DATABASE TUTORIAL

What's a database ?

A database is a collection of data organized in a particular way.
Databases can be of many types such as Flat File Databases, Relational Databases, Distributed Databases etc.

What's SQL ?

In 1971, IBM researchers created a simple non-procedural language called Structured English Query Language. or SEQUEL. This was based on Dr. Edgar F. (Ted) Codd's design of a relational model for data storage where he described a universal programming language for accessing databases.

In the late 80's ANSI and ISO (these are two organizations dealing with standards for a wide variety of things) came out with a standardized version called Structured Query Language or SQL. SQL is prounced as 'Sequel'. There have been several versions of SQL and the latest one is SQL-99. Though SQL-92 is the current universally adopted standard.

SQL is the language used to query all databases. It's simple to learn and appears to do very little but is the heart of a successful database application. Understanding SQL and using it efficiently is highly imperative in designing an efficient database application. The better your understanding of SQL the more versatile you'll be in getting information out of databases.

What's an RDBMS ?

This concept was first described around 1970 by Dr. Edgar F. Codd in an IBM research publication called "System R4 Relational".

A relational database uses the concept of linked two-dimensional tables which comprise of rows and columns. A user can draw relationships between multiple tables and present the output as a table again. A user of a relational database need not understand the representation of data in order to retrieve it. Relational programming is non-procedural.

[***What's procedural and non-procedural ?***

Programming languages are procedural if they use programming elements such as conditional statements (if-then-else, do-while etc.). SQL has none of these types of statements.]

In 1979, Relational Software released the world's first relational database called Oracle V.2

What a DBMS ?

MySQL and mSQL are database management systems or DBMS. These software packages are used to manipulate a database. All DBMSs use their own implementation of SQL. It may be a subset or a superset of the instructions provided by SQL 92.

MySQL, due to it's simplicity uses a subset of SQL 92 (also known as SQL2).

What's Database Normalization ?

Normalization is the process where a database is designed in a way that removes redundancies, and increases the clarity in organizing data in a database.

In easy English, it means take similar stuff out of a collection of data and place them into tables. Keep doing this for each new table recursively and you'll have a Normalized database. From this resultant database you should be able to recreate the data into it's original state if there is a need to do so.

The important thing here is to know when to Normalize and when to be practical. That will come with experience. For now, read on...

Normalization of a database helps in modifying the design at later times and helps in being prepared if a change is required in the database design. Normalization raises the efficiency of the datatabase in terms of management, data storage and scalability.

Now Normalization of a Database is achieved by following a set of rules called 'forms' in creating the database.

These rules are 5 in number (with one extra one stuck in-between 3&4) and they are:

1st Normal Form or 1NF:

Each Column Type is Unique.

2nd Normal Form or 2NF:

The entity under consideration should already be in the 1NF and all attributes within the entity should depend solely on the entity's unique identifier.

3rd Normal Form or 3NF:

The entity should already be in the 2NF and no column entry should be dependent on any other entry (value) other than the key for the table.
If such an entity exists, move it outside into a new table.

Now if these 3NF are achieved, the database is considered normalized. But there are three more 'extended' NF for the elitist.

These are:

BCNF (Boyce & Codd):

The database should be in 3NF and all tables can have only one primary key.

4NF:

Tables cannot have multi-valued dependencies on a Primary Key.

5NF:

There should be no cyclic dependencies in a composite key.

Well this is a highly simplified explanation for Database Normalization. One can study this process extensively though. After working with databases for some time you'll automatically create Normalized databases. As, it's logical and practical.

For now, don't worry too much about Normalization. The quickest way to grasp SQL and Databases is to plunge headlong into creating tables and start messing around with SQL statements. After you go through the tutorial examples and also the example contacts database, look at the example provided in creating a normalized database near the very end of this tutorial. And then try to think how you would like to create your own database.

Much of database design depends on how YOU want to keep the data. In real life situations often you may find it more convenient to store data in tables designed in a way that does fall a bit short of keeping all the NFs happy. But that's what databases are all about. Making your life simpler.

Onto SQL

There are four basic commands which are the workhorses for SQL and figure in almost all queries to a database.

INSERT - Insert Data

DELETE - Delete Data

SELECT - Pull Data

UPDATE - Change existing Data

As you can see SQL is like English.

Let's build a real world example database using MySQL and perform some SQL operations on it.

A database that practically anyone could use would be a Contacts database.

In our example we are going to create create a database with the following fields:

- FirstName
- LastName
- BirthDate
- StreetAddress
- City
- State
- Zip

- Country
- TelephoneHome
- TelephoneWork
- Email
- CompanyName
- Designation

First, lets decide how we are going to store this data in the database. For illustration purposes, we are going to keep this data in multiple tables.

This will let us exercise all of the SQL commands pertaining to retrieving data from multiple tables. Also we can separate different kinds of entities into different tables. So let's say you have thousands of friends and need to send a mass email to all of them, a SELECT statement (covered later) will look at only one table.

Well, we can keep the FirstName, LastName and BirthDate in one table.
Address related data in another.
Company Details in another.
Emails can be separated into another.
Telephones can be separated into another.

Let's build the database in MySQL.

While building a database - you need to understand the concept of data types. Data types allow the user to define how data is stored in fields or cells within a database. It's a way to define how your data will actually exist. Whether it's a Date or a string consisting of 20 characters, an integer etc. When we build tables within a database we also define the contents of each field in each row in the table using a data type. It's imperative that you use only the data type that fits your needs and don't use a data type that reserves more memory than the data in the field actually requires.

Let's look at various Data Types under MySQL.

Type	Size in bytes	Description
TINYINT (length)	**1**	Integer with unsigned range of 0-255 and a signed range from -128-127
SMALLINT (length)	**2**	Integer with unsigned range of 0-65535 and a signed range from -32768-32767
MEDIUMINT(length)	**3**	Integer with unsigned range of 0-16777215 and a signed range from -8388608-8388607
INT(length)	**4**	Integer with unsigned range of 0-429467295 and a signed range from -2147483648-2147483647
BIGINT(length)	**8**	Integer with unsigned range of 0-18446744 and a signed range from -9223372036854775808-9223372036854775807
FLOAT(length, decimal)	**4**	Floating point number with max. value +/- 3.402823466E38 and min.(non-zero) value +/11.175494351E-38
DOUBLEPRECISION(length, decimal)	**8**	Floating point number with max. value +/- -1.7976931348623157E308 and min. (non-zero)

		value +/-2.2250738585072014E-308
DECIMAL(length, decimal)	**length**	Floating point number with the range of the DOUBLE type that is stored as a CHAR field type.
TIMESTAMP(length)	**4**	YYYYMMDDHHMMSS or YYMMDDHHMMSS or YYYYMMDD, YYMMDD. A Timestamp value is updated each time the row changes value. A NULL value sets the field to the current time.
DATE	**3**	YYYY-MM-DD
TIME	**3**	HH:MM:DD
DATETIME	**8**	YYYY-MM-DD HH:MM:SS
YEAR	**1**	YYYY or YY
CHAR(length)	**length**	A fixed length text string where fields shorter than the assigned length are filled with trailing spaces.
VARCHAR(length)	**length**	A fixed length text string (255 Character Max) where unused trailing spaces are removed before storing.
TINYTEXT	**length+1**	A text field with max. length of 255 characters.
TINYBLOB	**length+1**	A binary field with max. length of 255 characters.
TEXT	**length+1**	64Kb of text
BLOB	**length+1**	64Kb of data
MEDIUMTEXT	**length+3**	16Mb of text
MEDIUMBLOB	**length+3**	16 Mb of data
LONGTEXT	**length+4**	4GB of text
LONGBLOB	**length+4**	4GB of data
ENUM	**1,2**	This field can contain one of a possible 65535 number of options. Ex: ENUM('abc','def','ghi')
SET	**1-8**	This type of field can contain any number of a set of predefined possible values.

The following examples will make things quite clear on declaring Data Types within SQL statements.

Steps in Creating the Database using MySQL

From the shell prompt (either in DOS or UNIX):

mysqladmin create contacts;

This will create an empty database called "contacts".

Now run the command line tool "mysql" and from the mysql prompt do the following:

mysql> use contacts;

(You'll get the response "Database changed")

The following commands entered into the MySQL prompt will create the tables in the database.

mysql> CREATE TABLE names (contact_id SMALLINT NOT NULL AUTO_INCREMENT PRIMARY KEY, FirstName CHAR(20), LastName CHAR(20), BirthDate DATE);

mysql> CREATE TABLE address(contact_id SMALLINT NOT NULL PRIMARY KEY, StreetAddress CHAR(50), City CHAR(20), State CHAR(20), Zip CHAR(15), Country CHAR(20));

mysql> CREATE TABLE telephones (contact_id SMALLINT NOT NULL PRIMARY KEY, TelephoneHome CHAR(20), TelephoneWork(20));

mysql> CREATE TABLE email (contact_id SMALLINT NOT NULL PRIMARY KEY, Email CHAR(20));

mysql> CREATE TABLE company_details (contact_id SMALLINT NOT NULL PRIMARY KEY, CompanyName CHAR(25), Designation CHAR(15));

Note: *Here we assume that one person will have only one email address. Now if there were a situation where one person has multiple addresses, this design would be a problem. We'd need another field which would keep values that indicated to whom the email address belonged to. In this particular case email data ownership is indicated by the primary key. The same is true for telephones. We are assuming that one person has only one home telephone and one work telephone number. This need not be true. Similarly one person could work for multiple companies at the same time holding two different designation. In all these cases an extra field will solve the issue. For now however let's work with this small design.*

KEYS:

The relationships between columns located in different tables are usually described through the use of keys.

As you can see we have a PRIMARY KEY in each table. The Primary key serves as a mechanism to refer to other fields within the same row. In this case, the Primary key is used to identify a relationship between a row under consideration and the person whose name is located inside the 'names' table. We use the AUTO_INCREMENT statement only for the 'names' table as we need to use the generated contact_id number in all the other tables for identification of the rows.

This type of table design where one table establishes a relationship with several other tables is known as a **'one to many'**relationship.
In a **'many to many'** relationship we could have several Auto Incremented Primary Keys in various tables with several inter-relationships.

Foreign Key:

A foreign key is a field in a table which is also the Primary Key in another table. This is known commonly as 'referential integrity'.

Execute the following commands to see the newly created tables and their contents.

To see the tables inside the database:

```
mysql> SHOW TABLES;
+----------------------+
| Tables in contacts |
+----------------------+
| address |
| company_details |
| email |
| names |
| telephones |
+----------------------+
5 rows in set (0.00 sec)
```

To see the columns within a particular table:

```
mysql>SHOW COLUMNS FROM address;
+--------------+------------+------+-----+---------+-------+-------------------------------+
| Field | Type | Null | Key | Default | Extra | Privileges
|
+--------------+------------+------+-----+---------+-------+-------------------------------+
| contact_id | smallint(6) | | PRI | 0 | | select,insert,update,references |
| StreetAddress | char(50) | YES | | NULL | | select,insert,update,references |
| City | char(20) | YES | | NULL | | select,insert,update,references |
| State | char(20) | YES | | NULL | | select,insert,update,references |
| Zip | char(10) | YES | | NULL | | select,insert,update,references |
| Country | char(20) | YES | | NULL | | select,insert,update,references |
+--------------+------------+------+-----+---------+-------+-------------------------------+
6 rows in set (0.00 sec)
```

So we have the tables created and ready. Now we put in some data.
Let's start with the 'names' table as it uses a unique AUTO_INCREMENT field which in turn is used in the other tables.

Inserting data, one row at a time:

```
mysql> INSERT INTO names (FirstName, LastName, BirthDate) VALUES ('Yamila','Diaz ','1974-10-13');
Query OK, 1 row affected (0.00 sec)
```

Inserting multiple rows at a time:

```
mysql> INSERT INTO names (FirstName, LastName, BirthDate) VALUES ('Nikki','Taylor','1972-03-
04'),('Tia','Carrera','1975-09-18');
Query OK, 2 rows affected (0.00 sec)
Records: 2 Duplicates: 0 Warnings: 0
```

Let's see what the data looks like inside the table. We use the SELECT command for this.

```
mysql> SELECT * from NAMES;
+------------+-----------+----------+------------+
| contact_id | FirstName | LastName | BirthDate |
+------------+-----------+----------+------------+
| 3 | Tia | Carrera | 1975-09-18 |
| 2 | Nikki | Taylor | 1972-03-04 |
| 1 | Yamila | Diaz | 1974-10-13 |
+------------+-----------+----------+------------+
3 rows in set (0.06 sec)
```

Try another handy command called 'DESCRIBE'.

```
mysql> DESCRIBE names;
+------------+------------+------+-----+---------+---------------+--------------------------------+
| Field | Type | Null | Key | Default | Extra | Privileges
|
+------------+------------+------+-----+---------+---------------+--------------------------------+
| contact_id | smallint(6) | | PRI | NULL | auto_increment | select,insert,update,references |
| FirstName | char(20) | YES | | NULL | | select,insert,update,references |
| LastName | char(20) | YES | | NULL | | select,insert,update,references |
| BirthDate | date | YES | | NULL | | select,insert,update,references |
+------------+------------+------+-----+---------+---------------+--------------------------------+
4 rows in set (0.00 sec)
```

Now lets populate the other tables. Observer the syntax used.

```
mysql> INSERT INTO address(contact_id, StreetAddress, City, State, Zip, Country) VALUES ('1', '300
Yamila Ave.', 'Los Angeles', 'CA', '300012', 'USA'),('2','4000 Nikki St.','Boca
Raton','FL','500034','USA'),('3','404 Tia Blvd.','New York','NY','10011','USA');
Query OK, 3 rows affected (0.05 sec)
Records: 3 Duplicates: 0 Warnings: 0

mysql> SELECT * FROM address;
+------------+----------------+------------+-------+--------+---------+
| contact_id | StreetAddress | City | State | Zip | Country |
+------------+----------------+------------+-------+--------+---------+
| 1 | 300 Yamila Ave. | Los Angeles | CA | 300012 | USA |
| 2 | 4000 Nikki St. | Boca Raton | FL | 500034 | USA |
| 3 | 404 Tia Blvd. | New York | NY | 10011 | USA |
+------------+----------------+------------+-------+--------+---------+
3 rows in set (0.00 sec)

mysql> INSERT INTO company_details (contact_id, CompanyName, Designation) VALUES
('1','Xerox','New Business Manager'), ('2','Cabletron','Customer Support Eng'),('3','Apple','Sales
Manager');
Query OK, 3 rows affected (0.05 sec)
Records: 3 Duplicates: 0 Warnings: 0

mysql> SELECT * FROM company_details;
+------------+-------------+---------------------+
| contact_id | CompanyName | Designation |
+------------+-------------+---------------------+
| 1 | Xerox | New Business Manager |
| 2 | Cabletron | Customer Support Eng |
| 3 | Apple | Sales Manager |
+------------+-------------+---------------------+
3 rows in set (0.06 sec)

mysql> INSERT INTO email (contact_id, Email) VALUES ('1', 'yamila@yamila.com'),( '2',
'nikki@nikki.com'),('3','tia@tia.com');
Query OK, 3 rows affected (0.00 sec)
Records: 3 Duplicates: 0 Warnings: 0

mysql> SELECT * FROM email;
+------------+------------------+
| contact_id | Email |
+------------+------------------+
| 1 | yamila@yamila.com |
| 2 | nikki@nikki.com |
| 3 | tia@tia.com |
+------------+------------------+
3 rows in set (0.06 sec)
```

```
mysql> INSERT INTO telephones (contact_id, TelephoneHome, TelephoneWork) VALUES ('1','333-
50000','333-60000'),('2','444-70000','444-80000'),('3','555-30000','55 5-40000');
Query OK, 3 rows affected (0.00 sec)
Records: 3 Duplicates: 0 Warnings: 0

mysql> SELECT * FROM telephones;
+------------+--------------+--------------+
| contact_id | TelephoneHome | TelephoneWork |
+------------+--------------+--------------+
| 1 | 333-50000 | 333-60000 |
| 2 | 444-70000 | 444-80000 |
| 3 | 555-30000 | 555-40000 |
+------------+--------------+--------------+
3 rows in set (0.00 sec)
```

Okay, so we now have all our data ready for experimentation.

Before we start experimenting with manipulating the data let's look at how MySQL stores the Data.

To do this execute the following command from the shell prompt.

mysqldump contacts > contacts.sql

Note: The reverse operation for this command is:

mysql contacts < contacts.sql

The file generated is a text file that contains all the data and SQL instruction needed to recreate the same database. As you can see, the SQL here is slightly different than what was typed in. Don't worry about this. It's all good ! It would also be obvious that this is a good way to backup your stuff.

```
# MySQL dump 8.2
#
# Host: localhost Database: contacts
#--------------------------------------------------------
# Server version 3.22.34-shareware-debug

#
# Table structure for table 'address'
#

CREATE TABLE address (
contact_id smallint(6) DEFAULT '0' NOT NULL,
StreetAddress char(50),
City char(20),
State char(20),
Zip char(10),
Country char(20),
PRIMARY KEY (contact_id)
);

#
# Dumping data for table 'address'
#
```

```
INSERT INTO address VALUES (1,'300 Yamila Ave.','Los Angeles','CA','300012','USA');
INSERT INTO address VALUES (2,'4000 Nikki St.','Boca Raton','FL','500034','USA');
INSERT INTO address VALUES (3,'404 Tia Blvd.','New York','NY','10011','USA');

#
# Table structure for table 'company_details'
#

CREATE TABLE company_details (
contact_id smallint(6) DEFAULT '0' NOT NULL,
CompanyName char(25),
Designation char(20),
PRIMARY KEY (contact_id)
);

#
# Dumping data for table 'company_details'
#

INSERT INTO company_details VALUES (1,'Xerox','New Business Manager');
INSERT INTO company_details VALUES (2,'Cabletron','Customer Support Eng');
INSERT INTO company_details VALUES (3,'Apple','Sales Manager');

#
# Table structure for table 'email'
#

CREATE TABLE email (
contact_id smallint(6) DEFAULT '0' NOT NULL,
Email char(20),
PRIMARY KEY (contact_id)
);

#
# Dumping data for table 'email'
#

INSERT INTO email VALUES (1,'yamila@yamila.com');
INSERT INTO email VALUES (2,'nikki@nikki.com');
INSERT INTO email VALUES (3,'tia@tia.com');

#
# Table structure for table 'names'
#

CREATE TABLE names (
contact_id smallint(6) DEFAULT '0' NOT NULL auto_increment,
FirstName char(20),
LastName char(20),
BirthDate date,
PRIMARY KEY (contact_id)
);

#
# Dumping data for table 'names'
#

INSERT INTO names VALUES (3,'Tia','Carrera','1975-09-18');
INSERT INTO names VALUES (2,'Nikki','Taylor','1972-03-04');
INSERT INTO names VALUES (1,'Yamila','Diaz','1974-10-13');
```

```
#
# Table structure for table 'telephones'
#

CREATE TABLE telephones (
contact_id smallint(6) DEFAULT '0' NOT NULL,
TelephoneHome char(20),
TelephoneWork char(20),
PRIMARY KEY (contact_id)
);

#
# Dumping data for table 'telephones'
#

INSERT INTO telephones VALUES (1,'333-50000','333-60000');
INSERT INTO telephones VALUES (2,'444-70000','444-80000');
INSERT INTO telephones VALUES (3,'555-30000','555-40000');
```

Let's try some SELECT statement variations:

To select all names whose corresponding contact_id is greater than 1.

```
mysql> SELECT * FROM names WHERE contact_id > 1;
+------------+-----------+----------+------------+
| contact_id | FirstName | LastName | BirthDate |
+------------+-----------+----------+------------+
| 3 | Tia | Carrera | 1975-09-18 |
| 2 | Nikki | Taylor | 1972-03-04 |
+------------+-----------+----------+------------+
2 rows in set (0.00 sec)
```

As a condition we can also use NOT NULL. This statement will return all names where there exists a contact_id.

```
mysql> SELECT * FROM names WHERE contact_id IS NOT NULL;
+------------+-----------+----------+------------+
| contact_id | FirstName | LastName | BirthDate |
+------------+-----------+----------+------------+
| 3 | Tia | Carrera | 1975-09-18 |
| 2 | Nikki | Taylor | 1972-03-04 |
| 1 | Yamila | Diaz | 1974-10-13 |
+------------+-----------+----------+------------+
3 rows in set (0.06 sec)
```

Result's can be arranged in a particular way using the statement ORDER BY.

```
mysql> SELECT * FROM names WHERE contact_id IS NOT NULL ORDER BY LastName;
+------------+-----------+----------+------------+
| contact_id | FirstName | LastName | BirthDate |
+------------+-----------+----------+------------+
| 3 | Tia | Carrera | 1975-09-18 |
| 1 | Yamila | Diaz | 1974-10-13 |
| 2 | Nikki | Taylor | 1972-03-04 |
+------------+-----------+----------+------------+
3 rows in set (0.06 sec)
```

'asc' and 'desc' stand for ascending and descending respectively and can be used to arrange the results.

```
mysql> SELECT * FROM names WHERE contact_id IS NOT NULL ORDER BY LastName desc;
+------------+-----------+----------+------------+
| contact_id | FirstName | LastName | BirthDate |
+------------+-----------+----------+------------+
| 2 | Nikki | Taylor | 1972-03-04 |
| 1 | Yamila | Diaz | 1974-10-13 |
| 3 | Tia | Carrera | 1975-09-18 |
+------------+-----------+----------+------------+
3 rows in set (0.04 sec)
```

You can also place date types into conditional statements.

```
mysql> SELECT * FROM names WHERE BirthDate > '1973-03-06';
+------------+-----------+----------+------------+
| contact_id | FirstName | LastName | BirthDate |
+------------+-----------+----------+------------+
| 3 | Tia | Carrera | 1975-09-18 |
| 1 | Yamila | Diaz | 1974-10-13 |
+------------+-----------+----------+------------+

2 rows in set (0.00 sec)
```

LIKE is a statement to match field values using wildcards. The % sign is used for denoting wildcards and can represent multiple characters.

```
mysql> SELECT FirstName, LastName FROM names WHERE LastName LIKE 'C%';
+-----------+----------+
| FirstName | LastName |
+-----------+----------+
| Tia | Carrera |
+-----------+----------+
1 row in set (0.06 sec)
```

'_' is used to represent a single wildcard.

```
mysql> SELECT FirstName, LastName FROM names WHERE LastName LIKE '_iaz';
+-----------+----------+
| FirstName | LastName |
+-----------+----------+
| Yamila | Diaz |
+-----------+----------+
1 row in set (0.00 sec)
```

SQL Logical Operations (operates from Left to Right)

1.NOT or !

2. AND or &&

3. OR or ||

4. = : Equal

5. <> or != : Not Equal

6. <=

7. >=

8 <,>

Here are some more variations with Logical Operators and using the 'IN' statement.

```
mysql> SELECT FirstName FROM names WHERE contact_id < 3 AND LastName LIKE 'D%';
+-----------+
| FirstName |
+-----------+
| Yamila |
+-----------+
1 row in set (0.00 sec)

mysql> SELECT contact_id FROM names WHERE LastName IN ('Diaz','Carrera');
+------------+
| contact_id |
+------------+
| 3 |
| 1 |
+------------+
2 rows in set (0.02 sec)
```

To return the number of rows in a table

```
mysql> SELECT count(*) FROM names;
+----------+
| count(*) |
+----------+
| 3 |
+----------+
1 row in set (0.02 sec)

mysql> SELECT count(FirstName) FROM names;
+-----------------+
| count(FirstName) |
+-----------------+
| 3 |
+-----------------+
1 row in set (0.00 sec)
```

To do some basic arithmetic aggregate functions.

```
mysql> SELECT SUM(contact_id) FROM names;
+----------------+
| SUM(contact_id) |
+----------------+
| 6 |
+----------------+
1 row in set (0.00 sec)
```

To select a largest value from a row. Substitute 'MIN' and see what happens next.

```
mysql> SELECT MAX(contact_id) FROM names;
+----------------+
| MAX(contact_id) |
+----------------+
```

```
| 3 |
+----------------+
1 row in set (0.00 sec)
```

HAVING

Take a look at the first query using the statement WHERE and the second statement using the statement HAVING.

```
mysql> SELECT * FROM names WHERE contact_id >=1;
+------------+-----------+----------+------------+
| contact_id | FirstName | LastName | BirthDate |
+------------+-----------+----------+------------+
| 1 | Yamila | Diaz | 1974-10-13 |
| 2 | Nikki | Taylor | 1972-03-04 |
| 3 | Tia | Carrera | 1975-09-18 |
+------------+-----------+----------+------------+
3 rows in set (0.03 sec)

mysql> SELECT * FROM names HAVING contact_id >=1;
+------------+-----------+----------+------------+
| contact_id | FirstName | LastName | BirthDate |
+------------+-----------+----------+------------+
| 3 | Tia | Carrera | 1975-09-18 |
| 2 | Nikki | Taylor | 1972-03-04 |
| 1 | Yamila | Diaz | 1974-10-13 |
+------------+-----------+----------+------------+
3 rows in set (0.00 sec)
```

Now lets work with multiple tables and see how information can be pulled out of the data.

```
mysql> SELECT names.contact_id, FirstName, LastName, Email FROM names, email WHERE
names.contact_id = email.contact_id;
+------------+-----------+----------+------------------+
| contact_id | FirstName | LastName | Email |
+------------+-----------+----------+------------------+
| 1 | Yamila | Diaz | yamila@yamila.com |
| 2 | Nikki | Taylor | nikki@nikki.com |
| 3 | Tia | Carrera | tia@tia.com |
+------------+-----------+----------+------------------+
3 rows in set (0.11 sec)

mysql> SELECT DISTINCT names.contact_id, FirstName, Email, TelephoneWork FROM names, email,
telephones WHERE names.contact_id=email.contact_id=telephones.contact_id;
+------------+-----------+------------------+--------------+
| contact_id | FirstName | Email | TelephoneWork |
+------------+-----------+------------------+--------------+
| 1 | Yamila | yamila@yamila.com | 333-60000 |
| 2 | Nikki | nikki@nikki.com | 333-60000 |
| 3 | Tia | tia@tia.com | 333-60000 |
+------------+-----------+------------------+--------------+


3 rows in set (0.05 sec)
```

JOIN

JOIN is the action performed on multiple tables that returns a result as a table. It's what makes a database 'relational'.

There are several types of joins. Let's look at LEFT JOIN (OUTER JOIN) and RIGHT JOIN

Let's first check out the contents of the tables we're going to use

```
mysql> SELECT * FROM names;
+------------+-----------+----------+------------+
| contact_id | FirstName | LastName | BirthDate |
+------------+-----------+----------+------------+
| 3 | Tia | Carrera | 1975-09-18 |
| 2 | Nikki | Taylor | 1972-03-04 |
| 1 | Yamila | Diaz | 1974-10-13 |
+------------+-----------+----------+------------+
3 rows in set (0.00 sec)

mysql> SELECT * FROM email;
+------------+-------------------+
| contact_id | Email |
+------------+-------------------+
| 1 | yamila@yamila.com |
| 2 | nikki@nikki.com |
| 3 | tia@tia.com |
+------------+-------------------+
3 rows in set (0.00 sec)
```

A LEFT JOIN First:

```
mysql> SELECT * FROM names LEFT JOIN email USING (contact_id);
+------------+-----------+----------+------------+------------+-------------------+
| contact_id | FirstName | LastName | BirthDate | contact_id | Email|
+------------+-----------+----------+------------+------------+-------------------+
| 3 | Tia | Carrera | 1975-09-18 | 3 | tia@tia.com |
| 2 | Nikki | Taylor | 1972-03-04 | 2 | nikki@nikki.com |
| 1 | Yamila | Diaz | 1974-10-13 | 1 | yamila@yamila.com |
+------------+-----------+----------+------------+------------+-------------------+
3 rows in set (0.16 sec)
```

To find the people who have a home phone number.

```
mysql> SELECT names.FirstName FROM names LEFT JOIN telephones ON names.contact_id =
telephones.contact_id WHERE TelephoneHome IS NOT NULL;
+-----------+
| FirstName |
+-----------+
| Tia |
| Nikki |
| Yamila |
+-----------+
3 rows in set (0.02 sec)
```

These same query leaving out 'names' (from names.FirstName) is still the same and will generate the same result.

```
mysql> SELECT FirstName FROM names LEFT JOIN telephones ON names.contact_id =
telephones.contact_id WHERE TelephoneHome IS NOT NULL;
```

```
+-----------+
| FirstName |
+-----------+
| Tia |
| Nikki |
| Yamila |
+-----------+
3 rows in set (0.00 sec)
```

And now a RIGHT JOIN:

```
mysql> SELECT * FROM names RIGHT JOIN email USING(contact_id);
+------------+-----------+----------+------------+------------+-----------------+
| contact_id | FirstName | LastName | BirthDate | contact_id | Email |
+------------+-----------+----------+------------+------------+-----------------+
| 1 | Yamila | Diaz | 1974-10-13 | 1 | yamila@yamila.com |
| 2 | Nikki | Taylor | 1972-03-04 | 2 | nikki@nikki.com
|
| 3 | Tia | Carrera | 1975-09-18 | 3 | tia@tia.com
|
+------------+-----------+----------+------------+------------+----------------
--+
3 rows in set (0.03 sec)
```

BETWEEN

This conditional statement is used to select data where a certain related contraint falls between a certain range of values. The following example illustrates it's use.

```
mysql> SELECT * FROM names;
+------------+-----------+----------+------------+
| contact_id | FirstName | LastName | BirthDate |
+------------+-----------+----------+------------+
| 3 | Tia | Carrera | 1975-09-18 |
| 2 | Nikki | Taylor | 1972-03-04 |
| 1 | Yamila | Diaz | 1974-10-13 |
+------------+-----------+----------+------------+
3 rows in set (0.06 sec)
```

```
mysql> SELECT FirstName, LastName FROM names WHERE contact_id BETWEEN 2 AND 3;
+-----------+----------+
| FirstName | LastName |
+-----------+----------+
| Tia | Carrera |
| Nikki | Taylor |
+-----------+----------+
2 rows in set (0.00 sec)
```

ALTER

The ALTER statement is used to add a new column to an existing table or to make changes to it.

```
mysql> ALTER TABLE names ADD Age SMALLINT;
Query OK, 3 rows affected (0.11 sec)
Records: 3 Duplicates: 0 Warnings: 0
```

Now let's take a look at the 'ALTER'ed Table.

```
mysql> SHOW COLUMNS FROM names;
+------------+------------+------+-----+---------+----------------+
| Field | Type | Null | Key | Default | Extra |
+------------+------------+------+-----+---------+----------------+
| contact_id | smallint(6) | | PRI | 0 | auto_increment |
| FirstName | char(20) | YES | | NULL | |
| LastName | char(20) | YES | | NULL | |
| BirthDate | date | YES | | NULL | |
| Age | smallint(6) | YES | | NULL | |
+------------+------------+------+-----+---------+----------------+
5 rows in set (0.06 sec)
```

But we don't require Age to be a SMALLINT type when a TINYINT would suffice. So we use another ALTER statement.

```
mysql> ALTER TABLE names CHANGE COLUMN Age Age TINYINT;
Query OK, 3 rows affected (0.02 sec)
Records: 3 Duplicates: 0 Warnings: 0

mysql> SHOW COLUMNS FROM names;
+------------+------------+------+-----+---------+--------+----------------+
| Field | Type | Null | Key | Default | Extra |
+------------+------------+------+-----+---------+--------+----------------+
| contact_id | smallint(6) | | PRI | NULL |
| FirstName | char(20) | YES | | NULL | |
| LastName | char(20) | YES | | NULL | |
| BirthDate | date | YES | | NULL | |
| Age | tinyint(4) | YES | | NULL | |
+------------+------------+------+-----+---------+--------+----------------+
5 rows in set (0.00 sec)
```

MODIFY

You can also use the MODIFY statement to change column data types.

```
mysql> ALTER TABLE names MODIFY COLUMN Age SMALLINT;
Query OK, 3 rows affected (0.03 sec)
Records: 3 Duplicates: 0 Warnings: 0

mysql> SHOW COLUMNS FROM names;
+------------+------------+------+-----+---------+----------------+----------------+
| Field | Type | Null | Key | Default | Extra |
+------------+------------+------+-----+---------+----------------+----------------+
| contact_id | smallint(6) | | PRI | NULL | auto_increment |
| FirstName | char(20) | YES | | NULL | |
| LastName | char(20) | YES | | NULL | |
| BirthDate | date | YES | | NULL | |
| Age | smallint(6) | YES | | NULL | |
+------------+------------+------+-----+---------+----------------+----------------+
5 rows in set (0.00 sec)
```

To Rename a Table:

```
mysql> ALTER TABLE names RENAME AS mynames;
Query OK, 0 rows affected (0.00 sec)

mysql> SHOW TABLES;
+--------------------+
| Tables_in_contacts |
+--------------------+
```

```
| address |
| company_details |
| email |
| mynames |
| telephones |
+------------------+
5 rows in set (0.00 sec)
```

We rename it back to the original name.

```
mysql> ALTER TABLE mynames RENAME AS names;
Query OK, 0 rows affected (0.01 sec)
```

UPDATE

The UPDATE command is used to add a value to a field in a table.

```
mysql> UPDATE names SET Age ='23' WHERE FirstName='Tia';
Query OK, 1 row affected (0.06 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

The Original Table:

```
mysql> SELECT * FROM names;
+------------+-----------+----------+------------+------+
| contact_id | FirstName | LastName | BirthDate | Age |
+------------+-----------+----------+------------+------+
| 3 | Tia | Carrera | 1975-09-18 | 23 |
| 2 | Nikki | Taylor | 1972-03-04 | NULL |
| 1 | Yamila | Diaz | 1974-10-13 | NULL |
+------------+-----------+----------+------------+------+
3 rows in set (0.05 sec)
```

The Modified Table:

```
mysql> SELECT * FROM names;
+------------+-----------+----------+------------+------+
| contact_id | FirstName | LastName | BirthDate | Age |
+------------+-----------+----------+------------+------+
| 3 | Tia | Carrera | 1975-09-18 | 24 |
| 2 | Nikki | Taylor | 1972-03-04 | NULL |
| 1 | Yamila | Diaz | 1974-10-13 | NULL |
+------------+-----------+----------+------------+------+
3 rows in set (0.00 sec)
```

DELETE

```
mysql> DELETE FROM names WHERE Age=23;
Query OK, 1 row affected (0.00 sec)
```

```
mysql> SELECT * FROM names;
+------------+-----------+----------+------------+------+
| contact_id | FirstName | LastName | BirthDate | Age |
+------------+-----------+----------+------------+------+
| 2 | Nikki | Taylor | 1972-03-04 | NULL |
| 1 | Yamila | Diaz | 1974-10-13 | NULL |
+------------+-----------+----------+------------+------+
2 rows in set (0.00 sec)
```

A DEADLY MISTAKE...

```
mysql> DELETE FROM names;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM names;
Empty set (0.00 sec)
```

One more destructive tool...

DROP TABLE

```
mysql> DROP TABLE names;
Query OK, 0 rows affected (0.00 sec)

mysql> SHOW TABLES;
+-------------------+
| Tables in contacts |
+-------------------+
| address |
| company_details |
| email |
| telephones |
+-------------------+
4 rows in set (0.05 sec)

mysql> DROP TABLE address ,company_details, telephones;
Query OK, 0 rows affected (0.06 sec)

mysql> SHOW TABLES;
Empty set (0.00 sec)
```

As you can see, the table 'names' no longer exists. MySQL does not give a warning so be careful.

FULL TEXT INDEXING and Searching

Since version 3.23.23, Full Text Indexing and Searching has been introduced into MySQL. FULLTEXT indexes can be created from VARCHAR and TEXT columns. FULLTEXT searches are performed with the MATCH function. The MATCH function matches a natural language query on a text collection and from each row in a table it returns relevance.The resultant rows are organized in order of relevance.

Full Text searches are a very powerful way to search through text. But is not ideal for small tables of text and may produce inconsistent results. Ideally it works with large collections of textual data.

Optimizing your Database

Well, databases do tend to get large at some or the other. And here arises the issue of database optimization. Queries are going to take longer and longer as the database grows and certain things can be done to speed things up.

Clustering

The easiest method is that of 'clustering'. Suppose you do a certain kind of query often, it would be faster if the database contents were arranged in a in the same way data was requested. To keep the tables in a sorted order you need a clustering index. Some databases keep stuff sorted automatically.

Ordered Indices

These are a kind of 'lookup' tables of sorts. For each column that may be of interest to you, you can create an ordered index.
It needs to be noted that again these kinds of optimization techniques produce a system load in terms of creating a new index each time the data is re-arranged.

There are additional method such as B-Trees, Hashing which you may like to read up about but will not be discussed here.

Replication

Replication is the term given to the process where databases synchronize with each other. In this process one database updates it's own data with respect to another or with reference to certain criteria for updates specified by the programmer. Replication can be used under various circumstances. Examples may be : safety and backup, to provide a closer location to the database for certain users.

What are Transactions ?

In an RDBMS, when several people access the same data or if a server dies in the middle of an update, there has to be a mechanism to protect the integrity of the data. Such a mechanism is called a Transaction. A transaction groups a set of database actions into a single instantaneous event. This event can either succeed or fail. i.e .either get the job done or fail.

The definition of a transaction can be provided by an Acronym called 'ACID'.

(A)tomicity: If an action consists of multiple steps - it's still considered as one operation.

(C) Consistency: The database exists in a valid and accurate operating state before and after a transaction.

(I) Isolation: Processes within one transaction are independent and cannot interfere with that in others.

(D) Durability: Changes affected by a transaction are permanent.

To enable transactions a mechanism called 'Logging' needs to be introduced. Logging involves a DBMS writing details on the tables, columns and results of a particular

transaction, both before and after, onto a log file. This log file is used in the process of recovery. Now to protect a certain database resource (ex. a table) from being used and written onto simulatneously several techniques are used. One of them is 'Locking' another is to put a 'time stamp' onto an action. In the case of Locking, to complete an action, the DBMS would need to acquire locks on all resources needed to complete the action. The locks are released only when the transaction is completed.

Now if there were say a large numbers of tables involved in a particular action, say 50, all 50 tables would be locked till a transaction is completed.

To improve things a bit, there is another technique used called 2 Phase Locking or 2PL. In this method of locking, locks are acquired only when needed but are released only when the transaction is completed.

This is done to make sure that that altered data can be safely restored if the transaction fails for any reason.

This technique can also result in problems such as "deadlocks".

In this case - 2 processes requiring the same resources lock each other up by preventing the other to complete an action. Options here are to abort one, or let the programmer handle it.

MySQL implements transactions by implementing the Berkeley DB libraries into its own code. So it's the source version you'd want here for MySQL installation. Read the MySQL manual on implementing this.

What are Views ?

A view allows you to assign the result of a query to a new private table. This table is given the name used in your VIEW query.
Although MySQL does not support views yet a sample SQL VIEW construct statement would look like:

CREATE VIEW TESTVIEW AS SELECT * FROM names;

What are Triggers ?

A trigger is a pre-programmed notification that performs a set of actions that may be commonly required. Triggers can be programmed to execute certain actions before or after an event occurs. Triggers are very useful as they they increase efficiency and accuracy in performing operations on databases and also are increase productivity by reducing the time for application development. Triggers however do carry a price in terms of processing overhead.

What are Procedures ?

Like triggers, Procedures or 'Stored' Procedures are productivity enhancers. Suppose you needed to perform an action using a programming interface to the database in say PERL and ASP. If a programmed action could be stored at the database level, it's obvious that it has to be written only once and cam be called by any programming language interacting with the database.

Procedures are executed using triggers.

Beyond RDBMS

Distributed Databases (DDB)

A distributed database is a collection of several, logically interrelated database located at multiple locations of a computer network. A distributed database management system permits the management of such a database and makes the operation transparent to the user. Good examples of distributed databases would be those utilized by banks, multinational firms with several office locations where each distributed data system works only with the data that is relevant to it's operations. DDBs have have full functionality of any DBMS. It's also important to know that the distributed databases are considered to be actually one database rather than discrete files and data within distributed databases are logically interrelated.

Object Database Management Systems or ODBMS

When the capabilities of a database are integrated with object programming language capababilities, the resulting product is an ODBMS. Database objects appear as programming objects in an ODBMS. Using an ODBMS offers several advantages. The ones that can be most readily appreciated are:

1. Efficiency

When you use an ODBMS, you're using data the way you store it. You will use less code as you're not dependent on an intermediary like SQL or ODBC. When this happens you can create highly complex data structures through your programming language.

2. Speed

When data is stored the way you'd like it to be stored (i.e. natively) there is a massive performance increase as no to-and-fro translation is required.

A Quick Tutorial on Database Normalization

Let's start off by taking some data represented in a Table.

Table Name: College Table

StudentName	CourseID1	CourseTitle1	CourseProfessor1	CourseID2	CourseTitle2	CourseProfessor2	StudentAdvisor	StudentID
Tia Carrera	CS123	Perl Regular Expressions	Don Corleone	CS003	Object Oriented Programming 1	Daffy Duck	Fred Flintstone	400
John Wayne	CS456	Socket Programming	DJ Tiesto	CS004	Algorithms	Homer Simpson	Barney Rubble	401

Lara Croft	CS789	OpenGL	Bill Clinton	CS001	Data Structures	Papa Smurf	Seven of Nine	402

(text size has been shrunk to aid printability on one page)

The First Normal Form: (Each Column Type is Unique and there are no repeating groups [types] of data)

This essentially means that you indentify data that can exist as a separate table and therefore reduce repetition and will reduce the width of the original table.

We can see that for every student, Course Information is repeated for each course. So if a student has three course, you'll need to add another set of columns for Course Title, Course Professor and CourseID. So Student information and Course Information can be considered to be two broad groups.

Table Name: Student Information
StudentID (Primary Key)
StudentName
AdvisorName

Table Name: Course Information
CourseID (Primary Key)
CourseTitle
CourseDescription
CourseProfessor

It's obvious that we have here a Many to Many relationship between Students and Courses.

Note: In a Many to Many relationship we need something called a relating table which basically contains information exclusively on which relatioships exist between two tables. In a One to Many relationship we use a foreign key.

So in this case we need another little table called: **Students and Courses**

Table Name: Students and Courses
SnCStudentID
SnCCourseID

The Second Normal Form: (All attributes within the entity should depend solely on the entity's unique identifier)

The AdvisorName under Student Information does not depend on the StudentID. Therefore it can be moved to it's own table.

Table Name: Student

Information
StudentID (Primary Key) StudentName

Table Name: Advisor Information
AdvisorID AdvisorName

Table Name: Course Information
CourseID (Primary Key) CourseTitle CourseDescription CourseProfessor

Table Name: Students and Courses
SnCStudentID SnCCourseID

Note: Relating Tables can be created as required.

The Third Normal Form:(no column entry should be dependent on any other entry (value) other than the key for the table)

In simple terms - a table should contain information about only one thing.

In Course Information, we can pull CourseProfessor information out and store it in another table.

Table Name: Student Information
StudentID (Primary Key) StudentName

Table Name: Advisor Information
AdvisorID AdvisorName

Table Name: Course Information
CourseID (Primary Key) CourseTitle CourseDescription

Table Name: Professor Information
ProfessorID CourseProfessor

Table Name: Students and Courses
SnCStudentID SnCCourseID

Note: Relating Tables can be created as required.

Well that's it. One you are done with 3NF the database is considered Normalized.

Now lets consider some cases where normalization would have to avoided for practical purposes.

Suppose we needed to store a students home address along with State and Zip Code information. Would you create a separate table for every zipcode in your country along with one for cities and one for states ? It actually depends on you. I would prefer just using a non-normalized address table and stick everything in there. So exceptions crop up often and it's up to your better judgement.