

CT216 Software Engineering Tutorial

Eoin O Fiachain

21st/28th October 2004

1 Introduction to Relational Databases

1.1 Databases and Database Management Systems

A *database* is an organized collection of information or data with regular structure. It is organized so that its contents can easily be accessed, managed, and updated.

A *database management system* (DBMS) is a suite of computer programs designed to manage a database. It sits between the users of a database and the physical database, controlling all access to the database.

Databases present a logical view of the database structure to the user known as a *programming model*. Several programming models are commonly associated with databases including the *hierarchical model*, *network model*, *relational model* and newer object-oriented models.

1.1.1 Relational Model

The *relational model* is a popular database model used in many modern databases.

The model is derived from branches of mathematics known as set theory and predicate logic. All data is stored in the form of mathematical relations.

In practice, what this means is that data is stored in a tabular form involving relations/tables, columns and rows.

A *relation* or *table* is a named two-dimensional table of data. Each relation consists of a set of named columns and an arbitrary number of named rows.

A *relational database* is a database that employs the *relational model*.

1.1.2 Relational DBMS

A Relational Database Management System (RDBMS) is a DBMS that is based on the relational model.

Popular RDBMSs include Oracle, Microsoft SQL Server, Microsoft Access, Sybase SQL Server, IBM's DB2, PostgreSQL and MySQL.

1.2 SQL

Structured Query Language or *SQL* is a standard computer language for communicating with relational databases. It is used to create, modify and query databases.

Most industrial-strength and many smaller database applications can be addressed using SQL. Each specific application will have its own version of SQL implementing features unique to that application, but all SQL-capable databases support a common subset of SQL.

We will focus on the MySQL dialect of SQL which is very similar to standardised SQL. Other popular dialects include Oracle's PL/SQL and Microsoft's Transact-SQL.

SQL differs from standard programming languages in that it is a *declarative language*. You specify "what kind of data" you want from the database rather than "how to retrieve" data. The RDBMS then determines how to retrieve the data efficiently.

The SQL language is divided into a number of different elements:

- **Data Manipulation Language (DML)** is a subset of SQL language used to query a database, add, update and delete data.
- **Data Definition Language (DDL)** allows the user to define new tables and associated elements.
- **Data Control Language (DCL)** allows the user to control access to the data and to the database.

1.3 Database Schema

A *database schema* is a model of a database illustrating the overall structure of the database. It is a definitive reference for a particular database indicating the structure for tables, attributes etc.

A database schema is generally represented as a file containing the SQL commands necessary in order to create the database. A schema is often also represented in the form of a Entity-Relationship Diagram.

1.4 MySQL

MySQL is the most popular open-source SQL-enabled RDBMS. It is freely available and its fast and reliable.

It does not support as wide a range of features as large enterprise databases such as Oracle or Microsoft SQL Server, but it is sufficient for many applications. It is often used in conjunction with PHP for server-side web development.

1.5 Client-Server Model

Normally MySQL operates in client-server mode. This is similar to the client-server paradigm we looked at with HTTP in the first tutorial.

A MySQL server program runs on a computer, operating on commands it receives from client programs. This server program controls all access to the underlying database files.

Various client programs connect to the server and execute SQL commands to interact with the database. The server can regulate access by clients based on both the network location they connect from and with username/password combinations.

There are many different types of programs that can act as clients to the MySQL server program (*mysqld.exe*). These can be specialised applications or custom-made programs.

In the next tutorial we will learn how to write PHP server-side scripts which act as MySQL clients. The scripts will connect to the MySQL server and retrieve data for the web-page output. This allows us to create web-pages whose content is dynamically generated from a database.

1.6 Running MySQL

1.6.1 Starting the MySQL Server

After MySQL is installed, the MySQL server application can be started using the *winmysqladmin.exe* program in the `bin` directory of the MySQL distribution.

This can be started through *Windows Explorer* or through the *Command Prompt*.

The server application must be running in order for a client to connect.

1.6.2 Running the MySQL Client

A special command-based client application (*mysql.exe*) is bundled with the MySQL server. This allows the developer to interact with the MySQL server to execute queries and commands.

This client application is designed to be executed from the *Command Prompt*. This is a command-line environment which allows you to execute DOS commands.

After starting the *Command Prompt* from *Start Menu - Programs - Accessories* a command sequence similar to the following can be used to start this MySQL client application:

```
cd c:\mysql\bin
mysql -u username -p databasename -h hostname
```

Replace *username* with the desired username.

Replace *databasename* with the desired databasename.

Replace *hostname* with the hostname/IP Address of the computer which the MySQL server is running on. If this is omitted then the client will automatically connect to the local PC (*localhost*).

2 Designing An Example Database

In order to demonstrate MySQL we will focus on a particular example of a small database about films. We will use this example database as a means of introducing various features and aspects of MySQL.

In the *Software Engineering* lectures Entity-Relationship diagrams are introduced as a data model compiled during the *analysis phase* of a project.

Database Modeling is outside the scope of this tutorial but a comprehensive tutorial is provided at:

<http://www.utexas.edu/its/windows/database/datamodeling/>

Considering our example scenario, we arbitrarily identify three entities that we wish to model: *film*, *actor* and *genre*. We then chose various *attributes* (properties associated with each entity) that we wish to consider.

Film	Name Year of Release
Actor	Last Name First Name Birth Date
Genre	Name Description

2.1 Choosing Primary Keys

Each entity should have a *primary key*. This is an attribute or set of attributes that uniquely identify each entity.

For the *genre* entity we select the *name* attribute as a primary key as it uniquely identifies each genre e.g. thriller, crime, comedy etc.

Although we could use the *name* attribute as a primary key for the *film* entity, instead we create a new artificial identifier *id* as two films may possibly have the same name.

Similarly we add an *id* attribute for the *actor* entity as a combination of first and last names may not necessarily be unique.

Using artificial id values also has the benefit of providing for easier references to particular films. Instead of using the entire title to refer to a film we can

simply use a much more compact id number. On a physical level this provides for less complex database references and decreased storage space.

2.2 Constructing an Entity-Relationship Diagram

We then illustrate our entities and attributes on an Entity-Relationship Diagram (ERD):

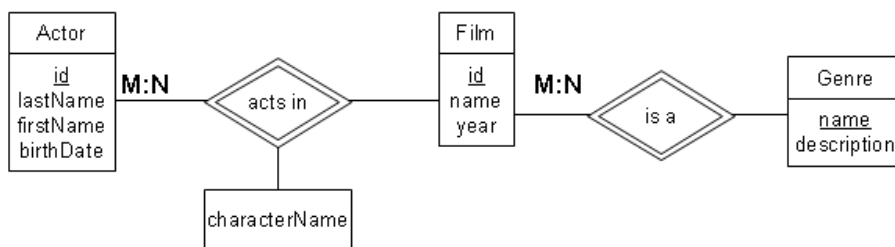


Figure 1: Initial ERD for Films Database

Our *entities* correspond to the boxes in the diagram.

The *relations* between our entities are indicated through the connections between entities.

The *attributes* of an entity are indicated in the bottom section of each entity.

Note that each attribute that forms part of a primary key is underlined.

2.2.1 Resolving Many-to-Many Links

Each film can be associated with many actors. Each actor can be associated with many films. Therefore we have a *many-to-many* relationship between the two entities.

Similarly, each film can be associated with many genres. Each genre is associated with many films. Therefore we also have a *many-to-many* relationship between the *film* and *genre* entities.

We cannot represent many-to-many relationships in a relational database. We resolve this problem by introducing an extra *association entity* into the diagram with one-to-many relationships to replace the existing many-to-many relationship.

We also add extra attributes to these *association entities* to allow us reference the primary keys of the original entities involved in the many-to-many relationship. These are called *foreign keys*.

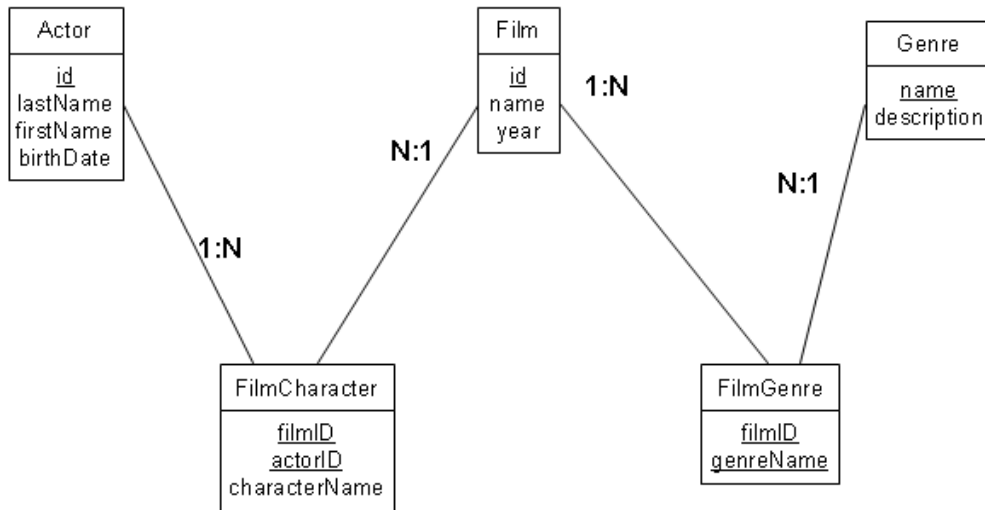


Figure 2: ERD after resolving many-to-many links

2.3 Normalization

Typically in database design, a *normalization* procedure would now occur. This is a method of cutting-down and simplifying data structures so as to remove repeated and unnecessary data.

There are defined processes for normalizing entities which involve the conversion of entities through 5 different stages from *1st Normal Form* to *5th Normal Form*.

These processes are largely beyond the scope of this tutorial but further information on database normalization is available in “An Introduction to Database Normalization” at <http://www.devshed.com/c/a/MySQL/>.

2.4 Converting ERD to Relations

In order to progress from the *analysis* to the *design* phase in a project, we must transform our initial Entity-Relationships model into a relational database structure suitable for our DBMS (MySQL in this case).

In order to convert an ER diagram to a set of relations we replace each entity in the ER diagram with a *table*. The attributes of the entity become the table columns.

2.4.1 Assigning of Data Types to Columns

Each column must have an associated data type (integer, characters etc.). All data items in a column must be of this data type.

We assign an appropriate data type to each column in our tables. There are a variety of data types available MySQL that we will examine in due course.

2.4.2 ERD View of MySQL Database Schema

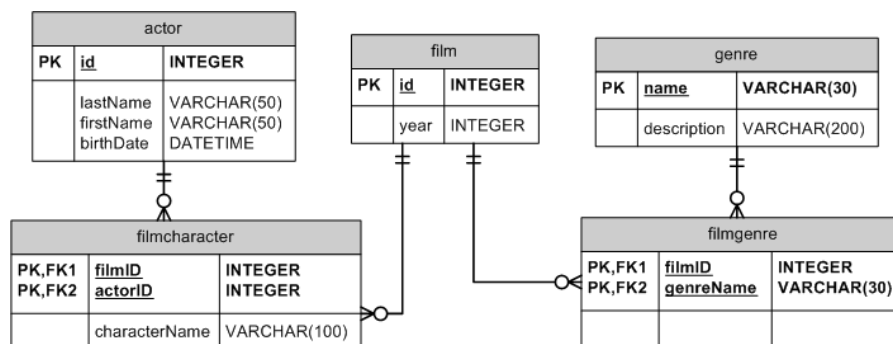


Figure 3: MySQL Entity-Relationships Diagram for Films Database

The above ERD diagram is similar but slightly different to those we created during analysis. It provides a “relational database” view of tables/columns rather than a “modeling view” of entities/attributes.

Note the *crow's feet* style used for describing relationship multiplicities. This corresponds directly to our $1:M$, $M:N$, etc. style used in previous ERDs.

The above ERD represents our final *database schema* for our films database.

3 Structured Query Language for MySQL

SQL (Structured Query Language) is composed of declarative statements. These indicate either what data is required from the server or what action the server should take.

The semi-colon ; is used to separate multiple statements.

SQL keywords and table names are not case-sensitive (e.g. *SELECT* is the same as *select*). However character literals are case-sensitive (e.g. 'DATA' is treated differently to 'data').

3.1 Creating and Dropping Databases

Each MySQL server can host multiple databases. Each database contains its own tables. Each table can contain its own columns and rows.

Tables are grouped together with related tables within a single database. Tables that have relationships with each other should be grouped together within a single database as MySQL doesn't allow cross-database queries.

Most applications will store all their data in a single database.

The *CREATE DATABASE* and *DROP DATABASE* statements can be used to create and delete a database respectively.

```
CREATE DATABASE films;      -- Creates 'films' database
DROP DATABASE films;       -- Deletes 'films' database
```

N.B. Comments in SQL occur after a space and then two hyphens.

Any connection to the MySQL server can only be associated with one database simultaneously. The *USE* statement is employed to activate a particular database.

```
USE films;      -- Switches to 'films' database
```

The *SHOW DATABASES* statement returns a list of all the databases on a particular server.

```
SHOW DATABASES;      -- Returns list of all databases
```

3.2 Creating and Dropping Tables

The *CREATE TABLE* statement allows a table to be created.

The *DROP TABLE* statement deletes a table.

The *SHOW TABLES* statement lists all available tables.

The *DESCRIBE* statement lists the columns for a particular table.

The syntax of these statements is:

```
CREATE TABLE table_name (  
    column_name column_type [column_attributes],  
    ...,  
    PRIMARY KEY( column_name, ... )  
);  
  
DROP TABLE table_name;  
  
SHOW TABLES;  
  
DESCRIBE table_name;
```

For example, a statement to create the *film* and *filmcharacter* tables from our example database would be as follows:

```
CREATE TABLE film (  
    id      INT NOT NULL AUTO_INCREMENT,  
    name   VARCHAR(100) NOT NULL,  
    year   INT,  
    PRIMARY KEY(id)  
);  
  
CREATE TABLE filmcharacter (  
    filmID   INT NOT NULL,  
    actorID  INT NOT NULL,  
    characterName varchar(100),  
    PRIMARY KEY(filmID, actorID)  
);
```

This statement would create a table *film* with 3 columns *id* of type INT, *name* of type VARCHAR(100) and *year* of type INT.

The *filmcharacter* table is created similarly.

The *PRIMARY KEY* declaration can be used to define which key or keys form the primary key. Although it is technically optional, all tables should have an associated primary key in a well-designed database.

3.2.1 Column Types

MySQL supports a wide variety of column types.

Some of the numerical types include:

Data Type	Description	Signed Range
BIGINT	A large integer	-9223372036854775808 to 9223372036854775807
INT or INTEGER	A normal-size integer	-2147483648 to 2147483647
MEDIUMINT	A medium-size integer	-8388608 to 8388607
SMALLINT	A small integer	-32768 to 32767
TINYINT or BIT	A very-small integer	-128 to 127
FLOAT(p)	A floating-point number	Precision $p = 0$ to 53

The *UNSIGNED* keyword can be appended to restrict the range of a type to positive numbers (e.g. *MEDIUMINT UNSIGNED*). In most cases this will also double the range of the numbers by doubling the maximum possible size.

Character types include:

Data Type	Description
CHAR(m)	A fixed-length string of length m (0-255 chars.)
VARCHAR(m)	A variable-length string of length m (0-255 chars.)
TEXT	A text column with maximum length of 65,535 chars.

Date/Time types include:

Data Type	Description
DATE	Date with range '1000-01-01' to '9999-12-31'
TIME	Time with range '-838:59:59' to '838:59:59'
DATETIME	Date and Time Combination

Note that character and date/time values are always expressed within single-quotes e.g. 'Galway City' or '6666'. A series of escape sequences can be used to represent special characters in a similar manner to C and PHP including single-quote (\'), newline (\n), percentage-sign (\%) etc.

Numerical values are expressed without quotes.

3.2.2 Column Attributes

Optional *attributes* follow the column type. These further define or restrict the definition of the column.

Some common attributes include:

- **NOT NULL** - disallows NULL values in a column.
- **NULL** - allows NULL values in a column (this occurs by default).
- **UNIQUE** - requires each value in the column to be unique.
- **AUTO_INCREMENT** - Replaces a NULL or 0 value automatically with an incrementing integer (useful for primary keys).

NULL is a special value representing 'no value'.

3.3 Inserting, Updating and Deleting

3.3.1 INSERT statement

The INSERT statement allows the user to add new rows to a table:

```
INSERT INTO table_name
  ( first_column_name, second_column_name, ... )
  VALUES ( first_column_value, second_column_value, ... );
```

An example statement to insert a new 'horror' genre into our *genre* table would be:

```
INSERT INTO genre (name, description)
  VALUES('horror', 'Horror Movies');
```

If you are inserting a value for every column in the table then the column list can be omitted.

```
INSERT INTO genre VALUES('horror', 'Horror Movies');
```

3.3.2 DELETE statement

The DELETE statement allows the user to delete a row or group of rows from a table.

An example statement follows that deletes all rows from the genre table where the *name* value is equal to 'horror':

```
DELETE FROM genre WHERE name='horror';
```

This statement deletes all films that were released before 2000:

```
DELETE FROM film WHERE year < 2000;
```

If the WHERE keyword and condition is omitted then all rows in a table are deleted:

```
DELETE FROM film;
```

3.3.3 UPDATE statement

The UPDATE statement allows the user to update values in an existing row or group of rows.

```
-- Sets a new name for any rows in the 'actor' table that  
-- have a name equal to 'Arnold Schwarzenegger'  
UPDATE actor SET name='Governor Arnie'  
WHERE name='Arnold Schwarzenegger';
```

Multiple columns can be updated simultaneously:

```
UPDATE film SET name='Reservoir Dogs', year=1992 WHERE id=2;
```

3.3.4 WHERE Conditions

WHERE clauses are frequently used in many SQL statements as a means of only selecting particular rows that match specified conditions.

Multiple conditions can be imposed by using the *AND* and *OR* logical operators to connect conditions. Brackets can be used to group conditions and to establish an order of precedence.

```
DELETE FROM actor
  WHERE (firstName='Tom' AND lastName='Hanks')
        OR (firstName='Meg' AND lastName='Ryan');
```

By using all of the primary key columns in a condition, a single row can be uniquely specified.

Some common comparison operators are:

Operator	Description
=	Equal
<> or !=	Not Equal
<=	Less Than or Equal
<	Less Than
=>	Greater Than or Equal
>	Greater Than
IS NULL	Whether a value is NULL
IS NOT NULL	Whether a value is not NULL

The *LIKE* operator can be used to see if a string value matches a particular string expression.

The % character is used as a wildcard to match 0 or more characters. The _ character is used as a wildcard to match exactly 1 character.

The following example deletes any rows in the *films* table whose *name* value begins with the string 'Star':

```
DELETE FROM film WHERE name LIKE 'Star%';
```

3.4 Querying Tables

Querying tables is the most common task performed using SQL. The declarative nature of SQL provides for a powerful and versatile means of selecting precisely what data we want to retrieve from a table or group of tables.

For the purposes of demonstration, let us assume that our example database has been populated with data as follows:

films table

id	name	year
1	Big Lebowski, The	1998
2	Reservoir Dogs	1992
3	Pulp Fiction	1994
4	Kill Bill, Volume 1	2003

actor table

id	lastName	firstName	birthDate
1	Bridges	Jeff	04/12/1949
2	Goodman	John	20/06/1952
3	Buscemi	Steve	13/12/1957
4	Keitel	Harvey	13/05/1939
5	Roth	Tim	14/05/1961
6	Thurman	Una	29/04/1970
7	Madsen	Michael	25/09/1958

genre table

name	description
action	Action film
comedy	Comedy Film
crime	Crime film
drama	Drama film
mystery	Mystery film
thriller	Thriller film

film character

filmID	actorID	characterName
0	0	Budd
1	1	The Dude
1	2	Walter
1	3	Donny
2	3	Mr. Pink
2	4	Mr. White
2	5	Mr. Orange
2	7	Mr. Blonde
3	4	The Wolf
3	5	Pumpkin
3	6	Mia Wallace
4	6	The Bride
4	7	Budd

filmgenre table

filmID	genreName
1	comedy
1	mystery
1	thriller
2	action
2	crime
2	mystery
2	thriller
3	crime
3	drama
4	action
4	crime
4	thriller

3.4.1 SELECT Command

The SELECT command is used to retrieve rows from one or more tables.

The following example retrieves the *name* column and the *year* column for all rows in the *film* table with a *year* of 1995 or above:

```
SELECT name, year FROM film WHERE year >= 1995;
```

```
+-----+-----+
| name                | year |
+-----+-----+
| Big Lebowski, The  | 1998 |
| Kill Bill, Volume 1 | 2003 |
+-----+-----+
```

If you wish to view all columns in a table in a query, you can replace the column list with the *** character:

```
SELECT * FROM film WHERE year >= 1995;
```

```
+----+-----+-----+
| id | name                | year |
+----+-----+-----+
|  1 | Big Lebowski, The  | 1998 |
|  4 | Kill Bill, Volume 1 | 2003 |
+----+-----+-----+
```

If no WHERE clause is specified then all rows are retrieved from a table:

```
SELECT * FROM film;
```

```
+----+-----+-----+
| id | name                | year |
+----+-----+-----+
|  1 | Big Lebowski, The  | 1998 |
|  2 | Reservoir Dogs     | 1992 |
|  3 | Pulp Fiction       | 1994 |
|  4 | Kill Bill, Volume 1 | 2003 |
+----+-----+-----+
```

3.4.2 ORDER BY

The ORDER BY clause can be used to sort the retrieved data.

The following query is sorted firstly by the *lastName* column and secondly by the *firstName* column. If two *lastName* values are the same, the *firstName* values are then used for a further comparison.

```
SELECT * FROM actor ORDER BY lastName, firstName;
```

id	lastName	firstName	birthDate
1	Bridges	Jeff	1949-12-04 00:00:00
3	Buscemi	Steve	1957-12-13 00:00:00
2	Goodman	John	1952-06-20 00:00:00
4	Keitel	Harvey	1939-05-13 00:00:00
7	Madsen	Michael	1958-09-25 00:00:00
5	Roth	Tim	1961-05-14 00:00:00
6	Thurman	Una	1970-04-29 00:00:00

The same query is now sorted by *birthDate* in descending order.

```
SELECT * FROM actor ORDER BY birthDate DESC;
```

id	lastName	firstName	birthDate
6	Thurman	Una	1970-04-29 00:00:00
5	Roth	Tim	1961-05-14 00:00:00
7	Madsen	Michael	1958-09-25 00:00:00
3	Buscemi	Steve	1957-12-13 00:00:00
2	Goodman	John	1952-06-20 00:00:00
1	Bridges	Jeff	1949-12-04 00:00:00
4	Keitel	Harvey	1939-05-13 00:00:00

3.4.3 Joins

Often times we wish to query information stored on two different tables. We need to connect both tables together by means of a common column.

For example, consider if we wished to retrieve a list of film characters along with the actors who play the characters.

Examining the *filmcharacter* table we have a list of characters. However, the corresponding actor's *firstName* and *lastName* is stored in the *actor* table.

The *filmcharacter* table contains an *actorID* column which is a reference to the *id* column in the *actor* table. Fortunately we can join the two tables together via the common *id* value.

```
SELECT
    filmcharacter.filmID,
    actor.lastName,
    actor.firstName,
    filmcharacter.characterName
FROM
    actor,
    filmcharacter
WHERE
    actor.ID = filmcharacter.actorID;
```

filmID	lastName	firstName	characterName
1	Bridges	Jeff	The Dude
1	Goodman	John	Walter
1	Buscemi	Steve	Donny
2	Buscemi	Steve	Mr. Pink
2	Keitel	Harvey	Mr. White
3	Keitel	Harvey	The Wolf
2	Roth	Tim	Mr. Orange
3	Roth	Tim	Pumpkin
3	Thurman	Una	Mia Wallace
2	Madsen	Michael	Mr. Blonde
4	Thurman	Una	The Bride
4	Madsen	Michael	Budd

3.4.4 Functions

MySQL contains many functions which we can use to manipulate our query output.

For example, the `CONCAT(str1,str2,...)` function concatenates (adds together) a number of strings.

In this example, we will use this function to combine the *firstName* and *lastName* of all the available actors. We put a single space character between the two parts of the name.

```
SELECT CONCAT(firstName, ' ', lastName) FROM actor;
```

```
+-----+
| CONCAT(firstName, ' ', lastName) |
+-----+
| Jeff Bridges                      |
| John Goodman                      |
| Steve Buscemi                     |
| Harvey Keitel                     |
| Tim Roth                          |
| Una Thurman                       |
| Michael Madsen                    |
+-----+
```

A full list of the available functions is available in the MySQL manual.

3.4.5 Aggregate Functions

Aggregate functions are special types of functions which operate on the entire set of rows in a query. Normal functions act separately on each individual row.

Some common aggregate functions are:

Aggregate Function	Returns
COUNT(expr)	A count of the number of NON NULL values
AVG(expr)	The average value
MIN(expr)	The minimum value
MAX(expr)	The maximum value
SUM(expr)	The sum of all values

For example:

```
SELECT SUM(year), AVG(year), COUNT(year),
       MAX(year), MIN(year) FROM film;
```

```
+-----+-----+-----+-----+-----+
| SUM(year) | AVG(year) | COUNT(year) | MAX(year) | MIN(year) |
+-----+-----+-----+-----+-----+
|      7987 | 1996.7500 |           4 |      2003 |      1992 |
+-----+-----+-----+-----+-----+
```

3.4.6 DISTINCT

The DISTINCT clause ensures that no identical rows exist in the query output.

If we execute the following query we get some genres repeated. This is because some genres are associated with more than one film.

```
SELECT genreName FROM filmgenre;
```

```
+-----+
| genreName |
+-----+
| comedy    |
| mystery   |
| thriller  |
| action    |
| crime     |
| mystery   |
| thriller  |
| crime     |
| drama     |
| action    |
| crime     |
| thriller  |
+-----+
```

However, if we add a `DISTINCT` clause no repetition takes place:

```
SELECT DISTINCT genreName FROM filmgenre;
```

```
+-----+
| genreName |
+-----+
| comedy    |
| mystery   |
| thriller   |
| action    |
| crime     |
| drama     |
+-----+
```

3.5 Security

Each time the user attempts to execute a statement, the MySQL server determines whether or whether not to allow the statement to execute based upon a permissions system. The permissions are based upon three criteria:

- username
- password
- client host (hostname or IP Address of client)

The *GRANT* and *REVOKE* statements can be used to add/remove permissions to access databases and tables. The full syntax for these statements is available in the MySQL manual.

The following example adds permission for a user called 'eoin' to connect from the same PC as the server with password 'mypassword'.

```
GRANT ALL PRIVILEGES ON test.* TO 'eoin'@'localhost'
IDENTIFIED BY 'mypassword'
```

3.6 MySQL Manual

The MySQL manual is an official comprehensive resource of documentation relating to the MySQL server. It lists all the available data types, statements, operators, functions etc.

It is an absolutely essential resource when using MySQL.

An English language version is available at:

<http://dev.mysql.com/doc/mysql/en/>