DBMS-Refresher-Notes

1. Data Attribute

A **data attribute** is a property or characteristic of an entity. Think of it as a column in a table.

Example:
Imagine a **student** database. The entity is **Student**, and its attributes could be:

- **Name** (John Doe)
- **Age** (20)
- **Email** (john@example.com)

Each row in the database stores specific values for these attributes.

---

2. Entity

An **entity** is an object or thing in the real world that can be stored in a database.

Example:

- A **student** in a school database
- A **car** in a vehicle database
- A **product** in an e-commerce store

Each entity has attributes (like a student has a name and roll number).

3. Cardinality

Cardinality defines the number of relationships between entities.

Types:

1. **One-to-One (1:1):** A person has **one passport**.
2. **One-to-Many (1:N):** A teacher can teach **many students**.
3. **Many-to-Many (N:N):** A student can enroll in **many courses**, and a course can have **many students**.

4. DB Design (Database Design)

Database design is about organizing data efficiently. It ensures that:
✅ Data is stored properly
✅ There are no duplicate records
✅ Queries run fast

Example:
If you're making a library database, you should have tables like:

1. **Books (ISBN, Title, Author## 1. Data Attribute****

A **data attribute** is a property or characteristic of an entity. Think of it as a column in a table.

Example:
Imagine a **student** database. The entity is **Student**, and its attributes could be:

- **Name** (John Doe)
- **Age** (20)

- **Email** (john@example.com)

Each row in the database stores specific values for these attributes.

5. Relationships

Relationships connect different tables in a database.

- **One-to-One (1:1)** → Each person has **one** unique ID card.
- **One-to-Many (1:N)** → A customer can place **many** orders.
- **Many-to-Many (N:N)** → A student enrolls in **many** courses, and a course has **many** students.

Example:
A **Users** table and an **Orders** table are connected because one user can place many orders.

One-to-one

A one-to-one relationship in a database means that each record in one table is linked to exactly one record in another table, and vice versa.

Imagine a system where each **employee** has **one unique company car**.

Table: Employees

EmployeeID	Name
1	Alice
2	Bob

Table: CompanyCars

CarID	EmployeeID	CarModel
101	1	Toyota Camry
102	2	Honda Civic

Here, each employee is associated with **one car**, and each car is assigned to **one employee**.

One-to-many

A one-to-many relationship in a database means that one record in a table can be associated with multiple records in

Example Scenario:

Imagine a system where **one teacher** can teach **many students**, but each student has **only one teacher**.

Table: Teachers

TeacherID	Name
1	Mr. Shah
2	Ms. Rao

Table: Students

StudentID	Name	TeacherID
101	Aditi	1
102	Rohan	1
103	Meera	2

Here:

- Mr. Shah teaches Aditi and Rohan.
- Ms. Rao teaches Meera.

Many-to-many

A many-to-many relationship in a database means that multiple records in one table can be associated with multiple records in another table.

Example Scenario:

Imagine a system where **students can enroll in multiple courses**, and **each course can have multiple students**.

Table: Students

StudentID	Name
1	Aditi
2	Rohan

Table: Courses

CourseID	Title
101	Math
102	Science

Linking Table: Enrollments

StudentID	CourseID
1	101
1	102
2	101

Here:

- Aditi is enrolled in Math and Science.
- Rohan is enrolled in Math.
- Math has both Aditi and Rohan.

6. Data Types

Each column in a database has a **data type**, which defines what kind of values it can store.

Common Data Types:

- **INT** → Numbers (e.g., age, price)
- **VARCHAR** → Text (e.g., names, addresses)
- **DATE** → Dates (e.g., 2025-03-07)
- **BOOLEAN** → True or False

Example:

A **Student** table may have:

- **Name** → `VARCHAR(100)`
- **Age** → `INT`
- **Enrollment Date** → `DATE`

7. Difference Between Primary and Unique Key

Feature	Primary Key	Unique Key
Purpose	Uniquely identifies each record	Ensures uniqueness but allows NULL values
NULL Allowed?	No	Yes
Number per Table	Only **one** primary key	Multiple unique keys allowed

Example:

A **Student** table may have:

- **StudentID (Primary Key)** → Always unique
- **Email (Unique Key)** → Ensures no two students have the same email

8. 1:1, 1:N, N:N Relationships

One-to-One (1:1): Each employee has **one** company car.

One-to-Many (1:N): A teacher teaches **many** students.

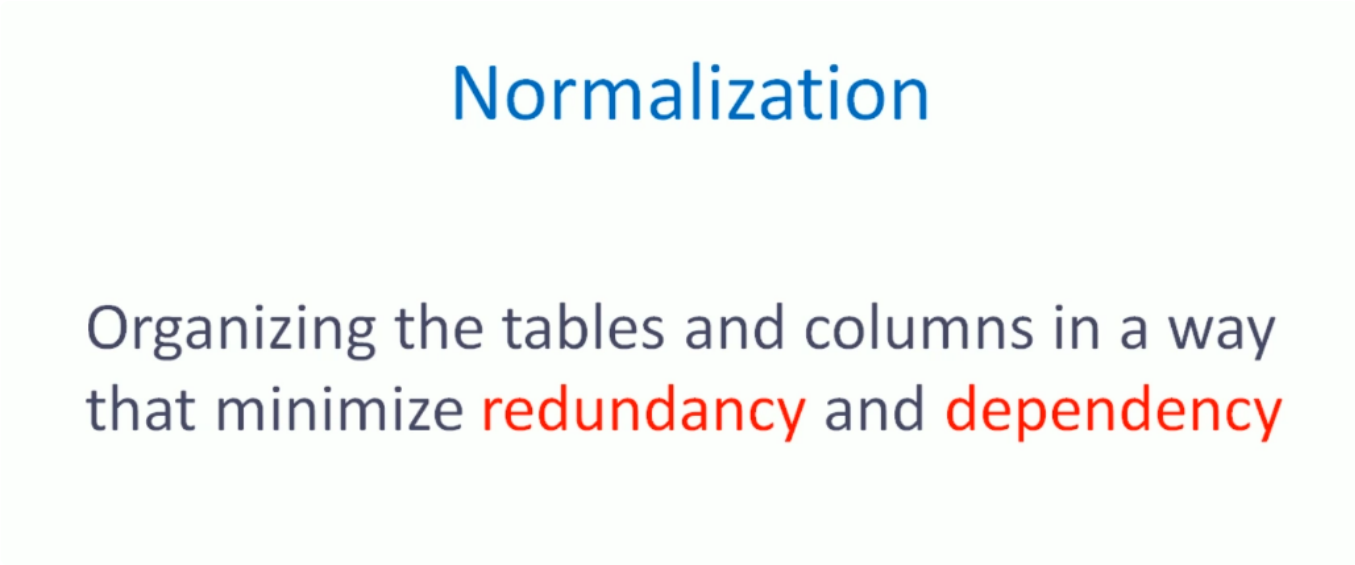
Many-to-Many (N:N): Students enroll in **many** courses, and a course has **many** students.

Example:

A **Library Database** may have:

- **Books** (BookID, Title)
- **Students** (StudentID, Name)
- **BorrowedBooks** (StudentID, BookID, BorrowDate) → (N:N relationship)

9. *Normalization* and its Types



Anomalies

Anomalies

- INSERT
  - Unable to insert customer before invoice is created.
- UPDATE
  - Customer contact details with each invoice. Need to update multiple places.
- DELETE
  - Customer details are lost when invoice is deleted.

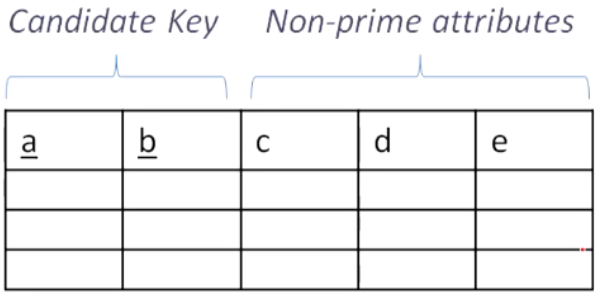
1NF

First normal form(1NF)

- Attributes/Columns are uniquely named.
- The order of columns and rows is insignificant.
- Every row and column intersection contains exactly one value of the applicable domain, and nothing else.
- No duplicate rows

2NF

Second Normal Form (2NF)



Note: A non-prime attribute of a table is an attribute that is not part of any candidate key of the table.

Every non-prime attribute of the table is dependent on the complete key (whole of a candidate key)

Example

PARTS SUPPLY

Supplier Id	Part Id	Part Name	Quantity
supplier1	BA	Base Assembly	10
supplier1	CS	Crank Shaft	5
supplier2	BA	Base Assembly	40
supplier2	BU	Battery Unit	35
supplier3	BA	Base Assembly	8
supplier3	CS	Crank Shaft	2
supplier3	BU	Battery Unit	12

- Here there are 2 candidate keys
  - supplier id
  - part id
- The non prime attributes here are
  - Part name
  - Quantity

- 2nf says that every non prime attribute is dependent on the complete key (both the candidate keys)
- Lets check each non prime attribute one by one
  - **Part name**
    - Part name is determined by Part ID
    - But its not determined by supplier ID as well
    - So its partially dependent on the complete key
  - **Quantity**
    - Quantity is determined by the combination of supplier ID and quantity
- Transforming it into 2NF
  - We need to split into 2 tables where 2nf is satisfied

PARTS SUPPLY

Supplier Id	Part Id	Quantity
supplier1	BA	10
supplier1	CS	5
supplier2	BA	40
supplier2	BU	35
supplier3	BA	8
supplier3	CS	2
supplier3	BU	12

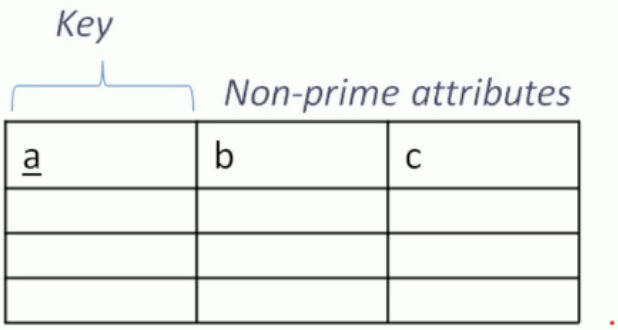
PARTS

Part Id	Part Name
BA	Base Assembly
CS	Crank Shaft
BU	Battery Unit

  -
    - Now quantity is determined by combination of supplier ID and part ID
    - Part name is determined by Part ID
    - Both tables follow 2nf

3NF

Third Normal Form (3NF)



a → b b → c

a ⇢ c

Every non-prime attribute of the table is directly dependent on the key

(non transitively dependent on the key)

Example

SALES

Order No	Customer	Product	Coupon Code	Discount(%)	Net Amount
ORD1001	Alice Cooper	Video Camera	NEWYEAR40	40	600.00
ORD1002	Barbara Park	Keyboard	NEWYEAR40	40	30.00
ORD1003	Cynthia Nixon	LCD Monitor	SUMMER30	30	70.00
ORD1004	Mohan Lal	Mobile phone	SUMMER30	30	350.00
ORD1005	Zoya Afrose	Hard Disk	SPRING25	25	450
ORD1006	Steve Smith	Printer	SPRING25	25	150.00
ORD1007	Barbara Park	Network Router	NOCOUPON	00	45.00

- Here our key is Order No, other columns are non prime attributes
- Here Discount depends on Coupon Code
- And coupon code depends on Order number
- So this is Transitively dependent
  - Discount -> Coupon Code -> Order Number
  - This is not allowed in 3NF
- We need to split the tables such that the transitive dependencies are removed
- We need a separate table for the columns in the transitive dependency

- The columns are Coupon code and discount

COUPON

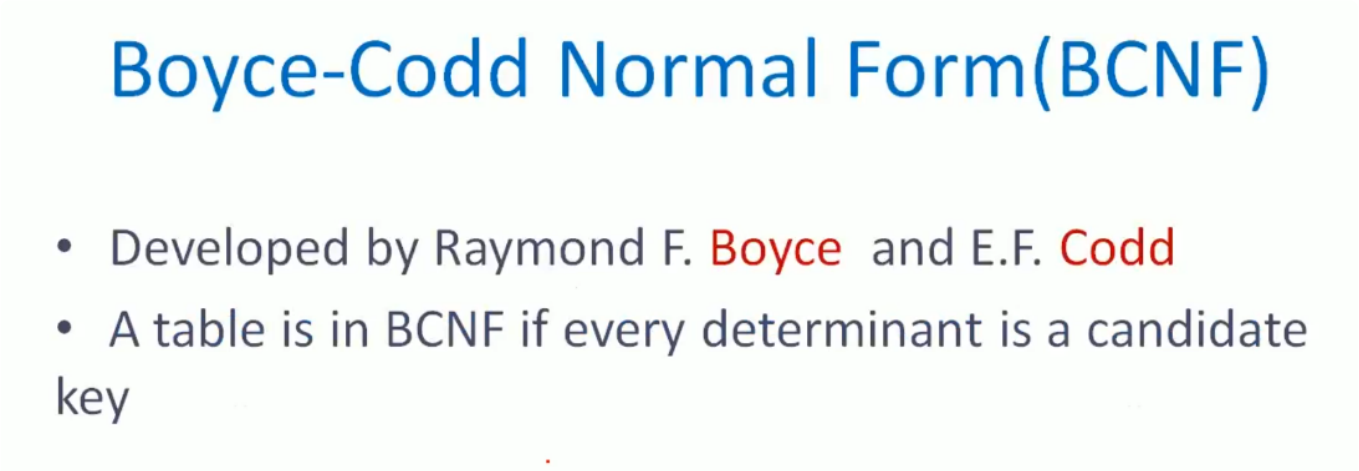
Coupon Code	Discount(%)
NEWYEAR40	40
SUMMER30	30
SPRING25	25
NOCOUPON	00

SALES

Order No	Customer	Product	Coupon Code	Net Amount
ORD1001	Alice Cooper	Video Camera	NEWYEAR40	600.00
ORD1002	Barbara Park	Keyboard	NEWYEAR40	30.00
ORD1003	Cynthia Nixon	LCD Monitor	SUMMER30	70.00
ORD1004	Mohan Lal	Mobile phone	SUMMER30	350.00
ORD1005	Zoya Afrose	Hard Disk	SPRING25	450
ORD1006	Steve Smith	Printer	SPRING25	150.00
ORD1007	Barbara Park	Network Router	NOCOUPON	45.00

- Now the Tables follow 3NF

BCNF



- Determinant is the value which can determine other attributes

PRODUCT SPECIALIST ASSIGNMENT

Customer Id	Product Id	Specialist Id	Date	Status
C10001	LED Television	Peter	01-Dec-2014	Active
C10002	Personal Computer	Reshmi	03-Jan-2014	Active
C10002	Network Router	Mallik	11-Feb-2014	Active
C10003	Mobile Phone	Roonie	07-Nov-2014	Inactive
C10005	LED Television	Peter	17-Oct-2014	Inactive
C10006	Network Router	Mallik	19-Mar-2014	Active

- Here the determinants are
- CustomerID, SpecialistID,ProductId are deteminants
- Combination of customerid, productid and customerid,specialist ID can be keys
- We need to split the tables

PRODUCT SPECIALİST

Specialist Id	Product Id	Specialization Level
Peter	LED Television	L1
Reshmi	Personal Computer	L1
Mallik	Network Router	L2
Roonie	Mobile Phone	L1

PRODUCT SPECIALIST ASSIGNMENT

Customer Id	Specialist Id	Date	Status
C10001	Peter	01-Dec-2014	Active
C10002	Reshmi	03-Jan-2014	Active
C10002	Mallik	11-Feb-2014	Active
C10003	Roonie	07-Nov-2014	Inactive
C10005	Peter	17-Oct-2014	Inactive
C10006	Mallik	19-Mar-2014	Active

, Category)**

1. **Students (ID, Name, Email)**
2. **BorrowedBooks (StudentID, ISBN, BorrowDate, ReturnDate)**

This way, the **BorrowedBooks** table links students and books without storing duplicate data.

10. Types of databases

1. Relational Databases (RDBMS)

- Structure: Tables with rows and columns.
- Examples: MySQL, PostgreSQL, Oracle, Microsoft SQL Server.
- Use Case: Structured data with clear relationships (e.g., banking, inventory systems).

2. NoSQL Databases

- These are non-relational and are designed for flexibility and scalability.

a. Document Stores

- Structure: JSON-like documents.
- Examples: MongoDB, CouchDB.
- Use Case: Content management, catalogs.

b. Key-Value Stores

- Structure: Key-value pairs.
- Examples: Redis, DynamoDB.
- Use Case: Caching, session management.

c. Column-Family Stores

- Structure: Columns grouped into families.
- Examples: Apache Cassandra, HBase.
- Use Case: Big data, real-time analytics.

d. Graph Databases

- Structure: Nodes and edges.
- Examples: Neo4j, Amazon Neptune.
- Use Case: Social networks, recommendation engines.

11. Benefits of relational databases

- Data Integrity and Accuracy
  - Enforced through constraints (e.g., primary keys, foreign keys, unique constraints).
  - Ensures that data is consistent and accurate across related tables.
- Data Relationships

    - Ideal for representing relationships between entities (e.g., customers and orders).
    - Foreign keys help maintain referential integrity.
- Scalability
    - Can handle large volumes of data with proper indexing and optimization.
    - Suitable for enterprise-level applications.

12. What is indexing and its types

Indexing is a technique used in databases to **speed up the retrieval of records** from a table.

Why is Indexing Important?

- **Faster search** (especially in large tables)
- **Efficient WHERE clause filtering**
- **Speeds up JOINs, ORDER BY, and GROUP BY**

1. Primary Index

- **Automatically created** on the **primary key** column.
- Ensures all values are **unique and not null**.
- Usually a **clustered index** in most databases.

Example:

```
CREATE TABLE students (
    student_id INT PRIMARY KEY,
    name VARCHAR(50)
);
```

> ✅ A **primary index** is created on `student_id`.

2. Unique Index

- Ensures **no duplicate values** in a column.
- Can be applied to non-primary key columns.

Example:

```
CREATE UNIQUE INDEX idx_email ON users(email);
```

> ✅ Ensures no two users can have the same email.

3. Clustered Index (Made Simple)

What it means:

- A **clustered index** is like **arranging the whole table in sorted order** based on one column.
- The actual data rows in the table are **stored in that order**.
- So, when you search or sort by that column, it becomes much faster.

Key Points:

- Only **one** clustered index is allowed per table (because you can't sort one table in two ways).
- Usually, the **primary key** automatically becomes the clustered index.

Real-life example:

Imagine you have a **phone book** sorted by **name**:

```
Adam  - 1234
Bob   - 5678
Charlie - 9999
```

Since the book is sorted by name, it's **easy to find any person quickly**.
That's what a clustered index does — it keeps the **table itself sorted**.

SQL Example:

```
CREATE CLUSTERED INDEX idx_name ON employees(name);
```

This means:

> ✅ The table `employees` is now **physically sorted** by the `name` column.

4. Non-Clustered Index

What it means:

- A **non-clustered index** is like a **separate table of shortcuts**.
- It stores the **values of the column** you indexed (like age), **along with a pointer** to where the full row is in the actual table.
- So when you search by age, it looks in this small "shortcut table" instead of scanning the whole big table.

Real-life example:

- Imagine your school attendance sheet is sorted by **roll number**, but you want to quickly find students by **age**.
- You create a **separate list** like this:

```
Age → Row Number
18  → Row 2
19  → Row 5
20  → Row 8
```

- Now when someone asks “show all students who are 19 years old,” you check this list and **jump straight to the right row** — instead of checking every row one by one.
- That’s what a **non-clustered index** does!

SQL Example:

```
CREATE NONCLUSTERED INDEX idx_age ON employees(age);
```

> ✅ Creates a **shortcut list** of ages with pointers to the real rows in the `employees` table.

Key Differences from Clustered Index:

Feature	Clustered Index	Non-Clustered Index
Data location	Data is **sorted** in the table itself	Index is **separate** from actual data
Number allowed	Only **one** per table	You can have **many**
Works best for	Sorting, range-based searches	Lookups and filtering

5. Composite Index (Multi-Column Index)

What it means:

- A **composite index** is an index made from **more than one column**.
- It helps the database quickly find rows when you're **filtering by multiple columns together**.

Real-life example:

- Imagine a school attendance register where students are grouped by **class name** and then by **roll number**.
- If someone asks:
  - "Find student in class `10A` with roll number `15`"
  - … you don't check all students — you first find class `10A`, then look for roll `15` inside it.
  That's what a **composite index** does — it creates a shortcut **based on multiple columns**.

SQL Example:

```
CREATE INDEX idx_name_age ON employees(name, age);
```

> ✅ This creates an index that helps with queries that use **both** `name` and `age`.

6. Full-Text Index

What it means:

A **Full-Text Index** helps you search **inside large text fields** — like articles, blog posts, or descriptions — **just like a search engine**.

Instead of matching exact text, it can match **words, phrases, or related terms** — like Google does.

Real-life example:

You're searching on a blog:

> "Show me all articles about **machine learning**"

Without a full-text index: the database checks **every article** word by word
With a full-text index: the database uses a **smart word index**

7. Spatial Index

A **Spatial Index** helps the database **quickly find locations** on a map — like cities, shops, or landmarks — based on **coordinates**, distance, or shapes.

It’s used when you're working with **geographic data** (latitude & longitude, maps, areas, etc.).

Real-life example:

You’re using a food delivery app and ask:

> “Show restaurants within **5 km** of me.”

Without an index: the app checks the distance to **every restaurant**
With a **spatial index**: it quickly **jumps to nearby ones only**

8. Bitmap Index

What it means:

- A **Bitmap Index** stores data as a series of **bits (0s and 1s)** instead of normal values.
- It’s super fast and space-efficient when a column has **only a few different values** — like Gender (Male/Female), Status (Active/Inactive), or Yes/No flags.

Real-life example:

Imagine you have a big list of employees and a column for gender:

```
Gender: F, M, F, F, M, M, F ...
```

Instead of storing the full text for each, the bitmap index stores:

- A **bit map** for females: 1 if female, 0 if not.
- Another bitmap for males: 1 if male, 0 if not.

So for 7 employees, Female bitmap might look like:

```
1 0 1 1 0 0 1
```

This helps quickly answer questions like:

> "How many females do we have?"
> Just **count the 1s** in the female bitmap

Summary Table:

Type	Description	Multiple Allowed?
Primary Index	Auto-created on primary key	❌ (Only one)
Unique Index	No duplicate values allowed	✅
Clustered Index	Sorts data physically by column	❌ (Only one)
Non-Clustered Index	Pointer to actual data rows	✅
Composite Index	Based on multiple columns	✅
Full-Text Index	Fast search for text content	✅
Spatial Index	Used for map/geo data	✅
Bitmap Index	Efficient for few distinct values	✅ (OLAP focus)

13. ACID Properties?

ACID is a set of 4 important rules that make sure a database works **correctly and reliably**, even if something goes wrong (like power failure or system crash).

A – Atomicity

- A transaction must **complete fully** or **not happen at all**.
- If one part of it fails, the whole thing is cancelled.
- Example:
  - You transfer ₹100 from your account to a friend.
  - Debit your account
  - Credit their account
  - If **either fails**, both should be rolled back.

C – Consistency

- The database should always follow **rules and constraints**.
- After any transaction, the data must be **valid**.
- Example:

- If a student can only have one roll number, the database must prevent two students from getting the same roll number.

I – Isolation

- If many people are using the database at the same time, their transactions should **not affect each other**.
- Example:
  - Two users trying to book the last train ticket — the database makes sure **only one succeeds**, even if they click at the same time.

D – Durability

- Once a transaction is done, the data is **permanently saved**, even if the system crashes.
- Example:
  - You make a payment → the system crashes → when it comes back, your payment is still recorded

Summary Table:

Property	Meaning	Simple Hint
Atomicity	All or nothing	Complete success or full rollback
Consistency	Always valid data	Rules are never broken
Isolation	Transactions don't mix	No interference
Durability	Changes are permanent	Crash won’t erase your work

14. Hash Join

Hash Join is a way for the database to **join two tables** by using a **hash table** to match values quickly.

It usually picks the **smaller table**, creates a **hash table** from its join column, and then goes through the larger table — using the hash to find matching rows.

This makes the join much **faster**, especially when data isn’t sorted.
It works best for **equality conditions** like `id = id`."

Think of it like this:

- Small table → make a quick **lookup table**
- Big table → for each row, **check in the hash**
- If match → combine rows

> "Let's say I have an `employees` table and a `departments` table.
> The database might use a **hash join** by creating a hash table of departments using `id`.
> Then, for each employee, it quickly checks their `dept_id` in the hash table to find the department name.
> This is much faster than comparing all rows manually."

> "It's ideal when one table is **small** and the data is **not sorted**.
> But if the data is already sorted, the DB might use a **merge join** instead."

Practice 10-second answer:

> "Hash join uses a hash table to quickly match rows between two tables. It builds the hash from the smaller table's join column and uses it to find matches in the bigger one. It's efficient for equality joins when the data isn't sorted."

15. Merge Join

What to Remember About Merge Join:

- **Merge = Sorted**
- Works like **merging two sorted lists**
- Very fast if both tables are **already sorted** on the join column

> "**Merge Join** is a join technique where the database combines two tables by **scanning them in sorted order**.
> It works best when both tables are **sorted on the join column**.
> The database moves through both tables like it's **merging two sorted lists**, finding matching rows along the way.
> It's very efficient for **large datasets** that are already sorted."

Example

> "Let's say we have a `customers` table and an `orders` table, both sorted by `customer_id`.
> The database can do a **merge join** by scanning both in order and matching customer IDs

> directly, without jumping around.”

10-second answer

> “Merge join combines two **sorted tables** by scanning them together and matching rows with the same value.
> It’s very efficient for large datasets that are sorted by the join column.”

16. Nested Join

What is a Nested Loop Join?

- It’s the **simplest** join method.
- Works like **double for-loops**:
  - For each row in Table A, check every row in Table B to find matches.

> "**Nested Loop Join** is the most basic type of join.
> The database takes each row from the first table and then **checks all rows in the second table** to find matches.
> It’s easy to implement, and it works best when **one table is small**, or when **indexes** help speed up the inner search."