Database Management Systems.

Lecture 2

Content:

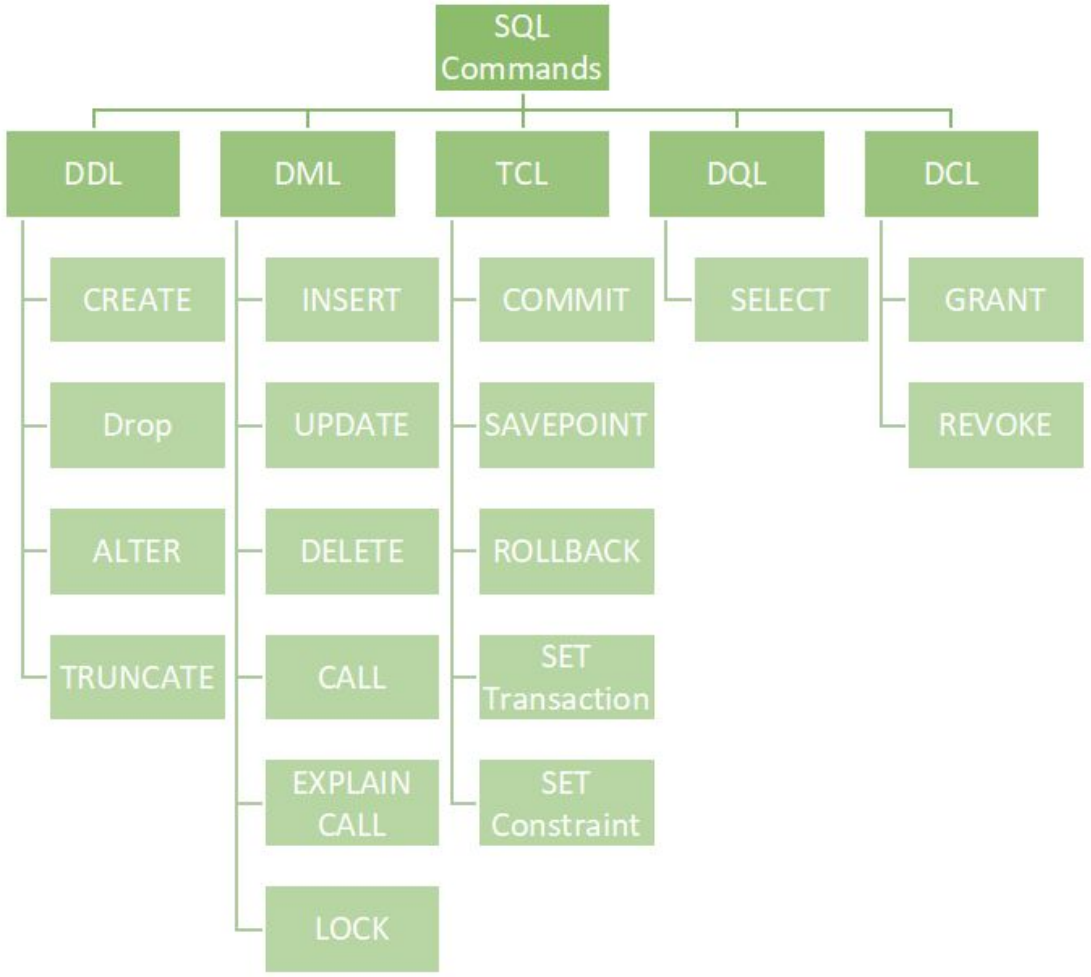
- Intro to SQL DDL statements;
- Managing Tables;
- Constraints;
- Primary and Foreign Keys.

SQL definition:

- **Structured Query Language(SQL)** as we all know is the database language using which we can perform certain operations on the existing database and, we can use this language to create a database.

- **SQL commands** are mainly categorized into **five** categories as:
- **DDL** – Data Definition Language;
- **DML** – Data Manipulation Language;
- **DCL** – Data Control Language;
- **DQL** – Data Query Language;
- **TCL** - Transaction Control Language;

SQL Commands



DDL (Data Definition Language):

- **DDL** or **Data Definition Language** consists of the SQL commands that can be used to define the database schema.
- **DDL** is a set of SQL commands used to **create**, **modify,** and **delete** database structures **but not data**.

DDL Commands:

- **CREATE**: This command is used to create the database or its objects (like table, index, function, views, store procedure, and triggers).
- **DROP**: This command is used to delete objects from the database.
- **ALTER:** This is used to alter the structure of the database.
- **TRUNCATE:** This is used to remove all records from a table, including all spaces allocated for the records are removed.
- **COMMENT**: This is used to add comments to the data dictionary.
- **RENAME:** This is used to rename an object existing in the database.

CREATE syntax:

- There are two **CREATE** statements available in SQL:

CREATE DATABASE:
`CREATE DATABASE database_name;`
	CREATE TABLE:
	`CREATE TABLE table_name (`
	`column1 data_type(size),`
	`column2 data_type(size),`
	`column3 data_type(size),`
	`.... );`

Data Types:

- **PostgreSQL supports the following data types:**
- **Boolean**
- Character types such as **char**, **varchar**, and **text**.
- Numeric types such as **integer, float** and **numeric**.
- Temporal types such as **date**, **time**, **timestamp.**
- **UUID** for storing Universally Unique Identifiers
- **Array** for storing array strings, numbers, etc.
- **JSON** stores JSON data
- **hstore** stores key-value pair
- Special types such as network address and geometric data.

PostgreSQL Boolean:

- **PostgreSQL supports a single Boolean data type:**
- **BOOLEAN** that can have three values: **true**, **false** and **NULL**.
- PostgreSQL uses **one byte** for storing a Boolean value in the database.
- The BOOLEAN can be abbreviated as BOOL.
- In standard SQL, a Boolean value can be TRUE, FALSE, or NULL. However, PostgreSQL is quite flexible when dealing with TRUE and FALSE values.
- The following table shows the valid literal values for TRUE and FALSE in PostgreSQL.

True	False
true	false
't'	'f '
'true'	'false'
'y'	'n'
'yes'	'no'
'1'	'0'

PostgreSQL Character Types:

Character Types	Description
`CHARACTER VARYING(n), VARCHAR(n)`	variable-length with length limit
`CHARACTER(n), CHAR(n)`	fixed-length, blank padded
`TEXT, VARCHAR`	variable unlimited length

- Both **CHAR(n)** and **VARCHAR(n)** can store up to **n** characters. If you try to store a string that has more than n characters, PostgreSQL **will issue an error.**
- However, one exception is that if the excessive characters are all spaces, PostgreSQL truncates the spaces to the maximum length (n) and stores the characters.
- The **TEXT** data type can store a string with **unlimited length**.
- If you do not specify the **n** integer for the **VARCHAR** data type, it behaves like the **TEXT** datatype. **The performance of the VARCHAR (without the size n) and TEXT are the same.**
- The **only advantage** of specifying the length specifier for the **VARCHAR** data type is that PostgreSQL **will issue an error** if you attempt to insert a string that has more than n characters into the VARCHAR(n) column.
- Unlike VARCHAR, The **CHARACTER** or **CHAR** without the length specifier (n) is the same as the **CHARACTER(1)** or **CHAR(1).**
- Different from other database systems, in PostgreSQL, there is **no performance difference among three character types.**

In most cases, you should use TEXT or VARCHAR. And you use the VARCHAR(n) when you want PostgreSQL to check for the length.

PostgreSQL Integer Data Types:

- To store the whole numbers in PostgreSQL, you use one of the following integer types: **SMALLINT, INTEGER, and BIGINT.**

Name	Storage Size	Min	Max
`SMALLINT`	2 bytes	-32,768	+32,767
`INTEGER`	4 bytes	-2,147,483,648	+2,147,483,647
`BIGINT`	8 bytes	-9,223,372,036,854,775,808	+9,223,372,036,854,775,807

SERIAL and AUTOINCREMENT:

- In PostgreSQL, a **sequence** is a special kind of database object that generates a sequence of integers. A sequence is often used as the primary key column in a table.
- When creating a new table, the sequence can be created through the **SERIAL** pseudo-type as follows:

```
CREATE TABLE table_name(
    id SERIAL
);
```

- **By assigning the SERIAL pseudo-type to the id column, PostgreSQL performs the following:**
- First, **create a sequence** object and set the next value generated by the sequence as the default value for the column.
- Second, **add a NOT NULL constraint** to the id column because a sequence always generates an integer, which is a non-null value.
- Third, **assign the owner of the sequence to the id column**; as a result, the sequence object is deleted when the id column or table is dropped

PostgreSQL NUMERIC Type:

- The **NUMERIC** type can store numbers with a lot of digits. Typically, you use the NUMERIC type for numbers that require exactness such as monetary amounts or quantities.
- The following illustrate the syntax of the NUMERIC type:

```
NUMERIC(precision, scale)
```

- In this syntax, the **precision** is the total number of digits and the scale is the number of digits in the fraction part. For example, the number 1234.567 has the precision 7 and scale 3.
- The **NUMERIC** type can **hold a value up to 131,072 digits before the decimal point 16,383 digits after the decimal point.**
- The scale of the NUMERIC type can be zero or positive. The following shows the syntax of NUMERIC type with scale zero:

```
NUMERIC(precision)
```

- If you omit both precision and scale, you can store any precision and scale up to the limit of the precision and scale mentioned above.

```
NUMERIC
```

- **If precision is not required, you should not use the NUMERIC type because calculations on NUMERIC values are typically slower than integers, floats, and double precisions.**

ALTER syntax:

- **ALTER TABLE** is used to add, delete/drop or modify columns in the existing table. It is also used to add and drop various constraints on the existing table.

- **ALTER TABLE - ADD** is used to add columns or constraints into the existing table:

```
ALTER TABLE table_name
          ADD(Columnname_1 datatype,
              Columnname_2 datatype,
              …
              Columnname_n datatype);
```

- **ALTER TABLE - DROP** is used to drop column in a table. Deleting the unwanted columns from the table:

```
ALTER TABLE table_name DROP COLUMN column_name;
```

DROP vs TRUNCATE:

- **DROP** is used to delete a whole database or just a table.
- The **DROP** statement destroys the objects like an existing database, table, index, or view.
- `DROP object object_name;`

- **TRUNCATE** statement is used to quickly delete all data from large tables.
- The **TRUNCATE TABLE** statement is logically (though not physically) equivalent to the **DELETE FROM** statement (without a WHERE clause).
- `TRUNCATE TABLE table_name;`

RENAME syntax:

- Sometimes we may want to rename our table to give it a more relevant name. For this purpose, we can use **ALTER TABLE** to rename the name of table.

```
ALTER TABLE table_name

RENAME TO new_table_name;
```

or (if we want to change column name):

```
ALTER TABLE table_name

RENAME COLUMN old_column_name TO
new_column_name;
```

COMMENT syntax:

- **COMMENT** is used to store a comment about database object.
- Only one comment string is stored for each object, so to modify a comment, issue a new COMMENT command for the same object.
- Comments are stored in data dictionary.
- Comments are automatically dropped when their object is dropped.
- `COMMENT ON object object_name IS 'some text';`

Constraints:

- Constraints are the rules enforced on data columns on table. These are used to prevent invalid data from being entered into the database. This ensures the accuracy and reliability of the data in the database.
- Constraints could be column level or table level.

- The following are commonly used constraints available in PostgreSQL:
- **NOT NULL Constraint** – Ensures that a column cannot have NULL value.
- **UNIQUE Constraint** – Ensures that all values in a column are different.
- **PRIMARY Key** – Uniquely identifies each row/record in a database table.
- **FOREIGN Key** – Constrains data based on columns in other tables.

NOT NULL:

- By default, a column can hold NULL values.
- If you do not want a column to have a NULL value, then you need to define such constraint on this column specifying that NULL is now not allowed for that column.
- A NOT NULL constraint is always written as a column constraint.

- **To add NOT NULL constraint:**
- `ALTER TABLE table_name ALTER COLUMN column_name SET NOT NULL;`

- **To drop NOT NULL constraint:**
- `ALTER TABLE table_name ALTER COLUMN column_name DROP NOT NULL;`

UNIQUE

- The UNIQUE Constraint prevents two records from having identical values in a particular column.

- **To add UNIQUE constraint on a column:**
- `ALTER TABLE table_name ADD CONSTRAINT constraint_name UNIQUE (column_name);`

- **To add UNIQUE constraint on multiple columns (using index):**
- `CREATE UNIQUE INDEX index_name ON table_name (column1, column2);`

- **To drop UNIQUE constraint:**
- `ALTER TABLE table_name DROP CONSTRAINT constraint_name;`

PRIMARY KEY

- The **PRIMARY KEY** constraint uniquely identifies each record in a database table. We use them to refer to table rows.
- A primary key is a field in a table, which uniquely identifies each row/record in a database table.
- Primary keys must contain **UNIQUE** values. A primary key column **cannot have NULL** values.
- There can be more UNIQUE or NOT NULL columns, but **only one primary key in a table.**
- When multiple fields are used as a primary key, they are called a **composite key**.

```
CREATE TABLE TABLE (
column_1 data_type PRIMARY KEY,
column_2 data_type,
… );
```

FOREIGN KEY:

- A **foreign key** constraint specifies that the values in a column (or a group of columns) must match the values appearing in some row of another table.
- We say this maintains the **referential integrity** between two related tables.
- They are called foreign keys because the constraints are foreign; that is, outside the table.
- Foreign keys are sometimes called a **referencing key.**

```
[CONSTRAINT fk_name]
FOREIGN KEY(fk_columns) REFERENCES
parent_table(parent_key_columns)
[ON DELETE delete_action]
[ON UPDATE update_action]
```

```
ALTER TABLE child_table ADD
CONSTRAINT constraint_name
FOREIGN KEY (fk_columns)
REFERENCES parent_table
(parent_key_columns);
```