DAY 2

Day 2 Contents

- DML (INSERT / REPLACE / UPDATE / DELETE/ ..)
- DRL (SELECT /WHERE/GROUP BY/ORDER BY/ ..)
- Subqueries.
- Union
- Views
- Transactions

Inserting Data

- Strings should always be enclosed in pairs of single or double quotation marks in MySQL. Numbers and dates do not need quotes.

```
INSERT INTO table [(column1, column2, column3,...)] VALUES
(value1, value2, value3,...);
INSERT INTO table
set column1 = ’ value1’,
column2 = ’ value2’,
column3 = ’ value3’;
```

Inserting Data

- In an INSERT statement, if you try to insert a row that contains a unique index or primary key value that already exists in the table, you aren't able to add that row. A REPLACE statement, however, **DELETES** the old row and adds the new row

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

Updating Data


```
UPDATE [LOW_PRIORITY] [IGNORE] tablename
SET column1=expression1,column2=expression2,...
[WHERE condition]
[ORDER BY order_criteria]
[LIMIT number]
```

Deleting Data


```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE] FROM table
[WHERE condition]
[ORDER BY order_cols]
[LIMIT number]
```

If you write

```
delete from table;
```

on its own, all the rows in a table will be deleted, **so be careful**!

TRUNCATE


```
TRUNCATE TABLE table_name;
```

- TRUNCATE statement **starts the AUTO_INCREMENT** count over again, unlike the DELETE statement.
- **TRUNCATE** is generally **faster** than using a **DELETE** statement as well.

Select

- PostgreSQL **SELECT** statement is used to fetch the data from a database table
- The basic syntax of SELECT statement is as follows:

```
SELECT column1, column2, columnN FROM table_name;
```

If you want to fetch all the fields available in the table then you can use the following syntax:

```
SELECT * FROM table_name;
```

Select


```
SELECT column1, column2

FROM table1

[ WHERE  conditions ]

[ GROUP BY column ]

[ HAVING conditions ]

[ ORDER BY column ]

[LIMIT no of rows]
```

Where

- WHERE clause is used to specify a condition while fetching the data from single table or joining with multiple tables.
- Example

```
SELECT * FROM COMPANY WHERE AGE >= 25 AND SALARY >= 65000;
```

Operators

- Operators are used to specify conditions in a PostgreSQL statement and to serve as conjunctions for multiple conditions in a statement.
- It can be classified into:
  - Arithmetic operators
  - Comparison operators
  - Logical operators

Arithmetic Operators

OP	Description
+	Addition - Adds values on either side of the operator
-	Subtraction - Subtracts right hand operand from left hand operand
*	Multiplication - Multiplies values on either side of the operator
/	Division - Divides left hand operand by right hand operand
%	Modulus - Divides left hand operand by right hand operand and returns remainder
^	Exponentiation - This gives the exponent value of the right hand operand, 2.0 ^ 3.0 =8
|/	square root, |/ 25.0=5
||/	Cube root, ||/ 27.0 =3
!/	Factorial, 5 ! =120

Comparison Operators

OP	Description
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.
!= <>	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.

Logical Operators

OP	Description
AND	The AND operator allows the existence of multiple conditions in a PostgresSQL statement's WHERE clause.
OR	The OR operator is used to combine multiple conditions in a PostgresSQL statement's WHERE clause.
NOT	The NOT operator reverses the meaning of the logical operator with which it is used. Eg. NOT EXISTS, NOT BETWEEN, NOT IN etc. This is negate operator.

Others Operators


```
SELECT * FROM COMPANY WHERE AGE IS NULL;
SELECT * FROM COMPANY WHERE AGE NOT IS NULL;
SELECT * FROM COMPANY WHERE AGE IS NOT NULL;

SELECT * FROM COMPANY WHERE AGE IN ( 25, 27 );
SELECT * FROM COMPANY WHERE AGE NOT IN ( 25, 27 );

SELECT * FROM COMPANY WHERE AGE BETWEEN 25 AND 27;
SELECT * FROM Orders WHERE OrderDate BETWEEN #07/04/1996#
AND #07/09/1996#;
```

Like Operators

Statement	Description
WHERE SALARY LIKE '200%'	Finds any values that start with 200
WHERE SALARY LIKE '%2'	Finds any values that end with 2
WHERE SALARY LIKE '_2%3'	Finds any values that have a 2 in the second position and end with a 3

Expressions

- The expression is a combination of one or more values with previous operators.
- Examples:

```
SELECT * FROM COMPANY WHERE SALARY = 10000;

SELECT * FROM COMPANY WHERE (SALARY + 6) = 1005;

SELECT * FROM COMPANY WHERE (SALARY * Age) >= 1005;
```

Aggregation Functions

- compute a single result from a set of input values

Such as: COUNT(), MAX(), MIN(), AVG(), SUM()

- Examples:

```
SELECT MAX(salary) FROM COMPANY;

SELECT MIN(salary) FROM COMPANY;

SELECT AVG(SALARY) FROM COMPANY;

SELECT SUM(salary) FROM company;

SELECT COUNT(salary) FROM COMPANY;
```

GROUP BY

The GROUP BY statement is used in conjunction with the aggregate functions to group the result-set by one or more columns. This is done to eliminate redundancy in the output and/or **compute aggregates** that apply to these groups:

```
SELECT NAME, SUM(SALARY) FROM COMPANY GROUP BY NAME;
```

HAVING

The HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions.

```
SELECT NAME FROM COMPANY GROUP BY name HAVING count(name)
< 2;
```

ORDER BY

- is used to sort the data in ascending or descending order, based on one or more columns:

```
SELECT * FROM COMPANY ORDER BY AGE ASC;        [default]

SELECT * FROM COMPANY ORDER BY NAME DESC;
```

LIMIT

- LIMIT is used to limit the data amount returned by the SELECT statement, OFFSET rows are skipped before starting to count the LIMIT rows that are returned.

```
SELECT * FROM COMPANY LIMIT 4;

SELECT * FROM COMPANY LIMIT 3 OFFSET 2;
```

Choosing Which Rows to Return

```
select name
from customers
limit 2, 3;
```

- This query can be read as, “Select name from customers, and then return 3 rows, starting from row 2 in the output.” Note that row numbers are zero indexed; that is, the first row in the output is row number zero.

SELECT DISTINCT

- SELECT DISTINCT is used to eliminate all the duplicate records and fetching only unique records:

```
SELECT DISTINCT name FROM COMPANY;

SELECT DISTINCT name, age FROM COMPANY;
```

Case Expression

The CASE expression is a generic conditional expression, similar to if/else statements in other programming languages

```
CASE WHEN condition THEN result
     [WHEN condition THEN result]
     [ELSE result]
END
```

```
SELECT column1,column2,[CASE Expression] FROM table
```

Case Expression


```
CREATE TABLE test (a integer);

SELECT a,
       CASE WHEN a=1 THEN 'one'
            WHEN a=2 THEN 'two'
            ELSE 'other'
       END
FROM test;
```

Database Engine

Storage Engine: is responsible for executing SQL Statements, and the way to store the Data/rows

Engine	Support	Comment	Transactions
InnoDB	DEFAULT	Percona-XtraDB, Supports transactions, row-level locking, and foreign keys	YES
CSV	YES	CSV storage engine	NO
MRG_MYISAM	YES	Collection of identical MyISAM tables	NO
BLACKHOLE	YES	/dev/null storage engine (anything you write to it disappears)	NO
MEMORY	YES	Hash based, stored in memory, useful for temporary tables	NO

Altering Tables After Creation

The basic form of this statement is:

```
ALTER TABLE tablename alteration [, alteration ...]
ALTER TABLE tbl_name ENGINE = engine_name;
```

ALTER

- Alter is used to change the definition of a table.

```
ALTER TABLE table_name action [, ... ]

ALTER TABLE table_name RENAME TO new_name

ALTER TABLE table_name ADD column data_type

ALTER TABLE table_name DROP column

ALTER TABLE table_name ALTER column SET DATA TYPE data_type

ALTER TABLE table_name RENAME COLUMN column TO new_column

ALTER TABLE table_name ADD CONSTRAINT my_fk FOREIGN KEY (col1)

REFERENCES foreign_table (foreign_field)  ON DELETE CASCADE;
```

Creating Tables

- The general form of a CREATE TABLE statement is:

```
CREATE TABLE [IF NOT EXISTS] tbl_name (column_specs) [ENGINE =
   engine_name];
```

- Show table definition:

```
SHOW CREATE TABLE t\G
```

- To create an empty copy of an existing table:

```
CREATE TABLE new_tbl_name LIKE tbl_name;
```

Creating Tables

- To create an empty copy of a table and then populate it from the original table:

```
CREATE TABLE new_tbl_name LIKE tbl_name;
INSERT INTO new_tbl_name SELECT * FROM tbl_name;
```

- To create a temporary table which disappears automatically when your connection to the server terminates:

```
CREATE TEMPORARY TABLE tbl_name;
```

Renaming Tables

- To rename a table you can use alter:

```
ALTER TABLE tbl_name RENAME TO new_tbl_name;
```

- Also you can use rename:

```
RENAME TABLE old_name TO new_name [,…,…];
```

Dropping Tables

- The basic form of this statement to drop table is:

```
DROP TABLE tbl_name;
DROP TABLE IF EXISTS tbl_name;
```

- Drop Temporary table:

```
DROP TEMPORARY TABLE tbl_name;
```

- To Drop multiple tables:

```
DROP TABLE tbl_name1, tbl_name2, ... ;
```

UNION

- UNION clause/operator is used to combine the results of two or more SELECT statements without returning any duplicate rows.
- each UNION query must have the same number of columns

```
SELECT EMP_ID, NAME, DEPT FROM COMPANY
UNION
SELECT EMP_ID, NAME, DEPT FROM COMPANY1;
```

Subqueries

- A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.
- Subqueries that return more than one row can only be used with multiple value operators, such as IN, NOT IN operator

```
SELECT * FROM COMPANY WHERE ID IN (SELECT ID
    FROM COMPANY_bkp WHERE SALARY > 45000);
```

```
INSERT INTO COMPANY_BKP (SELECT * FROM COMPANY);
```

Subqueries

Name	Sample Syntax	Description
ANY	SELECT c1 FROM t1 WHERE c1 > ANY (SELECT c1 FROM t2);	Returns true if the comparison is true for any of the rows in the subquery.
IN	SELECT c1 FROM t1 WHERE c1 IN (SELECT c1 from t2);	Equivalent to =ANY.
SOME	SELECT c1 FROM t1 WHERE c1 > SOME (SELECT c1 FROM t2);!	Alias for ANY; sometimes reads better to the human ear
ALL	SELECT c1 FROM t1 WHERE c1 > ALL (SELECT c1 from t2);	Returns true if the comparison is true for all of the rows in the subquery.

Views

- A view can represent a subset of a real table, selecting certain columns or certain rows from an ordinary table.
- A view can even represent joined tables.
- Views are not real tables, but appear as ordinary tables to SELECT.

Views

- The general form of view creation is:

```
CREATE [TEMPORARY] VIEW view_name AS
SELECT column1, column2.....
FROM table_name
[WHERE condition];
```

- TEMPORARY: the view will be created in the temporary space. Temporary views are automatically dropped at the end of the current session.

Views

- Example:

```
CREATE VIEW vCOMPANY AS

SELECT ID, NAME, AGE

FROM  COMPANY;

SELECT * FROM vCOMPANY;
```

- To drop view

```
DROP VIEW vCOMPANY;
```

Updatable Views

- To create an updateable view, the SELECT statement which defines view has to follow several rules as follow:
  - SELECT statement must not reference to more than one table. It means it **must not** contain more than **one table** in **FROM** clause, other tables in **JOIN** statement, or **UNION** with other tables.
  - SELECT statement **must not** use **GROUP BY** or **HAVING** clause.
  - SELECT statement **must not** use **DISTINCT** in the selection list.
  - SELECT statement **must not** reference to the **view** that is **not updatable**
  - SELECT statement **must not** contain any **expression** (aggregates, functions, computed columns...)

Updatable Views

- Create updatable view:

```
CREATE VIEW vOfficeInfo
AS SELECT officeCode, phone, city
FROM offices
```

- Then you can run update statement

```
UPDATE vOfficeInfo
 SET phone = '+33 14 723 5555'
WHERE officeCode = 4
```

Views

Views Advantages:

- **Simplify complex query.** A view is defined by an SQL statement with join on many underlying tables with complex business logic. The view now can be used to hides the complexity of underlying tables to the end users and external applications. Only simple SQL statement is used to work with view.
- **Limited access data to the specific users.** You may don't want a subset of sensitive data can be retrievable by all users (both human and applications). You can use view to expose what data to which user to limit the access.

Transactions

- The example of this involves a financial transfer where money from one account is placed into another account. Suppose that Moataz writes a check to ali for $100.00 and ali cashes the check. Moataz's account should be decremented by $100.00 and ali account incremented by the same amount:

```
UPDATE account SET balance = balance - 100 WHERE name =
'Moataz';
UPDATE account SET balance = balance + 100 WHERE name =
'ali';
```

Transactions

- If a crash occurs between the two statements, the operation is incomplete.

  Depending on which statement executes first, Moataz is $100 short without Islam having been credited, or Islam is given $100 without Motaz having been debited.
- Another use for transactions is to make sure that the rows involved in an operation are not modified by other clients while you're working with them.

Transactions

- A transaction is a set of SQL statements that execute as a unit. Either all the statements execute successfully, or none of them have any effect.
- This is achieved through the use of commit and rollback capabilities. If all of the statements in the transaction succeed, you commit it to record their effects permanently in the database. If an error occurs during the transaction, you roll it back to cancel it. Any statements executed up to that point within the transaction are undone, leaving the database in the state it was in prior to the point at which the transaction began.

Transactions

One way to perform a transaction is to issue a **`BEGIN TRANSACTION`** (or **`BEGIN`**) statement, execute the statements that make up the transaction, and end the transaction with a **`COMMIT`** or (**`END TRANSACTION`**) statement to make the changes permanent.

If an error occurs during the transaction, cancel it by issuing a **`ROLLBACK`** statement instead to undo the changes.

Transactions

- The following example illustrates this approach. First, create a table to use

```
CREATE TABLE t (name CHAR(20), UNIQUE (name));
```

- The statement creates a table, Next, initiate a transaction with BEGIN TRANSACTION, add a couple of rows to the table, commit the transaction, and then see what the table looks like:

```
BEGIN TRANSACTION;

INSERT INTO t SET name = ‘Islam';

INSERT INTO t SET name = ‘Moataz';

COMMIT;

SELECT * FROM t;
```

Transactions

- if you issue any of the following statements while a transaction is in progress, the server wait until the transaction ended before executing the statement:

```
ALTER TABLE

CREATE INDEX

DROP DATABASE

DROP INDEX

DROP TABLE

RENAME TABLE

TRUNCATE TABLE
```

Transactions

- Postgres enables you to perform a partial rollback of a transaction. To do this, issue a SAVEPOINT statement within the transaction to set a marker. To roll back to just that point in the transaction later, use a ROLLBACK statement that names the savepoint.

```
CREATE TABLE t (name CHAR(20), UNIQUE (name));
```

```
CREATE TABLE t (i INT);
BEGIN TRANSACTION;
INSERT INTO t VALUES(1);
SAVEPOINT my_savepoint;
INSERT INTO t VALUES(2);
ROLLBACK TO SAVEPOINT my_savepoint;
INSERT INTO t VALUES(3);
COMMIT;
```

Transactions

- To use transactions, you *must* use a transactional storage engine such as InnoDB Engines such as MyISAM and MEMORY will not work.
- By default, MySQL runs in **autocommit** *mode*, which means that changes made by individual statements are committed to the database immediately to make them permanent.

Transactions

- To perform transactions explicitly, **disable autocommit** mode and then tell MySQL when to commit or roll back changes.
- One way to perform a transaction is to issue a `START TRANSACTION` (or BEGIN) statement to suspend autocommit mode, execute the statements that make up the transaction, and end the transaction with a `COMMIT` statement to make the changes permanent. If an error occurs during the transaction, cancel it by issuing a ROLLBACK statement instead to undo the changes.

Transactions

- Another way to perform transactions is to manipulate the autocommit mode directly using SET statements:

```
SET autocommit = 0;
SET autocommit = 1;
```

- Setting the autocommit variable to zero disables autocommit mode. The effect of any statements that follow becomes part of the current transaction, which you end by issuing a COMMIT or ROLLBACK statement to commit or cancel it. With this method, autocommit mode remains off until you turn it back on, so ending one transaction also begins the next one. You can also commit a transaction by re-enabling autocommit mode.

Transactions

- To see how this approach works, begin with the same table as for the previous examples:

```
DROP TABLE t;
CREATE TABLE t (name CHAR(20), UNIQUE (name)) ENGINE =
InnoDB;
```

- Then disable autocommit mode, insert some rows, and commit the transaction:

```
SET autocommit = 0;
INSERT INTO t SET name = ‘Islam';
INSERT INTO t SET name = ‘Moataz';
COMMIT;
SELECT * FROM t;
```

Transactions SavePoint

- MySQL enables you to perform a partial rollback of a transaction. To do this, issue a SAVEPOINT statement within the transaction to set a marker.To roll back to just that point in the transaction later, use a ROLLBACK statement that names the savepoint. The following statements illustrate how this works:

```
CREATE TABLE t (i INT) ENGINE = InnoDB;
START TRANSACTION;
INSERT INTO t VALUES(1);
SAVEPOINT my_savepoint;
INSERT INTO t VALUES(2);
ROLLBACK TO SAVEPOINT my_savepoint;
INSERT INTO t VALUES(3);
COMMIT;
```