**Create a Database:**

**After Logged Into MySql Terminal:**

create database trialDatabase;

Where, trialDatabase is the database name.

**Without Logged Into MySql Terminal:**

**If authentication is not enabled:**

mysqladmin create trialDatabase2

Where, trialDatabase2 is the database name.

**If authentication is enabled:**

mysqladmin -u root -p create trialDatabase2

It will prompt for password.

Just enter the password.

**Show All The Databases Created:**show databases;

**Drop Database:**

Suppose, a database named trialDatabase needs to be dropped.

**From MySql Terminal:**

drop database trialDatabase

**Without Logged Into MySql Terminal:**

**If authentication is not enabled:**

mysqladmin drop trialDatabase2

Where, trialDatabase2 is the database name.

**If authentication is enabled:**

mysqladmin -u root -p drop trialDatabase2

It will prompt for password.

Just enter the password.

**Choose/ Select A DataBase:**use trialDatabase;

**DataTypes In MySql:**

MySql uses all the standard ANSI SQL numeric data types. It could be really helpful if an user starts using mysql after he/she used other database system for a long time. The following list shows the common numeric data type and their descriptions:

* **INT:** A normal-sized integer that can be signed or unsigned. If signed, the allowable range is from -231 to 231 -1. If, unsigned , the allowable range is from 0 to 232 .

Note: How to use unsigned int datatype?

Example: create table example\_tbl(id INT UNSIGNED NOT NULL AUTO\_INCREMENT, title VARCHAR(40) NOT NULL, author VARCHAR(40) NOT NULL, PRIMARY KEY(id));  
  
That is a query example where UNSIGNED INT is used.

* **TINYINT:** A very small integer that can be signed or unsigned. If signed, the allowable range is from -128 to 127. If unsigned, the allowable range is from 0 to 255
* **SMALLINT:** A small integer that can be signed or unsigned. If signed, the allowable range is from -32768 to 32767. (-215 to 215-1). If unsigned, the allowable range is from 0 to 65535. (0 to 216-1).
* **MEDIUMINT:** A medium-sized integer that can be signed or unsigned. If signed, the allowable range is from -8388608 to 8388607. (-223 to 223-1). If unsigned, the allowable range is from 0 to 16777215. (0 to 224)
* **BIGINT:** A large integer that can be signed or unsigned. If signed, the allowable range is from-9223372036854775808 to 9223372036854775807. (-263 to 263-1). If unsigned, the allowable range is from 0 to 18446744073709551615. (264)
* **FLOAT(M,D)** − A floating-point number that cannot be unsigned. You can define the display length (M) and the number of decimals (D). This is not required and will default to 10,2, where 2 is the number of decimals and 10 is the total number of digits (including decimals). Decimal precision can go to 24 places for a FLOAT. (Default is 10,2, where 2 is the number of decimals and 10 is the total number of digits. That means, 8 digits before decimal point and 2 digits after decimal point).
* **DOUBLE(M,D)** − A double precision floating-point number that cannot be unsigned. You can define the display length (M) and the number of decimals (D). This is not required and will default to 16,4, where 4 is the number of decimals. Decimal precision can go to 53 places for a DOUBLE. REAL is a synonym for DOUBLE.
* **DECIMAL(M,D)-** An unpacked floating-point number that cannot be unsigned. In the unpacked decimals, each decimal corresponds to one byte. Defining the display length (M) and the number of decimals (D) is required. NUMERIC is a synonym for DECIMAL. (Now, this means, M and D always required to be specified if DECIMAL datatype is to be used.

**Now, difference between FLOAT(M,D), DOUBLE(M,D) and DECIMAL(M,D):**

If you understand the difference between DOUBLE(M,D) and DECIMAL(M,D) :

Main issue is: DOUBLE causes rounding issues (Not actually rounding/truncating issue, but precision issues) . And if you do something like 0.1 + 0.2 it gives you something like 0.30000000000000004. So, if one of your table has a currency related field, imagine the mess it could create.

However, difference between FLOAT and DOUBLE is precision. FLOAT is for single digit precision whereas, double is for multiple digit precision. FLOAT is accurate to approximately 7 decimal places, and DOUBLE up to 14.

Decimal’s declaration and functioning is similar to Double. But there is one big difference between floating point values and decimal (numeric) values. We use DECIMAL data type to store exact numeric values, where we do not want precision but exact and accurate values. A Decimal type can store a Maximum of ****65 Digits****, with ****30 digits**** after decimal point.

**Date and Time Types:**

* **DATE** − A date in YYYY-MM-DD format, between 1000-01-01 and 9999-12-31. For example, December 30th, 1973 would be stored as 1973-12-30.
* **DATETIME** − A date and time combination in YYYY-MM-DD HH:MM:SS format, between 1000-01-01 00:00:00 and 9999-12-31 23:59:59. For example, 3:30 in the afternoon on December 30th, 1973 would be stored as 1973-12-30 15:30:00.
* **TIMESTAMP** − A timestamp between midnight, January 1st, 1970 and sometime in 2037. This looks like the previous DATETIME format, only without the hyphens between numbers; 3:30 in the afternoon on December 30th, 1973 would be stored as 19731230153000 ( YYYYMMDDHHMMSS ).
* **TIME** − Stores the time in a HH:MM:SS format.
* **YEAR(M)** − Stores a year in a 2-digit or a 4-digit format.

**String Datatypes:**

* **CHAR(M)** − A fixed-length string between 1 and 255 characters in length (for example CHAR(5)), right-padded with spaces to the specified length when stored. Defining a length is not required, but the default is 1.
* **VARCHAR(M)** − A variable-length string between 1 and 255 characters in length. For example, VARCHAR(25). You must define a length when creating a VARCHAR field.

**Difference Between CHAR(M) and VARCHAR(M):**

VARCHAR is variable-length whereas CHAR is fixed length.

If your content is a fixed size, you'll get better performance with CHAR.

Like, suppose, you want to store MD5 hash, MD5 hash has always 32 characters. Therefore to maximize your performance use CHAR(32) since CHAR is fixed length.

When, it is required to choose between CHAR and VACHAR the above issue is always important. Also, there is a trade off between performance and memory allocation.

For fixed length, CHAR is always better than VARCHAR in terms of performance.

Whereas, for variable length VARCHAR is always better than CHAR is terms of memory optimization.

* **BLOB or TEXT** − A field with a maximum length of 65535 characters (0 to 216-1). BLOBs are "Binary Large Objects" and are used to store large amounts of binary data, such as images or other types of files. Fields defined as TEXT also hold large amounts of data. The difference between the two is that the sorts and comparisons on the stored data are **case sensitive** on BLOBs and are **not case sensitive** in TEXT fields. You do not specify a length with BLOB or TEXT. (216-1)
* **TINYBLOB or TINYTEXT** − A BLOB or TEXT column with a maximum length of 255 characters. You do not specify a length with TINYBLOB or TINYTEXT. (0 to 28-1)
* **MEDIUMBLOB or MEDIUMTEXT** − A BLOB or TEXT column with a maximum length of 16777215 characters. You do not specify a length with MEDIUMBLOB or MEDIUMTEXT. (0 to 224-1).
* **LONGBLOB or LONGTEXT** − A BLOB or TEXT column with a maximum length of 4294967295 characters. You do not specify a length with LONGBLOB or LONGTEXT. (232-1)
* **ENUM** − An enumeration, which is a fancy term for list. When defining an ENUM, you are creating a list of items from which the value must be selected (or it can be NULL). For example, if you wanted your field to contain "A" or "B" or "C", you would define your ENUM as ENUM ('A', 'B', 'C') and only those values (or NULL) could ever populate that field.

**Note: BLOB or Text field cannot be set as primary key.**

An example where you can use enum variable:

CREATE TABLE shirts

(

name VARCHAR(40),

size ENUM('x-small', 'small', 'medium', 'large', 'x-large')

);

INSERT INTO shirts (name, size) VALUES ('dress shirt','large'), ('t-shirt','medium'),

('polo shirt','small');

**Now, perform a select query:**

SELECT name, size FROM shirts WHERE size = 'medium';

|  |  |
| --- | --- |
| Name | size |
| t-shirt | medium |

**And, to perform an update query:**

UPDATE shirts SET size = 'small' WHERE size = 'large';

COMMIT;

**Operators in MySql: (as per mysql 5.7)**

|  |  |
| --- | --- |
| [BETWEEN ... AND ...](https://dev.mysql.com/doc/refman/5.7/en/comparison-operators.html" \l "operator_between) | Check whether a value is within a range of values |
| [COALESCE()](https://dev.mysql.com/doc/refman/5.7/en/comparison-operators.html" \l "function_coalesce) | Return the first non-NULL argument |
| [=](https://dev.mysql.com/doc/refman/5.7/en/comparison-operators.html" \l "operator_equal) | Equal operator |
| [<=>](https://dev.mysql.com/doc/refman/5.7/en/comparison-operators.html" \l "operator_equal-to) | NULL-safe equal to operator |
| [>](https://dev.mysql.com/doc/refman/5.7/en/comparison-operators.html" \l "operator_greater-than) | Greater than operator |
| [>=](https://dev.mysql.com/doc/refman/5.7/en/comparison-operators.html" \l "operator_greater-than-or-equal) | Greater than or equal operator |
| [GREATEST()](https://dev.mysql.com/doc/refman/5.7/en/comparison-operators.html" \l "function_greatest) | Return the largest argument |
| [IN()](https://dev.mysql.com/doc/refman/5.7/en/comparison-operators.html" \l "function_in) | Check whether a value is within a set of values |
| [INTERVAL()](https://dev.mysql.com/doc/refman/5.7/en/comparison-operators.html" \l "function_interval) | Return the index of the argument that is less than the first argument |
| [IS](https://dev.mysql.com/doc/refman/5.7/en/comparison-operators.html" \l "operator_is) | Test a value against a boolean |
| [IS NOT](https://dev.mysql.com/doc/refman/5.7/en/comparison-operators.html" \l "operator_is-not) | Test a value against a boolean |
| [IS NOT NULL](https://dev.mysql.com/doc/refman/5.7/en/comparison-operators.html" \l "operator_is-not-null) | NOT NULL value test |
| [IS NULL](https://dev.mysql.com/doc/refman/5.7/en/comparison-operators.html" \l "operator_is-null) | NULL value test |
| [ISNULL()](https://dev.mysql.com/doc/refman/5.7/en/comparison-operators.html" \l "function_isnull) | Test whether the argument is NULL |
| [LEAST()](https://dev.mysql.com/doc/refman/5.7/en/comparison-operators.html" \l "function_least) | Return the smallest argument |
| [<](https://dev.mysql.com/doc/refman/5.7/en/comparison-operators.html" \l "operator_less-than) | Less than operator |
| [<=](https://dev.mysql.com/doc/refman/5.7/en/comparison-operators.html" \l "operator_less-than-or-equal) | Less than or equal operator |
| [LIKE](https://dev.mysql.com/doc/refman/5.7/en/string-comparison-functions.html" \l "operator_like) | Simple pattern matching |
| [NOT BETWEEN ... AND ...](https://dev.mysql.com/doc/refman/5.7/en/comparison-operators.html" \l "operator_not-between) | Check whether a value is not within a range of values |
| [!=, <>](https://dev.mysql.com/doc/refman/5.7/en/comparison-operators.html" \l "operator_not-equal) | Not equal operator |
| [NOT IN()](https://dev.mysql.com/doc/refman/5.7/en/comparison-operators.html" \l "function_not-in) | Check whether a value is not within a set of values |
| [NOT LIKE](https://dev.mysql.com/doc/refman/5.7/en/string-comparison-functions.html" \l "operator_not-like) | Negation of simple pattern matching |
| [STRCMP()](https://dev.mysql.com/doc/refman/5.7/en/string-comparison-functions.html" \l "function_strcmp) | Compare two strings |

Not so known operator:

**COALESCE:**

Return the first non null argument.

Select COALESCE(NULL,1);

Now, an example is constructed which might not be very appropiate.

Suppose, you have two tables

One is user Table, which looks like following:

|  |  |
| --- | --- |
| **Name** | **Type** |
| Id | INT(11) |
| Name | VARCHAR(80) |
| Age | TINYINT |
| PhoneNumber | VARCHAR(20) |

And UserAddressMap:

|  |  |
| --- | --- |
| **Name** | **Type** |
| Id | INT(11) |
| Address | TINYBLOB |

Now, suppose, you want to display the the first non null address corresponding to user id as his/her primary address:

(

why the address will be null under any circumstance? Suppose, the table UserAddressMap have 3 addressed saved for one particular user. Now, Somehow, under some circumstances, one address can no more be claimed as that particular user’s address. Now, if instead of deleting the whole row (though deleting whole row would be much better) , we just update the address as null (just consider the scenario to make that example happen), we need to make a query about the first non null address.

)

select \*,COALESCE(select b.Address from user.a, userAddressMap.b where b.Id=a.Id) from user;

Example:

**Create the user table:**

create table user(id INT(11) NOT NULL AUTO\_INCREMENT,Name VARCHAR(80) NOT NULL, Age TINYINT NOT NULL,PhoneNumber VARCHAR(20),PRIMARY KEY(id));

**Create the userAddressMap:**

create table userAddressMap(id INT(11) NOT NULL,Address TINYBLOB);

(No primary key. By the way, the id field cannot be set as primary key. And, Even id, Address combo cannot be set as composite primary key. Since, Type of Address is BLOB)

**Insert Into User Table:**

insert into user(id,Name,Age,PhoneNumber)VALUES(1,"Sayak Haldar",23,"9674465435");

**Insert Into userAddressMap:**

insert into userAddressMap(id,Address)VALUES(1,"j-2,102/B, DDA Flats, Kalkaji:-110019");

insert into userAddressMap(id,Address)VALUES(1,"j-4,106/B, DDA Flats, Kalkaji, Delhi-110019");

insert into userAddressMap(id,Address)VALUES(1,"D-185, Okhla Phase 1, Reve Systems, Delhi-110020");

But, COALESCE does not work like that.

COALESCE actually works on Columns.

Check the following example:

**Create the table as the following:**

create table newUser(id INT(11) NOT NULL AUTO\_INCREMENT, name VARCHAR(80) NOT NULL, age TINYINT NOT NULL, phoneNumber VARCHAR(20) NOT NULL, address1 TINYBLOB, address2 TINYBLOB, PRIMARY KEY(id));

**Insert into The Table:**

insert into newUser(id,name,age,phoneNumber,address1,address2)VALUES(1,"Sayak Haldar",23,"9674465435",NULL,"j-2,102/B, DDA Flats, Kalkaji, New Delhi-110019");

**Use Of COALESCE During Fetching Of The Value:**

select id as ID, name as Name, age as Age, phoneNumber as PhoneNumber, COALESCE(address1,address2) as PrimaryAddress from newUser;

This will work.

**Interval:**

Returns 0 if N < N1, 1 if N < N2 and so on or -1 if N is NULL. All arguments are treated as integers. It is required that N1 < N2 < N3 < ... < Nn for this function to work correctly. This is because a binary search is used (very fast).

mysql> SELECT INTERVAL(23, 1, 15, 17, 30, 44, 200);

-> 3

mysql> SELECT INTERVAL(10, 1, 10, 100, 1000);

-> 2

mysql> SELECT INTERVAL(22, 23, 30, 44, 200);

-> 0

**Logical Operators:**

|  |  |
| --- | --- |
| **Name** | **Description** |
| AND, && | Logical And |
| Not, ! | Negates Value |
| ||, OR | Logical Or |
| XOR | Logical XOR |

**Assignment Operator:**

|  |  |
| --- | --- |
| **Name** | **Description** |
| [=](https://dev.mysql.com/doc/refman/5.7/en/assignment-operators.html" \l "operator_assign-equal) | Assign a value (as part of a SET statement, or as part of the SET clause in an UPDATE statement) |
| := | Assigns a value |

**Difference between = and :=**

Unlike [=](https://dev.mysql.com/doc/refman/5.7/en/assignment-operators.html" \l "operator_assign-equal), the [:=](https://dev.mysql.com/doc/refman/5.7/en/assignment-operators.html" \l "operator_assign-value) operator is never interpreted as a comparison operator. This means you can use [:=](https://dev.mysql.com/doc/refman/5.7/en/assignment-operators.html" \l "operator_assign-value) in any valid SQL statement (not just in [SET](https://dev.mysql.com/doc/refman/5.7/en/set-variable.html" \o "13.7.4.1 SET Syntax for Variable Assignment) statements) to assign a value to a variable.

**Create Table:**

**Generic Syntax To Create A MySql Table:**

CREATE TABLE table\_name (column\_name column\_type);

Check the following syntax:

**create table forumAnswers(answer\_id INT NOT NULL AUTO\_INCREMENT, answer\_content BLOB NOT NULL, author VARCHAR(80), posted\_date DATETIME, PRIMARY KEY(answer\_id));**

Field Attribute **NOT NULL** is being used because we do not want this field to be NULL. So, if a user will try to create a record with a NULL value, then MySQL will raise an error.

Field Attribute **AUTO\_INCREMENT** tells MySQL to go ahead and add the next available number to the id field.

Keyword **PRIMARY KEY** is used to define a column as a primary key. You can use multiple columns separated by a comma to define a primary key.

**Some Other Field Attribute/Column Attribute:**

**UNSIGNED:** Use of it is already explained. However, it is again explained here.  it is possible to declare integer type columns as UNSIGNED. An unsigned integer type allows only non-negative values to be represented but increases the range of possible positive values.

**ZEROFILL:** The ZEROFILL column attribute can be used with any numeric column type. Whenever displaying a value of a column with ZEROFILL specified, the output will be padded with leading zeros up to the display width of the column. It can be used together with an explicit declaration of the display length of the column—for example:

column\_name INT(M) ZEROFILL

where M is the display width of an INT column, or like this:

column\_name FLOAT(M,D) ZEROFILL

**Default:** The default column attribute can be used. (is Default a column attribute? ) to set the default value.

Check the following syntax:

Suppose, you are creating an app and you give user some free credit as a token. (in a fixed app defined currency) Now, A default value is to be given at the time of insertion. (is pretty much required in this case).

So, check the following example for that:

**create Table UserBalance(UserId INT(11) NOT NULL,Balance DOUBLE NOT NULL DEFAULT 10000.00, PRIMARY KEY(UserId,Balance));**

Now, the table is displayed:

**show columns from UserBalance;**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Field** | **Type** | **Null** | **Key** | **Default** | **Extra** |
| UserId | INT(11) | No | Pri | NULL |  |
| Balance | Double | No | Pri | 10000 |  |

**Drop A Table:**

Drop table table\_name;

**Inserting In A Table:**

INSERT INTO table\_name ( field1, field2,...fieldN ) VALUES( value1, value2,...valueN );

Remember the table created with the following command:

**create table forumAnswers(answer\_id INT NOT NULL AUTO\_INCREMENT, answer\_content BLOB NOT NULL, author VARCHAR(80), posted\_date DATETIME);**

Now, let’s insert some data in that table:

**insert**

**into forumAnswers(answer\_content,posted\_date)VALUES("Savitar is a time remnant of Barry himself",now());**

Now, if the entry is to be seen:

select \* from forumAnswers;

|  |  |  |  |
| --- | --- | --- | --- |
| Answer\_id | Answer\_content | Author | Posted\_Date |
| 1 | Savitar is a time remnant of Barry himself | NULL | 2017-05-19 02:44:27 |

Note: now() is a sql function which returns the current date and time.

**Fetch Informations From A Table:**

**Generic Select Syntax:**

SELECT field1, field2,...fieldN table\_name1, table\_name2...

[WHERE Clause] [OFFSET M ][LIMIT N]

**DELETE Information From A Table:**

**Generic Delete Syntax:**

DELETE FROM table\_name [WHERE Clause]

**Update Information From A Table:**

UPDATE table\_name SET field1 = new-value1, field2 = new-value2 [WHERE Clause]

**Where Clause:**

**Generic syntax for where clause is:**

SELECT field1, field2,...fieldN table\_name1, table\_name2...

[WHERE condition1 [AND [OR]] condition2..…

Now, more than one condition can be applied to where clause using AND or OR operator.

Also, the following logical operators can be used in the condition part of where clause:

|  |  |
| --- | --- |
| Name | Description |
| AND,&& | LOGICAL AND |
| Not, ! | Negates Value |
| ||,OR | Logical Or |
| XOR | Logical XOR |

(these are logical operators according to MySQL 5.7 Reference Manual)

* **WHERE Clause For Fetch Informations From A Table:**

**Generic syntax for where clause is the following:**

SELECT field1, field2,...fieldN table\_name1, table\_name2...

[WHERE condition1 [AND [OR]] condition2..…

Now, as you can see, you can use more than one clause for where condition using AND or OR operator.

You can use any operator (mentioned before) with where clause.

**Like Clause:**

Like is previously listed as Comparison Operator.

SELECT field1, field2,...fieldN table\_name1, table\_name2...

WHERE field1 LIKE condition1 [AND [OR]] filed2 = 'somevalue'.

Sorting Results:

**Generic Syntax:**

SELECT field1, field2,...fieldN table\_name1, table\_name2...

ORDER BY field: You can sort the returned result on any field, if that field is being listed out.

You can sort the result on more than one field.

You can use the keyword ASC or DESC to get result in ascending or descending order. By default, it's the ascending order.

Now, sorting on the basis of numeric datatypes is easy. (I mean types like INT, TINYINT, MEDIUMINT, BIGINT, FLOAT, DOUBLE etc) However, sorting based on ENUM is fancy.

For that, you need to know about index values for enumeration literals:

(

Now, it has already been discussed how to create a table with enumeration, insert in a table with enumeration, delete some rows based on enumeration values, update some rows based on enumeration values

**This is an example of creating a table with a enum datatype field: (from mysql reference page)**

CREATE TABLE shirts (

name VARCHAR(40),

size ENUM('x-small', 'small', 'medium', 'large', 'x-large')

);

**This is an example of inserting in a table which contain a enum datatype field :**

INSERT INTO shirts (name, size) VALUES ('dress shirt','large'), ('t-shirt','medium'),('polo shirt','small');

Now, here, you should notice the pattern to insert multiple rows in the table at once.

**This is an example of fetching rows in a table which contain a enum datatype field:**

SELECT name, size FROM shirts WHERE size = 'medium';

|  |  |
| --- | --- |
| **Name** | **Size** |
| t-shirt | medium |

UPDATE shirts SET size = 'small' WHERE size = 'large';

COMMIT;

**And some important notes:**

Trailing spaces are automatically deleted from enum number values in the table definition when a table is created.

When retrieved, values stored into an ENUM column are displayed using the lettercase that was used in the column definition. Note that ENUM columns can be assigned a character set and collation. For binary or case-sensitive collations, lettercase is taken into account when assigning values to the column.

It is also recommended that *not* to use numbers as enumeration values, because it does not save on storage over the appropriate [TINYINT](https://dev.mysql.com/doc/refman/5.7/en/integer-types.html" \o "11.2.1 Integer Types (Exact Value) - INTEGER, INT, SMALLINT, TINYINT, MEDIUMINT, BIGINT) or [SMALLINT](https://dev.mysql.com/doc/refman/5.7/en/integer-types.html" \o "11.2.1 Integer Types (Exact Value) - INTEGER, INT, SMALLINT, TINYINT, MEDIUMINT, BIGINT) type, and it is easy to mix up the strings and the underlying number values (which might not be the same) if you quote the ENUM values incorrectly. If you do use a number as an enumeration value, always enclose it in quotation marks. If the quotation marks are omitted, the number is regarded as an index.

)

Now, every enumeration literal has an index. Sorting is actually happened based on that. Now, some things which are to be mentioned about enumeration literal:

* The elements listed as as enumeration values are assigned index numbers, beginning with 1.
* The index value of the empty string error value is 0. This means that you can use the following SELECT statement to find rows into which invalid ENUM values were assigned:

**SELECT \* FROM tbl\_name WHERE enum\_col=0;**

If enum\_col is the column name assigned to t

he column having the enum field.

* The index of the NULL value is NULL.
* The term “index” here refers to a position within the list of enumeration values. It has nothing to do with table indexes.

For example, the enumeration indices for a column specified as ENUM('Mercury', 'Venus', 'Earth') is illustrated below:

|  |  |
| --- | --- |
| **Value** | **Index** |
| NULL | NULL |
| ‘’ | 0 |
| ‘Mercury’ | 1 |
| ‘Venus’ | 2 |
| ‘Earth’ | 3 |

Now, based on that index value, sorting is performed. (and that index number is completely depended on the order in which the enumeration was listed in the column specification.)

**Join Query:**

Here, in this document, the basic join example will be provided.

Check the following example:

Suppose we have two tables, name of first table is Person and name of Second table is PersonPhoneNumberMap.

Now, first table is created in the following way:

create table Person(id INT(11) NOT NULL AUTO\_INCREMENT, name VARCHAR(80) NOT NULL, age TINYINT NOT NULL,PRIMARY KEY(id));

And Second table is created in the following way:

**create table PersonPhoneNumberMap(id INT(11) NOT NULL, PhoneNumber VARCHAR(20), PRIMARY KEY(id,PhoneNumber));**

Now, Insertion queries in the both tables are given below:

**insert into Person(id,name,age)VALUES(1,"Sayak Haldar",23);**

And,

**insert into PersonPhoneNumberMap(id,PhoneNumber)Values(1,9674465435);**

So, after insertion both table looks like following:

|  |  |  |
| --- | --- | --- |
| **Id** | **Name** | **Age** |
| 1 | Sayak Haldar | 23 |

And,

|  |  |
| --- | --- |
| **id** | **PhoneNumber** |
| 1 | 9674465435 |

**Now, select a.id,a.name,a.age,b.PhoneNumber from Person a,PersonPhoneNumberMap b where a.id=b.id;**

This query gives us the result

|  |  |  |  |
| --- | --- | --- | --- |
| **Id** | **Name** | **Age** | **PhoneNumber** |
| 1 | Sayak Haldar | 23 | 9674465435 |

This is the example of basic join query. Join query is a vast thing which cannot be covered in one example.

**Null Values:**

We have seen the SQL select command along with the where clause to fetch data from mysql table, but when we try to give a condition, which compares the field or the column value to **NULL**, it does not work properly.

To handle such a situation, MySQL provides three operators −

* **IS NULL:** the operator returns true, if the column value is NULL.
* **IS NOT NULL:** This operator returns true, if the column value is not NULL.
* <=>: This operator compares values, which (unlike the operator =) is true even fro two null values.

Why do we have three operators that deal with null values?

Because, =NULL or !=NULL cannot be used. NULL is a special case.

**Transactions:**

A transaction is a sequential group of database manipulation operations, which is performed as if it were one single unit. In other words, a transaction will never be complete unless each individual operation within the group is successful. If any operation within the transaction fails, the transaction is aborted at the point of failure and previous operations are rolled back to their formal state.

**Properties Of Transactions:**

Transactions have the following four properties, usually referred to by the acronym **ACID**:

**Atomicity:** This ensures that all operations within the work unit are completed successfully. Otherwise, the transaction is aborted at the point of failure and previous operations are rolled back to their former state.

**Consistency:** This ensures that the database properly changes state upon a successfully committed transaction.

**Isolation** − This enables transactions to operate independently on and transparent to each other.

**Durability** − This ensures that the result or effect of a committed transaction persists in case of a system failure.

In MySQL, the transactions begin with the statement **BEGIN WORK** and end with either a **COMMIT** or a **ROLLBACK** statement. The SQL commands between the beginning and ending statements form the bulk of the transaction.

**Commit And RollBack:**

* These two keywords **Commit** and **Rollback** are mainly used for MySQL Transactions.When a successful transaction is completed, the COMMIT command should be issued so that the changes to all involved tables will take effect.
* If a failure occurs, a ROLLBACK command should be issued to return every table referenced in the transaction to its previous state.

The behavior of a transaction is controlled by setting session variable called **AUTOCOMMIT**. If AUTOCOMMIT is set to 1 (the default), then each SQL statement (within a transaction or not) is considered a complete transaction and committed by default when it finishes.

When AUTOCOMMIT is set to 0, by issuing the **SET AUTOCOMMIT = 0** command, the subsequent series of statements acts like a transaction and no activities are committed until an explicit COMMIT statement is issued.

**A Generic Example Of Transaction:**

Begin transaction by issuing the sql command BEGIN WORK.

Issue one or more sql commands Like INSERT, SELECT, UPDATE or DELETE.

Check if their is no error and everything is according to your requirement.

If their is any error, issue a **rollback** command, otherwise issue a **commit** command.

**Transaction Safe Tables In MySql:**

Transactions cannot be directly used in major cases. Also, transactions are not safe and guaranteed. So, in case of using transactions in mysql programming, the tables need to be created in a special way. There are many types of tables, which support transactions, but the most popular one is InnoDB.

Support for InnoDB tables requires a specific compilation parameter when compiling mysql from the source. Now, suppose it is supported in the mysql version used by the user.

In that case, simply type=InnoDB definition is needed to be added in the table creation statement.

**create table forumAnswers(answer\_id INT NOT NULL AUTO\_INCREMENT, answer\_content BLOB NOT NULL, author VARCHAR(80), posted\_date DATETIME) type=InnoDB;**

This will work if InnoDB table is supported by mysql version used.

**Showing All Columns Of A Table:**

Now, the following can be done to see all the columns values:

**select \* from table\_name;**

However, if some one wants to see all the columns:

**Show columns from table\_name;**

**Difference between these two:**

Suppose, there is no entry is a table named Person:

**Select \* from Person will show:**

Empty set (0.00 sec)

**Now, if there’s some entry in the table:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **Name** | **Age** | **PhoneNumber** | **Address1** |
| 1 | Sayak Haldar | 23 | 9674465435 | j-2,102, DDA Flats, Kalkaji-110019 |

**Whereas, show columns from Person will result in:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Field** | **Type** | **NULL** | **Key** | **Default** | **Extra** |
| **ID** | **INT(11)** | **No** | **PRI** | **NULL** | **AUTO\_**  **INCREMENT** |
| **Name** | **VARCHAR(80)** | **No** |  | **NULL** |  |
| **Age** | **TINYINT (4)** | **No** |  | **NULL** |  |
| **PhoneNumber** | **VARCHAR(20)** | **No** | **PRI** | **NULL** |  |
| **Address** | **VARCHAR(90)** | **Yes** |  |  |  |

**Alter Command To Drop, Add or Reposition A Column:**

**Drop A Column:**

Suppose, a column of a table is dropped:

Suppose, a table is created like the following:

Create table Person(ID INT NOT NULL AUTO\_INCREMENT, Name VARCHAR(80) NOT NULL, Age TINYINT NOT NULL, PhoneNumber VARCHAR(20) NOT NULL, PRIMARY KEY(ID));

Now, certainly, it is felt that PhoneNumber is not required to maintain in this table. Rather, a separate table would be created named PersonPhoneNumberMap. So, it is needed to drop the PhoneNumber column from the table:

**alter table Person DROP COLUMN phoneNumber;**

That is to drop column.

**Add A Column:**

Suppose, the person table is created as following:

Create table Person(ID INT NOT NULL AUTO\_INCREMENT, Name VARCHAR(80) NOT NULL, Age TINYINT NOT NULL, PhoneNumber VARCHAR(20) NOT NULL, PRIMARY KEY(ID));

Now, Suppose, another column named Address of type VARCHAR(80) is required to be added.

alter table Person ADD COLUMN Address VARCHAR(80) ;

**Change A Column Definition:**

alter table Person Modify Address VARCHAR(100);

**Change A Column Name As Well As Definition:**

Alter table Person CHANGE Address Address1 VARCHAR(90);

**Setting Default Value For A Column:**

**Suppose, a table is created as following:**

create table Employee(EmployeeId INT NOT NULL AUTO\_INCREMENT, Name VARCHAR(80) NOT NULL, Age TINYINT NOT NULL, PhoneNumber VARCHAR(20) NOT NULL, Salary DOUBLE, PRIMARY KEY(EmployeeId));

Now, since, NOT NULL is not set for Salary Double, Salary field can be NULL.

Now, with alter command it can be changed, I.e. a default value could be set as salary.

**ALTER TABLE Employee MODIFY Salary DOUBLE NOT NULL DEFAULT 45000.00;**

Note that, even though datatype of the column is not modified, it is to be mentioned.

**Dropping Default Value For A Column:**

**ALTER TABLE Employee ALTER SALARY DROP DEFAULT;**

**Alter a Table Type:**

It could happen. Like, it is required to change the table type of Employee as InnoDB to make it transaction safe.

So, the syntax would be like:

ALTER TABLE Employee Type=InnoDB;

**Altering A Table Name:**

Alter table Employee Rename to Employee\_Table;

**Add Primary Key:**

Suppose, a table named Person already exists and the field ID needs to be added as primary key:

**ALTER TABLE Person ADD PRIMARY KEY (ID);**

The statement only works when Table has no id.

If the table has one existing primary key, you need to drop it first.

Now, suppose, the table description of the table Person is following:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Field** | **Type** | **Null** | **Key** | **Default** | **Extra** |
| ID | INT(11) | NO | PRI | NULL | AUTO\_  INCREMENT |
| Name | VARCHAR(80) | NO |  | NULL |  |
| Age | TINYINT(4) | NO |  | NULL |  |
| PhoneNumber | VARCHAR(20) | NO |  | NULL |  |
| Address1 | VARCHAR(90) | YES |  | NULL |  |

Now, as it can be seen, ID is already existed as primary key of the table.

So, in this case, it will not work.

**alter table Person DROP PRIMARY KEY, ADD PRIMARY KEY(ID,PhoneNumber);**

**Drop Primary Key:**

**alter table Person DROP PRIMARY KEY;**

However, if there exists only one primary key of the table, and AUTO\_INCREMENT is done to that field, it cannot be performed.

**Indexing in MySql:**

A database index is a data structure that improves the speed of operations in a table. Indexes can be created using one or more columns, providing the basis for both rapid random lookups and efficient ordering of access to records.

While creating index, it should be taken into consideration which all columns will be used to make SQL queries and create one or more indexes on those columns.Practically, indexes are also a type of tables, which keep primary key or index field and a pointer to each record into the actual table.

The users cannot see the indexes, they are just used to speed up queries and will be used by the Database Search Engine to locate records very fast.

The INSERT and UPDATE statements take more time on tables having indexes, whereas the SELECT statements become fast on those tables. The reason is that while doing insert or update, a database needs to insert or update the index values as well.

**Simple And Unique Index:**

**Unique Index:**

You can create a unique index on a table. A unique index means that two rows cannot have the same index value. Here is the syntax to create an Index on a table.

CREATE UNIQUE INDEX index\_name

ON table\_name ( column1, column2,...);

You can use one or more columns to create an index.

Now, if the keyword UNIQUE is removed, it will create a simple index.

**Now, what’s the difference between simple and unique index?**

Suppose, an index is created on COLUMN1 and COLUMN2, Now, if unique index is created, it wont allow duplicate values for two different rows. However, if the index is simple, it will allow duplicate values.

**Temporary Tables:**

Temporary tables were added in the MySQL Version 3.23. If one use an older version of MySQL than 3.23, he/she use the temporary tables, but in lower version, one can use Heap Tables.

Now, temporary tables will only last as long as the session is alive.

Let’s create a temporary table:

**create Temporary Table SalesSummary(Product\_Name VARCHAR(50) NOT NULL, Total\_sales DECIMAL(12,2) NOT NULL DEFAULT 0.00, Avg\_unit\_price DECIMAL(7,2) NOT NULL DEFAULT 0.00, Total\_units\_sold INT UNSIGNED NOT NULL DEFAULT 0);**

Now, one can insert data, update data until the session is terminated.

Once, the session is terminated, the table no longer exists.

Also, Another thing about temporary table, it will not be shown when one use:

**show tables;**

**Create A Clone Table:**

Suppose, a table named Person is initially created with the following create table query:

**create table Person(ID INT NOT NULL AUTO\_INCREMENT,Name VARCHAR(80) NOT NULL,Age TINYINT NOT NULL,PhoneNumber VARCHAR(20) NOT NULL,Address1 VARCHAR(90), PRIMARY KEY(ID,PhoneNumber));**

Now, Some insertions are done:

**Insert Into Person(ID,Name,Age,PhoneNumber,Address1)VALUES(1,'Sayak Haldar',23,'9674465435','j-2,102, DDA Flats, Kalkaji-110019');**

Now, the table ClonePerson is created:

**create table ClonePerson(ID INT NOT NULL AUTO\_INCREMENT,Name VARCHAR(80) NOT NULL,Age TINYINT NOT NULL,PhoneNumber VARCHAR(20) NOT NULL,Address1 VARCHAR(90), PRIMARY KEY(ID,PhoneNumber));**

Now, insert all the data from Person to ClonePerson:

**create table ClonePerson(ID INT NOT NULL AUTO\_INCREMENT,Name VARCHAR(80) NOT NULL,Age TINYINT NOT NULL,PhoneNumber VARCHAR(20) NOT NULL,Address1 VARCHAR(90), PRIMARY KEY(ID,PhoneNumber));**

The basic document about sql ends here.