WHAT IS A DATABASE?

A database is any organized collection of data that fulfills some purpose. As weather researchers, you will often have to access and evaluate large amounts of weather data, and this data will be organized into some kind of database.

There are at least six commonly known database types: flat, hierarchical, network, relational, dimensional, and object.

Flat databases are just a table of data. The data can be organized into groups either by row or column, and certain relations or attributes exist in the corresponding columns or rows. Spreadsheets are a good example of flat databases.

MORE DATABASE TYPES

Hierarchical databases are tree structures. Each set of data has a single parent set to which it is related. Data sets on the same level of the tree are sorted in some fashion. Ordered and nested data fits nicely into hierarchical databases, such as tables of contents or recipes.

Network databases are linked lists of related sets of data. Each set of data can have a related link to some other set. The links form a semi-ordered listing of database elements. Early web search engines (and possibly current ones) used this database type.

Dimensional databases are a cross between relational databases and hierarchical databases. A dimensional database consists of one large table that describes dimensions and measures. Dimensions are the context of the data and measures are the quantities of the data. Dimensions are organized hierarchically and measures are usually ordered.

MORE DATABASE TYPES

Object databases attempt to encapsulate data in the same manner as object-oriented programming languages. There is no set standard for this type of database currently.

Relational databases are composed of tables of data (each similar to a flat database). Each column in a table describes some attribute of the rows of the table. Each row is an actual object (usually called a record) in the database with the attributes corresponding to the columns. Records in relational databases are unique (you cannot have two copies of the same data in a table).

Tables are related to each other through the use of keys. Keys are orderings of column(s) of a table.

Keys relate records from different tables to each other (hence the name relational database). We will see more on this later.

Each type of database has its use. The type of application that needs to access the data and the amount of data stored will determine the best type of database to use.

Relational databases are good for large volumes of data with predefined relationships, particularly where flexibility and speed are necessary. A well structured relational database is also good at saving disk space when it comes to storage due to low data replication.

CREATING RELATIONAL DATABASES

To use relational databases properly, it is important to first understand how they are created. It is important to create a good structure when defining a relational database. This reduces the amount of space the database will take and eliminates data replication (places where the same data appears more than once). Data replication creates problems when we need to update the database (as we have to find and change every instance of a piece of data). To demonstrate this process, we are going to create a database for weather data.

Imagine that we have a small network of weather stations. Each weather station reports weather data to us hourly. We need some way to track and store this data.

ENTITIES

To design the database, we need to identify the entities (things we are storing data about) and the relationships (how entities tie together). In our case, weather stations report air temperature, relative humidity, pressure, wind speed, wind direction, total rainfall, and solar radiation once per hour. Each weather station also has a location, manufacturer, and maintenance record.

What are the entities we need? On first blush, we might consider two entities; weather reports and stations. The relationship is that each report is tied to a single station that gave the report.

NORMALIZATION

Normalization is the removal of data redundancy from a database design. As previously mentioned, data replication creates problems with updating the database and also takes more space. After we have identified our initial entities and relationships, it is important to normalize the database structure to remove as much redundancy as possible.

There are three stages of normalization: first normal form, second normal form, and third normal form.

FIRST NORMAL FORM

A database is in first normal form when the attributes of all the entities are single valued.

Let's look at the entities we have so far.

Weather Station
Location
Manufacturer
Maintenance Record

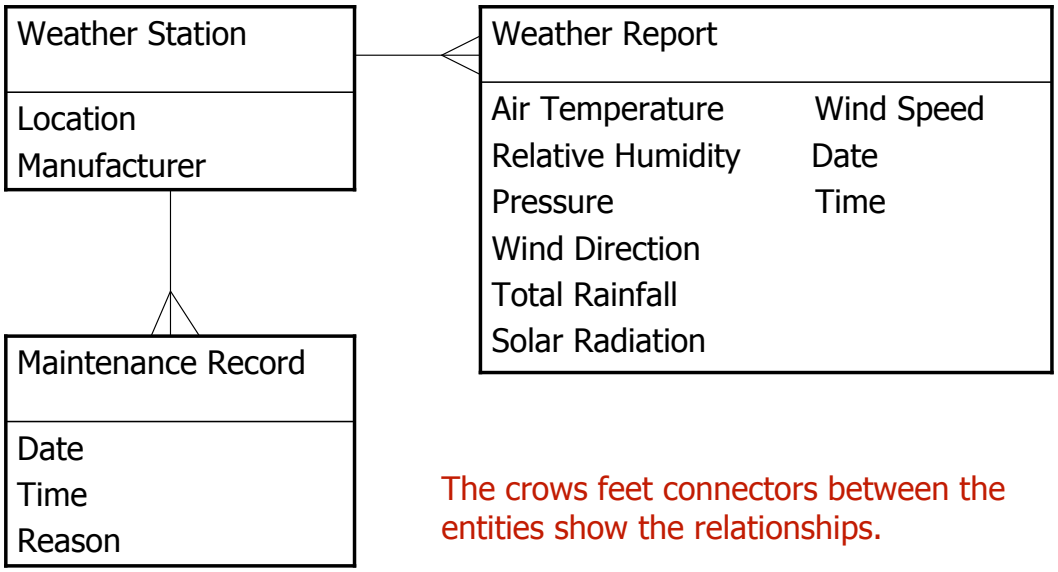
Weather Report	
Air Temperature	Wind Speed
Relative Humidity	Date
Pressure	Time
Wind Direction	
Total Rainfall	
Solar Radiation	

FIRST NORMAL FORM

Starting with weather station, we know that each station is only at one location and has only one manufacturer, so those attributes are single valued. The maintenance record, however, would be a log of all the times and reasons the station was serviced. This would have more than one value so it should be its own entity.

Consider weather report now. Each attribute of weather report would be single valued for a given report (we would not expect multiple values for a given variable, date or time for a single report). So that entity is already in first normal form. Let's look at what we have so far.

FIRST NORMAL FORM



The crows feet connectors between the entities show the relationships.

RELATIONSHIPS

There are three kinds of relationships between entities in a relational database: one-to-one, one-to-many, and many-to-many.

A one-to-one relationship is a direct link between two specific entities. Suppose that we considered location an entity and that there was only one station at each location. In that case, station and location would have a one-to-one relationship (one station for each location).

One-to-many relationships links many of one kind of entity to a single kind of another entity. For example, stations and maintenance records have a one-to-many relationship. A single station can have many maintenance records.

A many-to-many relationship, as you might guess, links many of one kind of entity to many of another kind. We will see these in the assignment.

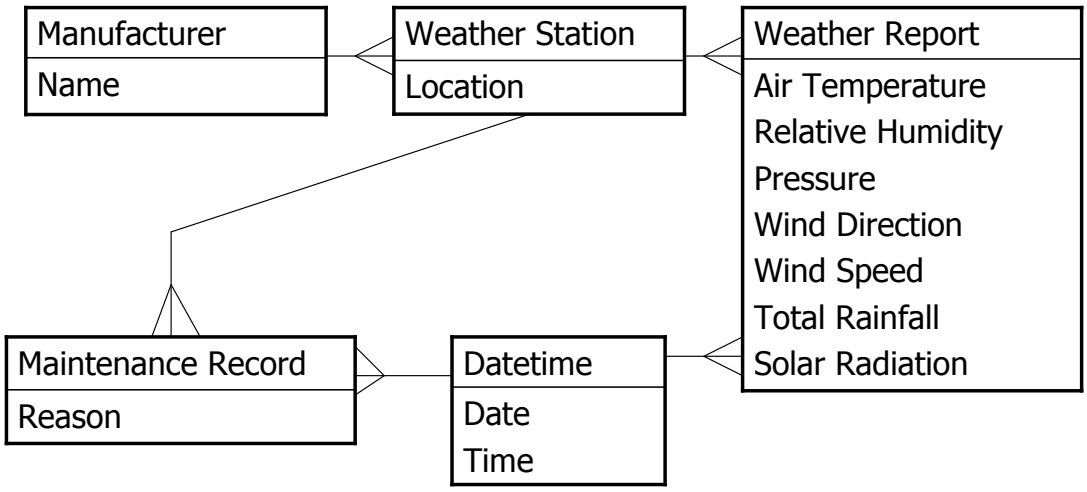
SECOND NORMAL FORM

A database is in second normal form when it is already in first normal form and all non-identifying attributes of an entity are dependent on the entity's unique identifier.

What this means is that different entities should not have an attribute that shares the same values. For example, our weather station entity has a manufacturer attribute. However, we could have, and probably would have, different weather stations with the same manufacturer. So we should make manufacturer its own entity. By the same token, weather reports and maintenance records both have dates and times that could be shared between different reports and records.

SECOND NORMAL FORM

Let's make these changes.



THIRD NORMAL FORM

A database is in third normal form when it is already in second normal form and no non-identifying attributes of an entity are dependent on any other non-identifying attributes. Once a database structure is in third normal form, it is ready to be translated into code.

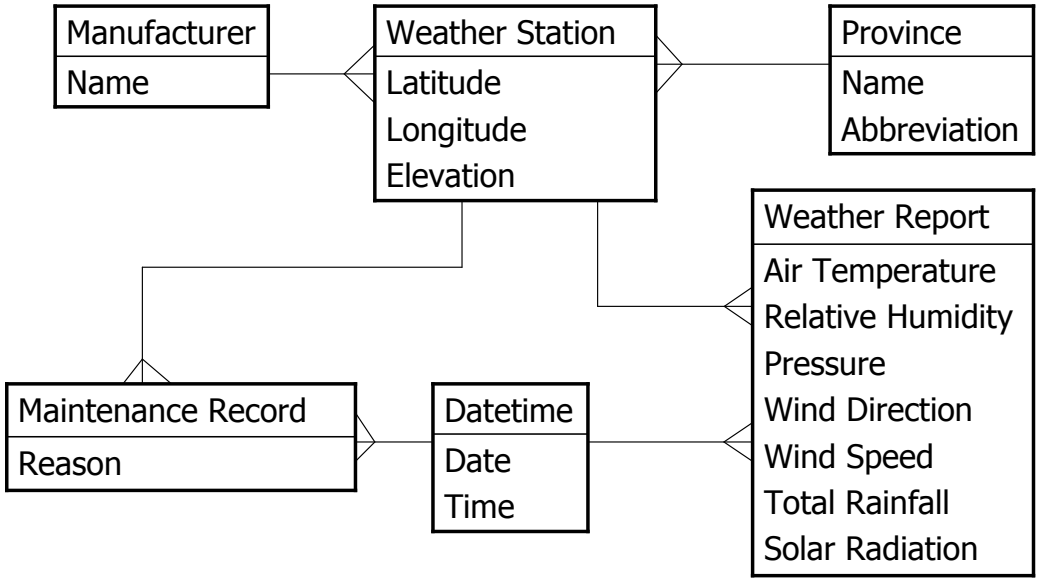
In the example, there are no attributes we listed that are dependent on each other, however, we can change one so that it is. Suppose that we break down the information in a station's location. It would have latitutde, longitude, elevation, province, and an abbreviation for province.

THIRD NORMAL FORM

If this were the case, we would have a problem with province and the abbreviation for province being in Weather Station. If we changed either, we would have to change the other. In this case, we would be best off removing province data from Weather Station and making it an entity.

Our final model would look like:

THIRD NORMAL FORM

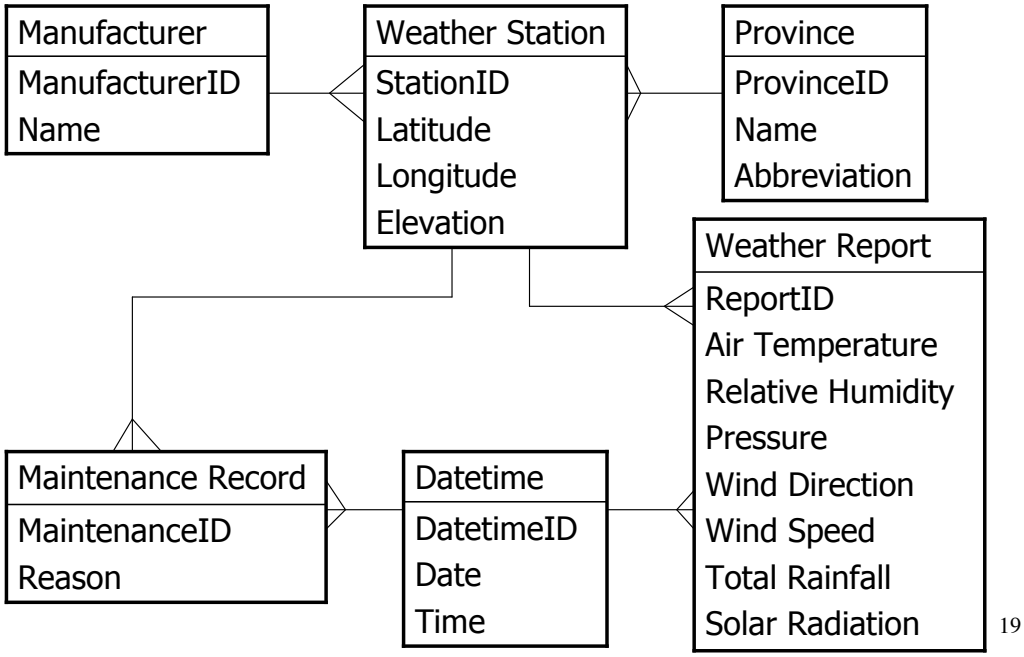


UNIQUE IDENTIFIERS

We now have a model for our database and the relationships between the different entities. However, to facilitate the creation of these relationships, we need to add something to the design. Each entity in the database needs to have its own unique identifier. For example, each of our weather stations needs a name or number that defines it as separate from every other station.

Unique identifiers are often a number internal to the database design (information outside users never see or care about). This makes sorting and searching the database faster, and ensures we never have to worry about duplicate names.

UNIQUE IDENTIFIERS



CODE TRANSLATION

Now that we have a design, we need to translate that into code to create the database. The first step is to recognize what the different elements of our design become in a coded database.

1. Entities become tables.
2. Attributes become columns in the tables. Each attribute will have a specific data type (ie string, integer, float)
3. Unique identifiers are columns that cannot contain NULL values and form the primary key for the table.
4. Relationships become foreign keys which are special attributes linking tables.

Once we make these translations, we will be ready for SQL.

CODE TRANSLATION

Going back to the example:

TABLE	COLUMN	TYPE	KEY
WeatherStation	StationID	Integer	Primary Key
	Latitude	Float	
	Longitude	Float	
	Elevation	Float	
	ProvinceID	Integer	Foreign Key
	ManufacturerID	Integer	Foreign Key
Manufacturer	ManufacturerID	Integer	Primary Key
	Name	String	
Province	ProvinceID	Integer	Primary Key
	Name	String	
	Abbreviation	String	

CODE TRANSLATION

TABLE	COLUMN	TYPE	KEY
Maintenance Record	MaintenanceID	Integer	Primary Key
	Reason	String	
	DatetimeID	Integer	Foreign Key
	StationID	Integer	Foreign Key
Datetime	DatetimeID	Integer	Primary Key
	Date	Integer	
	Time	Integer	
WeatherReport	ReportID	Integer	Primary Key
	…	…	…
	DatetimeID	Integer	Foreign Key
	StationID	Integer	Foreign Key

STRUCTURED QUERY LANGUAGE (SQL)

Now we have a series of tables and attributes to store all the data and relationships. However, we still have not actually written code to make the database work.

Nowadays, there are many different applications that will implement a database; MySQL, PostGres, Oracle, and Access to name a few. These applications merely provide the framework for creating and querying the database. (After all, the purpose of storing the data is getting back at it later). The language that most of these applications use to create and query a database is SQL.

STRUCTURED QUERY LANGUAGE (SQL)

The commands issued to a database are referred to as queries. SQL defines a set of rules and keywords for how to structure queries. Because the SQL is supported by many database applications, it is possible to implement a database on many different types of systems without changing the SQL code.

There are seven basic types of queries we will concern ourselves with: use, create, delete, drop, insert, update, and select.

On a side note, SQL is not case sensitive except with names. By convention, keywords are usually capitalized.

USE

To actually make use of a database, we need to first use it. This is the simplest form of query.

USE <name>;

All queries are terminated by a semicolon. The <name> is what the database is called. Names of databases and tables should not have any spaces in them. SQL recognizes spaces as a delimiter and will not understand names with spaces in them. Either remove spaces, or replace them with underscores in names.

CREATE

To create a database, and the tables that go inside it, we use the create query. To create a database;

CREATE DATABASE <name>;

To create our observation database, we would use the query:

CREATE DATABASE Observations;

CREATE

Creating tables within a database is a little trickier. First we must issue a use query to make sure we are using the database we want to create the tables in. Then we can issue table creation queries that look like:

CREATE TABLE <name> (<attribute> <type> [options], …);

<name> conforms to the same rules as for database names. <attribute> will be a column within the table, essentially the attributes we defined in our example earlier. <type> is what type of data that <attribute> is, like integer. [options] specify any additional limits on the attribute.

CREATE

SQL has many types, but the commonly used ones are INT for integer, FLOAT for real numbers, DATE for any year/month/day combinations, TIME for hour/minute/second combinations, CHAR for fixed length strings, VARCHAR for variable length strings, and ENUM for predefined datasets.

Like types, there are a great many options that can limit table data. The commonly used ones are:

AUTO_INCREMENT	automatically updates this field each time data is inserted by adding one to the previous value.
DEFAULT <value>	when data is inserted, if a value for this attribute is not specified, then it is set as <value>

CREATE

NOT NULL	NULL, or an empty data string, cannot be supplied for this attribute
NULL	NULL, or an empty data string, can be supplied for this attribute (this is the default behaviour of most databases)
PRIMARY KEY	specifies that this attribute is the primary key for the table
UNSIGNED	can be used with integer type to specify that the integer values are only positive (essentially doubles the maximum integer value that can be stored)
ZEROFILL	can be used with the integer type to pad out the integer field with zeroes (ie 132 would be stored as 00000132)

CREATE

Here is a query for creating our weather station table.

```
CREATE TABLE WeatherStation (
    StationID INT PRIMARY KEY AUTO_INCREMENT,
    Latitude FLOAT NOT NULL,
    Longitude FLOAT NOT NULL,
    Elevation FLOAT NOT NULL,
    ProvinceID INT,
    ManufacturerID INT
);
```

Notice how we did not specify NOT NULL for StationID. Primary keys cannot, by definition, be NULL so we do not need to specify it.

CREATE

One element of confusion is that there is no FOREIGN KEY flag for specifying attributes as foreign keys. That is because we do not have to. What is important is to have the attributes in the table that match primary keys in other tables. Looking at WeatherStation

```
CREATE TABLE WeatherStation (
    StationID INT PRIMARY KEY AUTO_INCREMENT,
    Latitude FLOAT NOT NULL,
    Longitude FLOAT NOT NULL,
    Elevation FLOAT NOT NULL,
    ProvinceID INT,          <--- THIS IS A FOREIGN KEY
    ManufacturerID INT       <--- THIS IS A FOREIGN KEY
);
```

More recent versions of MySQL have a way to enforce foreign key alignment between tables, but it is outside the scope of the course.

DELETE

To remove data from a database, we use DELETE.

```
DELETE FROM <table> [WHERE clause];
```

<table> is the name of the table we want to delete data from. The WHERE clause is optional. If we do not include a WHERE clause, then all the data in <table> is deleted. The WHERE clause allows us to restrict the deletion to only specified records. We will see more about WHERE clauses when we get to SELECT statements.

It is important to use the DELETE statement with caution. Unless the database is backed up, the data loss is permanent.

DROP

To remove a table from a database, or a database itself, we use DROP.

```
DROP TABLE <name>;
DROP DATABASE <name>;
```

The DROP TABLE query can only be used once a database is selected with USE. Tables and databases are permanently removed with this command unless the database has been backed up. For example, we could get rid of our weather station table with:

```
DROP TABLE WeatherStation;
```


INSERT

Of course, we need a way to put data into tables, and that is INSERT.

```
INSERT INTO <table> [ (<attribute>, …) ] VALUES (<value>, …);
```

<table> is the name of the table we want to put data into. After that, we can, optionally, include a list of attributes we want to insert data for. When we run an insert query, it creates a new record within <table>. If we do not want to fully specify the data within the record (either because some fields are auto increment, or NULL), we can list only the attributes we want to store data about. After VALUES, we list the data. If we do not specify the attributes we are storing for, then we must list enough values for all the attributes in the table.

INSERT

Here are some examples based on our WeatherStation table.

```
INSERT INTO WeatherStation VALUES (1, 48.5, -123.23, 155.0, 3, 10);
```

This would insert a record into WeatherStation with StationID = 1, Latitude = 48.5, Longitude = -123.23, Elevation = 155.0, ProvinceID = 3, and ManufacturerID = 10.

```
INSERT INTO WeatherStation (Latitude, Longitude, Elevation) VALUES (54,33, -119.77, 1205.2);
```

This would insert a record with StationID = 2 (since StationID is auto incremented), Latitude = 54.33, Longitude = -119.77, Elevation = 1205.2, ProvinceID = NULL, and ManufacturerID = NULL.

UPDATE

Sometimes we want to change the data in an existing record. To do this, we use the UPDATE query.

```
UPDATE <table> SET <attribute>=<value>, … [WHERE clause];
```

<table> is the name of the table where the record(s) reside. <attribute> is the column we want to change in the record(s) and <value> is the new value. We can change multiple columns by having a comma delimited list of <attribute>=<value>. The WHERE clause is optional and allows us to select the particular records we want to update. If the WHERE clause is not used, all records in the table are updated with the new value(s).

SELECT

Now we come to the most powerful, common, and complex query, SELECT. SELECT queries are how we get data from database tables.

The form of SELECT queries looks like this:

SELECT [DISTINCT] <attribute>[, <attribute> …] FROM <table>[, <table>…] [WHERE clause];

That probably looks pretty complex, so let's break it down. We start the query with SELECT. We can then, optionally, include the DISTINCT keyword. This tells the database not to return duplicates of the attribute that follows. By default, the database will normally return all data available.

SELECT

After that we have a list of attributes that we would like to have returned comma delimited. For example, we could query our WeatherStation table for just the Latitude and Longitude of stations. At least one attribute must be specified in a SELECT query. To specify all attributes in a table, rather than listing all attributes we can put * instead of the attribute list. Attributes can be specified just by name or by <table>.<name>, or even <database>.<table>.<name>.

After this comes the FROM keyword (to let the database know that we are not listing anymore attributes) followed by a comma delimited list of tables to draw the data from. If we are gathering data from related tables, we must list all tables we plan to use.

SELECT

Finally, we can include a WHERE clause. If we do not include this clause, the SELECT query will pull every record from all the tables listed (which on a sizeable database can be millions or billions). More than any other query, WHERE clauses are crucial to making useful SELECT queries.

WHERE clause

Simply put, WHERE clauses are a series of conditions. Only data which meets all the conditions will be returned.

Like conditionals from programming/scripting languages, the conditions of WHERE clauses are grouped together using AND and OR. To negate a condition, preface it with NOT. Numerical comparisons can be conducted with =, <, <=, >, >=, <>. Strings can be compared using LIKE and the wildcards % (which will match any group of characters) and _ (which will match any single character).

WHERE clause

Finally, results can be sorted or grouped by attributes using GROUP BY <attribute>[, <attribute>…] and ORDER BY <attribute>[, <attribute>…] [DESC].

GROUP BY acts like the DISTINCT keyword. It will return only the first record from any group of records that all match the same data on a given attribute (or list of attributes).

ORDER BY will sort the output by the attributes listed (each in order from the first attribute listed to the last). Normally, ORDER BY sorts into ascending order, however it is possible to make it sort into descending order by adding DESC to the end.

SELECT cont'd

Let's look at some examples.

```
SELECT DatetimeID FROM Datetime WHERE Date = "2006-03-12"
AND Time = "12:55:00";

SELECT * from WeatherReport WHERE DatetimeID = 12;

SELECT AirTemperature FROM WeatherReport, Datetime,
WeatherStation WHERE WeatherReport.DatetimeID =
Datetime.DatetimeID AND WeatherReport.StationID =
WeatherStation.StationID AND Date = "2007-02-12" AND Time =
"09:03:00" AND Latitude >= 48.5 AND Latitude < = 49.5 AND
Longitude >= -122.0 AND Longitude < -121.0;
```

SELECT cont'd

```
SELECT * FROM Manufacturer WHERE Name LIKE "%Vista";

SELECT * FROM WeatherStation ORDER BY StationID;

SELECT * FROM WeatherStation GROUP BY ManufacturerID;

SELECT Reason FROM MaintenanceRecord, WeatherStation,
Datetime WHERE MaintenanceRecord.DatetimeID =
Datetime.DatetimeID AND MaintenanceRecord.StationID =
WeatherStation.StationID AND Date = "2007-01-31" AND
WeatherStation.StationID = 122;
```


MySQL

Now that we have seen how to design a database and create the queries necessary to make databases, put data into them, and get data out, it is time to look at how this actually works with a production database engine.

MySQL is a free, enterprise level database engine. Until recent versions, MySQL was primarily used for small to midsize databases (hundreds to hundreds of thousands of records) with liberal transaction checking where speed was essential. However, versions 4 and 5 have incorporated more strict transaction locking capabilities, and shown improvements which allow for larger databases (millions of records). Security features have also improved allowing MySQL to become one of the most common engines.

MySQL

A database engine is an application that allows a user to create, modify, and query databases. MySQL is not a database itself.

There are two ways to utilize MySQL to run the queries we have learned. One way is through programming language interfaces. C, PERL, and Python all have special functions designed to allow a programmer to write a program to interact with a database. Because of the limited time we have spent on programming with C and PERL, these functions are outside the scope of this course, but you can check the MySQL reference in the course outline for more details.

MySQL

The other way is to invoke MySQL and input queries on the command line. To do this, type mysql -p on a terminal command line. This will attempt to log you into the MySQL server with your username. The -p flag indicates that you should supply a password to log in. For exercises, you will be given a password for your account.

Once you are logged in, MySQL will give you its own command line prompt >. You can type any of the queries we have learned on this line and MySQL will execute them. You can also type quit to leave MySQL.

MySQL

One last note, there are three queries specific to certain engines including MySQL that we have not discussed that give you information about database structures or systems that you will find useful.

SHOW databases;
SHOW tables;
DESC <table>;

SHOW databases; will list off all the databases the MySQL engine knows about (that you are allowed to see). This is a good way to find out what databases can be accessed.

MySQL

SHOW tables; will list off all the tables in the database you are using (remember the USE command).

DESC <table>; where <table> is a given table name, will show the columns of a table and what their types are, as well as any special information about them.

Tidbits

There are a few other helpful functions that can be used in queries to limit or change data that is returned. The three most useful to us are MAX, MIN, and COUNT. Each takes an attribute as a parameter and are part of the attribute list for a select query. For example:

```
SELECT MAX(StationID) FROM WeatherStation;
```

Would return the largest StationID from the WeatherStation table.

```
SELECT COUNT(DatetimeID) FROM Datetime WHERE Date =
20070302;
```

Would return the number of DatetimeIDs for which the date was March 2, 2007.

THE END?

This is as far as we will cover with databases. The *Managing & Using MySQL* text is a good reference if you want to get into serious database work and programming. You can also find additional information about databases online at www.wikipedia.com.