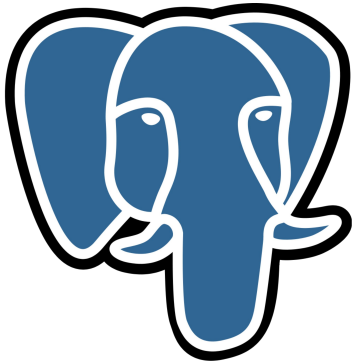


Database | SQL | PostgreSQL

Session 1

- Introduction
- Relational databases (SQL DBs)
- Structured Query Language (SQL)
- PostgreSQL
- PostgreSQL commands
- PostgreSQL Data Types
- Database statements
- Select statements
- Data statements
- Foreign Key

by Mohammad Amin H.B. Tehrani - Reza Yazdani

www.maktabsharif.ir

Introduction

What is a Database?

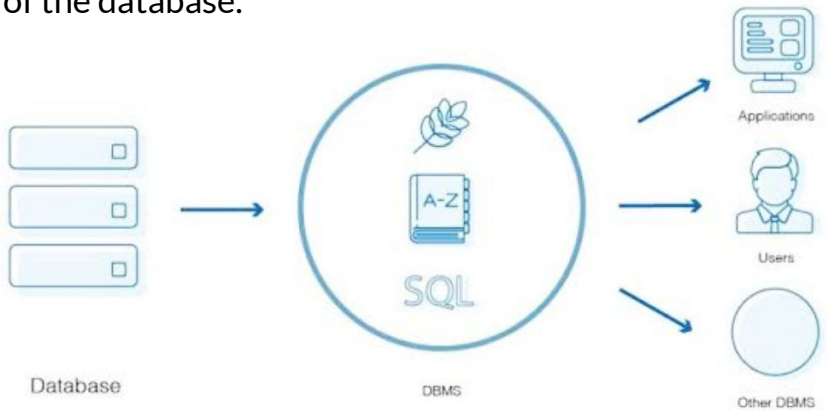
A **database** is an organized collection of structured information. In simple words, a systematic collection of data is called a “Database”. That is because it is made very organized. So that the data can be easily accessed, managed and updated. The basic function of database is to make data management easy.

Let’s understand with an example, suppose a university has to prepare a student record of its students. In which he has to write all the information of every student’s name, class, subject and parents name in a big register with him, so that he can easily get information about a student. So for this, he will write rows, columns, tables in the register number wise in an orderly manner so that he can easily find the information further.

DataBase Management System (DBMS)

DBMS or database management system is a software, which allows us to create data, data store and data update in database. Due to which the user can store, access and analyze data very easily. Dbms provides an interface or tool for this.
a DBMS manages three important things:

- **Database engine:** which access, lock and modify data.
- **Database schema:** This defines the logical structure of the database.
- **Data**



Type of Databases

There are many different types of databases.

*The **best database** for a specific organization depends on how the organization intends to use the data.*

- **Relational databases (SQL databases)**
  Relational databases became dominant in the 1980s. Items in a relational database are organized as a set of tables with columns and rows. Relational database technology provides the most efficient and flexible way to access structured information.

- NoSQL databases
  A NoSQL, or nonrelational database, allows unstructured and semistructured data to be stored and manipulated (in contrast to a relational database, which defines how all data inserted into the database must be composed). NoSQL databases grew popular as web applications became more common and more complex.

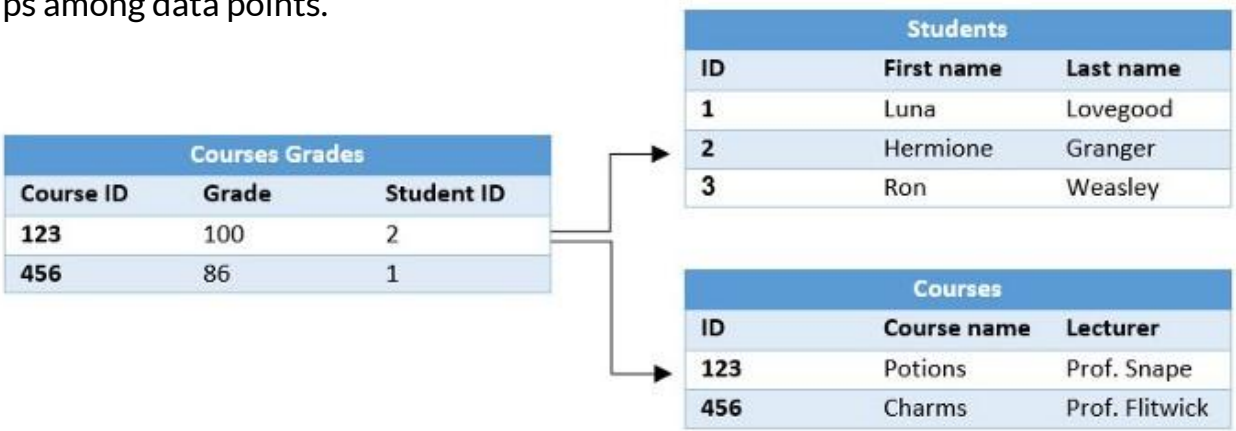
- Object-oriented databases
  Information in an object-oriented database is represented in the form of objects, as in object-oriented programming.

- Graph databases
  A graph database stores data in terms of entities and the relationships between entities.
  OLTP databases. An OLTP database is a speedy, analytic database designed for large numbers of transactions performed by multiple users.

Relational Databases (SQL Databases)

What is a Relational Database?

A **relational database** is a type of database that stores and provides access to data points that are related to one another.
Relational databases are based on the relational model, an intuitive, straightforward way of **representing data in tables**.
In a relational database, each row in the table is a record with a **unique ID called the key.**
The **columns** of the table hold **attributes of the data,** and each record usually has a value for each attribute, making it easy to establish the relationships among data points.



Courses Grades

Course ID	Grade	Student ID
123	100	2
456	86	1

Students

ID	First name	Last name
1	Luna	Lovegood
2	Hermione	Granger
3	Ron	Weasley

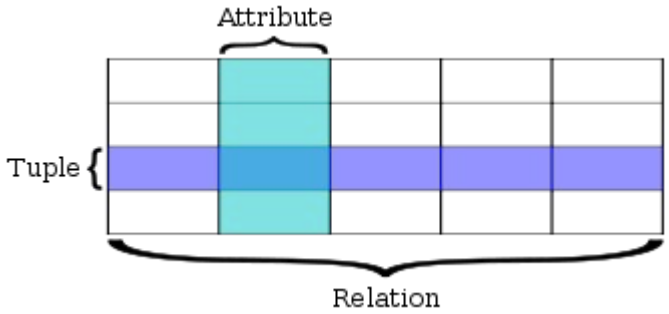
Courses

ID	Course name	Lecturer
123	Potions	Prof. Snape
456	Charms	Prof. Flitwick

Relational model

This model organizes data into one or more **tables (or "relations")** of **columns** and **rows**, with a unique key identifying each row. Rows are also called records or tuples. Columns are also called attributes. Generally, each table/relation represents one "entity type" (such as customer or product). The rows represent instances of that type of entity (such as "Lee" or "chair") and the columns representing values attributed to that instance (such as address or price).

SQL term	Relational database term	Description
Row	**Tuple** or **record**	A data set representing a single item
Column	**Attribute** or **field**	A labeled element of a tuple, e.g. "Address" or "Date of birth"
Table	**Relation**	A set of tuples sharing the same attributes; a set of columns and rows



Structured Query Language (SQL)

What is SQL?

SQL is a standard language for accessing and manipulating databases.
Structured Query Language) is a domain-specific language used in programming and designed for managing data held in a **relational database management system (RDBMS),** or for stream processing in a relational data stream management system (RDSMS). It is particularly useful in handling structured data, i.e. data incorporating relations among entities and variables.

What is SQL?

- SQL stands for Structured Query Language
- SQL lets you access and manipulate databases
- SQL became a standard of the American National Standards Institute (ANSI) in 1986, and of the International Organization for Standardization (ISO) in 1987

What Can SQL do?

- SQL can execute queries against a database
- SQL can retrieve data from a database
- SQL can insert records in a database
- SQL can update records in a database
- SQL can delete records from a database
- SQL can create new databases

Note: Although SQL is an ANSI/ISO standard, there are **different versions** of the SQL language.
However, to be compliant with the ANSI standard, they all support at least the major commands in a similar manner.

SQL

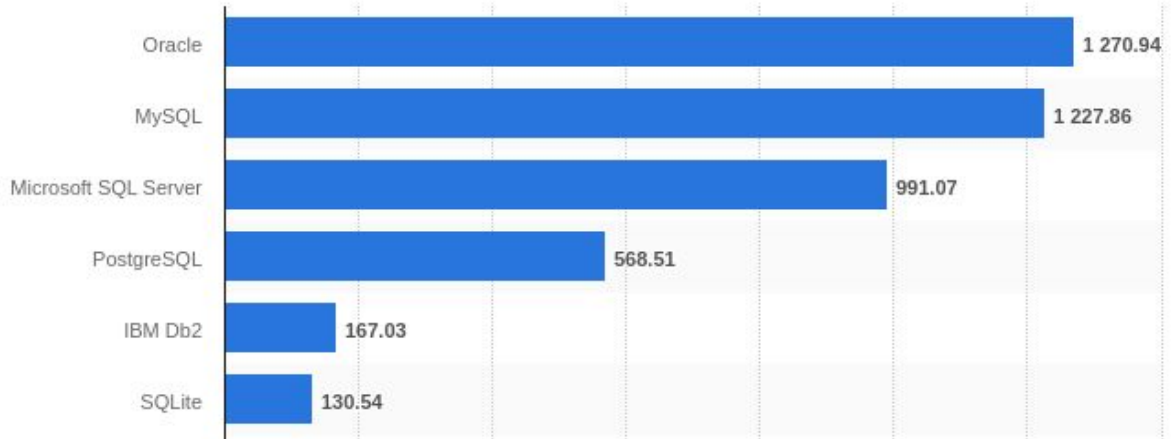
RDBMS

RDBMS stands for Relational Database Management System. RDBMS is the basis for **SQL**.

The data in RDBMS is stored in database objects called **tables**. A table is a collection of related data entries and it consists of **columns** and **rows**.

Top RDBMSs:

- ➔ **PostgreSQL**
- ➔ **MySQL**
- ➔ **OracleDB**
- ➔ **MS SQL Server**
- ➔ **SQLite**
- ➔ **...**



SQL examples

```
SELECT Fruit_Name FROM Fruits
```

Results | Messages

	Fruit_Name
1	Banana
2	Apple
3	Lemon
4	Strawberry
5	Watermelon
6	Lime

```
SELECT * FROM Fruits
```

Results | Messages

	ID	Fruit_Name	Fruit_Color
1	1	Banana	Yellow
2	2	Apple	Red
3	3	Lemon	Yellow
4	4	Strawberry	Red
5	5	Watermelon	Green

```
SELECT * FROM Fruits WHERE Fruit_Color='Red'
```

Results | Messages

	ID	Fruit_Name	Fruit_Color
1	2	Apple	Red
2	4	Strawberry	Red

'Fruits' table:

ID	Fruit_Name	Fruit_Color
1	Banana	Yellow
2	Apple	Red
3	Lemon	Yellow
4	Strawberry	Red
5	Watermelon	Green
6	Lime	Green

PostgreSQL

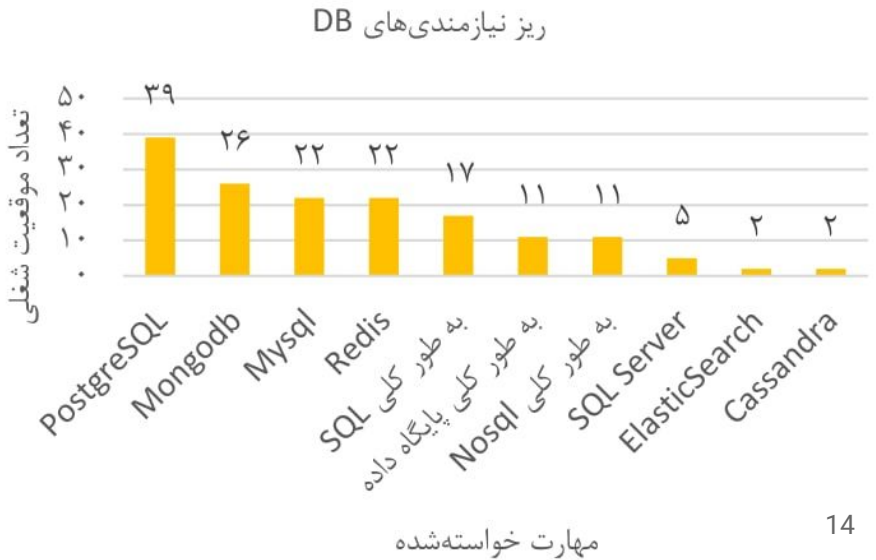
Intro

PostgreSQL: The **World's Most Advanced** Open Source Relational Database
PostgreSQL is a powerful, open source object-relational database system with over 30 years of active development that has earned it a strong reputation for reliability, feature robustness, and performance.

Website: https://www.postgresql.org/

Why PostgreSQL?

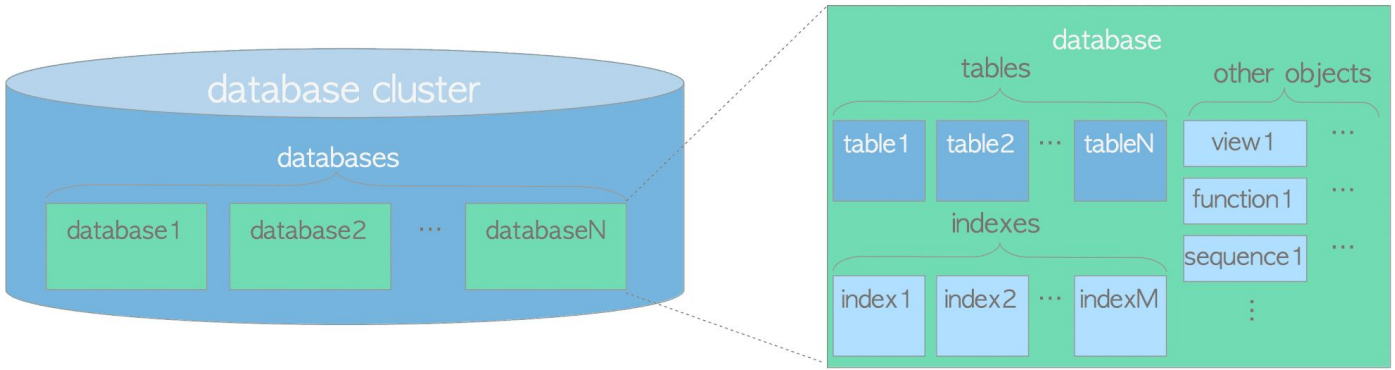
Postgres allows you to store large and sophisticated data safely. It helps developers to build the most complex applications, run administrative tasks and create integral environments.



Postgres Structure

Postgres has several basic components to store and manage data:

1. Database cluster
2. **Database**
3. Schema
4. **Table**
5. View
6. procedure
7. functions
8. …



Postgres Structure

Database:

A database is a named collection of SQL objects ("database objects"). Generally, every database object (tables, functions, etc.) belongs to one and only one database. More accurately, a database is a **collection** of schemas and the schemas contain the tables, functions, etc. So the full hierarchy is: server, database, schema, table (or some other kind of object, such as a function).

Table:

A table in a relational database is much **like a table on paper:** It consists of **rows and columns.** The number and order of the columns is fixed, and each column has a name. The number of rows is variable — it reflects how much data is stored at a given moment. SQL does not make any guarantees about the order of the rows in a table.

Download & Install

Official download page: https://www.postgresql.org/download/

Linux:
Follow the instructions below:

1. Installing Postgres: https://www.postgresguide.com/setup/install/
2. Connecting to Postgres: https://www.postgresqltutorial.com/install-postgresql-linux/

Windows:
Download from **HERE**
Follow the instructions then: **https://www.postgresqltutorial.com/install-postgresql/**

PostgreSQL commands

PostgreSQL commands

Connect to PostgreSQL database

Connecting to postgres database simply:

```
psql -U  user
```

Selecting database (-d) with password (-W):

```
psql -d database -U  user -W
```

Selecting database (-d) with password (-W), on a special host:

```
psql -h host -d database -U  user -W
```

Example:

```
psql -h localhost -d postgres -U postgres -W
```

Switch connection to a new database

Once you are connected to a database, you can switch the connection to a new database under a user specified by user. The previous connection will be closed.

If you omit the user parameter, the current user is assumed.

```
\c dbname
```

```
\c dbname username
```

Example:

```
postgres=# \c laboratory
You are now connected to database "laboratory" as user "postgres".
```

List available databases

To **list** all databases in the current PostgreSQL database server, you use \l command:

```
\l
```

Example:

```
postgres=# \l
                                  List of databases
     Name      |  Owner   | Encoding |   Collate   |    Ctype    | Access privileges
---------------+----------+----------+-------------+-------------+-------------------
 digitalsign   | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
 laboratory    | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
 postgres      | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
 ...
```

PostgreSQL commands

List available tables

To **list** all **tables** in the c**urrent database**, you use \dt command:

```
\dt
```

Example:

```
laboratory=# \dt
          List of relations
 Schema |   Name    | Type  |  Owner
--------+-----------+-------+----------
 public | doctor    | table | postgres
 public | patient   | table | postgres
 public | result    | table | postgres
 public | specimen  | table | postgres
(4 rows)
```

PostgreSQL commands

Describe a table

To **describe** a table such as a column, type, modifiers of columns, etc., you use the following command:

```
\d table_name
```

Example:

```
laboratory=# \d doctor
                        Table "public.doctor"
   Column   |          Type          | Collation | Nullable | Default
------------+------------------------+-----------+----------+---------
 id         | integer                |           | not null |
 first_name | character varying(40)  |           |          |
 last_name  | character varying(40)  |           |          |
 degree     | character varying(40)  |           |          |
 birthday   | date                   |           |          |
 ...
```

List users and their roles

To list all **users** and their **assign roles**, you use \du command:

```
\du
```

Example:

```
laboratory=# \du
                                   List of roles
 Role name |                         Attributes                                | Member of
-----------+---------------------------------------------------------------------+-----------
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
```

PostgreSQL commands

Quit psql

To quit psql, you use \q command and press enter to exit psql.

```
\q
```

```
exit
```

```
quit
```

PostgreSQL commands

More commands

Find more practical commands HERE.

or

to know all available psql commands:

```
\?
```

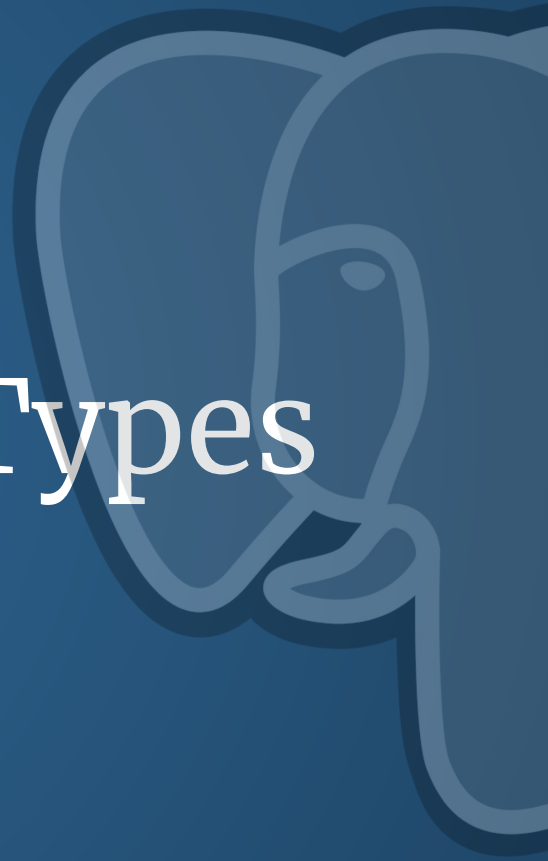
or

to get help on specific PostgreSQL statement:

```
\h ALTER TABLE
```

Command	Description
`\g`	Execute the last command again
`\s`	Display command history
`\s filename`	Save the command history to a file
`\i filename`	Execute psql commands from a file
`\e`	Edit command in your own editor
`\a`	Switch from aligned to non-aligned column output
`\H`	Switch the output to HTML format

PostgreSQL Data Types



Overview of PostgreSQL data types

PostgreSQL supports the following data types:

- **Boolean**
- **Character types such as char, varchar, and text.**
- **Numeric types such as integer and floating-point number.**
- **Temporal types such as date, time, timestamp, and interval**
- **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.**

See the full document here: https://www.postgresqltutorial.com/postgresql-data-types/

PostgreSQL Data Types

boolean (bool)

A Boolean data type can hold one of three possible values: true, false or null. You use **boolean** or **bool** keyword to declare a column with the Boolean data type.

When you insert data into a Boolean column, PostgreSQL converts it to a Boolean value

- 1, yes, y, t, true values are converted to true
- 0, no, false, f values are converted to false.

When you select data from a Boolean column, PostgreSQL converts the values back e.g., t to true, f to false and space to null.

PostgreSQL Data Types

Character

PostgreSQL provides three character data types: **CHAR(n), VARCHAR(n),** and **TEXT**

- **CHAR(n)** is the fixed-length character with space padded. If you insert a string that is shorter than the length of the column, PostgreSQL pads spaces. If you insert a string that is longer than the length of the column, PostgreSQL will issue an error.
- **VARCHAR(n)** is the variable-length character string. With VARCHAR(n), you can store up to n characters. PostgreSQL does not pad spaces when the stored string is shorter than the length of the column.
- **TEXT** is the variable-length character string. Theoretically, text data is a character string with unlimited length.

PostgreSQL Data Types

Numeric

Integer:

There are three kinds of integers in PostgreSQL:

- Small integer (**SMALLINT**) is 2-byte signed integer that has a range from -32,768 to 32,767.
- Integer (**INT**) is a 4-byte integer that has a range from -2,147,483,648 to 2,147,483,647.
- **Serial** is the same as integer except that PostgreSQL will **automatically generate and populate values into the SERIAL column.** (This is similar to AUTO_INCREMENT column in MySQL or AUTOINCREMENT column in SQLite.)

Float:

There three main types of floating-point numbers:

- **float(n)** is a floating-point number whose precision, at least, n, up to a maximum of 8 bytes.
- **real** or **float8** is a 4-byte floating-point number.
- **numeric** or **numeric(p,s)** is a real number with p digits with s number after the decimal point. The numeric(p,s) is the exact number.

Temporal data types

The temporal data types allow you to store date and /or time data. PostgreSQL has five main temporal data types:

- **DATE** stores the dates only.
- **TIME** stores the time of day values.
- **TIMESTAMP** stores both date and time values.
- **TIMESTAMPTZ** is a timezone-aware timestamp data type. It is the abbreviation for timestamp with the time zone.
- **INTERVAL** stores periods of time.

Find more data types here:

https://www.postgresqltutorial.com/postgresql-data-types/

Database statements

PostgreSQL commands

CREATE DATABASE

The CREATE DATABASE statement is used to create a new SQL database.

```
CREATE DATABASE databasename;
```

Example:

```
postgres=# CREATE DATABASE testDB;
CREATE DATABASE
postgres=# \l
                                  List of databases
     Name      |  Owner   | Encoding |   Collate   |    Ctype    |   Access
privileges
--------------+----------+----------+-------------+-------------+----------------
 testdb        | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
  ...
```

DROP DATABASE

The **DROP DATABASE** statement is used to drop an existing SQL database.

```
DROP DATABASE databasename;
```

Example:

```
postgres=# DROP DATABASE testDB;
DROP DATABASE
postgres=# \l
                                  List of databases
     Name      |  Owner   | Encoding |   Collate   |    Ctype    |Access privileges
---------------+----------+----------+-------------+-------------+-----------------
```

PostgreSQL commands

CREATE TABLE

The **CREATE TABLE** statement is used to create a new table in a database.

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
   ....
);
```

```
testdb=# CREATE TABLE Persons (
    PersonID int,
    LastName varchar(255),
    FirstName varchar(255),
    Address varchar(255),
    City varchar(255)
);
```

```
testdb=# \d Persons;
                    Table "public.persons"
  Column   |          Type          | Collation | Nullable | Default
-----------+------------------------+-----------+----------+---------
 personid  | integer                |           |          |
 lastname  | character varying(255) |           |          |
 firstname | character varying(255) |           |          |
 address   | character varying(255) |           |          |
 city      | character varying(255) |           |          |
```

DROP TABLE, TRUNCATE TABLE

The **DROP TABLE** statement is used to drop an existing table in a database.

Note: Be careful before dropping a table. Deleting a table will result in loss of complete information stored in the table!

```
DROP TABLE table_name;
```

The **TRUNCATE TABLE** statement is used to delete the data inside a table, but not the table itself.

```
TRUNCATE TABLE table_name;
```

PostgreSQL commands

ALTER TABLE

To change the structure of an existing table, you use PostgreSQL ALTER TABLE statement.

The following illustrates the basic syntax of the ALTER TABLE statement:

```
ALTER TABLE table_name action;
```

To add a new column to a table, you use **ALTER TABLE ADD COLUMN** statement:

```
ALTER TABLE table_name
ADD COLUMN column_name datatype column_constraint;
```

To drop a column from a table, you use ALTER TABLE DROP COLUMN statement:

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

Find full tutorial Here!

Constraints

SQL constraints are used to specify rules for data in a table.

Constraints are used to l**imit the type of data that can go into a table.** This ensures the accuracy and reliability of the data in the table. If there is any violation between the constraint and the data action, the action is aborted.

Constraints can be **column level** or **table level.** Column level constraints apply to a column, and table level constraints apply to the whole table.

Constraints can be specified when the table is created with the CREATE TABLE statement, or after the table is created with the ALTER TABLE statement.

```
CREATE TABLE table_name (
      column1 datatype constraint,
      column2 datatype constraint,
      column3 datatype constraint,
      ....
);
```

Constraints

The following constraints are commonly used in SQL:

- NOT NULL - Ensures that a column cannot have a NULL value
- UNIQUE - Ensures that all values in a column are different
- PRIMARY KEY - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
- FOREIGN KEY - Prevents actions that would destroy links between tables
- CHECK - Ensures that the values in a column satisfies a specific condition
- DEFAULT - Sets a default value for a column if no value is specified
- CREATE INDEX - Used to create and retrieve data from the database very quickly

Select statements

SELECT Statement

The SELECT statement is one of the **most complex** statements in PostgreSQL. It has many clauses that you can use to form a flexible query.

Let's start with the basic form of the SELECT statement that retrieves data from a single table.
The following illustrates the syntax of the SELECT statement:

```
SELECT
   select_list
FROM
   table_name;
```

```
laboratory=# SELECT * FROM doctor;
first_name | last_name | degree |  birthday  |    phone    | id
-----------+-----------+--------+------------+-------------+--
--
mamad      | akbari    | Phd    | 1990-04-04 | 09123456787 |  4
asqar      | rezaii    | Phd    | 1990-04-04 | 09123456788 |  5
akbar      | babaii    | Phd    | 1998-04-04 | 09123456789 |  6
(3 rows)
```

See full tutorial Here!

SELECT Column Alias

A column alias allows you to assign a column or an expression in the select list of a SELECT statement a temporary name. The column alias exists temporarily during the execution of the query.

The **AS** keyword is ***optional*** in Postgres

```
SELECT column name AS alias_name
FROM table_name;
```

```
SELECT column name alias_name
FROM table_name;
```

See full tutorial Here!

```
laboratory=# SELECT id as ID, last_name as "Last name", degree deg FROM doctor;
id | Last name | deg
----+-----------+-----
 4 | akbari    | Phd
 5 | rezaii    | Phd
 6 | babaii    | Phd
(3 rows)
```

ORDER BY clause

When you query data from a table, the SELECT statement returns rows in an unspecified order. To sort the rows of the result set, you use the ORDER BY clause in the SELECT statement.

The ORDER BY clause allows you to sort rows returned by a SELECT clause in ascending or descending order based on a sort expression.

```
SELECT select_list FROM table_name
ORDER BY
     sort expression1 [ASC | DESC],
        ...
     sort_expressionN [ASC | DESC];
```

See full tutorial Here!

```
laboratory=# SELECT * FROM doctor ORDER BY degree, phone ASC;
first_name | last_name | degree |  birthday  |    phone    | id
-----------+-----------+--------+------------+-------------+----
asqar      | rezaii    | Master | 1990-04-04 | 09123456788 |  5
mamad      | akbari    | Phd    | 1990-04-04 | 09123456787 |  4
akbar      | babaii    | Phd    | 1998-04-04 | 09123456789 |  6
(3 rows)
```

WHERE clause

The SELECT statement returns all rows from one or more columns in a table. To select rows that satisfy a specified condition, you use a **WHERE clause.**

```
SELECT select list
FROM table_name
WHERE condition
ORDER BY sort_expression
```

See full tutorial Here!

```
laboratory=# SELECT * FROM doctor WHERE id = 5;
first_name | last_name | degree |  birthday  |    phone    | id
-----------+-----------+--------+------------+-------------+----
asqar      | rezaii    | Master | 1990-04-04 | 09123456788 |  5
```

```
laboratory=# SELECT * FROM doctor  WHERE degree = 'Phd' AND birthday < '1992-04-04';
first_name | last_name | degree |  birthday  |    phone    | id
-----------+-----------+--------+------------+-------------+----
mamad      | akbari    | Phd    | 1990-04-04 | 09123456787 |  4
```

WHERE operators

Operator	Description
=	Equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
<> or !=	Not equal
AND	Logical operator AND
OR	Logical operator OR

Operator	Description
IN	Return true if a value matches any value in a list
BETWEEN	Return true if a value is between a range of values
LIKE	Return true if a value matches a pattern
IS NULL	Return true if a value is NULL
NOT	Negate the result of other operators

Data statements

LIKE Operator

See full tutorial Here

The `LIKE` operator is used in a `WHERE` clause to search for a specified pattern in a column.

There are two wildcards often used in conjunction with the `LIKE` operator:

- The percent sign (%) represents zero, one, or multiple characters
- The underscore sign (_) represents one, single character

```
SELECT select_list
FROM table name
WHERE column LIKE pattern
```

LIKE Operator	Description
`LIKE 'a%'`	Finds any values that starts with "a"
`LIKE '%a'`	Finds any values that ends with "a"
`LIKE '%or%'`	Finds any values that have "or" in any position
`LIKE '_r%'`	Finds any values that have "r" in the second position
`LIKE 'a__%'`	Finds any values that starts with "a" and are at least 3 characters
`LIKE 'a%o'`	Finds any values that starts with "a" and ends with

LIMIT clause

See full tutorial Here

PostgreSQL LIMIT is an optional clause of the SELECT statement that constrains the number of rows returned by the query.

```
SELECT select list
FROM table_name
ORDER BY sort_expression
LIMIT row_count;
```

```
SELECT select list
FROM table_name
ORDER BY sort_expression
LIMIT row_count OFFSET rows_to_skip;
```

```
laboratory=# SELECT * FROM doctor LIMIT 1;
first_name | last_name | degree |  birthday  |    phone    | id
-----------+-----------+--------+------------+-------------+----
mamad      | akbari    | Phd    | 1990-04-04 | 09123456787 |  4
(1 row)
```

```
laboratory=# SELECT * FROM doctor LIMIT 1 OFFSET 2;
first_name | last_name | degree |  birthday  |    phone    | id
-----------+-----------+--------+------------+-------------+----
asqar      | rezaii    | Master | 1990-04-04 | 09123456788 |  5
(1 row)
```

Data statements

GROUP BY clause

See full tutorial Here

The GROUP BY clause divides the rows returned from the SELECT statement into groups. For each group, you can apply an aggregate function e.g., SUM() to calculate the sum of items or COUNT() to get the number of items in the groups.

The following statement illustrates the basic syntax of the GROUP BY clause:

```
SELECT
   column_1,
   column_2,
   ...,
   aggregate_function(column_3)
FROM table_name
GROUP BY
   column 1,
   column_2,
   ...;
```

```
laboratory=#
SELECT COUNT(*), degree
     FROM doctor
     GROUP BY degree;

count | degree
-------+--------
    2 | Phd
    1 | Master
(2 rows)
```

Data statements

HAVING clause

See full tutorial Here

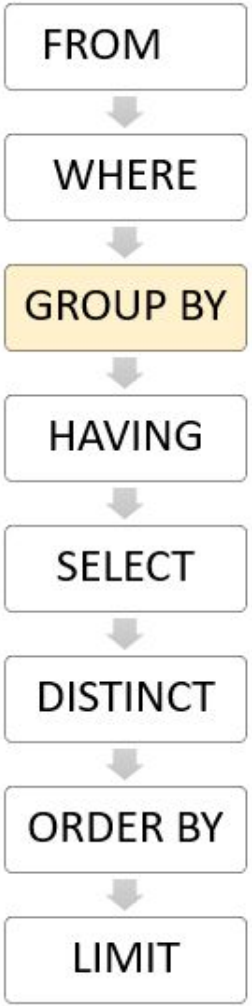
The HAVING clause specifies a search condition for a group or an aggregate. The HAVING clause is often used with the GROUP BY clause to filter groups or aggregates based on a specified condition.

The following statement illustrates the basic syntax of the HAVING clause:

```
SELECT
     column1,
     ...
     aggregate_function (column2)
FROM
     table_name
GROUP BY
     column1
HAVING
     condition;
```

```
laboratory=#
SELECT COUNT(*), degree
     FROM doctor
     GROUP BY degree
     HAVING degree = 'Phd';

 count | degree
-------+--------
     2 | Phd
(1 row)
```



Data statements

INSERT

The PostgreSQL INSERT statement allows you to insert **a new row** into a table.

The following illustrates the most basic syntax of the INSERT statement:

```
INSERT INTO table name(column1, column2, …)
VALUES (value1, value2, …);
```

In this syntax:

- First, specify the name of the table (table_name) that you want to insert data after the INSERT INTO keywords and a list of comma-separated columns (colum1, column2, ...).
- Second, supply a list of comma-separated values in a parentheses (value1, value2, ...) after the VALUES keyword. The columns and values in the column and value lists must be in the same order.

UPDATE

The PostgreSQL **UPDATE** statement allows you to modify data in a table. The following illustrates the syntax of the UPDATE statement:

```
UPDATE table name
SET column1 = value1,
    column2 = value2,
    ...
WHERE condition;
```

In this syntax:

- First, specify the name of the table that you want to update data after the UPDATE keyword.
- Second, specify columns and their new values after SET keyword. The columns that do not appear in the SET clause retain their original values.
- Third, determine which rows to update in the condition of the WHERE clause.

The WHERE clause is optional

If you omit the WHERE clause, the UPDATE statement will update all rows in the table.

DELETE

The PostgreSQL DELETE statement allows you to delete one or more rows from a table.

The following shows basic syntax of the DELETE statement:

```
DELETE FROM table_name
WHERE condition;
```

In this syntax:

- First, specify the name of the table from which you want to delete data after the DELETE FROM keywords.
- Second, use a condition in the WHERE clause to specify which rows from the table to delete.

The WHERE clause is optional.

If you omit the WHERE clause, the DELETE statement will delete all rows in the table.

Example

```
laboratory=# insert intO doctor(first_name, last_name, degree, birthday, phone)
VALUES ('akbar', 'babaii', 'Phd', '1998-04-04', '09123456789');
INSERT 0 1
```

```
laboratory=# INSERT INTO doctor(first_name, last_name, degree, birthday, phone)
VALUES ('mamad', 'akbari', 'Phd', '1990-04-04', '09123456787'),
('asqar', 'rezaii', 'Phd', '1990-04-04', '09123456788');
INSERT 0 2
```

```
UPDATE doctor SET first_name = 'Akbar' where first_name = 'akbar';
UPDATE 1
```

```
laboratory=# UPDATE doctor SET degree = 'PHD';
UPDATE 3
```

```
laboratory=# DELETE FROM doctor;
DELETE 3
```

Exercise 1

1. Create a database **"laboratory"**
2. Create a table **"patient"** based on the table below.
   Note 1: Choose a suitable **Data Type** for each column)
   Note 2: ID is the PK, phone & national_id is **unique,**
3. Insert data on the table below into the **patient** table.
4. Alter the table and add a column **"blood_type"**
5. Update the records on the table and set blood_types:
   [B+, O-, AB-, AB+, A+, B-, B-]
6. Select the second 4 records sorted by age.
7. Select all patients with a **AB** blood_type
8. Select the patients whose names start with 'A' and ages > 40
9. Group the records whose phone_number starts with **'0912'** by their last_name having **'ba'** in their last_name
10. *Update all the patients' phone number, and change 09 to +98

id	first_name	last_name	phone	national_id	age
1	Akbar	Babaii	09123456789	0123456784	23
2	Asqar	Mamadi	09373456788	0123456782	52
3	Reza	Babaii	09373456787	0123456789	44
4	Reza	Akbari	09363456786	0123456787	33
5	Akbar	Akbari	09123456785	0123456785	65
6	Mamad	Babaii	09123456784	0123456783	23
7	Akbar	Salari	09373456783	0123456781	23

Foreign Key

Data statements

Foreign Key

See full tutorial Here

A foreign key is a column or a group of columns in a table that reference the primary key of another table.

The table that contains the foreign key is called the referencing table or child table. And the table referenced by the foreign key is called the referenced table or parent table.

A table can have multiple foreign keys depending on its relationships with other tables.

In PostgreSQL, you define a foreign key using the foreign key constraint. The foreign key constraint helps maintain the referential integrity of data between the child and parent tables.

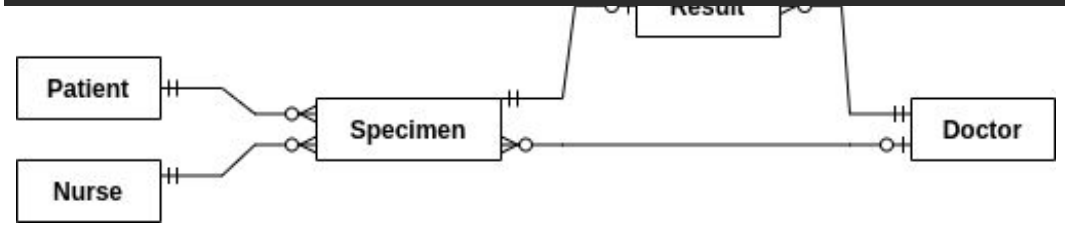
A foreign key constraint indicates that values in a column or a group of columns in the child table equal the values in a column or a group of columns of the parent table.

The following illustrates a foreign key constraint syntax:

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

Foreign Key: Example

```
CREATE TABLE specimen(
      id SERIAL PRIMARY KEY,
      patient_id INT REFERENCES patient ON DELETE CASCADE,
      nurse_id INT,
      create_timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
      data TEXT DEFAULT '',
      CONSTRAINT fk_nurse
            FOREIGN KEY (nurse_id)
                  REFERENCES nurse(id)
                        ON DELETE SET NULL
                        ON UPDATE CASCADE
);
```




```
CREATE TABLE doctor (
    first_name VARCHAR(40),
    last_name VARCHAR(40),
    degree VARCHAR(40),
    birthday DATE,
    phone VARCHAR(20) UNIQUE,
    id INT PRIMARY KEY);
```

```
CREATE TABLE patient (
    id integer PRIMARY KEY,
    first_name VARCHAR(40),
    last_name VARCHAR(40),
    phone VARCHAR(20) UNIQUE,
    national_id VARCHAR(10) UNIQUE,
    age INT);
```

```
CREATE TABLE nurse (
    id integer PRIMARY KEY,
    first_name VARCHAR(40),
    last_name VARCHAR(40),
    phone VARCHAR(20) UNIQUE,
    national_id VARCHAR(10) UNIQUE,
    age INT);
```

On delete/update Strategies

The delete and update actions determine the behaviors when the primary key in the parent table is deleted and updated. Since the primary key is rarely updated, the ON UPDATE action is not often used in practice. We'll focus on the ON DELETE action.

PostgreSQL supports the following actions:

- **NO ACTION**
- **RESTRICT**
- **SET NULL:** The SET NULL automatically sets NULL to the foreign key columns in the referencing rows of the child table when the referenced rows in the parent table are deleted.
- **SET DEFAULT:** The ON DELETE SET DEFAULT sets the default value to the foreign key column of the referencing rows in the child table when the referenced rows from the parent table are deleted.
- **CASCADE:** The ON DELETE CASCADE automatically deletes all the referencing rows in the child table when the referenced rows in the parent table are deleted. In practice, the ON DELETE CASCADE is the most commonly used option.

Complex/Nested selects (SubQuery)

See full tutorial Here

A nested SELECT is a query within a query, i.e. when you have a SELECT statement within the main SELECT. To make the concept clearer, let's go through an example together.

Examples:

```
SELECT
     film id,
     title,
     rental_rate
FROM
     film
WHERE
     rental_rate > (
          SELECT AVG (rental_rate)
          FROM film
     );
```

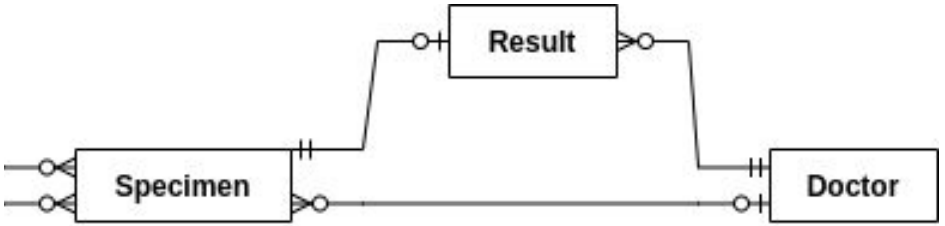
```
SELECT
     first name,
     last_name
FROM
     customer
WHERE
     EXISTS (
          SELECT 1 FROM payment
          WHERE
          payment.customer_id = customer.customer_id
     );
```

Exercise 2: Section A

1. Select patients who have at least 1 specimen
2. Select patients who have been visited by Nurse #1
3. Select nurses who have visited more than one patient.
4. Select the patient's full name, the nurse's Lastname, and the specimen on the same table.
5. Select patients who have been visited by two different nurses.
6. Select specimen whose patient and nurse are the same age.

Exercise 2: Section B

1. Create a table **"Result"** containing fields below:
   - ***id***
   - ***specimen_id***
   - ***doctor_id***
   - ***result (+/-)***
   - ***description***
   - ***create timestamp***



2. Make sure the "Result" relation has suitable **Unique**, **Not null,** and **Foreign Key** constraint(s) based on the following ERD.
3. Show (Select) the patient's national_id, nurse's name, doctor's name, the specimen id, and the result on the same table.
4. Select all results whose doctor has a PhD degree.
5. Select all results whose patient is less than 30 years old.
6. Select all results whose nurse's last name starts with "a".
7. Select all patients whose specimen resulted negative.

Pre-reading

- * Aggregate functions
- * Index
- * Joins
- CHECK constraint
- psycopg2