ATSC 212 - MySQL

ATSC 212 - MySQL

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, dimensional, object, and relational.

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 designed for efficient data analysis. Data is described by dimensions and measures. The user can then pick a number of dimensions (such as store and product) and quickly compute a measure (such as revenue).

ATSC 212 - MySQL

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. Records in relational databases are unique (you cannot have two copies of the same data in a table).

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 due to low data replication.

For the remainder of our sessions on databases, we will stick to relational database. There are two parts:

Part1: Designing databases
Part2: Using databases

PART 1

Designing databases

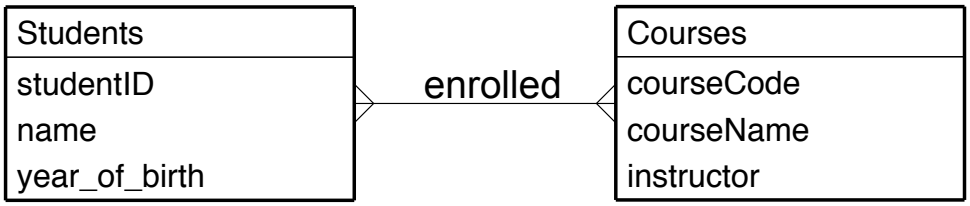
ATSC 212 - MySQL

RELATIONAL DATABASES

Relational databases consist of entities and relationships.

Entities describe data. For example, we could have a Students entity, which encapsulates all information about a student. The information is provided through attributes. For the Students entity we could have the attributes studentID, name, and year_of_birth.

Relationships capture how two entities relate to each other. For example the students entity could be related to the courses entity through the relationship enrolled.



ATSC 212 - MySQL

RELATIONAL DATABASES

Data in the database can be visualized as separate tables for each entity with attributes as columns and records as rows:

courseCode	courseName	instructor
ATSC201	Meteorology of storms	Stull
ATSC212	Introductory computing …	Stull
ATSC301	Remote sensing	Austin
ATSC303	Instrumentation	Novak

KEYS

A key is one or more attributes used to uniquely identify a record. studentID and courseCode are keys to the Students and Courses entities respectively. Two students cannot have the same studentID and two courses cannot have the same course code.

The entity books shown below needs both name and edition as its key because name alone will not uniquely identify a particular book.

Books
name
edition
number_sold

Some entities can have several different sets of attributes that can be keys. A primary key is the key we have selected to be the unique identifier for a table.

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.

What are the entities we need? At first glance, 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.

Weather stations report air temperature, relative humidity, pressure, wind speed, wind direction, total rainfall, and solar radiation once per hour.

Each weather station is described by StationID, Province, Province abbreviation, Manufacturer name, Manufacturer phone number and Maintenance record.

StationIDs are unique within each province (i.e. two stations can have the same StationID as long as they are in different provinces).

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.

Each step of the normalization process may create new entities and relationships in order to remove redundancy.

There are three stages of normalization that we will look at are: first normal form, second normal form, and third normal form.

Further normalization steps exists beyond the third normal form, but they tend to remove redundancy that only exists in very obscure entity relations.

FIRST NORMAL FORM

A database is in first normal form when the attributes of all the entities are single valued and each entity has a key.

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

Weather Station
StationID
Province
Province abbreviation
Manufacturer name
Manufacturer phone#
Maintenance Record

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

FIRST NORMAL FORM

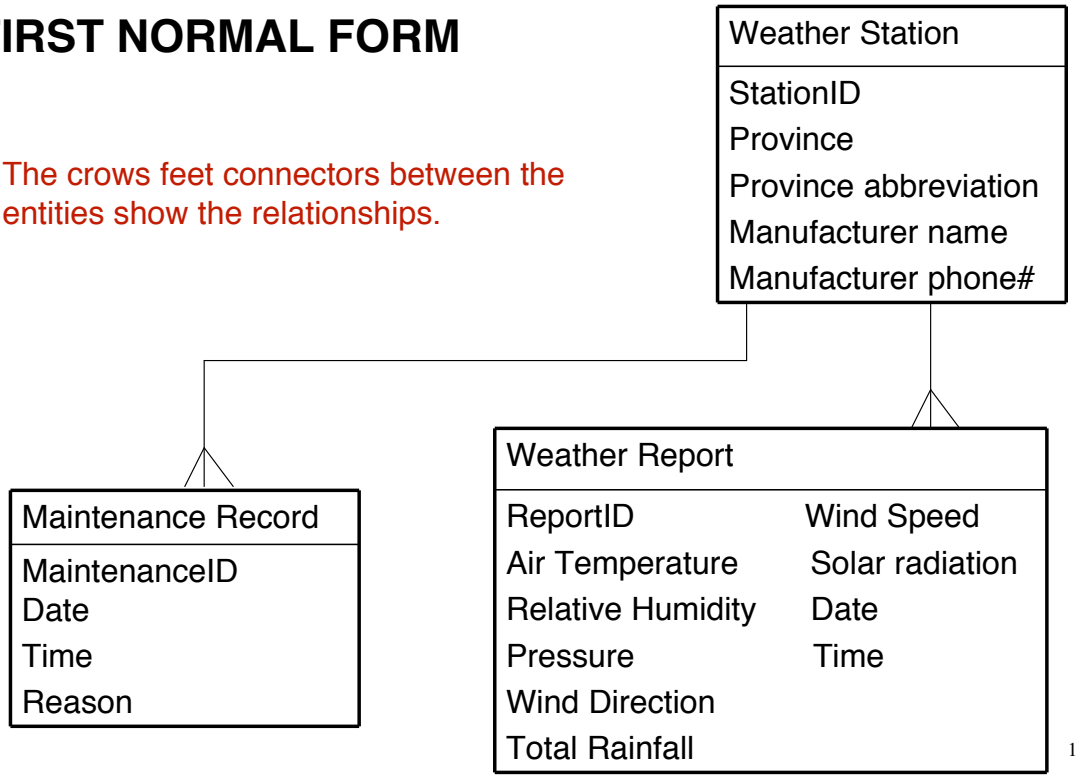
Starting with weather station, we know that each station is only in one province and has only one manufacturer, so those attributes are single valued. The maintenance record, however, would be a log of all the times and reasons the station was serviced. This would have more than one value so it should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

Weather report must also have a uniquely identifying key. We will add a ReportID attribute that will serve as the key.

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. Consider the entities driver and license. Each driver can only have one driver's license and each driver's license can only belong to one person. In that case, driver and license would have a one-to-one relationship. We will represent one-to-one relationships by a straight line.

driver ——— license

One-to-many relationships link 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.

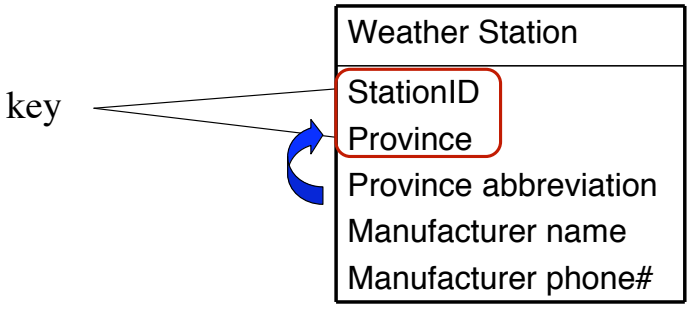
RELATIONSHIPS

We will represent one-to-many relationships with a straight line ending on the many side with 3 prongs.

Finally, a many-to-many relationship, as you might guess, links many of one kind of entity to many of another kind. You may see these in the assignment. We will represent these using 3 prongs on both ends of the line.

SECOND NORMAL FORM

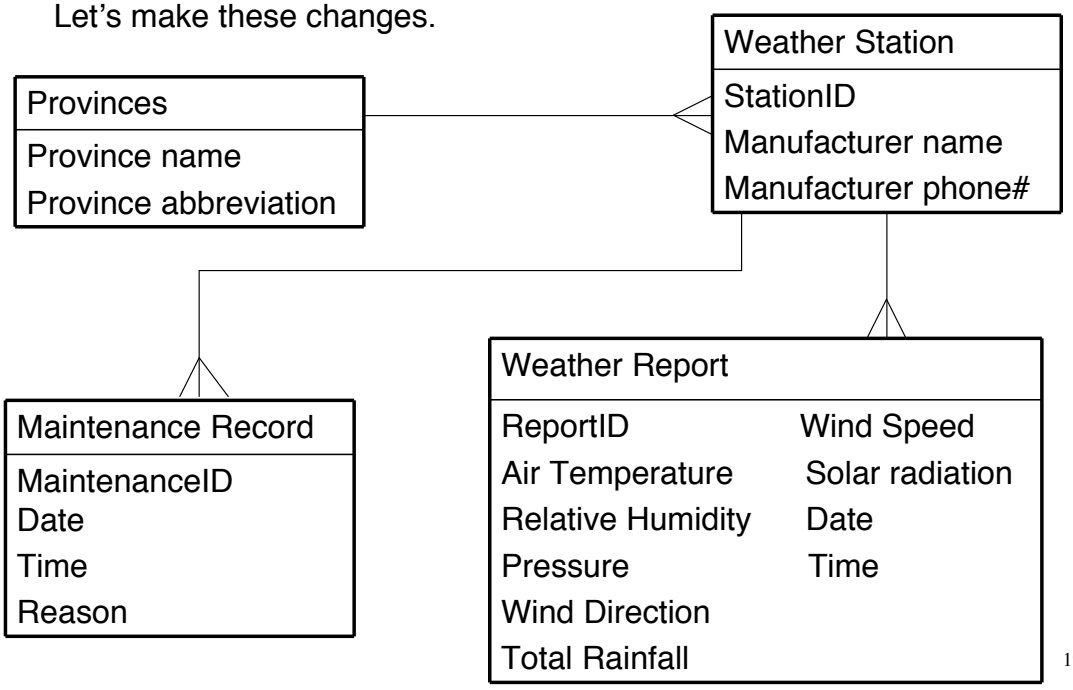
A database is in second normal form when it is already in first normal form and no non-identifying attributes of an entity are dependent on part of the entity's unique identifier.



Note that Province abbreviation is dependent on Province, but not on StationID. It is therefore only dependant on part of the key. This table is therefore not in second normal form. Manufacturer name depends fully on the entire key and Manufacturer phone# depends only on Manufacturer name. Therefore these attributes do not violate second normal form.

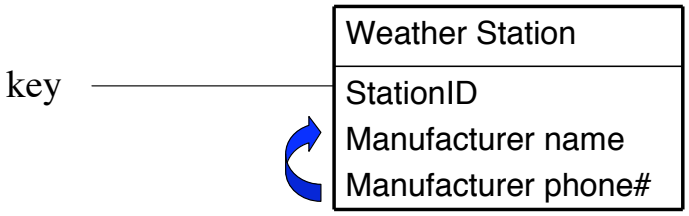
SECOND NORMAL FORM

Let's make these changes.



THIRD NORMAL FORM

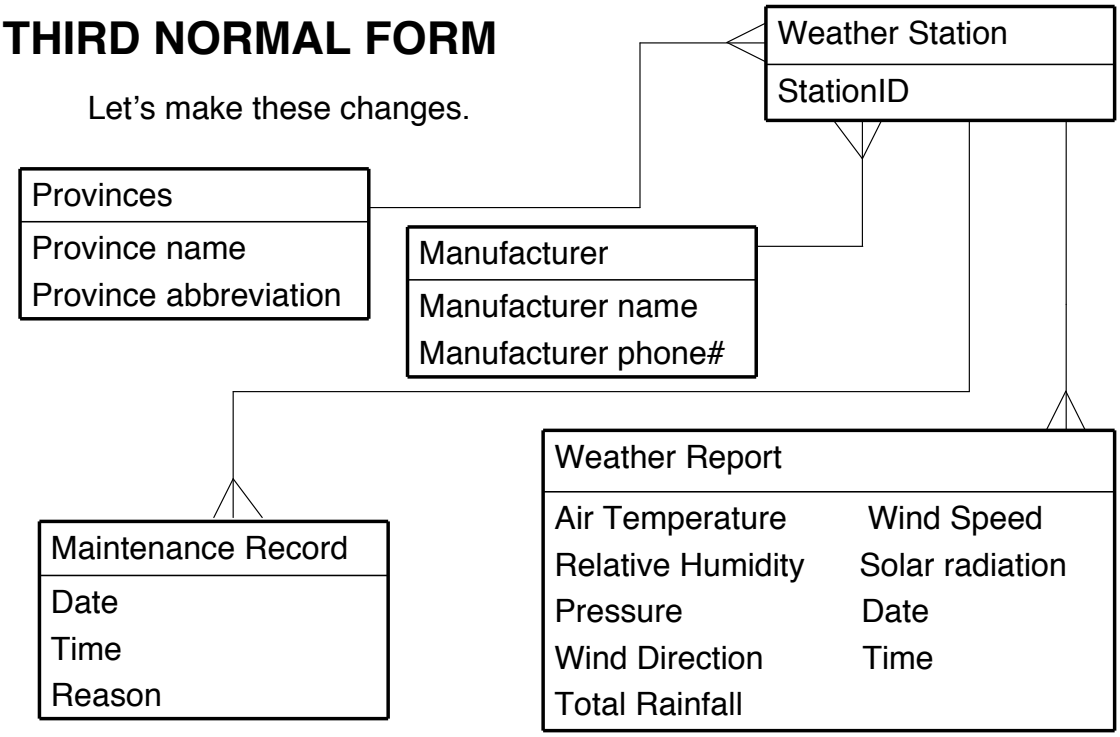
A database is in third normal form when it is already in second normal form and no non-identifying attributes of an entity are dependent on any other non-identifying attributes.



Note that Manufacturer phone# is dependent on Manufacturer name and must therefore be made its own entity.

THIRD NORMAL FORM

Let's make these changes.



UNIQUE IDENTIFIERS

We now have a model for our database and the relationships between the different entities. Every entity now has a key.

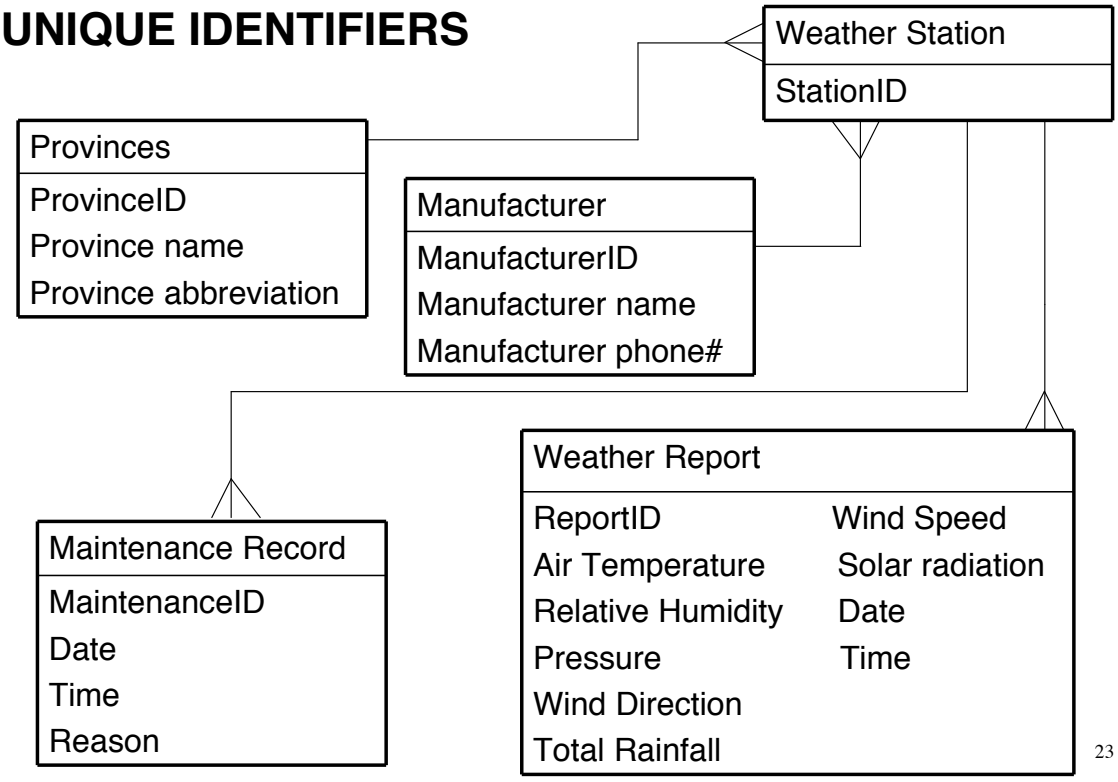
For tables that have primary keys involving strings it is often desirable to introduce a new attribute that will be used internally in the database. This is because it is faster for the database system to search for an integer than a string of characters.

For these tables we will create a new attribute ending in "ID" to serve as this internal number.

We will add ProvinceID to the Provinces entity and ManufacturerID to the Manufacturer entity.

ATSC 212 - MySQL

UNIQUE IDENTIFIERS



ATSC 212 - MySQL

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 (i.e. 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.

CODE TRANSLATION

How we translate relationships depends on the type of relationship.

One-to-one:

In this case the two entities can be merged into one table.

Drivers and licenses	
Driver name	license number
Driver SIN	

One-to-many:

Here we will add a foreign key attribute to the entity on the "Many" side. Note that this design disallows a station to be in more than one province (which is what we want).

Provinces
ProvinceID (primary key)
Province name
Province abbreviation

Weather Station
StationID (primary key)
ProvinceID (foreign key)

CODE TRANSLATION

Many-to-many:

This relationship requires us to create a brand new table that encapsulates the relationship.

Students
studentID (primary key)
name
year_of_birth

Courses
courseCode (primary key)
courseName
instructor

Enrolled
enrolledID (primary key)
studentID (foreign key)
courseCode (foreign key)

ATSC 212 - MySQL

CODE TRANSLATION

Going back to the weather data example, we can summarize the tables, attributes, and relationships we need as follows:

TABLE	COLUMN	TYPE	KEY
WeatherStation	StationID ProvinceID ManufacturerID	Integer Integer Integer	Primary Key Foreign Key Foreign Key
Manufacturer	ManufacturerID Name Phone number	Integer String String	Primary Key
Province	ProvinceID Name Abbreviation	Integer String String	Primary Key

ATSC 212 - MySQL

CODE TRANSLATION

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

PART 2

Using databases

ATSC 212 - MySQL

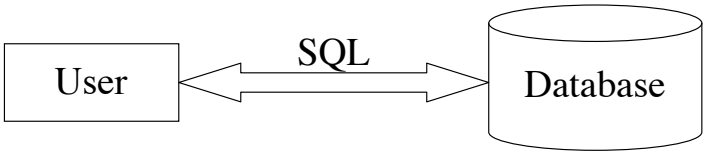
MySQL

A database engine is an application that allows a user to create, modify, and query databases.

MySQL is a free, enterprise level database engine. MySQL is not a database itself. 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.

STRUCTURED QUERY LANGUAGE (SQL)

- SQL is a set of commands used to interact with databases
- These commands are called Queries



- Some examples of databases that support SQL: MySQL, Oracle, Microsoft Access
- SQL is portable, therefore you do not need to change it when moving to a different database system.

STRUCTURED QUERY LANGUAGE (SQL)

- There are seven basic types of queries we will look at: USE, CREATE, DELETE, DROP, INSERT, UPDATE, and SELECT.

- SQL is not case sensitive except with names. By convention, keywords are usually capitalized.

MySQL - Executing SQL queries

There are two ways to utilize MySQL to run queries. 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 - Executing SQL queries

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 will learn on this line and MySQL will execute them. You can also type quit to leave MySQL.

MySQL - Showing available databases

Before we look at the types of queries we can use, it helps to know what databases and tables are currently available in our database system. The following three queries are specific to certain engines including MySQL that give you information about database structures 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.

We will look at SHOW tables and DESC later.

USE

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

USE <name>;

A quick side note about syntax; 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.

MySQL - Showing table information

Now that we have selected a database to use, we can find out what tables are in the database using SHOW tables;.

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.

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 a Students database, we would use the query:

CREATE DATABASE Students;

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 Studnts table.

```
CREATE TABLE Students (
    StudentID INT PRIMARY KEY AUTO_INCREMENT,
    Name varchar(40) NOT NULL,
    FavColour varchar(40) NULL DEFAULT "Blue",
    CountryID INT
);
```

Notice how we did not specify NOT NULL for StudentID. 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 Students

```
CREATE TABLE Students (
    StudentID INT PRIMARY KEY AUTO_INCREMENT,
    Name varchar(40) NOT NULL,
    FavColour varchar(40) NULL DEFAULT "Blue",
    CountryID INT        <--- We don't specify 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 Students table with:

DROP TABLE Students;

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 Students table.

INSERT INTO Students VALUES (1, "Thomas", "Green", 3);

This would insert a record into Students with StudentID = 1, Name = “Thomas”, FavColour = “Green”, and CountryID = 3.

INSERT INTO Students (Name, CountryID) VALUES ("George", 2);

This would insert a record with StudentID = 2 (since StudentID is auto incremented), Name = “George”, FavColour=“Blue”, CountryID = 2.

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).

ATSC 212 - MySQL

Meta and obs database

To get some experience with real data, we have uploaded a simplified version of our observations and meta database.

Meta: Contains Cities, Stations, and Spacetimes tables
Obs: Stores Air_Temperature, Wind_Speed, etc.

Let's switch to the meta database for now.

```
USE meta;
```

Take a look at what this database contains:

```
SHOW tables;
```


ATSC 212 - MySQL

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 the Stations 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 any more 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).

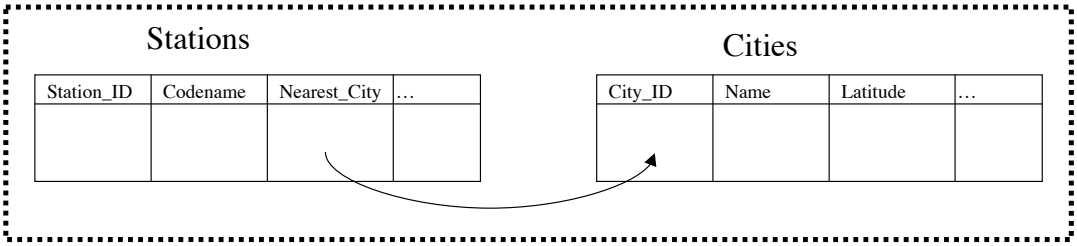
Examples:

```
SELECT * FROM Cities WHERE Name LIKE "Van%";
SELECT * FROM Stations WHERE Elevation>2200 AND Country="USA";
```


WHERE clause

We can also use the WHERE clause to join related tables together.

For example, the Stations table has a Nearest_City attribute which relates to a City_ID in the Cities table.



If we include both Stations and Cities in the FROM clause of the SELECT statement, we must link the two in the WHERE clause as follows:

```
SELECT * FROM Cities, Stations WHERE City_ID=Nearest_City;
```

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).

SELECT * FROM Stations GROUP BY Providing_Agency_ID;

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 * FROM Cities ORDER BY Latitude_DMS;

WHERE clause - Duplicate names

Sometimes when joining two tables together we end up with two fields with the same name. For example both Stations and Spacetimes have the field Station_ID. We can prefix each variable name with the table name to specify which variable we are referring to:

Stations.Station_ID and Spacetimes.Station_ID

SELECT - Examples

Let's look at some more examples.

```
SELECT * FROM Stations WHERE Sea_Land_Mask="water";

SELECT * FROM Stations, Spacetimes
        WHERE Stations.Station_ID=Spacetimes.Station_ID
            AND Date="2007-01-31"
            AND Time="22:30:00"
            AND Province="YT";
```


SELECT - Examples

1) Find all inactive stations.

2) Show the elevation and station_ID of all stations in Ontario, sorted by elevation.

3) Find all stations whose nearest city is Vancouver.

4) Find all stations whose nearest city is Vancouver, WA.

5) Find the temperature at station 2628 for all available times.

ATSC 212 - MySQL

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(Station_ID) FROM Stations;
```

would return the largest Station_ID from the Stations table.

```
SELECT COUNT(Spacetime) FROM Spacetimes WHERE Date = 20070131;
```

would return the number of Spacetimes for which the date was January 31, 2007.

ATSC 212 - MySQL

SELECT - Examples

1) Find the highest elevation.

2) Find the number of stations in each province (hint: GROUP BY)

3) For each city show how many stations are closest to that city

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.