



APRIL 25, 2019

MYSQL NOTES

VERSION 1.0

RAJ SOLANKI

Contents¹

Section 1. Sorting data	2
Section 2. MySQL Join	3
Section 3. MySQL GROUP BY clause	4
Section 4. MySQL Subquery	6
Section 5. Modifying Data in MySQL	10
Section 6. Managing MySQL Databases & Tables.....	15
Section 7. Managing MySQL databases and tables	18
Section 8. Some important datatypes	19
Section 9. MySQL Stored Procedures	26
Section 10. MySQL Triggers	35
Section 11. MySQL Administration	39

¹ Notes are derived from <http://www.mysqltutorial.org/>

Section 1. Sorting data

ORDER BY

Syntax:

```
SELECT    column1,
          column2,
          ...
from      tbl
ORDER BY  column1 [ASC|DESC],
          column2 [ASC|DESC],
          ...
```

Comments

1. Sorting by column is possible
2. You can use the **FIELD ()** function to do a custom sort as shown below.

```
SELECT ordernumber,
       status
FROM   orders
ORDER BY Field (status, 'In Process', 'On Hold', 'Cancelled',
               'Resolved', 'Disputed', 'Shipped')
```

LIMIT

Syntax:

```
SELECT column1,
       column2,
       ...
       from table
LIMIT offset, count;
```

Comments

1. **offset** specifies the offset of the first row to return. The offset for the 1st row starts at 0.
2. **count** specifies the maximum number of rows to be returned.
3. You can use the **LIMIT** clause to get the Nth highest value by first using **ORDER BY** then specifying the **LIMIT** clause.

IS NULL

Comments

1. Checks for null values
2. MySQL performs the same optimization for **IS NULL** as for the (=) operator.

Section 2. MySQL Join

Summary

1. Relational databases consist of multiple table that are linked via foreign key columns.
2. MySQL supports the following joins: **Cross Join, Inner Join, Left Join, and Right Join**
3. **Full Outer Join** is currently not supported.

MySQL CROSS JOIN

Comments

1. **CROSS JOIN** is a cartesian product of rows between multiple tables
2. For example, if you join tables t1 and t2 then the result set will be combinations of rows between t1 and t2.

Syntax

```
SELECT t1.id,  
       t2.id  
FROM   t1  
       CROSS JOIN t2;
```

MySQL LEFT JOIN

Comments

1. A **LEFT JOIN** returns all rows in the left table including rows that are not in the right table so NULLs appear in columns in the right table.

MySQL INNER JOIN

Comments

1. **INNER JOIN** matches rows in one table with rows in another table.
2. In order to use **INNER JOIN**, 3 things need to be specified:
 - i. 1st the main table is specified in the **FROM** clause
 - ii. 2nd the table that is being joined to appears in the **INNER JOIN** clause
 - iii. 3rd the join condition (join predicate) is specified after the **ON** keyword.
This rule specifies how to match the rows with the main table.

Syntax

```
SELECT column_list
FROM   t1
       INNER JOIN t2
           ON join_condition1
       INNER JOIN t3
           ON join_condition2 ...
WHERE  where_conditions
```

A Practical Use of MySQL CROSS JOIN Clause

<http://www.mysqltutorial.org/mysql-cross-join/>

Section 3. MySQL GROUP BY clause

Summary

1. Groups a set of rows into a set of summary rows by values of columns or expressions.
2. **GROUP BY** returns 1 row for each group sorted.
3. You can use **GROUP BY** with aggregate functions.
4. **GROUP BY** should appear after **FROM** and **WHERE** clauses.
5. MySQL allows aliases to be used with **GROUP BYs** shown below.

```
SELECT Year(orderdate) AS year,
       Count(ordernumber)
FROM   orders
GROUP BY year;
```

6. Sorting of groups is also allowed in MySQL.

Syntax

```
SELECT    c1,  
          c2,  
          ...,  
          cn,  
          aggregate_function(ci)  
FROM      TABLE  
WHERE     where_conditions  
GROUP BY  c1,  
          c2,  
          ...,  
          cn;
```

HAVING clause

Notes

1. **HAVING** clause is used to filter **GROUP BYs**.
2. If the **GROUP BY** is omitted, then the **HAVING** clause behaves like a **WHERE** clause.

MySQL ROLLUP

Notes

1. **ROLLUP** generates subtotals and grand totals for the order values.
2. Having more than 1 column in the **GROUP BY** clause causes the **ROLLUP** clause to assume a hierarchy among the input columns. It will also generate a subtotal for each inner most group and a grand total.
3. **GROUPING () function** is used to check whether or not a **NULL** in the result set represents the subtotals or grand totals. It returns 1 when **NULL** occurs in a super-aggregate row otherwise returns 0.

Syntax

```
SELECT select_list  
FROM   table_name  
GROUP BY c1,  
         c2,  
         c3 WITH rollup
```

Section 4. MySQL Subquery

Summary

1. A subquery is a query nested within another query and enclosed within parenthesis.
2. Additional subqueries can be nested inside another subquery.
3. A subquery can be used anywhere that an **expression** is used.
4. Subqueries on its own can execute on its own hence they are independent.

Example

```
SELECT customernumber,  
       checknumber,  
       amount  
FROM   payments  
WHERE  amount > (SELECT Avg(amount)  
                 FROM   payments);
```

MySQL subquery in the FROM clause

Notes

1. When a subquery is in the **FROM** clause, the result set returned from a subquery is used as a temporary table. This table is known as a **derived table** or a **materialized subquery**.

Example

```
SELECT Max(items),  
       Min(items),  
       Floor(Avg(items))  
FROM   (SELECT ordernumber,  
              Count(ordernumber) AS items  
        FROM   orderdetails  
        GROUP BY ordernumber) AS lineitems;
```

MySQL correlated subquery

Notes

1. Correlated subqueries depend on the outer query.
2. Correlated subqueries are evaluated once for each row in the outer row.

Example

```
SELECT productname,  
       buyprice  
FROM   products p1  
WHERE  buyprice > (SELECT Avg(buyprice)  
                  FROM   products  
                  WHERE  productline = p1.productline);
```

MySQL subquery with EXISTS and NOT EXISTS

Notes

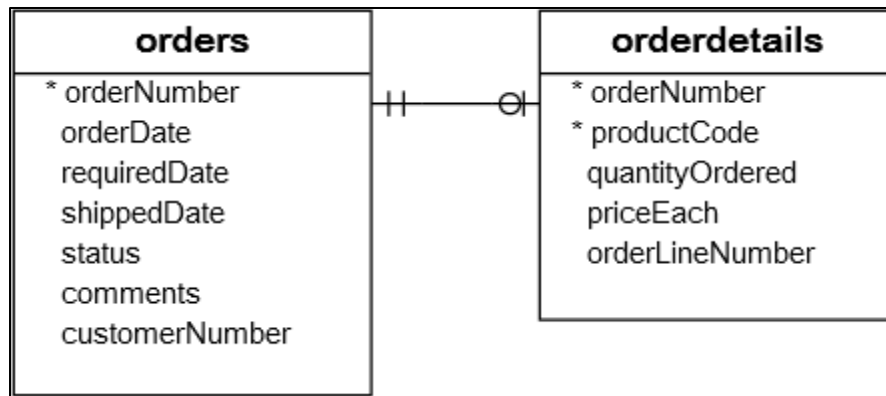
1. Subqueries that are used with **EXISTS** or **NOT EXISTS** returns a **Boolean** value.
2. The Boolean value returned is based on if any rows are returned are from the subquery.
3. **EXISTS** and **NOT EXISTS** are often used with correct
4. **EXISTS** operator works on the principle of “at least found” which means that it stops once it finds at least one matching row which makes it very fast.
5. In general use **EXISTS** for large datasets and **IN** for smaller datasets.

Syntax

```
SELECT *  
FROM   table_name  
WHERE  EXISTS( subquery );
```

Example 1

1. In the example below we are going to use a correlated subquery to find customers who placed at least one sales order with a total value great then 60k using the **Exists** operator.



2. Example Syntax

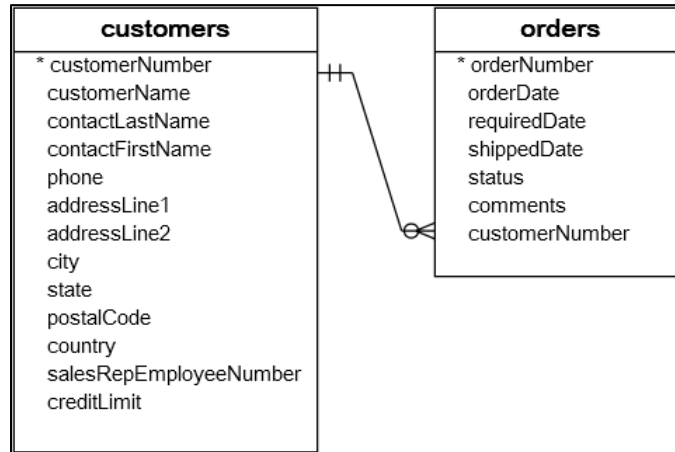
```

SELECT customernumber,
       customername
FROM   customers
WHERE  EXISTS (SELECT ordernumber,
                      Sum(priceeach * quantityordered)
                FROM   orderdetails
                INNER JOIN orders USING (ordernumber)
                WHERE  customernumber = customers.customernumber

                GROUP BY ordernumber
                HAVING Sum(priceeach * quantityordered) > 60000)
;
  
```

Example 2

1. Suppose you want to find the customer who has placed at least one sales order, you use the **EXISTS** operator.



2. Example Syntax

```
SELECT customernumber,  
       customername  
FROM   customers  
WHERE  EXISTS (SELECT 1  
               FROM   orders  
               WHERE  orders.customernumber = customers.customernum  
ber);
```

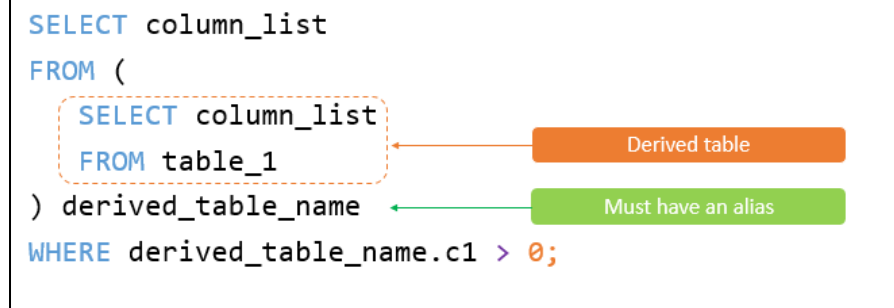
MySQL derived table

Notes:

1. Derived tables are a virtual table returned from a SELECT statement.
2. Subquery and derived tables are interchangeable
3. A derived table **must** have an alias so you reference its name later in the query.

Syntax:

```
SELECT column_list
FROM (
    SELECT column_list
    FROM table_1
) derived_table_name
WHERE derived_table_name.c1 > 0;
```



Union

Notes

1. UNION appends result sets vertically and JOIN combine sets horizontally

SQL MINUS operator

Notes

1. **MINUS** operator is not supported in MySQL
2. To simulate the **MINUS** operator, use a **LEFT JOIN** and filter on the **NULLs** on the right table as show below.

Syntax

```
SELECT column_list
FROM table_1
     LEFT JOIN table_2
           ON join_predicate
WHERE table_2.id IS NULL;
```

Section 5. Modifying Data in MySQL

MySQL Insert

Syntax

```
INSERT INTO table (c1,c2,...)
VALUES
    (v11,v12,...) ,
```

```
(v21, v22, ...),  
...  
(vnn, vn2, ...);
```

Notes

1. In syntax above c1, c2, ... are the columns and v11, ... are the values.
2. To store date values in a column you use the format 'YYYY-MM-DD'.

MySQL INSERT INTO SELECT

Syntax

```
INSERT INTO table_name  
          (column_list)  
SELECT select_list  
FROM    another_table;
```

Notes

1. This is another way to insert data into a table where the **SELECT** statement can retrieve data from one of more tables and insert it into another table.
2. Very useful for copying data over from 1 or more table.

MySQL INSERT ON DUPLICATE KEY UPDATE statement

Notes

1. While inserting a new row into a table may cause an error if a duplicate is inserted in a **UNIQUE** index or **PRIMARY KEY**.
2. To mitigate errors due to duplicates, use **ON DUPLICATE KEY UPDATE** option in the **INSERT** statement then MySQL will override the existing row with the new values instead.

Syntax

```
INSERT INTO table  
          (column_list)  
VALUES  
          (value_list)  
on duplicate KEY
```

```
UPDATE c1 = v1,  
       c2 = v2,  
       ...;
```

MySQL INSERT IGNORE Statement

Notes

1. **INSERT IGNORE** is used to avoid errors caused by invalid data. So, any row with invalid data is ignored and the rows with valid data are inserted.
2. MySQL will issue a warning for each error and will try to adjust the values to make them valid before insertion.

Syntax

```
INSERT IGNORE  
into      table  
      ( column_list )  
VALUES  
      ( value_list ),  
      ( value_list ),  
      ...
```

MySQL UPDATE

Notes

1. **UPDATE** statement is used for several things such as updating existing data, change column values of a single to multiple rows.

Syntax

```
UPDATE [LOW_PRIORITY] [IGNORE] table_name  
SET    column_name1 = expr1,  
       column_name2 = expr2,  
       ...  
[WHERE condition];
```

2. In the syntax above we can specify several things
 - a. **LOW_PRIORITY** – This modifier tells **UPDATE** to delay the modification until there is no connection reading data from the table.
 - b. **IGNORE** – This enables the **UPDATE** statement to continue updating rows despite any errors such as duplicate-key.

- c. In the **SET** clause, choose what columns you want to modify and the new values. (Required)
- d. Can add several columns separated by a comma. (Required)
- e. Columns can be set to a value, an expression, or a subquery (Required)
- f. In the **WHERE** clause you can specify what rows are updated (Optional)

MySQL UPDATE JOIN syntax

Notes

1. Use **JOIN** clauses in the **UPDATE** statement to perform cross-table updates.

Syntax

```
UPDATE t1, t2,  
[INNER JOIN | LEFT JOIN] t1 on t1.c1 = t2. c1  
SET          t1.c2 = t2.c2,  
             t2.c3 = expr  
WHERE        CONDITION
```

Example

```
UPDATE employees  
    INNER JOIN merits  
        ON employees.performance = merits.performance  
SET salary = salary + salary * percentage;
```

MySQL DELETE

Notes

1. Here we specify the table where we want to delete data.
2. Any row that satisfies the condition in the **WHERE** clause will be deleted.
3. An alternative is to use **TRUNCATE TABLE** to delete all rows and better performance.
4. Any table with a foreign key constraint, any rows deleted in the parent table is deleted in the child tables by specifying **ON DELETE CASCADE** option in the schema.

Syntax

```
DELETE FROM table_name  
WHERE CONDITION;
```

MySQL REPLACE statement

Notes

1. **REPLACE** works in the following ways
 - a. Any new row that doesn't exist, **REPLACE** inserts it as a new row
 - b. If new row exists, then **REPLACE** deletes the old row and then inserts new row.
 - c. To determine if table has the row, MySQL uses the index column to check if the row is duplicate or not.
2. To use **REPLACE**, user must have both **INSERT** and **DELETE** privileges.
3. Avoid the **REPLACE** statement because other DBs may not support it so use **DELETE** and **INSERT** instead.

Syntax

Syntax (1st form)

```
REPLACE INTO table_name(column_list) VALUES(value_list);
```

Syntax (2nd form)

```
REPLACE INTO TABLE SET column1 = value1, column2 = value2;
```

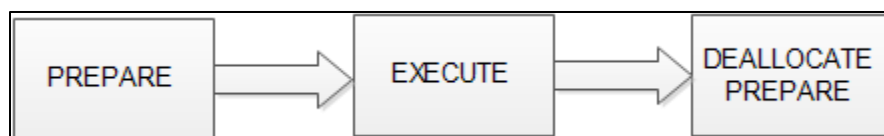
Syntax (3rd form)

```
REPLACE INTO table_1(column_list) SELECT column_list FROM table_2 WHERE where_condition;
```

MySQL Prepared Statement

Notes

1. Prepared Statements allow queries to execute faster and more secure
2. How to use prepared statements? – Workflow below
 - a. Specify PREPARE which prepares statement for execution
 - b. Then specify EXECUTE which runs a prepared statement
 - c. Finally specify DEALLOCATE PREPARE which releases a prepared statement.



Example

1. In this example we are using a **PREPARED** statement to select the products with productCode "s10_1678".

```
PREPARE stmt1 FROM 'SELECT productCode, productName FROM products
WHERE productCode = ?';
SET @pc = 'S10_1678';
EXECUTE stmt1 USING @pc;
DEALLOCATE prepare stmt1;
```

Section 6. Managing MySQL Databases & Tables

MySQL CREATE TABLE syntax

Syntax

```
I. CREATE TABLE [IF NOT EXISTS] table_name
    ( column_list )
```

```
II. column_name data_type(length) [NOT NULL] [DEFAULT value]
    [AUTO_INCREMENT]
```

Notes

1. In the **column_list** section the columns are separated by commas.
2. To define a column, use the syntax in II.
 - a. Each column has a specific datatype and a maximum length
 - b. **NOT NULL** indicates that the column does not allow NULL.
 - c. **DEFAULT** value is used to specify the default value of the column.
 - d. **AUTO_INCREMENT** indicates that the value of the column is automatically increased by 1 when a new row is inserted into a table. Each table can only have only 1 of these columns
 - e. **PRIMARY KEY (COL1, COL2,...)** can be added at the end to indicate the primary key columns.

Introduction to MySQL ALTER TABLE statement

Notes

1. **ALTER TABLE** statement allows you to do many things such as drop columns and change the data type
2. How to turn on the **auto_increment** attribute for a column?


```
ALTER TABLE tasks
  CHANGE COLUMN task_id task_id INT(11) NOT NULL auto_increment;
```

3. How to add a new column into a table?

```
ALTER TABLE tasks
  ADD COLUMN complete DECIMAL (2, 1) NULL after description;
```

4. How to rename tables?

```
ALTER TABLE tasks
  RENAME TO work_items;
```

MySQL RENAME TABLE statement

Notes

1. Used to rename tables but can't rename temporary tables or views.

Syntax

```
RENAME TABLE old_table_name_1 TO new_table_name_2,
old_table_name_2 TO new_table_name_2, ...
```

MySQL ADD COLUMN statement

Syntax

```
ALTER TABLE table ADD [COLUMN] COLUMN_NAME column_definition
[FIRST|AFTER existing_column];
```

Example

```
ALTER TABLE vendors
  ADD COLUMN phone VARCHAR (15) after name;
```

DROP COLUMN statement

Syntax

```
ALTER TABLE table
  DROP COLUMN column;
```

Notes

1. Removing columns means that another db objects such as stored procedures, triggers, etc. that depend on these columns means the object(s) will need to be updated that depend which can impair performance.

2. Trying to remove a column that is a foreign key will cause an error, so need to remove the foreign key constraint first.

DROP TABLE statement syntax

Syntax

```
DROP [temporary] TABLE [IF EXISTS] table_name [, table_name] ...  
[RESTRICT | cascade]
```

Notes

1. It does not remove specific user privileges associated with the removed tables.

MySQL transactions

Notes

1. MySQL transactions allow users to execute a set of transactions completely and never results in a partial set of operations.
2. If any part of the transaction fails, then the entire transaction is rolled back. So, the database returns to original state. Otherwise the transaction is committed
3. How to setup a transaction?
 - a. Use the **START TRANSACTION** Statement
 - b. To commit the current transaction and make it permanent use **COMMIT** statement
 - c. To rollback the current transaction use **ROLLBACK** statement
 - d. To disable/enable auto-commit mode use **SET autocommit= 0/1**.

Table Locking

Notes

1. By default, MySQL allows a client session to explicitly acquire a table lock which prevents other sessions from accessing the table.
2. Client sessions can only acquire/release table locks only for itself and not locks from other sessions.
3. To release a lock on a table, use, **UNLOCK TABLES**;

Syntax

```
LOCK tables table_name [READ | WRITE]
```

Read Locks

Notes

1. A **READ** lock for a table can be acquired by multiple sessions at a time. In fact, other sessions can read from the table without acquiring the lock.
2. Other sessions cannot write data to the table until the **READ** locks are released
3. If a session is terminated, MySQL lifts all read locks.
4. Use **SHOW PROCESSLIST** to see more detailed information.

Write Locks

Notes

1. The only session that holds a **WRITE** lock can read & write data from the table
2. Other sessions cannot read data/write data to the table till the write lock is lifted.
3. Read locks are shared locks and prevent a write lock from being acquired but not read locks.

Section 7. Managing MySQL databases and tables

MySQL TRUNCATE TABLE statement

Notes

1. **TRUNCATE TABLE** checks if there are any foreign key constraints in which case it deletes row by row till it encounters a row referenced by a row in a child table (**Engine= InnoDB**)
2. If there are no foreign key constraints, it will drop the table and recreate a new empty one which is faster especially for bigger tables.
3. Anything with a **DELETE CASCADE**, the child rows will be deleted as well.

Syntax

```
TRUNCATE TABLE table_name;
```

MySQL temporary table

Notes

1. Temporary tables are great when a query is very large or complex.
2. Use CREATE TEMPORARY TABLE.
3. MySQL automatically drops the temporary table when the session ends, or you may drop it.
4. Different sessions have access to only their temporary tables and not others.

Section 8. Some important datatypes

MySQL DATETIME functions

Notes

1. How to set a variable to the current date and time?

```
SET @dt = now();
```

2. How to use **DATE_FORMAT ()** function to format a **DATETIME** value?

```
SELECT Date_format(@dt, '%H:%i:%s - %W %M %Y');
```

3. Using **DATE_ADD ()** to add time to a **DATETIME** variable examples

```
SELECT @dt start,  
       Date_add(@dt, INTERVAL 1 second) '1 second later',  
       date_add(@dt, INTERVAL 1 minute) '1 minute later',  
       date_add(@dt, INTERVAL 1 hour) '1 hour later',  
       date_add(@dt, INTERVAL 1 day) '1 day later',  
       date_add(@dt, INTERVAL 1 week) '1 week later',  
       date_add(@dt, INTERVAL 1 month) '1 month later',  
       date_add(@dt, INTERVAL 1 year) '1 year later';
```

4. Use **DATE_SUB ()** to subtract time off a **DATETIME** value.

```
SELECT @dt start,  
       Date_sub(@dt, INTERVAL 1 second) '1 second before',  
       date_sub(@dt, INTERVAL 1 minute) '1 minute before',  
       date_sub(@dt, INTERVAL 1 hour) '1 hour before',  
       date_sub(@dt, INTERVAL 1 day) '1 day before',
```

```
date_sub(@dt, INTERVAL 1 week) '1 week before',  
date_sub(@dt, INTERVAL 1 month) '1 month before',  
date_sub(@dt, INTERVAL 1 year) '1 year before';
```

5. Use **DATE_DIFF ()** to calculate the difference in days between two **DATETIME** values.

MySQL ENUM

Notes

1. An **ENUM** is a list of permitted values defined at time of table creation.
2. **ENUMs** provide compact data storage since it uses numeric indexes to represent string values
3. **ENUMs** provide readable queries and output
4. **ENUMs** has some disadvantages such as
 - a. Changing enumeration members requires redoing the entire table using the **ALTER TABLE** statement.
 - b. Other databases may not support **ENUMs** so exporting **ENUMs** may cause issues

Syntax

```
CREATE TABLE table_name  
( ...  
    col enum ('value1','value2','value3'),  
    ...  
);
```

MySQL TIME data type

Notes

1. MySQL uses the '**HH:MM:SS**' format for querying and displaying a time value

MySQL NOT NULL constraint

Notes

1. NOT NULL constraint is a column constraint that forces the values of a column to non-NULL values

Syntax

```
column_name data_type NOT NULL;
```

Example

```
CREATE TABLE tasks
(
    id          INT auto_increment PRIMARY KEY,
    title       VARCHAR (255) NOT NULL,
    start_date  DATE NOT NULL,
    end_date    DATE
);
```

MySQL primary key

Notes

1. **Primary key** is a column (s) that uniquely identifies each row in the table.
2. How to define a primary key?
 - a. **Primary key** must contain all unique values.
 - b. Cannot contain null values and should be declared **NOT NULL** even though MySQL will do that implicitly as well.
 - c. In MySQL can only have one primary key.
 - d. The datatype of a Primary Key should be an **integer**.
 - e. Primary Keys often use AUTO_INCREMENT to generate a unique ID for the next row automatically.

Example 1

```
CREATE TABLE users
(
    user_id  INT auto_increment PRIMARY KEY,
    username VARCHAR(40) ,
    password VARCHAR(255) ,
    email    VARCHAR(255)
    /** or PRIMARY KEY (user_id) **/
);
```

Example 2

```
ALTER TABLE table_name
ADD PRIMARY KEY(primary_key_column);
```

PRIMARY KEY vs. UNIQUE KEY vs. KEY

Notes

1. Key is a synonym for INDEX and you can use the KEY when you want to create an index for a column or set of columns that is not part of a primary key.
2. UNIQUE index adds an additional constraint that the values must be unique but allows NULLS. Table can more than 1 UNIQUE index.

Syntax

```
/**Add a UNIQUE index for the username column.**/  
ALTER TABLE users ADD UNIQUE index username_unique  
(username ASC) ;
```

MySQL foreign key

Notes

1. A **foreign key** is a field that matches to another field of another table.
2. It places constraints on data in the related tables to maintain referential integrity.

Syntax

```
CONSTRAINT constraint_name  
FOREIGN KEY foreign_key_name (columns)  
REFERENCES parent_table(columns)  
ON  
DELETE action  
ON  
UPDATE action
```

3. **CONSTRAINT** – define a constraint name else omit it
4. **FOREIGN KEY** – This clause has the child columns that will refer to the primary key columns in the parent table. The name is optional
5. **ON DELETE** – If nothing is specified here, MySQL will reject any deletions in the primary key column of parent table.
 - a. **ON DELETE CASCADE** – You may specify this rather here if you want the related child rows to be deleted as well when the related parent record is deleted

- b. **ON DELETE SET NULL** – This will set the related child foreign key rows to NULL when the parent row(s) are being deleted.
- 6. **ON UPDATE** – allows you to specify what happens to the rows in the child table when rows in the parent table are updated.
 - a. If omitted, MySQL rejects any updates to the parent table
 - b. **ON UPDATE CASCADE** – will perform cross-table updates
 - c. **ON UPDATE SET NULL** – sets values in the rows in the child table to NULL when rows in the parent table are updated.

Dropping MySQL foreign key

Syntax

```
ALTER TABLE table_name  
DROP FOREIGN KEY constraint_name;
```

Notes

1. Disabling foreign key checks allow you to import data from an external file into a table without having to worry about loading data into the parent table 1st. To disable it use the following syntax below:

```
SET foreign_key_checks = 0;
```

MySQL UNIQUE constraint

Notes

1. Enforces the uniqueness in a column and it's a constraint

Syntax

```
/** Single Column **/  
CREATE TABLE table_1  
(  
    column_name_1 DATA_TYPE UNIQUE,  
);  
  
/**Multiple columns**/CREATE TABLE table_1  
(  
    column_name_1 data_type,  
    column_name_2 data type,  
    UNIQUE(column_name_1,column_name_2)  
);  
  
/** Alter Syntax **/  
ALTER TABLE suppliers ADD CONSTRAINT uc_name_address UNIQUE (name,address);
```


SQL CHECK constraint

Notes

1. You can implement CHECK constraints in MySQL using triggers that checks the data before inserting it.

Example

```
DELIMITER $
CREATE PROCEDURE
  `check_parts`(IN cost  DECIMAL(10,2),
                IN price DECIMAL(10,2))
begin
IF cost < 0 THEN
  signal SQLSTATE '45000'
  SET message_text = 'check constraint on parts.cost failed';
END IF;
IF price < 0 THEN
  signal SQLSTATE '45001'
  SET message_text = 'check constraint on parts.price failed';
END IF;
IF price < cost THEN
  signal SQLSTATE '45002'
  SET message_text = 'check constraint on parts.price & parts.cost failed';
END IF;
END$
delimiter ;
-- before insert
DELIMITER $
CREATE TRIGGER `parts_before_insert` BEFORE
  INSERT
  ON `parts` FOR EACH row begin CALL check_parts
    (
      new.cost,
      new.price
    );
END$
delimiter ;
-- before update
DELIMITER $
CREATE TRIGGER `parts_before_update` BEFORE
  UPDATE
  ON `parts` FOR EACH row begin CALL check_parts(new.cost,new.price);
END$
delimiter;
```

Strings

Notes

1. **LENGTH ()** function returns length of a string in bytes
2. **CHAR_LENGTH ()** returns the # of characters in a string

3. **Convert ()** converts a string into a specific character set.

Import CSV File into MySQL Table

Notes

1. User needs to have **FILE** and **INSERT** privileges

```
LOAD data INFILE 'c:/tmp/discounts.csv'  
INTO TABLE discounts  
fields TERMINATED BY ','  
ENCLOSED BY '"'  
LINES TERMINATED BY '\n'  
IGNORE 1 rows;
```

2. **FIELD TERMINATED BY <symbol>** indicates what each line in the file is terminated by.
3. **ENCLOSED BY '<symbol>'** indicates what some of the elements in the file is captured in.

Transforming data while importing

Notes

1. You can transform data using the SET clause in the LOAD DATA INFILE statement.

Example

```
LOAD data INFILE 'c:/tmp/discounts_2.csv'  
INTO TABLE discounts  
fields TERMINATED BY ',' ENCLOSED BY '"'  
LINES TERMINATED BY '\n'  
IGNORE 1 rows  
(title,@expired_date,amount)  
SET expired_date = str_to_date(@expired_date, '%m/%d/%Y');
```

Exporting Data

Notes

1. Before exporting data ensure MySQL server's process has write access to target folder that has the CSV file and CSV file must exist.

```
SELECT ordernumber,  
       orderdate,  
       Ifnull(shippeddate, 'N/A')  
FROM   orders  
INTO   OUTFILE 'C:/tmp/orders2.csv'  
fields ENCLOSED BY '"'
```

```
TERMINATED BY ';'
ESCAPED BY '"'
LINES TERMINATED BY '\r\n';
```

Section 9. MySQL Stored Procedures

Notes

1. A stored procedure is a set of declarative SQL statements.
2. MySQL store procedures are compiled on demand and maintains it in its cache vs being compiled already and then stored.
3. If a session has multiple stored procedures then the compiled version is used otherwise it functions as a query.
4. **Advantages**
 - a. Stored procedures help reduce traffic between application and database because the application only sends the name and parameters of the stored procedure.
 - b. Stored procedures are reusable and transparent
 - c. Store procedures are secure since users can be granted access to the procedure rather than the underlying data.
5. **Disadvantages**
 - a. Can use large amount of memory and MySQL does not handle a large number of logical operations very well
 - b. Not designed to accommodate complex & flexible business logic
 - c. Difficult to debug
 - d. Requires a specialized skillset to maintain and develop

Writing the first MySQL stored procedure

Example

```
DELIMITER //
CREATE PROCEDURE
    getallproducts()
begin
    SELECT *
    FROM    products;

END //
delimiter;
```

Notes

1. **DELIMITER //** - changes the standard delimiter to //. We do this because we want to pass the stored procedure to the server as a whole rather than letting MySQL interpret each statement at a time. We change the delimiter to; at the end.
2. **getallproducts ()** - **This** is the procedure name and add parenthesis.
3. Place all the logic between begin and end.

Declaring variables

Notes

1. To declare a variable inside a stored procedure use the DECLARE statement.

Syntax

```
DECLARE variable_name datatype(size) DEFAULT default_value;
```

2. When declaring a variable its initial value is a NULL
3. Any variables that begins with @ symbol is a session variable that is available till the session ends.

MySQL stored procedure parameters

Notes

1. There are 3 modes for the parameters.
2. **IN** – This is the default mode and you define an IN parameter when you must pass an argument. The value of the IN parameter is protected which means if the value of the IN parameter is changed in the procedure, the original value is still retained at the end. Thus, it works on a copy
3. **OUT**- **This** is the value that is passed back out of the stored procedure. Its value can be changed inside the procedure
4. **INOUT** – It's a combination parameter so that means that you can use this to pass in an argument and modify it as well from within.

Syntax

```
MODE param_name param_type(param_size)
```

5. The **MODE** is **IN**, **OUT**, or **INOUT** in the syntax.
6. Each parameter is separated by a comma.

Example 1

```
DELIMITER //
```

```
CREATE PROCEDURE
```

```
    getofficebycountry(IN countryname VARCHAR(255))
```

```
begin
```

```
    SELECT *
```

```
    FROM    offices
```

```
    WHERE   country = countryname;
```

```
END //
```

```
delimiter ;
```

```
DELIMITER $$
```

```
CREATE PROCEDURE
```

```
    countorderbystatus(
```

```
                                IN orderstatus VARCHAR(25),
```

```
                                OUT total      INT)
```

```
begin
```

```
    SELECT count(ordernumber)
```

```
    INTO   total
```

```
    FROM   orders
```

```
    WHERE  status = orderstatus;
```

```
END$$
```

```
delimiter ;
```

```
CALL countorderbystatus('Shipped',@total);
```

```
SELECT @total;
```

Example 2

```
DELIMITER $$
```

```
CREATE PROCEDURE
```

```
    set_counter(INOUT count INT(4),
```

```
                IN inc      INT(4))
```

```
begin
```

```
    SET count = count + inc;
```

```
END$$
```

```
delimiter ;
```

```
SET @counter = 1;
```

```
CALL set_counter(@counter, 1); -- 2
```

```
CALL set_counter(@counter, 1); -- 3
CALL set_counter(@counter, 5); -- 8
SELECT @counter; -- 8
```

7. In example 2, **set_counter ()** accepts 1 **INOUT** parameter and one **IN** parameter.
8. Inside **set_counter ()** count variable is increased by *inc*.
9. We make a series of calls to **set_counter ()** and show the output of counter off to the side.

MySQL IF Statement

Syntax

```
IF expression THEN
    statements;
ELSEIF elseif -expression THEN
ELSEIF-statements;
...
ELSE
    else-statements;
END IF;
```

Example

```
DELIMITER $$
CREATE PROCEDURE
    getcustomerlevel(
        IN p_customernumber INT(11),
        OUT p_customerlevel VARCHAR(10))
begin
    DECLARE creditlim DOUBLE;
    SELECT creditlimit
    INTO    creditlim
    FROM    customers
    WHERE   customernumber = p_customernumber;
    IF creditlim > 50000 THEN
        SET p_customerlevel = 'PLATINUM';
    ELSEIF (creditlim <= 50000 AND creditlim >= 10000) THEN
        SET p_customerlevel = 'GOLD';
    ELSEIF creditlim < 10000 THEN
        SET p_customerlevel = 'SILVER';
    END IF;
END$$
```

CASE statement

Syntax

```
CASE case_expression
WHEN when_expression_1 THEN commands
WHEN when_expression_2
THEN commands ... ELSE commands
END CASE
```

MySQL Loop in Stored Procedures

Notes

1. There are three loop statements in MySQL: **WHILE**, **REPEAT** and **LOOP**.

Example 1 – While Loop

```
DECLARE x INT;
DECLARE str VARCHAR(255);
SET x = 1;
SET str = '';
WHILE x <= 5 do
SET str = concat(str,x,',');
SET x = x + 1;
END WHILE;
```

Example 2- REPEAT loop

```
REPEAT
    statements;
UNTIL expression
end REPEAT
```

Introduction to MySQL cursor

Notes

1. **Cursors** allow you to iterate through a set of rows returned by a query and process each row accordingly.
2. In MySQL cursors are **read only**, **non-scrollable**, and **sensitive** described below.

- a. **Read-only** – Cannot update the data in the underlying table through a cursor
 - b. **Non-Scrollable**- Cursors go from top to bottom in result set returned by the **SELECT** statement.
 - c. **Sensitive** – There are 2 kinds of cursors here
 - i. **Sensitive Cursor** - This cursor points to the actual data and performs fastest. The only issue with this cursor type is if any of the underlying data is updated then will affect the data being used by the cursor.
 - ii. **Insensitive Cursor** – This cursor uses a temporary copy of the data.
3. How to work with a **MySQL Cursor**?
- a. First you declare a cursor using **DECLARE**. A cursor declaration **must** happen after any variable declaration else an error will occur.

```
DECLARE cursor_name CURSOR FOR select_statement;
```

- b. Next initiate the **CURSOR** by using the **OPEN** statement. This must be specified before fetching rows from result set

```
open cursor_name;
```

- c. Next use the **FETCH** statement to retrieve the next row point by the cursor and move to the next row in the result set.

```
FETCH cursor_name INTO variables list;
```

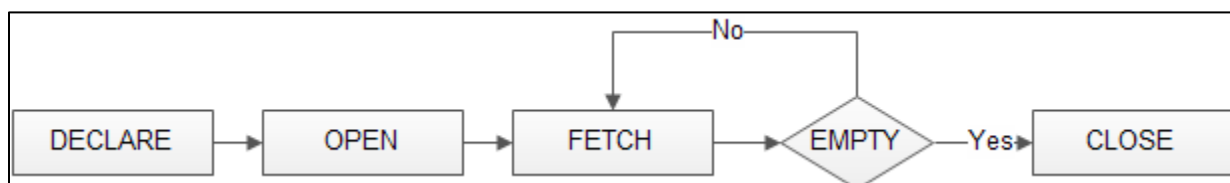
- d. Check to see if there is any row available before fetching it. You would use a **NOT FOUND** handler to handle the situation where no row is found since the cursor will try to attempt to read the next row.

```
DECLARE CONTINUE handler FOR NOT found SET finished = 1;
```

- e. **CLOSE** the cursor to deactivate it.

```
close cursor_name;
```

- f. The following diagram illustrates how MySQL cursor works.



Example


```

DELIMITER $$
CREATE PROCEDURE
    build_email_list (INOUT email_list VARCHAR(4000))
begin
DECLARE v_finished INTEGER DEFAULT 0;
DECLARE v_email VARCHAR (100) DEFAULT "";
    - declare cursor for employee email

DECLARE email_cursor CURSOR FOR
    SELECT email
    FROM    employees;

    -- declare NOT FOUND handler

DECLARE CONTINUE handler
    FOR NOT found SET v_finished = 1;

open email_cursor;

get_email:
    LOOP
        FETCH email_cursor
        INTO    v_email;
IF v_finished = 1 THEN
    LEAVE get_email;
END IF;
    - build email list

SET email_list = concat(v_email,";", email_list);
END LOOP get_email;

close email_cursor;

END$$
    delimiter ;

```

Displaying characteristics of stored procedures

Notes

1. To display properties of a stored procedure you use the **SHOW PROCEDURE STATUS**.

Syntax

```
SHOW PROCEDURE status [LIKE 'pattern' | WHERE expr];
```

MySQL Error Handling in Stored Procedures

Notes

1. You want to be able to handle the errors that may occur when running a procedure. In this case you would use a handler that would issue warnings or exceptions to specific conditions via Error codes.
2. How to declare a handler?
 - a. Specify a **condition_value** and then MySQL will execute the **statement** and continue/exit the current block of code based on **action**.
 - b. **action** – Here you specify **CONTINUE** or **EXIT**.
 - c. **condition_value** – specifies condition (s) that activate the handler. It can take 1 of the following values
 - i. MySQL error code
 - ii. Can be a **SQLSTATE**, **SQLWARNING**, **NOTFOUND**, or **SQLEXCEPTION** conditions. **NOTFOUND** is used in the case of a cursor or **SELECT INTO** statement.
 - iii. A named condition associated with a MySQL error code or a **SQLSTATE**.
 - d. Error types specified in **condition_value** has an order of precedence. MySQL will look to the most specific defined down in the order of **MySQL error code**, **SQLSTATE**, and then **SQLEXCEPTION**.

```
DECLARE action handler FOR condition_value statement;
```

Example 1

```
DELIMITER $$
CREATE PROCEDURE
    insert_article_tags(IN article_id INT,
                      IN tag_id INT)
begin
    DECLARE CONTINUE handler FOR 1062
    SELECT concat('duplicate keys (',article_id,',',tag_id,') found') AS msg;

    -- insert a new record into article_tags
    INSERT INTO article_tags
        (
            article_id,
            tag_id
        )
    VALUES
        (
            article_id,
            tag_id
        );

    -- return tag count for the article
```

```
SELECT Count(*)  
FROM article_tags;  
END
```

3. Using named conditions which allows you to avoid becoming cluttered with the different numbering on the error codes.

Syntax

```
DECLARE condition_name CONDITION FOR condition_value;
```

Example

```
DECLARE table_not_found CONDITION FOR 1051;  
  
DECLARE EXIT handler FOR table_not_found SELECT  
'Please create table abc first';  
  
SELECT * FROM abc;
```

MySQL SIGNAL statement

Notes

1. **SIGNAL** statements are useful to return an error or working condition to the caller from a stored procedure, trigger, or event.

Syntax

```
SIGNAL sqlstate | condition_name;  
SET condition_information_item_name_1 = value_1,  
condition_information_item_name_2 = value_2,  
etc
```

MySQL Stored Function

Notes

1. It is a special class of stored program that returns a single value.
2. Unlike a stored procedure, functions can be used anywhere an expression is used.
3. Great to improve readability and maintenance of code.

Syntax

```
CREATE function function_name(param1, param2,...)
returns datatype
[NOT] DETERMINISTIC
statements
```

4. How to create a function?

- a. 1st specify the **CREATE FUNCTION** clause
- b. 2nd all parameters in the function are **IN** and cannot be changed.
- c. 3rd specify the return data type in the **RETURN** statement and can be any of the MySQL data types.
- d. 4th If the stored function returns the same result for the same input parameters then the function is considered **DETERMINISTIC** otherwise if it changes then its not. It should be declared right since it could result in bad performance.
- e. 5th Specify something in the **RETURN** statement.

Syntax

```
DELIMITER $$
CREATE function
customerlevel(p_creditlimit DOUBLE) returns VARCHAR (10)
DETERMINISTIC
begin
    DECLARE lvl VARCHAR (10);
    IF p_creditlimit > 50000 THEN
        SET lvl = 'PLATINUM';
    ELSEIF
        (p_creditlimit <= 50000 AND p_creditlimit >= 10000) THEN
        SET lvl = 'GOLD';
    ELSEIF
        p_creditlimit < 10000 THEN
        SET lvl = 'SILVER';
    END IF;
    RETURN (lvl);
END
```

Section 10. MySQL Triggers

Notes

1. A SQL trigger is a set of SQL statements stored in the database when an event occurs when an insert, update, or delete occurs.
2. SQL triggers are a special case of stored procedure since it is fired upon a data modification event.
3. Triggers are only fired when an Insert, Delete, or Update statement are used.
4. In MySQL there can be a max of 6 triggers for each table.
5. **Advantages**

- a. Provide a way to check for data integrity
- b. Can catch errors in business logic
- c. Provide a way to run scheduled task as it is run before or after a change is made.
- d. Very useful to keep an audit log of changes

6. Disadvantages

- a. Triggers can only provide an extended validation and cannot replace all validation as some has to be done in the App layer.
- b. May increase the overhead on the db server.
- c. They cannot Use **SHOW, LOAD DATA, LOAD TABLE, BACKUP DATABASE, RESTORE, FLUSH** and **RETURN** statements
- d. They cannot Use statements that commit or rollback implicitly or explicitly such as **COMMIT, ROLLBACK, START TRANSACTION, LOCK/UNLOCK TABLES, ALTER, CREATE, DROP, RENAME**.
- e. They cannot use prepared statements such as **PREPARE** and **EXECUTE**.

7. What are the trigger events?

- a. **BEFORE INSERT** – activated before data is inserted into the table.
- b. **AFTER INSERT** – activated after data is inserted into the table.
- c. **BEFORE UPDATE** – activated before data in the table is updated.
- d. **AFTER UPDATE** – activated after data in the table is updated.
- e. **BEFORE DELETE** – activated before data is removed from the table.
- f. **AFTER DELETE** – activated after data is removed from the table.

Syntax

```
CREATE TRIGGER trigger_name trigger_time trigger_event
ON table_name
FOR EACH row
begin
...
end;
```

8. **trigger_time** – Specify BEFORE or AFTER the change

9. **trigger_event** – This can be either INSERT, UPDATE, or DELETE. To have a trigger handling multiple events, each event must have a trigger associated with it.

Example

```
DELIMITER $$
CREATE TRIGGER before_employee_update
BEFORE
UPDATE
```

```

ON employees FOR EACH row begin
INSERT INTO employees_audit SET action = 'update',
    employeenumber = old.employeenumber,
    lastname = old.lastname,
    changedate = now();

END$$
delimiter ;

```

Create Multiple Triggers for The Same Trigger Event and Action Time

Notes

1. MySQL will invoke multiple triggers in the order they were created for the same event in a table.
2. You can specify **FOLLOWS** or **PRECEDES** after the **FOR EACH ROW** to specify the order in which the trigger should fire.

Syntax

```

DELIMITER $$
CREATE TRIGGER trigger_name
    [BEFORE|AFTER]
    [ INSERT|    UPDATE|    DELETE]
    ON table_name
FOR EACH row [FOLLOWS|PRECEDES] existing_trigger_name
begin
...
end$$
delimiter;

```

Working with MySQL Scheduled Events

Notes

1. MySQL event is a task that runs on a predefined schedule or scheduled event.
2. Events can be triggered once or more in intervals.
3. MySQL event is also known as a temporal trigger.
4. MySQL uses a special thread called **event schedule** to execute events.
5. To see the event scheduler process, use **SHOW PROCESSLIST**;

6. To stop the event from happening execute **SET GLOBAL event_scheduler=OFF;**

Creating new MySQL events

Syntax

```
CREATE event
[IF NOT EXIST] event_name
ON schedule schedule
do
event_body
```

Notes

1. **event_name** should be unique within a database
2. **ON SCHEDULE** – Here you specify if it's a one-time event then use syntax AT timestamp[+INTERVAL] or a recurring one you use EVERY interval STARTS timestamp [+INTERVAL] ENDS timestamp [+INTERVAL].
3. **event_body** - Here you specify SQL statements and you can call a stored procedure here as well.

Example 1

```
CREATE event
IF NOT EXISTS test_event_01
ON schedule at CURRENT_TIMESTAMP
do
INSERT INTO messages
(
message,
created_at
)
VALUES
(
'Test MySQL Event 1',
now()
);

/* Recurring Event example */

CREATE event test_event_03
ON schedule every 1 minute
starts CURRENT_TIMESTAMP
ends CURRENT_TIMESTAMP + INTERVAL 1 hour
do
```

```
INSERT INTO messages
(
    message,
    created_at
)
VALUES
(
    'Test MySQL recurring Event',
    now()
);
```

Modifying MySQL Events

Notes

1. MySQL allows you to change various attributes to existing event using **ALTER EVENT**.

```
ALTER event event_name
ON schedule schedule
ON completion [NOT] preserve
RENAME TO new_event_name
enable | disable
do
event_body
```

Rename events

Syntax

```
ALTER event old_event_name RENAME TO new_event_name;
```

Move events to another database

```
ALTER event oldDb.old_event_name RENAME TO newDB.new_event_name;
```

Section 11. MySQL Administration

Getting Started with MySQL Access Control System

Notes

1. MySQL has a sophisticated user account system
2. MySQL access control has 2 stages when a client connects to the server
 - a. **Connection Verification** – A client needs to have credentials and the host from which the client connects must match the host on the MySQL grant table.
 - b. **Request Verification** – Each request made by a client is checked by MySQL to ensure the user has sufficient privileges to execute a particular statement.
3. There is a database called *mysql* that contains the 5 main grant tables etc.
4. The different grant tables are as follows:
 - a. **user:** contains user account and global privileges. MySQL uses the user table to accept or reject a connection from host. Any privilege granted here is effective to all databases.
 - b. **db:** Contains all database level privileges. A privilege granted here applied to all database objects such as tables and triggers.
 - c. **table_priv & columns_priv**- Contains all table and column level privileges.
 - d. **procs_priv** - contains stored functions and stored procedures privileges

Creating Users via MySQL CREATE USER Statement

Notes

1. A user account has a username and a host name separated by @ character.

```
CREATE user user_account identified BY password;
```

2. **user_account** is in the format 'username'@'hostname'

Example

```
CREATE user dbadmin@localhost identified BY 'secret';
```

3. To view the privileges of a user account use SHOW GRANTS statement. The *. * in the result means that the user can only login and have no other privileges.

```
SHOW grants FOR dbadmin@localhost;
```

4. To allow user to connect from any host, you can use (%) wildcard.

```
CREATE user superadmin@ '%' identified BY 'secret';
```

Best Ways to Change MySQL User Password by Examples

Notes

1. Change MySQL user password using UPDATE statement

Syntax

```
USE mysql;

UPDATE user
SET authentication_string = Password('dolphin')
WHERE user = 'dbadmin'
AND host = 'localhost';

FLUSH privileges;
```

2. Change MySQL user password using the SET PASSWORD statement

```
SET password FOR 'dbadmin'@'localhost' = bigshark;
```

3. Use the ALTER USER statement

Change MySQL user password using ALTER USER statement

```
ALTER USER dbadmin@localhost identified BY 'littlewhale';
```

How to Use MySQL GRANT Statement to Grant Privileges to a User

Notes

1. To give user privileges use GRANT and the syntax below

```
GRANT privilege, [privilege], ... ON privilege_level
TO USER [IDENTIFIED BY password]
[require tsl_option]
[WITH [grant_option | resource_option]];
```

2. If you are specifying more than 1 privilege then separate them by a comma.
3. **privilege_level** determines the level to apply the grants to. MySQL supports global specified by (*.*), database (**database.***), table (**database.table**), and columns which you must specify each one in a list.
4. If user exists **GRANT** will update the privileges for that user else, it creates a new user.
5. **WITH GRANT OPTION** allows you to grant other users or remove others the privileges you possess as well as allocate MySQL server resources to set how many connections/statements the user can use per hour.
6. How to grant all privileges to all databases?

```
GRANT ALL ON *.* TO 'super'@'localhost' WITH GRANT OPTION;
```

Privilege	Meaning	Level					
		Global	Database	Table	Column	Procedure	Proxy
ALL [PRIVILEGES]	Grant all privileges at specified access level except GRANT OPTION						
ALTER	Allow user to use of ALTER TABLE statement	X	X	X			
ALTER ROUTINE	Allow user to alter or drop stored routine	X	X				X
CREATE	Allow user to create database and table	X	X	X			
CREATE ROUTINE	Allow user to create stored routine	X	X				
CREATE TABLESPACE	Allow user to create, alter or drop tablespaces and log file groups	X					
CREATE TEMPORARY TABLES	Allow user to create temporary table by using CREATE TEMPORARY TABLE	X	X				
CREATE USER	Allow user to use the CREATE USER, DROP USER, RENAME USER, and REVOKE ALL PRIVILEGES statements.	X					
CREATE VIEW	Allow user to create or modify view.	X	X	X			
DELETE	Allow user to use DELETE	X	X	X			
DROP	Allow user to drop database, table and view	X	X	X			
EVENT	Enable use of events for the Event Scheduler.	X	X				
EXECUTE	Allow user to execute stored routines	X	X	X			
FILE	Allow user to read any file in the database directory.	X					
GRANT OPTION	Allow user to have privileges to grant or revoke privileges from other accounts.	X	X	X		X	X
INDEX	Allow user to create or remove indexes.	X	X	X			

INSERT	Allow user to use INSERT statement	X	X	X	X
LOCK TABLES	Allow user to use LOCK TABLES on tables for which you have the SELECT privilege	X	X		
PROCESS	Allow user to see all processes with SHOW PROCESSLIST statement.	X			
PROXY	Enable user proxying.				
REFERENCES	Allow user to create foreign key	X	X	X	X
RELOAD	Allow user to use FLUSH operations	X			
REPLICATION CLIENT	Allow user to query to see where master or slave servers are	X			
REPLICATION SLAVE	Allow the user to use replicate slaves to read binary log events from the master.	X			
SELECT	Allow user to use SELECT statement	X	X	X	X
SHOW DATABASES	Allow user to show all databases	X			
SHOW VIEW	Allow user to use SHOW CREATE VIEW statement	X	X	X	
SHUTDOWN	Allow user to use mysqladmin shutdown command	X			
SUPER	Allow user to use other administrative operations such as CHANGE MASTER TO, KILL, PURGE BINARY LOGS, SET GLOBAL, and mysqladmin command	X			
TRIGGER	Allow user to use TRIGGER operations.	X	X	X	
UPDATE	Allow user to use UPDATE statement	X	X	X	X
USAGE	Equivalent to “no privileges”				

Revoking Privileges from Users Using MySQL REVOKE

Syntax

```
REVOKE privilege_type [(column_list)] [, priv_type  
[(column_list)]]... ON [object_type] privilege_level FROM USER  
[, user]...
```

Notes

1. First specify a list of privileges you want to revoke from a user after **REVOKE**.
2. Second, specify the privilege level at which privileges is revoked in the **ON** clause.
3. Third, specify the user account that you want to revoke the privileges in the **FROM** clause.
4. **Note** that to revoke privileges from a user account, you must have **GRANT OPTION** privilege and the privileges that you are revoking.

Example

```
REVOKE UPDATE, DELETE ON classicmodels.* FROM rfc;  
  
GRANT SELECT ON 'classicmodels'.* TO 'rfc'@'%'
```

MySQL DROP USER statement

Syntax

```
DROP user [IF EXISTS] user, [user], ...;
```

Notes

1. You specify the account name in the format 'user_name'@'host_name'
2. You can remove more than 1 user by separating with commas.
3. Can remove all privileges etc. from a user.

How to back up a MySQL database

Syntax

```
mysqldump u [username]  
p[password]  
[database_name] > [dump_file.sql]
```

Notes

1. **[username]**: valid MySQL username.
2. **[password]**: valid password for the user. Note that there is no space between `-p` and the password.
3. **[database_name]**: database name you want to backup
4. **[dump_file.sql]**: dump file you want to generate.

Example

```
mysqldump -u  
mysqltutorial -psecret classicmodels > C:\temp\backup001.SQL
```

5. How to backup MySQL database structure only?
 - a. If you only want to backup database structure only you just need to add an option `-no-data` to tell mysqldump that only database structure needs to export as follows:

```
mysqldump -u [username] -p[password] -no-  
data [database_name] > [dump_file.sql]
```