

MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE
NATIONAL TECHNICAL UNIVERSITY OF UKRAINE
“IGOR SIKORSKY KYIV POLYTECHNIC INSTITUTE”

A.V.Petrashenko, V.I.Pavlovskyi

DATABASES

PRACTICUM

Recommended by the Methodical Council of Igor Sikorsky Kyiv Polytechnic Institute as a study guideline for bachelor's degree holders in 121–"Software Engineering" specialty.

Electronic network educational publication

Kyiv
Igor Sikorsky Kyiv Polytechnic Institute
2022

Reviewer Klyatchenko Y.M., PhD, associate professor,
Igor Sikorsky Kyiv Polytechnic Institute

Responsible Zabolotnia, T.M., PhD, associate professor,
editor Igor Sikorsky Kyiv Polytechnic Institute

*The classification was provided by the Methodical Council of Igor Sikorsky Kyiv Polytechnic
Institute*

*(protocol No. 3 dated January 27, 2022)
on the proposal of the Academic Council of the Faculty of Applied Mathematics*

(protocol No. 5 dated 12/28/2021)

*Andriy Vasylyovych Petrashenko, PhD, associate professor
Volodymyr Ilyich Pavlovsky, PhD, associate professor*

DATABASES PRACTICUM

Databases. Practicum [Electronic resource]: education textbook for students of specialty 121 – Software engineering. / A.V. Petrashenko, V.I. Pavlovsky; Igor Sikorsky Kyiv Polytechnic Institute. – Electronic text data (1 file: 3.3 MB). – Kyiv: Igor Sikorsky Kyiv Polytechnic Institute, 2022. – 124 pp.

The practicum is designed to acquaint students with the methodology of using SQL language and PostgreSQL database server as the support teaching materials for laboratory works of the "Databases" discipline. The tutorial provides information on ERD-modelling, SQL commands syntax and usage as well as on installing PostgreSQL and using it effectively, including creating and modifying database objects, and the transactions management.

The study guide is intended for full-time students of the specialty 121 - "Software Engineering" of Applied Mathematics Faculty of Igor Sikorsky Kyiv Polytechnic Institute.

© A.V.Petrashenko, V.I.Pavlovskiy
©Igor Sikorsky Kyiv Polytechnic Institute, 2022

CONTENTS

1	Introduction.....	7
1.1	Basic definitions.....	7
1.2	Database life cycle	8
2	Database Modeling.....	11
2.1	Entity-Relational Diagrams (ERD)	11
2.1.1	Example of Entity-Relationship Diagram	14
2.1.2	The algorithm of translation ERD to relational database schema.	15
2.2	Logical design of database: relational model	17
2.2.1	Structural aspect of the relational model	17
2.2.2	The integrity aspect of the relational model.....	19
2.2.3	Functional aspect of the relational model.....	23
3	PostgreSQL Data Analysis using SQL Select operator	25
3.1	Syntax of PostgreSQL Select command	25
3.2	Other syntax of PostgreSQL select command.....	26
3.3	Examples of Select command in PostgreSQL	27
4	Database implementation using PostgreSQL.....	31
4.1	Installing the PostgreSQL.....	32
4.2	Verifying the Installation of PostgreSQL.....	37
4.3	Database design using pgAdmin 4.....	38
4.3.1	Create Table Example	39
4.3.2	Query Tool Example	42
4.3.3	pgAdmin 4 ERD Tool.....	43
4.3.4	PostgreSQL – Data Types.....	47
5	Basics of SQL language in PostgreSQL.....	51
5.1	PostgreSQL SELECT examples.....	51
5.1.1	Using PostgreSQL SELECT statement to query data from all columns of a table example.....	52
5.1.2	Using PostgreSQL SELECT statement with expressions example	53
5.2	PostgreSQL ORDER BY clause	53

5.2.1	Using PostgreSQL ORDER BY clause to sort rows by one column	54
5.2.2	Using PostgreSQL ORDER BY clause to sort rows by one column in descending order.....	55
5.2.3	Using PostgreSQL ORDER BY clause to sort rows by multiple columns.....	56
5.3	PostgreSQL WHERE clause.....	57
5.3.1	Using WHERE clause with the equal (=) operator example.....	59
5.3.2	Using WHERE clause with the AND operator example.....	59
5.3.3	Using the WHERE clause with the LIKE operator example.....	60
5.4	PostgreSQL Joins.....	60
5.4.1	PostgreSQL inner join	62
5.4.2	PostgreSQL left join.....	63
5.4.3	PostgreSQL right join	64
5.4.4	PostgreSQL full outer join.....	65
5.5	PostgreSQL GROUP BY clause	65
5.5.1	Using PostgreSQL GROUP BY without an aggregate function example.....	66
5.5.2	Using PostgreSQL GROUP BY with SUM() function example.....	67
5.6	Common Table Expressions.....	68
5.6.1	Recursive Queries.....	69
5.6.2	PostgreSQL recursive queries example (Factorial)	70
5.6.3	A Tree example	71
5.7	SQL: Data Manipulation Commands.....	73
5.7.1	PostgreSQL INSERT statement.....	73
5.7.2	INSERT statement examples.....	74
5.8	UPDATE statement	75
5.8.1	UPDATE examples	76
5.9	DELETE statement	77
6	PostgreSQL Schema.....	78
6.1	Creating a Schema.....	78

6.2	PostgreSQL Schema Objects.....	79
6.3	PostgreSQL Views	80
6.3.1	Views examples	80
6.3.2	View example: access restriction.....	81
6.3.3	Updatable & Temporary Views	82
6.3.4	Materialized Views.....	82
6.3.5	Materialized Views Example	83
6.4	Triggers.....	84
6.5	Indexing.....	87
6.5.1	Types of PostgreSQL Indexes.....	88
6.5.2	PostgreSQL Create Index	90
6.5.3	Disadvantages of using the PostgreSQL Indexes.....	91
7	PL/pgSQL procedural language.....	91
7.1	PL/pgSQL Block Structure	92
7.2	Basic examples of functions.....	94
8	SQL EXPLAIN	95
8.1	EXPLAIN select example	97
8.1.1	EXPLAIN Estimations.....	99
8.1.2	EXPLAIN and Functional Indexes.....	103
8.1.3	EXPLAIN and Partial indexes.....	103
9	Table Scan Modes and Joins.....	104
9.1	Sequential Scan	105
9.2	Index Scan	106
9.3	Index Only Scan.....	106
9.4	Bitmap Scan	107
9.5	Joins implementation	110
9.5.1	Nested Loop Join.....	110
9.5.2	Hash Join	111
9.5.3	Merge Join	112
10	Transactions and Concurrency Control	114

10.1	States of Transactions	114
10.2	ACID: Properties of Transaction	116
10.3	Transaction Isolation Levels (Phenomena)	117
10.3.1	Black and White Example	118
10.4	Four Isolation Levels	121
10.5	Schedule (Serialization)	122
10.5.1	Concurrency Control	122
10.5.2	Concurrency Control techniques	122
REFERENCES		124

1 Introduction

The history of modern databases dates back to the 1960s, which developed almost simultaneously with many other software development technologies, such as high-level programming languages, operating systems, algorithms and data structures, etc. In the 1970s, the scope of database applications expanded significantly, so there was a need to develop an appropriate scientific basis capable of systematizing tasks that rely on databases, on the one hand, and on the other hand, providing developers with the opportunity to build universal software tools that can be used in various subject areas. Among the typical tasks solved with the help of databases, the following can be distinguished: reliable storage and maintenance of the integrity of the data of the subject field, filtering, sorting and grouping of data, continuous and simultaneous access to data by many users and others. At the same time, an extremely important aspect of the scientific basis of database development is the data model - an abstract presentation of formalized data structures describing the subject field, their relationships, as well as operations that are allowed to be performed on these data structures. During its development, several interesting data models have been proposed, including hierarchical, network, and object-oriented, but the most successful, which has been in use for almost half a century, is the relational model developed by Edgar Codd in the late 1960s. It is worth noting that the vast majority of modern database management systems (DBMS), such as Oracle, PostgreSQL, MySQL, MS SQL Server, etc., are based on the ideas proposed in the relational model.

Thus, the purpose of this cycle of laboratory work is to acquire basic knowledge and skills in designing, developing and maintaining the functioning of modern relational databases.

1.1 Basic definitions

Database (DB) is a named set of interconnected data of a certain subject area or a domain. The subject area can be almost any sphere of human activity: business, education, science, public administration, etc. For example, a database of a university, a trading company, a ministry of social policy, a register of real estate, etc. As a rule, solving the problems of effective use of the database relies on the database management system (DBMS). A DBMS is a set of language and software tools designed for joint maintenance (input, modification, filtering, sorting, etc.)

of a database by many users at the same time. Examples of DBMS are PostgreSQL, Oracle, MS SQL Server, redis, MongoDB, etc. The DBMS's ability to process data from various subject areas is ensured by the availability of a universal formal (mathematical) representation of information - a data model. The data model makes it possible to separate the most essential aspects of the subject field by defining its meaningful objects (entities) and the connections between them, as well as to determine restrictions and basic operations on given objects. Examples of the database model are relational (object - relation, operations - union, intersection, Cartesian product, and others), hierarchical (in particular, based on the XML language family), object-oriented (based on the principles of object - oriented programming). as well as define restrictions and basic operations on given objects. Examples of the database model are relational (object - relation, operations - union, intersection, Cartesian product, and others), hierarchical (in particular, based on the XML language family), object-oriented (based on the principles of object -oriented programming). as well as define restrictions and basic operations on given objects. Examples of the database model are relational (object - relation, operations - union, intersection, Cartesian product, and others), hierarchical (in particular, based on the XML language family), object-oriented (based on the principles of object -oriented programming).

1.2 Database life cycle

In software engineering, the life cycle of an information system (system development life cycle) is considered as a sequence of clearly defined steps: planning, analysis, design, implementation, testing and operation. The database, as a rule, is part of the information system and has its own specific interpretation of the specified stages.

1. Database planning. At this stage, they develop a generalized plan for the development and integration of the database into the existing information infrastructure of the organization: they determine the technical requirements and scope of work, methods of data collection, the cost of the project, as well as documentation requirements. For example, a class schedule database should be added to the information infrastructure of the university. Therefore, it is important to combine this database with the database of student performance evaluations, the database of the personnel department and other university systems.

2. Analysis of database requirements. At this stage, the planned volumes of data, what is planned to be stored in the database, the number of users who will have access to the data at the same time, and scaling methods in case of increased server load are determined. For example, for the class schedule database, it is important to analyze the server load at peak times, in particular, at the beginning of the semester, which index structures should be developed, how much memory, processor time, etc. are needed.

3. Database design. This stage aims to determine the data to be stored in the database and the relationships between them, or in other words, the transition from informal description by a person to a formal representation in a computer. Typically, this work is done by a database designer along with a subject matter expert. The first knows how to submit data in a computer, and the second knows the specifics of the subject area. Structuring data in the form of a certain model is the main goal of database design. According to the level of structuring, the following will be distinguished:

unstructured data (scanned documents, presentations, spreadsheets, multimedia data, text messages, geolocation data, etc.) – data that do not have a clearly defined model, data types;

semi structured data (Web pages, XML, CSV and JSON documents, etc.) – have a certain level of data organization, but transferring them to a database with a defined model is complicated;

structured data (data of relational tables in the database) - have a formal structure, in particular, a defined data type, for example, a number, string, time stamp, and others. Having a data type allows you to most efficiently perform operations such as searching, sorting, and grouping data.

To solve the problem of data structuring, the following stages of database design are performed, which ensure a step-by-step transition from the customer's informal ideas to the presentation of the database in a specific DBMS with defined technical and non-technical requirements.

3.1. Conceptual design. At the same time, graphical information and logical models are used, which allow to perform a primary analysis of the subject area in order to highlight the main entities and their relationships. The most widespread model is considered to be "Entity-connection". Its main characteristics are the simplicity of graphical interpretation of the subject area and the possibility of automated conversion to models used in modern DBMSs. For an example of a

class schedule database, you can select "Semester", "Group", "Day of the week", "Subject" as an entity. Connections exist between subject and group, semester and day of the week, etc.

3.2. Logical design makes it possible to move from a generalized, somewhat abstract representation of a database to a data model specific to a particular DBMS, most often a relational one. This design stage is characterized by the specification of the concept of an entity, when its attribute is set to a data type, and various types of relationships between entities (one-to-one, one-to-many, and many-to-many) are implemented in the form of database tables. A special aspect of logical design for a relational model is normalization - an iterative process of overcoming the so-called harmful redundancy of data.

3.3. Physical design is intended for the organization of efficient database storage on physical media, as well as ensuring competitive access of a large number of database users.

The implementation of the database with the help of the existing means of operating systems, application and system software consists in the preparation of information carriers, the creation of the necessary requests to the database, the development of server procedures for data processing, the development of a software and visual interface for user interaction with the database, as well as ensuring control user access to the database in accordance with their functional powers.

4. Database testing consists in checking the compliance of the developed database structure with the real subject area and specifications, controlling the relationships between tables, evaluating the system's performance and its ability to scale if necessary. Testing takes place both directly by potential users of the database and with the help of special software that evaluates various characteristics of the database server.

5. The operation of the database includes monitoring the performance of executing user requests, maintaining and analyzing logs of the operation of various subsystems related to the operation of the DBMS, monitoring the information security of the database, in particular, preventing unauthorized access to it, as well as managing the storage of information on appropriate media and periodic backup of the database.

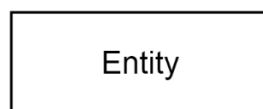
2 Database Modeling

2.1 Entity-Relational Diagrams (ERD)

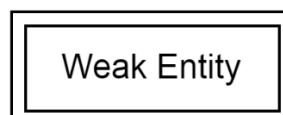
The main goal of the database design stage is the formalization of data and their relationships in the subject area under consideration. The first stage of formalization or the transition from unstructured to structured data is the presentation of information in the form of one of the so-called information-logical models. The most used among such models is "entity-relationship" (ER, entity-relationship). It allows you to define the main entities, connections between entities and their attributes in an intuitively understandable graphic form. It should be noted that an important advantage of this model is the possibility of automatic transformation into a scheme of a real database, in particular, a relational one.

So, the entity-relationship model consists of the following elements.

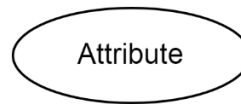
1. An entity that describes a real or imaginary class of objects in the subject field. For example, a student, a book, a computer, an order, etc. An instance of an entity is a specific representative of a class of objects, for example, student "Dmytrenko", book "C++", computer "Model M1", order "N321", etc. Graphically, the entity is indicated by a rectangle:



1.1. A weak entity, instances of which cannot exist in the subject field independently, but require the presence of another entity on which the given depends. For example, the entity "order item" cannot exist without the entity "order", "blog post" cannot exist without its author. It is indicated by a rectangle with a double border.

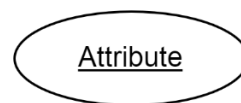


2. An attribute that defines a certain characteristic of an entity. For example, in the Student entity, the attributes are "record book number", "surname", "date of birth", etc. Graphically, the attribute is denoted as an oval:

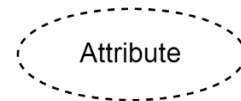


In some specific cases, different types of attributes are used:

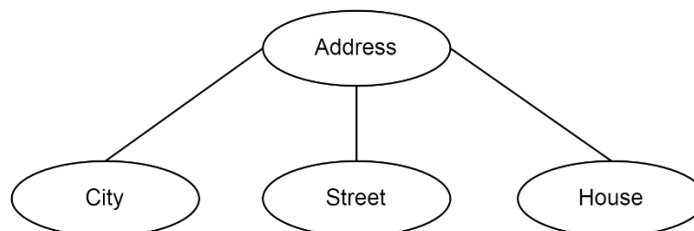
2.1. A key attribute intended to uniquely identify a specific instance of an entity. For example, the "Number of the student's record book" or "VIN number of the car". Note that any entity must have a key attribute or, in the case of a composite key, key attributes. It is graphically represented as a normal attribute with an underlined name:



2.2. A derived attribute that is intended to represent an entity characteristic whose value depends on another attribute, such as the "discount price" of an item, is calculated based on the "base price" and "discount". It should be noted that saving derived attributes in databases is impractical in most cases, as it leads to saving redundant information. It is graphically displayed as follows:



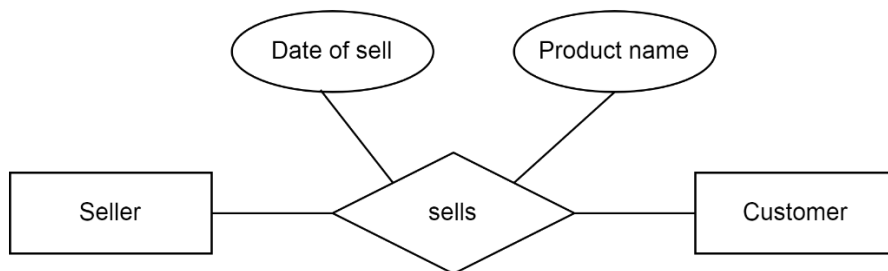
2.3. A composite (composite) attribute defines a complex characteristic of an entity, which also consists of a group of other attributes. An example of a composite attribute is an address consisting of a city, a street, a house number, and an apartment. It is graphically indicated as follows:



2.4. A multivalued attribute displays characteristics that can have multiple values at the same time, for example, a person can have multiple email addresses or phone numbers. This attribute is denoted as follows:



3. Connection between entities determines the degree of their relationship. The most common type of relationship is binary, which connects two entities. A connection is denoted as a rhombus connecting two entities with lines and, like entities, can have its own attributes. For example, the relation "sold" between the entities Seller and Buyer can have the attributes "Sold Date" and "Item Name". The following figure graphically illustrates this:



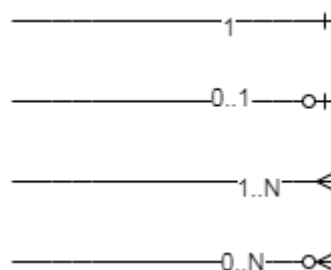
3.1. The cardinality of relationships is an important quantitative characteristic of the ratio of instances of entities. The following types of connections are distinguished:

1:1- to each other

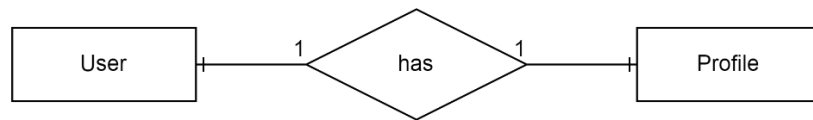
1:N- one to many

N:M- many to many

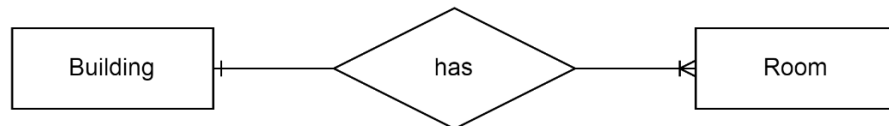
Graphically, the cardinality of connections is indicated by a special sign next to the end of the line:



At the same time, in the case when the connection is not mandatory, the symbol "0" is set. Similar special signs must be placed on both sides of the line as in the example:

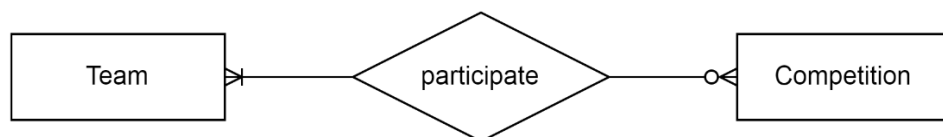


Such a scheme should be read as follows: "each user of the site has one profile and each profile on the site corresponds to exactly one user." The following example illustrates a one-to-many relationship:



At the same time, the educational building necessarily contains one or more classrooms, and one classroom is contained in one and only one building.

An example of using a many-to-many (N:M) relationship is illustrated in the following figure:



In the picture: one sports team participates in several competitions and several teams participate in one competition.

2.1.1 Example of Entity-Relationship Diagram

The idea of an example is to create a conceptual model of data for polling application. The idea is to store user accounts of two types: administrators and regular users. Administrators can create a poll, enter some questions and possible options. On the other hand, regular users can enter answers to these questions.

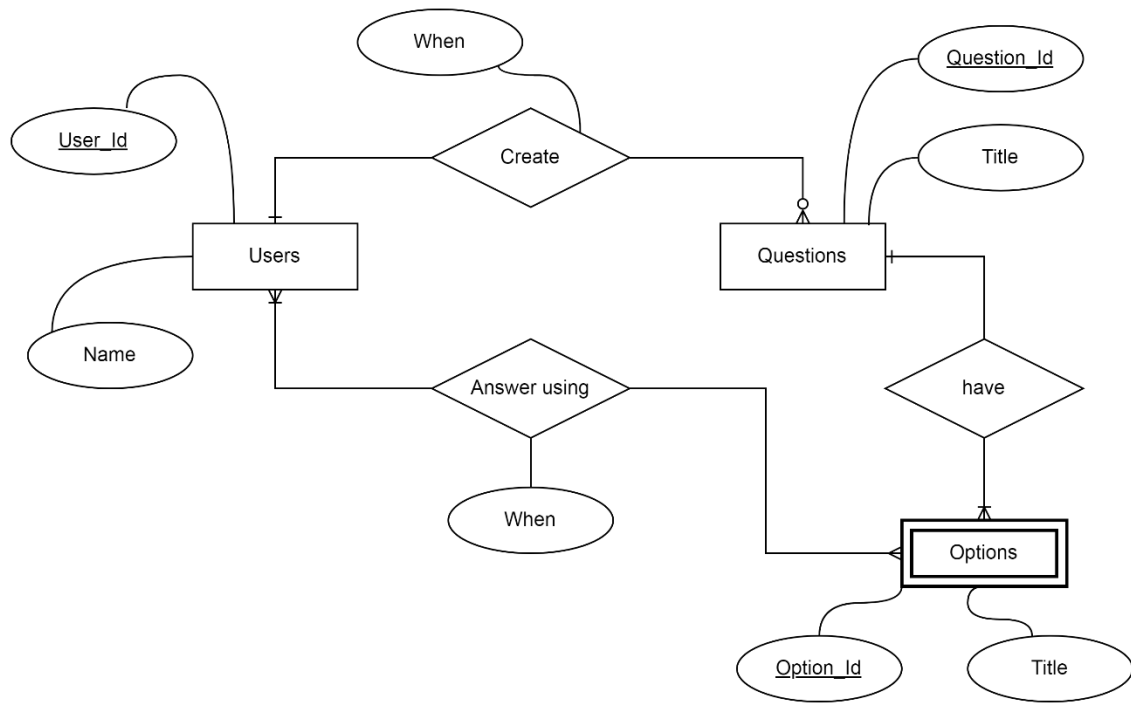


Figure 2.1 Polling ERD

The diagram at the Figure 2.1 has four entities: users, questions, options and answers. There are some relationships between them. For example, users can select options to the questions. There are also some different types of cardinality: 1:N and N:M.

2.1.2 The algorithm of translation ERD to relational database schema

Since ER diagram gives us the good knowledge about the requirement and the mapping of the entities in it, we can easily convert them as tables and columns. i.e.; using ER diagrams one can easily create relational data model, which nothing but the logical view of the database.

Follow the steps given below for the conversion of the ER diagrams to tables in the database management system (DBMS).

Step 1 – Conversion of strong entities

For each strong entity create a separate table with the same name.

Includes all attributes, if there is any composite attribute divided into simple attributes and has to be included. Ignore multivalued attributes at this stage. Select the primary key for the table.

Step 2 – Conversion of weak entity

For each weak entity create a separate table with the same name. Include all attributes. Include the primary key of a strong entity as foreign key is the weak entity. Declare the combination of foreign key and discriminator attribute as P key from the weak entity.

Step 3 – *Conversion of one-to-one relationship*

For each one to one relation, say A and B modify either A side or B side to include the primary key of the other side as a foreign key. If A or B is having total participation, then that should be a modified table. If a relationship consists of attributes, include them also in the modified table.

Step 4 – *Conversion of one-to-many relationship*

For each one to many relationships, modify the M side to include the primary key of one side as a foreign key. If relationships consist of attributes, include them as well.

Step 5 – *Conversion of many-many relationship*

For each many-many relationship, create a separate table including the primary key of M side and N side as foreign keys in the new table. Declare the combination of foreign keys as P for the new table. If relationships consist of attributes, include them also in the new table.

Step 6 – *Conversion of multivalued attributes*

For each multivalued attribute create a separate table and include the primary key of the present table as foreign key. Declare the combination of foreign key and multivalued attribute as primary keys.

Step 7 – *Conversion of n-ary relationship*

For each n-ary relationship create a separate table and include the primary key of all entities as foreign key. Declare the combination of foreign keys as primary key.

Let's consider the ER diagram from the Figure 2.1.

The basic rule for converting the ER diagrams into tables is *to convert all the Entities in the diagram to tables.*

All the entities represented in the rectangular box in the ER diagram become independent tables in the database. In the diagram, USERS, QUESTIONS and OPTIONS forms individual tables.

All single valued attributes of an entity are converted to a column of the table.

All the attributes, whose value at any instance of time is unique, are considered as columns of that table. In the USER Entity, *name* forms the columns of USER table. Similarly, *Title* form the columns of QUESTIONS table. And so on.

Key attribute in the ER diagram becomes the Primary key of the table.

In diagram above, User_id, Question_Id and Option_id are the key attributes of the entities. Hence, we consider them as the primary keys of respective table.

2.2 Logical design of database: relational model

The relational model is built on the basis of set theory and the theory of relational databases specially developed by E. Codd in the 1970s. The model considers three aspects of the data:

structural, which describes the model object, its composition and properties;

integrity, which sets the rules and limitations of data use;

functional, which defines a certain set of operations on the model object.

2.2.1 Structural aspect of the relational model

The main object of the relational model is called a relation. A relation is defined as a subset of the Cartesian product. In turn, the Cartesian product $D_1 \times D_2 \times \dots \times D_n$ for given finite sets D_1, D_2, \dots, D_n (not necessarily different) is called the set of products of the form $d_1 \times d_2 \times \dots \times d_n$, where $n > 0, d_1 \in D_1, d_2 \in D_2, \dots, d_n \in D_n$. Sets D_1, D_2, \dots, D_n are called domains. Let us give an example of a Cartesian product. Let two sets $D_1 = \{1,2,3\}$ and $D_2 = \{a,b\}$ be given, then the Cartesian product of D will be:

$$D = D_1 \times D_2 = (1 \times a, 1 \times b, 2 \times a, 2 \times b, 3 \times a, 3 \times b).$$

Therefore, the relation defined on the sets D_1, D_2, \dots, D_n is a subset $D_1 \times D_2 \times \dots \times D_n$. For the given example, the ratios are $(1 \times a, 3 \times a), (2 \times a), (1 \times a, 1 \times b, 2 \times a, 2 \times b), (1 \times a, 1 \times b, 2 \times a, 2 \times b, 3 \times a, 3 \times b), \emptyset$ and others. It should be noted that the empty set \emptyset is also a correct relation.

Tuple is an element of the Cartesian product. The number n (the number of domains) determines the degree of the relation, and the number of tuples is called the cardinal number or power of the relation. In the example, the tuples are $1 \times a$, $1 \times b$, $2 \times a$, etc., the degree of the relation is 2, and the power is $6 = 2 \times 3$, where 2 is the number of elements in $D1$ and 3 is the number of elements in $D2$.

Domains play a special role in this regard. They define a certain set of values of the same type. In practice, a domain is defined as: a list of valid values, a range of values, or some function. Domains have a similar purpose to data types in programming languages: they define valid operations on data and allow them to be compared.

Visually, it is convenient to present the relationship in tabular form, where the columns correspond to the value of the tuple element from the corresponding domain and are called attributes, and the rows are tuples. For the example above, the tabular representation of the Cartesian product will have the following form:

1	a
1	b
2	a
2	b
3	a
3	b

In practice, the tabular representation of the relation contains a special first row - the relation header, where each of its values determines the name of the corresponding attribute, and the data of this column belongs to a certain domain. For example, in the relation "Movie", the attributes are the name of the film (domain - string), the director's last name (domain - string), the year of release (domain - integer), and the duration of the tape (domain - integer):

The name of the movie	Director's last name	Year of release	Duration (min)
Star Wars. Episode IV: New Hope	George Lucas	1977	121
Joker	Todd Phillips	2019	123
Aladdin	Guy Ritchie	2019	128

Based on the above, the following properties of the relation follow.

1. The absence of identical tuples, which is a consequence of defining a relation as a set of tuples, and elements in a set cannot be repeated.
2. Lack of ordering of tuples, which also follows from the multiple nature of the relation: the elements of sets in mathematics are not ordered.
3. Atomicity of attribute values, which is a consequence of defining the domain as a set of atomic values of a certain set. This means that the relation at the intersection of a column and a row is allowed to have only one value, and not a list of them. For example, according to the rules of the subject field, in relation "Book" each instance can have several authors, but it is forbidden to write them in one cell in the relational model:

Book	The authors
The C Programming language	B. Kernighan, D. Ritchie

From the point of view of the relational model, the following relation will be correct:

Book	The authors
The C Programming language	B. Kernigan
The C Programming language	D. Ritchie

In this version, the "Authors" attribute contains only one last name in each cell.

4. The absence of ordering of attributes means that the position of data elements (values) in a tuple is arbitrary and does not affect the content of the tuple, provided that the order of attributes in all tuples is the same. In practice, this means that the columns of the relation can be rearranged in any way.

2.2.2 The integrity aspect of the relational model

In the context of the relational model, integrity means the correctness of the data after the operation of modifying, extracting and inserting data into the relation. From a practical point of view, integrity safeguards provide protection against

violations of certain domain rules and restrictions. Integrity is ensured by the following types of constraints.

Structural: a relational DBMS is capable of processing only relational relations that have the properties discussed above, in particular, regarding the uniqueness of tuples. The uniqueness of tuples is ensured by the presence of a primary key: a specially selected minimum possible set of relation attributes that uniquely identify each tuple and also do not contain empty (NULL) values. For example, for the relationship "Book" with the attributes "title", "author's last name", "genre", "number of pages", the primary key can be "title" in the event that the database knows in advance that there will be no identical book titles. Otherwise, you should choose "title" and "author's last name". If two books of the same author with the same title are allowed (for example, with the release of a new edition), then the attribute "number of pages" should be added to the primary key.

When there are too many attributes in the primary key (usually more than two or three), then a separate attribute is artificially added to the relationship - the so-called surrogate key. It usually has a numeric data type (domain), its values are automatically generated by the DBMS as unique and, as a result, each tuple regardless of the values of the remaining attributes will be unique. However, surrogate keys should be used with caution and only in the case of the impracticality of choosing "natural" attributes, as this leads to the actual loss of control over the data. For example, for the "Book" relation in question, it will be possible to enter more than one book with completely identical data (except for the value of the surrogate key), which will lead to negative consequences of working with the data. It should be remembered that each relationship must have a primary key. then they artificially add a separate attribute to the relationship - the so-called **surrogate key**. It usually has a numeric data type (domain), its values are automatically generated by the DBMS as unique and, as a result, each tuple regardless of the values of the remaining attributes will be unique. However, surrogate keys should be used with caution and only in the case of the impracticality of choosing "natural" attributes, as this leads to the actual loss of control over the data. For example, for the "Book" relation in question, it will be possible to enter more than one book with completely identical data (except for the value of the surrogate key), which will lead to negative consequences of working with the data.

A special value in relational databases is NULL. This value is not the same as the Boolean "False", the numeric "0", or the empty string. According to the definition

of the primary key, the attributes included in it cannot take the value NULL. The rest of the attributes can receive this value by default (although there is an option to disallow this possibility by specifying a NOT NULL constraint). In this regard, logical operations NOT, AND and OR are generally performed in three-digit logic, where: 1=True, 0.5=NULL, 0=False. At the same time, the calculation of these functions is performed according to general rules: AND as MIN(a,b), OR as MAX(a,b) and NOT as 1-a, where a, b are function arguments. For example, 1 AND NULL = NULL, 0 OR NULL = NULL, 1 OR NULL = 1, NOT(NULL) = NULL. However, NULL ≠ NULL, and the expression NULL=NULL returns NULL instead of True.

Another type of constraint is referential integrity, which binds two relationships in a master-child pattern or as a one-to-many relationship, where "one" corresponds to the master relationship and "many" to the child relationship. At the same time, for each value of the foreign key defined in the subordinate relation, there must exist a tuple with the same value of the primary key in the main relation. In other words, if the child relation is related to the main one by means of a certain attribute(s), then the value of this attribute(s) must belong to the set of values of the primary key of the main relation. Let there be two relations: Group (group code), Student (group code, student code, last name), where a is the primary key of the corresponding relation, and "group code" is an attribute of the foreign key. The following figure graphically illustrates this scheme:

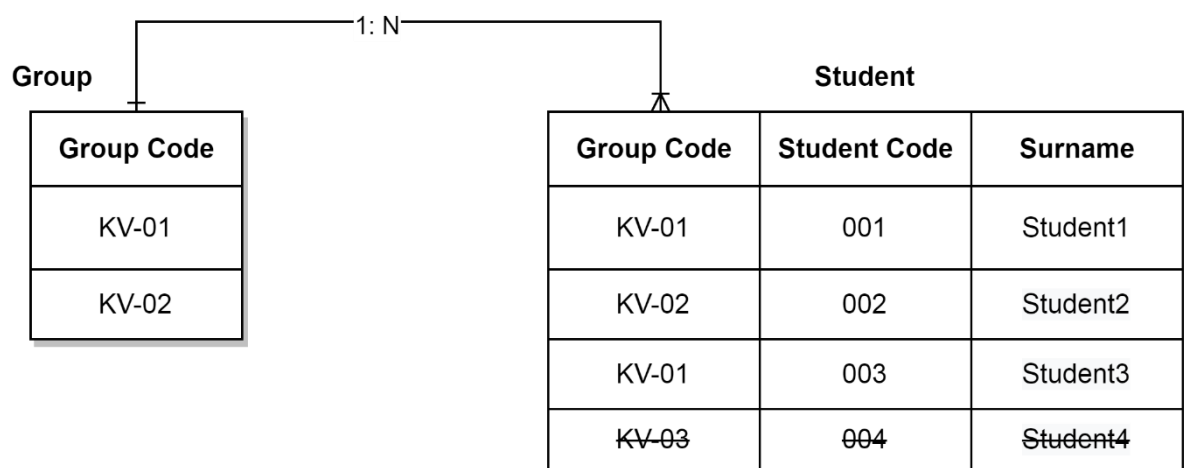


Figure 2.2 Group, Student

Thus, the value of the "Group Code" attribute in the "Student" relationship can only be KV-01 or KV-02, since no other values are entered in the "Group"

relationship. That is why the value of KV-03 is crossed out, illustrating that entering this value is prohibited.

Linking relations using a foreign key imposes certain restrictions on operations to modify the values of attributes of the primary key of the main relation, as well as on extracting tuples of the main relation that have dependent data in the child. For example, what will happen to tuples of the relation "Student" with student codes 001 and 003, if the related tuple KV-01 is removed from the relation "Group"? Or how will changing the name of the KV-02 group in the "Group" relation affect the tuple of the "Student" relation with the student code 002? To answer such questions in the relational model, the following options are provided:

1. In the presence of dependent data in the subordinate relationship, prohibit the execution of extraction/editing operations in the main relationship.
- 2a. When performing an extraction operation in the main relation if there is dependent data in the subordinate relation, extract the cascaded dependent data in the subordinate relation. For example, removing the tuple "KV-01" from the "Group" relation will automatically remove the tuples with the student code 001 and 003.
- 2b. When performing an operation to edit data in the primary key of the main relation, the data in the corresponding attribute(s) of the subordinate relation will automatically change. For example, changing the group code KV-02 in the "Group" relation to KV02, the same changes will occur in the "Student" relation for the tuple with the student code 002.
3. When performing extraction/editing operations in the main relation (if there is dependent data in the subordinate relation), set the foreign key attribute to NULL in the subordinate relation. For example, removing a tuple with the group code KV-01 from the "Group" relation will automatically set the value NULL to the attribute group code of the "Student" relation of the tuples with the student code 001 and 003.

The next aspect of integrity is language. At the same time, it is claimed that access to the data of the relational database occurs only by means of the special SQL language (structural query language) and cannot be done in any other way. In this way, the correctness of relational database processing is guaranteed. The SQL language is covered in the following sections.

The final aspect of integrity in the relational model is the semantic or domain aspect, which allows for domain-specific constraints on relational attribute data. First of all, it is the domain of an attribute or a certain data type. For example, in the relation "Student", the attribute student_code refers to the numeric data type. Other limitations are:

NOT NULL– the prohibition to leave the attribute value empty (performed unconditionally for primary key attributes).

UNIQUE– attribute value must be unique, i.e. no attribute value can be repeated.

DEFAULT– if the value is not inserted, then automatically substitute the default value.

CHECK– set an additional limit on the value, for example, in the form of a comparison on a range, specific values, etc.

2.2.3 Functional aspect of the relational model

A necessary condition for the existence of a relational model is the functional aspect, which considers the performance of certain operations on data. At the same time, operations are built in such a way that their arguments and results are relations, that is, the model has the property of closure, which in turn provides the opportunity to build chains of calculations and expand the list of possible operations. Based on this, basic and derivative operations are considered.

Table. Basic operations of relational algebra

Name and operation designation	Appointment	Examples
Sample or Select (σ)	Filtering tuples in a relation based on a given condition (predicate): $\sigma_{\text{predicate}}(R)$, where R is a relation.	$\sigma_{\text{year} > 2016}(\text{Person})$ $\sigma_{\text{id} = 20 \text{ or } \text{pages} > 100}(\text{Book})$ $\sigma_{\text{mark} = 'A' \text{ and } \text{stud_id} = 1}(\text{Mark})$
Projection or Projection (Π)	Selection of relationship attributes and removal of duplicate tuples: $\Pi_{\text{attr1}, \dots, \text{attr}}(R)$, where $\text{attr1}.. \text{attrn}$ are the attributes to be selected, R is the relation.	$\Pi_{\text{email}, \text{username}}(\text{User})$ $\Pi_{\text{name}}(\text{Book})$ $\Pi_{\text{title}, \text{comment}}(\text{Post})$

Association or Union (\cup)	Union of tuples of two relations, compatible by the number and types of the corresponding attributes (domains): $R \cup S$, where R,S where R and S are relations.	User \cup Person $\pi_{city}(City) \cup \pi_{city}(Capital)$
Difference or Minus (-)	Difference of tuples of relations R and S: such tuples from R that are not in S. $R - S$	User – Student User - $\sigma_{online}(User)$
Cartesian product (\times)	The concatenation of each tuple R with each tuple S: $R \times S$	User \times User UserX Student X Book
Renaming or Rename (ρ)	Renaming a relation or its attribute: $\rho(\text{new name, old name})$	$\rho(\text{School, University})$ $\rho(\text{Attr, Attribute})$

Table. Derivative operations of relational algebra

Name and operation designation	Appointment	Examples
Natural connection or Natural Join (\bowtie)	Join is a derived operation from the Cartesian product, combined with the sampling operation. A natural join uses a sample with the condition of equality of values of attributes shared by name and domain. $R \bowtie S$	Product type \bowtie Goods Group \bowtie Student
Intersection or Intersection (\cap)	Selection of identical tuples of two relations, compatible by the number and types of corresponding attributes (domains): $R \cap S$, where R,S where R and S are relations. Can be expressed through basic operations: $R \cap S = R - (R - S)$	Product in stock \cap The product is in the store Things for Monday \cap Things for Tuesday

Division or Division (\div)	<p>For relations R(A,B) and S(B) returns a from A such that for each b from B there exists a tuple (a, b) in R. If $R \times S = T$, then $T \div R = S$ and $T \div S = R$.</p> <p>Calculation rules:</p> <ol style="list-style-type: none">1) $T1 = \prod A (R)$;2) $T2 = T1 \times S$;3) $T3 = T2 - R$;4) $T = T1 - \prod A (T3)$	<table><tr><th>St</th><th>Kr</th></tr><tr><td>s1</td><td>a1</td></tr><tr><td>s2</td><td>a2</td></tr><tr><td>s1</td><td>a2</td></tr></table> \div <table><tr><th>Kr</th></tr><tr><td>a1</td></tr><tr><td>a2</td></tr></table> $=$ <table><tr><th>St</th></tr><tr><td>s1</td></tr></table>	St	Kr	s1	a1	s2	a2	s1	a2	Kr	a1	a2	St	s1
St	Kr														
s1	a1														
s2	a2														
s1	a2														
Kr															
a1															
a2															
St															
s1															

3 PostgreSQL Data Analysis using SQL Select operator

In PostgreSQL, the SELECT command is the core command used to retrieve data from a database table, and the data is returned in the form of a result table, which is called **result-sets**.

The **select** command contains several clauses that we can use to write a query easily. The basic task while performing the select command is to query data from tables within the database.

The various clauses of SELECT command are as follows:

- Sort rows with the help of **the ORDER BY** clause.
- Group rows into groups using **GROUP BY** clause
- Filter the rows with the help of **the WHERE** clause.
- Filter the groups with the help of **the HAVING** clause.
- Select separate rows with the help of a **DISTINCT** operator.
- Perform set operations with the help of **UNION, INTERSECT, and EXCEPT**.
- Join with other tables with joins such as **LEFT JOIN, INNER JOIN, CROSS JOIN, FULL OUTER JOIN** conditions.

3.1 Syntax of PostgreSQL Select command

The **SELECT command** is used to recover data from a single table.

The syntax of the SELECT command is as follows:

SELECT select_list

FROM table_name;

The following are the parameters used in the above syntax:

Parameters	Description
Select_list	It is used to define a select list which can be a column or a list of columns in a table from which we want to retrieve the data
Table_name	In this, we will define the name of the table from which we want to query data.

Note: If we describe a list of columns, we can use a comma to separate between two columns. If we do not need to select data from all the columns in the table, we can use an asterisk (*) instead of describing all the column names because the select list can have the exact values or the expressions.

The SQL language is case insensitive, which means **select** or **SELECT** has the same result.

3.2 Other syntax of PostgreSQL select command

SELECT column1, column2,

.....

columnN

FROM table_name;

Here we use the below parameter:

Parameters	Description
column1, column2,...columnN	These are used to describe the columns from where we retrieve the data.

If we want to retrieve all the fields from the table, we have to use the following syntax:

SELECT * FROM table_name;

3.3 Examples of Select command in PostgreSQL

Here, we will understand the use of **Select command** in PostgreSQL with the following examples.

We will use the **Employee** table for better understanding.

Employee
*id
Name
Age
Address
Salary

To query data from one column using the SELECT command

In this example, we will find the names of all **Employee's** from the **employee table** with **SELECT command's** help:

Select name

from "Company".employee;

Output

Once we perform the above query, we will get the below result:

Data Output		Explain	Messages	Notifications
	name text			
1	John			
2	Mike			
3	Emily			
4	James			
5	Sophia			

Note:

To separate two SQL statements, we will use the semicolon (;).

In this above query, at the end of the select command, we added a semicolon (;). At this point, the semicolon is not a part of the SQL declaration as it is used to indicate PostgreSQL the end of an SQL command.

To query data from multiple columns using the SELECT command

If we want to see the data of multiple columns of a particular table, we can execute the below query.

For example, let us assume that we need to get the **employee's name, age, and address**. Therefore, we can define these column names in the SELECT command as we see in the below query:

select

name,

age,

address

from "Company".employee;

Output

After executing the above command, we will get the below outcome:

	Data Output	Explain	Messages	Notifications
	name text	age integer	address character (50)	
1	John	24	New York	...
2	Mike	22	Chicago	...
3	Emily	24	Boston	...
4	James	20	Philadelphia	...
5	Sophia	21	New York	...

To query data in all columns of a table using the Select Command

If we want to get all the columns data in a particular table, we can execute the below query.

Here, we select all the columns and rows from an **employee** table under the **Company** schema with the below query's help:

AD

SELECT

*

FROM

"Company".employee;

Output

After executing the above query, we will get the following result:

9

```
select * from "Company".employee;
```

Data Output

Explain

Messages

Notifications

	Id [PK] integer	name text	age integer	address character (50)	salary real
1	101	John	24	New York	25000
2	102	Mike	22	Chicago	30000
3	103	Emily	24	Boston	20000
4	104	James	20	Philadelphia	45000
5	105	Sophia	21	New York	50000

In the above example, we used the (*) asterisk symbol rather than writing all the column's names in the select command. Sometimes we have n-numbers of columns in the table, and writing all the column names became a tedious process.

But sometimes it is not a good process to use the asterisk (*) in the SELECT command.

If we are using the embedded SQL statements in the code because of the following reasons:

We have one table with several columns, and it is not necessary to use the SELECT command with an asterisk (*) shorthand for recovering the data from all columns of the table.

Besides that, if we retrieve the database's unimportant data, it will enhance the load between the database and application layers. And the output of our applications will be less accessible and deliberate. Thus, it is the best approach to describe the column names openly in the SELECT clause because every time, it is likely to get only needed data from a table.

Using the SELECT command with expressions

In the below example, we will return the full name and address of all the employee with the help of select command:

SELECT

name AS full_name,

address

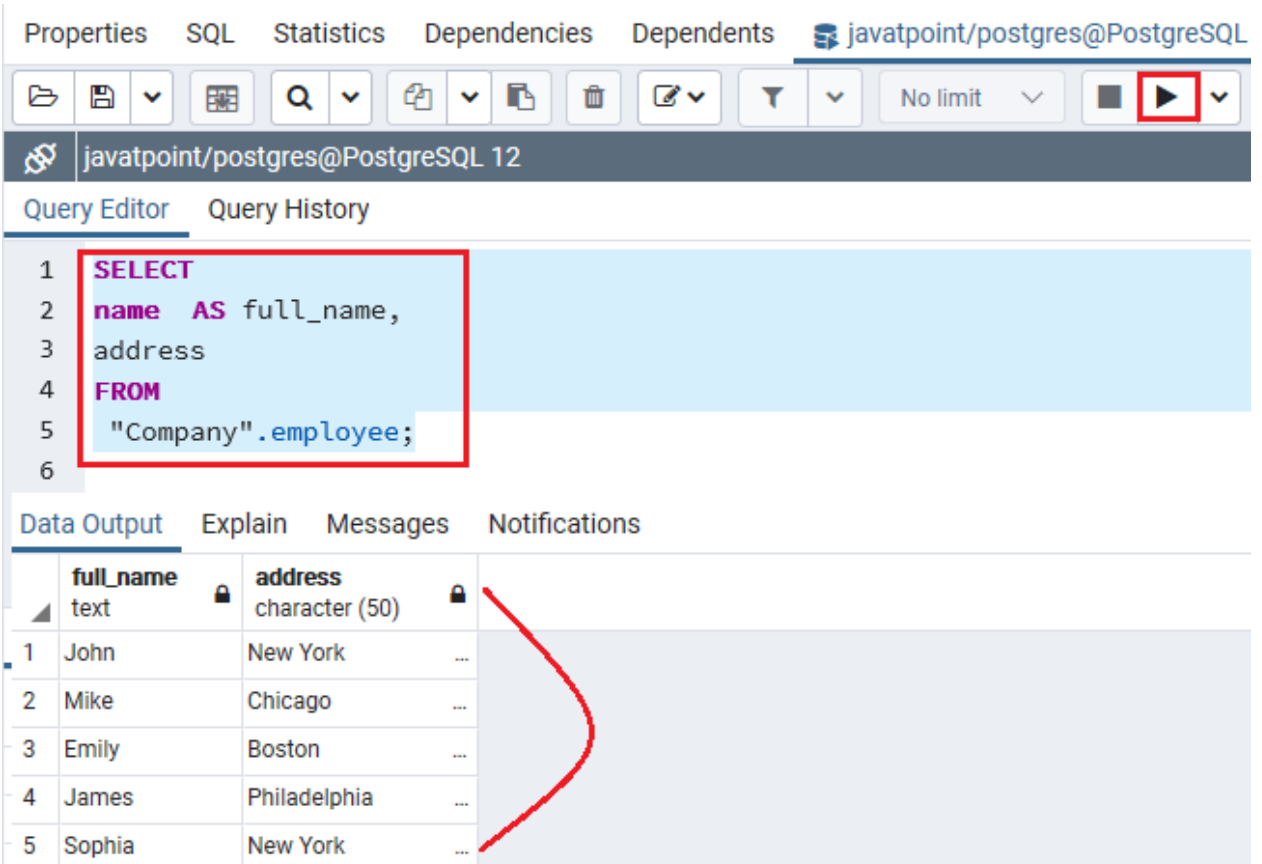
FROM

"Company".employee;

In the above query, we will use a column alias AS full_name to assign a column heading to the name expression.

Output

After executing the above query, we will get the below result:



The screenshot shows a PostgreSQL query editor interface. The query editor displays the following SQL query:

```
1 SELECT
2 name AS full_name,
3 address
4 FROM
5 "Company".employee;
6
```

The query is highlighted with a red box. Below the query editor, the "Data Output" tab is selected, showing the results of the query. The results are displayed in a table with two columns: full_name and address. The table contains five rows of data:

	full_name	address
1	John	New York
2	Mike	Chicago
3	Emily	Boston
4	James	Philadelphia
5	Sophia	New York

A red arrow points from the "Data Output" tab to the table of results.

Using the SELECT command with expressions

Here, we will perform the select command with an expression where we skip the **From clause** into the select command, as the command does not refer to any table.

SELECT 4*2 **AS** result;

Output

We will get the below output once we execute the above command:

The screenshot shows a SQL query execution interface. At the top, the query `SELECT 4*2 AS result;` is entered in a text box. Below the query, there are tabs for **Data Output**, **Explain**, **Messages**, and **Notifications**. The **Data Output** tab is selected, showing a table with one column named **result** of type **integer**. The table contains one row with the value **8**. Below the table, a green message box with a checkmark icon states: **Successfully run. Total query runtime: 329 msec. 1 rows affected.**

	result
1	8

4 Database implementation using PostgreSQL

PostgreSQL is one of the most advanced general-purpose object-relational database management system and is open-source. Being an open-source software, its source code is available under PostgreSQL license, a liberal open source license. Anyone with the right skills is free to use, modify, and distribute PostgreSQL in any form. As it is highly stable, very low effort is required to maintain this DBMS.

The key features that make PostgreSQL a reliable and user-friendly are listed below:

- User-defined types
- Table inheritance
- Sophisticated locking mechanism
- Foreign key referential integrity
- Views, rules, subquery

- Nested transactions (savepoints)
- Multi-version concurrency control (MVCC)
- Asynchronous replication
- Native Microsoft Windows Server version
- Tablespaces
- Point-in-time recovery

We will be installing PostgreSQL version 11.3 on Windows 10 in this article.

There are three crucial steps for the installation of PostgreSQL as follows:

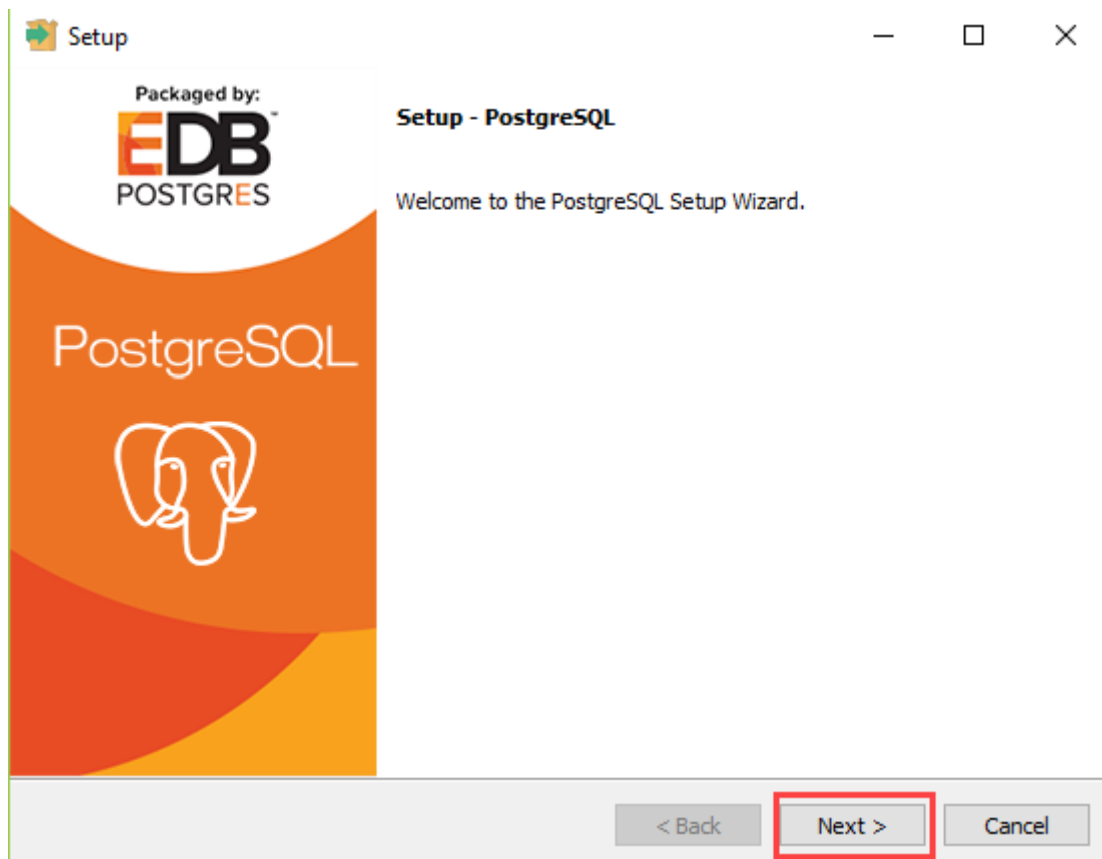
1. Download PostgreSQL installer for Windows
2. Install PostgreSQL
3. Verify the installation

You can download the latest stable PostgreSQL Installer specific to your Windows by clicking [here](#).

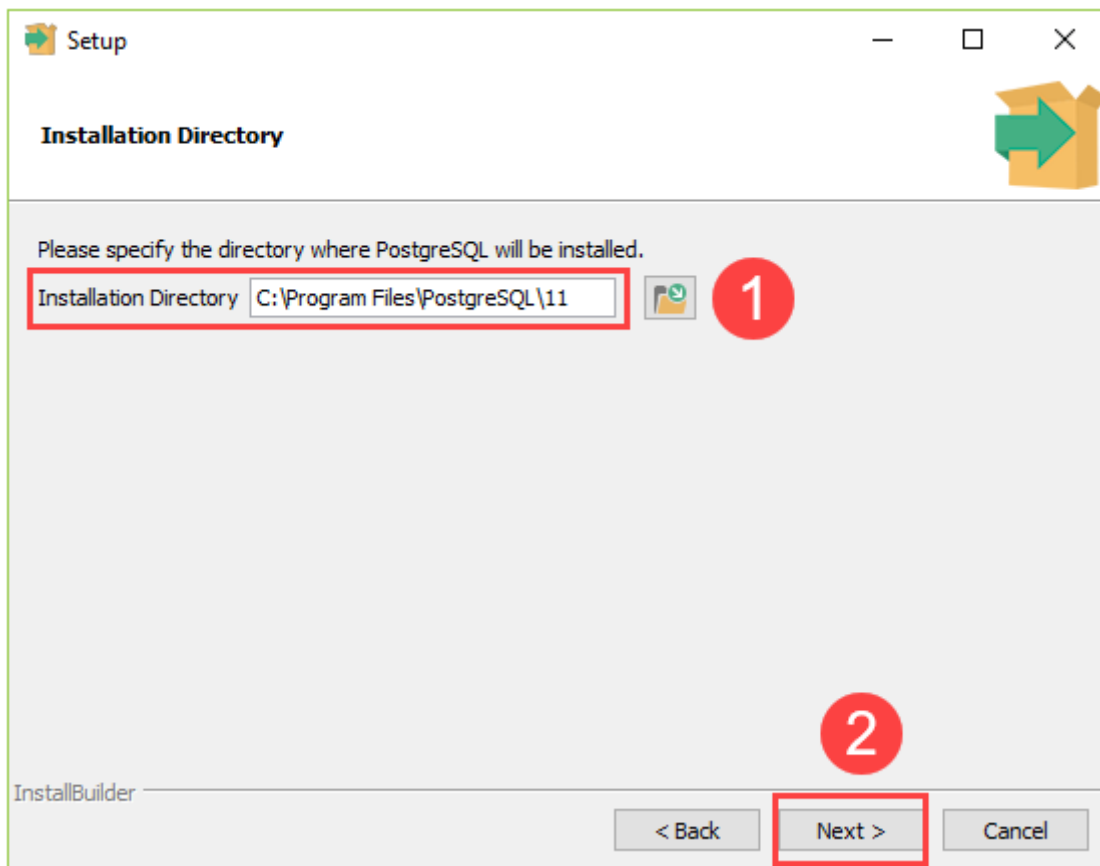
4.1 Installing the PostgreSQL

After downloading the installer double click on it and follow the below steps:

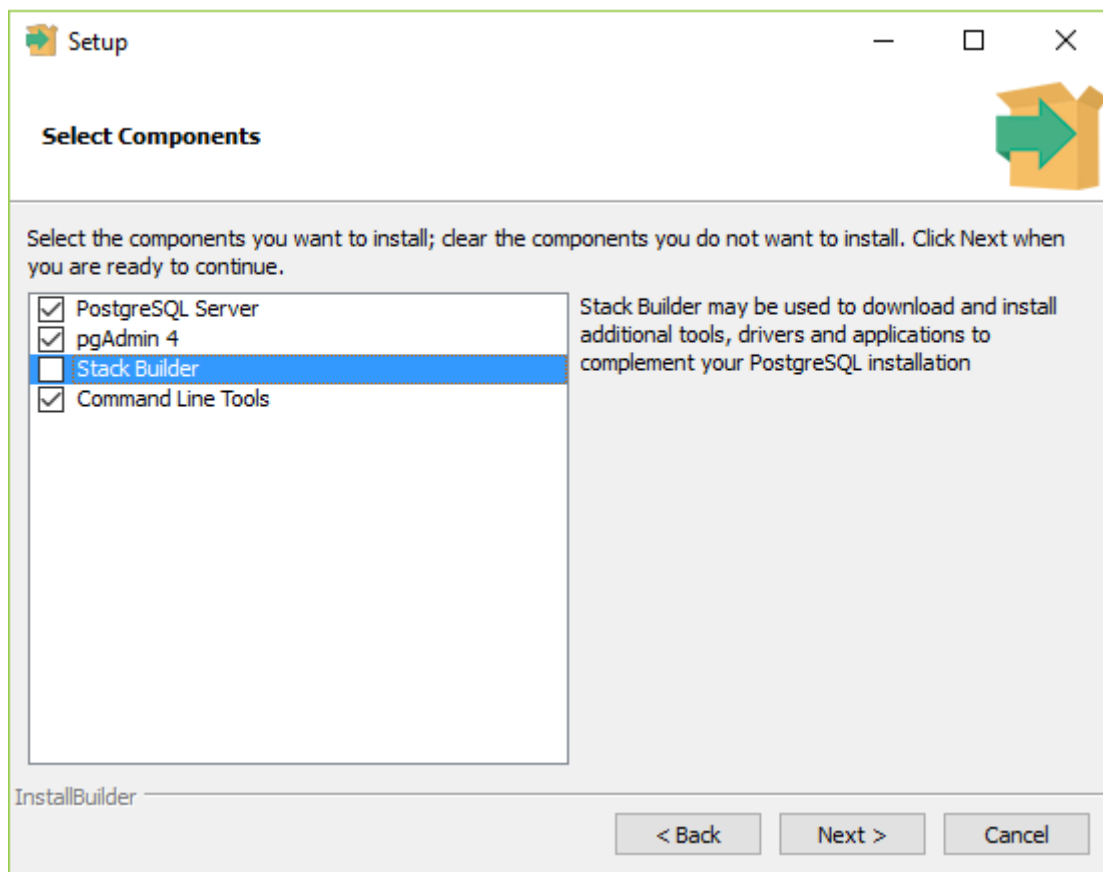
Step 1: Click the Next button



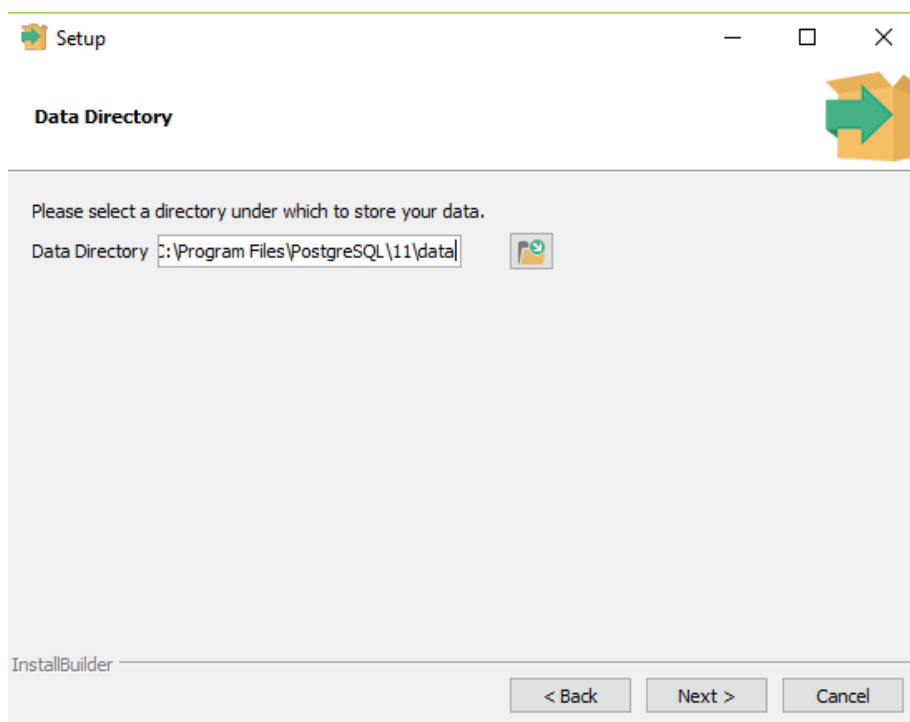
Step 2: Choose the installation folder, where you want PostgreSQL to be installed, and click on Next.



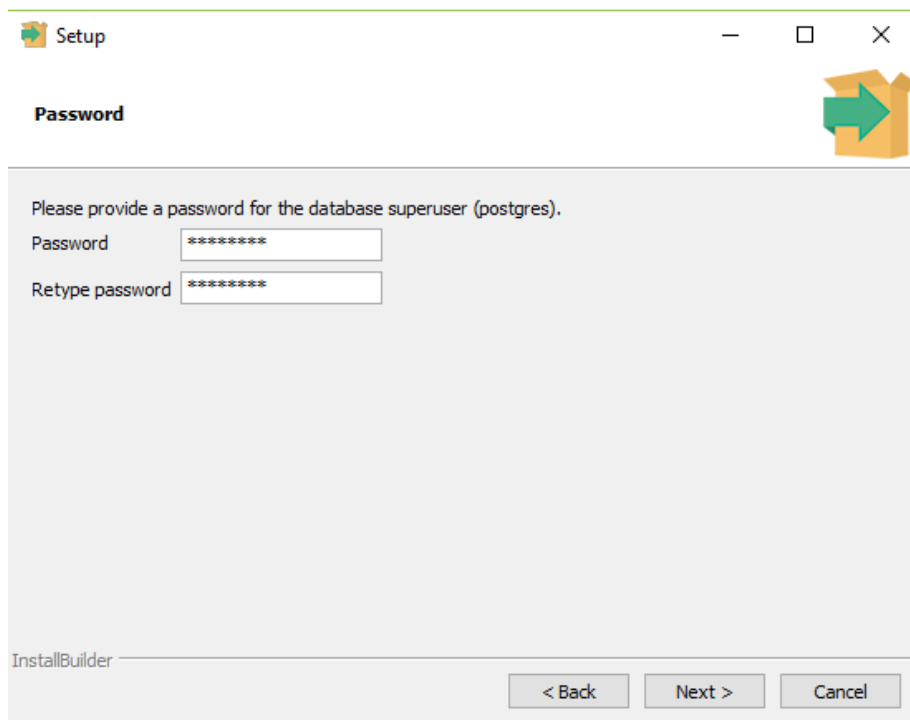
Step 3: Select the components as per your requirement to install and click the Next button.



Step 4: Select the database directory where you want to store the data and click on Next.

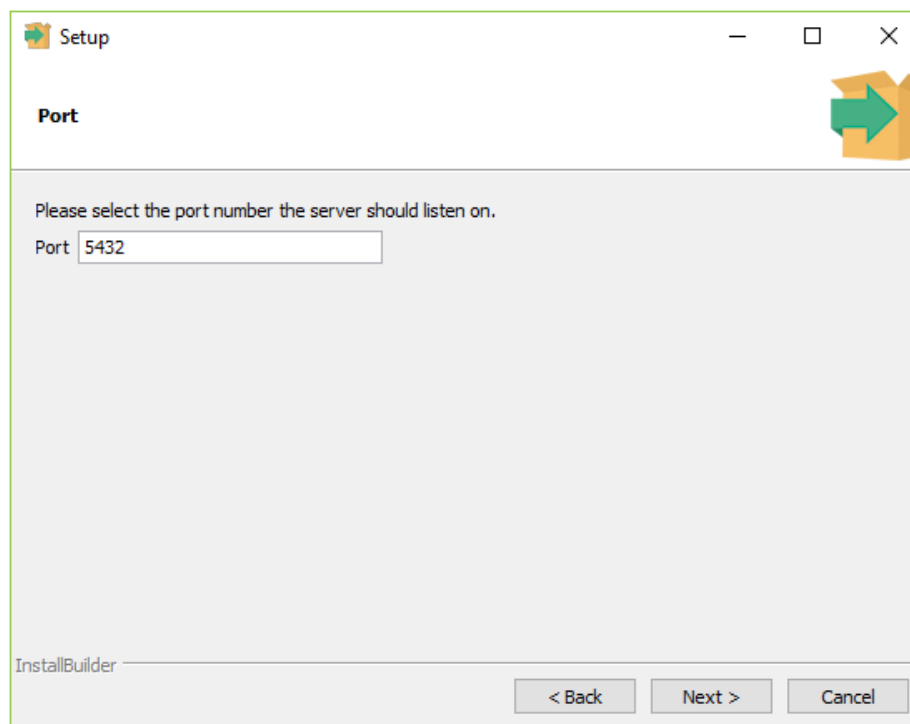


Step 5: Set the password for the database superuser (Postgres)



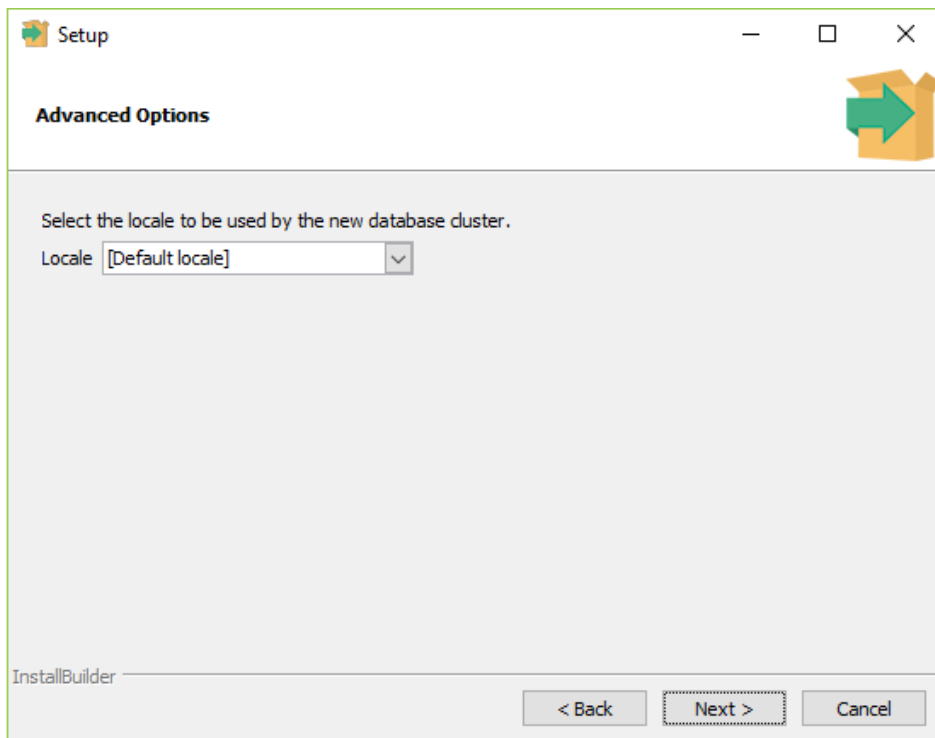
The screenshot shows a window titled "Setup" with a close button and a maximize button. The window has a title bar with a green icon. The main content area is titled "Password" and contains the text "Please provide a password for the database superuser (postgres)." Below this text are two input fields: "Password" and "Retype password", both containing seven asterisks. At the bottom of the window, there is a status bar with the text "InstallBuilder" and three buttons: "< Back", "Next >", and "Cancel".

Step 6: Set the port for PostgreSQL. Make sure that no other applications are using this port. If unsure leave it to its default (5432) and click on Next.

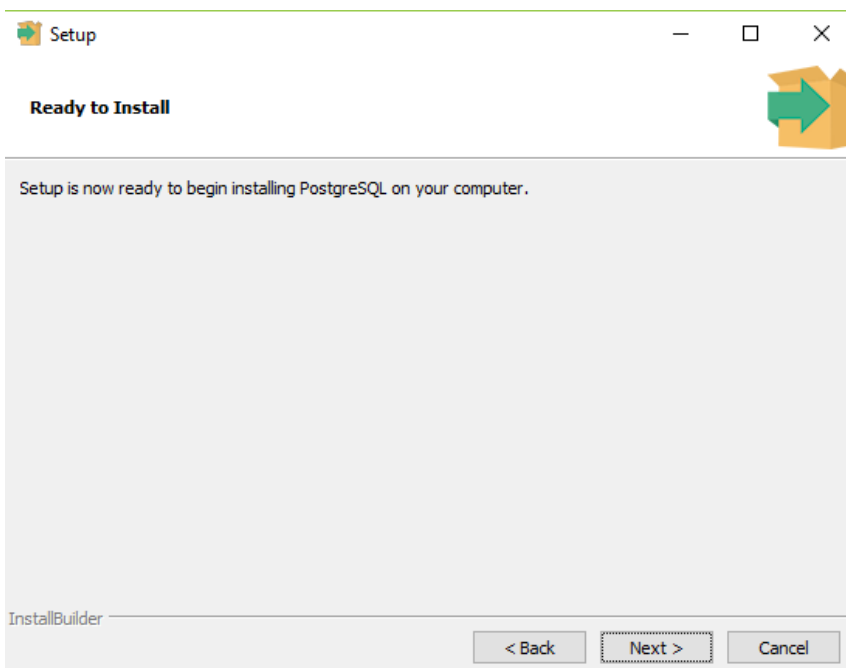


The screenshot shows a window titled "Setup" with a close button and a maximize button. The window has a title bar with a green icon. The main content area is titled "Port" and contains the text "Please select the port number the server should listen on." Below this text is a single input field labeled "Port" containing the number "5432". At the bottom of the window, there is a status bar with the text "InstallBuilder" and three buttons: "< Back", "Next >", and "Cancel".

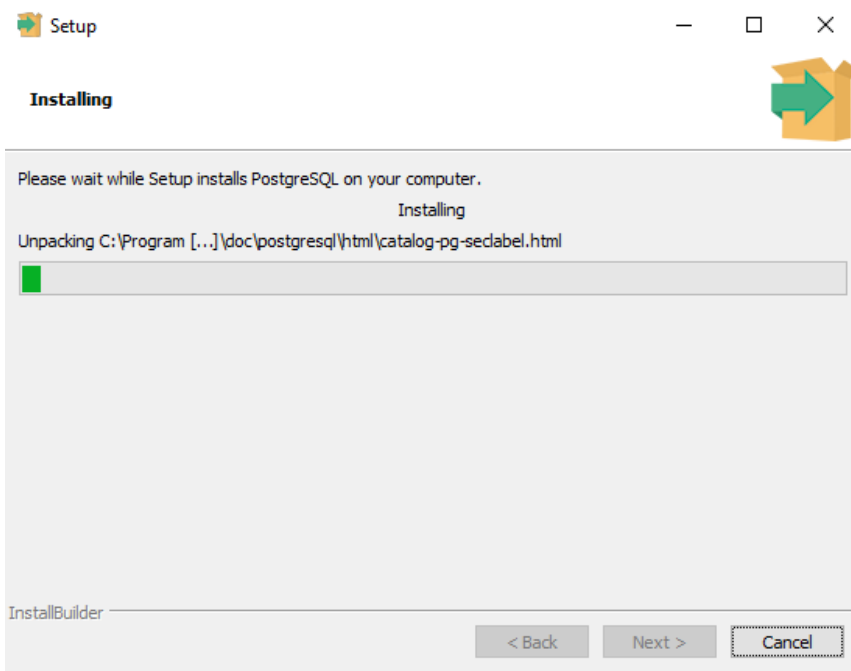
Step 7: Choose the default locale used by the database and click the Next button.



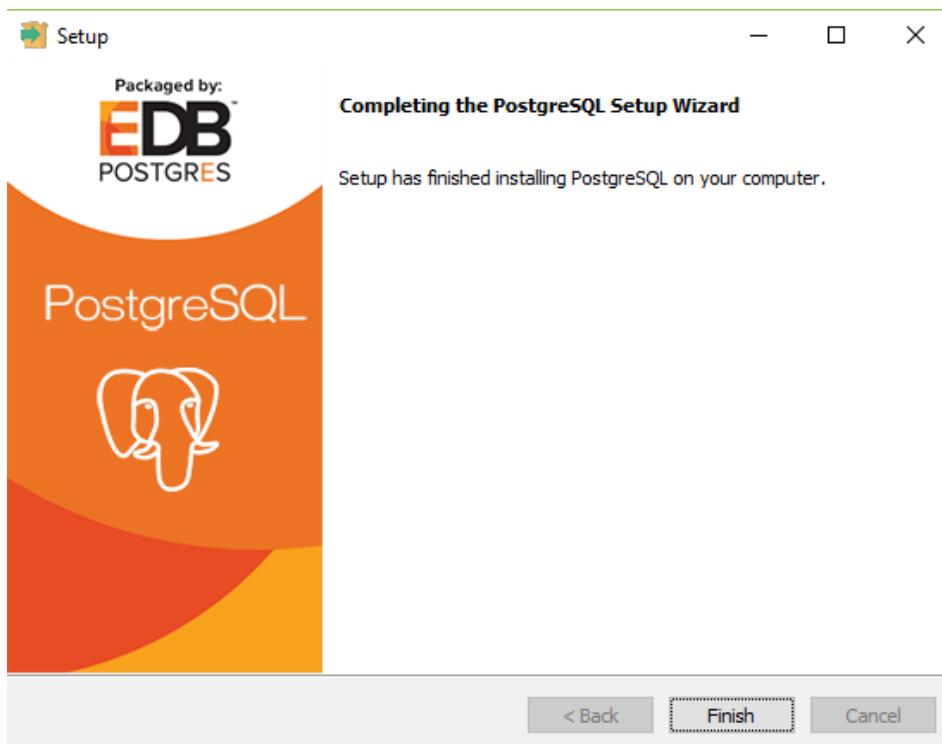
Step 8: Click the Next button to start the installation.



Wait for the installation to complete, it might take a few minutes.



Step 9: Click the Finish button to complete the PostgreSQL installation.



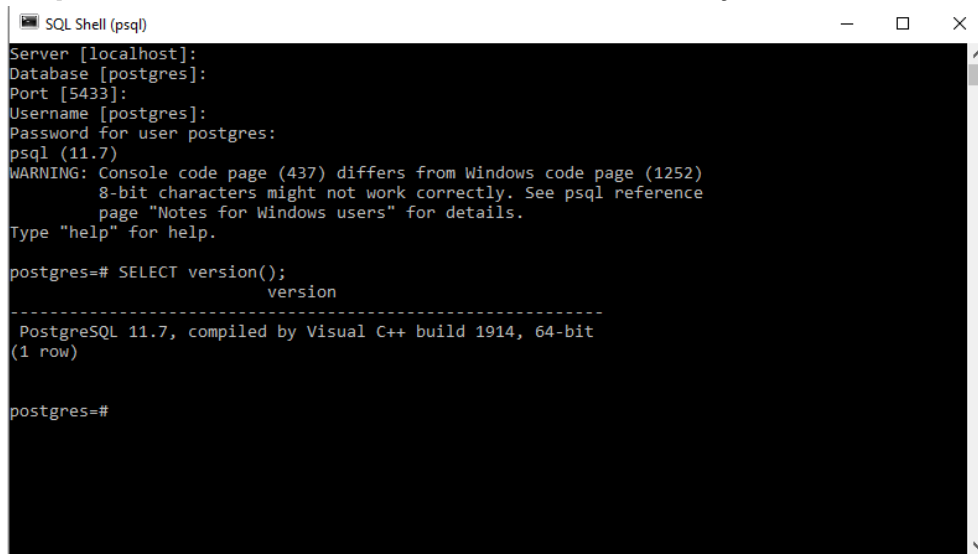
4.2 Verifying the Installation of PostgreSQL

There are couple of ways to verify the installation of PostgreSQL like connecting to the database server using some client applications like **pgAdmin** or **psql**. The quickest way though is to use the psql shell. For that follow the below steps:

Step 1: Search for the **psql shell** in the windows search bar and open it.

Step 2: Enter all the necessary information like the server, database, port, username, and password and press **Enter**.

Step 3: Use the command `SELECT version();` you will see the following result:



```
SQL Shell (psql)
Server [localhost]:
Database [postgres]:
Port [5433]:
Username [postgres]:
Password for user postgres:
psql (11.7)
WARNING: Console code page (437) differs from Windows code page (1252)
8-bit characters might not work correctly. See psql reference
page "Notes for Windows users" for details.
Type "help" for help.

postgres=# SELECT version();
              version
-----
PostgreSQL 11.7, compiled by Visual C++ build 1914, 64-bit
(1 row)

postgres=#
```

4.3 Database design using pgAdmin 4

pgAdmin 4 is a very popular open source platform fully dedicated to PostgreSQL and has a graphical user interface administration tools to manage your relational databases. Some features include a query tool for SQL statements and importing/exporting csv files.

The Entity-Relationship Diagram (ERD) tool is a database design tool that provides a graphical representation of database tables, columns, and inter-relationships. ERD can give sufficient information for the database administrator to follow when developing and maintaining the database. The ERD Tool allows you to:

- Design and visualize the database tables and their relationships.
- Add notes to the diagram.
- Auto-align the tables and links for cleaner visualization.
- Save the diagram and open it later to continue working on it.
- Generate ready to run SQL from the database design.
- Generate the database diagram for an existing database.
- Drag and drop tables from browser tree to the diagram.

PgAdmin will use your preferred web browser to display a graphical user interface. You don't need internet to view local servers. It will prompt you for a master password every time you open pgAdmin to get access. After getting access

click Servers(1) on the left side to open up your PostgreSQL 12 server. If you don't see a server, try restarting pgAdmin.

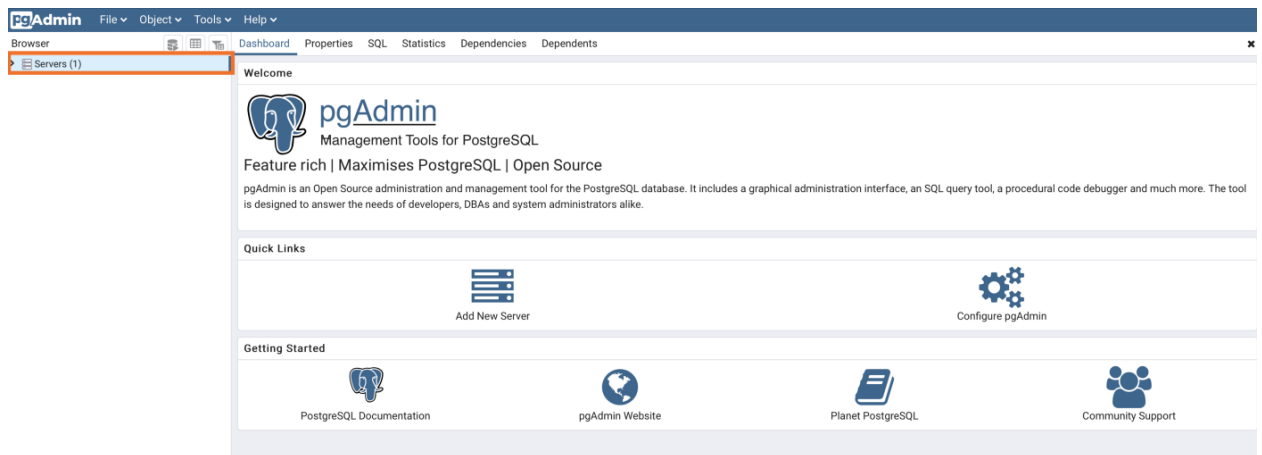


Figure 4.1 Main window of pgAdmin 4 tool.

After you open up your database you'll get a tree view menu according to the picture below. You will see a database named **postgres**. Selecting the database will bring up an activity dashboard to view traffic information (Figure 4.2).

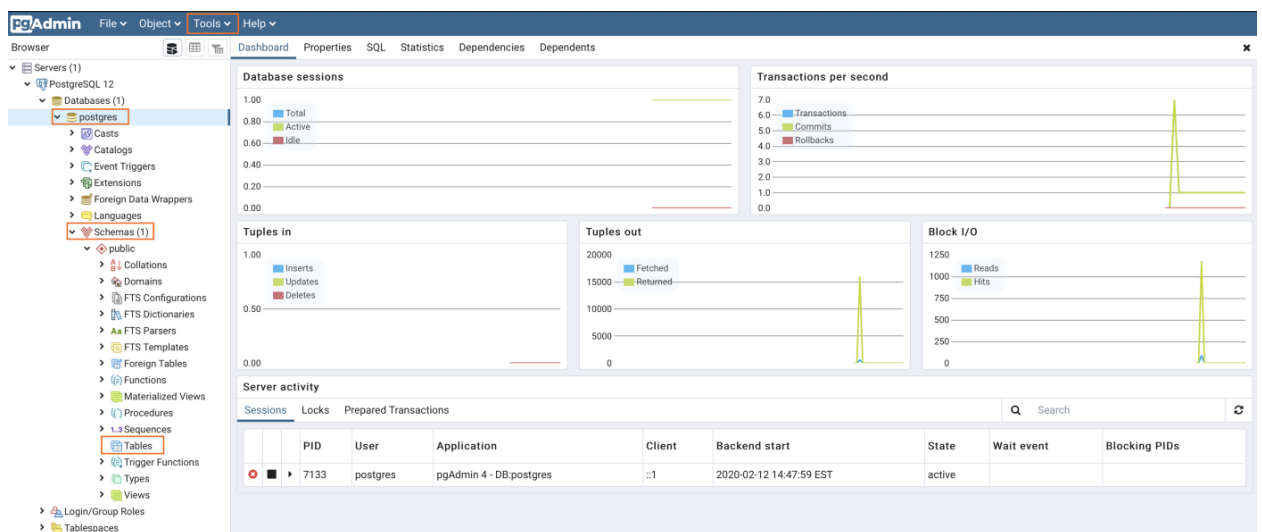


Figure 4.2 A tree of objects of Postgresql

You have access to useful tools by right-clicking on any object in the tree view menu. We can **create** a new database, schema, and tables. As well as viewing individual table data and customizing an existing table.

4.3.1 Create Table Example

Let's create a table by right clicking on **Tables** and click **Table...**

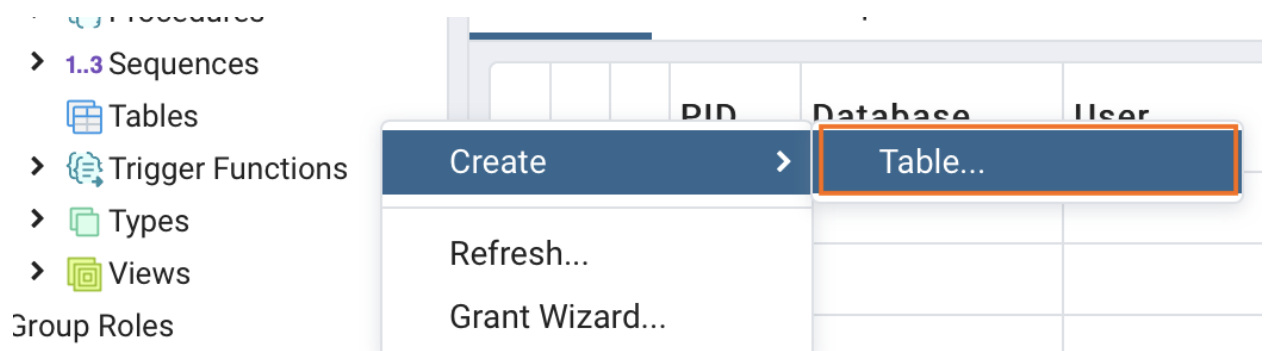


Figure 4.3 Creation of a table

Give your table a **name** and then click **Columns**

The screenshot shows the 'Create - Table' dialog box. The 'Columns' tab is selected and highlighted with an orange border. The 'Name' field is empty and has a red warning icon next to it. Below the 'Name' field, there are several fields: 'Owner' (set to 'postgres'), 'Schema' (set to 'public'), 'Tablespace' (set to 'Select an item...'), 'Partitioned table?' (set to 'No'), and 'Comment' (a large text area). The 'General' tab is also visible in the background.

Figure 4.4 Entering column data

Click the + symbol to add columns

Create - Table

×

General
Columns
Constraints
Advanced
Partitions
Parameters
Security
SQL

Inherited from table(s)

Select to inherit from...

Columns

+

	Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?

Figure 4.5 Insertion of a column

Name your column, select the **data type**, and give a **length** if needed. Then click **Save**

Create - Table

×

General
Columns
Constraints
Advanced
Partitions
Parameters
Security
SQL

Inherited from table(s)

Select to inherit from...

Columns

+

	Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?
	id	integer			<input type="checkbox"/> No	<input type="checkbox"/> No
	number	integer			<input type="checkbox"/> No	<input type="checkbox"/> No
	character	character varying	30		<input type="checkbox"/> No	<input type="checkbox"/> No

i

?

✕ Cancel

↺ Reset

💾 Save

Figure 4.6 Saving columns data

Now that you have created a table, view it under the **Tables** object. Right click and refresh if it didn't update.

4.3.2 Query Tool Example

When an object is selected under the database tree view menu, you can click the **Tools** tab and click **Query Tool**. Query tool brings up an editor to execute SQL statements. You can also right click the database and click **Query Tool ...**

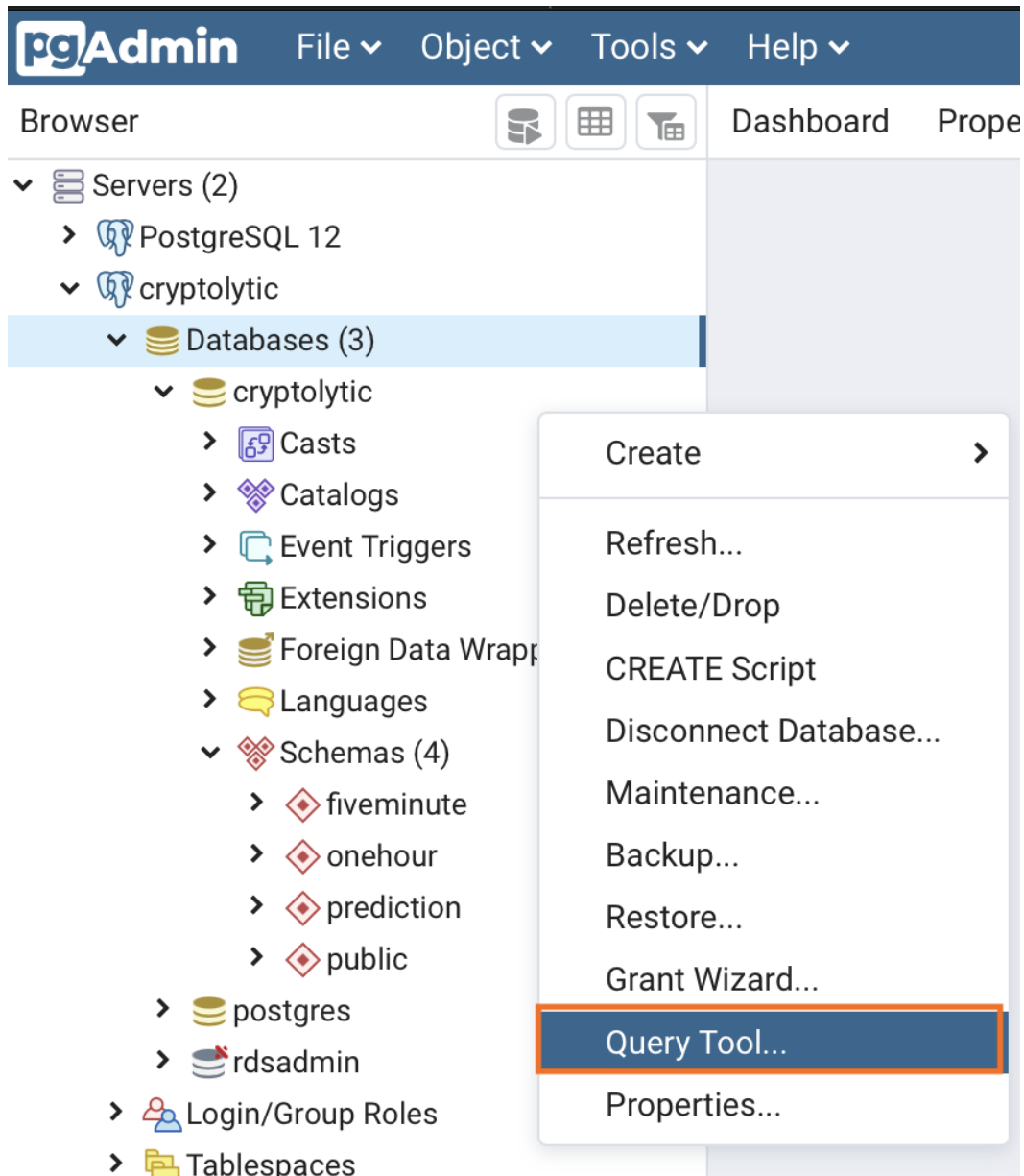


Figure 4.7 A Query tool

A tab called **query editor** will open up on the right. You can write SQL statements and click the **play button** to execute. Your results will show below the editor on the **Data Output** tab.

	p_time character varying (35)	c_time character varying (35)	exchange character varying (25)	trading_pair character varying (25)	prediction character varying (25)
1	2020-01-30 00:00:00	2020-01-30 05:30:14.739639	bitfinex	btc_usd	Down
2	2020-01-30 00:00:00	2020-01-30 05:30:15.596891	bitfinex	eth_usd	Up
3	2020-01-30 00:00:00	2020-01-30 05:30:16.527597	bitfinex	ltc_usd	Down
4	2020-01-30 00:00:00	2020-01-30 05:30:17.225373	coinbase_pro	btc_usd	Down
5	2020-01-30 00:00:00	2020-01-30 05:30:18.053876	coinbase_pro	eth_usd	Down
6	2020-01-30 00:00:00	2020-01-30 05:30:18.802752	coinbase_pro	ltc_usd	Up
7	2020-01-30 00:00:00	2020-01-30 05:30:19.578356	hitbtc	btc_usdt	Down
8	2020-01-30 00:00:00	2020-01-30 05:30:20.230374	hitbtc	eth_usdt	Up
9	2020-01-30 00:00:00	2020-01-30 05:30:20.928177	hitbtc	ltc_usdt	Down

Figure 4.8 A query execution results

4.3.3 pgAdmin 4 ERD Tool

The Entity-Relationship Diagram (ERD) tool is a database design tool that provides a graphical representation of database tables, columns, and inter-relationships. ERD can give sufficient information for the database administrator to follow when developing and maintaining the database. The ERD Tool allows you to:

- Design and visualize the database tables and their relationships.
- Add notes to the diagram.
- Auto-align the tables and links for cleaner visualization.
- Save the diagram and open it later to continue working on it.
- Generate ready to run SQL from the database design.
- Generate the database diagram for an existing database.
- Drag and drop tables from browser tree to the diagram.

Toolbar

The *ERD Tool* toolbar uses context-sensitive icons that provide shortcuts to frequently performed tasks. The option is enabled for the highlighted icon and is disabled for the grayed-out icon.



Figure 4.9 ERD toolbar

Hover over an icon on Toolbar to display a tooltip that describes the icon's functionality.

Table Dialog

A dialog box titled 'Table: users (public)'. It has four tabs: 'General', 'Columns', 'Advanced', and 'Constraints'. The 'General' tab is selected. It contains three fields: 'Name' with the value 'users', 'Schema' with the value 'public' and a dropdown arrow, and 'Comment' with a large text area. At the bottom, there are three buttons: 'Close', 'Reset', and 'Save'.

Figure 4.10 Table dialog

The table dialog allows you to:

- Change the table structure details.
- It can be used edit an existing table or add a new one.
- Refer table dialog for information on different fields.

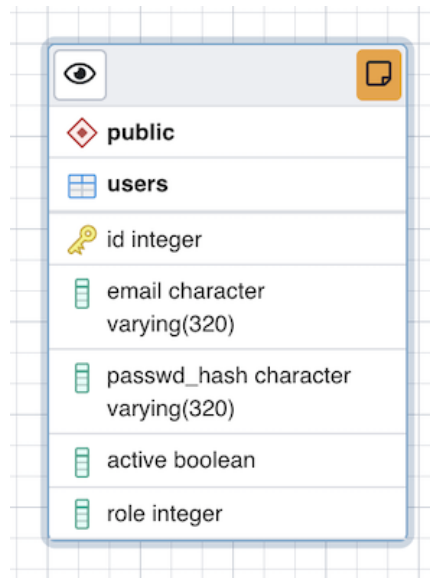
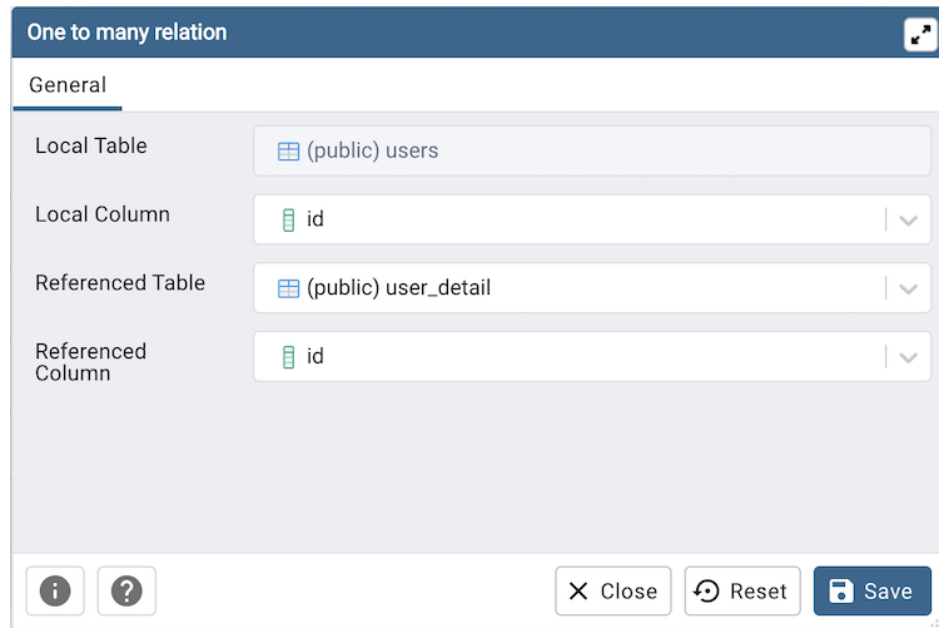


Figure 4.11 Table attributes

The table node (Figure 4.11) shows table details in a graphical representation:

- The top bar has a *details toggle button* that is used to toggle column details visibility. There is also a *note button* that is visible only if there is some note added. you can click on this button to quickly change the note.
- The first row shows the schema name of the table. Eg. *public* in above image.
- The second row shows the table name. Eg. *users* in above image.
- All other rows below the table name are the columns of the table along with data type. If the column is a primary key then it will have lock key icon eg. *id* is the primary key in above image. Otherwise, it will have column icon.
- you can click on the node and drag to move on the canvas.
- Upon double click on the table node or by clicking the edit button from the toolbar, the table dialog opens where you can change the table details. Refer table dialog for information on different fields.

The One to Many Link Dialog



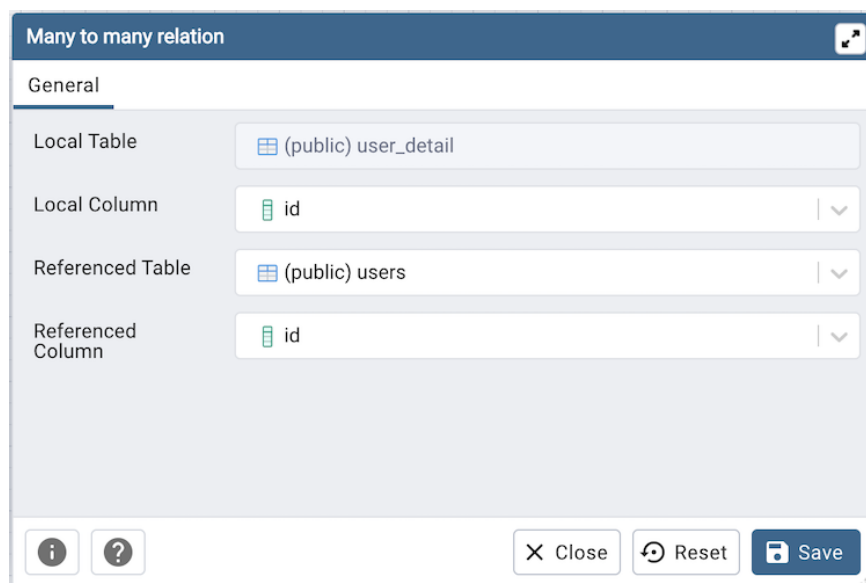
The 'One to many relation' dialog box is shown with the 'General' tab selected. It contains four input fields: 'Local Table' with '(public) users', 'Local Column' with 'id', 'Referenced Table' with '(public) user_detail', and 'Referenced Column' with 'id'. Each field has a table icon on the left and a dropdown arrow on the right. At the bottom, there are three buttons: 'Close' (with an 'X' icon), 'Reset' (with a circular arrow icon), and 'Save' (with a floppy disk icon). There are also two small circular icons with an 'i' and a '?' on the left side of the bottom bar.

Figure 4.12 One to many relation

The one to many link dialog allows you to:

- Add a foreign key relationship between two tables.
- *Local Table* is the table that references a table and has the *many* end point.
- *Local Column* the column that references.
- *Referenced Table* is the table that is being referred and has the *one* end point.
- *Referenced Column* the column that is being referred.

The Many to Many Link Dialog



The 'Many to many relation' dialog box is shown with the 'General' tab selected. It contains four input fields: 'Local Table' with '(public) user_detail', 'Local Column' with 'id', 'Referenced Table' with '(public) users', and 'Referenced Column' with 'id'. Each field has a table icon on the left and a dropdown arrow on the right. At the bottom, there are three buttons: 'Close' (with an 'X' icon), 'Reset' (with a circular arrow icon), and 'Save' (with a floppy disk icon). There are also two small circular icons with an 'i' and a '?' on the left side of the bottom bar.

Figure 4.13 Many to many relation

The many to many link dialog allows you to:

- Add a many to many relationship between two tables.
- It creates a relationship tables having columns derived from the two tables and link them to the tables.
- *Left Table* is the first table that is to be linked. It will receive the *one* endpoint of the link with the new relation table.
- *Left Column* the column of the first table, that will always be a primary key.
- *Right Table* is the second table that is to be linked. It will receive the *one* endpoint of the link with the new relation table.
- *Right Column* the column of the second table, that will always be a primary key.

The Table Link

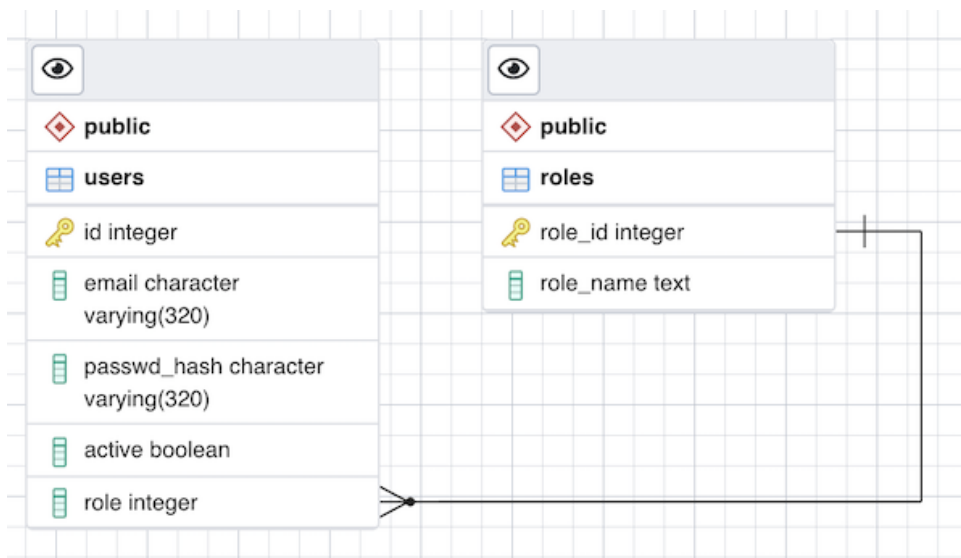


Figure 4.14 The table link

The table link shows relationship between tables:

- The single line endpoint of the link shows the column that is being referred.
- The three line endpoint of the link shows the column that refers.
- If one of the columns that is being referred or that refers is removed from the table then the link will get dropped.
- you can click on the link and drag to move on the canvas.

4.3.4 PostgreSQL – Data Types

The following data types are supported by PostgreSQL:

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 UUID (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*]

Now let's get an overview of the above-mentioned data types.

Boolean

In PostgreSQL, the “bool” or “boolean” keyword is used to initialize a Boolean data type. These data types can hold *true*, *false*, and *null* values. A boolean data type is stored in the database according to the following:

1, yes, y, t, true values are converted to true

0, no, false, f values are converted to false

When queried for these boolean data types are converted and returned according to the following:

t to true

f to false

space to null

Characters

PostgreSQL has three character data types namely, **CHAR(n)**, **VARCHAR(n)**, and **TEXT**.

CHAR(n) is used for data(string) with a fixed-length of characters with padded spaces. In case the length of the string is smaller than the value of “n”, then the rest of the remaining spaces are automatically padded. Similarly for a string with a length greater than the value of “n”, PostgreSQL throws an error.

VARCHAR(n) is the variable-length character string. Similar to *CHAR(n)*, it can store “n” length data. But unlike *CHAR(n)* no padding is done in case the data length is smaller than the value of “n”.

TEXT is the variable-length character string. It can store data with unlimited length.

Numeric

PostgreSQL has 2 types of numbers namely, integers and floating-point numbers.

1. Integer

Small integer (SMALLINT) has a range -32, 768 to 32, 767 and has a size of 2-byte.

Integer (INT) has a range -2, 147, 483, 648 to 2, 147, 483, 647 and has a size of 4-byte.

Serial (SERIAL) works similar to the integers except these are automatically generated in the columns by PostgreSQL.

Floating-point number

float(n) is used for floating-point numbers with n precision and can have a maximum of 8-bytes.

float8 or *real* is used to represent 4-byte floating-point numbers.

A real number *N(d,p)* meaning with d number of digits and p number of decimal points after, are part of *numeric* or *numeric(d, p)*. These are generally very precise.

Temporal data type

This data type is used to store date-time data. PostgreSQL has 5 temporal data type:

DATE is used to store the dates only.

TIME is used to stores the time of day values.

TIMESTAMP is used to stores both date and time values.

TIMESTAMPTZ is used to store a timezone-aware timestamp data type.

INTERVAL is used to store periods of time.

Arrays

In PostgreSQL, an array column can be used to store an array of strings or an array of integers etc. It can be handy when storing data like storing days of months, a year, or even a week, etc.

JSON

PostgreSQL supports 2 types of JSON types namely JSON and JSONB(Binary JSON). The JSON data type is used to store plain JSON data that get parsed every time it's called by a query. Whereas the JSONB data type is used to store JSON data in a binary format. It is one hand makes querying data faster whereas slows down the data insertion process as it supports indexing of table data.

UUID

The UUID data type allows you to store Universal Unique Identifiers defined by RFC 4122. The UUID values guarantee a better uniqueness than SERIAL and can be used to hide sensitive data exposed to the public such as values of id in URL.

The UUID stands for Unique Universal Identifiers. These are used to give a unique ID to a data that is unique throughout the database. The UUID data type are used to store UUID of the data defined by RFC 4122. These are generally used to protect sensitive data like credit card information and is better compared to SERIAL data type in the context of uniqueness.

Special data types

In addition to the primitive data types, PostgreSQL also supports some special data types that are related to network or geometric. These special data types are listed below:

box: It is used to store rectangular box.

point: It is used to store geometric pair of numbers.

lseg: It is used to store line segment.

point: It is used to store geometric pair of numbers.

polygon: It is used to store closed geometric.

inet: It is used to store an IP4 address.

macaddr: It is used to store a MAC address.

5 Basics of SQL language in PostgreSQL

One of the most common tasks, when you work with the database, is to query data from tables by using the 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.

The SELECT statement has the following clauses:

- Select distinct rows using DISTINCT operator.
- Sort rows using ORDER BY clause.
- Filter rows using WHERE clause.
- Select a subset of rows from a table using LIMIT or FETCH clause.
- Group rows into groups using GROUP BY clause.
- Filter groups using HAVING clause.
- Join with other tables using joins such as INNER JOIN, LEFT JOIN, FULL OUTER JOIN, CROSS JOIN clauses.
- Perform set operations using UNION, INTERSECT, and EXCEPT.

5.1 PostgreSQL SELECT examples

Let's take a look at some examples of using PostgreSQL SELECT statement.

We will use the following customer table in the sample database for the demonstration.

customer
* customer_id
store_id
first_name
last_name
email
address_id
activebool
create_date
last_update
active

Figure 5.1 The sample table

5.1.1 Using PostgreSQL SELECT statement to query data from all columns of a table example

The following query uses the SELECT statement to select data from all columns of the customer table:

```
SELECT * FROM customer;
```

	customer_id integer	store_id smallint	first_name character varying (45)	last_name character varying (45)	email character varying (50)	address_id smallint	activebool boolean	create_date date	last_update timestamp without time zone	active integer
1	524	1	Jared	Ely	jared.ely@sakilacustomer.org	530	true	2006-02-14	2013-05-26 14:49:45.738	1
2	1	1	Mary	Smith	mary.smith@sakilacustomer...	5	true	2006-02-14	2013-05-26 14:49:45.738	1
3	2	1	Patricia	Johnson	patricia.johnson@sakilacust...	6	true	2006-02-14	2013-05-26 14:49:45.738	1
4	3	1	Linda	Williams	linda.williams@sakilacusto...	7	true	2006-02-14	2013-05-26 14:49:45.738	1
5	4	2	Barbara	Jones	barbara.jones@sakilacusto...	8	true	2006-02-14	2013-05-26 14:49:45.738	1
6	5	1	Elizabeth	Brown	elizabeth.brown@sakilacust...	9	true	2006-02-14	2013-05-26 14:49:45.738	1
7	6	2	Jennifer	Davis	jennifer.davis@sakilacustom...	10	true	2006-02-14	2013-05-26 14:49:45.738	1
8	7	1	Maria	Miller	maria.miller@sakilacustome...	11	true	2006-02-14	2013-05-26 14:49:45.738	1
9	8	2	Susan	Wilson	susan.wilson@sakilacustom...	12	true	2006-02-14	2013-05-26 14:49:45.738	1
10	9	2	Margaret	Moore	margaret.moore@sakilacust...	13	true	2006-02-14	2013-05-26 14:49:45.738	1
11	10	1	Dorothy	Taylor	dorothy.taylor@sakilacusto...	14	true	2006-02-14	2013-05-26 14:49:45.738	1

Figure 5.2 Select from all columns

In this example, we used an asterisk (*) in the SELECT clause, which is a shorthand for all columns. Instead of listing all columns in the SELECT clause, we just used the asterisk (*) to save some typing.

However, it is not a good practice to use the asterisk (*) in the SELECT statement when you embed SQL statements in the application code like Python, Java, Node.js, or PHP due to the following reasons:

- Database performance. Suppose you have a table with many columns and a lot of data, the SELECT statement with the asterisk (*) shorthand will select data from all the columns of the table, which may not be necessary to the application.
- Application performance. Retrieving unnecessary data from the database increases the traffic between the database server and application server. In consequence, your applications may be slower to respond and less scalable.

Because of these reasons, it is a good practice to explicitly specify the column names in the SELECT clause whenever possible to get only necessary data from the database.

And you should only use the asterisk (*) shorthand for the ad-hoc queries that examine data from the database.

5.1.2 Using PostgreSQL `SELECT` statement with expressions example

The following example uses the `SELECT` statement to return full names and emails of all customers:

```
SELECT
```

```
    first_name || ' ' || last_name,
```

```
    email
```

```
FROM
```

```
    customer;
```

In this example, we used the concatenation operator `||` to concatenate the first name, space, and last name of every customer.

	?column? text	email character varying (50)
1	Jared Ely	jared.ely@sakilacustomer.org
2	Mary Smith	mary.smith@sakilacustomer.org
3	Patricia Johnson	patricia.johnson@sakilacustomer.org
4	Linda Williams	linda.williams@sakilacustomer.org
5	Barbara Jones	barbara.jones@sakilacustomer.org
6	Elizabeth Brown	elizabeth.brown@sakilacustomer.org
7	Jennifer Davis	jennifer.davis@sakilacustomer.org
8	Maria Miller	maria.miller@sakilacustomer.org
9	Susan Wilson	susan.wilson@sakilacustomer.org
10	Margaret Moore	margaret.moore@sakilacustomer.org
11	Dorothy Taylor	dorothy.taylor@sakilacustomer.org

Figure 5.3 Select with concatenation

5.2 PostgreSQL `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.

The following illustrates the syntax of the `ORDER BY` clause:

```
SELECT
    select_list
FROM
    table_name
ORDER BY
    sort_expression1 [ASC | DESC],
    ...
    sort_expressionN [ASC | DESC];
```

In this syntax:

- First, specify a sort expression, which can be a column or an expression, that you want to sort after the ORDER BY keywords. If you want to sort the result set based on multiple columns or expressions, you need to place a comma (,) between two columns or expressions to separate them.
- Second, you use the ASC option to sort rows in ascending order and the DESC option to sort rows in descending order. If you omit the ASC or DESC option, the ORDER BY uses ASC by default.

5.2.1 Using PostgreSQL ORDER BY clause to sort rows by one column

The following query uses the ORDER BY clause to sort customers by their first names in ascending order:

```
SELECT
    first_name,
    last_name
FROM
    customer
ORDER BY
    first_name ASC;
```

	first_name character varying (45)	last_name character varying (45)
1	Aaron	Selby
2	Adam	Gooch
3	Adrian	Clary
4	Agnes	Bishop
5	Alan	Kahn
6	Albert	Crouse
7	Alberto	Henning
8	Alex	Gresham
9	Alexander	Fennell
10	Alfred	Casillas
11	Alfredo	Mcadams
12	Alice	Stewart
13	Alicia	Mills

Figure 5.4 Select with ORDER BY

5.2.2 Using PostgreSQL ORDER BY clause to sort rows by one column in descending order

The following statement selects the first name and last name from the customer table and sorts the rows by values in the last name column in descending order:

```
SELECT
    first_name,
    last_name
FROM
    customer
ORDER BY
    last_name DESC;
```

	first_name character varying (45)	last_name character varying (45)
1	Cynthia	Young
2	Marvin	Yee
3	Luis	Yanez
4	Brian	Wyman
5	Brenda	Wright
6	Tyler	Wren
7	Florence	Woods
8	Lori	Wood
9	Virgil	Wofford
10	Darren	Windham
11	Susan	Wilson
12	Bernice	Willis
13	Gina	Williamson
14	Linda	Williams
15	Jon	Wiles

Figure 5.5 Select with ORDER BY descending

5.2.3 Using PostgreSQL ORDER BY clause to sort rows by multiple columns

The following statement selects the first name and last name from the customer table and sorts the rows by the first name in ascending order and last name in descending order:

```
SELECT
    first_name,
    last_name
FROM
    customer
ORDER BY
    first_name ASC,
    last_name DESC;
```


	first_name character varying (45)	last_name character varying (45)
321	Kathleen	Adams
322	Kathryn	Coleman
323	Kathy	James
324	Katie	Elliott
325	Kay	Caldwell
326	Keith	Rico
327	Kelly	Torres
328	Kelly	Knott
329	Ken	Prewitt
330	Kenneth	Gooden
331	Kent	Arsenault
332	Kevin	Schuler
333	Kim	Cruz
334	Kimberly	Lee
335	Kirk	Stclair
336	Kristen	Chavez
337	Kristin	Johnston
338	Kristina	Chambers
339	Kurt	Emmons

Figure 5.6 Select with ORDER BY two columns

In this example, the ORDER BY clause sorts rows by values in the first name column first. And then it sorts the sorted rows by values in the last name column.

As you can see clearly from the output, two customers with the same first name Kelly have the last name sorted in descending order.

5.3 PostgreSQL WHERE clause

The syntax of the PostgreSQL WHERE clause is as follows:

SELECT select_list

FROM table_name

WHERE condition

ORDER BY sort_expression

The WHERE clause appears right after the FROM clause of the SELECT statement. The WHERE clause uses the condition to filter the rows returned from the SELECT clause.

The condition must evaluate to true, false, or unknown. It can be a boolean expression or a combination of boolean expressions using the AND and OR operators.

The query returns only rows that satisfy the condition in the WHERE clause. In other words, only rows that cause the condition evaluates to true will be included in the result set.

If you use column aliases in the SELECT clause, you cannot use them in the WHERE clause.

Besides the SELECT statement, you can use the WHERE clause in the UPDATE and DELETE statement to specify rows to be updated or deleted.

To form the condition in the WHERE clause, you use comparison and logical operators:

Table 5.1 SELECT WHERE condition 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
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

5.3.1 Using WHERE clause with the equal (=) operator example

The following statement uses the WHERE clause customers whose first names are Jamie:

```
SELECT
    last_name,
    first_name
FROM
    customer
WHERE
    first_name = 'Jamie';
```

	last_name character varying (45)	first_name character varying (45)
1	Rice	Jamie
2	Waugh	Jamie

Figure 5.7 Filter by 'Jamie'

5.3.2 Using WHERE clause with the AND operator example

The following example finds customers whose first name and last name are Jamie and rice by using the AND logical operator to combine two Boolean expressions:

```
SELECT
    last_name,
    first_name
FROM
    customer
WHERE
    first_name = 'Jamie' AND
    last_name = 'Rice';
```

	last_name character varying (45)	first_name character varying (45)
1	Rice	Jamie

Figure 5.8 Filter by 'Rice'

5.3.3 Using the WHERE clause with the LIKE operator example

To find a string that matches a specified pattern, you use the LIKE operator. The following example returns all customers whose first names start with the string Ann:

```
SELECT
    first_name,
    last_name
FROM
    customer
WHERE
    first_name LIKE 'Ann%'
```

	first_name character varying (45)	last_name character varying (45)
1	Anna	Hill
2	Ann	Evans
3	Anne	Powell
4	Annie	Russell
5	Annette	Olson

Figure 5.9 Filter using 'LIKE' operator

5.4 PostgreSQL Joins

PostgreSQL join is used to combine columns from one (self-join) or more tables based on the values of the common columns between related tables. The common columns are typically the primary key columns of the first table and foreign key columns of the second table.

PostgreSQL supports inner join, left join, right join, full outer join, cross join, natural join, and a special kind of join called self-join.

Setting up sample tables

Suppose you have two tables called basket_a and basket_b that store fruits:

```
CREATE TABLE basket_a (  
    a INT PRIMARY KEY,  
    fruit_a VARCHAR (100) NOT NULL  
);
```

```
CREATE TABLE basket_b (  
    b INT PRIMARY KEY,  
    fruit_b VARCHAR (100) NOT NULL  
);
```

```
INSERT INTO basket_a (a, fruit_a)
```

```
VALUES
```

```
(1, 'Apple'),  
(2, 'Orange'),  
(3, 'Banana'),  
(4, 'Cucumber');
```

```
INSERT INTO basket_b (b, fruit_b)
```

```
VALUES
```

```
(1, 'Orange'),  
(2, 'Apple'),  
(3, 'Watermelon'),  
(4, 'Pear');
```

The tables have some common fruits such as apple and orange.

The following statement returns data from the basket_a table:

	a	fruit_a
	integer	character varying (100)
1	1	Apple
2	2	Orange
3	3	Banana
4	4	Cucumber

Figure 5.10 Basket_a table

And the following statement returns data from the basket_b table:

	b	fruit_b
	integer	character varying (100)
1	1	Orange
2	2	Apple
3	3	Watermelon
4	4	Pear

Figure 5.11 Basket_b table

5.4.1 PostgreSQL inner join

The following statement joins the first table (basket_a) with the second table (basket_b) by matching the values in the fruit_a and fruit_b columns:

SELECT

a,

fruit_a,

b,

fruit_b

FROM

basket_a

INNER JOIN basket_b

ON fruit_a = fruit_b;

	a	fruit_a	b	fruit_b
	integer	character varying (100)	integer	character varying (100)
1	1	Apple	2	Apple
2	2	Orange	1	Orange

Figure 5.12 The Inner Join example

The inner join examines each row in the first table (basket_a). It compares the value in the fruit_a column with the value in the fruit_b column of each row in the second table (basket_b). If these values are equal, the inner join creates a new row that contains columns from both tables and adds this new row to the result set.

5.4.2 PostgreSQL left join

The following statement uses the left join clause to join the basket_a table with the basket_b table. In the left join context, the first table is called the left table and the second table is called the right table.

```
SELECT
    a,
    fruit_a,
    b,
    fruit_b
FROM
    basket_a
LEFT JOIN basket_b
    ON fruit_a = fruit_b;
```

	a integer	fruit_a character varying (100)	b integer	fruit_b character varying (100)
1	1	Apple	2	Apple
2	2	Orange	1	Orange
3	3	Banana	[null]	[null]
4	4	Cucumber	[null]	[null]

Figure 5.13 The LEFT JOIN example

The left join starts selecting data from the left table. It compares values in the fruit_a column with the values in the fruit_b column in the basket_b table.

If these values are equal, the left join creates a new row that contains columns of both tables and adds this new row to the result set. (see the row #1 and #2 in the result set).

In case the values do not equal, the left join also creates a new row that contains columns from both tables and adds it to the result set. However, it fills the

columns of the right table (basket_b) with null. (see the row #3 and #4 in the result set).

5.4.3 PostgreSQL right join

The right join is a reversed version of the left join. The right join starts selecting data from the right table. It compares each value in the fruit_b column of every row in the right table with each value in the fruit_a column of every row in the fruit_a table.

If these values are equal, the right join creates a new row that contains columns from both tables.

In case these values are not equal, the right join also creates a new row that contains columns from both tables. However, it fills the columns in the left table with NULL.

The following statement uses the right join to join the basket_a table with the basket_b table:

```
SELECT
    a,
    fruit_a,
    b,
    fruit_b
FROM
    basket_a
RIGHT JOIN basket_b ON fruit_a = fruit_b;
```

Here is the output:

	a integer	fruit_a character varying (100)	b integer	fruit_b character varying (100)
1	2	Orange	1	Orange
2	1	Apple	2	Apple
3	[null]	[null]	3	Watermelon
4	[null]	[null]	4	Pear

Figure 5.14 The Right Join example

5.4.4 PostgreSQL full outer join

The full outer join or full join returns a result set that contains all rows from both left and right tables, with the matching rows from both sides if available. In case there is no match, the columns of the table will be filled with NULL.

```
SELECT
    a,
    fruit_a,
    b,
    fruit_b
FROM
    basket_a
FULL OUTER JOIN basket_b
    ON fruit_a = fruit_b;
```

Code language: SQL (Structured Query Language) (sql)

Output:

	a integer	fruit_a character varying (100)	b integer	fruit_b character varying (100)
1	1	Apple	2	Apple
2	2	Orange	1	Orange
3	3	Banana	[null]	[null]
4	4	Cucumber	[null]	[null]
5	[null]	[null]	3	Watermelon
6	[null]	[null]	4	Pear

Figure 5.15 The Full Outer join example

5.5 PostgreSQL GROUP BY clause

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,
    ...;
```

In this syntax:

First, select the columns that you want to group e.g., column1 and column2, and column that you want to apply an aggregate function (column3).

Second, list the columns that you want to group in the GROUP BY clause.

The statement clause divides the rows by the values of the columns specified in the GROUP BY clause and calculates a value for each group.

It's possible to use other clauses of the SELECT statement with the GROUP BY clause.

PostgreSQL evaluates the GROUP BY clause after the FROM and WHERE clauses and before the HAVING SELECT, DISTINCT, ORDER BY and LIMIT clauses

5.5.1 Using PostgreSQL GROUP BY **without an aggregate function example**

You can use the GROUP BY clause without applying an aggregate function. The following query gets data from the payment table and groups the result by customer id.

```
SELECT
    customer_id
```

```
FROM
    payment
GROUP BY
    customer_id;
```

	customer_id smallint
1	184
2	87
3	477
4	273
5	550
6	51
7	394
8	272
9	70

Figure 5.16 GROUP BY without aggregation

5.5.2 Using PostgreSQL GROUP BY with SUM() function example

The GROUP BY clause is useful when it is used in conjunction with an aggregate function.

For example, to select the total amount that each customer has been paid, you use the GROUP BY clause to divide the rows in the payment table into groups grouped by customer id. For each group, you calculate the total amounts using the SUM() function.

The following query uses the GROUP BY clause to get total amount that each customer has been paid:

```
SELECT
    customer_id,
    SUM (amount)
FROM
```

payment

GROUP BY

customer_id;

	customer_id smallint	sum numeric
1	184	80.80
2	87	137.72
3	477	106.79
4	273	130.72
5	550	151.69
6	51	123.70
7	394	77.80
8	272	65.87
9	70	75.83
10	190	102.75
11	350	63.79

Figure 5.17 GROUP BY with aggregation

5.6 Common Table Expressions

A common table expression is a temporary result set which you can reference within another SQL statement including SELECT, INSERT, UPDATE or DELETE.

```
WITH cte_name (column_list) AS (  
    CTE_query_definition  
)
```

statement;

Example:

*with a1 (name, pages) as (select * from book) select max(pages) from a1;*

The advantages of Common Table Expressions are:

The following are some advantages of using common table expressions or CTEs:

- Improve the readability of complex queries.

- Ability to create recursive queries. Recursive queries are queries that reference themselves. The recursive queries come in handy when you want to query hierarchical data such as organization chart or bill of materials.
- Use in conjunction with window functions. You can use CTEs in conjunction with window functions to create an initial result set and use another select statement to further process this result set.

WITH provides a way to write auxiliary statements for use in a larger query. These statements, which are often referred to as Common Table Expressions or CTEs, can be thought of as defining temporary tables that exist just for one query. Each auxiliary statement in a WITH clause can be a SELECT, INSERT, UPDATE, or DELETE; and the WITH clause itself is attached to a primary statement that can also be a SELECT, INSERT, UPDATE, or DELETE.

5.6.1 Recursive Queries

The optional RECURSIVE modifier changes WITH from a mere syntactic convenience into a feature that accomplishes things not otherwise possible in standard SQL. Using RECURSIVE, a WITH query can refer to its own output. A very simple example is this query to sum the integers from 1 through 100:

```
WITH RECURSIVE t(n) AS (
    VALUES (1)
    UNION ALL
    SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

The general form of a recursive WITH query is always a non-recursive term, then UNION (or UNION ALL), then a recursive term, where only the recursive term can contain a reference to the query's own output. Such a query is executed as follows:

1. Evaluate the non-recursive term. For UNION (but not UNION ALL), discard duplicate rows. Include all remaining rows in the result of the recursive query, and also place them in a temporary working table.
2. So long as the working table is not empty, repeat these steps:

- a. Evaluate the recursive term, substituting the current contents of the working table for the recursive self-reference. For UNION (but not UNION ALL), discard duplicate rows and rows that duplicate any previous result row. Include all remaining rows in the result of the recursive query, and also place them in a temporary intermediate table.
- b. Replace the contents of the working table with the contents of the intermediate table, then empty the intermediate table.

Note. While RECURSIVE allows queries to be specified recursively, internally such queries are evaluated iteratively.

5.6.2 PostgreSQL recursive queries example (Factorial)

Factorial is a non-negative integer. It is the product of all positive integers less than or equal to that number you ask for factorial. It is denoted by an exclamation sign (!).

Example:

$$n! = n * (n-1) * (n-2) * \dots * 1$$

$$4! = 4 \times 3 \times 2 \times 1 = 24$$

The factorial value of 4 is 24.

The recursive Select allows to calculate Factorial values:

```
WITH RECURSIVE t(n,m) AS (
  VALUES (1,1)
  UNION ALL
  SELECT n+1, m*(n+1) FROM t WHERE n < 7
)
SELECT n,m FROM t;
```

Data Output		Messages	
	n integer	m integer	
1	1	1	
2	2	2	
3	3	6	
4	4	24	
5	5	120	
6	6	720	
7	7	5040	

Figure 5.18 Recursive Queries

Recursive queries are typically used to deal with hierarchical or tree-structured data. A useful example is this query to find all paths in a tree:

5.6.3 A Tree example

Let’s consider a tree of the following structure:

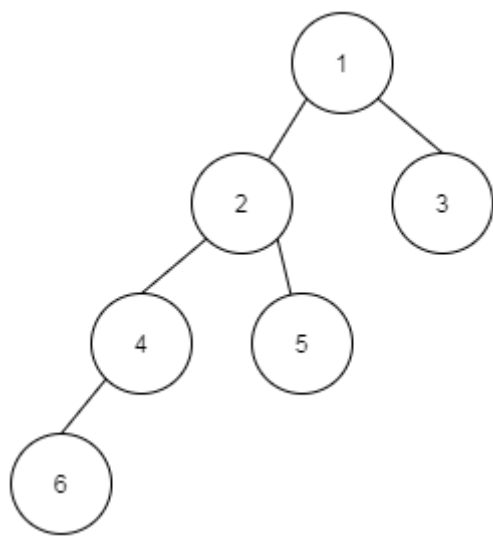


Figure 5.19 The tree example

It can be represented as the following table:

	id [PK] integer	pid integer	name character varying
1	1	[null]	1
2	2	1	2
3	3	1	3
4	4	2	4
5	5	2	5
6	6	4	6

Figure 5.20 Tree as a table

To find all paths to the root as array we can create the query:

```
WITH RECURSIVE test AS (  
  SELECT id, ARRAY[]::INTEGER[] AS ancestors  
  FROM tree WHERE pid IS NULL  
  UNION ALL
```

```

SELECT tree.id, test.ancestors || tree.pid
FROM test, tree
WHERE tree.pid = test.id
)
SELECT * FROM test WHERE 1 = ANY(test.ancestors);

```

The result is here:

	id integer	ancestors integer[]
1	2	{1}
2	3	{1}
3	4	{1,2}
4	5	{1,2}
5	6	{1,2,4}

Figure 5.21 Result of the recursive query

To find the Nodes of the Level 2 (starting from 0) we can perform the next query:

```

WITH RECURSIVE test AS (
  SELECT id, ARRAY[]::INTEGER[] AS ancestors, 0 depth
  FROM tree WHERE pid IS NULL
  UNION ALL
  SELECT tree.id, test.ancestors || tree.pid, depth+1
  FROM test, tree
  WHERE tree.pid = test.id
) SELECT * FROM test WHERE 1 = ANY(test.ancestors) and depth=2;

```

The result is here:

	id integer	ancestors integer[]	depth integer
1	4	{1,2}	2
2	5	{1,2}	2

Figure 5.22 The paths of the length 2

5.7 SQL: Data Manipulation Commands

Data Manipulation Language or DML is a subset of operations used to insert, delete, and update data in a database. A DML is often a sublanguage of a more extensive language like SQL; DML comprises some of the operators in the language. Selecting read-only data is closely related and is sometimes also considered a component of a DML, as some users can perform both read and write selection.

DML represents a collection of programming languages explicitly used to make changes to the database, such as:

- CRUD operations to create, read, update and delete data.
- Using INSERT, SELECT, UPDATE, and DELETE commands.
- DML commands are often part of a more extensive database language, for example, SQL (structured query language). These DMLs can have a specific syntax to handle data in that language.

DML has two main classifications which are procedural and non-procedural programming, which is also called declarative programming. The SQL dealing with the manipulation of data present in the database belongs to the DML or Data Manipulation Language, including most of the SQL statements.

5.7.1 PostgreSQL INSERT statement

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 (column1, column2,).

Second, supply a list of comma-separated values in a parenthesis (value1, value2, ...) after the VALUES keyword. The columns and values in the column and value lists must be in the same order.

RETURNING clause

The INSERT statement also has an optional RETURNING clause that returns the information of the inserted row.

If you want to return the entire inserted row, you use an asterisk (*) after the RETURNING keyword:

```
INSERT INTO table_name(column1, column2, ...)
VALUES (value1, value2, ...)
RETURNING *;
```

If you want to return just some information of the inserted row, you can specify one or more columns after the RETURNING clause.

For example, the following statement returns the id of the inserted row:

```
INSERT INTO table_name(column1, column2, ...)
VALUES (value1, value2, ...)
RETURNING id;
```

5.7.2 INSERT statement examples

The following statement creates a new table called *users* for the demonstration:

```
DROP TABLE IF EXISTS users;
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    description VARCHAR (255),
);
```

INSERT – Inserting a single row into a table

The following statement inserts a new row into the users table:

```
INSERT INTO users (name)
```

```
VALUES('User1',Description for a user');
```

The statement returns the following output:

```
INSERT 0 1
```

To insert character data, you enclose it in single quotes (') for example 'PostgreSQL Tutorial'.

If you omit required columns in the INSERT statement, PostgreSQL will issue an error. In case you omit an optional column, PostgreSQL will use the column default value for insert.

In this example, the description is an optional column because it doesn't have a NOT NULL constraint. Therefore, PostgreSQL uses NULL to insert into the description column.

PostgreSQL automatically generates a sequential number for the serial column so you do not have to supply a value for the serial column in the INSERT statement.

5.8 UPDATE statement

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.

When the UPDATE statement is executed successfully, it returns the following command tag:

UPDATE count

The count is the number of rows updated including rows whose values did not change.

Returning updated rows

The UPDATE statement has an optional RETURNING clause that returns the updated rows:

UPDATE table_name

SET column1 = value1,

column2 = value2,

...

WHERE condition

RETURNING * | output_expression AS output_name;

Code language: SQL (Structured Query Language) (sql)

5.8.1 UPDATE examples

Let's update the rows from the 'users' table:

UPDATE users

SET description = 'a new description'

WHERE id = 1;

It modifies the description field of a user having id=1.

5.9 DELETE statement

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.

The DELETE statement returns the number of rows deleted. It returns zero if the DELETE statement did not delete any row.

To return the deleted row(s) to the client, you use the RETURNING clause as follows:

```
DELETE FROM table_name
```

```
WHERE condition
```

```
RETURNING (select_list | *)
```

The asterisk (*) allows you to return all columns of the deleted row from the table_name.

To return specific columns, you specify them after the RETURNING keyword.

Note that the DELETE statement only removes data from a table. It doesn't modify the structure of the table. If you want to change the structure of a table such as removing a column, you should use the ALTER TABLE statement.

DELETE statement examples

Let's delete all the rows from the 'users' table:

```
DELETE FROM users;
```

If we going to delete a part of rows, we provide a WHERE condition for that purpose:

```
DELETE FROM users WHERE id=1;
```

6 PostgreSQL Schema

A database contains one or more named schemas, which in turn contain tables. Schemas also contain other kinds of named objects, including data types, functions, and operators. The same object name can be used in different schemas without conflict; for example, both `schema1` and `myschema` can contain tables named `mytable`. Unlike databases, schemas are not rigidly separated: a user can access objects in any of the schemas in the database they are connected to, if they have privileges to do so.

There are several reasons why one might want to use schemas:

- To allow many users to use one database without interfering with each other.
- To organize database objects into logical groups to make them more manageable.
- Third-party applications can be put into separate schemas so they do not collide with the names of other objects.

Schemas are analogous to directories at the operating system level, except that schemas cannot be nested.

6.1 Creating a Schema

To create a schema, use the `CREATE SCHEMA` command. Give the schema a name of your choice. For example:

```
CREATE SCHEMA myschema;
```

To create or access objects in a schema, write a qualified name consisting of the schema name and table name separated by a dot:

schema.table

This works anywhere a table name is expected, including the table modification commands and the data access commands discussed in the following chapters. (For brevity we will speak of tables only, but the same ideas apply to other kinds of named objects, such as types and functions.)

Actually, the even more general syntax

database.schema.table

can be used too, but at present this is just for pro forma compliance with the SQL standard. If you write a database name, it must be the same as the database you are connected to.

So, to create a table in the new schema, use:

```
CREATE TABLE myschema.mytable (  
  
...  
);
```

To drop a schema if it's empty (all objects in it have been dropped), use:

```
DROP SCHEMA myschema;
```

To drop a schema including all contained objects, use:

```
DROP SCHEMA myschema CASCADE;
```

The Public Schema

In the previous sections we created tables without specifying any schema names. By default such tables (and other objects) are automatically put into a schema named “public”. Every new database contains such a schema. Thus, the following are equivalent:

```
CREATE TABLE products ( ... );
```

and:

```
CREATE TABLE public.products ( ... );
```

6.2 PostgreSQL Schema Objects

The Schema is a namespace which provides several objects such as tables, views, indexes, data types, functions, stored procedures and operators.

- Views/Materialized Views (alternative data view)
- PL/pgSQL (user-defined functions, procedures)
- Triggers (DML and Event Triggers are event handlers)
- Indexes (speed up query execution)

6.3 PostgreSQL Views

A view is a named query that provides another way to present data in the database tables. A view is defined based on one or more tables which are known as base tables. When you create a view, you basically create a query and assign a name to the query. Therefore, a view is useful for wrapping a commonly used complex query.

Note that regular views do not store any data except the materialized views. In PostgreSQL, you can create special views called materialized views that store data physically and periodically refresh data from the base tables. The materialized views are handy in many scenarios, such as faster data access to a remote server and caching.

A view can be very useful in some cases such as:

- A view helps simplify the complexity of a query because you can query a view, which is based on a complex query, using a simple SELECT statement.
- Like a table, you can grant permission to users through a view that contains specific data that the users are authorized to see.
- A view provides a consistent layer even the columns of underlying table changes.

6.3.1 Views examples

create view factorial_view as

```
WITH RECURSIVE t(n,m) AS (
    VALUES (1,1)
    UNION ALL
    SELECT n+1, m*(n+1) FROM t WHERE n < 7
)
SELECT n,m FROM t;
```


1	<code>select * from factorial_view</code>
2	<code>where n<5</code>

Data Output	Explain	Messages	Notificatio
	n integer	m integer	
1	1	1	
2	2	2	
3	3	6	
4	4	24	

Figure 6.1 The usage of a view

6.3.2 View example: access restriction

In order to illustrate access restriction of a view we need to do the following.

Create a sensitive table:

```
create table private_data(id serial primary key, name varchar(30), salary money);
```

Insert some public & private data:

```
insert into private_data(name, salary) values ('john', 10000), ('paul', 15000);
```

Create a public oriented view:

```
create view public_data as select name from private_data;
```

Create a user with restricted access:

```
create role restricted_user with noinherit login password 'pwd';
```

Grant a permission to access the view only:

```
GRANT SELECT ON public_data TO restricted_user;
```

Assume the “restricted_user” signed in.

1	<code>select * from private_data</code>
---	---

Data Output	Explain	Messages	Notifications
ERROR:	no access to the relation	private_data	

Figure 6.2 User without needed permissions

On the other hand, if a user make a query to the `public_data`, he will get:

```
1 select * from public_data
```

Data Output	Explain	Messages	Notificat				
<table><tr><th></th><th>name</th></tr><tr><td></td><td>character varying (30)</td></tr></table>		name		character varying (30)			
	name						
	character varying (30)						
1	john						
2	paul						

Figure 6.3 User with needed permissions

6.3.3 Updatable & Temporary Views

A PostgreSQL view is updatable when it meets the following conditions:

The defining query of the view must have exactly one entry in the FROM clause, which can be a table or another updatable view.

The defining query must not contain one of the following clauses at the top level: GROUP BY, HAVING, LIMIT, OFFSET, DISTINCT, WITH, UNION, INTERSECT, and EXCEPT.

The selection list must not contain any window function , any set-returning function, or any aggregate function such as SUM, COUNT, AVG, MIN, and MAX.

Temporary views are automatically dropped at the end of the current session. If any of the tables referenced by the view are temporary, the view is created as a temporary view. (create temp view ...)

6.3.4 Materialized Views

PostgreSQL materialized views that allow to store result of a query physically and update the data periodically.

```
CREATE MATERIALIZED VIEW view_name
```

```
AS
```

```
query
```

WITH [NO] DATA;

REFRESH MATERIALIZED VIEW view_name;

PostgreSQL locks the entire table therefore you cannot query data against it. To avoid this, you can use the CONCURRENTLY option.

MATERIALIZED VIEW provides some additional features that VIEW lacks:

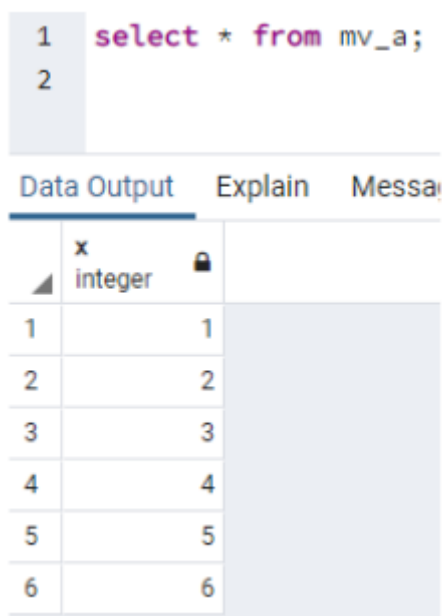
- providing a consistent snapshot of data for users to work with
- giving users the ability to index the underlying snapshot.

6.3.5 Materialized Views Example

Assume we have a table 'a' containing just only one field and would like to create a materialized view for it:

```
create materialized view mv_a as select * from a ;
```

```
select * from mv_a; -- the same as table 'a'
```



The screenshot shows a SQL query editor with the following text:

```
1 select * from mv_a;  
2
```

Below the editor, there are three tabs: "Data Output", "Explain", and "Messages". The "Data Output" tab is selected, displaying a table with the following data:

	x integer
1	1
2	2
3	3
4	4
5	5
6	6

Figure 6.4 A materialized view before INSERT a value '7' into the table

```
insert into a values (7) -- (1)
```

```
refresh materialized view mv_a;-- (2)
```

```
select * from mv_a; -- the same as table 'a'
```

1	<code>select * from mv_a;</code>		
2			
	Data Output	Explain	Message
	<div><div></div><div>x integer</div><div></div></div>		
1	1		
2	2		
3	3		
4	4		
5	5		
6	6		
7	7		

Figure 6.5 A materialized view after INSERT a value '7' and REFRESH into the table

6.4 Triggers

A PostgreSQL trigger is a function invoked automatically whenever an event associated with a table occurs. An event could be any of the following: INSERT, UPDATE, DELETE or TRUNCATE.

A trigger is a special user-defined function associated with a table. To create a new trigger:

- define a trigger function first,
- bind this trigger function to a table.

The difference between a trigger and a user-defined function is that a trigger is automatically invoked when a triggering event occurs.

Main types of triggers:

- Row-level (on every row affected);
- Statement-level triggers (just only once per statement).

Time when the trigger is invoked:

- before event (skip the operation for the current row or even change the row being updated or inserted);
- after an event (all changes are available to the trigger);
- instead of.

Triggers in PostgreSQL has some specific features:

- PostgreSQL fires trigger for the TRUNCATE event.
- PostgreSQL allows you to define the statement-level trigger on views.
- PostgreSQL requires you to define a user-defined function as the action of the trigger, while the SQL standard allows you to use any SQL commands.

Triggers' use cases:

- to maintain complex data integrity rules;
- perform additional actions when data being changed (custom log, versioning)

Syntax of Triggers:

```
CREATE FUNCTION trigger_function()
```

```
RETURNS TRIGGER
```

```
LANGUAGE PLPGSQL
```

```
AS $$
```

```
BEGIN
```

```
-- trigger logic
```

```
END; $$
```

A trigger function receives data about its calling environment:

When a PL/pgSQL function is called as a trigger, several special variables are created automatically in the top-level block. They are:

NEW

Data type RECORD; variable holding the new database row for INSERT/UPDATE operations in row-level triggers. This variable is null in statement-level triggers and for DELETE operations.

OLD

Data type RECORD; variable holding the old database row for UPDATE/DELETE operations in row-level triggers. This variable is null in statement-level triggers and for INSERT operations.

TG_NAME

Data type name; variable that contains the name of the trigger actually fired.

TG_WHEN

Data type text; a string of BEFORE, AFTER, or INSTEAD OF, depending on the trigger's definition.

TG_LEVEL

Data type text; a string of either ROW or STATEMENT depending on the trigger's definition.

TG_OP

Data type text; a string of INSERT, UPDATE, DELETE, or TRUNCATE telling for which operation the trigger was fired.

TG_RELNAME

Data type name; the name of the table that caused the trigger invocation. This is now deprecated, and could disappear in a future release. Use TG_TABLE_NAME instead.

TG_TABLE_NAME

Data type name; the name of the table that caused the trigger invocation.

TG_TABLE_SCHEMA

Data type name; the name of the schema of the table that caused the trigger invocation.

CREATE TRIGGER Syntax

```
CREATE TRIGGER trigger_name
  {BEFORE | AFTER | INSTEAD OF} { event }
  ON table_name
  [FOR [EACH] { ROW | STATEMENT }]
  EXECUTE PROCEDURE trigger_function
```

CREATE TRIGGER Example

```
CREATE OR REPLACE FUNCTION Mul1000()
  RETURNS trigger AS
```

```

$mycode$
BEGIN
  IF OLD.id = '100' THEN
    NEW.ID := NEW.ID || '1000';
  END IF;
  RETURN NEW;
END;
$mycode$ LANGUAGE plpgsql;

drop trigger
drop trigger if exists last_name_changes on test;

```

```

CREATE TRIGGER last_name_changes
  BEFORE UPDATE ON test
  FOR EACH ROW
  EXECUTE PROCEDURE Mul1000();
update test set id='101' where id='100';
select * from test;

```

6.5 Indexing

In PostgreSQL, Indexes is the special tool used to enhance the retrieval of data from the databases.

A database index is parallel to the index of a book. An index creates an access for all the values, which displays on the indexed columns.

The indexes tend to help the database server to identify the defined rows much faster than it could do without indexes. We have to use the Indexes properly to get the significant result.

Features of PostgreSQL indexes

Some of the essential features of the PostgreSQL indexes are as follows:

- An index is used to enhance the data output with SELECT and WHERE
- If we are using the INSERT and UPDATE commands, it slows down data input.
- Without affecting any of the data, we can CREATE and DROP the
- We can generate an index with the CREATE INDEX command's help by defining the index name and table or column name on which the index is created.
- We can also create a unique index, which is similar to the UNIQUE constraint.

6.5.1 Types of PostgreSQL Indexes

All the index type uses various algorithm and storage structure to manage different types of commands.

In PostgreSQL, the indexes can be categorized into various parts, which are as follows:

Hash indexes

When an indexed column is included in the table and compared to the equal (=) operator, the Hash indexes can cope only with simple equality comparison (=) operator.

B-tree indexes

The most important used indexes in PostgreSQL is B-tree indexes.

The B-Tree index is a balance tree, which keeps the sorted data and permits the insertions, searches, deletions, and sequential access in logarithmic time.

The PostgreSQL developer will consider using a B-tree index when index columns are included in an assessment, which uses one of the below operators list:

<

<=

=

>=

BETWEEN

IN

IS NULL

IS NOT NULL

Furthermore, for the pattern matching operator LIKE and ~ commands, the query developer can use a B-tree index.

GIN indexes

The next type of PostgreSQL indexes is GIN, which stands for Generalized Inverted Indexes, and it is usually denoted as GIN.

If we have several values stored in a single column such as range type, array, jsonb, and hstore, the GIN indexes are most beneficial.

GiST Indexes

The GiST indexes are most commonly used for indexing in full-text search and geometric data types.

The Generalized Search Tree denotes GiST indexes, which provides a building of general tree structures.

SP-GiST indexes

The Space-Partitioned GiST is denoted as SP- GiST that keeps up partitioned search trees, which enable the development of an extensive range of dissimilar non-balanced data structures.

The data which contains a natural clustering element is also not an equally balanced tree, like, multimedia, GIS, IP routing, phone routing, and IP routing, in such cases, we can use the SP-GiST

BRIN

The BRIN indexes can be maintained easily as it is less costly and much smaller as compared to the B-tree index, and it stands for Block Range Indexes.

Regularly, the BRIN indexes are used on a column, which contains a linear sort order, such as the generated date column of the sales order

In PostgreSQL indexes, the BRIN allows the use of an index on a huge table, which would earlier be unusable with B-tree without parallel partitioning.

Disadvantages of using the PostgreSQL Indexes

We have the following reasons for avoiding the PostgreSQL Indexes:

- The PostgreSQL Indexes should not be used on columns, which include a large number of NULL values.
- The PostgreSQL indexes cannot be used with the small tables.
- We do not create indexes for columns, which are often deployed.
- We do not create indexes for tables, with frequent, large batch update or insert operations.

6.5.2 PostgreSQL Create Index

The syntax of creating an Indexes command is as follows:

```
CREATE INDEX index_name ON table_name [USING method]
(
    column_name [ASC | DESC] [NULLS {FIRST | LAST}],
    ...
);
```

In the above syntax, we have used the following parameters, as shown in the below table:

Table 6.1 Create index command

Parameters	Description
Index_name	It is used to define the name of the index. And it should be written after the CREATE INDEX Here, we should try to give the easier and significant name of the index, which can be easily recalled.
Table_name	The table_name parameter is used to define the table name, which is linked with the Indexes. And it is specified after the ON keyword.

Using[method]	<p>It is used to specify the index methods, such as B-tree, GIN, HASH, GiST, BRIN, and SP-GiST.</p> <p>By default, PostgreSQL uses B-tree Index.</p>
Column_name	<p>The column_name parameter is used to define the list if we have several columns stored in the index.</p> <p>The ASC and DESC are used to define the sort order. And by default, it is ASC.</p> <p>The NULLS FIRST or NULLS LAST is used to describe the nulls sort before or after non-null values.</p> <p>When DESC is defined, then the NULLS FIRST is considered as the default.</p> <p>And when DESC is not defined, then NULLS LAST is considered as default.</p>

6.5.3 Disadvantages of using the PostgreSQL Indexes

We have the following reasons for avoiding the PostgreSQL Indexes:

The PostgreSQL Indexes should not be used on columns, which include a large number of NULL values.

The PostgreSQL indexes cannot be used with the small tables.

We do not create indexes for columns, which are often deployed.

We do not create indexes for tables, with frequent, large batch update or insert operations.

7 PL/pgSQL procedural language

PL/pgSQL procedural language adds many procedural elements, e.g., control structures, loops, and complex computations, to extend standard SQL. It allows you to develop complex functions and stored procedures in PostgreSQL that may not be possible using plain SQL.

PL/pgSQL procedural language is similar to the Oracle PL/SQL. The following are reasons to learn PL/pgSQL:

- PL/pgSQL is easy to learn and simple to use.
- PL/pgSQL comes with PostgreSQL by default. The user-defined functions and stored procedures developed in PL/pgSQL can be used like any built-in functions and stored procedures.
- PL/pgSQL inherits all user-defined types, functions, and operators.
- PL/pgSQL has many features that allow you to develop complex functions and stored procedures.
- PL/pgSQL can be defined to be trusted by the PostgreSQL database server.

PL/pgSQL allows you to extend the functionality of the PostgreSQL database server by creating server objects with complex logic.

PL/pgSQL was designed to:

Create user-defined functions, stored procedures, and triggers.

Extend standard SQL by adding control structures such as if, case, and loop statements.

Inherit all user-defined functions, operators, and types.

7.1 PL/pgSQL Block Structure

PL/pgSQL is a block-structured language, therefore, a PL/pgSQL function or stored procedure is organized into blocks.

The following illustrates the syntax of a complete block in PL/pgSQL:

```
[ <<label>> ]
```

```
[ declare
```

```
    declarations ]
```

```
begin
```

```
    statements;
```

```
    ...
```

```
end [ label ];
```

Let's examine the block structure in more detail:

- Each block has two sections: declaration and body. The declaration section is optional while the body section is required. A block is ended with a semicolon (;) after the END keyword.

- A block may have an optional label located at the beginning and at the end. You use the block label when you want to specify it in the EXIT statement of the block body or when you want to qualify the names of variables declared in the block.
- The declaration section is where you declare all variables used within the body section. Each statement in the declaration section is terminated with a semicolon (;).
- The body section is where you place the code. Each statement in the body section is also terminated with a semicolon (;).

Supported Argument and Result Data Types

PL/pgSQL allows:

- accept as arguments any scalar or array data type supported by the server, and they can return a result of any of these types;
- return a “set” (or table) of any data type. Such a function generates its output by executing RETURN NEXT for each desired element of the result set, or by using RETURN QUERY to output the result of evaluating a query;
- accept or return any composite type (row type);
- RETURNS TABLE notation can also be used in place of RETURNS SETOF in order to return set of rows.

Examples of Variables

num1 integer;

val3 numeric(5);

title varchar;

myrow tablename%ROWTYPE;

myfield tablename.columnname%TYPE;

arow RECORD;

qty integer DEFAULT 1;

roll_no CONSTANT integer := 10;

url varchar := 'http://example.com';

7.2 Basic examples of functions

The following function 'coeff' gets a real variable and multiplies it by 0.06:

```
CREATE FUNCTION coeff(real) RETURNS real AS $a1$
```

```
DECLARE
```

```
    subtotal ALIAS FOR $1;
```

```
BEGIN
```

```
    RETURN subtotal * 0.06;
```

```
END;
```

```
$a1$ LANGUAGE plpgsql;
```

```
====
```

```
CREATE FUNCTION TODAY_IS () RETURNS CHAR(22) AS '
```

```
BEGIN
```

```
    RETURN "Today""is " || CAST(CURRENT_DATE AS CHAR(10));
```

```
END;
```

```
,
```

```
LANGUAGE PLPGSQL
```

The next functions show how to work with date and how to pass variables IN and OUT:

```
CREATE FUNCTION FUN_TO_TEST(dt DATE, ing INTEGER)
```

```
RETURNS DATE AS $test$
```

```
DECLARE ss  ALIAS FOR dt;
```

```
    ff  ALIAS FOR ing;
```

```
BEGIN
```

```
    RETURN ss + ff * INTERVAL '2 DAY';
```

```
END;
```

```
$test$
```

```
LANGUAGE PLPGSQL
```

```
CREATE FUNCTION sum_n_product(x int, y int, OUT sum int, OUT prod int) AS $$
BEGIN
    sum := x + y;
    prod := x * y;
END;
$$ LANGUAGE plpgsql;
```

The next examples show EXCEPTION Handling syntax of the language:

```
BEGIN
    SELECT * INTO STRICT myrec FROM emp WHERE empname = myname;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE EXCEPTION '% not found', myname;
    WHEN TOO_MANY_ROWS THEN
        RAISE EXCEPTION '% already exists', myname;
END;
```

8 SQL EXPLAIN

The EXPLAIN statement returns the execution plan which PostgreSQL planner generates for a given statement.

The EXPLAIN shows how tables involved in a statement will be scanned by index scan or sequential scan, etc., and if multiple tables are used, what kind of join algorithm will be used.

The most important and useful information that the EXPLAIN statement returns are start-cost before the first row can be returned and the total cost to return the complete result set.

The syntax of the EXPLAIN command looks as the following:

```
EXPLAIN [ ( option [, ...] ) ] sql_statement;
```

where option can be one of the following:

ANALYZE [boolean]

VERBOSE [boolean]

COSTS [boolean]

BUFFERS [boolean]

TIMING [boolean]

SUMMARY [boolean]

FORMAT { TEXT | XML | JSON | YAML }

The ANALYZE statement actually executes the SQL statement and discards the output information, therefore, if you want to analyze any statement such as INSERT, UPDATE, or DELETE without affecting the data, you should wrap the EXPLAIN ANALYZE in a transaction, as follows:

```
BEGIN;
```

```
    EXPLAIN ANALYZE sql_statement;
```

```
ROLLBACK;
```

VERBOSE

The VERBOSE parameter allows you to show additional information regarding the plan. This parameter sets to FALSE by default.

COSTS

The COSTS option includes the estimated startup and total costs of each plan node, as well as the estimated number of rows and the estimated width of each row in the query plan. The COSTS defaults to TRUE.

BUFFERS

This parameter adds information to the buffer usage. BUFFERS only can be used when ANALYZE is enabled. By default, the BUFFERS parameter set to FALSE.

TIMING

This parameter includes the actual startup time and time spent in each node in the output. The TIMING defaults to TRUE and it may only be used when ANALYZE is enabled.

SUMMARY

The SUMMARY parameter adds summary information such as total timing after the query plan. Note that when ANALYZE option is used, the summary information is included by default.

FORMAT

Specify the output format of the query plan such as TEXT, XML, JSON, and YAML. This parameter is set to TEXT by default.

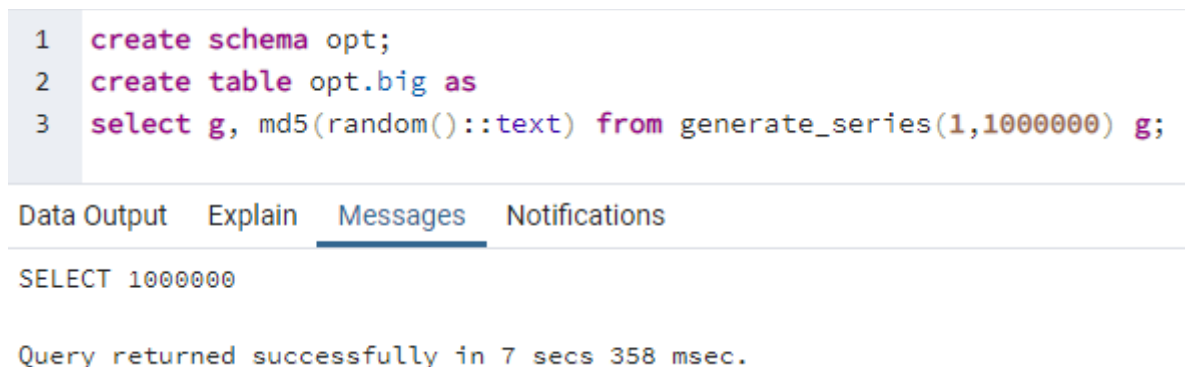
8.1 EXPLAIN select example

Let's create a table filled out with random values:

create schema opt;

create table opt.big as

select g, md5(random()::text) from generate_series(1,1000000) g;



```
1 create schema opt;
2 create table opt.big as
3 select g, md5(random()::text) from generate_series(1,1000000) g;
```

Data Output Explain Messages Notifications

SELECT 10000000

Query returned successfully in 7 secs 358 msec.

Figure 8.1 The table initialization

Let's select some data:

```
1 select * from opt.big limit 5
```

Data Output Explain Messages Notifications

	g integer	md5 text
1		d3b0fb2368dfb454252eb8c4702402bf
2		cedf1995e811c8b860e8ec07be6295ab
3		87bd8f475535fe8c988626063624ffbf
4		68f654fbca648a6eabc1a66c274b2edd
5		431210fe92fc290d3456054ed1716b0c

Figure 8.2 Top 5 rows from the BIG table

Let's apply the EXPLAIN Command:

```
1 explain select * from opt.big limit 5
```

Data Output Explain Messages Notifications

	QUERY PLAN text
1	Limit (cost=0.00..0.09 rows=5 width=37)
2	-> Seq Scan on big (cost=0.00..18334.00 rows=1000000 width=37)

Figure 8.3 Explain command results (Limiting 5 rows)

```
explain select * from opt.big
```

Data Output Explain Messages Notifications

	QUERY PLAN text
	Seq Scan on big (cost=0.00..18334.00 rows=1000000 width=37)

Figure 8.4 Explain command results (All rows)

```
1 explain analyze select * from opt.big |
```

Data Output Explain Messages Notifications

QUERY PLAN		
	text	
1	Seq Scan on big (cost=0.00..18334.00 rows=1000000 width=37) (actual time=0.029..318.206 rows=1000000 loops=1)	
2	Planning Time: 0.089 ms	
3	Execution Time: 371.713 ms	

Figure 8.5 EXPLAIN ANALYZE = real execution

How to read the EXPLAIN Command results:

Estimated start-up cost. This is the time expended before the output phase can begin, e.g., time to do the sorting in a sort node.

Estimated total cost. This is stated on the assumption that the plan node is run to completion, i.e., all available rows are retrieved. In practice a node's parent node might stop short of reading all available rows (see the LIMIT example above).

Estimated number of rows output by this plan node. Again, the node is assumed to be run to completion.

Estimated average width of rows output by this plan node (in bytes).

8.1.1 EXPLAIN Estimations

Rows

number of rows: 1 000 000

```
1 SELECT relname, relpages, reltuples
2 FROM pg_class WHERE relname='big';
```

Data Output Explain Messages Notifications

	relname name	relpages integer	reltuples real
1	big	8334	1e+06

Figure 8.6 RelPages and RelTuples

Width

average width of a row in bytes: 36 (~37)

```
SELECT attname, avg_width
FROM pg_stats
WHERE tablename='big' and schemaname='opt';
```

ta	Output	Explain	Messages	Notifications
attname	avg_width			
name	integer			
g	4			
md5	33			

Figure 8.7 Width in bytes of a row in the table

Explain Cost

Cost:

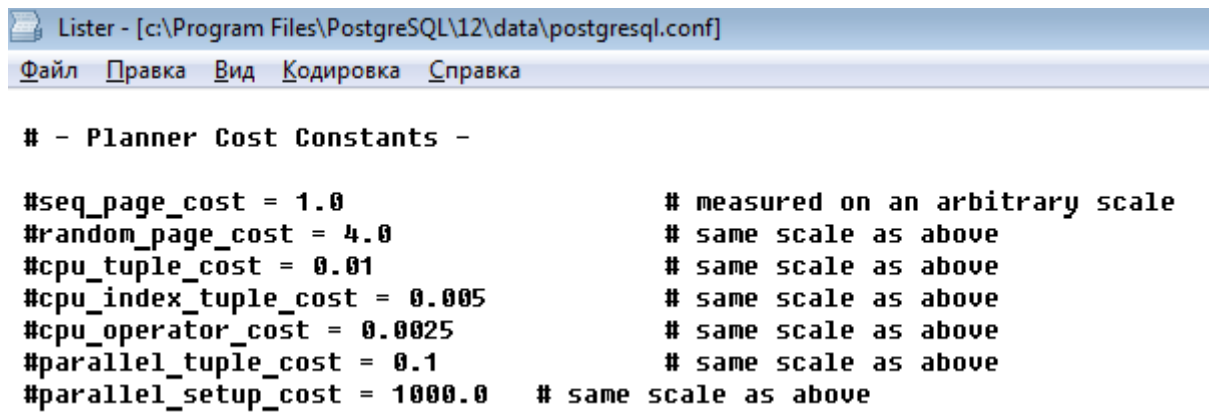
```
SELECT relpages*current_setting('seq_page_cost')::float4
+ reltuples*current_setting('cpu_tuple_cost')::float4
AS total_cost, reltuples,
current_setting('seq_page_cost') as seq_page_cost,
current_setting('cpu_tuple_cost') as cpu_tuple_cost
FROM pg_class
WHERE relname='big';
```

ta	Output	Explain	Messages	Notifications
total_cost	reltuples	seq_page_cost	cpu_tuple_cost	
double precision	real	text	text	
18334	1e+06	1	0.01	

cost to get the first row: 0.00

cost to get all rows: 18334.00

The configuration of the PostgreSQL server allows to change some of the parameters in the postgresql.conf file:



```
# - Planner Cost Constants -

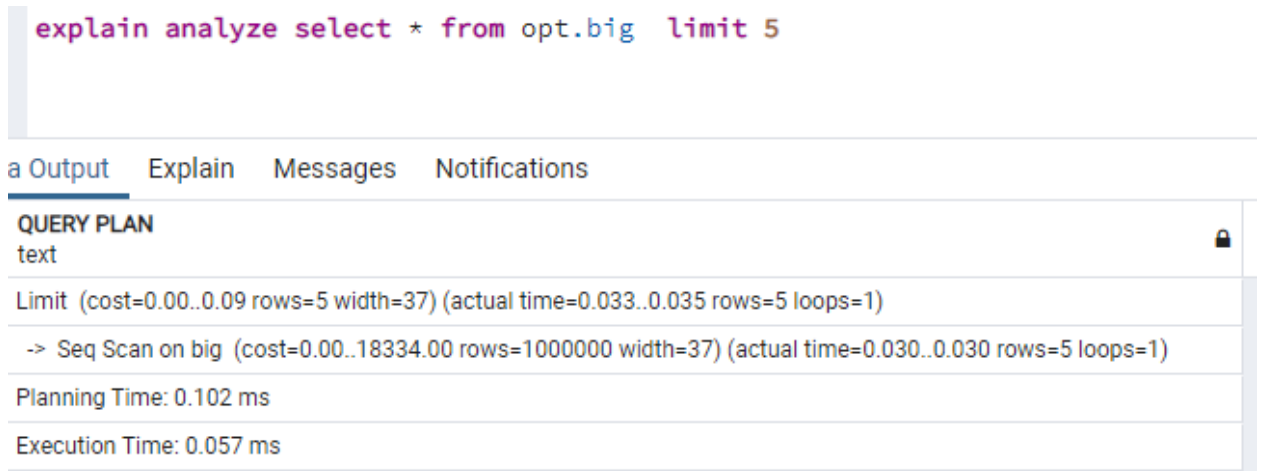
#seq_page_cost = 1.0           # measured on an arbitrary scale
#random_page_cost = 4.0        # same scale as above
#cpu_tuple_cost = 0.01         # same scale as above
#cpu_index_tuple_cost = 0.005  # same scale as above
#cpu_operator_cost = 0.0025    # same scale as above
#parallel_tuple_cost = 0.1      # same scale as above
#parallel_setup_cost = 1000.0   # same scale as above
```

Figure 8.8 The configuration file

seq_page_cost - to read one sequential page

cpu_tuple_cost - to check every row

Explain analyze in details



```
explain analyze select * from opt.big limit 5
```

a Output Explain Messages Notifications

QUERY PLAN

text

Limit (cost=0.00..0.09 rows=5 width=37) (actual time=0.033..0.035 rows=5 loops=1)

-> Seq Scan on big (cost=0.00..18334.00 rows=1000000 width=37) (actual time=0.030..0.030 rows=5 loops=1)

Planning Time: 0.102 ms

Execution Time: 0.057 ms

Figure 8.9 Explain analyze in details

Extra information:

real execution time in milliseconds = “actual time”

real number of lines = “rows”

number of loops = “loops”

Explain analyze – buffers

```
1 explain (analyze, buffers) select * from opt.big
```

Data Output Explain Messages Notifications

QUERY PLAN	
text	
1	Seq Scan on big (cost=0.00..18334.00 rows=1000000 width=37) (actual time=0.078..442.907 rows=1000000 loops=1)
2	Buffers: shared hit=32 read=8302
3	Planning Time: 0.072 ms
4	Execution Time: 511.583 ms

Figure 8.10 Buffers

hit=32 (32*8Kb=256Kb) - taken from the shared_buffer = memory

read=8302 (8302*8Kb=66416Kb) - taken from the disk = disk

After 10+ executions:

```
explain (analyze, buffers) select * from opt.big
```

a Output Explain Messages Notifications

QUERY PLAN	
text	
Seq Scan on big (cost=0.00..18334.00 rows=1000000 width=37) (actual time=0.059..235.104 rows=1000000 loops=1)	
Buffers: shared hit=1056 read=7278	
Planning Time: 0.055 ms	
Execution Time: 281.617 ms	

Figure 8.11 Buffers after 10 executions

EXPLAIN: Select with Where

Cost is: 00..20834

<pre> explain (analyze, buffers) select * from opt.big where g>1000 </pre>			
a	Output	Explain	Messages Notifications
QUERY PLAN text			
Seq Scan on big (cost=0.00..20834.00 rows=998957 width=37) (actual time=0.408..652.594 rows=999000 loops=1)			
Filter: (g > 1000)			
Rows Removed by Filter: 1000			
Buffers: shared hit=64 read=8270			
Planning Time: 0.127 ms			
Execution Time: 736.505 ms			

Figure 8.12 EXPLAIN with WHERE

8.1.2 EXPLAIN and Functional Indexes

We can create an index based on an expression that involves table columns. This index is called an index on expression or functional-based indexes.

Note that indexes on expressions are quite expensive to maintain because PostgreSQL has to evaluate the expression for each row when it is inserted or updated and use the result for indexing.

Example:

create index on opt.big(abs(g));

1	<pre> explain analyze select abs(g) from opt.big where abs(g) < 3000 </pre>		
	Data	Output	Explain Messages Notifications
	QUERY PLAN text		
1	Index Scan using big_abs_idx on big (cost=0.42..116.77 rows=2867 width=4) (actual time=0.028..1.263 rows=2999 loop...		
2	Index Cond: (abs(g) < 3000)		
3	Planning Time: 0.206 ms		
4	Execution Time: 1.428 ms		

Figure 8.13 Explain and functional indexes

8.1.3 EXPLAIN and Partial indexes

PostgreSQL partial index even allows us to specify the rows of a table that should be indexed. This partial index helps:

speed up the query and
reducing the size of the index.
create index on opt.big(g) where g < 500001;

```
explain analyze select * from opt.big where g = 400000
```

ta Output	Explain	Messages	Notifications
QUERY PLAN			
text			
Index Scan using big_g_idx on big (cost=0.42..8.44 rows=1 width=37) (actual time=0.029..)			
Index Cond: (g = 400000)			
Planning Time: 0.161 ms			
Execution Time: 0.056 ms			

Figure 8.14 Partial indexes when $g < 500\,000$ (Index is used)

```
explain analyze select * from opt.big where g = 600000
```

a Output	Explain	Messages	Notifications
QUERY PLAN			
text			
Gather (cost=1000.00..14542.43 rows=1 width=37) (actual time=609.083..640.751 rows=1 loops=1)			
Workers Planned: 2			
Workers Launched: 2			
-> Parallel Seq Scan on big (cost=0.00..13542.33 rows=1 width=37) (actual time=248.952..293.379 rows=0 loops=3)			
Filter: (g = 600000)			
Rows Removed by Filter: 333333			
Planning Time: 0.145 ms			
Execution Time: 640.783 ms			

Figure 8.15 Partial indexes when $g < 500\,000$ (Index is NOT used)

9 Table Scan Modes and Joins

In PostgreSQL it is required to generate a best possible plan which corresponds to the execution of the query with least time and resources.

Currently, PostgreSQL supports below scan methods by which all required data can be read from the table:

- Sequential Scan
- Index Scan
- Index Only Scan
- Bitmap Scan
- TID Scan

Each of these scan methods are equally useful depending on:

- the query itself
- table cardinality (the uniqueness of data values: id has high, gender has low and name has normal cardinality)
- table selectivity (the number of rows with that value, divided by the total number of rows. A lower selectivity value is better)
- disk I/O cost
- random I/O cost
- sequence I/O cost
- CPU cost etc

9.1 Sequential Scan

In order for the sequential scan to be used at-least below criteria should match:

1. No Index available on key, which is part of the predicate (where part).
2. Majority of rows are getting fetched as part of the SQL query.

```
explain (analyze, buffers)
select * from opt.big
```

ta Output	Explain	Messages	Notifications
QUERY PLAN			
text			
Seq Scan on big (cost=0.00..18334.00 rows=1000000 width=37) (actual time=0.069..268.111 rows=1000000 loops=1)			
Buffers: shared hit=3073 read=5261			
Planning Time: 0.105 ms			
Execution Time: 360.890 ms			

Figure 9.1 Seq scan example

9.2 Index Scan

It works well when number of rows selected is small.

```
set enable_indexscan=on;
explain (analyze, buffers)
select * from opt.big where g<100|
```

Output	Explain	Messages	Notifications
QUERY PLAN			
text			
Index Scan using big_g_idx on big (cost=0.42..10.33 rows=109 width=37) (actual time=0.025..0.055 rows=99 loops=1)			
Index Cond: (g < 100)			
Buffers: shared hit=4			
Planning Time: 0.259 ms			
Execution Time: 0.094 ms			

Figure 9.2 Index Scan

So there are two steps for index scan:

1. Fetch data from index related data structure. It returns the TID of corresponding data in heap.
2. Then the corresponding heap page is directly accessed to get whole data. This additional step is required for the below reasons (compare to index only scan):
 - a. Query might have requested to fetch columns more than whatever available in the corresponding index (Random I/O is involved!).
 - b. Visibility information is not maintained along with index data. So in order to check the visibility of data as per isolation level, it needs to access heap data.

9.3 Index Only Scan

Index Only Scan is similar to Index Scan except for the second step i.e. as the name implies it only scans index data structure.

There are two additional pre-condition in order to choose Index Only Scan compare to Index Scan:

Query should be fetching only key columns which are part of the index.

All tuples (records) on the selected heap page should be visible because index data structure does not maintain visibility information so in order to select data

only from index we should avoid checking for visibility and this could happen if all data of that page are considered visible.

```
explain (analyze, buffers)
select g from opt.big where g<100
```

ta Output	Explain	Messages	Notifications
QUERY PLAN			
text			
Index Only Scan using big_g_idx on big (cost=0.42..6.33 rows=109 width=4) (actual time=0.016..0.036 rows=99 loops=1)			
Index Cond: (g < 100)			
Heap Fetches: 0			
Buffers: shared hit=4			
Planning Time: 0.204 ms			
Execution Time: 0.072 ms			

Figure 9.3 Index only scan

9.4 Bitmap Scan

Bitmap scan is a mix of Index Scan and Sequential Scan. It tries to solve the disadvantage of Index scan but still keeps its full advantage: bitmap scan method leverage the benefit of index scan without random I/O.

This works in two levels as below:

Bitmap Index Scan: First it fetches all index data from the index data structure and creates a bit map of all TID: this bitmap contains a hash of all pages (hashed based on page no) and each page entry contains an array of all offset within that page.

Bitmap Heap Scan: it reads through bitmap of pages and then scans data from heap corresponding to stored page and offset. At the end, it checks for visibility

and predicate etc and returns the tuple based on the outcome of all these checks.

```
create table opt.big1 as SELECT random() * 1000 as v FROM generate_series(1, 1000000);
CREATE INDEX ON opt.big1(v);
explain analyze select * from opt.big1 where v < 300
```

Output	Explain	Messages	Notifications
QUERY PLAN			
text			
Bitmap Heap Scan on big1 (cost=5558.64..13690.44 rows=296544 width=8) (actual time=108.736..287.847 rows=300624 loops=1)			
Recheck Cond: (v < '300'::double precision)			
Heap Blocks: exact=4425			
-> Bitmap Index Scan on big1_v_idx (cost=0.00..5484.51 rows=296544 width=0) (actual time=105.675..105.675 rows=300624 loops=1)			
Index Cond: (v < '300'::double precision)			
Planning Time: 0.169 ms			
Execution Time: 302.411 ms			

Figure 9.4 Bitmap Scan: 30% selected

```
--create table opt.big1 as SELECT random() * 1000 as v FROM generate_series(1, 1000000);
--CREATE INDEX ON opt.big1(v);
explain analyze select * from opt.big1 where v < 400
```

a Output	Explain	Messages	Notifications
QUERY PLAN			
text			
Seq Scan on big1 (cost=0.00..16925.00 rows=394962 width=8) (actual time=0.140..311.319 rows=400518 loops=1)			
Filter: (v < '400'::double precision)			
Rows Removed by Filter: 599482			
Planning Time: 0.280 ms			
Execution Time: 330.960 ms			

Figure 9.5 Bitmap Scan (~40%) --> SeqScan

Auxiliary [scan] nodes

Some of the auxiliary nodes generated by the PostgreSQL query optimizer are as below:

- Sort
- Aggregate
- Group By Aggregate
- Limit
- Unique
- LockRows
- SetOp

```
explain analyze select * from opt.big1 order by abs(v)
```

ta Output Explain Messages Notifications

QUERY PLAN

text

Sort (cost=133673.34..136173.34 rows=1000000 width=16) (actual time=2039.230..2585.781 rows=1000000 loops=1)

Sort Key: (abs(v))

Sort Method: external merge Disk: 25480kB

-> Seq Scan on big1 (cost=0.00..16925.00 rows=1000000 width=16) (actual time=0.034..349.947 rows=1000000 loops=1)

Planning Time: 0.138 ms

Execution Time: 2715.434 ms

Figure 9.6 Sort node

```
explain analyze select count(*) from opt.big1
```

ta Output Explain Messages Notifications

QUERY PLAN

text

Finalize Aggregate (cost=10633.55..10633.56 rows=1 width=8) (actual time=266.510..311.557 rows=1 loops=1)

-> Gather (cost=10633.33..10633.54 rows=2 width=8) (actual time=265.650..311.541 rows=3 loops=1)

Workers Planned: 2

Workers Launched: 2

-> Partial Aggregate (cost=9633.33..9633.34 rows=1 width=8) (actual time=108.896..108.897 rows=1 loops=3)

-> Parallel Seq Scan on big1 (cost=0.00..8591.67 rows=416667 width=0) (actual time=0.028..70.207 rows=333333 loops=3)

Planning Time: 0.268 ms

Execution Time: 311.625 ms

Figure 9.7 Aggregate node

```
explain analyze select count(*) from opt.big1 group by v
```

ta Output Explain Messages Notifications

QUERY PLAN

text

GroupAggregate (cost=0.42..58680.36 rows=1000000 width=16) (actual time=0.137..5378.946 rows=1000000 loops=1)

Group Key: v

-> Index Only Scan using big1_v_idx on big1 (cost=0.42..43680.36 rows=1000000 width=8) (actual time=0.119..4158.944 rows=1000000 loops=1)

Heap Fetches: 1000000

Planning Time: 0.254 ms

Execution Time: 5515.230 ms

Figure 9.8 GroupAggregate/HashAggregate node

9.5 Joins implementation

PostgreSQL supports the below kind of joins:

Nested Loop Join ('=', '<', '>'...)

Hash Join (only '=')

Merge Join (only '=')

Each of these Join methods are equally useful depending on the query and other parameters e.g. query, table data, join clause, selectivity, memory etc.

9.5.1 Nested Loop Join

Nested Loop Join (NLJ) is the simplest join algorithm wherein each record of outer relation is matched with each record of inner relation. The Join between relation A and B with condition $A.ID < B.ID$ can be represented as below:

For each tuple r in A

 For each tuple s in B

 If ($r.ID < s.ID$)

 Emit output tuple (r,s)

Nested Loop Join (NLJ) is the most common joining method and it can be used almost on any dataset with any type of join clause. Since this algorithm scan all tuples of inner and outer relation, it is considered to be the most costly join operation:

```
create table opt.tab1 as select (random()*10000)::int as id from
generate_series(1,10000);
```

```
create table opt.tab2 as select (random()*1000)::int as id from
generate_series(1,1000)
```

```
1 explain analyze select * from opt.tab1 t1 inner join opt.tab2 t2 on t1.id > t2.id
```

Data Output	Explain	Messages	Notifications
<div> <div>QUERY PLAN</div> <div>text</div> </div>			
1	Nested Loop (cost=0.00..150162.50 rows=3333333 width=8) (actual time=0.070..3569.310 rows=9471566 loops=1)		
2	Join Filter: (t1.id > t2.id)		
3	Rows Removed by Join Filter: 528434		
4	-> Seq Scan on tab1 t1 (cost=0.00..145.00 rows=10000 width=4) (actual time=0.032..11.550 rows=10000 loops=1)		
5	-> Materialize (cost=0.00..20.00 rows=1000 width=4) (actual time=0.000..0.098 rows=1000 loops=10000)		
6	-> Seq Scan on tab2 t2 (cost=0.00..15.00 rows=1000 width=4) (actual time=0.022..0.199 rows=1000 loops=1)		
7	Planning Time: 0.141 ms		
8	Execution Time: 3947.517 ms		

Figure 9.9 Nested loop

9.5.2 Hash Join

This algorithm works in two phases:

Build Phase: A Hash table is built (in main memory!) using the inner relation records. The hash key is calculated based on the join clause key.

Probe Phase: An outer relation record is hashed based on the join clause key to find matching entry in the hash table.

The join between relation A and B with condition A.ID = B.ID can be represented as below:

Build Phase

For each tuple r in inner relation B

Insert r into hash table HashTab with bucket number hash(r.ID)

Probe Phase

For each tuple s in outer relation A

For each tuple r in bucket number hash(s.ID)

If (s.ID = r.ID)

Emit output tuple (r,s)

```
explain analyze select * from opt.tab1 t1 inner join opt.tab2 t2 on t1.id = t2.id
```

Output Explain Messages Notifications

QUERY PLAN

text

Hash Join (cost=27.50..288.22 rows=1572 width=8) (actual time=0.471..18.418 rows=1016 loops=1)

Hash Cond: (t1.id = t2.id)

-> Seq Scan on tab1 t1 (cost=0.00..145.00 rows=10000 width=4) (actual time=0.041..7.334 rows=10000 loops=1)

-> Hash (cost=15.00..15.00 rows=1000 width=4) (actual time=0.408..0.408 rows=1000 loops=1)

Buckets: 1024 Batches: 1 Memory Usage: 44kB

-> Seq Scan on tab2 t2 (cost=0.00..15.00 rows=1000 width=4) (actual time=0.023..0.174 rows=1000 loops=1)

Planning Time: 0.472 ms

Execution Time: 18.517 ms

Figure 9.10 Hash Join

9.5.3 Merge Join

Merge Join is an algorithm wherein each record of outer relation is matched with each record of inner relation until there is a possibility of join clause matching.

This join algorithm is only used:

- if both relations are sorted and
- join clause operator is “=”.

The join between relation A and B with condition A.ID = B.ID can be represented as below:

For each tuple r in A

For each tuple s in B

If (r.ID = s.ID)

Emit output tuple (r,s)

Break;

If (r.ID > s.ID)

Continue;

Else

Break;


```
create index on opt.tab1(id);
explain analyze select * from opt.tab1 t1 inner join opt.tab2 t2 on t1.id = t2.id
```

Output	Explain	Messages	Notifications
QUERY PLAN text			
Merge Join (cost=65.16..136.86 rows=1572 width=8) (actual time=0.734..4.895 rows=1016 loops=1)			
Merge Cond: (t1.id = t2.id)			
-> Index Only Scan using tab1_id_idx on tab1 t1 (cost=0.29..450.28 rows=10000 width=4) (actual time=0.035..1.420 rows=1023 loops=1)			
Heap Fetches: 1023			
-> Sort (cost=64.83..67.33 rows=1000 width=4) (actual time=0.677..2.700 rows=1386 loops=1)			
Sort Key: t2.id			
Sort Method: quicksort Memory: 71kB			
-> Seq Scan on tab2 t2 (cost=0.00..15.00 rows=1000 width=4) (actual time=0.030..0.181 rows=1000 loops=1)			
Planning Time: 1.093 ms			
Execution Time: 5.001 ms			

Figure 9.11 Merge Join (one index)

```
--create index on opt.tab1(id);
create index on opt.tab2(id);
explain analyze select * from opt.tab1 t1 inner join opt.tab2 t2 on t1.id = t2.id
```

Output	Explain	Messages	Notifications
QUERY PLAN text			
Merge Join (cost=0.61..121.93 rows=1572 width=8) (actual time=0.051..3.127 rows=1016 loops=1)			
Merge Cond: (t2.id = t1.id)			
-> Index Only Scan using tab2_id_idx on tab2 t2 (cost=0.28..55.26 rows=1000 width=4) (actual time=0.021..0.876 rows=1000 loops=1)			
Heap Fetches: 1000			
-> Index Only Scan using tab1_id_idx on tab1 t1 (cost=0.29..450.28 rows=10000 width=4) (actual time=0.017..1.588 rows=1385 loops=1)			
Heap Fetches: 1385			
Planning Time: 0.965 ms			
Execution Time: 3.223 ms			

Figure 9.12 Merge Join (two indexes)

PostgreSQL joins settings

Below are the planner configuration parameters specific to join methods.

enable_nestloop: It corresponds to Nested Loop Join.

enable_hashjoin: It corresponds to Hash Join.

enable_mergejoin: It corresponds to Merge Join.

For example: set enable_hashjoin to off;

10 Transactions and Concurrency Control

A Database Transaction is a logical unit of processing in a DBMS which entails one or more database access operation (set of Select, Insert, Update and Delete operations).

All Select & DML commands which are held between the beginning and end transaction statements are considered as a single logical transaction in DBMS.

During the transaction the database may be inconsistent.

Only once the database is committed the state is changed from one consistent state to another.

Transaction features

- A transaction is a program unit whose execution may or may not change the contents of a database.
- The transaction concept in DBMS is executed as a single unit.
- If the database operations do not update the database but only retrieve data, this type of transaction is called a read-only transaction.
- DBMS transactions must be atomic, consistent, isolated and durable (ACID);
- If the database were in an inconsistent state before a transaction, it would remain in the inconsistent state after the transaction.

Transactions and Concurrent Access

A database is a shared resource so it is used by many users and processes concurrently.

For example, the banking system, railway, and air reservations systems, stock market monitoring, supermarket inventory, and checkouts, etc.

Not managing concurrent access may create issues like:

Hardware failure and system crashes

Concurrent execution of the same transaction, deadlock, or slow performance

10.1 States of Transactions

The various states of a transaction concept in DBMS are listed below:

Table 10.1 States of transactions

State	Transaction types
Active	A transaction is said to an active state if the instruction is executing.
Partially Committed	Partially committed state means that all the instructions are executed but changes are temporary and not updated in the database.
Committed	When changes are made permanent than it is said to be committed.
Failed	If any problem is detected either during active state or partially committed state than transaction enters in a failed state.
Aborted	Aborted state means all the changes that were in the local buffer are deleted. Either we are committed or aborted database is consistent.

Graphically they can be represented as the following:

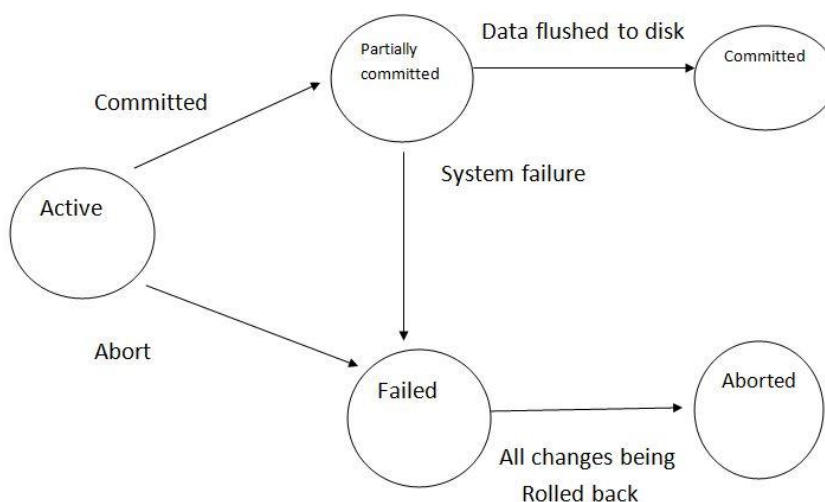


Figure 10.1 State transitions of transaction

10.2 ACID: Properties of Transaction

ACID properties in DBMS make the transaction over the database more reliable and secure. This is one of the advantages of the database management system over the file system.

In the context of transaction processing, the acronym ACID refers to the four key properties of a transaction: atomicity, consistency, isolation, and durability.

Atomicity

All changes to data are performed as if they are a single operation. That is, all the changes are performed, or none of them are.

For example, in an application that transfers funds from one account to another, the atomicity property ensures that, if a debit is made successfully from one account, the corresponding credit is made to the other account.

Let's check ACID properties in DBMS with examples.

Here, the set of operations are:

1. Deduct the amount of \$100 from Alice's account.
2. Add amount \$100 to Bob's account.

All operations in this set should be done.

If the system fails to add the amount in Bob's account after deducting from Alice's account, revert the operation on Alice's account.

Consistency

Data is in a consistent state when a transaction starts and when it ends.

For example, in an application that transfers funds from one account to another, the consistency property ensures that the total value of funds in both the accounts is the same at the start and end of each transaction.

Example:

The total amount in Alice's and Bob's account should be the same before and after the transaction. The sum of the money in Alice and Bob's account before and after the transaction is \$200. So, this transaction preserves consistency ACID properties in DBMS.

Isolation

The intermediate state of a transaction is invisible to other transactions. As a result, transactions that run concurrently appear to be serialized.

For example, in an application that transfers funds from one account to another, the isolation property ensures that another transaction sees the transferred funds in one account or the other, but not in both, nor in neither.

Example:

If there is any other transaction (between Mac and Alice) going, it should not make any effect on the transaction between Alice and Bob. Both the transactions should be isolated.

Durability

After a transaction successfully completes, changes to data persist and are not undone, even in the event of a system failure.

For example, in an application that transfers funds from one account to another, the durability property ensures that the changes made to each account will not be reversed.

The changes made during the transaction should exist after completion of the transaction.

Sometimes it may happen as all the operation in the transaction completed but the system fails immediately. In that case, changes made while transactions should persist. The system should return to its previous stable state.

Example:

It may happen. A system gets crashed after completion of all the operations. If the system restarts it should preserve the stable state. An amount in Alice and Bob's account should be the same before and after the system gets a restart.

10.3 Transaction Isolation Levels (Phenomena)

Isolation levels define the degree to which a transaction must be isolated from the data modifications made by any other transaction in the database system. A transaction isolation level is defined by the following phenomena:

1. **Dirty reads** occur when:

Transaction A inserts a row into a table.

Transaction B reads the new row.

Transaction A rolls back.

Transaction B may have done work to the system based on the row inserted by transaction A, but that row never became a permanent part of the database.

2. **Non Repeatable reads** occur when:

Transaction A reads a row.

Transaction B changes the row.

Transaction A reads the same row a second time and gets the new results.

Phantom reads occur when:

Transaction A reads all rows that satisfy a WHERE clause on an SQL query.

Transaction B inserts an additional row that satisfies the WHERE clause.

Transaction A re-evaluates the WHERE condition and picks up the additional row.

3. **Phantom reads** occur when:

Transaction A reads all rows that satisfy a WHERE clause on an SQL query.

Transaction B inserts an additional row that satisfies the WHERE clause.

Transaction A re-evaluates the WHERE condition and picks up the additional row.

4. **Serialization anomaly** (Write Skew)

Assume that we have several concurrent transactions in progress, both do some reading and writing with a table. In case if the final table state will depend on the order of running and committing these transactions, then it is Serialization anomaly.

Snapshot Isolation (SI) is for REPEATABLE READ

Serializable Snapshot Isolation (SSI) is for SERIALIZABLE.

10.3.1 **Black and White Example**

The example is from SSI (PostgreSQL Wiki):

create table dots

(

id int not null primary key,

color text not null

);

insert into dots

with x(id) as (select generate_series(1,10))

select id, case when id % 2 = 1 then 'black'

else 'white' end from x;

Let's run transactions T1 and T2:

```
begin;  
update dots set color = 'black'  
  where color = 'white';  
select * from dots;
```

a Output Explain Messages Notificatio

id [PK] integer	color text
1	black
3	black
5	black
7	black
9	black
2	black
4	black
6	black
8	black
10	black

Figure 10.2 Commands of Transaction T1

```

begin;
update dots set color = 'white'
  where color = 'black';
select * from dots;

```

[a Output](#)
[Explain](#)
[Messages](#)
[Notification](#)

id [PK] integer	color text
2	white
4	white
6	white
8	white
10	white
1	white
3	white
5	white
7	white
9	white

Figure 10.3 Commands of Transaction T2

```

commit;
select * from dots;

```

[a Output](#)
[Explain](#)
[Message](#)

id [PK] integer	color text
2	black
4	black
6	black
8	black
10	black
1	white
3	white
5	white
7	white
9	white

Figure 10.4 Commit command's result of T1


```
commit;
select * from dots;
```

id	color
2	white
4	white
6	white
8	white
10	white
1	white
3	white
5	white
7	white
9	white

Figure 10.5 Commit command's result of T2

10.4 Four Isolation Levels

Read Uncommitted is the lowest isolation level. In this level, one transaction may read not yet committed changes made by other transaction, thereby allowing dirty reads. In this level, transactions are not isolated from each other. PostgreSQL does not support it.

Read Committed guarantees that any data read is committed at the moment it is read. Thus, it does not allow dirty read. The transaction holds a read or write lock on the current row, and thus prevent other transactions from reading, updating or deleting it.

Repeatable Read is the most restrictive isolation level. The transaction holds read locks on all rows it references and writes locks on all rows it inserts, updates, or deletes. Since other transaction cannot read, update or delete these rows, consequently it avoids non-repeatable read.

Serializable is the highest isolation level. A serializable execution is guaranteed to be serializable. Serializable execution is defined to be an execution of operations in which concurrently executing transactions appears to be serially executing.

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly
Read uncommitted	Allowed, but not in PG	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Allowed, but not in PG	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

Figure 10.6 Isolation levels

10.5 Schedule (Serialization)

A Schedule is a process creating a single group of the multiple parallel transactions and executing them one by one.

It should preserve the order in which the instructions appear in each transaction.

If two transactions are executed at the same time, the result of one transaction may affect the output of other.

Example:

Initial Value is 10

Transaction 1: Update Value to 20

Transaction 2: Read Product Quantity

10.5.1 Concurrency Control

Concurrency Control in Database Management System is a procedure of managing simultaneous operations without conflicting with each other.

It ensures that Database transactions are performed concurrently and accurately to produce correct results without violating data integrity of the respective Database.

Concurrent access is easy if all users are just reading data.

DBMS Concurrency Control is used to address conflicts of Read/Write operations, which mostly occur with a multi-user system.

10.5.2 Concurrency Control techniques

Lock-Based Protocols is a mechanism in which a transaction cannot Read or Write the data until it acquires an appropriate lock (pessimistic).

Two Phase Locking Protocol (2PL) is a method of concurrency control that ensures serializability by applying a lock to the transaction data which blocks other transactions to access the same data simultaneously.

This locking protocol divides the execution phase of a transaction into three different parts: 1) when the transaction begins to execute, it requires permission for the locks it needs; 2) where the transaction obtains all the locks. When a transaction releases its first lock, then 3) the transaction cannot demand any new locks. Instead, it only releases the acquired locks.

Strict 2PL: never releases a lock after using it. It holds all the locks until the commit point where all locks are released.

Timestamp-Based Protocols is an algorithm which uses the System Time or Logical Counter as a timestamp to serialize the execution of concurrent transactions. The Timestamp-based protocol ensures that every conflicting read and write operations are executed in a timestamp order.

Validation-Based Protocols also known as Optimistic Concurrency Control Technique is a method to avoid concurrency in transactions. In this protocol, the local copies of the transaction data are updated rather than the data itself, which results in less interference while execution of the transaction. There are three phases: Read Phase (locally, then changes), Validation Phase (checks if no conflicts), Write Phase (writes or rollbacks).

Multi Version Concurrency Control (MVCC)

MVCC is an algorithm to provide fine concurrency control by maintaining multiple versions of the same object so that READ and WRITE operation do not conflict. WRITE means UPDATE and DELETE, as newly INSERTed record anyway will be protected as per isolation level.

Each WRITE operation produces a new version of the object and each concurrent read operation reads a different version of the object depending on the isolation level.

Since read and write both operating on different versions of the same object so none of these operations required to completely lock and hence both can operate concurrently.

The only case where the contention can still exist is when two concurrent transaction tries to WRITE the same record.

REFERENCES

1. “PostgreSQL Tutorial - Learn PostgreSQL From Scratch.” PostgreSQL Tutorial - Learn PostgreSQL From Scratch, www.postgresqltutorial.com . Accessed 11 Nov. 2022.
2. “pgAdmin 4 — pgAdmin 4 6.15 Documentation.” pgAdmin 4 — pgAdmin 4 6.15 Documentation, www.pgadmin.org/docs/pgadmin4/latest/index.html. Accessed 11 Nov. 2022.
3. “PostgreSQL: Documentation.” PostgreSQL: Documentation, www.postgresql.org/docs . Accessed 11 Nov. 2022.
4. “Learn PostgreSQL Tutorial - Javatpoint.” www.javatpoint.com, www.javatpoint.com/postgresql-tutorial . Accessed 11 Nov. 2022.
5. Forta, Ben. *SQL in 10 Minutes a Day, Sams Teach Yourself*. 5th ed., Pearson Education, 2019.
6. Vasilik, Sylvia Moestl. *SQL Practice Problems: 57 Beginning, Intermediate, and Advanced Challenges for You to Solve Using a Learn-by-Doing Approach*. 2017.
7. Beaulieu, Alan. *Learning SQL*. 2nd ed., O’Reilly Media, 2009.
8. Beighley, Lynn. *Head First SQL: Your Brain on SQL -- A Learner’s Guide*. O’Reilly Media, 2007.
9. Regina, and Leo Hsu. *PostgreSQL - Up and Running* 3e. O’Reilly Media, 2017.
10. Dombrovskaya, Henrietta, et al. *PostgreSQL Query Optimization: The Ultimate Guide to Building Efficient Queries*. 1st ed., APress, 2021.
11. Kumar, Vallarapu Naga Avinash. *PostgreSQL 13 Cookbook: Over 120 Recipes to Build High-Performance and Fault-Tolerant PostgreSQL Database Solutions*. Packt Publishing, 2021.
12. Stones, Richard, and Neil Matthew. *Beginning Databases with PostgreSQL: From Novice to Professional*. 2nd ed., APress, 2005.
13. Debarros, Anthony. *Practical Sql, 2nd Edition: A Beginner’s Guide to Storytelling with Data*. No Starch Press, 2022.
14. Bagui, Sikha Saha, and Richard Walsh Earp. *Database Design Using Entity-Relationship Diagrams*. 3rd ed., Taylor & Francis, 2022.
15. Harrington, Jan L. *Relational Database Design and Implementation*. 4th ed., Morgan Kaufmann, 2016.