Module 8: Databases & MySQL

Introduction to Databases

SQL on MySQL databases

Database tables

Database records

CS 85:
PHP PROGRAMMING

Santa Monica College
Computer Science &
Information Systems Dept.

Relational Databases

Before learning SQL, relational databases have several concepts that are important to learn first. Databases store the data of an information system. We regroup data by groups of comparable data (all the employees, all the projects, all the offices...). For each group of comparable data, we create a *table*. This table is specially designed to suit this type of data (its attributes). For instance, a table named `employee` which stores all the employees would be designed like this:

employee the table	
id_employee the primary key	an integer
firstname a column	a string of characters a column type
lastname	a string of characters
phone	10 numbers
mail	a string of characters

And the company employees would be stored like this:

employee				
id_employee	**firstname**	**lastname**	**phone**	**mail**
1 a column value	Big	BOSS	936854270	big.boss@company.com
2	John	DOE	936854271	john.doe@company.com
3	Linus	TORVALDS	936854272	linus.torvalds@company.com

4	Jimmy	WALES	936854273	jimmy.wales@company.com
5	Larry	PAGE	936854274	larry.page@company.com

The data stored in a table is called *entities*. As a table is usually represented as an array, the data *attributes or fields* (first name, last name...) are shown in the *columns* and the *records* (the employees) are the *rows*. `id_employee` is a database specific technical identifier called a *primary key*. It is used to link the entities from a table to another. To do so, it must be unique for each row. A primary key is usually underlined. Any unique attribute (for instance, the mail) or group of attributes (for instance, the first name and last name) can be the table primary key but it is recommended to use an additional technical id (id_employee) for primary key.

Let's create a second table called `project` which stores the company projects:

employee	
id_employee	an integer
firstname	a string of characters
lastname	a string of characters
phone	10 numbers
mail	a string of characters

project	
id_project	an integer
name	a string of characters
created_on	a date
ended_on	a date
# manager	an integer

Project tables with records:

employee				
id_employee	firstname	lastname	phone	mail
1	Big	BOSS	936854270	big.boss@company.com
2	John	DOE	936854271	john.doe@company.com
3	Linus	TORVALDS	936854272	linus.torvalds@company.com
4	Jimmy	WALES	936854273	jimmy.wales@company.com
5	Larry	PAGE	936854274	larry.page@company.com

project				
id_project	name	created_on	ended_on	# manager
1	Google	1998-09-08	NULL	5
2	Linux	1991-01-01	NULL	3
3	Wikipedia	2001-01-01	NULL	4

Table Relationships

Relational databases normally consist of one or more tables. Relationship can exist between the tables. In the example id_project is the primary key of the project table and manager is a *foreign key*. A foreign key is a technical id which is equal to one of the primary keys stored in another table (here, the employee table). Doing this, the Google project is linked to the employee Larry PAGE. This link is called a *relationship*. A foreign key is usually preceded by a sharp. Note that several projects can point to a common manager so an employee can be the manager of several projects.

Now, we want to create, not a single link, but multiple links. So we create a *junction table*. A junction table is a table that isn't used to store data but links the entities of other tables. Let's create a table called members which links employees to project:

employee	
id_employee	an integer
firstname	a string of characters
lastname	a string of characters
phone	10 numbers
mail	a string of characters

members	
# id_employee	an integer
# id_project	an integer

project	
id_project	an integer
name	a string of characters
created_on	a date
ended_on	a date
# manager	an integer

And the employees and the projects can be linked like this:

employee				
id_employee	**firstname**	**lastname**	**phone**	**mail**
1	Big	BOSS	936854270	big.boss@company.com
2	John	DOE	936854271	john.doe@company.com
3	Linus	TORVALDS	936854272	linus.torvalds@company.com
4	Jimmy	WALES	936854273	jimmy.wales@company.com
5	Larry	PAGE	936854274	larry.page@company.com
6	Max	THE GOOGLER	936854275	max.the-googler@company.com
7	Jenny	THE WIKIPEDIAN	936854276	jenny.the-wikipedian@company.com

project				
id_project	**name**	**created_on**	**ended_on**	**# manager**
1	Google	1998-09-08	NULL	5
2	Linux	1991-01-01	NULL	3
3	Wikipedia	2001-01-01	NULL	4

members	
# id_employee	**# id_project**
3	2
2	1
4	3
5	1
2	3
6	1
7	3

An employee can be associated to several projects (John Doe with Google and Wikipedia) and a project can be associated to several employees (Wikipedia with Jimmy, John and Jenny), which is impossible with just a foreign key. A junction table hasn't its own primary key. Its primary key is the couple of foreign keys, as this couple is unique. A junction table can link more than two entity tables by containing more columns.

Relationships

So let's list the different types of relationships:

One to one

A one-to-one relationship exists between two tables when a related table contains exactly one record for each record in the primary table. You create one-to-one relationships when you want to break information into multiple, logical sets.

One to many

A one-to-many relationship exists in a relational database when one record in a primary table has many related records in a related table. You create a one-to-many relationship to eliminate redundant information in a single table. Ideally, primary and foreign keys are the only pieces of information in a relational database table that should be duplicated. Breaking tables into multiple related tables to reduce redundant and duplicate information is called ***normalization***.

Many to many

A many to many relationship exist in a relational database when many records in one table are related to many records in another table.

For each type of relationships, there is a way to link the entities:

- One to many relationship: create a foreign key from an entity table to the other,
- Many to many relationship: create a junction table,
- One to one relationship: just merge the two tables.

Now you know how to design a database schema and to put the data of your information into it. Now let's look at MySQL specifically and how to use SQL to create databases, tables, attributes and records.

MySQL

MySQL is the most popular database used with PHP. PHP with MySQL is a powerful combination showing the real power of Server-Side scripting. PHP has a wide range of MySQL functions available with the help of a separate module.

MySQL allows users to create tables, where data can be stored much more efficiently than the way data is stored in arrays.

In order to use MySQL or databases in general effectively, you need to understand SQL (Structured Query Language).

Note that this page uses the mysqli functions and not the old mysql functions.

Who uses MySQL

- Some of the most visited websites like Flickr, Facebook, Wikipedia, YouTube.
- Content Management Systems like WordPress, phpBB, Drupal, Joomla
- Last but not least, a large number of web developers across the world.

What is SQL?

Structured **Q**uery **L**anguage is a third generation language for working with relational databases. Being a 3G language it is closer to human language than machine language and therefore easier to understand and work with.

Why MySQL?

- Free: Released with GPL version 2 license (though a different license can be bought from Oracle, see below)
- Support - Online tutorials, forums, mailing list (lists.mysql.com), paid support contracts.
- Speed - One of the fastest databases available.
- Functionality - supports most of ANSI SQL commands.
- Ease of use - less need of training / retraining.
- Portability - easily import / export from Excel and other databases
- Scalable - Useful for both small as well as large databases containing billions of records and terabytes of data in hundreds of thousands of tables.
- Permission Control - selectively grant or revoke permissions to users.

MySQL Practical Guide

Installing MySQL

All in one solutions

As MySQL alone isn't enough to run a real database server, the more practical way to install it is to deploy an all in one pack in this purpose, including all the needed additional elements: Apache and PHP.

1. On Linux: XAMP or LAMP.
2. On Windows: XAMP, WAMP, or EasyPHP.

First go to the XAMPP Control Panel and start your local MySQL/MariaDB database server.

About the MySQL package:

After installing MySQL via XAMPP or WAMP. Take the following steps to secure your MySQL server.

MySQL comes with a build in command to secure your MySQL server with basic best practice security measures. If all your answers are "yes" to the “mysql_secure_installation”, this cleans up your installation, forces you to set a root password, asks you to test for anonymous users and makes your database internal.

Just be careful. Be sure that you are configuring MySQL to the specifications you want.

Here's the code:

```
mysql_secure_installation
```

Creating your own MySQL account and database:

Now that MySQL is installed, you wouldn't necessarily have your own account, so you have to log in as root.

To do this type:

```
sudo mysql -u root -p
```

(This means that you're logging on as the user "root" (**-u root**) and that you're requesting the password for "root" (**-p**))

Once you've managed to log in, your command-line should look like this: **mysql>**

Now you can check what databases (if any) are available to your user (in this case "root"):

```
show databases;
```

Let's get straight to the chase and create our own database. Let's call it **people**. While we're doing this we can also create our own user account. Two birds with one stone.

So first create the database:

```
create database people;
```

(NOTE: in this particular case, you have to be "root" to create new databases.)

Now we want to grant (**GRANT**) all user rights (**ALL**) from (**ON**) the entire (*****) **people** database to (**TO**) your account (*yourusername***@localhost**) with your user password being *stuffedpoodle* (**IDENTIFIED BY "stuffedpoodle"**).

So we'd input this as:

```
GRANT ALL ON people.* TO yourusername@localhost IDENTIFIED BY "stuffedpoodle";
```

Tada! You now have your own user account. Let's say you chose **ted** as your username. You've configured MySQL to say that **ted** can play around with the **people** database in whatever ways he wishes.

Now get out of MySQL by typing

```
exit
```

To start working with the **people** database, you can now login as **ted**:

```
mysql -u ted -p
```

Creating tables with information in your database:

In MySQL information is stored in tables. Tables contain columns and rows.

Ted has now created a **people** database. So we want now to enter some information into a table.

Login as **ted**.

Firstly, we need to make sure we're working with the **people** database. So typing:

```
select database();
```

will show you what database you're currently using. You should see a **NULL**, meaning that you're working with nothing at the moment.

So to start using the people database, type:

```
\u people
```

(NOTICE: Typing: **USE people** OR logging in as **mysql people -u ted -p** is also acceptable.)

So how to create a table. Keep in mind that we need to set all the column values (like surname, age etc.).

Now, remember that **->** symbol? MySQL reads your command as just one command, not a series. So, **->** enables you to enter your inputs in a nicer way than just writing everything on one line. (NOTE: The problem with this method is that if you screw up on a line and press ENTER to go to the next line, you can't go back and fix your mistake. That's why a nice way to do this is using something like *Notepad++* (set language to **SQL**) to write your code and just copy/paste that into the shell.)

First I have to explain a few things. If you don't know, we use brackets **()** to **encapsulate** code. (Often called *parenthesis*).

The first thing we will be writing after the **CREATE TABLE *tableName*** and the first bracket will be the *database ID* number (we use integers) of each person, mainly known as the **Primary Key**. It's kind of like a passport ID number. Each number is unique to its owner and it has to be to prevent duplication and imposters.

Now, any variable in SQL is created as

```
variableNAME variableTYPE otherVariableAttributes
```

So in order to **define** the Primary Key variable, we need to type for example:

peopleID(variableNAME) **int**(variableTYPE - short for "integer") **unsigned**(means we want our integer value to always be a positive number) **not null**(we want each row to have a value, so obviously the value can't be empty(NULL)) **auto_increment**(this ensures that each new row that is created will be a unique value) **primary key**(we are saying that this particular variable will be our Primary Key for this Table.)**,** (a reminder that the **,** symbol indicates the end of this line so MySQL knows to go to the next line)

There is the **int** data type and the **string** data type. These are both called **varchar** which stands for *variable characters*. You set the amount of characters someone is able to input into a **varchar** variable. Like this: **nameOfFattestMooseAlive varchar(30)** So **nameOfFattestMooseAlive** can have a maximum of 30 characters.

Note: Page 21 full list of variables types.

Okay, so let's see an example of how to create a table relating to the **people** database:

```
CREATE TABLE peopleInfo
(
peopleID int unsigned not null auto_increment primary key,
firstName varchar(30),
lastName varchar(30),
age int,
gender varchar(13)
);
```

Now you can type: **CREATE TABLE peopleInfo** and press ENTER if you'd like to start **->** and write the rest of the code or you can use SCITE and copy/paste it into your shell.

Great. We now completed our first Table.

<u>Now comes the part when we have to get some actual people into our **peopleInfo** Table.</u>

Since you're already using the **people** database, you can type

```
show tables;
```

to see what tables are currently in your database. To see the *properties* of your table type:

```
describe peopleInfo;
```

So, how to fill in our **peopleInfo** table with people...

This is done by telling MySQL **what** *rows* you are filling in and the **actual information/data** you want to fill in. So we want to **insert into** our table (specifying the rows) and inputting the **values**(actual data) that we want. (NOTE: We are not filling in the primary key.)

To create our first person you would type this:

```
INSERT INTO peopleInfo
(firstName, lastName, age, gender)
```

```
Values
("Bill", "Harper", 17, "male");
```

Great. Now if you want to printout to the screen all the information about your table, type:

```
select * from peopleInfo;
```

and there you have it. Your table now has one person stored in it.

Inserting lots of information into your table:

A brief point that shall be covered later, MySQL backs-up itself in .sql files. The reason this is smart is because it backs-up the actual code inside the text file. Keeping this in mind, let's say we want to add 10 other people into your peopleInfo table. It would be one hell of a hassle typing each person into existence. What if there were a 1000?

Here is a sample set of 10 INSERT commands. Create a blank .txt file and copy/paste this information into it, saving it as **tenPeople.sql** .

```
INSERT INTO peopleInfo (firstName, lastName, age, gender) values ("Mary", "Jones", 21, "female");
INSERT INTO peopleInfo (firstName, lastName, age, gender) values ("Jill", "Harrington", 19, "female");
INSERT INTO peopleInfo (firstName, lastName, age, gender) values ("Bob", "Mill", 26, "male");
INSERT INTO peopleInfo (firstName, lastName, age, gender) values ("Alfred", "Jinks", 23, "male");
INSERT INTO peopleInfo (firstName, lastName, age, gender) values ("Sandra", "Tussel", 31, "female");
INSERT INTO peopleInfo (firstName, lastName, age, gender) values ("Mike", "Habraha", 45, "male");
INSERT INTO peopleInfo (firstName, lastName, age, gender) values ("John", "Murry", 22, "male");
INSERT INTO peopleInfo (firstName, lastName, age, gender) values ("Jake", "Mechowsky", 34, "male");
```

```
INSERT INTO peopleInfo (firstName, lastName, age, gender) values ("Hobrah", "Hinbrah", 24, "male");
INSERT INTO peopleInfo (firstName, lastName, age, gender) values ("Laura", "Smith", 17, "female");
```

Excellent. Now we want to get all these people in our table. **exit** MySQL and go to the directory where you saved the **tenPeople.sql** file.

Once there, to get all the data into your database, type:

```
mysql -u ted -p people <tenPeople.sql
```

and enter your password.

Now log into MySQL and remember to select the database your using. **\u people**

Now check again what information you have.

Manipulating your database:

Now that we have a database full of people. We can display that information anyway we want.

A brief example would be

```
select firstName, lastName, gender from peopleInfo;
```

This would display to the screen only peoples name, surname and gender. You've not specified that you want peoples Database ID Number or Age to be displayed. And the great thing is you can choose whatever you want from the database to be displayed. Now, if you want to delete your table, simply type:

```
drop table peopleInfo;
```

Extra conditions:

You can also you extra conditions (filters) through when displaying data.

```
select * from peopleInfo where gender = 'female';
```

will display everyone who is female.

You can also compare numbers. For example:

```
select * from peopleInfo where age > 17;
```

will show everyone in your table who is older than 17.

```
>   greater than
<   less than
>=  greater or equal to
<=  less than or equal to
<>  not equal to
```

Let's say we wanted to display all people whose first names began with the letter "j". We would use the LIKE condition. (Makes sense, is your name LIKE the letter "j", well it start with j so yes.)

About the LIKE condition.

```
select * from peopleInfo where firstName LIKE "j%";
```

MySQL Login6y78u

To enter the SQL commands:

- Launch MySQL in shell:
    - Linux: `mysql` -h localhost -u root MyDB
        - Windows: `"` `C:\xampp\mysql\bin\mysql.exe"` -h localhost -u root -p

Databases SQL Commands

Creation Database: Create new database

```
CREATE DATABASE databaseName;
```

Delete Database: Delete existing database

```
DROP DATABASE databaseName;
```

Language/Browsing the databases

INFORMATION_SCHEMA

`information_schema` is a virtual database that contains metadata about the server and the databases.You can't modify structure and data of `information_schema`. You can only query the tables. Many `information_schema` tables provide the same data you can retrieve with a SHOW statement. While using SHOW commands is faster (the server responds much faster and you type less characters), the `information_schema` provides a more flexible way to obtain and organize the metadata.

List databases

The INFORMATION_SCHEMA table containing the databases information is SCHEMATA.

The following SQL commands provide information about the databases located on the current server.

Show all databases:

```
SHOW DATABASES;
```

The SCHEMA keywords can be used in place of DATABASES. MySQL doesn't support standard SQL SCHEMAs, so SCHEMA is a synonym of database. It has been added for compatibility with other DBMSs.

Add a filter on the databases names

```
SHOW DATABASES LIKE 'pattern';
```

The LIKE operator here works as in normal SELECTs or DML statements. So you can list all databases whose name starts with 'my':

```
SHOW DATABASES LIKE 'my%';
```

Add complex filters

You can add more complex filters using the WHERE clause:

```
SHOW DATABASES WHERE conditions;
```

WHERE clause allows you to use regular expressions, '=', '<' and '>' operators, string functions or other useful expressions to filter the records returned by SHOW DATABASES.

List tables and views

The following SQL commands provide information about the tables and views contained in a database. The INFORMATION_SCHEMA tables containing this information are `TABLES` and `VIEWS`.

Since the following statements provide very little information about views, if you need to get metadata about them you'll probably prefer to query the VIEWS table.

The `mysqlshow` command line tool can be used instead.

Show all tables

```
USE `database`;
SHOW TABLES;
SHOW TABLES FROM `database`;
```

The two forms shown above are equivalent.

Apply a filter

You can apply a filter to the tables names, to show only tables whose name match a pattern. You can use the LIKE operators, as you do in SELECTs or in the DML statements:

```
SHOW TABLES LIKE `pattern`;
```

Also, you can apply a more complex filter to any column returned by the SHOW TABLES command using the WHERE clause:

```
SHOW TABLES WHERE condition;
```

Extra info

By default, SHOW TABLES returns only one column containing the name of the table. You can get extra information by using the FULL keyword:

```
SHOW FULL TABLES;
```

This will add a column called `Table_type`. This can have three values: 'BASE TABLE' for tables, 'VIEW' for views and 'SYSTEM VIEW' for special tables created by the server (normally used only INFORMATION_SCHEMA tables).

So you can only list tables:

```
SHOW FULL TABLES WHERE `Table_type`='BASE TABLE';
```

Or, you can only list views:

```
SHOW FULL TABLES WHERE `Table_type`='VIEW';
```

Show only open tables

You can get a list of the non-temporary tables (not views) which are open in the cache:

```
SHOW OPEN TABLES;
```

This command has the same parameters as SHOW TABLES, except for FULL (useless in this case). You can't get this information from the INFORMATION_SCHEMA.

List fields

The following SQL commands provide information about the columns in a table or in a view. The INFORMATION_SCHEMA table containing this information is COLUMNS.

The `mysqlshow` command line tool can be used instead.

DESCRIBE

```
DESCRIBE `table`;
DESCRIBE `database`.`table`;
DESCRIBE `table` 'filter';
```

DESC can be used as a shortcut for DESCRIBE.

'filter' can be a column name. If a column name is specified, only that column will be shown. If 'filter' contains the '%' or the '_' characters, it will be evaluated as a LIKE condition. For example, you can list all fields which start with 'my':

```
DESC `table` 'my%';
```

EXPLAIN

A synonym is:

```
EXPLAIN `table`;
```

SHOW FIELDS

Another synonym is:

```
SHOW FIELDS FROM `table`;
```

SHOW COLUMNS

Another synonym is:

```
SHOW COLUMNS FROM `table`;

-- possible clauses:
SHOW COLUMNS FROM `table` FROM `database`;
SHOW COLUMNS FROM `table` LIKE 'pattern';
SHOW COLUMNS FROM `table` WHERE condition;
```

FIELDS and COLUMNS are synonyms. EXPLAIN is a synonym of SHOW COLUMNS / FIELDS too, but it doesn't support all of its clauses.

A databases name can be specified both in the form

```
SHOW COLUMNS FROM `table` FROM `database`;
```

both:

```
SHOW COLUMNS FROM `database`.`table`;
```

List indexes

The following SQL commands provide information about the indexes in a table. Information about keys is contained in the `COLUMNS` table in the INFORMATION_SCHEMA.

The `mysqlshow -k` command line tool can be used instead.

```
SHOW INDEX FROM `TABLE`;
SHOW INDEX FROM `TABLE` FROM `databases`;
```

The KEYS reserved word can be used as a synonym of INDEX. No other clauses are provided.

Result example:

Table	Non_uniq	Key_name	Seq_in_in	Column	Collation	Cardinalit	Sub_part	Packed	Null	Index_ty	Comment	Index_co
Table 1	0	PRIMARY	1	id	A	19	NULL	NULL		BTREE		

To remove an index:

```
DROP INDEX `date_2` on `Table1`
```

Language/Alias

An expression and a column may be given aliases using AS. The alias is used as the expression's column name and can be used with order by or having clauses. For e.g.

```
SELECT
    CONCAT(last_name,' ', first_name) AS full_name,
    nickname AS nick
FROM
    mytable
ORDER BY
    full_name
```

These aliases can be used in ORDER BY, GROUP BY and HAVING clauses. They should not be used in WHERE clause.

A table name can have a shorter name for reference using AS. You can omit the AS word and still use aliasing. For e.g.

```
SELECT
    COUNT(B.Booking_ID), U.User_Location
FROM
    Users U
LEFT OUTER JOIN
    Bookings AS B
ON
    U.User_ID    = B.Rep_ID AND
    B.Project_ID = '10'
GROUP BY
    (U.User_Location)
```

Aliasing plays a crucial role while you are using self joins. For e.g. people table has been referred to as p and c aliases!

```
SELECT
    p.name                          AS parent,
    c.name                          AS child,
    MIN((TO_DAYS(NOW())-TO_DAYS(c.dob))/365) AS minage
FROM
    people AS p
LEFT JOIN
    people AS c
```

```
ON
   p.name=c.parent WHERE c.name IS NOT NULL
GROUP BY
   parent HAVING minage > 50 ORDER BY p.dob;
```

Language/Data Types

VARCHAR

VARCHAR is shorthand for CHARACTER VARYING. 'n' represents the maximum column length (upto 65,535 characters). A VARCHAR(10) column can hold a string with a maximum length of 10 characters. The actual storage required is the length of the string (L), plus 1 or 2 bytes (1 if the length is < 255) to record the length of the string.
For the string 'abcd', L is 4 and the storage requirement is 5 bytes.

CHAR(n) is similar to varchar(n) with the only difference that char will occupy fixed length of space in the database whereas varchar will need the space to store the actual text.

TEXT and BLOB

A BLOB or TEXT column with a maximum length of 65,535 characters. The required space is the real length of the stored data plus 2 bytes (1 byte if length is < 255). The BLOB / TEXT data is not stored in the table's datafile. This makes all operations (INSERT / UPDATE / DELETE / SELECT) involving the BLOB / TEXT data slower, but makes all other operations faster.

Integer

Specifying an n value has no effect whatsoever. Regardless of a supplied value for n, maximum (unsigned) value stored is 429 crores. If you want to add negative numbers, add the "signed" keyword next to it.

Decimal

decimal(n,m) decimal(4,2) means numbers upto 99.99 (and NOT 9999.99 as you may expect) can be saved. Four digits with the last 2 reserved for decimal.

Dates

Out of the three types DATETIME, DATE, and TIMESTAMP, the DATE type is used when you need only a date value, without a time part. MySQL retrieves and displays DATE values in 'YYYY-MM-DD' format. The DATETIME type is used when you need values that contain both date and time information. The difference between DATETIME and TIMESTAMP is that the TIMESTAMP range is limited to 1970-2037 (see below).

TIME can be used to only store the time of day (HH:MM:SS), without the date. It can also be used to represent a time interval (for example: -02:00:00 for "two hours in the past"). Range: '-838:59:59' => '838:59:59'.

YEAR can be used to store the year number only.

If you manipulate dates, you have to specify the actual date, not only the time - that is, MySQL will not automagically use today as the current date. On the contrary, MySQL will even interpret the HH:MM:SS time as a YY:MM:DD value, which will probably be invalid.

The following examples show the precise date range for Unix-based timestamps, which starts at the

Unix Epoch and stops just before the first new year before the usual limit (2038).

```
mysql> SET time_zone = '+00:00'; -- GMT
Query OK, 0 rows affected (0.00 sec)
mysql> SELECT FROM_UNIXTIME(-1);
+-------------------+
| FROM_UNIXTIME(-1) |
+-------------------+
| NULL              |
+-------------------+
1 row in set (0.00 sec)

mysql> SELECT FROM_UNIXTIME(0); -- "Epoch"
+---------------------+
| FROM_UNIXTIME(0)    |
+---------------------+
| 1970-01-01 00:00:00 |
+---------------------+
1 row in set (0.00 sec)
```

```
mysql> SELECT FROM_UNIXTIME(2145916799);
```

```
+---------------------------+
| FROM_UNIXTIME(2145916799) |
+---------------------------+
| 2037-12-31 23:59:59       |
+---------------------------+
1 row in set (0.00 sec)

mysql> SELECT FROM_UNIXTIME(2145916800);
```

```
+---------------------------+
| FROM_UNIXTIME(2145916800) |
+---------------------------+
| NULL                      |
+---------------------------+
1 row in set (0.00 sec)
```

Language/Table manipulation

CREATE TABLE

Create table syntax is:

```
Create table tablename (FieldName1 DataType, FieldName2 DataType)
```

The rows returned by the "select" query can be saved as a new table. The datatype will be the same as the old table. For e.g.

```
CREATE TABLE LearnHindi
select english.tag, english.Inenglish as english, hindi.Inhindi as hindi
FROM english, hindi
WHERE english.tag = hindi.tag
```

The table size limit depends on the filesystem, and is generally around 2TB[1].

Copy a table

To duplicate the same structure (names, fields types, and indexes, but no record):

```
CREATE TABLE `new1` LIKE `old1`;
```

To copy the records into the previous result:

```
INSERT INTO `new1` SELECT * FROM `old1`;
```

Temporary tables

It's possible to create variables of type "table", which as the other variables, will be erased at the end of their scripts. It's called the "temporary tables":

```
CREATE TEMPORARY TABLE IF NOT EXISTS MyTempTable1 AS (SELECT * FROM MyTable1)
```

Example with a named column:

```
CREATE TEMPORARY TABLE IF NOT EXISTS MyTempTable1(id INT) AS (SELECT id FROM MyTable1)
```

Attention: if the temporary table column name doesn't correspond to the field which fills it, an additional column will be added with this field name. Eg:

```
CREATE TEMPORARY TABLE IF NOT EXISTS MyTempTable1(id1 INT) AS (SELECT id FROM
MyTable1);
SHOW FIELDS FROM MyTempTable1;
Field    Type    Null   Key    Default     Extra
id1    int(11)      YES          NULL
id    int(11)      NO           0
```

Attention: all temporary tables are dropped at the end of the MySQL connection which had created them.

ALTER TABLE

ALTER TABLE command can be used when you want to add/delete/modify the columns and/or the indexes; or, it can be used to change other table properties.

Add a column:

```
ALTER TABLE awards
ADD COLUMN AwardCode int(2)
```

Modify a column:

```
ALTER TABLE awards
```

```
CHANGE COLUMN AwardCode VARCHAR(2) NOT NULL
```

```
ALTER TABLE awards
MODIFY COLUMN AwardCode VARCHAR(2) NOT NULL
```

Drop a column:

```
ALTER TABLE awards
```

```
DROP COLUMN AwardCode
```

Re-order the record in a table:

```
ALTER TABLE awards ORDER BY id
```

(this operation is only supported by some Storage Engines; it could make some query faster)

Rename a table

In order to rename a table, you must have ALTER and DROP privileges on the old table name (or on all the tables), and CREATE and INSERT privileges on the new table name (or on all the tables).

You can use ALTER TABLE to rename a table:

```
RENAME TABLE `old_name` TO `new_name`
```

You can rename more than one table with a single command:

```
RENAME TABLE `old1` TO `new1`, `old2` TO `new2`, ...
```

RENAME is a shortcut. You can also use the ALTER TABLE statement:

```
ALTER TABLE `old` RENAME `new`
```

Using ALTER TABLE you can only rename one table per statement, but it's the only way to rename temporary tables.

DROP TABLE

```
DROP TABLE `awards`
```

Will completely delete the table and all the records it contains.

You can also drop more than one table with a single statement:

```
DROP TABLE `table1`, `table2`, ...
```

There are some optional keywords:

```
DROP TEMPORARY TABLE `table`;
DROP TABLE `table` IF EXISTS;
```

TEMPORARY must be specified, to drop a temporary table. IF EXISTS tells the server that it must not raise an error if the table doesn't exist.

Language/Data manipulation

INSERT

The syntax is as follows:

Insert value1 into Column1, value2 into Column2, and value3 into Column3:

```
INSERT INTO TableName (Column1, Column2, Column3)
VALUES (value1, value2, value3)
```

Insert one record (values are inserted in the order that the columns appear in the database):

```
INSERT INTO TableName
VALUES (value1, value2, value3)
```

Insert two records:

```
INSERT INTO TableName
VALUES (value1, value2, value3), (value4, value5, value6)

INSERT INTO antiques VALUES (21, 01, 'Ottoman', 200.00);
INSERT INTO antiques (buyerid, sellerid, item) VALUES (01, 21, 'Ottoman');
```

You can also insert records 'selected' from other table.

```
INSERT INTO table1(field1, field2)
SELECT field1, field2
FROM table2
```

```
INSERT INTO World_Events SELECT * FROM National_Events
```

UPDATE

The syntax is:

```
UPDATE table SET field1 = newvalue1, field2 = newvalue2 WHERE criteria ORDER BY field LIMIT n
```

Examples are:

```
UPDATE owner SET ownerfirstname = 'John'
  WHERE ownerid = (SELECT buyerid FROM antiques WHERE item = 'Bookcase');

UPDATE antiques SET price = 500.00 WHERE item = 'Chair'

UPDATE order SET discount=discount * 1.05

UPDATE tbl1 JOIN tbl2 ON tbl1.ID = tbl2.ID
  SET tbl1.col1 = tbl1.col1 + 1
  WHERE tbl2.status='Active'

UPDATE tbl SET names = REPLACE(names, 'aaa', 'zzz')

UPDATE products_categories AS pc
  INNER JOIN products AS p ON pc.prod_id = p.id
  SET pc.prod_sequential_id = p.sequential_id

UPDATE table_name SET col_name =
  REPLACE(col_name, 'host.domain.com', 'host2.domain.com')

UPDATE posts SET deleted=True
  ORDER BY date LIMIT 1
```

With ORDER BY you can order the rows before updating them, and only update a given number of rows (LIMIT).

It is currently not possible to update a table while performing a subquery on the same table. For example, if I want to reset a password I forgot in SPIP:

```
mysql> UPDATE spip_auteurs SET pass =
 (SELECT pass FROM spip_auteurs WHERE login='paul') where login='admin';
ERROR 1093 (HY000): You can't specify target table 'spip_auteurs' for update in FROM clause
```

TODO: describes a work-around that I couldn't make to work with MySQL 4.1. Currently the work-around is not use 2 subqueries, possibly with transactions[1].

Performance tips

- UPDATEs speed depends of how many indexes are updated.
- If you UPDATE a MyISAM table which uses dynamic format, if you make rows larger they could be split in more than one part. This causes reading overhead. So, if your applications often do this, you may want to regularly run an OPTIMIZE TABLE statement.
- Performing many UPDATEs all together on a LOCKed table is faster than performing them individually.

REPLACE

REPLACE works exactly like INSERT, except that if an old record in the table has the same value as a new record for a PRIMARY KEY or a UNIQUE index, the old record is deleted before the new record is inserted.

IGNORE

Since MySQL 5.5[2], "INSERT IGNORE" and "REPLACE IGNORE" allow, when a duplicate key error occurs, to display some warnings and avoid the statement to abort.

Prior to MySQL 4.0.1, INSERT ... SELECT implicitly operates in IGNORE mode. As of MySQL 4.0.1, specify IGNORE explicitly to ignore records that would cause duplicate-key violations.

DELETE and TRUNCATE

```
DELETE [QUICK] FROM `table1`
TRUNCATE [TABLE] `table1`
```

- If you don't use a WHERE clause with DELETE, all records will be deleted.
- It can be very slow in a large table, especially if the table has many indexes.
- If the table has many indexes, you can make the cache larger to try making the DELETE faster (key_buffer_size variable).
- For indexed MyISAM tables, in some cases DELETEs are faster if you specify the QUICK keyword (DELETE QUICK FROM ...). This is only useful for tables where DELETEd index values will be reused.
- TRUNCATE will delete all rows quickly by DROPping and reCREATE-ing the table (not all Storage Engines support this operation).
- TRUNCATE is not transaction-safe nor lock-safe.
- DELETE informs you how many rows have been removed, but TRUNCATE doesn't.
- After DELETing many rows (about 30%), an OPTIMIZE TABLE command should make next statements faster.
- For a InnoDB table with FOREIGN KEYs constraints, TRUNCATE behaves like DELETE.

```
DELETE FROM `antiques`
```

```
WHERE item = 'Ottoman'
ORDER BY `id`
LIMIT 1
```

You can order the rows before deleting them, and then delete only a given number of rows.

Language/Queries

SELECT

select syntax is as follows:

```
SELECT *
FROM a_table_name
WHERE condition
GROUP BY grouped_field
HAVING group_name condition
ORDER BY ordered_field
LIMIT limit_number, offset
```

List of fields

You must specify what data you're going to retrieve in the SELECT clause:

```
SELECT DATABASE() -- returns the current db's name
SELECT CURRENT_USER() -- returns your username
SELECT 1+1 -- returns 2
```

Any SQL expression is allowed here.

You can also retrieve all fields from a table:

```
SELECT * FROM `stats`
```

If you SELECT only the necessary fields, the query will be faster.

The table's name

If you are retrieving results from a table or a view, usually you specify the table's name in the FROM clause:

```
SELECT id FROM `stats` -- retrieve a field called id from a table called stats
```

Or:

```
SELECT MAX(id) FROM `stats`
SELECT id*2 FROM `stats`
```

You can also use the `db_name`.`table_name` syntax:

```
SELECT id FROM `sitedb`.`stats`
```

But you can also specify the table's name in the SELECT clause:

```
SELECT `stats`.`id` -- retrieve a field called id from a table
SELECT `sitedb`.`stats`.`id`
```

WHERE

You can set a filter to decide what records must be retrieved.

For example, you can retrieve only the record which has an id of 42:

```
SELECT * FROM `stats` WHERE `id`=42
```

Or you can read more than one record:

```
SELECT * FROM `antiques` WHERE buyerid IS NOT NULL
```

GROUP BY

You can group all records by one or more fields. The record which have the same value for that field will be grouped in one computed record. You can only select the grouped record and the result of some aggregate functions, which will be computed on all records of each group.

For example, the following will group all records in the table `users` by the field `city`. For each group of users living in the same city, the maximum age, the minimum age and the average age will be returned:

```
SELECT city, MAX(age), MIN(age), AVG(age) GROUP BY `city`
```

In the following example, the users are grouped by city and sex, so that we'll know the max, min and avg age of male/female users in each city:

```
SELECT city, sex, MAX(age), MIN(age), AVG(age) GROUP BY `city`, `sex`
```

HAVING

The HAVING clause declares a filter for the records which are computed by the GROUP BY clause. It's different from the WHERE clause, that operates before the GROUP BY. Here's what happens:

1. The records which match to the WHERE clause are retrieved
2. Those records are used to compute new records as defined in the GROUP BY clause
3. The new records that match to the HAVING conditions are returned

This means which WHERE decides what record are used to compose the new computed records.

HAVING decides what computed records are returned, so it can operate on the results of aggregate functions. HAVING is not optimized and can't use indexes.

Incorrect use of HAVING:

```
SELECT city, sex, MAX(age), MIN(age), AVG(age) GROUP BY `city` HAVING sex='m'
```

This probably gives a wrong result. MAX(age) and other aggregate calculations are made using all values, even if the record's sex value is 'f'. This is hardly the expected result.

Incorrect use of HAVING:

```
SELECT city, sex, MAX(age), MIN(age), AVG(age) GROUP BY `city`, `sex` HAVING sex='m'
```

This is correct and returns the expected results, but the execution of this query is not optimized. The WHERE clause can and should be used, because, so that MySQL doesn't compute records which are excluded later.

Correct use of HAVING:

```
SELECT city, sex, MAX(age), MIN(age), AVG(age) GROUP BY `city` HAVING MAX(age) > 80
```

It must group all records, because can't decide the max age of each city before the GROUP BY clause is execute. Later, it returns only the record with a MAX(age)>80.

ORDER BY

You can set an arbitrary order for the records you retrieve. The order may be alphabetical or numeric.

```
SELECT * FROM `stats` ORDER BY `id`
```

By default, the order is ASCENDING. You can also specify that the order must be DESCENDING:

```
SELECT * FROM `stats` ORDER BY `id` ASC -- default
SELECT * FROM `stats` ORDER BY `id` DESC -- inverted
```

NULLs values are considered as minor than any other value.

You can also specify the field position, in place of the field name:

```
SELECT `name`, `buyerid` FROM `antiques` ORDER BY 1 -- name
SELECT `name`, `buyerid` FROM `antiques` ORDER BY 2 -- buyerid
SELECT `name`, `buyerid` FROM `antiques` ORDER BY 1 DESC
```

SQL expressions are allowed:

```
SELECT `name` FROM `antiques` ORDER BY REVERSE(`name`)
```

You can retrieve records in a random order:

```
SELECT `name` FROM `antiques` ORDER BY RAND()
```

If a GROUP BY clause is specified, the results are ordered by the fields named in GROUP BY, unless an ORDER BY clause is present. You can even specify in the GROUP BY clause if the order must be ascending or descending:

```
SELECT city, sex, MAX(age) GROUP BY `city` ASC, `sex` DESC
```

If you have a GROUP BY but you don't want the records to be ordered, you can use ORDER BY NULL:

```
SELECT city, sex, MAX(age) GROUP BY `city`, `sex` ORDER BY NULL
```

LIMIT

You can specify the maximum of rows that you want to read:

```
SELECT * FROM `antiques` ORDER BY id LIMIT 10
```

This statement returns a maximum of 10 rows. If there are less than 10 rows, it returns the number of rows found. The limit clause is usually used with ORDER BY.

You can get a given number of random records:

```
SELECT * FROM `antiques` ORDER BY rand() LIMIT 1 -- one random record
SELECT * FROM `antiques` ORDER BY rand() LIMIT 3
```

You can specify how many rows should be skipped before starting to return the records found. The first record is 0, not one:

```
SELECT * FROM `antiques` ORDER BY id LIMIT 10
SELECT * FROM `antiques` ORDER BY id LIMIT 0, 10 -- synonym
```

You can use the LIMIT clause to get the pagination of results:

```
SELECT * FROM `antiques` ORDER BY id LIMIT 0, 10 -- first page
SELECT * FROM `antiques` ORDER BY id LIMIT 10, 10 -- second page
SELECT * FROM `antiques` ORDER BY id LIMIT 20, 10 -- third page
```

Also, the following syntax is acceptable:

```
SELECT * FROM `antiques` ORDER BY id LIMIT 10 OFFSET 10
```

You can use the LIMIT clause to check the syntax of a query without waiting for it to return the results:

```
SELECT ... LIMIT 0
```

Optimization tips:

- SQL_CALC_FOUND_ROWS may speed up a query [1][2]
- LIMIT is particularly useful for SELECTs which use ORDER BY, DISTINCT and GROUP BY, because their calculations don't have to involve all the rows.
- If the query is resolved by the server copying internally the results into a temporary table, LIMIT helps MySQL to calculate how much memory is required by the table.

DISTINCT

The DISTINCT keyword can be used to remove all duplicate rows from the resultset:

```
SELECT DISTINCT * FROM `stats` -- no duplicate rows
SELECT DISTINCTROW * FROM `stats` -- synonym
SELECT ALL * FROM `stats` -- duplicate rows returned (default)
```

You can use it to get the list of all values contained in one field:

```
SELECT DISTINCT `type` FROM `antiques` ORDER BY `type`
```

Or you can use it to get the existing combinations of some values:

```
SELECT DISTINCT `type`, `age` FROM `antiques` ORDER BY `type`
```

If one of the fields you are SELECTing is the PRIMARY KEY or has a UNIQUE index, DISTINCT is useless. Also, it's useless to use DISTINCT in conjunction with the GROUP BY clause.

IN and NOT IN

```
SELECT id
FROM stats
WHERE position IN ('Manager', 'Staff')

SELECT ownerid, 'is in both orders & antiques'
FROM orders, antiques WHERE ownerid = buyerid
UNION
SELECT buyerid, 'is in antiques only'
FROM antiques WHERE buyerid NOT IN (SELECT ownerid FROM orders)
```

EXISTS and ALL

```
SELECT ownerfirstname, ownerlastname

FROM owner

WHERE EXISTS (SELECT * FROM antiques WHERE item = 'chair')

SELECT buyerid, item
FROM antiques
WHERE price = ALL (SELECT price FROM antiques)
```

Optimization hints

There are some hints you may want to give to the server to better optimize the SELECTs. If you give more than one hints, the order of the keywords is important:

```
SELECT [ALL | DISTINCT | DISTINCTROW ]
  [HIGH_PRIORITY] [STRAIGHT_JOIN]
  [SQL_SMALL_RESULT | SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
  [SQL_CACHE | SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS]
  ...
```

HIGH_PRIORITY

Usually, DML commands (INSERT, DELETE, UPDATE) have higher priority than SELECTs. If you specify HIGH_PRIORITY though, the SELECT will have higher priority than DML statements.

STRAIGHT_JOIN Force MySQL to evaluate the tables of a JOIN in the same order they are named, from the leftmost.

SQL_SMALL_RESULT It's useful only while using DISTINCT or GROUP BY. Tells the optimizer that the query will return few rows.

SQL_BIG_RESULT It's useful only while using DISTINCT or GROUP BY. Tells the optimizer that the query will return a many rows.

SQL_BUFFER_RESULT Force MySQL to copy the result into a temporary table. This is useful to remove LOCKs as soon as possible.

SQL_CACHE Forces MySQL to copy the result into the query cache. Only works if the value of query_cache_type is DEMAND or 2.

SQL_NO_CACHE Tells MySQL not to cache the result. Useful if the query occurs very seldom or if the result often change.

SQL_CALC_FOUND_ROWS Useful if you are using the LIMIT clause. Tells the server to calculate how many rows would have been returned if there were no LIMIT. You can retrieve that number with another query:

```
SELECT SQL_CALC_FOUND_ROWS * FROM `stats` LIMIT 10 OFFSET 100;
```

```
SELECT FOUND_ROWS();
```

Index hints

- `USE INDEX`: specifies to research some records preferably by browsing the tables indexes.
- `FORCE INDEX`: idem in more restrictive. A table will be browsed without index only if the optimizer doesn't have the choice.

- `IGNORE INDEX`: request to not favor the indexes.

Example:

```
SELECT *
FROM table1 USE INDEX (date)
WHERE date between '20150101' and '20150131'
```

UNION and UNION All

(Compatible: Mysql 4+)

Following query will return all the records from both tables.

```
SELECT * FROM english
UNION ALL
SELECT * FROM hindi
```

UNION is the same as UNION DISTINCT.
If you type only UNION, then it is considered that you are asking for distinct records. If you want all records, you have to use UNION ALL.

```
SELECT word FROM word_table WHERE id = 1
UNION
SELECT word FROM word_table WHERE id = 2

(SELECT magazine FROM pages)
UNION DISTINCT
(SELECT magazine FROM pdflog)
ORDER BY magazine

(SELECT ID_ENTRY FROM table WHERE ID_AGE = 1)
UNION DISTINCT
(SELECT ID_ENTRY FROM table WHERE ID_AGE=2)
```

Joins

The Most important aspect of SQL is its relational features. You can query, compare and calculate two different tables having entirely different structure. Joins and subselects are the two methods to join tables. Both methods of joining tables should give the same results. The natural join is faster on most SQL platforms.

In the following example a student is trying to learn what the numbers are called in hindi.

```
CREATE TABLE english (Tag int, Inenglish varchar(255));
CREATE TABLE hindi (Tag int, Inhindi varchar(255));

INSERT INTO english (Tag, Inenglish) VALUES (1, 'One');
INSERT INTO english (Tag, Inenglish) VALUES (2, 'Two');
INSERT INTO english (Tag, Inenglish) VALUES (3, 'Three');

INSERT INTO hindi (Tag, Inhindi) VALUES (2, 'Do');
INSERT INTO hindi (Tag, Inhindi) VALUES (3, 'Teen');
INSERT INTO hindi (Tag, Inhindi) VALUES (4, 'Char');
```

select * from english

Tag	Inenglish
1	One
2	Two
3	Three

select * from hindi

Tag	Inhindi
2	Do
3	Teen
4	Char

Cartesian join (CROSS JOIN)

A Cartesian join is when you join every row of one table to every row of another table.

```
SELECT * FROM english, hindi
```

It is also called Cross Join and may be written in this way:

```
SELECT * FROM english CROSS JOIN hindi
```

Tag	Inenglish	Tag	Inhindi
1	One	2	Do
2	Two	2	Do
3	Three	2	Do
1	One	3	Teen
2	Two	3	Teen
3	Three	3	Teen
1	One	4	Char
2	Two	4	Char

3	Three	4	Char

Inner Join

```
SELECT hindi.Tag, english.Inenglish, hindi.Inhindi
FROM english, hindi
WHERE english.Tag = hindi.Tag
-- equal
SELECT hindi.Tag, english.Inenglish, hindi.Inhindi
FROM english INNER JOIN hindi ON english.Tag = hindi.Tag
```

Tag	Inenglish	Inhindi
2	Two	Do
3	Three	Teen

You can also write the same query as

```
SELECT hindi.Tag, english.Inenglish, hindi.Inhindi
FROM english INNER JOIN hindi
ON english.Tag = hindi.Tag
```

Natural Joins using "using" (Compatible: MySQL 4+; but changed in MySQL 5) The following statement using "USING" method will display the same results.

```
SELECT hindi.tag, hindi.Inhindi, english.Inenglish
FROM hindi NATURAL JOIN english
USING (Tag)
```

Outer Joins

Tag	Inenglish	Tag	Inhindi
1	One		
2	Two	2	Do
3	Three	3	Teen
		4	Char

LEFT JOIN / LEFT OUTER JOIN

The syntax is as follows:

```
SELECT field1, field2 FROM table1 LEFT JOIN table2 ON field1=field2

SELECT e.Inenglish as English, e.Tag, '--no row--' as Hindi
FROM english AS e LEFT JOIN hindi AS h
ON e.Tag=h.Tag
WHERE h.Inhindi IS NULL

English  tag   Hindi
One      1     --no row-
```

Right Outer Join

```
SELECT '--no row--' AS English, h.tag, h.Inhindi AS Hindi
FROM english AS e RIGHT JOIN hindi AS h
ON e.Tag=h.Tag
WHERE e.Inenglish IS NULL
```

English tag Hindi --no row-- 4 Char

- Make sure that you have the same name and same data type in both tables.
- The keywords LEFT and RIGHT are not absolute, they only operate within the context of the given statement: we can reverse the order of the tables and reverse the keywords, and the result would be the same.
- If the type of join is not specified as inner or outer then it will be executed as an INNER JOIN.

Full Outer Join

As for v5.1, MySQL does not provide FULL OUTER JOIN. You may emulate it this way:

```
    (SELECT a.*, b*
        FROM tab1 a LEFT JOIN tab2 b
        ON a.id = b.id)
UNION
    (SELECT a.*, b*
        FROM tab1 a RIGHT JOIN tab2 b
        ON a.id = b.id)
```

Multiple joins

It is possible to join more than just two tables:

```
SELECT ... FROM a JOIN (b JOIN c on b.id=c.id) ON a.id=b.id
```

Here is an example from *Savane*:

```
mysql> SELECT group_type.type_id, group_type.name, COUNT(people_job.job_id) AS count
       FROM group_type
        JOIN (groups JOIN people_job
        ON groups.group_id = people_job.group_id)
        ON group_type.type_id = groups.type
       GROUP BY type_id ORDER BY type_id
+---------+-------------------------------------+-------+
| type_id | name                                | count |
+---------+-------------------------------------+-------+
|       1 | Official GNU software               |   148 |
|       2 | non-GNU software and documentation  |   268 |
|       3 | www.gnu.org portion                 |     4 |
|       6 | www.gnu.org translation team        |     5 |
+---------+-------------------------------------+-------+
4 rows in set (0.02 sec)
```

Subqueries

- SQL subqueries let you use the results of one query as part of another query.
- Subqueries are often natural ways of writing a statement.
- Let you break a query into pieces and assemble it.
- Allow some queries that otherwise can't be constructed. Without using a subquery, you have to do it in two steps.
- Subqueries always appear as part of the WHERE (or HAVING) clause.
- Only one field can be in the subquery SELECT. It means Subquery can only produce a single column of data as its result.
- ORDER BY is not allowed; it would not make sense.
- Usually refer to name of a main table column in the subquery.
- This defines the current row of the main table for which the subquery is being run. This is called an outer reference.

For e.g. If RepOffice= OfficeNbr from Offices table, list the offices where the sales quota for the office exceeds the sum of individual salespersons' quotas

```
SELECT City FROM Offices WHERE Target > ???
```

??? is the sum of the quotas of the salespeople, i.e.

```
SELECT SUM(Quota)
FROM SalesReps
WHERE RepOffice = OfficeNbr
```

We combine these to get

```
SELECT City FROM Offices
WHERE Target > (SELECT SUM(Quota) FROM SalesReps
WHERE RepOffice = OfficeNbr)
```

Display all customers with orders or credit limits > $50,000. Use the DISTINCT word to list the customer just once.

```
SELECT DISTINCT CustNbr
FROM Customers, Orders
WHERE CustNbr = Cust AND (CreditLimit>50000 OR Amt>50000);
```

Language/Using NULL

Null is a special logical value in SQL. Most programming languages have 2 values of logic: True and False. SQL also has NULL which means "Unknown". A NULL value can be set.

NULL is a non-value, so it can be assigned to TEXT columns, INTEGER columns or any other datatype. A column can not contain NULLs only if it has been declared as NOT NULL (see ALTER TABLE).

```
INSERT into Singer
    (F_Name, L_Name, Birth_place, Language)
    values
    ("", "Homer", NULL, "Greek"),
    ("", "Sting", NULL, "English"),
    ("Jonny", "Five", NULL, "Binary");
```

Do not quote the NULL. If you quote a Null then you name the person NULL. For some strange reason, NULLs do not show visually on windows XP in Varchar fields but they do in Fedora's version, so versions of mysql can give different outputs. Here we set the value of Sting and Homer's first name to a zero length string "", because we KNOW they have NO first name, but we KNOW we do not know the place they were born. To check for a NULLs use

```
SELECT * from Singer WHERE Birth_place IS NULL;
```

```
or
SELECT * from Singer WHERE Birth_place IS NOT NULL;
or
SELECT * from Singer WHERE isNull(Birth_place)
```

Remember, COUNT never counts NULLS.

```
select count(Birth_place) from Singer;
0
and sum(NULL) gives a NULL answer.
```

Normal operations (comparisons, expressions...) return NULL if at least one of the compared items is NULL:

```
SELECT (NULL=NULL) OR (NULL<>NULL) OR (NOT NULL) OR (1<NULL) OR (1>NULL) OR (1 + NULL)
OR (1 LIKE NULL)
```

because all the expressions between in parenthesis return NULL. It's definitely logical: if you don't know the value represented by NULL, you don't know is it's =1 or <>1. Be aware that even (NULL=NULL and (NOT NULL) return NULL.

Dealing with NULL

The function 'COALESCE' can simplify working with null values. for example, to avoid showing null values by treating null as zero, you can type:

```
SELECT COALESCE(colname,0) from table where COALESCE(colname,0) > 1;
```

In a date field, to treat NULL as the current date:

```
ORDER BY (COALESCE(TO_DAYS(date),TO_DAYS(CURDATE()))-TO_DAYS(CURDATE()))
EXP(SUM(LOG(COALESCE(''*the field you want to multiply*'',1)))
```

The coalesce() function is there to guard against trying to calculate the logarithm of a null value and may be optional depending on your circumstances.

```
SELECT t4.gene_name, COALESCE(g2d.score,0),
COALESCE(dgp.score,0), COALESCE(pocus.score,0)
FROM t4
LEFT JOIN g2d ON t4.gene_name=g2d.gene_name
LEFT JOIN dgp ON t4.gene_name=dgp.gene_name
LEFT JOIN pocus ON t4.gene_name=pocus.gene_name;
```

Use of IFNULL() in your SELECT statement is to make the NULL any value you wish.

```
IFNULL(expr1,expr2)
```

If expr1 is not NULL, IFNULL() returns expr1, else it returns expr2.

IFNULL() returns a numeric or string value, depending on the context in which it is used:

```
mysql> SELECT IFNULL(1,0);
-> 1
mysql> SELECT IFNULL(NULL,10);
-> 10
mysql> SELECT IFNULL(1/0,10);
-> 10
mysql> SELECT IFNULL(1/0,'yes');
-> 'yes'
```

Null handling can be very counter intuitive and could cause problems if you have an incorrect function in a delete statement that returns null. For example the following query will delete all entries.

```
DELETE FROM my_table WHERE field > NULL (or function returning NULL)
```

If you want to have NULL values presented last when doing an ORDER BY, try this:

```
SELECT * FROM my_table ORDER BY ISNULL(field), field [ ASC | DESC ]
```

Language/Operators

MySQL uses some standard SQL operators and some non-standard operators. They can be used to write expressions which involve constant values, variables, values contained in fields and / or other expressions.

Comparison operators

Equality

If you want to check if 2 values are equal, you must use the = operator:

```
SELECT True = True -- returns 1
SELECT True = False -- returns 0
```

If you want to check if 2 values are different, you can use the <> or != operators, which have the same meaning:

```
SELECT True <> False -- returns 1
SELECT True != True -- returns 0
```

<> return 1 where = returns 0 and vice versa.

IS and NULL-safe comparison

When you compare a NULL value with a non-NULL value, you'll get NULL. If you want to check if a value is null, you can use IS:

```
SELECT (NULL IS NULL) -- returns 1
SELECT (1 IS NULL) -- returns 0
SELECT (True IS True) -- returns an error!
```

You can check if a value is non-NULL:

```
SELECT (True IS NOT NULL) -- returns 1
```

There is also an equality operator which considers NULL as a normal value, so it returns 1 (not NULL) if both values are NULL and returns 0 (not NULL) if one of the values is NULL:

```
SELECT NULL <=> NULL -- 1
SELECT True <=> True -- 1
SELECT col1 <=> col2 FROM myTable
```

There is not a NULL-safe non-equality operator, but you can type the following:

```
SELECT NOT (col1 <=> col2) FROM myTable
```

IS and Boolean comparisons

IS and IS NOT can also be used for Boolean comparisons. You can use them with the reserved words TRUE, FALSE and UNKNOWN (which is merely a synonym for NULL).

```
SELECT 1 IS TRUE -- returns 1
SELECT 1 IS NOT TRUE -- returns 0
SELECT 1 IS FALSE -- returns 0
SELECT (NULL IS NOT FALSE) -- returns 1: unknown is not false
SELECT (NULL IS UNKOWN) -- returns 1
SELECT (NULL IS NOT UNKNOWN) -- returns 0
```

Greater, Less...

You can check if a value is greater than another value:

```
SELECT 100 > 0 -- returns 1
SELECT 4 > 5 -- return 0
```

You can also check if a value is minor than another value:

```
SELECT 1 < 2 -- returns 1
SELECT 2 < 2 -- returns 0
```

This kind of comparisons also works on TEXT values:

```
SELECT 'a' < 'b' -- returns 1
```

Generally speaking, alphabetical order is used for TEXT comparisons. However, the exact rules are defined by the COLLATION used. A COLLATION defines the sorting rules for a given CHARACTER SET. For example, a COLLATION may be case-sensitive, while another COLLATION may be case-insensitive.

You can check if a value is equal or greater than another value. For example, the following queries have the same meaning:

```
SELECT `a` >= `b` FROM `myTable`
SELECT NOT (`a` < `b`) FROM `myTable`
```

Similarly, you can check if a value is less or equal to another value:

```
SELECT `a` <= `b` FROM `myTable`
```

BETWEEN

If you want to check if a value is included in a given range (boundaries included), you can use the BETWEEN ... AND ... operator. AND doesn't have its usual meaning. Example:

```
SELECT 2 BETWEEN 10 AND 100    -- 0
SELECT 10 BETWEEN 10 AND 100   -- 1
SELECT 20 BETWEEN 10 AND 100   -- 1
```

The value after BETWEEN and the value after AND are included in the range.

You can also use NOT BETWEEN to check if a value is not included in a range:

```
SELECT 8 NOT BETWEEN 5 AND 10 -- returns 0
```

IN

You can use the IN operator to check if a value is included in a list of values:

```
SELECT 5 IN (5, 6, 7) -- returns 1
SELECT 1 IN (5, 6, 7) -- returns 0
```

You should not include in the list both numbers and strings, or the results may be unpredictable. If you have numbers, you should quote them:

```
SELECT 4 IN ('a', 'z', '5')
```

There is not a theoretical limit to the number of values included in the IN operator.

You can also use NOT IN:

```
SELECT 1 NOT IN (1, 2, 3) -- returns 0
```

Logical operators

MySQL Boolean logic

MySQL doesn't have a real BOOLEAN datatype.

FALSE is a synonym for 0. Empty strings are considered as FALSE in a Boolean context.

TRUE is a synonym for 1. All non-NULL and non-FALSE data are considered as TRUE in a boolean context.

UNKNOWN is a synonym for NULL. The special date 0/0/0 is NULL.

NOT

NOT is the only operator which has only one operand. It returns 0 if the operand is TRUE, returns 1 if the operand is FALSE and returns NULL if the operand is NULL.

```
SELECT NOT 1 -- returns 0
SELECT NOT FALSE -- returns 1
SELECT NOT NULL -- returns NULL
SELECT NOT UNKNOWN -- returns NULL
```

! is a synonym for NOT.

```
SELECT !1
```

AND

AND returns 1 if both the operands are TRUE, else returns 0; if at least one of the operands is NULL, returns NULL.

```
SELECT 1 AND 1 -- returns 1
SELECT 1 AND '' -- return 0
SELECT '' AND NULL -- returns NULL
```

&& is a synonym for AND.

```
SELECT 1 && 1
```

OR

OR returns TRUE if at least one of the operands is TRUE, else returns FALSE; if the two operands are NULL, returns NULL.

```
SELECT TRUE OR FALSE -- returns 1
SELECT 1 OR 1 -- returns 1
SELECT FALSE OR FALSE -- returns 0
SELECT NULL OR TRUE -- returns NULL
```

|| is a synonym for OR.

```
SELECT 1 || 0
```

XOR

XOR (eXclusive OR) returns 1 if only one of the operands is TRUE and the other operand is FALSE; returns 0 if both the operands are TRUE o both the operands are FALSE; returns NULL if one of the operands is NULL.

```
SELECT 1 XOR 0 -- returns 1
SELECT FALSE XOR TRUE -- returns 1
SELECT 1 XOR TRUE -- returns 0
SELECT 0 XOR FALSE -- returns 0
SELECT NULL XOR 1 -- returns NULL
```

Synonyms

- AND can be written as &&
- OR can be written ad ||
- NOT can be written as !

Only NOT (usually) has a different precedence from its synonym. See operator precedence for detail.

Arithmetic operators

MySQL supports operands which perform all basic arithmetic operations.

You can type positive values with a '+', if you want:

```
SELECT +1 -- return 1
```

You can type negative values with a '-'. - is an inversion operand:

```
SELECT -1 -- returns -1
SELECT -+1 -- returns -1
SELECT --1 -- returns 1
```

You can make sums with '+':

```
SELECT 1 + 1 -- returns 2
```

You can make subtractions with '-':

```
SELECT True - 1 -- returns 0
```

You can multiply a number with '*':

```
SELECT 1 * 1 -- returns 1
```

You can make divisions with '/'. Returns a FLOAT number:

```
SELECT 10 / 2 -- returns 5.0000
SELECT 1 / 1 -- returns 1.0000
SELECT 1 / 0 -- returns NULL (not an error)
```

You can make integer divisions with DIV. Resulting number is an INTEGER. No reminder.

```
SELECT 10 DIV 3 -- returns 3
```

You can get the reminder of a division with '%' or MOD:

```
SELECT 10 MOD 3 -- returns 1
```

Using + to cast data

You can convert an INTEGER to a FLOAT doing so:

```
SELECT 1 + 0.0 -- returns 1.0
SELECT 1 + 0.000 -- returns 1.000
SELECT TRUE + 0.000 -- returns 1.000
```

You can't convert a string to a FLOAT value by adding 0.0, but you can cast it to an INTEGER:

```
SELECT '1' + 0 -- returns 1
SELECT '1' + FALSE -- returns 1
```

```
SELECT <nowiki>''</nowiki> + <nowiki>''</nowiki> -- returns 0
```

Text operators

There are no concatenation operators in MySQL.

Arithmetic operators convert the values into numbers and then perform arithmetic operations, so you can't use + to concatenate strings.

You can use the CONCAT() function instead.

LIKE

The LIKE operator may be used to check if a string matches to a pattern. A simple example:

```
SELECT * FROM articles WHERE title LIKE 'hello world'
```

The pattern matching is usually case insensitive. There are two exceptions:

- when a LIKE comparison is performed against a column which has been declared with the BINARY flag (see CREATE TABLE);
- when the expression contains the BINARY clause:

```
SELECT * 'test' LIKE BINARY 'TEST' -- returns 0
```

You can use two special characters for LIKE comparisons:

- **_** means "any character" (but must be 1 char, not 0 or 2)
- **%** means "any sequence of chars" (even 0 chars or 1000 chars)

Note that "\" also escapes quotes ("'") and this behaviour can't be changed by the ESCAPE clause. Also, the escape character does not escape itself.

Common uses of LIKE:

- Find titles starting with the word "hello":

```
SELECT * FROM articles WHERE title LIKE 'hello%'
```

- Find titles ending with the word "world":

```
SELECT * FROM articles WHERE title LIKE '%world'
```

- Find titles containing the word "gnu":

```
SELECT * FROM articles WHERE title LIKE '%gnu%'
```

These special chars may be contained in the pattern itself: for example, you could need to search for the "_" character. In that case, you need to "escape" the char:

```sql
SELECT * FROM articles WHERE title LIKE '\_%' -- titles starting with _
SELECT * FROM articles WHERE title LIKE '\%%' -- titles starting with %
```

Sometimes, you may want to use an escape character different from "\". For example, you could use "/":

```sql
SELECT * FROM articles WHERE title LIKE '/_%' ESCAPE '/'
```

When you use = operator, trailing spaces are ignored. When you use LIKE, they are taken into account.

```sql
SELECT 'word' = 'word ' -- returns 1
SELECT 'word' LIKE 'word ' -- returns 0
```

LIKE also works with numbers.

```sql
SELECT 123 LIKE '%2%' -- returns 1
```

If you want to check if a pattern doesn't match, you can use NOT LIKE:

```sql
SELECT 'a' NOT LIKE 'b' -- returns 1
```

SOUNDS LIKE

You can use SOUNDS LIKE to check if 2 text values are pronounced in the same way. SOUNDS LIKE uses the SOUNDEX algorithm, which is based on English rules and is very approximate (but simple and thus fast).

```sql
SELECT `word1` SOUNDS LIKE `word2` FROM `wordList` -- short form
SELECT SOUNDEX(`word1`) = SOUNDEX(`word2`) FROM `wordList` -- long form
```

SOUNDS LIKE is a MySQL-specific extension to SQL. It has been added in MySQL 4.1.

Regular expressions

You can use REGEXP to check if a string matches to a pattern using regular expressions.

```sql
SELECT 'string' REGEXP 'pattern'
```

You can use RLIKE as a synonym for REGEXP.

Bitwise operators

Bit-NOT:

```
SELECT ~0 -- returns 18446744073709551615
SELECT ~1 -- returns 18446744073709551614
```

Bit-AND:

```
SELECT 1 & 1 -- returns 1
SELECT 1 & 3 -- returns 1
SELECT 2 & 3 -- returns 2
```

Bit-OR:

```
SELECT 1 | 0 -- returns 1
SELECT 3 | 0 -- returns 3
SELECT 4 | 2 -- returns 6
```

Bit-XOR:

```
SELECT 1 ^ 0 -- returns 1
SELECT 1 ^ 1 -- returns 0
SELECT 3 ^ 1 -- returns 2
```

Left shift:

```
SELECT 1 << 2 -- returns 4
```

Right shift:

```
SELECT 1 >> 2 -- 0
```

Conditions

The structure `IF ... THEN ... ELSE ... END IF;` only functions in the stored procedures. To manage a condition out of them, we can use:

1. `IF(condition, ifTrue, ifFalse);`
2. `SELECT CASE WHEN condition THEN ifTrue ELSE ifFalse END;`

Precedence

Operator precedence

Table of operator precedence:

```
INTERVAL
BINARY, COLLATE
!
- (unary minus), ~ (unary bit inversion)
^
*, /, DIV, %, MOD
-, +
<<, >>
&
|
=, <=>, >=, >, <=, <, <>, !=, IS, LIKE, REGEXP, IN
BETWEEN, CASE, WHEN, THEN, ELSE
NOT
&&, AND
XOR
||, OR
:=
```

Modifiers:

- **PIPES_AS_CONCAT** - If this SQL mode is enabled, || has precedence on ^, but - and ~ have precedence on ||.
- **HIGH_NOT_PRECEDENCE** - If this SQL mode is enabled, NOT has the same precedence level as !.

Use of parenthesis

You can use parenthesis to force MySQL to evaluate a subexpression before another independently from operator precedence:

```
SELECT (1 + 1) * 5 -- returns 10
```

You can also use parenthesis to make an expression more readable by humans, even if they don't affect the precedence:

```
SELECT 1 + (2 * 5) -- the same as 1 + 2 * 5
```

Assignment operators

You can use the = operator to assign a value to a column:

```
UPDATE `myTable` SET `uselessField`=0
```

When you want to assign a value to a variable, you must use the := operator, because the use of = would be ambiguous (is it as assignment or a comparison?)

```
SELECT @myvar := 1
```

You can also use SELECT INTO to assign values to one or more variables.