code that processes this form is not totally straightforward. It starts off pretty simply as we add the quote to the quote table. As an author is required, we make sure that $\_POST['aid'] contains a value. This prevents the administrator from choosing the Select One option in the author select list (that choice has a value of "", the empty string).

<?php

$dbcnx = @mysql\_connect('localhost', 'username', 'password');

if (!$dbcnx) {

exit('<p>Unable to connect to the ' .

'database server at this time.</p>');

}

if (!@mysql\_select\_db('your\_databaseName')) {

exit('<p>Unable to locate the ' .

'database at this time.</p>');

}

if (isset($\_POST['quotetext'])):

// A new quote has been entered

// using the form.

$aid = $\_POST['aid'];

$quotetext = $\_POST['quotetext'];

$cats = $\_POST['cats'];

if ($aid == '')

{

exit('<p>You must choose an author for this quote. Click

"Back" and try again.</p>');

}

$sql = "INSERT INTO fq\_quotes SET

quotetext='$quotetext',

quotedate=CURDATE(),

author\_id='$aid'";

if (@mysql\_query($sql))

{

echo '<p>New Quote added</p>';

}

else

{

exit('<p>Error adding new quote: ' . mysql\_error() . '</p>');

}

$qid = mysql\_insert\_id();

The last line in the above code uses a function that we haven’t seen before: *mysql\_insert\_id*. This function returns the number assigned to the last-inserted entry by the AUTO\_INCREMENT feature in MySQL. In other words, it retrieves the ID of the newly inserted quote, which we’ll need momentarily.

The code that adds the entries to *quotecategory* based on which check boxes were checked is not so simple. First of all, we’ve never seen how a check box passes its value to a PHP variable before. Also, we need to deal with the fact that these particular check boxes will submit into an array variable.

A typical check box will pass its value to a PHP variable if it is checked, and will do nothing when it is unchecked. Check boxes without assigned values pass 'on' as the value of their corresponding variables when they are checked. However, we’ve assigned values to our check boxes (the category IDs), so this is not an issue.

The fact that these check boxes submit into an array actually adds quite a measure of convenience to our code. In essence, what we’ll receive from the submitted form is either:

1. an array of category IDs to add the quote to

2. nothing at all (if none of the check boxes were checked)

First, let’s handle the latter, special case by creating an empty array when we find that the $\_POST['cats'] variable is empty:

if (isset($\_POST['cats']))

{

$cats = $\_POST['cats'];

}

else

{

$cats = array();

}

The array function that appears here is used to create a new array in PHP. The parameters that are passed to it become the elements of the array. Because we’re not passing parameters to it here, it will simply create an empty array.

Now that we’ve guaranteed that the $cats variable contains an array, we can use a loop to consider each category ID in the array in turn, and to insert the appropriate entry into the database. Since this array isn’t based on a database row, you might wonder how we can access the values in the array. After all, we’ve usually retrieved an array value using its database column name (e.g. $cat['name']). In this case, our array was created simply by feeding a series of values into the same variable name. When this happens, PHP automatically assigns numerical indices to the values in the array.

For instance, the value of the first check box that was checked will be submitted first into the array and will be accessible as $cat[0]. That is, PHP assigns it an array index of 0. The second check box that is checked will have its value stored with an index of 1, accessible as $cat[1]. So if *n* check boxes are checked, then the value of the last check box will be in $cat[*n* - 1]. By counting up through the array indexes as we proceed through a loop in our code, we can process the elements of this array one at a time.

But wait… what is *n*? We have no way of knowing in advance how many check boxes will be checked, so how should the loop know when to stop counting? Well, there are two ways we can ascertain this value. The first is to use a PHP function called count that takes an array as a parameter and counts the number of elements in it. Here’s what our while loop would look like if we use this method:

$i = 0; // First index

while ($i < count($cats))

{

// While we're not at the end

*// process $cats[$i]*

++$i; // Increment to the next index

}

As you can see, this loop uses a counter variable ($i), that is, a variable that counts the number of times the loop has executed. The first time through the loop, it will have a value of 0; at the end of the loop we’ll add 1 to it. Therefore, the second time through the loop, it will have a value of 1, and so on. Within the loop, we can use this variable as the array index to pull a category ID out of the $cats array. The loop stops executing when $i reaches count($cats), the number of elements in the $cats array. If $cats doesn’t contain any elements (i.e. no categories were selected), then $i will start out equal to count($cats), and the contents of the loop won’t be executed at all.

This all seems very slick, but there’s actually a better method. Instead of using the count function, we can keep going until we reach a value of $i for which $cat[$i] is empty. When we do, we’ll know we’ve reached the end of the list of category IDs:

$i = 0; // First index

while (isset($cats[$i]))

{

// While we're not at the end

*// process $cats[$i]*

++$i; // Increment to the next index

}

This will run a little faster because we don’t call a function each time through the loop. Plus, it’s a teensy bit more clever, and we programmers have to have our fun when we can!

Going one step further, the especially attentive may recognize this while loop as a perfect candidate for replacement with a for loop. Here’s what the equivalent *for* loop looks like:

for ($i = 0; isset($cats[$i]); ++$i)

{

*// process $cats[$i]*

}

Not bad, right? Well, believe it or not, PHP spoils our fun by having a completely separate type of loop that’s specialized for looping through arrays, called a ***foreach loop***. Here’s what the code looks like in this case:

foreach ($cats as $catID)

{

*// Process $catID*

}

This *foreach* loop will execute the code inside the loop once *for each* item in the $cats array (you see where the *foreach* loop gets its name), and will assign the item for each loop to the variable $catID. Since this code is indisputably tidier than the equivalent *while* loop, we’ll settle on this as a solution. All that remains is to determine what to do for each selected category ID.

Before we became sidetracked by all these different types of loops, we were about to take our array of category IDs and use it to place our newly-inserted quote into its corresponding categories. A cursory examination of our database layout reveals that we just have to insert an entry into the *quotecategory* table for each category of which that quote should be a member. Recall that each entry in the *quotecategory* table consists of a quote ID and a category ID, which together indicate that a particular quote belongs to a particular category. Here’s the finished *foreach* loop:

$numCats = 0;

foreach ($cats as $catID)

{

$sql = "INSERT IGNORE INTO fq\_quotecategory

SET quote\_id = $qid, category\_id = $catID";

$ok = @mysql\_query($sql);

if ($ok)

{

$numCats = $numCats + 1;

}

else

{

echo "<p>Error inserting quote into category $catID: " .

mysql\_error() . '</p>';

}

}

?>

<p>Quote was added to <?php echo $numCats; ?> categories.</p>

<p><a href="<?php echo $\_SERVER['PHP\_SELF']; ?>">Add another quote</a></p>

<p><a href="quotes.php">Return to quote search</a></p>

The word IGNORE in the INSERT query used here is a precaution only. Recall that, when we defined the *quotecategory* table, we set the quote\_id and category\_id columns to be the primary key for the table. If somehow the quote\_id/category\_id pair that is inserted already exists in the table, an attempt to insert it again would normally cause an error. By adding IGNORE to the command, a reinsert of the same pair is simply ignored by MySQL and no error occurs. This situation should never actually happen, but it’s better to be safe than sorry.

**\*File Name: newquote.php**

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

<title>Quote CMS: Add New Quote</title>

<meta http-equiv="content-type"

content="text/html; charset=iso-8859-1" />

</head>

<body>

<?php

$dbcnx = @mysql\_connect('localhost', 'username', 'password');

if (!$dbcnx)

{

exit('<p>Unable to connect to the ' .

'database server at this time.</p>');

}

if (!@mysql\_select\_db('your\_databaseName'))

{

exit('<p>Unable to locate the ' .

'database at this time.</p>');

}

if (isset($\_POST['quotetext'])):

// A new quote has been entered

// using the form.

$aid = $\_POST['aid'];

$quotetext = $\_POST['quotetext'];

$cats = $\_POST['cats'];

if ($aid == '')

{

exit('<p>You must choose an author for this quote. Click

"Back" and try again.</p>');

}

$sql = "INSERT INTO fq\_quotes SET

quotetext='$quotetext',

quotedate=CURDATE(),

author\_id='$aid'";

if (@mysql\_query($sql))

{

echo '<p>New Quote added</p>';

}

else

{

exit('<p>Error adding new quote: ' . mysql\_error() . '</p>');

}

$qid = mysql\_insert\_id();

if (isset($\_POST['cats']))

{

$cats = $\_POST['cats'];

}

else

{

$cats = array();

}

$numCats = 0;

foreach ($cats as $catID)

{

$sql = "INSERT IGNORE INTO fq\_quotecategory

SET quote\_id = $qid, category\_id = $catID";

$ok = @mysql\_query($sql);

if ($ok)

{

$numCats = $numCats + 1;

}

else

{

echo "<p>Error inserting quote into category $catID: " .

mysql\_error() . '</p>';

}

}

?>

<p>Quote was added to <?php echo $numCats; ?> categories.</p>

<p><a href="<?php echo $\_SERVER['PHP\_SELF']; ?>">Add another quote</a></p>

<p><a href="quotes.php">Return to quote search</a></p>

<?php

else: // Allow the user to enter a new quote

$authors = @mysql\_query('SELECT author\_id, name FROM

fq\_author');

if (!$authors)

{

exit('<p>Unable to obtain author list from the

database.</p>');

}

*$cats = @mysql\_query('SELECT category\_id, name FROM*

*fq\_category');*

if (!$cats)

{

exit('<p>Unable to obtain category list from the

database.</p>');

}

?>

<form action="<?php echo $\_SERVER['PHP\_SELF']; ?>" method="post">

<p>Enter the new quote:<br />

<textarea name="quotetext" rows="5" cols="45">

</textarea></p>

<p>Author:

<select name="aid" size="1">

<option selected value="">Select One</option>

<option value="">---------</option>

<?php

while ($author = mysql\_fetch\_array($authors))

{

$aid = $author['author\_id'];

$aname = htmlspecialchars($author['name']);

echo "<option value='$aid'>$aname</option>\n";

}

?>

</select></p>

<p>Place in categories:<br />

<?php

while ($cat = mysql\_fetch\_array($cats))

{

$cid = $cat['category\_id'];

$cname = htmlspecialchars($cat['name']);

echo "<label><input type='checkbox' name='cats[]'

value='$cid' />$cname</label><br />\n";

}

?>

</p>

<input type="submit" value="SUBMIT" />

</form>

<?php endif; ?>

</body>

</html>

***Editing and Deleting Quotes***

The two files that remain, *editquote.php* and *deletequote.php*, mirror their author and category counterparts, with minor adjustments. *editquote.php* must provide the same author select box and category check boxes as *newquote.php*, except that this time they must be initialized to reflect those values stored in the database for the particular quote we’ve selected. *deletequote.php*, meanwhile, must delete the selected quote from the quote table, and must also remove any entries in the *quotecategory* table for that quote.

As all of this falls within reach of the skills you’ve developed in the preceding sections, I’ll present the code for these files without comment. Take some time to browse through them and make sure you’re comfortable with everything that’s going on within.

**\*File Name: editquote.php**

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

<title>Quote CMS: Edit Quote</title>

<meta http-equiv="content-type"

content="text/html; charset=iso-8859-1" />

</head>

<body>

<?php

$dbcnx = @mysql\_connect('localhost', 'username', 'password');

if (!$dbcnx)

{

exit('<p>Unable to connect to the ' .

'database server at this time.</p>');

}

if (!@mysql\_select\_db('your\_databaseName'))

{

exit('<p>Unable to locate the ' .

'database at this time.</p>');

}

if (isset($\_POST['quotetext'])):

// The quote's details have

// been updated.

$qid = $\_POST['qid'];

$aid = $\_POST['aid'];

$quotetext = $\_POST['quotetext'];

$sql = "UPDATE fq\_quotes SET

quotetext='$quotetext',

author\_id='$aid'

WHERE quote\_id='$qid'";

if (mysql\_query($sql))

{

echo '<p>Quote details updated.</p>';

}

else

{

exit('<p>Error updating quote details: ' .

mysql\_error() . '</p>');

}

// Delete all existing entries for this

// quote from the quotecategory table

$ok = mysql\_query("DELETE FROM fq\_quotecategory

WHERE quote\_id = '$qid'");

if (!$ok)

{

exit('<p>Error removing quote from all categories:' .

mysql\_error() . '</p>');

}

if (isset($\_POST['cats']))

{

$cats = $\_POST['cats'];

}

else

{

$cats = array();

}

foreach ($cats as $catID)

{

$sql = "INSERT IGNORE INTO fq\_quotecategory

SET quote\_id = '$qid', category\_id = '$catID'";

$ok = @mysql\_query($sql);

if (!$ok)

{

echo "<p>Error inserting quote into category $catID: " .

mysql\_error() . '</p>';

}

}

?>

<p><a href="quotes.php">New quote search</a></p>

<?php else: // Allow the user to edit the quote

$qid = $\_GET['qid'];

$quote = @mysql\_query(

"SELECT quotetext, author\_id FROM fq\_quotes WHERE quote\_id =

'$qid'");

if (!$quote)

{

exit('<p>Error fetching quote details: ' .

mysql\_error() . '</p>');

}

$quote = mysql\_fetch\_array($quote);

$quotetext = $quote['quotetext'];

$authid = $quote['author\_id'];

// Convert HTML special characters

// in database value for use in

// an HTML document.

$quotetext = htmlspecialchars($quotetext);

// Get lists of authors and categories for

// the select box and checkboxes.

$authors = @mysql\_query('SELECT author\_id, name FROM fq\_author');

if (!$authors)

{

exit('<p>Unable to obtain author list from the database.</p>');

}

$cats = @mysql\_query('SELECT category\_id, name FROM

fq\_category');

if (!$cats)

{

exit('<p>Unable to obtain category list from the

database.</p>');

}

?>

<form action="<?php echo $\_SERVER['PHP\_SELF']; ?>" method="post">

<p>Edit the quote:<br />

<textarea name="quotetext" rows="5" cols="45">

<?php echo $quotetext; ?></textarea>

<p>Author:

<select name="aid" size="1">

<?php

while ($author = mysql\_fetch\_array($authors))

{

$aid = $author['author\_id'];

$aname = htmlspecialchars($author['name']);

if ($aid == $authid)

{

echo "<option selected='selected'

value='$aid'>$aname</option>\n";

}

else

{

echo "<option value='$aid'>$aname</option>\n";

}

}

?>

</select></p>

<p>In categories:<br />

<?php

while ($cat = mysql\_fetch\_array($cats))

{

$cid = $cat['category\_id'];

$cname = htmlspecialchars($cat['name']);

// Check if the quote is in this category

$result = @mysql\_query(

"SELECT \* FROM fq\_quotecategory

WHERE quote\_id = '$qid' AND category\_id = '$cid'");

if (!$result)

{

exit('<p>Error fetching quote details: ' .

mysql\_error() . '</p>');

}

// mysql\_num\_rows gives the number of entries

// in a result set. In this case, if the result

// contains one or more rows, the condition

// below will evaluate to true to indicate that

// the quote does belong to the category, and the

// checkbox should be checked.

if (mysql\_num\_rows($result))

{

echo "<input type='checkbox' checked='checked'

name='cats[]' value='$cid' />$cname<br />\n";

}

else

{

echo "<input type='checkbox' name='cats[]' value='$cid'

/>$cname<br />\n";

}

}

?>

</p>

<input type="hidden" name="qid" value="<?php echo $qid; ?>" />

<input type="submit" value="SUBMIT" />

</form>

<?php endif; ?>

</body>

</html>

**\*File Name: deletequote.php**

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

<title>Quote CMS: Delete Quote</title>

<meta http-equiv="content-type"

content="text/html; charset=iso-8859-1" />

</head>

<body>

<?php

$dbcnx = @mysql\_connect('localhost', 'username', 'password');

if (!$dbcnx)

{

exit('<p>Unable to connect to the ' .

'database server at this time.</p>');

}

if (!@mysql\_select\_db('your\_databaseName'))

{

exit('<p>Unable to locate the ' .

'database at this time.</p>');

}

// Delete all quote lookup entries for the

// quote along with the entry for the quote.

$qid = $\_GET['qid'];

$ok1 = @mysql\_query("DELETE FROM fq\_quotecategory WHERE quote\_id = '$qid'");

$ok2 = @mysql\_query("DELETE FROM fq\_quotes WHERE quote\_id = '$qid'");

if ($ok1 and $ok2)

{

echo '<p>Quote deleted successfully!</p>';

}

else

{

echo '<p>Error deleting quote from database!<br />'.

'Error: ' . mysql\_error() . '</p>';

}

?>

<p><a href="quotes.php">New quote search</a></p>

</body>

</html>

***Summary***

There are a few minor tasks of which our content management system is still incapable. For example, it’s unable to provide a listing of all quotes that don’t belong to *any* category—something that could come in very handy as the number of quotes in the database grows. You might also like to sort quote listings by various criteria. These particular capabilities require a few more advanced SQL tricks.

If we ignore these little details for the moment, you’ll see that you now have a system that allows someone without SQL or database knowledge to administer your database of quotes with ease! Together with a set of PHP-powered pages through, which regular site visitors can view the quotes, this content management system allows us to set up a complete database-driven Website that can be maintained by someone with absolutely no database knowledge. And if you think that sounds like a valuable commodity to businesses looking to get on the Web today, you’re right!

In fact, only one aspect of our site requires users to have special knowledge (beyond the use of a Web browser): content formatting. For example, it would not be unusual for someone to want to enter a quote that contained more than one paragraph of text. In our current system, this could be accomplished by entering the HTML code for the quote directly into the new quote form. Why is this unacceptable?

As we stated way back in the introduction to this tutorial, one of the most desirable features of a database-driven Website is that the people responsible for adding content to the site need not be familiar with HTML. If we require knowledge of HTML for something as simple as dividing a quote into paragraphs, we have failed to achieve our goal.

If there is time at the end of the class, we’ll see how we can make use of some features of PHP to provide a simpler means by which we can format content without requiring site administrators to know the ins and outs of HTML. We’ll also bring back the “submit your own quote” link, and discover how we can safely accept content submissions from casual site visitors.

You also learned a little more about arrays in PHP. You learned how a set of form elements can submit their values into a single array variable, and you learned how to process that array on the receiving end by looping through it with a *while* loop, a *for* loop, and a *foreach* loop.

**Project 5 - Enhancements to Project 4 - 5 Points**

Now that you have finished Project 4 you will need to make the following changes that will make the Web site a little more user friendly.

As Project 4 exists now the quote list does not display the author of the quote. Modify the quotelist.php file to display the author of the quote and add another heading to the table.

The author display uses an Unordered List to display the author name. Modify the authors.php file to use a table instead of the unordered list and also add the author’s email to the display.

Instead of a blank space or a pipe symbol between the Edit and Delete hyperlinks display a bullet symbol. Do this for all scripts that display those options.

To make the application consistent move the Create option for new quotes to just above the hyperlink to return to the front page. This link is in the quotes.php script. Lesson #21

|  |  |  |
| --- | --- | --- |
| Topic | Database Administration | Points |
| Chapter | Nine |  |
| Activity | Practical Application: protecting against SQL injection | 4 |

**Main Topics:**

* Overview of Administrative Duties
* Security
* Grant and Revoke commands
* Managing Users

The MySQL database system consists of several components. You should be familiar with what these components are and the purpose of each, so that you understand both the nature of the system and the tools available to help you.

* The MySQL server. The server mysqld, is the hub of a MySQL installation; it performs all manipulation of databases and tables.
* The server’s language, SQL. You should be able to talk to the server in it’s own language.
* The Data Directory. The data directory is where the server stores its databases and status files.

General Administration deals primarily with the operation of mysqld, the MySQL server, and with providing your users with access to the server. The following duties are most important in carrying out this responsibility:

* Server startup and shutdown
* User account maintenance
* Log file maintenance
* Server configuration and tuning
* Multiple servers
* Database replication
* Software updates

**Security**

When you maintain a MySQL installation, it’s important to make sure that the information your users entrust to their databases is kept secure. The MySQL administrator is responsible for controlling access to the data directory and the server, and should understand the following issues:

* Filesystem security
* MySQL server security

A MySQL administrator should know how to set up MySQL user accounts by specifying which users can connect to the server, where they can connect from, and what they can do while connected. This information is stored in the grant tables in the mysql database, and is managed primarily be means of these statements:

* GRANT creates MySQL accounts and specifies privileges
* REVOKE removes privileges from existing accounts
* DROP USER removes the last traces of an account from which you have revoked all privileges.
* SET PASSWORD assigns a password to an existing account

**Practical Application**: Depending on how you designed your query web form, you may have exposed your database to attack. Research the topic of SQL injection. Specifically, determine how you can examine user input in an automated way that will prevent SQL injection tricks. Implement a security solution that you found. In the PHP, include in comments the source of the injection prevention code. Submit your source code to your instructor.

Lesson #22

|  |  |  |
| --- | --- | --- |
| Topic | Transactions and Locking | Points |
| Chapter | Seven |  |

**Main Topics:**

* Transactions
* Locking

A transaction is a set of SQL statements that execute as a unit. Either all the statements execute successfully, or none of them have any effect. This is achieved through the use of commit and rollback capabilities. If all of the statements in the transaction succeed, you commit it to record their effect permanently in the database. If an error occurs during the transaction, you roll it back to cancel it. Any statements executed up to that point within the transaction are undone, leaving the database in the state it was prior to when the transaction began.

Locks can be applied to prevent concurrent users from interacting destructively with one other’s data. A read lock allows you to prevent other users from changing data while you are reading and processing the data, while a write lock tells other users that the data is being changed and that they should not read or modify it.

Lesson #23

|  |  |  |
| --- | --- | --- |
| Topics | Quiz 3 and 4, Test 2 | Points |
|  | Lessons 13 - 22 | 25 |

**Review Terms/Topics:**

**Review Terms: 5 points**

The following 29 terms with definitions/explanations will be turned in word format.

View

PHP Basics

* Statements
* Comments
* Identifiers
* Data types
* Arrays
* Variables
* Expressions and Operators
* Flow-Control statements
* Embedding PHP in Web pages
* Strings

Connecting to a MySQL server

Selecting a database

Submitting a query to MySQL

Displaying data from MySQL

Viewing and selecting records

Adding a record

Editing a record

Committing changes to the database

Deleting records

Web Site navigation

Overview of Administrative Duties

Security

GRANT command

REVOKE command

Managing Users

Transactions

Locking

XML

Alpine Adventures

Database

**Item**

|  |  |  |
| --- | --- | --- |
| Item\_Id | Description | Category |
| 1 | Women’s Hiking Shorts | Women’s Clothing |
| 2 | Women’s Fleece Pullover | Women’s Clothing |
| 3 | Children’s Beachcomber Sandals | Children’s Clothing |
| 4 | Men’s Expedition Parka | Men’s Clothing |
| 5 | 3-Season Tent | Outdoor Gear |

|  |  |  |
| --- | --- | --- |
| Column | Null ? | Type |
| ITEM\_ID\* | NOT NULL | INT |
| DESCRIPTION | NOT NULL | VARCHAR(40) |
| CATEGORY | NOT NULL | VARCHAR(20) |

\* Primary Key

**Customer (Part 1)**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Cust\_Id | Last | First | MI | Address | City | State |
| 1 | Jones | Cindy | E | 1156 Water Street Apt. 3 | Osseo | WI |
| 2 | Edwards | Mitch | M | 4204 Garner Street | Washburn | WI |
| 3 | Sorenson | Betty | H | 2211 Pine Drive | Radisson | WI |
| 4 | Miller | Lee |  | 699 Pluto St. NE | Silver Lake | WI |
| 5 | White | Alissa | R | 987 Durham Rd. | Sister Bay | WI |

|  |  |  |
| --- | --- | --- |
| Column | Null ? | Type |
| Cust\_Id | NOT NULL | INT |
| Last | NOT NULL | VARCHAR(25) |
| First | NOT NULL | VARCHAR(20) |
| MI | NULL | VARCHAR(1) |
| Address | NULL | VARCHAR(30) |
| City | NULL | VARCHAR(25) |
| State | NULL | VARCHAR(2) |
| Zip | NULL | VARCHAR(10) |
| Phone | NULL | VARCHAR2(10) |
| Email | NULL | VARCHAR(30) |

**Customer (Part 2)**

|  |  |  |
| --- | --- | --- |
| Zip | Phone | Email |
| 54705 | 7155558943 | cjones@hotmail.com |
| 54891 | 7155558243 | medwards@gmail.com |
| 54867 | 7155558332 | betty1@yahoo.com |
| 53821 | 7155554978 | leemiller@gmail.com |
| 54234 | 7155557651 | awhite@hotmail.com |

**Orders**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Order\_ID | Order\_Date | Payment | Cust\_Id | Ordersource |
| 1 | 2007-10-10 | CC | 1 | 152 |
| 2 | 2007-10-31 | CC | 2 | WEBSITE |
| 3 | 2007-11-22 | CHECK | 3 | 152 |
| 4 | 2007-11-19 | CC | 3 | 153 |
| 5 | 2007-12-12 | CC | 5 | WEBSITE |
| 6 | 2007-12-24 | CC | 5 | WEBSITE |

|  |  |  |
| --- | --- | --- |
| Column | Null ? | Type |
| ORDER\_ID | NOT NULL | INT |
| ORDER\_DATE | NOT NULL | DATE |
| PAYMENT | NOT NULL | VARCHAR(5) |
| CUST\_ID | NOT NULL | INT |
| ORDERSOURCE |  | VARCHAR(7) |

**Orderline**

|  |  |  |  |
| --- | --- | --- | --- |
| Order\_Id | Inv\_Id | Order\_Price | Qty |
| 1 | 1 | 274.99 | 1 |
| 1 | 6 | 32.95 | 2 |
| 2 | 10 | 64.95 | 1 |
| 3 | 16 | 15.99 | 1 |
| 3 | 18 | 15.99 | 1 |
| 4 | 23 | 199.95 | 1 |
| 5 | 21 | 15.99 | 2 |
| 5 | 7 | 32.95 | 1 |
| 6 | 10 | 64.95 | 1 |
| 6 | 26 | 209.95 | 1 |

|  |  |  |
| --- | --- | --- |
| Column | Null ? | Type |
| ORDER\_ID\* | NOT NULL | INT |
| INV\_ID | NOT NULL | INT |
| ORDER\_PRICE | NOT NULL | DECIMAL(6,2) |
| QTY | NOT NULL | SMALLINT |

**Ordersource**

|  |
| --- |
| Ordersource |
| 122 |
| 123 |
| 145 |
| 146 |
| 151 |
| 152 |
| 153 |
| 211 |
| 99 |
| WEBSITE |

|  |  |  |
| --- | --- | --- |
| Column | Null ? | Type |
| ORDERSOURCE | NOT NULL | VARCHAR(7) |

**Color**

|  |
| --- |
| Color |
| Blue |
| Brown |
| Coral |
| Forrest |
| Green |
| Khaki |
| Navy |
| Olive |
| Red |
| Sienna |
| Teal |

**Description**

|  |  |  |
| --- | --- | --- |
| Column | Null ? | Type |
| COLOR | NOT NULL | VARCHAR(15) |

**Shipping**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Ship\_Id | Inv\_Id | Date\_Exp | Qty\_Exp | Date\_Rec | Qty\_Rec |
| 1 | 1 | 2008-06-18 | 10 |  |  |
| 1 | 2 | 2008-06-18 | 10 |  |  |
| 2 | 5 | 2008-07-10 | 50 |  |  |
| 3 | 12 | 2008-08-19 | 50 |  |  |
| 4 | 20 | 2008-09-25 | 50 |  |  |
| 4 | 22 | 2008-09-25 | 50 |  |  |
| 5 | 8 | 2008-10-31 | 30 |  |  |
| 6 | 17 | 2008-11-05 | 20 |  |  |
| 7 | 14 | 2008-05-18 | 50 | 2008-05-18 | 50 |
| 8 | 11 | 2008-05-29 | 50 | 2008-05-29 | 50 |
| 9 | 24 | 2008-05-30 | 30 | 2008-05-30 | 0 |
| 10 | 25 | 2008-05-30 | 30 | 2008-05-30 | 0 |

|  |  |  |
| --- | --- | --- |
| Column | Null ? | Type |
| SHIP\_ID | NOT NULL | INT |
| INV\_ID | NOT NULL | INT |
| DATE\_EXP | NOT NULL | DATE |
| QTY\_EXP | NOT NULL | SMALLINT |
| DATE\_REC |  | DATE |
| QTY\_REC |  | SMALLINT |

**Backorder**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Backorder\_Id | Ship\_Id | Inv\_Id | Date\_Exp | Qty\_Exp | Date\_Rec | Qty\_Rec |
| 1 | 9 | 24 | 2008-07-21 | 30 |  |  |
| 2 | 10 | 25 | 2008-07-21 | 30 |  |  |

|  |  |  |
| --- | --- | --- |
| Column | Null ? | Type |
| BACKORDER\_ID | NOT NULL | INT |
| SHIP\_ID | NOT NULL | INT |
| INV\_ID | NOT NULL | INT |
| DATE\_EXP | NOT NULL | DATE |
| QTY\_EXP | NOT NULL | SMALLINT |
| DATE\_REC |  | DATE |
| QTY\_REC |  | SMALLINT |

**Inventory**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Inv\_Id | Item\_Id | Item\_Size | Color | Price | Quantity\_On\_Hand |
| 1 | 5 |  | Sienna | 274.99 | 14 |
| 2 | 5 |  | Forest | 274.99 | 8 |
| 3 | 1 | S | Khaki | 32.95 | 57 |
| 4 | 1 | M | Khaki | 32.95 | 89 |
| 5 | 1 | L | Khaki | 32.95 | 0 |
| 6 | 1 | S | Olive | 32.95 | 110 |
| 7 | 1 | M | Olive | 32.95 | 51 |
| 8 | 1 | L | Olive | 32.95 | 23 |
| 9 | 2 | S | Teal | 64.95 | 112 |
| 10 | 2 | M | Teal | 64.95 | 37 |
| 11 | 2 | L | Teal | 64.95 | 125 |
| 12 | 2 | S | Coral | 64.95 | 0 |
| 13 | 2 | M | Coral | 64.95 | 86 |
| 14 | 2 | L | Coral | 64.95 | 140 |
| 15 | 3 | 10 | Blue | 15.99 | 78 |
| 16 | 3 | 11 | Blue | 15.99 | 86 |
| 17 | 3 | 12 | Blue | 15.99 | 23 |
| 18 | 3 | 6 | Blue | 15.99 | 89 |
| 19 | 3 | 10 | Red | 15.99 | 56 |
| 20 | 3 | 11 | Red | 15.99 | 35 |
| 21 | 3 | 12 | Red | 15.99 | 84 |
| 22 | 3 | 6 | Red | 15.99 | 0 |
| 23 | 4 | S | Green | 199.95 | 92 |
| 24 | 4 | M | Green | 199.95 | 17 |
| 25 | 4 | L | Green | 199.95 | 0 |
| 26 | 4 | XL | Green | 209.95 | 12 |

|  |  |  |
| --- | --- | --- |
| Column | Null ? | Type |
| INV\_ID | NOT NULL | INT |
| ITEM\_ID | NOT NULL | INT |
| ITEM\_SIZE |  | VARCHAR(2) |
| COLOR |  | VARCHAR(15) |
| PRICE |  | DECIMAL(6,2) |
| QUANTITY\_ON\_HAND |  | INT |