**Data:**

Data is generally a raw format of any information (can be anything and everything) It may be text information or unprocessed information.

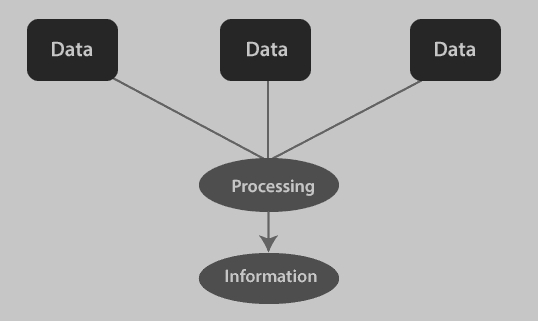
Can be stored into database.

For example − Employee name, Product name, Name of the student, Marks of the student, Mobile number, Image etc.

**Information:**

Information is nothing but that data being viewed in a structured forma

For example: Report card sheet.



**Database:**

A database is an organized collection of structured information, or data, typically stored electronically in a computer system.

**Purpose of stroing:** The data can then be easily accessed, managed, modified, updated, controlled, and organized. Most databases use structured query language (SQL) for writing and querying data.

**DBMS:**

A database typically requires a comprehensive database software program known as a database management system (DBMS).

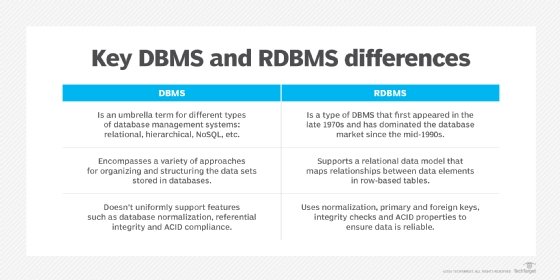
A DBMS serves as an interface between the database and its end users or programs, allowing users to retrieve, update, and manage how the information is organized and optimized.

**Examples:** MySQL, PostreSQL, Oracle , MongoDB etc..

**RDBMS** :

stands for Relational Database Management System.  
 It is software used to store, manage, and organize data in tables (also called relations). These tables are related to each other using keys (like primary and foreign keys)

Examples: Oracle, MySQL, PostgreSQL, Microsoft SQL Server, IBM DB2



**Why Do We Need RDBMS?**

**1. Data Management:**

**RDBMS** helps us **store, organize, and access** a large amount of data easily and efficiently.  
Instead of using multiple files or spreadsheets, all the data is kept in one place and can be retrieved quickly using queries.

### **2. Data Integrity:**

It ensures that the data in the database is **accurate and consistent**.  
 For example, if a student’s age is stored as 20, it should not be wrongly updated to “twenty” or a negative number.  
 Rules and constraints in DBMS help avoid such mistakes.

### **3. Concurrency Control:**

In big applications, **many users work at the same time**.  
A DBMS makes sure they can read/write data **without interfering with each other**.  
This avoids data conflicts or errors — like two people trying to book the same seat on a train.

### **4. Security:**

DBMS lets us **control who can access what data**.  
 For example, an employee might be allowed to see only their personal info, but not others’.  
 It helps prevent **unauthorized access** and keeps sensitive data safe.

### **5. Backup and Recovery:**

In case of a system crash or power failure, a DBMS can **restore lost data** using backups.  
This means your data is **safe and can be recovered** without starting from scratch.

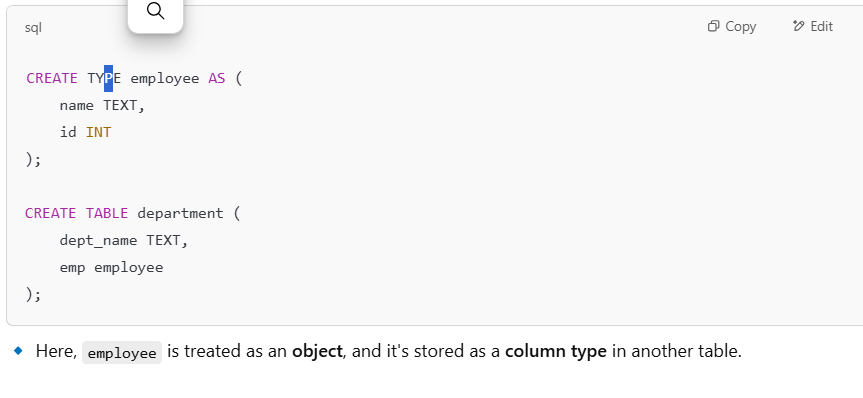
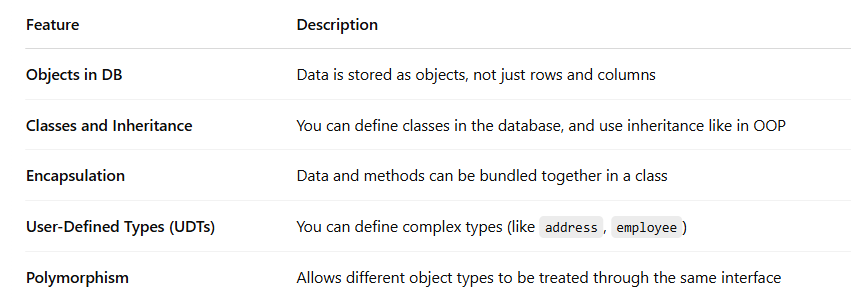
**PostgreSQL Overview**:

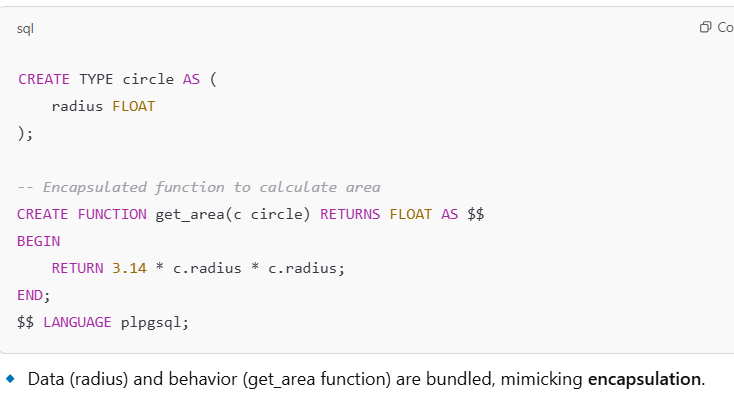
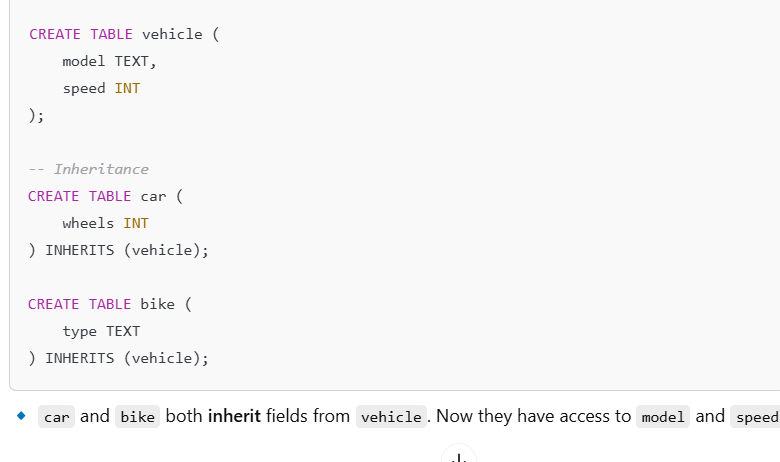
PostgreSQL is called an **ORDBMS** (Object-Relational Database Management System) because it combines features of both **relational** and **object-oriented** database models.

1. **Relational Model**: PostgreSQL is fundamentally based on the relational model. It stores data in tables (relations), supports SQL (Structured Query Language) for querying, and ensures ACID properties (Atomicity, Consistency, Isolation, Durability) for transactions.
2. **Object-Oriented Features**: PostgreSQL extends the traditional relational model by incorporating object-oriented concepts such as:
   1. **User-defined types (UDTs)**: You can create custom data types (like complex numbers, geometric data types) and store them in tables, which is not typically available in standard relational databases.
   2. **Inheritance**: PostgreSQL allows tables to inherit attributes from other tables, much like object-oriented classes inherit from other classes. This supports a hierarchy structure, which is unique for a relational database.
   3. **Functions and Methods**: You can define functions and store them in the database, similar to object methods in OOP. These functions can operate on user-defined types, enabling encapsulation of business logic.
   4. **Polymorphism**: Functions in PostgreSQL can be overloaded, which is a feature associated with object-oriented programming. This allows functions to accept different types or numbers of arguments.
3. **Support for Arrays**: PostgreSQL supports arrays as a data type, allowing multi-dimensional arrays, which is closer to the concept of objects in object-oriented programming.

By blending these object-oriented features with its core relational model, PostgreSQL qualifies as an **Object-Relational Database Management System (ORDBMS)**.

Thus, PostgreSQL's ability to handle both relational data and object-like structures is what gives it the ORDBMS designation.





By **default**, PostgreSQL **automatically includes rows from all child tables** that inherit from vehicle.



PostgreSQL supports both SQL (relational) and JSON (non-relational) querying.

The PostgreSQL project started in 1986 at California.

In 1996, the POSTGRES project was renamed to PostgreSQL to clearly illustrate its support for SQL.

The pgAdmin application allows you to interact with the PostgreSQL database server via an intuitive user interface.

# **Multi-Version Concurrency Control (MVCC):**

PostgreSQL uses a method called **MVCC** (Multi-Version Concurrency Control) instead of traditional **locking** to handle multiple users accessing the database at the same time.

* **Reading never blocks writing**, and **writing never blocks reading**.
* So your SELECT query doesn’t wait for UPDATEs or INSERTs to finish.
* This makes the database **faster and more responsive**, even with many users.

### 1. **Snapshots** – (Like Taking a Photo of the Database)

When a transaction starts, PostgreSQL gives it a **snapshot** of the database – a frozen view of the data **at that moment**.

#### Example:

* You **open your bank app** at 10:00 AM to **check your balance**.
* At the **same time**, someone **transfers money** to your account (maybe your friend sends ₹500).
* Both actions happen **at the same time**, but:
  + You're reading (SELECT)
  + The other person is writing (UPDATE)

### **Without MVCC (Old way using locks):**

* When someone is writing (updating your balance), your app would have to **wait** until the update finishes before showing your balance.
* Or worse — you might see **half-finished or wrong data** if it didn’t wait.

**With MVCC (PostgreSQL’s way):**

Here’s what happens:

1. **MVCC creates a snapshot** of the database as it looked the moment *you started your transaction* (when you opened the app).
2. Even though the ₹500 transfer is happening, your app **still sees the old version** (your balance before the transfer).
3. Once your transaction is over, you can start a new one — and that one will see the **new balance** including the transfer.

### **Think of it like this:**

* It's like the database gives you a **frozen photo** of your account at 10:00 AM.
* You can read and look at that photo without being disturbed by updates happening in real-time.
* The person sending money is creating a **new photo (version)**.
* Later, when you refresh your app, you'll see the **updated photo** with ₹500 added.

That’s how a transaction in PostgreSQL works. It always reads from the state of the database **as it was when it started**.

### 2. **Tuple Versions** – (Like Saving Old Copies of a Page)

In PostgreSQL, a **tuple** is just a fancy word for a **row in a table**.

When you update a row, PostgreSQL **doesn’t overwrite** the old one. It **creates a new version** of that row.

#### Example:

Let’s say there’s a row:  
 id = 1, name = 'Lavanya'

You update the name to 'Lav'.

PostgreSQL **keeps both versions**:

* Old: id = 1, name = 'Lavanya'
* New: id = 1, name = 'Lav'

Different transactions will **see different versions**, depending on when they started (snapshot).

MVCC is a cornerstone of PostgreSQL’s concurrency control, enabling efficient and safe concurrent transactions. By maintaining multiple versions of data, it ensures each transaction has a consistent view of the database, significantly improving performance and reducing conflicts

### 3. **Transaction IDs (XIDs)** – (Like Giving Everyone a Number)

Every time a transaction starts, PostgreSQL gives it a unique number called a **Transaction ID** (or XID)

Example:

* Transaction A starts → gets XID = 101
* Transaction B starts later → gets XID = 102

PostgreSQL uses these IDs to decide **which row version** (tuple) each transaction should see based on when it started.

### 4. **Visibility Rules** – (Who Sees What and When)

PostgreSQL uses **snapshot + XIDs + tuple versions** to apply **visibility rules** — to decide which version of a row a transaction is allowed to see.

#### Example:

* At 10:00 AM, Transaction A starts (XID = 101).
* At 10:02 AM, someone updates a row (creates a new version).
* At 10:03 AM, Transaction B starts (XID = 102).

Now:

* Transaction A **sees the old version**, because that’s what was in its snapshot.
* Transaction B **sees the new version**, since it started after the update

### **Insertions** – (New rows are visible only to you at first)

When you insert a new row in a transaction, **you can see it immediately**, but **others can't** until you **commit**.

#### Example:

You start a transaction and insert

### **PostgreSQL vs MySQL** — **MVCC (Multi-Version Concurrency Control)**

Both **PostgreSQL** and **MySQL (InnoDB engine)** use MVCC to allow multiple transactions to happen at the same time **without blocking** each other.

But how they implement MVCC is **very different**.

### Example:

Let’s say two users are working on the same row.

#### In PostgreSQL:

* **User A** updates the row → PostgreSQL creates a **new version** of the row in the table.
* **User B** reads the table → sees the **old version** if their transaction started before the update.

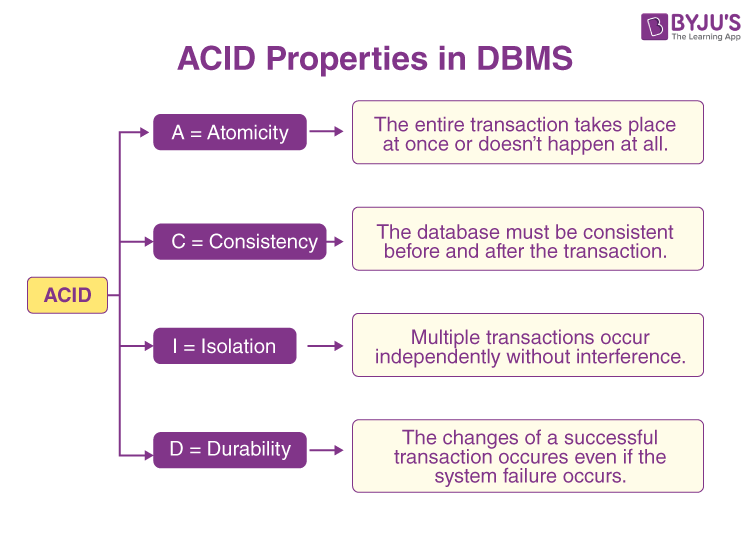
No locks, no wait, both run smoothly.

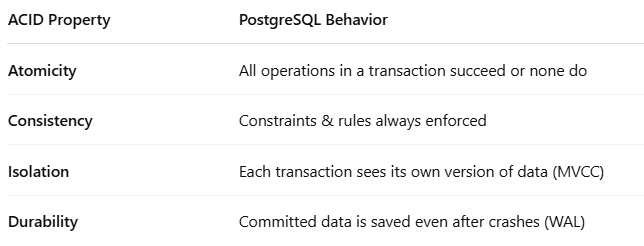
#### In MySQL (InnoDB):

* **User A** updates the row → MySQL stores the **old version** in an undo log.
* **User B** reads the table → MySQL reconstructs the old version from the undo log.

Works fine, but **more I/O overhead** than PostgreSQL, especially under heavy load.

Acid Complaince:



ostgreSQL is known for **strong ACID compliance by default** — making it super reliable for banking, e-commerce, and any app where **data accuracy matters**.

**Postgresql vs mysql:**

**ORDBMS vs RDBMS**

* **PostgreSQL**:  
   ➤ It’s an **Object-Relational Database Management System (ORDBMS)**.  
   ➤ This means it supports not just tables and rows, but also **custom data types, arrays, inheritance**, etc.
* **MySQL**:  
   ➤ It’s a **Relational Database Management System (RDBMS)**.  
   ➤ Focuses mainly on traditional relational data with rows and tables.

️ *"PostgreSQL handles complex data models better than MySQL."*

## **2. SQL Standards Compliance**

* **PostgreSQL**:  
   Very strict in following **ANSI SQL standards**.  
   Supports advanced SQL features like **window functions, CTEs, full joins**.
* **MySQL**:  
   Only **partially follows** SQL standards.  
   Some features behave differently or are missing (e.g., limited support for CTEs in older versions).

*"If your app depends on strict SQL rules, PostgreSQL is more reliable."*

## **3. Extensibility**

* **PostgreSQL**:  
   ➤ You can **add your own data types, operators, functions, and even languages** (like Python or Perl).  
   ➤ Ideal for custom or scientific applications.
* **MySQL**:  
   ➤ Limited ability to extend — fewer options for adding custom features.

️ *"PostgreSQL is more flexible when your app grows or needs custom features."*

## **4. ACID Compliance**

* **PostgreSQL**:  
   **Fully ACID-compliant** in all cases, by default.  
   Uses **WAL (Write-Ahead Logging)** for crash recovery.
* **MySQL**:  
   Only ACID-compliant **if you use the InnoDB engine** and proper settings.  
   Other engines (like MyISAM) don’t support ACID at all.

️ *"PostgreSQL is safer by design for critical applications."*

## **5. Concurrency**

* **PostgreSQL**:  
   Uses **true MVCC** with **Snapshot Isolation**.  
   No need for read locks — high performance with many users.
* **MySQL**:  
   Uses MVCC with **Repeatable Read** (by default),  
   But it can **allow anomalies** like phantom reads or lost updates.

*"PostgreSQL handles multi-user environments better."*

## **6. Triggers & Procedures**

* **PostgreSQL**:  
   Very **powerful trigger system** (can run on insert, update, delete, etc.).  
   Supports **stored procedures** with multiple languages (PL/pgSQL, Python, etc.)
* **MySQL**:  
   Limited trigger support — one trigger per action per table.  
   Stored procedures are less powerful and flexible.

*"If you use business logic inside the DB, PostgreSQL is more capable."*

## **7. JSON & NoSQL Features**

* **PostgreSQL**:  
   Has **advanced JSON/JSONB support** — can index and query JSON data.  
   ️ Acts like a mini NoSQL DB when needed.
* **MySQL**:  
   Basic JSON support (from version 5.7),  
   But lacks advanced features like indexing, JSON functions.

*"PostgreSQL is better for hybrid apps (relational + document data)."*

## **8. Performance Tuning**

* **PostgreSQL**:  
   Offers **fine-grained control** over the query planner, indexes, etc.  
   ️ Can optimize for performance manually.
* **MySQL**:  
   Easier for beginners — works well with defaults.  
   But fewer tuning options for complex performance scenarios.

️ *"PostgreSQL gives more control, MySQL is easier to get started."*

## **9. Replication**

* **PostgreSQL**:  
   Supports **logical and physical replication**,  
   Also supports **synchronous** and **asynchronous** replication.
* **MySQL**:  
   Has basic **asynchronous replication**  
   **Synchronous replication** is not native (requires Group Replication or Galera).

*"PostgreSQL gives more robust and flexible replication options."*

Maximum size for a database? unlimited (32 TB databases exist)  
Maximum size for a table? 32 TB  
Maximum size for a row? 400 GB  
Maximum size for a field? 1 GB  
Maximum number of rows in a table? unlimited  
Maximum number of columns in a table? 250-1600 depending on column types  
Maximum number of indexes on a table? Unlimited

**SCHEMA:**

1. **What is the difference between a schema and a database in PostgreSQL?**
2. **Can two schemas have tables with the same name? How does PostgreSQL resolve such conflicts when querying?**
3. **If a user has access to a database, does it automatically mean they can access all schemas and tables within it? Why or why not?**
4. **How does the search\_path setting affect which schema is used during SQL operations?**
5. **What happens if two schemas have a function with the same name? How does PostgreSQL decide which one to execute?**
6. **Assume there are two schemas: public and sales. Both have a table customers. If you run SELECT \* FROM customers;, which table will be used?**
7. **You created a new schema inventory and added a table products. You try SELECT \* FROM products; and get an error. Why?**
8. **You have a user john who needs access only to the hr schema. How would you grant this access securely without exposing other schemas?**
9. **How would you transfer all tables from one schema (old\_schema) to another (new\_schema) without losing data or structure?**
10. **Can a table in one schema have a foreign key referencing a table in another schema? If yes, how?**
11. **Can you rename a schema? What are the implications of doing so in a production environment?**
12. **What is the impact of dropping a schema with the CASCADE option? Can this be reversed?**
13. **How would you set a default schema for a specific user session?**
14. **How can schema-based multitenancy be implemented in PostgreSQL? What are the pros and cons?**
15. **Explain how schemas can be used for versioning database objects (like in staging vs. production)?**
16. **You grant a user USAGE on a schema but they still can't SELECT from a table. Why? Hint: Think about what USAGE means vs. table-level privileges.**
17. **How can you temporarily override the search\_path for a single query?**
18. **What happens if a function references a table without schema qualification and that table exists in multiple schemas?**
19. **You want to duplicate a schema including its structure and data. How would you do it without using pg\_dump?**
20. **You renamed a schema and now a view is failing. What likely went wrong?**
21. **If you revoke access to the public schema, what effect does it have on new users?**
22. **Can you restrict a user from creating objects in a schema but still allow them to read data?**
23. **You gave a role permission on a schema, but they still can’t access a function inside it. Why?**
24. **How does PostgreSQL resolve function overloading in different schemas with the same function name and signature?**
25. **If a trigger function exists in schema\_a and is attached to a table in schema\_b, will it work? What should be considered?**
26. **How does PostgreSQL handle the same table name in different schemas during a JOIN without schema qualification?**
27. **What happens if you drop the schema of a function being referenced in a trigger?**
28. **How can schemas be used to implement feature flags or A/B testing in a PostgreSQL app?**
29. **You want to create a “read-only view” of your production schema for analytics. How can you safely do this using schemas?**
30. **How would you switch from one version of your schema to another with zero downtime?**