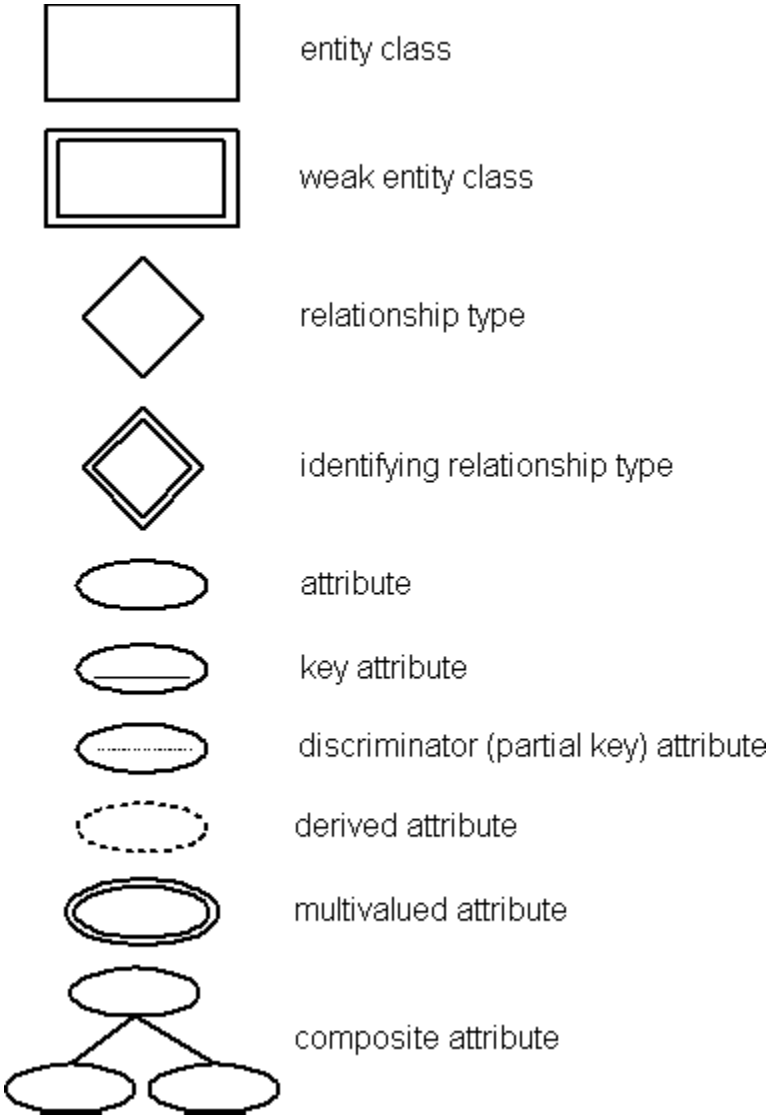
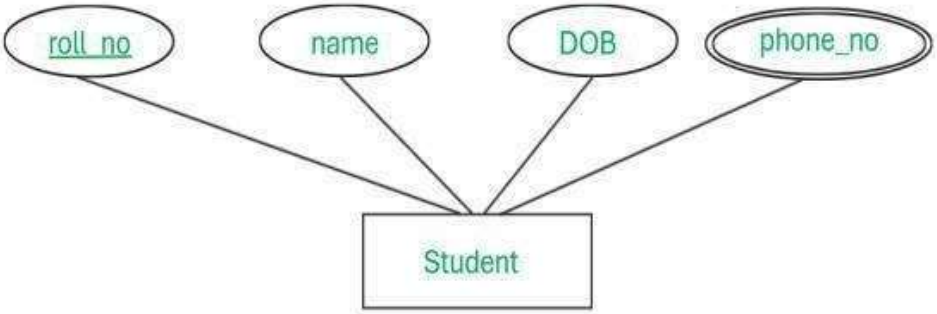
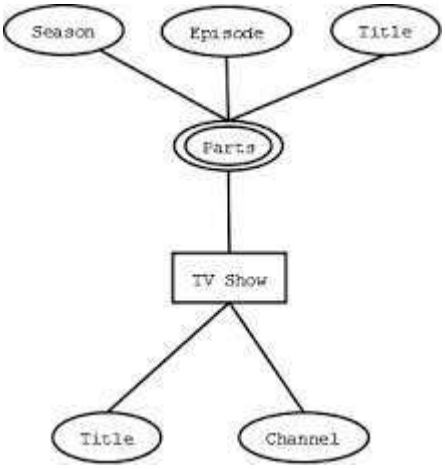
ADS

1

Define Entity and Attribute. Explain Types of Attributes in ER Model







Introduction

In the **Entity–Relationship (ER) model**, real-world objects and their properties are represented in a structured graphical form. The two fundamental concepts of this model are **Entity** and **Attribute**, which together help in designing a clear and efficient database schema.

---

1. Entity

An **Entity** is a **real-world object** that has an independent existence and can be uniquely identified.

Examples:

- Student
- Employee
- Book

– In an ER diagram, an **entity** is represented by a **rectangle**.

---

2. Attribute

An **Attribute** is a **property or characteristic** that describes an entity.

Examples:

- Student_ID, Name, Age (attributes of Student entity)
- Emp_ID, Salary (attributes of Employee entity)

—In an ER diagram, an **attribute** is represented by an **oval**.

Types of Attributes in ER Diagram

1. Simple (Atomic) Attribute

- Cannot be divided further.
- Stores indivisible data.

Example:

- Age
- Roll_No

---

2. Composite Attribute

- Can be divided into smaller sub-attributes.

Example:

- Name → First_Name, Middle_Name, Last_Name
- Address → Street, City, State, Pin_Code

---

3. Single-Valued Attribute

- Holds **only one value** for each entity.

Example:

- Date_of_Birth
- Gender

---

4. Multi-Valued Attribute

- Can store **multiple values** for a single entity.

Example:

- Phone_Number (a person may have more than one number)

- Email_IDs

– Represented by a **double oval** in ER diagram.

---

5. Derived Attribute

- Value is **derived from another attribute**.
- Usually not stored in the database.

Example:

- Age (derived from Date_of_Birth)

– Represented by a **dashed oval**.

---

6. Key Attribute

- Uniquely identifies each entity in an entity set.

Example:

- Student_ID
- Employee_ID

– Represented by **underlining** the attribute.

Example ER Representation (Explanation)

Entity: Student
Attributes:

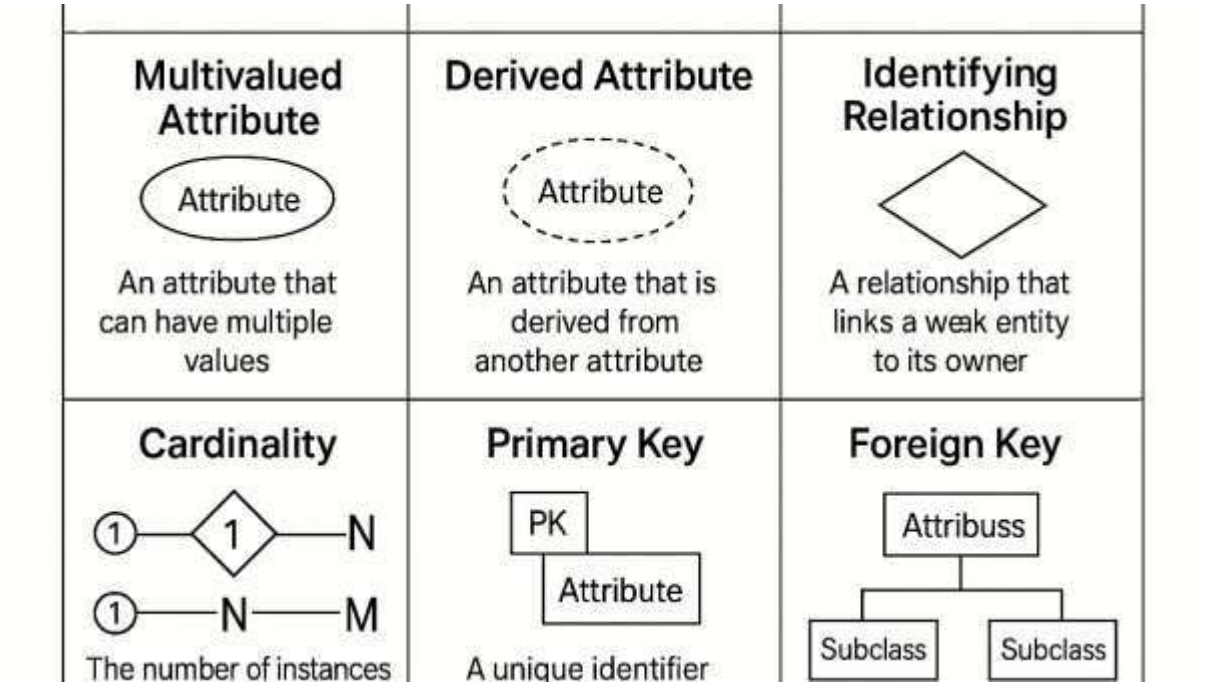
- Student_ID (Key attribute)
- Name (Composite attribute)
- Age (Derived attribute)
- Phone_Number (Multi-valued attribute)

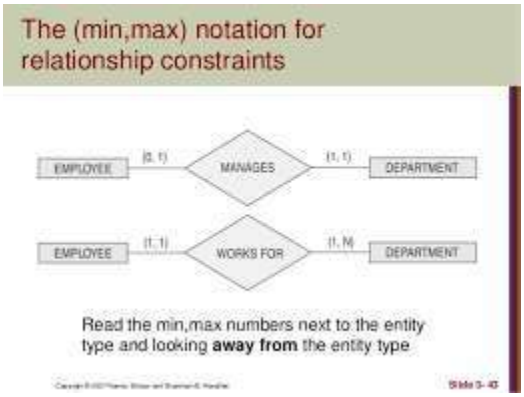
Conclusion

An **Entity** represents a real-world object, while **Attributes** describe its properties. Understanding the different types of attributes—such as **simple, composite, multi-valued, derived, and key attributes**—is essential for designing accurate and efficient ER diagrams in database systems.

2

ER Diagram Symbols s Library Management System ER Diagram





Introduction

The **Entity–Relationship (ER) model** uses standard symbols to visually represent entities, attributes, and relationships in a database system. These symbols help in designing and understanding the logical structure of a database before implementation.

---

1. Symbols Used in ER Diagram

Symbol	Meaning
Rectangle	Entity
Double Rectangle	Weak Entity
Oval	Attribute

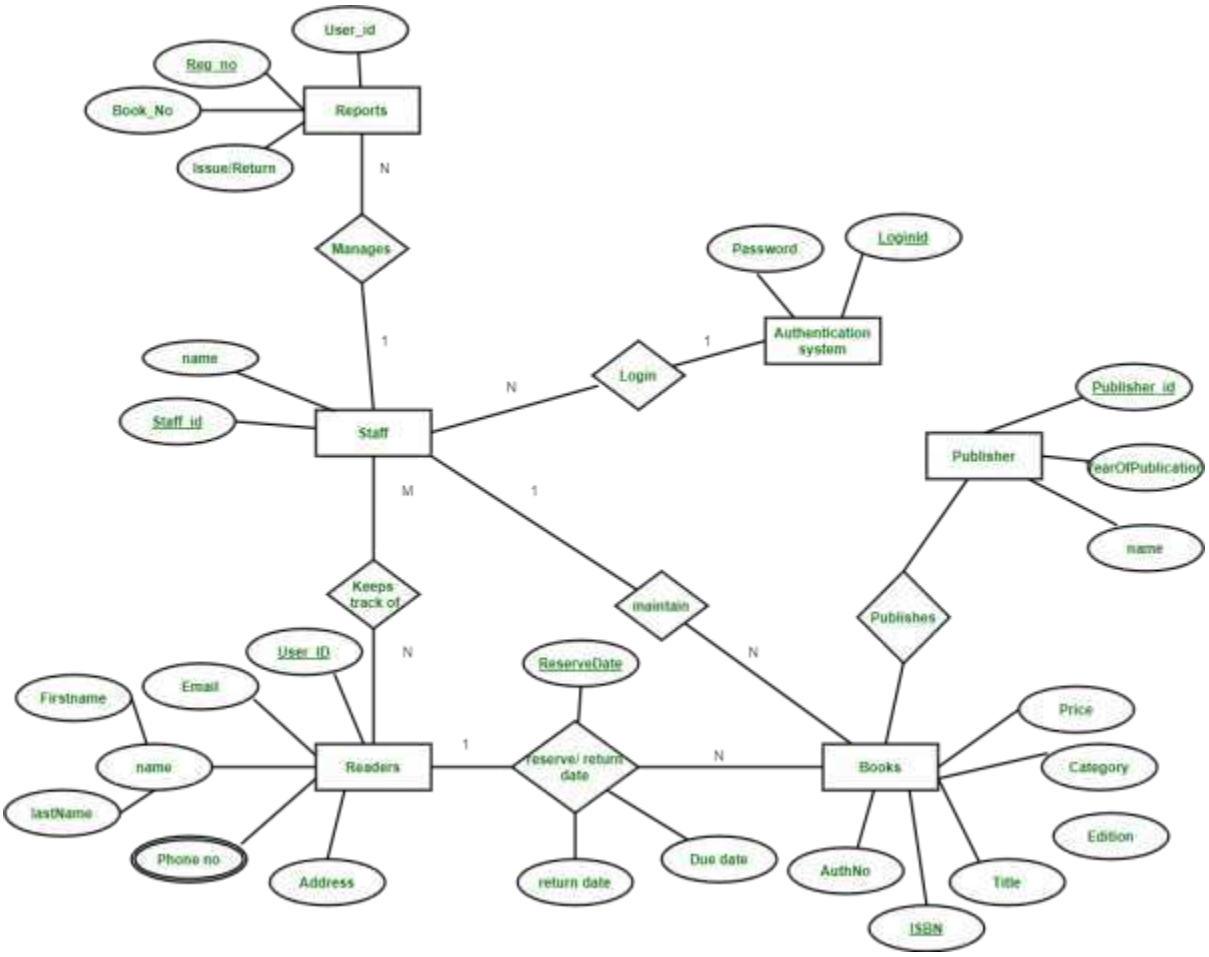
Symbol	Meaning
Double Oval	Multi-valued Attribute
Dashed Oval	Derived Attribute
Underlined Attribute	Key Attribute
Diamond	Relationship
Double Diamond	Identifying Relationship
Single Line	Partial Participation
Double Line	Total Participation

Explanation (Exam-friendly)

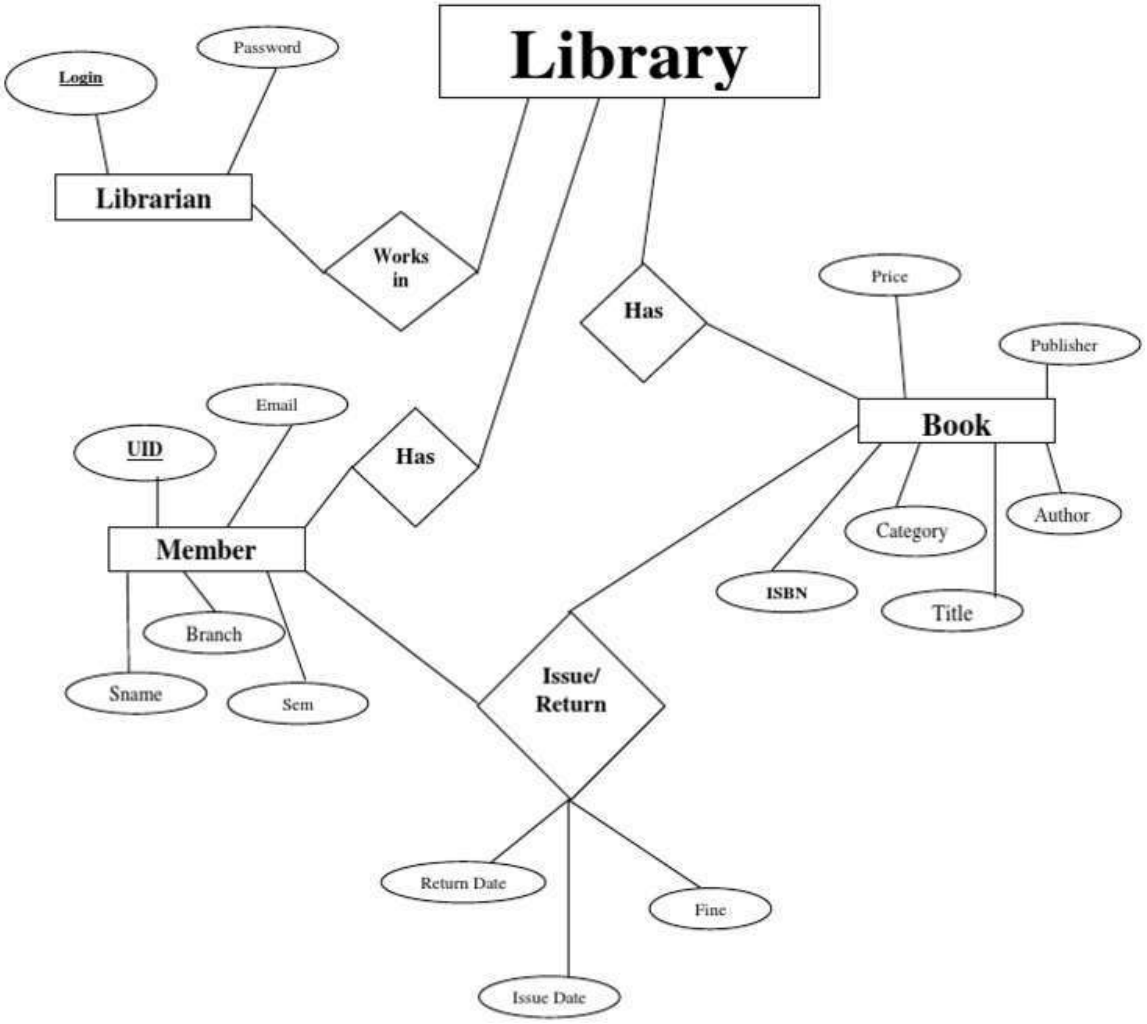
- **Entity:** Represents real-world objects (Student, Book).
- **Attribute:** Describes properties of an entity (Book_ID, Name).
- **Relationship:** Shows association between entities (Issue, Returns).
- **Key Attribute:** Uniquely identifies an entity instance.
- **Weak Entity:** Depends on a strong entity for identification.

---

2. ER Diagram for Library Management System


User_id
Reg_no
Book_No
Issue/Return
Reports
N
Manages
1
Password
Loginid
Authentication system
1
Login
N
name
Staff_id
Staff
M
1
Publisher_id
Publisher
name
Keeps track of
maintain
Publishes
N
N
User_ID
Email
Firstname
name
lastName
Phone no
Address
Readers
1
ReserveDate
Reserve/ return date
return date
Due date
N
Price
Category
Edition
Books
AuthNo
Title
ISBN

ER Diagram: Library Management System



Entities and Attributes

1. **Student**
    - Student_ID *(Key)*
    - Name
    - Department

    - Phone_No
2. **Book**
    - Book_ID *(Key)*
    - Title
    - Author
    - Publisher
3. **Librarian**
    - Librarian_ID *(Key)*
    - Name

Relationships

- **Issue**
    - Between **Student** and **Book**
    - Attributes: Issue_Date, Return_Date
- **Manages**
    - Between **Librarian** and **Book**

Diagram Description (How to draw in exam)

1. Draw **rectangles** for *Student*, *Book*, and *Librarian*.
2. Draw **ovals** for their attributes; underline the key attributes.
3. Connect *Student* and *Book* using a **diamond** named *Issue*.
4. Add attributes *Issue_Date* and *Return_Date* to the *Issue* relationship.
5. Connect *Librarian* to *Book* using a *Manages* relationship.

Conclusion

ER diagram symbols provide a **standardized visual language** for database design. The **Library Management System ER diagram** clearly represents entities, attributes, and relationships, helping in efficient database development and understanding.

3

A) Define and Differentiate Relational Algebra Operators

1) Cartesian Product (×) and 2) Natural Join (⋈)

Taster

identifier	name
g_001	Leila
g_002	Mark
g_003	Luke
g_004	Sarah

Variety

identifier	price_per_kilo	maturity	taste
Red Delicious	3.19	Late Sept.	sweet
Braeburn	3.49	Mid. Oct.	sweet/tart
Gala	3.19	Mid. Sept.	sweet

PRODUCT

Taster_Variety

identifier	name	label	price_per_kilo	maturity	taste
g_001	Leila	Red Delicious	3.19	Late Sept.	sweet
g_001	Leila	Braeburn	3.49	Mid. Oct.	sweet/tart
g_001	Leila	Gala	3.19	Mid. Sept.	sweet
g_002	Mark	Red Delicious	3.19	Late Sept.	sweet
g_002	Mark	Braeburn	3.49	Mid. Oct.	sweet/tart
g_002	Mark	Gala	3.19	Mid. Sept.	sweet
g_003	Luke	Red Delicious	3.19	Late Sept.	sweet
g_003	Luke	Braeburn	3.49	Mid. Oct.	sweet/tart
g_003	Luke	Gala	3.19	Mid. Sept.	sweet
g_004	Sarah	Red Delicious	3.19	Late Sept.	sweet
g_004	Sarah	Braeburn	3.49	Mid. Oct.	sweet/tart
g_004	Sarah	Gala	3.19	Mid. Sept.	sweet

- Relations r, s:

A	B	C	D
α	1	α	a
β	2	γ	a
γ	4	β	b
α	1	γ	a
δ	2	β	b

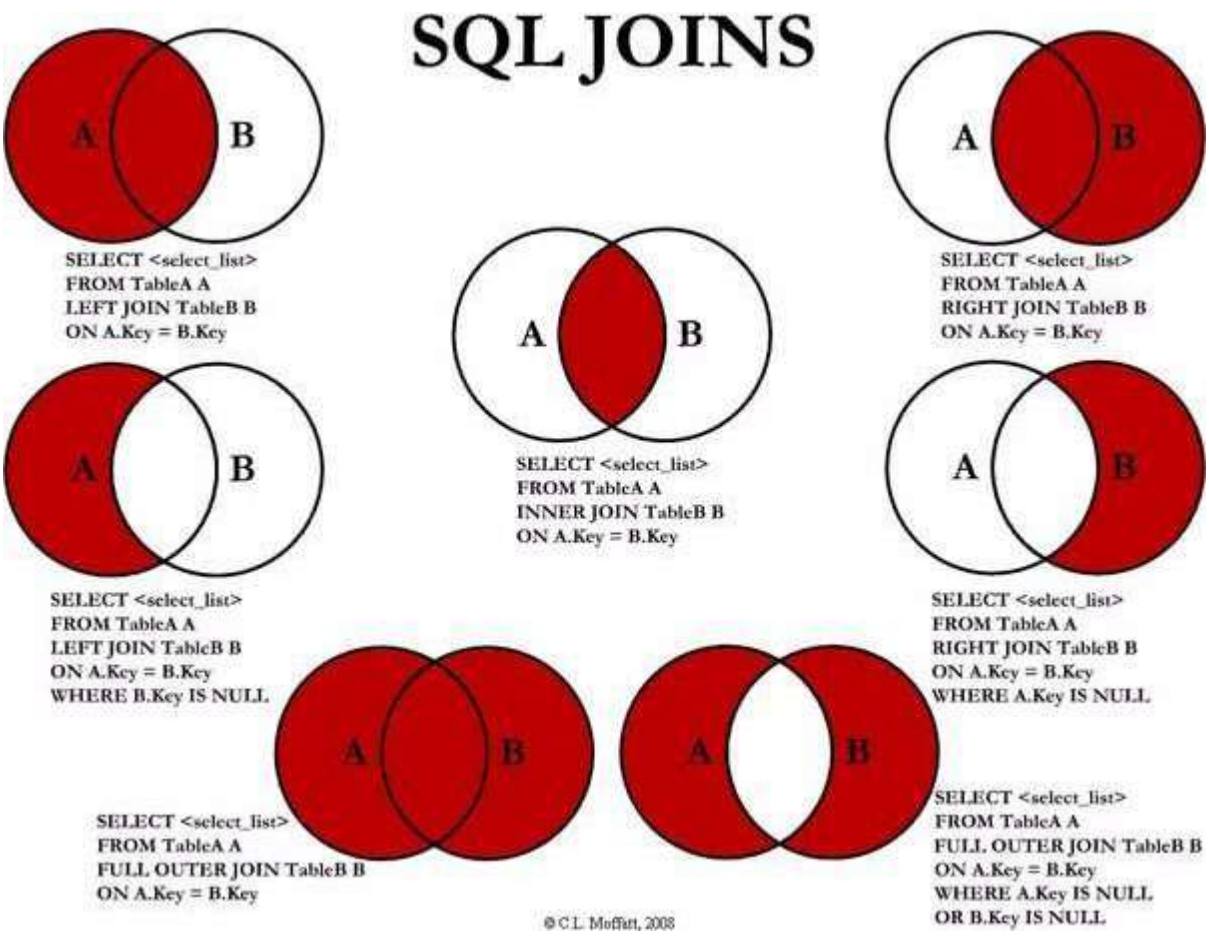
r

B	D	E
1	a	α
3	a	β
1	a	γ
2	b	δ
3	b	ε

s

- r ⋈ s

A	B	C	D	E
α	1	α	a	α
α	1	α	a	γ
α	1	γ	a	α
α	1	γ	a	γ
δ	2	β	b	δ



Introduction

In **relational algebra**, operators are used to retrieve and manipulate data from relations (tables). **Cartesian product** and **Natural join** are fundamental operators used to combine relations, but they differ significantly in purpose and result size.

---

1. Cartesian Product (×)

Definition

The **Cartesian product** of two relations **R × S** produces a new relation that contains **all possible combinations of tuples** from relation R with tuples from relation S.

Key Points

- No condition is applied.
- Number of tuples = |R| × |S|.
- Attribute set = attributes of R + attributes of S.

Example

Let
Student(Sid, Name)

Sid	Name
1	Amit
2	Riya

Course(Cid, Cname)

Cid	Cname
C1	DBMS
C2	CN

Cartesian Product:
Student × Course

Sid	Name	Cid	Cname
1	Amit	C1	DBMS
1	Amit	C2	CN
2	Riya	C1	DBMS
2	Riya	C2	CN

---

2. Natural Join (⋈)

Definition

A **Natural join** combines two relations by **matching tuples on all common attributes with the same name** and removes duplicate columns.

Key Points

- Join condition is implicit (based on common attributes).
- Reduces unnecessary tuples.
- Duplicate attributes are removed.

Example

Let
Student(Sid, Name, Dept)

Sid	Name	Dept
1	Amit	CSE

Sid	Name	Dept
2	Riya	IT

Enroll(Sid, Course)

Sid	Course
1	DBMS
2	CN

Natural Join:
Student ⋈ Enroll

Sid	Name	Dept	Course
1	Amit	CSE	DBMS
2	Riya	IT	CN

Difference Between Cartesian Product and Natural Join

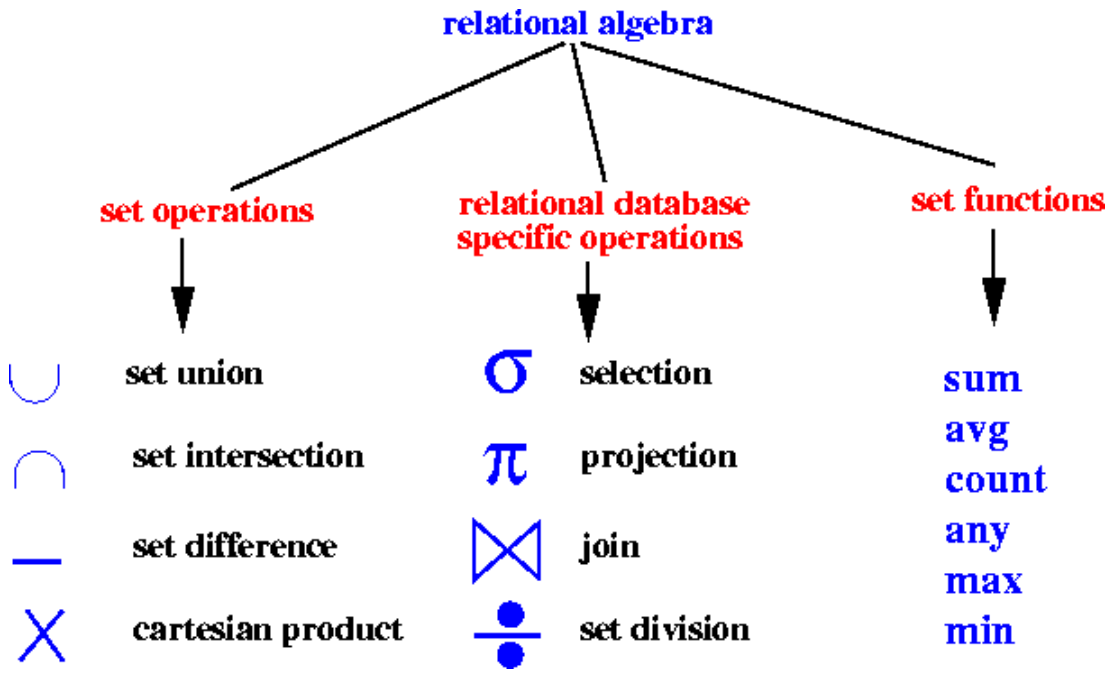
Aspect	Cartesian Product (×)	Natural Join (⋈)
Join Condition	No condition	Based on common attributes
Result Size	Very large	Smaller, meaningful
Duplicate Attributes	Not removed	Removed
Usage	Intermediate operation	Final data retrieval
Tuples	All combinations	Only matching tuples

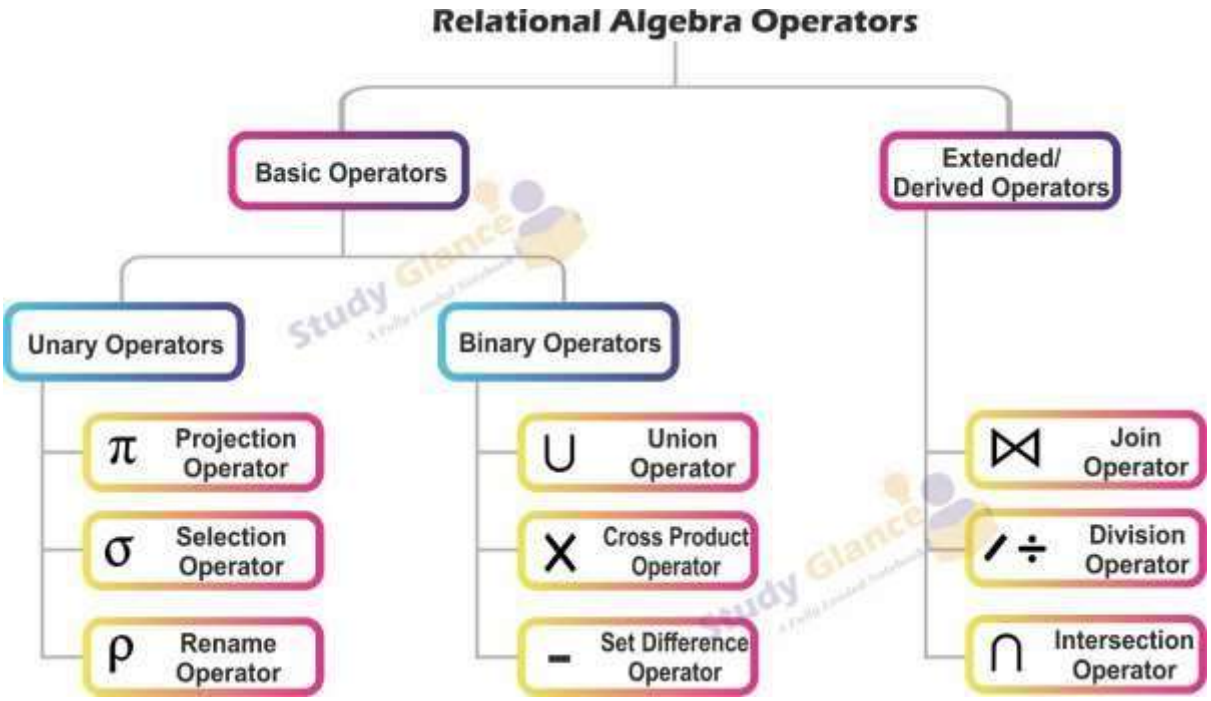
Conclusion

The **Cartesian product** creates all possible tuple combinations and is mainly used as an intermediate step in joins, while the **Natural join** produces meaningful results by combining related tuples based on common attributes. Natural join is preferred in practical database queries.

4

Relational Algebra: Definition and Basic Operations (with Symbols)





Introduction

Relational Algebra is a **procedural query language** used to define operations on relations (tables) in a relational database. It provides a **formal foundation** for query processing and optimization in DBMS by specifying **how** data is retrieved using a sequence of operations.

Definition

Relational Algebra consists of a set of **operations** that take one or more relations as input and produce a **new relation** as output. These operations are closed over relations.

Basic Operations of Relational Algebra

1. Selection (σ)

- **Purpose:** Selects rows (tuples) that satisfy a given condition.
- **Symbol:** σ (sigma)

Syntax:
σ_{condition}(Relation)

Example:
σ_{Dept = 'CSE'}(STUDENT)

---

2. Projection (π)

- **Purpose:** Selects specific columns (attributes) from a relation.
- **Symbol:** π (pi)

Syntax:
π_{attribute-list}(Relation)

Example:
π_{Name, Dept}(STUDENT)

---

3. Union (∪)

- **Purpose:** Combines tuples from two relations.
- **Condition:** Relations must be **union-compatible**.
- **Symbol:** ∪

Syntax:
R ∪ S

Example:
STUDENT ∪ ALUMNI

---

4. Set Difference (−)

- **Purpose:** Returns tuples present in one relation but not in another.
- **Symbol:** −

Syntax:
R − S

Example:
STUDENT − DROPOUT

---

5. Cartesian Product (×)

- **Purpose:** Combines each tuple of one relation with every tuple of another.
- **Symbol:** ×

Syntax:
R × S

Example:
STUDENT × COURSE

---

6. Rename (ρ)

- **Purpose:** Renames relation or attributes.
- **Symbol:** ρ (rho)

Syntax:
ρ(NewName)(Relation)

Example:
ρ(S)(STUDENT)

Summary Table of Basic Operations

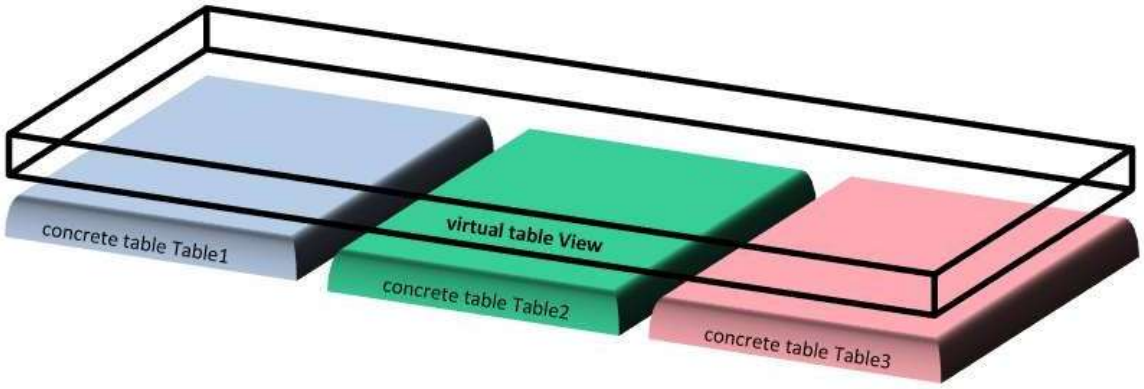
Operation	Symbol	Description
Selection	σ	Select rows
Projection	π	Select columns
Union	∪	Combine relations
Set Difference	−	Subtract relations
Cartesian Product	×	Pair all tuples
Rename	ρ	Rename relation

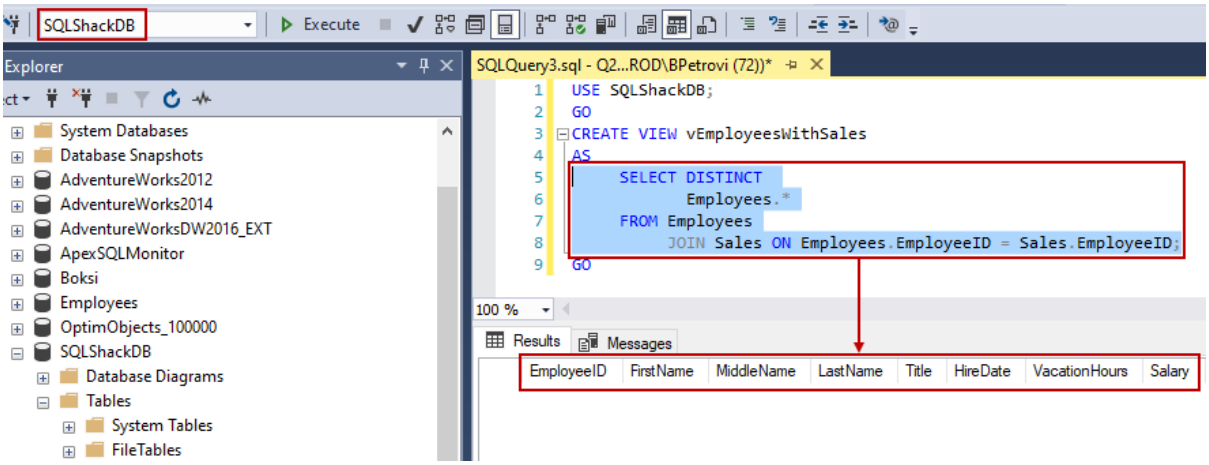
Conclusion

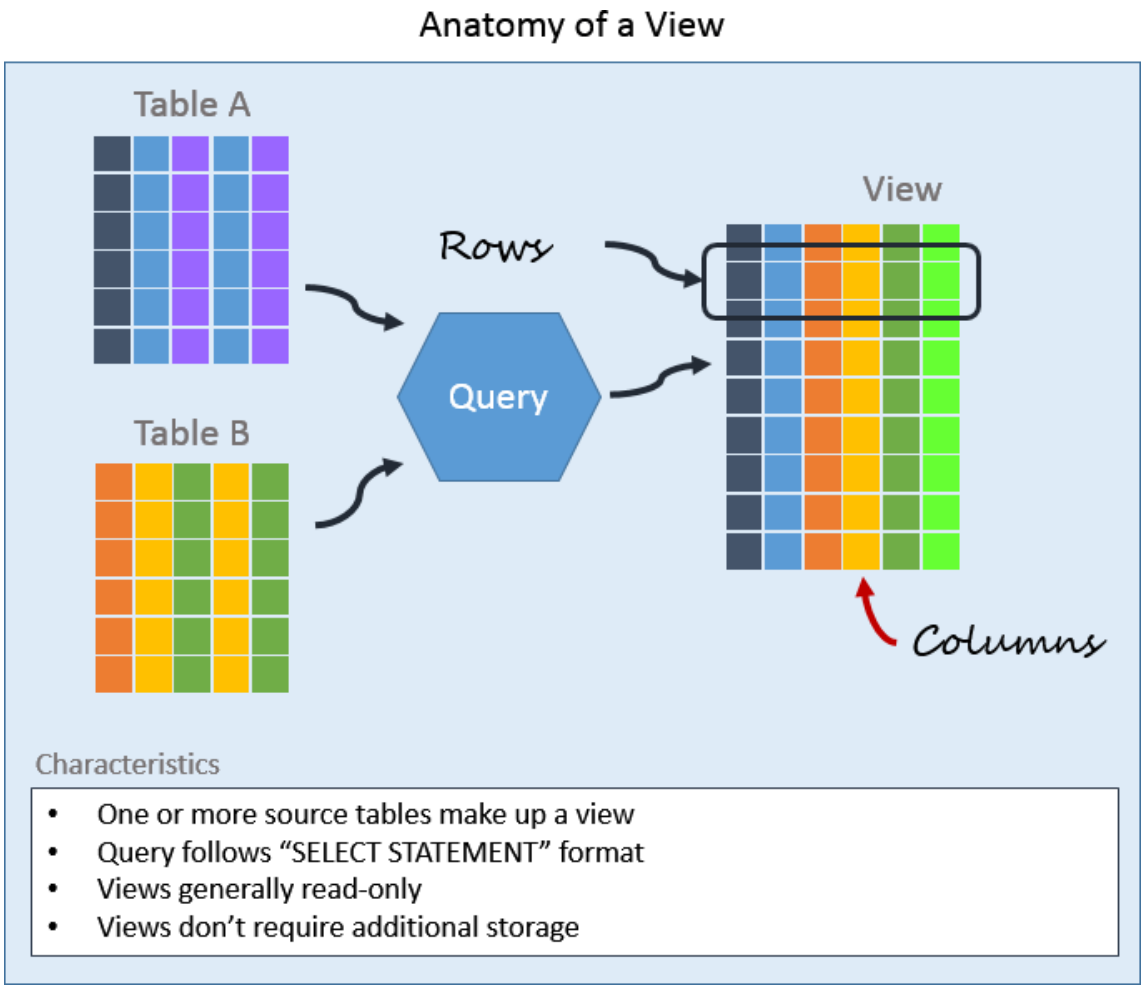
Relational Algebra provides a **theoretical and operational framework** for querying relational databases. Understanding its **basic operations and symbols** is essential for database design, query optimization, and advanced DBMS concepts.

5

View in SQL: Definition, Advantages, and Detailed Explanation with Example







Introduction

In a database system, users often do not require access to the complete database. They usually need **specific data in a simplified form**. To fulfill this requirement, SQL provides a powerful feature called a **View**. A view helps in presenting data logically without storing it physically and plays a crucial role in **security, abstraction, and query simplification**.

What is a View?

A **View** is a **virtual table** that is created using a **SELECT query** on one or more base tables. It does not store data itself; instead, it dynamically displays data derived from the underlying tables whenever it is accessed.

–In simple words, a view is a **stored query with a name**.

Characteristics of a View

- A view contains **rows and columns like a table**

- Data is **not stored physically**
- Changes in base tables are automatically reflected in the view
- A view can be created from **one or multiple tables**

Advantages of View

1. Data Security

Views restrict access to sensitive information.

- Users can see only required columns
- Confidential data (salary, password) can be hidden

Example:
A clerk can view student names but not their marks.

---

2. Simplicity and Ease of Use

Views simplify complex queries.

- Complex joins and conditions are hidden
- Users can query the view as a normal table

---

3. Data Abstraction

Views hide the complexity of database structure.

- Logical independence from base tables
- Changes in table structure do not affect users

---

4. Consistency

Views provide a consistent representation of data.

- Ensures uniform data access
- Useful when multiple users access data

---

5. Reduced Storage Requirement

- Views do not occupy extra storage
- Only query definition is stored

Explanation of Views in SQL with Example

Base Table: STUDENT

Student_ID	Name	Department	Marks
1	Amit	CSE	85
2	Riya	IT	78
3	Neha	CSE	92

Creating a View

Create a view to display only CSE students:

```
CREATE VIEW CSE_Students AS
SELECT Student_ID, Name, Marks
FROM STUDENT
WHERE Department = 'CSE';
```

View: CSE_Students

Student_ID	Name	Marks
1	Amit	85
3	Neha	92

Using a View

```
SELECT * FROM CSE_Students;
```

Updating Data Through View

If the view satisfies update conditions, changes affect the base table:

```
UPDATE CSE_Students
SET Marks = 90
WHERE Student_ID = 1;
```

Dropping a View

```
DROP VIEW CSE_Students;
```

Types of Views (Additional Points for Scoring)

1. Simple View

- Created from a single table
- No GROUP BY or aggregate functions

2. Complex View

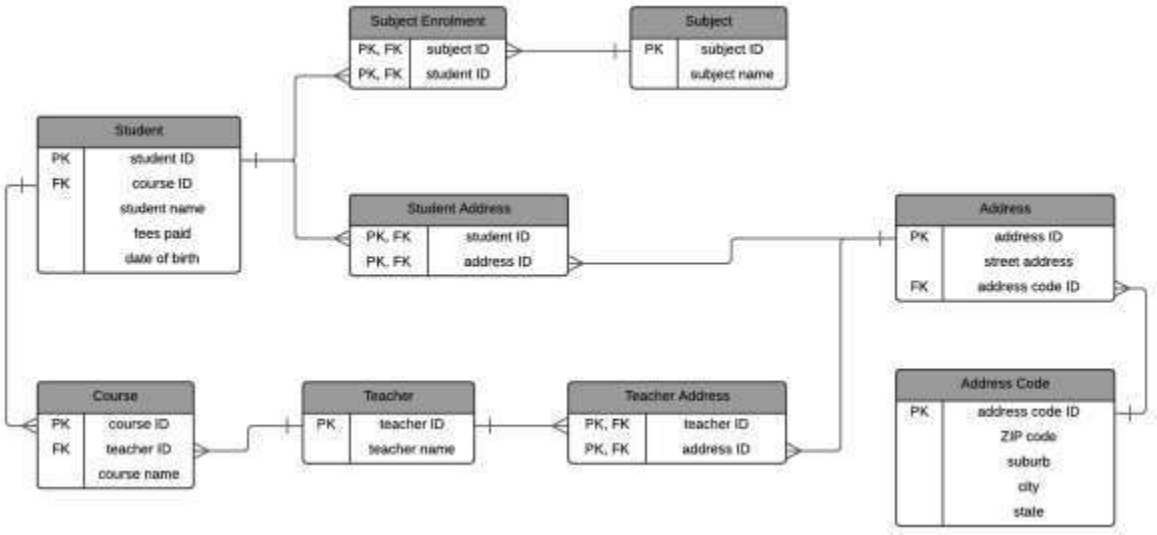
- Created from multiple tables
- Uses JOIN, GROUP BY, aggregate functions

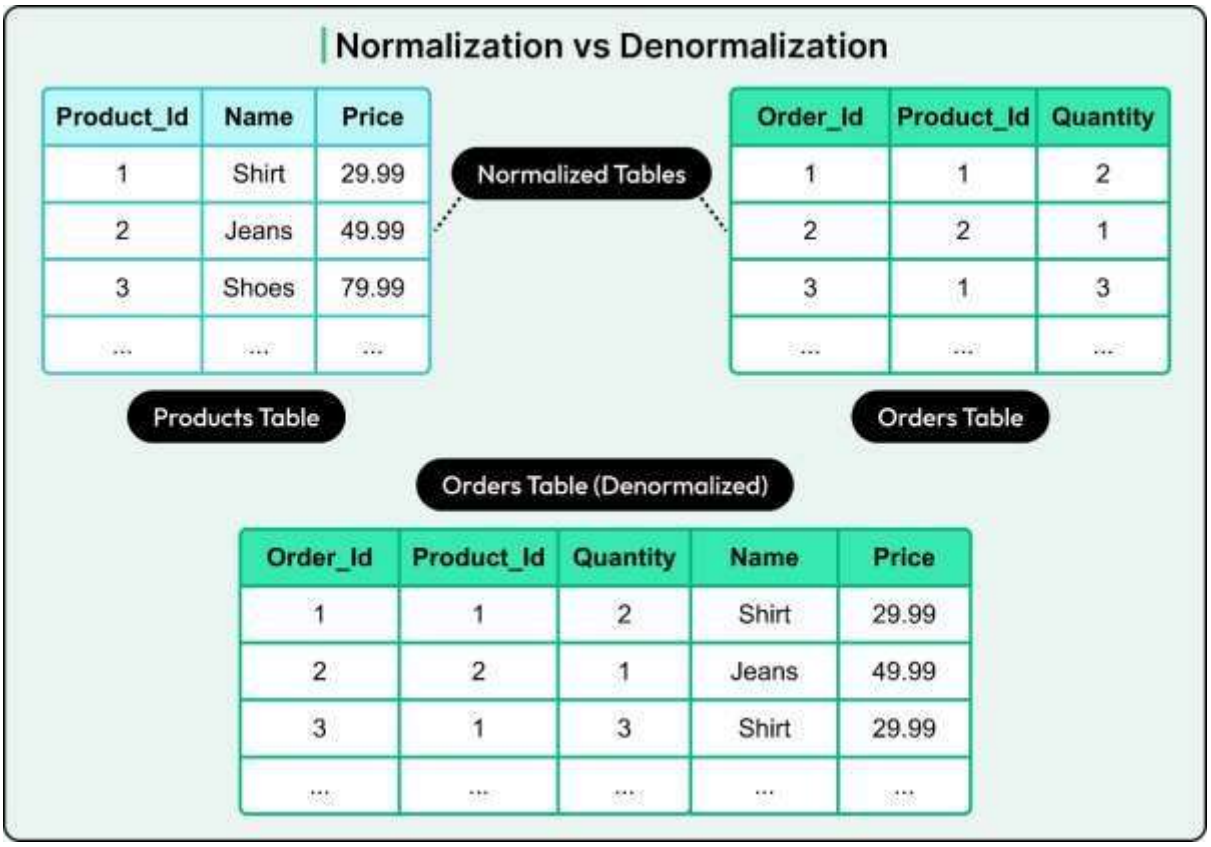
Conclusion

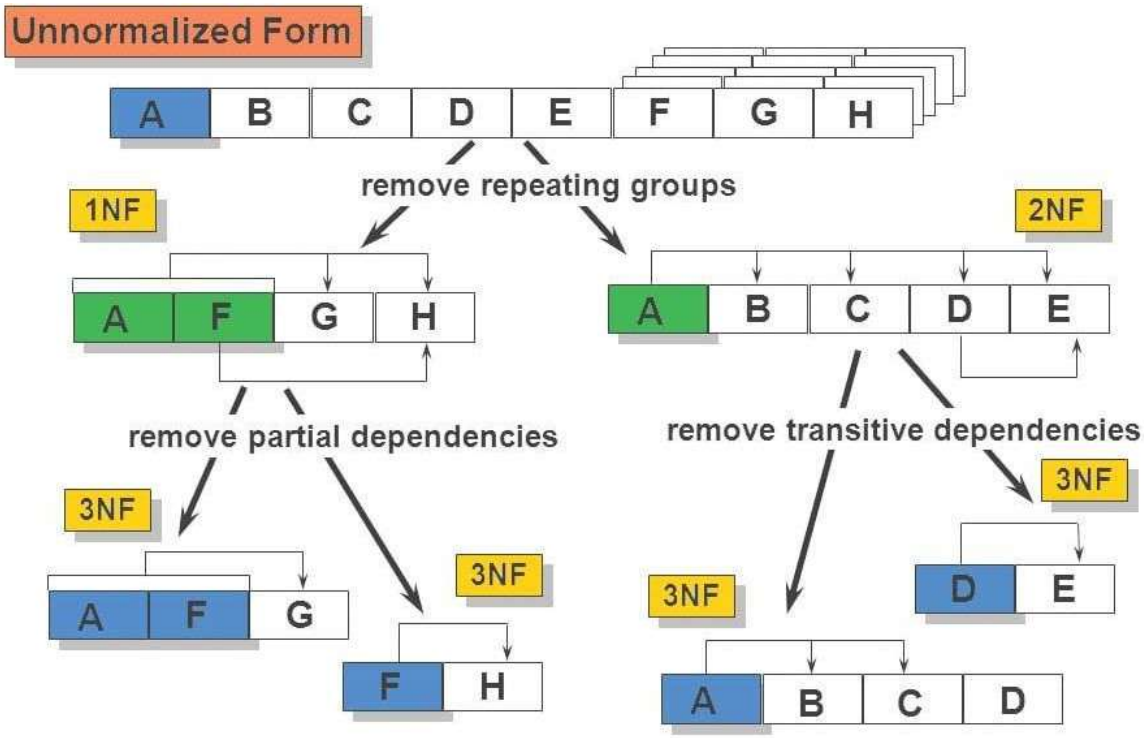
A **View** is an essential database object that provides **security, abstraction, simplicity, and consistency**. By using views, database systems become easier to manage and safer to access, especially in large multi-user environments. Views are widely used in real-world applications to control data visibility and reduce query complexity.

6

Normalizati7on: Definition and Need for Normalizing Database Tables







Introduction

In database design, storing data without proper structure often leads to **data redundancy, inconsistency, and update anomalies**. To overcome these problems, the concept of

Normalization is used. Normalization is a systematic approach to organize data in a database efficiently.

What is Normalization?

Normalization is the process of **organizing data in a database** to minimize redundancy and dependency by dividing large tables into smaller, well-structured tables and establishing relationships between them.

– In simple words, normalization ensures that **each fact is stored only once** in the database.

Definition

Normalization is a database design technique that uses a set of rules called **normal forms** (1NF, 2NF, 3NF, etc.) to reduce data duplication and maintain data integrity.

Why Do We Need to Normalize Database Tables?

1. To Reduce Data Redundancy

- Avoids repetition of the same data in multiple rows
- Saves storage space

Example:
Storing student details repeatedly for every subject causes redundancy.

---

2. To Eliminate Update Anomalies

Normalization prevents problems during data modification.

- **Insertion Anomaly:** Unable to insert data without other data
- **Deletion Anomaly:** Deleting data unintentionally removes important information
- **Update Anomaly:** Updating data in one place but not in others

---

3. To Improve Data Consistency

- Ensures uniform and accurate data
- Avoids conflicting values for the same attribute

---

4. To Improve Data Integrity

- Maintains logical relationships using keys
- Reduces chances of invalid data

---

5. To Simplify Database Design

- Tables become smaller and more manageable
- Easier to understand and maintain

---

6. To Improve Query Performance

- Efficient indexing
- Faster retrieval in structured tables

Example (Before and After Normalization)

Unnormalized Table

Student_ID	Student_Name	Subject	Faculty
1	Amit	DBMS	Rao
1	Amit	CN	Sharma

+ Student data is repeated.

Normalized Tables

Student Table

Student_ID	Student_Name
1	Amit

Subject Table

Subject	Faculty
DBMS	Rao
CN	Sharma

Enrollment Table

Student_ID	Subject
1	DBMS
1	CN

✓ Redundancy removed and data properly structured.

Conclusion

Normalization is an essential process in database design that ensures **minimum redundancy, maximum data integrity, and efficient data management**. By normalizing database tables, we achieve a well-organized, consistent, and reliable database structure suitable for real-world applications.

7

SQL Syntax with Examples

Introduction

SQL (Structured Query Language) provides different commands to **retrieve, modify structure, and update data** in database tables. Among them, **SELECT**, **ALTER**, and **UPDATE** are the most commonly used commands in day-to-day database operations.

---

1. SELECT Statement

Purpose

The **SELECT** statement is used to **retrieve data** from one or more tables in a database.

Syntax

SELECT column1, column2, ...

FROM table_name

WHERE condition;

Example

Consider a table **STUDENT**

Roll_No	Name	Dept	Marks
1	Amit	CSE	85
2	Riya	IT	78

SELECT Name, Marks

FROM STUDENT

WHERE Dept = 'CSE';

Output

Name	Marks
Amit	85

---

2. ALTER Statement

Purpose

The **ALTER** statement is used to **modify the structure of an existing table**, such as adding, deleting, or modifying columns.

Syntax

ALTER TABLE table_name

ADD column_name datatype;

OR

ALTER TABLE table_name

MODIFY column_name datatype;

Example

Add a new column **Email** to STUDENT table:

ALTER TABLE STUDENT

ADD Email VARCHAR(50);

---

3. UPDATE Statement

Purpose

The **UPDATE** statement is used to **modify existing records** in a table.

Syntax

UPDATE table_name

SET column_name = value

WHERE condition;

Example

Update marks of student with Roll_No = 1:

UPDATE STUDENT

SET Marks = 90

WHERE Roll_No = 1;

Summary Table

SQL Command	Use
SELECT	Retrieve data from table

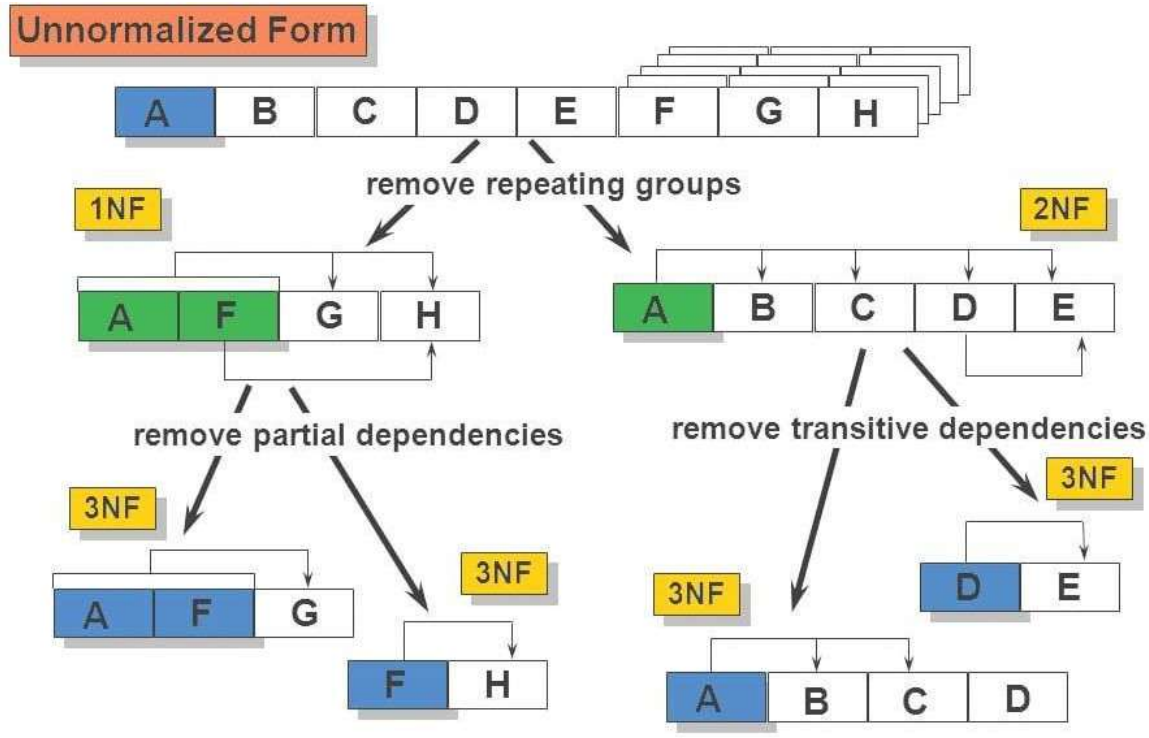
SQL Command	Use
ALTER	Change table structure
UPDATE	Modify existing data

Conclusion

The **SELECT**, **ALTER**, and **UPDATE** commands are essential SQL statements used for **data retrieval, table modification, and data updating**. Understanding their syntax and usage is fundamental for effective database management

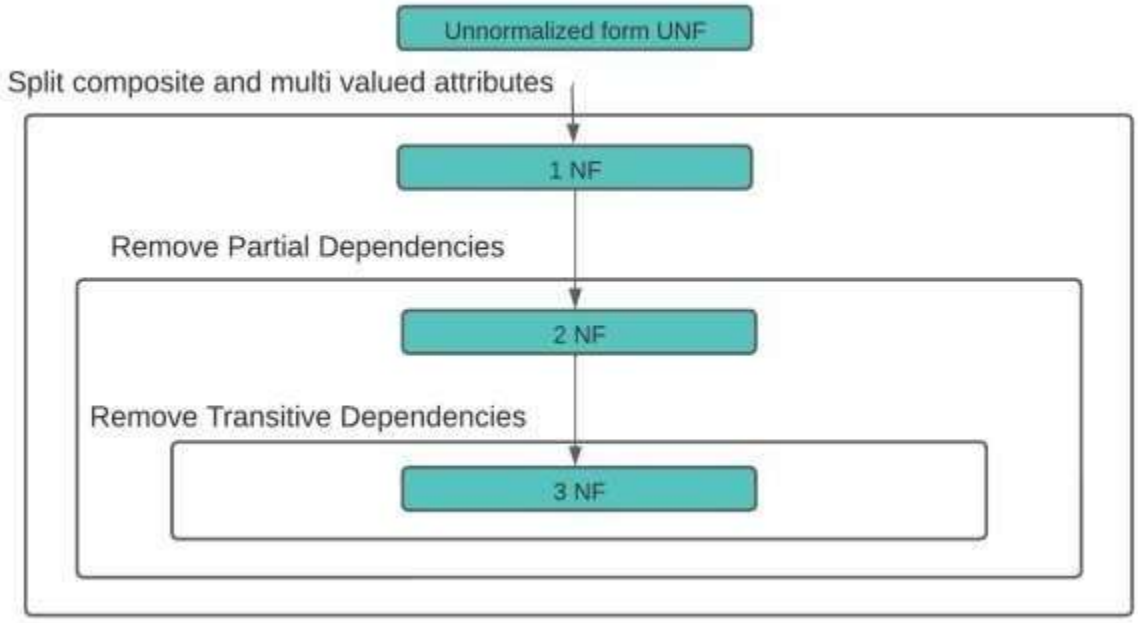
8

First, Second and Third Normal Form (1NF, 2NF, 3NF) with Examples



Number	Student	Book number	Address	Subject	Teacher	Department	Mark
1	Johnson J.	2535	London	Physics	Trump D.	Natural science subjects	4
2	Johnson J.	2535	London	Chemistry	Wilson G.	Natural science subjects	3
3	Matthew S.	2580	Manchester	Physics	Trump D.	Natural science subjects	4
4	Holmes M.	2676	Liverpool	Physics	Trump D.	Natural science subjects	5
5	Holmes M.	2676	Liverpool	Chemistry	Wilson G.	Natural science subjects	3
6	Johnson J.	2535	London	Informatics	Wonder S.	Mathematical subjects	4
7	???	???	???	Mathematics	Trump D.	Mathematical subjects	???

Subject	Teacher	Department
Mathematics	Jackson	Mathematical subjects



Introduction

In database design, improper structuring of tables leads to **data redundancy, inconsistency, and anomalies**. To overcome these issues, **Normalization** is applied. Normalization organizes data into well-structured tables using a series of rules called **Normal Forms**. The most commonly used normal forms are **First Normal Form (1NF), Second Normal Form (2NF), and Third Normal Form (3NF)**.

---

1. First Normal Form (1NF)

Definition

A relation is said to be in **First Normal Form (1NF)** if:

- All attributes contain **atomic (indivisible) values**
- No repeating groups or multi-valued attributes exist
- Each record can be uniquely identified

Example (Before 1NF – Not Normalized)

Student_ID	Name	Subjects
1	Amit	DBMS, CN
2	Riya	OS

Problem:

- Subjects column contains multiple values (not atomic)

After Applying 1NF

Student_ID	Name	Subject
1	Amit	DBMS
1	Amit	CN
2	Riya	OS

✓ All attributes now contain atomic values.

---

2. Second Normal Form (2NF)

Definition

A relation is in **Second Normal Form (2NF)** if:

- It is already in **1NF**
- **No partial dependency** exists
  (i.e., no non-key attribute depends on part of a composite primary key)

Example (In 1NF but Not in 2NF)

Student_ID	Subject	Student_Name	Faculty
1	DBMS	Amit	Rao

Student_ID	Subject	Student_Name	Faculty
1	CN	Amit	Sharma

Primary Key: (Student_ID, Subject)

➕ Problems:

- Student_Name depends only on Student_ID
- Faculty depends only on Subject
  → **Partial dependency exists**

After Applying 2NF

Student Table

Student_ID	Student_Name
1	Amit

Subject Table

Subject	Faculty
DBMS	Rao
CN	Sharma

Enrollment Table

Student_ID	Subject
1	DBMS
1	CN

✓ Partial dependency removed.

---

3. Third Normal Form (3NF)

Definition

A relation is in **Third Normal Form (3NF)** if:

- It is already in **2NF**
- **No transitive dependency** exists
  (i.e., non-key attributes should not depend on other non-key attributes)

Example (In 2NF but Not in 3NF)

Emp_ID Emp_Name Dept_ID Dept_Name

➕ Problem:

- Dept_Name depends on Dept_ID
- Dept_ID depends on Emp_ID
  → **Transitive dependency exists**

After Applying 3NF

Employee Table

Emp_ID Emp_Name Dept_ID

Department Table

Dept_ID Dept_Name

✓ Transitive dependency eliminated.

Summary of Normal Forms

Normal Form	Condition
1NF	Atomic values, no repeating groups
2NF	No partial dependency
3NF	No transitive dependency

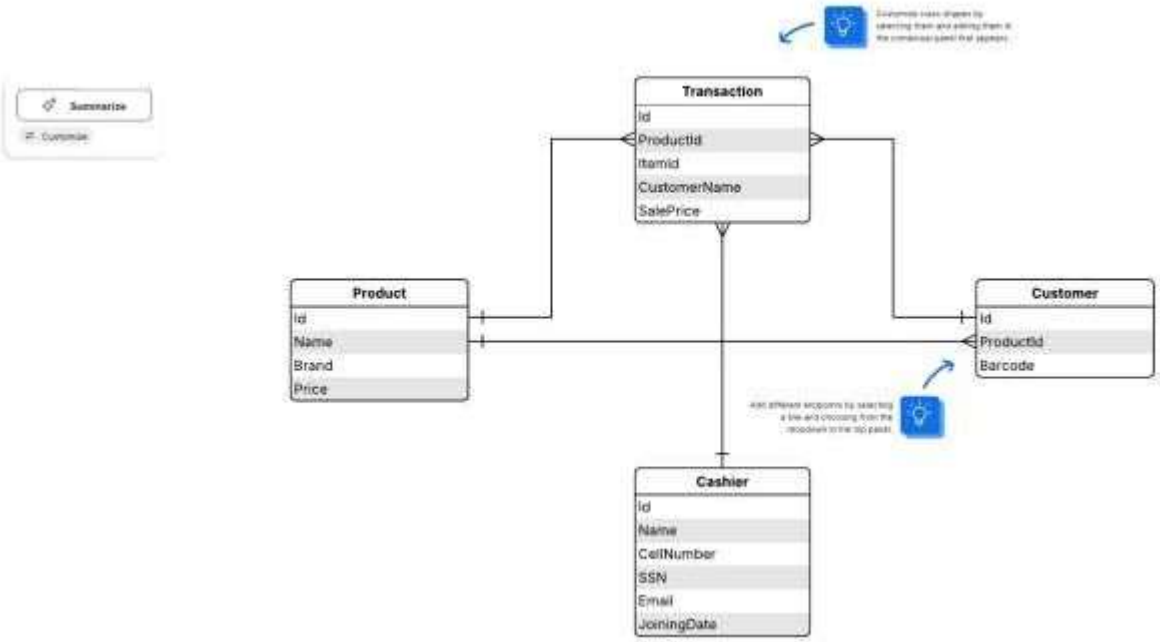
Conclusion

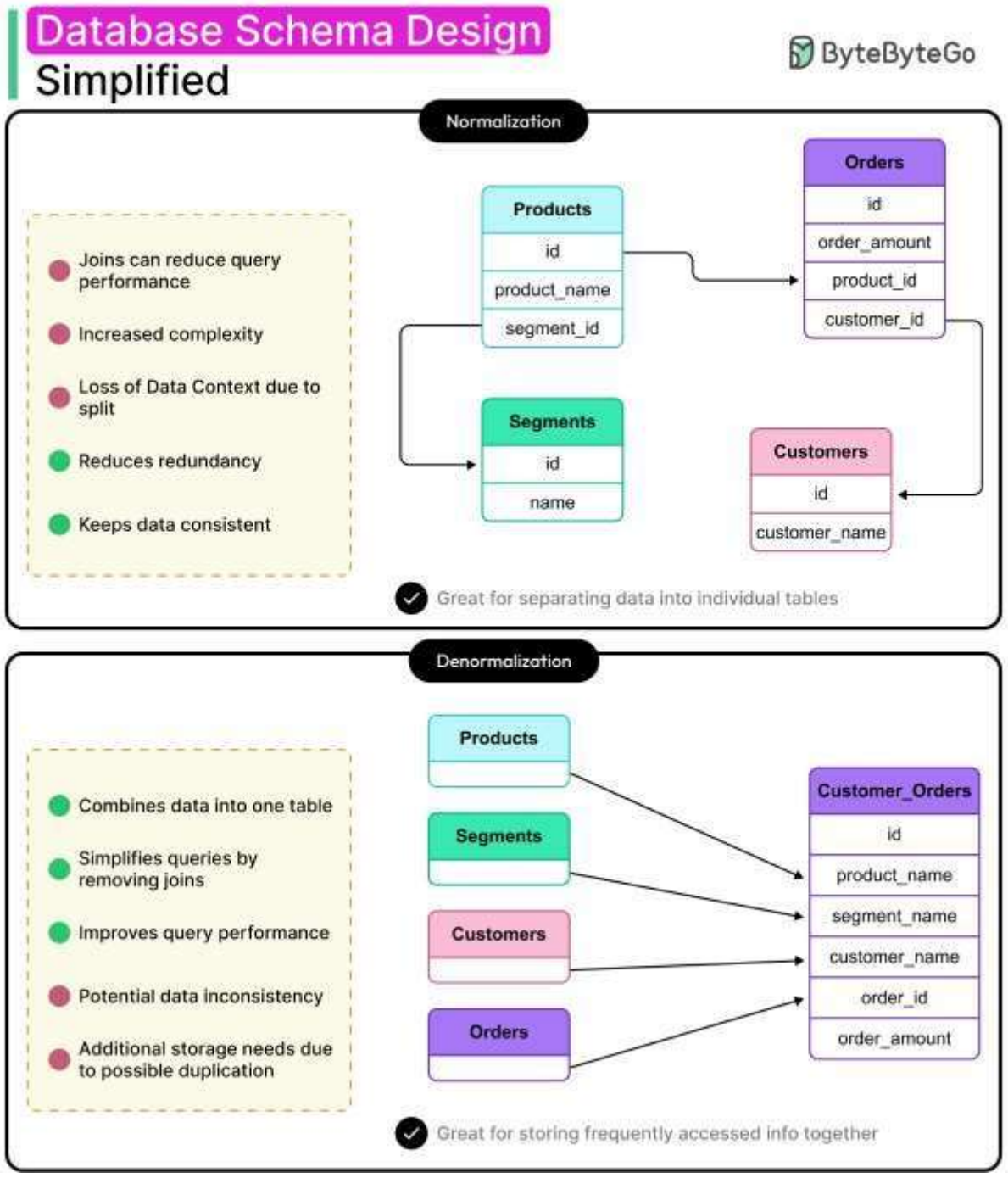
Normalization improves database quality by **eliminating redundancy, preventing anomalies, and ensuring data integrity**.

- **1NF** removes repeating groups
- **2NF** removes partial dependency
- **3NF** removes transitive dependency

Together, these normal forms help in designing **efficient, reliable, and scalable databases**.

Features of a Good Relational Database Design


Summarize
Customize
Transaction
Id
ProductId
ItemId
CustomerName
SalePrice
Product
Id
Name
Brand
Price
Customer
Id
ProductId
Barcode
Cashier
Id
Name
CellNumber
SSN
Email
JoiningDate



Introduction

A **good relational database design** ensures that data is stored in a structured, consistent, and efficient manner. Proper design reduces redundancy, avoids anomalies, improves performance, and makes the database easy to maintain and scale. The quality of a database largely depends on how well its relations (tables) are designed.

Features of a Good Relational Design

1. Minimal Data Redundancy

- Same data should not be stored repeatedly in multiple tables.
- Reduces storage space and avoids inconsistency.

Example:
Student details should be stored once, not repeated for every subject.

---

2. Elimination of Update Anomalies

A good design avoids:

- **Insertion anomaly**
- **Deletion anomaly**
- **Update anomaly**

This is achieved mainly through **normalization**.

---

3. Data Integrity

- Ensures correctness and validity of data.
- Maintained using **primary keys**, **foreign keys**, and **constraints**.

Example:
A foreign key ensures that a student cannot enroll in a non-existing course.

---

4. Proper Use of Keys

- Every table should have a **primary key**.
- Relationships between tables should be defined using **foreign keys**.

This uniquely identifies records and maintains relationships.

---

5. Well-Defined Relationships

- Relationships between entities should be clear and meaningful.
- Proper cardinality (one-to-one, one-to-many, many-to-many) should be defined.

---

6. Normalized Structure

- Tables should be normalized at least up to **Third Normal Form (3NF)**.
- Removes partial and transitive dependencies.

---

7. Simplicity and Clarity

- Table structures should be easy to understand.
- Attribute names should be meaningful and self-explanatory.

Example:
Use Student_ID instead of SID1.

---

8. Flexibility and Scalability

- Design should allow future changes without major restructuring.
- New attributes or tables can be added easily.

G. Efficient Query Performance

- Avoids unnecessary joins.
- Supports indexing for faster data retrieval.

---

10. Logical Data Independence

- Changes in database structure should not affect application programs.
- Achieved through abstraction and proper schema design.

Summary Table

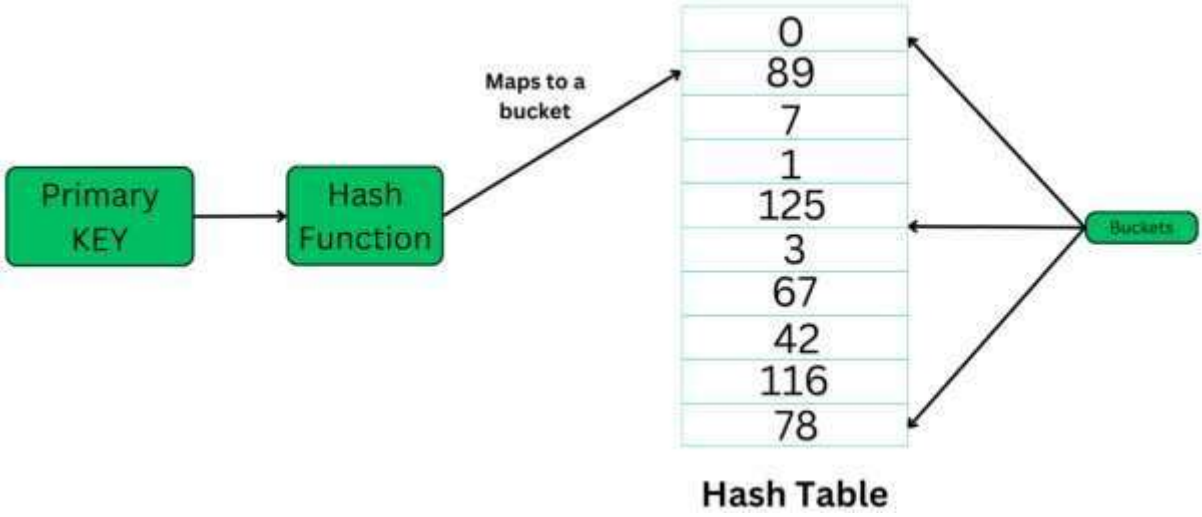
Feature	Purpose
Minimal redundancy	Saves space, avoids inconsistency
No anomalies	Reliable data operations
Data integrity	Correct and valid data
Proper keys	Unique identification
Normalization	Efficient structure
Simplicity	Easy maintenance
Scalability	Future growth
Performance	Faster queries

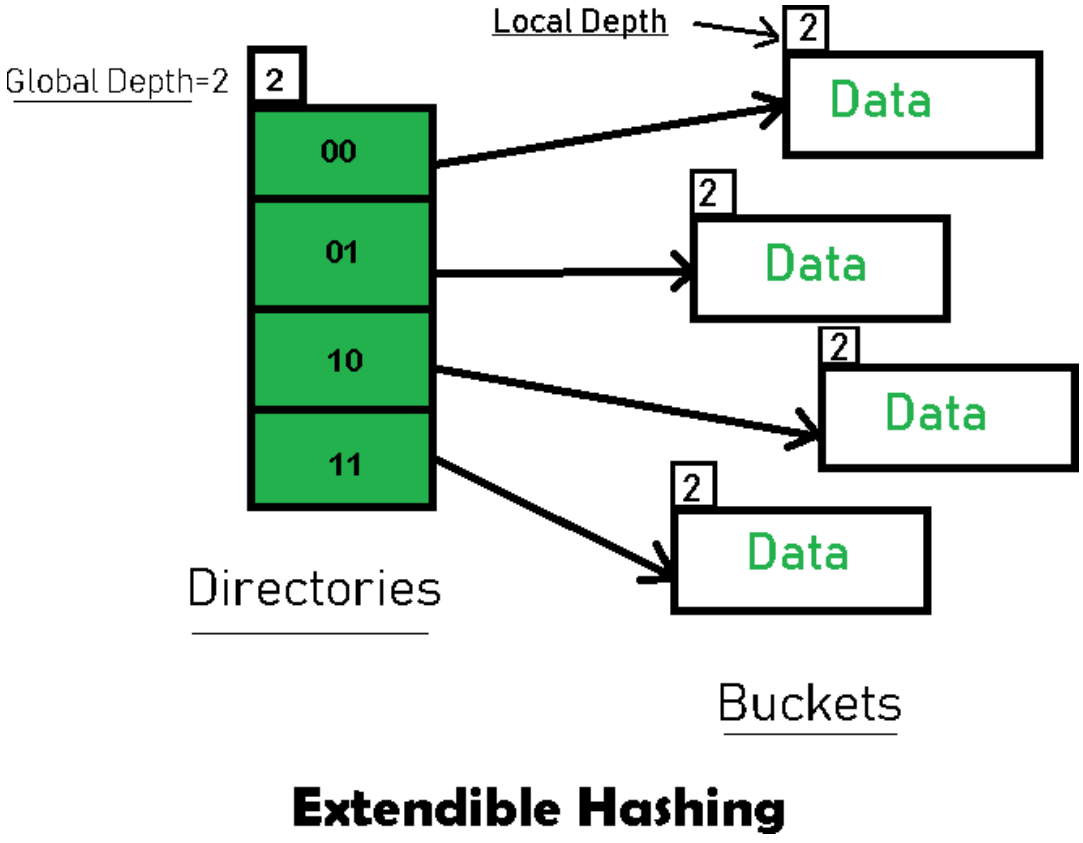
Conclusion

A good relational database design ensures **accuracy, consistency, efficiency, and maintainability** of data. By minimizing redundancy, enforcing integrity, using proper keys, and following normalization principles, a database becomes reliable and suitable for real-world applications.

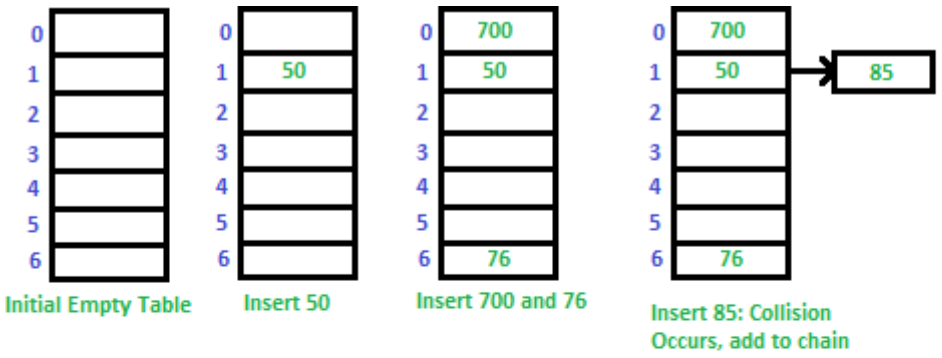
10

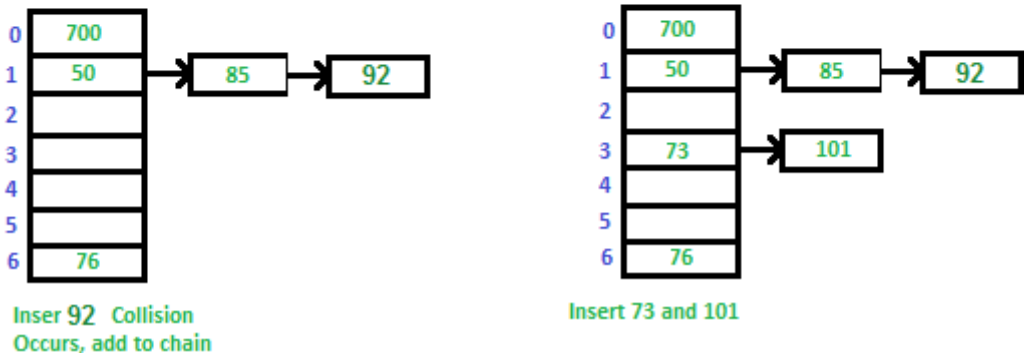
Static Hashing and Dynamic Hashing

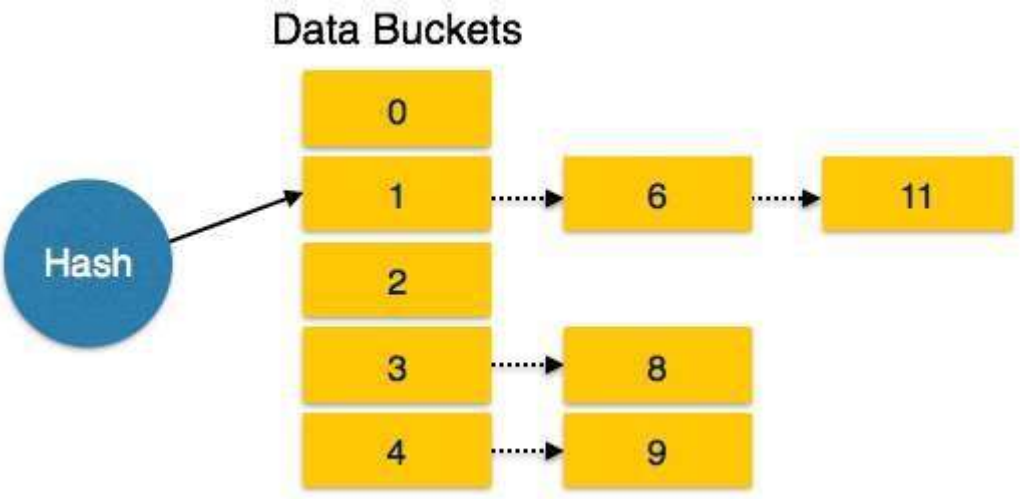




Extendible Hashing







Introduction

Hashing is a file organization technique used to **store and retrieve records quickly** by computing the address of a record using a **hash function**. Based on how the hash structure adapts to data growth, hashing is classified into **Static Hashing** and **Dynamic Hashing**.

---

1. Static Hashing

Definition

In **Static Hashing**, the **number of buckets is fixed** at the time of file creation. The hash function always maps a key to the **same bucket address**.

Working

- A hash function h(key) generates a bucket number.
- Each bucket can store a limited number of records.
- When a bucket overflows, **overflow handling** is required.

Overflow Handling Methods

- **Overflow chaining** (linked list of overflow buckets)
- **Open addressing**

Example

If number of buckets = 10
Hash function: h(key) = key mod 10

For keys: 12, 22, 32
All map to bucket 2 → causes overflow.

Advantages

- Simple to implement
- Fast access when data size is stable

Disadvantages

- Performance degrades with data growth
- Overflow chains increase access time
- Not suitable for frequently growing files

---

2. Dynamic Hashing

Definition

In **Dynamic Hashing**, the hash structure **dynamically changes** with database growth or shrinkage. Buckets are **split or merged** as needed to avoid overflow.

Working (Extendible Hashing – common method)

- Uses a **directory** that points to buckets
- Hash value bits decide bucket placement
- When overflow occurs:
    - Bucket is split
    - Directory may be expanded

Example

- Initially, directory size = 2
- When a bucket overflows, directory size doubles
- Records are redistributed based on additional hash bits

Advantages

- No long overflow chains
- Efficient even when data size changes
- Better performance for large databases

Disadvantages

- More complex implementation
- Directory consumes extra memory

Difference Between Static and Dynamic Hashing

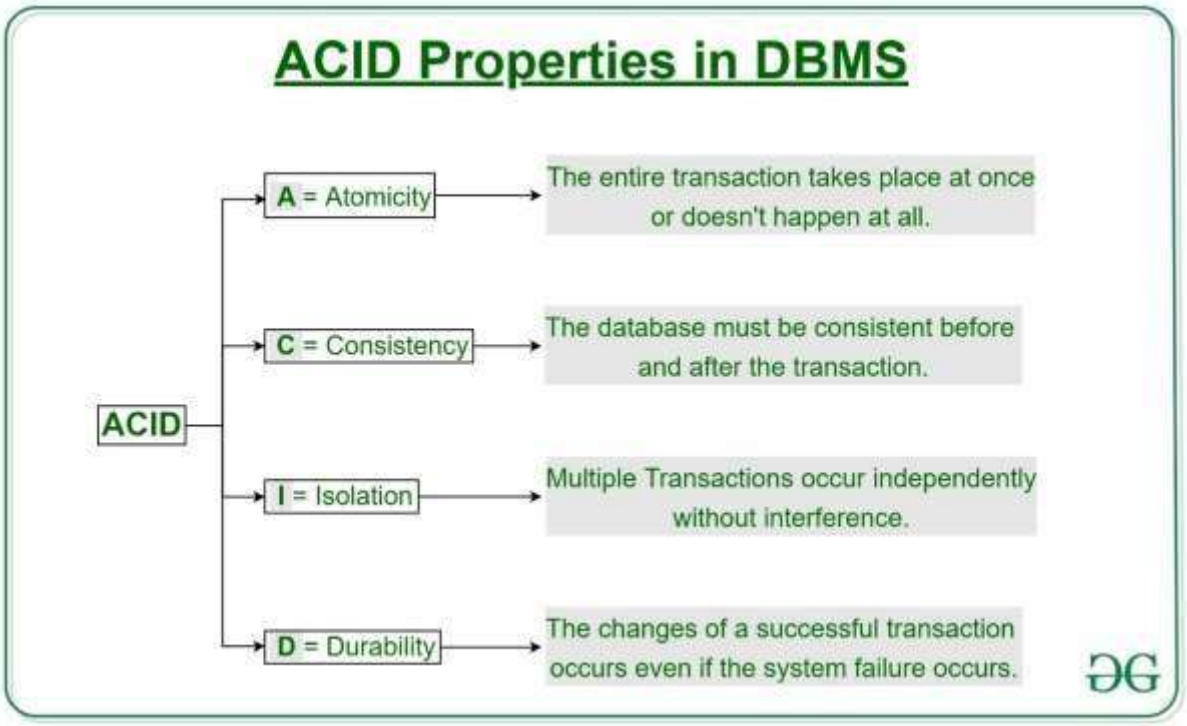
Aspect	Static Hashing	Dynamic Hashing
Number of Buckets	Fixed	Variable
Overflow Handling	Required	Mostly avoided
Performance	Degrades with growth	Remains efficient
Flexibility	Low	High
Complexity	Simple	Complex
Suitability	Small, stable databases	Large, growing databases

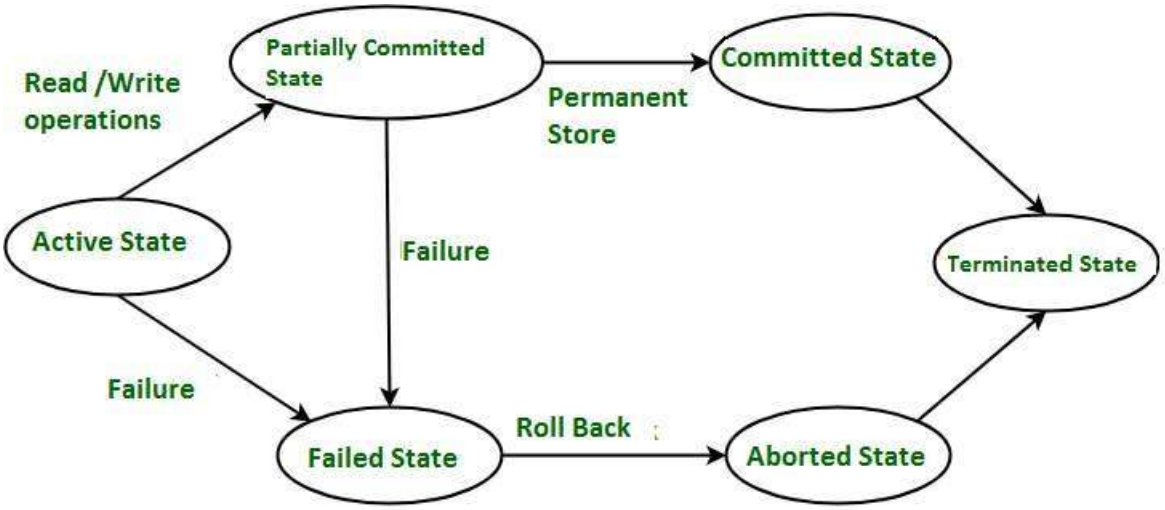
Conclusion

Static Hashing is suitable for small databases with a fixed number of records, but suffers from overflow problems as data grows. **Dynamic Hashing**, by adjusting its structure dynamically, provides better performance and scalability, making it ideal for modern database systems.

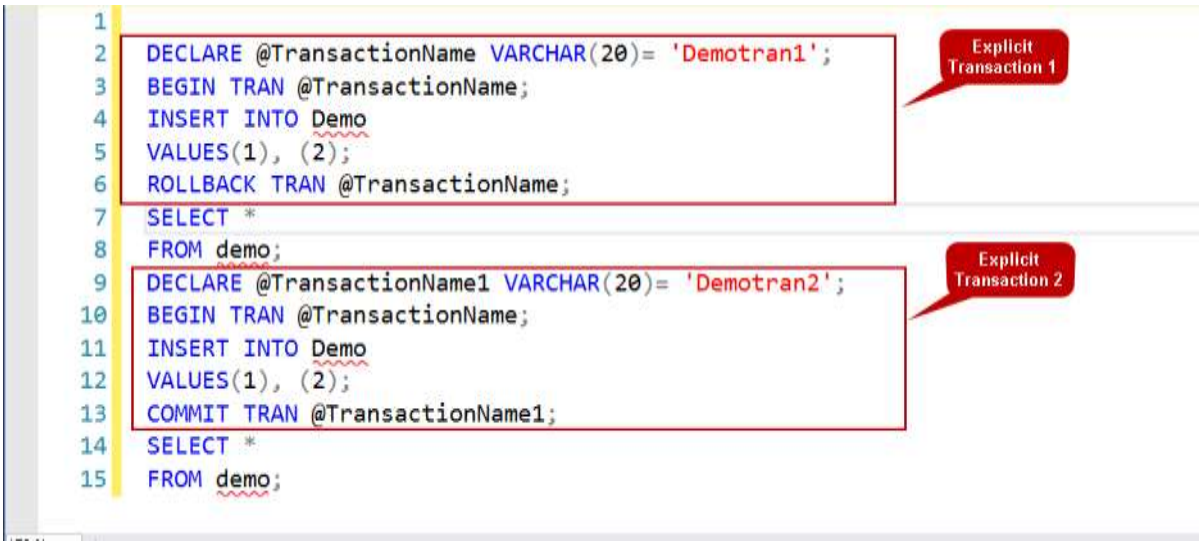
11

Transaction and Its Properties (ACID) with Suitable Example





Transaction States in DBMS




```
1
2  DECLARE @TransactionName VARCHAR(20)= 'Demotran1';
3  BEGIN TRAN @TransactionName;
4  INSERT INTO Demo
5  VALUES(1), (2);
6  ROLLBACK TRAN @TransactionName;
7  SELECT *
8  FROM demo;
9  DECLARE @TransactionName1 VARCHAR(20)= 'Demotran2';
10 BEGIN TRAN @TransactionName;
11 INSERT INTO Demo
12 VALUES(1), (2);
13 COMMIT TRAN @TransactionName1;
14 SELECT *
15 FROM demo;
```

Introduction

In a database system, multiple operations are performed continuously by different users. To ensure that the database remains **correct, consistent, and reliable**, these operations are grouped into a logical unit called a **transaction**. Proper transaction management is a core responsibility of a DBMS.

Definition of Transaction

A **transaction** is a **sequence of one or more database operations** (read, write, update, delete) that are executed as a **single logical unit of work**.
A transaction must be completed **entirely or not at all**.

– Example operations of a transaction:

- Read data
- Update data
- Write data
- Commit or Rollback

Example of a Transaction (Banking System)

Transfer ₹1000 from Account A to Account B

Steps:

1. Read balance of Account A
2. Balance(A) = Balance(A) − 1000
3. Read balance of Account B

4. Balance(B) = Balance(B) + 1000

✓ All steps together form **one transaction**.
If any step fails, the entire transaction must be rolled back.

Properties of Transaction (ACID Properties)

1. Atomicity

- Ensures that a transaction is **all-or-nothing**.
- Either all operations are performed or none are.

Example:
If ₹1000 is debited from Account A but not credited to Account B due to failure, the transaction is rolled back and no money is deducted.

---

2. Consistency

- Ensures that the database moves from **one valid state to another valid state**.
- All integrity constraints must be satisfied.

Example:
Total money in the bank remains the same before and after the transaction.

---

3. Isolation

- Ensures that **concurrent transactions do not interfere** with each other.
- Intermediate results of a transaction are not visible to others.

Example:
While Transaction T1 is transferring money, Transaction T2 cannot see the partial deduction from Account A.

---

4. Durability

- Ensures that once a transaction is **committed**, its effects are **permanently stored** in the database.
- Changes survive system crashes or power failures.

Example:
After successful transfer and commit, account balances remain updated even if the system crashes.

Summary of ACID Properties

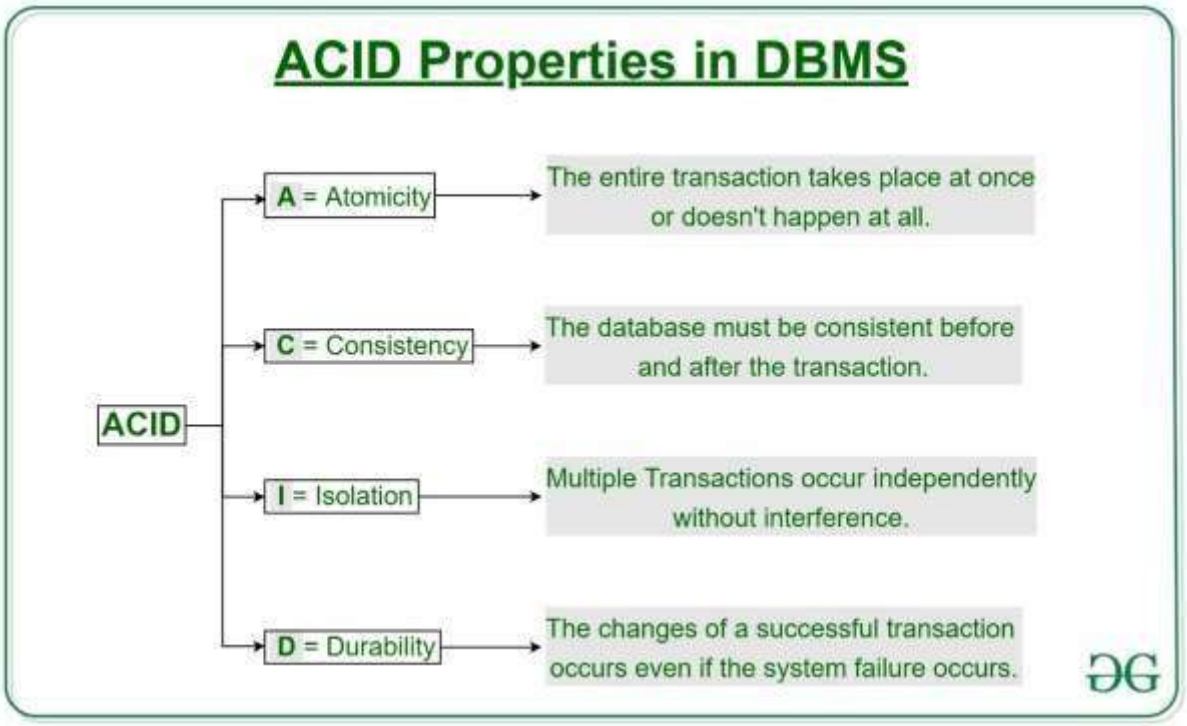
Property	Description
Atomicity	All or nothing execution
Consistency	Maintains database correctness
Isolation	Transactions execute independently
Durability	Committed data is permanent

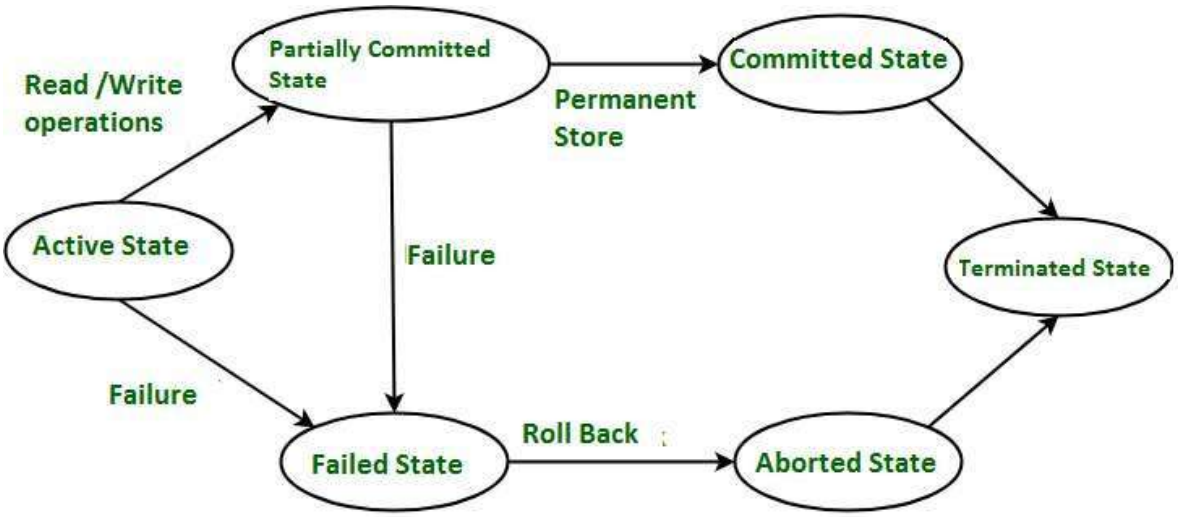
Conclusion

A **transaction** is the fundamental unit of database processing. The **ACID properties** ensure reliability, correctness, and integrity of the database even in the presence of failures and concurrent access. Without these properties, a database system cannot function safely in real-world applications like banking, e-commerce, and reservation systems

12

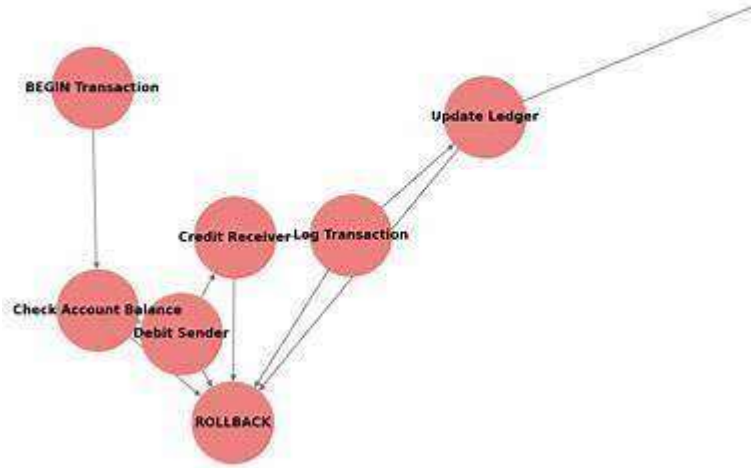
ACID Properties of a Transaction (Detailed Explanation)





Transaction States in DBMS

ACID Transaction Flow Diagram



Introduction

In a database system, transactions must be executed reliably even in the presence of **system failures, power crashes, or concurrent access by multiple users**. To ensure this reliability, every transaction in a DBMS follows a set of fundamental rules known as the **ACID properties**. These properties guarantee the **correctness, consistency, and dependability** of database transactions.

ACID Properties

ACID stands for:

- **A**tomicity
- **C**onsistency

- **I**solation
- **D**urability

---

1. Atomicity

Definition

Atomicity ensures that a transaction is treated as a **single indivisible unit**, which means:

- Either **all operations of the transaction are executed**, or
- **None of them are executed**

There is **no partial execution**.

Explanation

If a transaction fails at any point, the DBMS rolls back all changes made by that transaction, restoring the database to its previous consistent state.

Example

Bank Transaction: Transfer ₹1000 from Account A to Account B

Steps:

1. Debit ₹1000 from Account A
2. Credit ₹1000 to Account B

If the system crashes after step 1 and before step 2:

- Atomicity ensures that ₹1000 is **not deducted** from Account A.
- The entire transaction is rolled back.

Importance

- Prevents partial updates
- Maintains database correctness

---

2. Consistency

Definition

Consistency ensures that a transaction takes the database from **one valid (consistent) state to another valid state**, while obeying:

- Integrity constraints

- Rules and business logic

Explanation

A consistent database satisfies all constraints such as:

- Primary key constraint
- Foreign key constraint
- Domain constraints

If a transaction violates any constraint, it is **aborted**.

Example

If total bank balance before transaction = ₹50,000
After transaction = ₹50,000

✓ Consistency is maintained.

If after a transaction, total becomes ₹49,000 → + inconsistent state.

Importance

- Ensures correctness of data
- Preserves database rules

---

3. Isolation

Definition

Isolation ensures that **multiple transactions executing concurrently** do not interfere with each other.
Each transaction is executed as if it is **the only transaction in the system**.

Explanation

Intermediate results of a transaction are **not visible** to other transactions until the transaction is committed.

This avoids problems like:

- Dirty read
- Non-repeatable read
- Phantom read

Example

Transaction T1 transfers ₹1000 from Account A to B.
Transaction T2 checks Account A balance.

Isolation ensures:

- T2 sees either the **old balance** or the **final committed balance**
- T2 never sees an intermediate balance

Importance

- Prevents inconsistency due to concurrent access
- Ensures reliable multi-user operation

---

4. Durability

Definition

Durability guarantees that once a transaction is **successfully committed**, its changes are **permanently saved** in the database.

Explanation

Even if a system crash, power failure, or restart occurs immediately after commit, the committed data **will not be lost**.

This is achieved using:

- Log files
- Stable storage
- Recovery mechanisms

Example

After transferring ₹1000 and committing the transaction:

- System crashes
- On restart, updated balances are still present

✓ Durability is maintained.

Importance

- Ensures reliability of committed data
- Essential for critical applications like banking

Summary Table of ACID Properties

Property	Meaning	Ensures
Atomicity	All or nothing execution	No partial updates
Consistency	Valid state to valid state	Data correctness
Isolation	Independent execution	No interference
Durability	Permanent storage	Data persistence

Conclusion

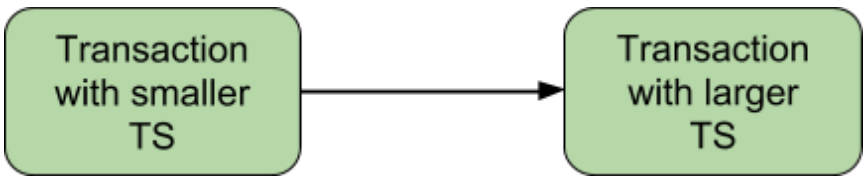
The **ACID properties** form the backbone of reliable transaction processing in a DBMS.

- **Atomicity** prevents partial execution
- **Consistency** maintains database rules
- **Isolation** enables safe concurrent access
- **Durability** ensures permanent storage

Together, these properties make databases dependable for real-world applications such as **banking, airline reservations, e-commerce, and financial systems**.

13

Time Stamp–Based Protocols



Time of Transaction	T1 (Timestamp = 100)	T2 (Timestamp = 200)	T3 (Timestamp = 300)
Time 1	R (A)		
Time 2		R (B)	
Time 3	W (C)		
Time 4			R (B)
Time 5	R (C)		
Time 6		W (B)	
Time 7			W (A)

Introduction

In a multi-user database system, several transactions may execute **concurrently**. To maintain **database consistency and isolation**, concurrency control mechanisms are required.
Time Stamp–Based Protocols are non-locking concurrency control techniques that use **time stamps** to order transactions and ensure serializability.

What is a Time Stamp?

A **time stamp** is a **unique identifier** assigned to each transaction when it enters the system. It represents the **starting time** of the transaction.

- Usually denoted as:

TS(T) → Time stamp of transaction T

The transaction with a **smaller time stamp is older**, and one with a **larger time stamp is younger**.

Basic Idea of Time Stamp–Based Protocol

- Transactions are executed according to their **time stamp order**
- If an operation violates the time stamp order, the transaction is **rolled back**
- No locks are used → avoids deadlock

Data Item Time Stamps

For each data item **X**, the system maintains:

- **RTS(X)** → Read Time Stamp (largest TS of any transaction that read X)
- **WTS(X)** → Write Time Stamp (largest TS of any transaction that wrote X)

Rules of Time Stamp–Based Protocol

1. Read Operation Rule

Transaction **T** wants to read data item **X**:

- If **TS(T) < WTS(X)**
  ✗ Read is rejected
  → T is rolled back (trying to read a newer value)
- Else
  ✓ Read is allowed
  → RTS(X) = max(RTS(X), TS(T))

---

2. Write Operation Rule

Transaction **T** wants to write data item **X**:

- If **TS(T) < RTS(X)**
  ✗ Write is rejected
  → T is rolled back (value already read by newer transaction)
- If **TS(T) < WTS(X)**
  ✗ Write is rejected
  → T is rolled back
- Else
  ✓ Write is allowed
  → WTS(X) = TS(T)

Example

Assume:

- **T1** with TS = 10
- **T2** with TS = 20

Initially:

- RTS(X) = 0
- WTS(X) = 0

Step 1: T1 reads X

✓ Allowed
RTS(X) = 10

Step 2: T2 writes X

✓ Allowed
WTS(X) = 20

Step 3: T1 tries to write X

✗ Not allowed because:
TS(T1) < WTS(X) → 10 < 20

➡ T1 is rolled back

Types of Time Stamp–Based Protocols

1. Basic Time Stamp Ordering Protocol

- Ensures serializability strictly based on time stamps
- Transactions violating rules are rolled back

---

2. Thomas Write Rule (Improvement)

- Allows **obsolete write operations** to be ignored
- Reduces unnecessary rollbacks
- More efficient than basic protocol

Advantages of Time Stamp–Based Protocol

- No deadlock (no locks used)
- Ensures serializability
- Suitable for real-time systems
- Simple conceptual design

Disadvantages

- Higher rollback rate
- Starvation possible for older transactions
- Maintaining time stamps adds overhead

Summary Table

Aspect	**Time Stamp–Based Protocol**
Locking	Not used
Deadlock	Not possible
Rollback	Frequent

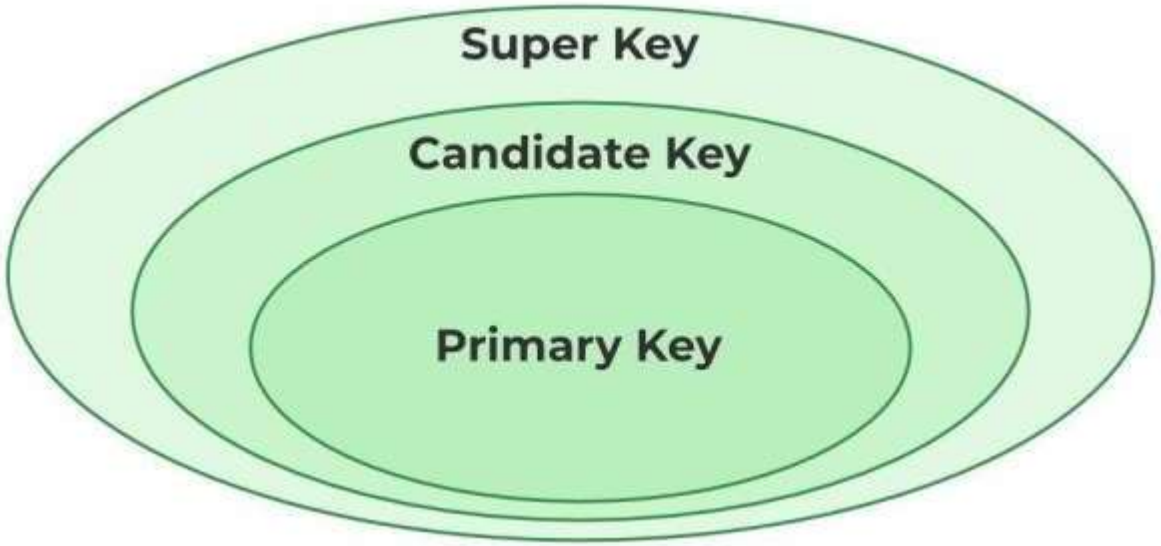
Aspect	Time Stamp–Based Protocol
Ordering	Based on time stamp
Serializability	Guaranteed

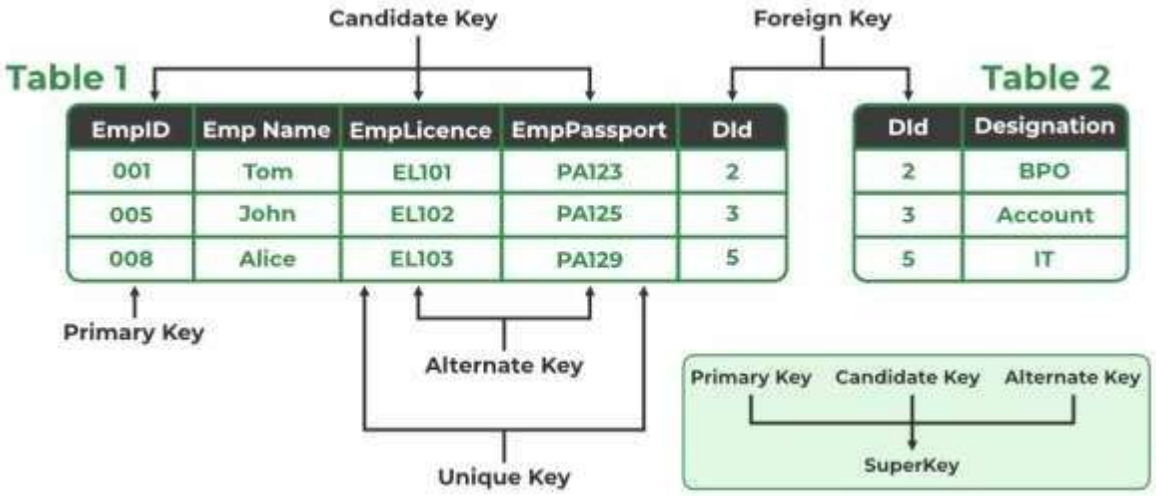
Conclusion

Time Stamp–Based Protocols control concurrency by enforcing a **global ordering of transactions using time stamps**. They ensure serializability without using locks and eliminate deadlocks. However, frequent rollbacks and starvation are major drawbacks. These protocols are especially useful in **real-time and distributed database systems**.

14

Example)





Introduction

In a relational database, **keys** are used to **uniquely identify records (tuples)** in a table. Among different types of keys, **Super Key, Candidate Key, and Primary Key** play a crucial role in ensuring **data integrity, uniqueness, and proper database design**.

---

1. Super Key

Definition

A **Super Key** is a **set of one or more attributes** that can uniquely identify a tuple in a relation.

- It may contain **extra (unnecessary) attributes**
- Ensures uniqueness but not minimality

Example

Consider the table **STUDENT**

Student_ID Roll_No Email Name

Possible **Super Keys**:

- {Student_ID}
- {Roll_No}
- {Email}
- {Student_ID, Name}
- {Roll_No, Email}

✓ All can uniquely identify a student, hence they are super keys.

---

2. Candidate Key

Definition

A **Candidate Key** is a **minimal super key**, i.e.,

- It uniquely identifies a tuple
- **No proper subset** of it can uniquely identify the tuple

—In simple words, a candidate key has **no redundant attributes**.

Example

From the STUDENT table:

Candidate Keys:

- {Student_ID}
- {Roll_No}
- {Email}

+ {Student_ID, Name} is NOT a candidate key (Name is extra).

---

3. Primary Key

Definition

A **Primary Key** is **one candidate key** selected by the database designer to **uniquely identify tuples** in a table.

- Cannot be NULL
- Must be unique
- Only **one primary key** per table

Example

From the candidate keys:

- {Student_ID}, {Roll_No}, {Email}

If we choose **Student_ID**, then:

- **Primary Key = Student_ID**

Key Differences

Aspect	Super Key	Candidate Key	Primary Key
Uniqueness	Yes	Yes	Yes
Minimal	No	Yes	Yes
Redundant attributes	Allowed	Not allowed	Not allowed
NULL values	Allowed	Allowed	Not allowed
Count in table	Many	One or more	Only one
Selection	All possible	Best minimal keys	Chosen from candidate keys

---

Relationship among Keys

- Every **Primary Key** is a **Candidate Key**
- Every **Candidate Key** is a **Super Key**
- But not every Super Key is a Candidate or Primary Key

Conclusion

- **Super Keys** ensure uniqueness
- **Candidate Keys** ensure uniqueness with minimal attributes
- **Primary Key** is the most important key, selected from candidate keys, used for tuple identification

Understanding the difference between these keys is essential for **proper relational database design and maintaining data integrity**.

Part A: Database System Applications with Examples

Introduction

A **Database System** is used to **store, manage, and retrieve large amounts of data efficiently**. Database applications are widely used in almost every sector where data handling, consistency, and security are critical.

Major Database System Applications

1. Banking Systems

Databases are used to store:

- Customer details
- Account information
- Transactions

Example:
ATM transactions, fund transfer, loan management systems.

---

2. Airline / Railway Reservation Systems

Databases manage:

- Passenger details
- Ticket booking
- Seat availability

Example:
Online railway ticket reservation system.

---

3. University / Education Systems

Databases store:

- Student records
- Course registrations
- Examination results

Example:
College management system.

---

4. Hospital Management Systems

Databases manage:

- Patient records
- Doctor schedules
- Medical history

Example:
Electronic Health Record (EHR) systems.

---

5. E-commerce Applications

Databases store:

- Product catalogs
- Customer orders
- Payment details

Example:
Amazon, Flipkart backend databases.

---

6. Telecommunication Systems

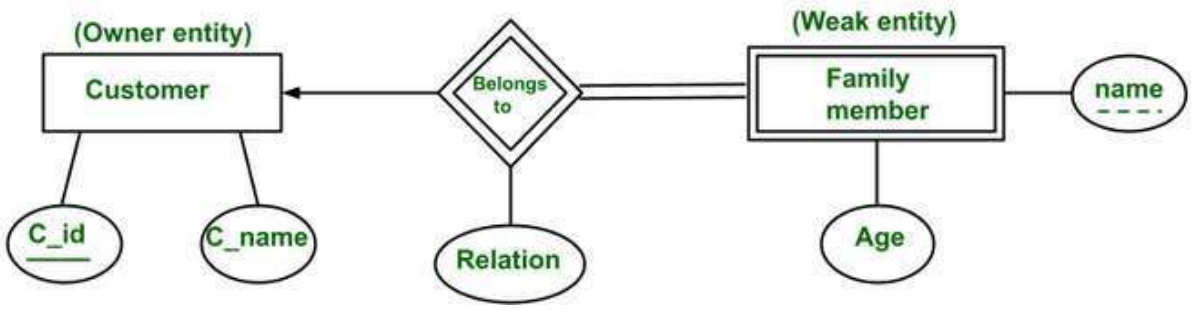
Databases are used for:

- Call records
- Customer billing
- Network usage

Conclusion (Applications)

Database systems form the **backbone of modern applications**, enabling efficient data storage, retrieval, security, and consistency across industries.

Part B: Difference between Strong and Weak Entity Sets in ER Modeling



Strong Entity Set

Definition

A **Strong Entity Set** is an entity set that:

- Has its **own primary key**
- Can be uniquely identified independently

Characteristics

- Existence independent of other entities
- Represented by a **single rectangle**
- Has a primary key

Example

Student

- Student_ID (Primary Key)
- Name, Dept

Weak Entity Set

Definition

A **Weak Entity Set** is an entity set that:

- **Does not have a primary key of its own**
- Depends on a strong entity for identification

Characteristics

- Existence dependent on strong entity
- Identified using:
    - Partial key
    - Primary key of strong entity
- Represented by a **double rectangle**
- Identifying relationship shown by **double diamond**

Example

Dependent

- Dependent_Name (Partial Key)
- Related to Employee

Difference between Strong and Weak Entity Sets

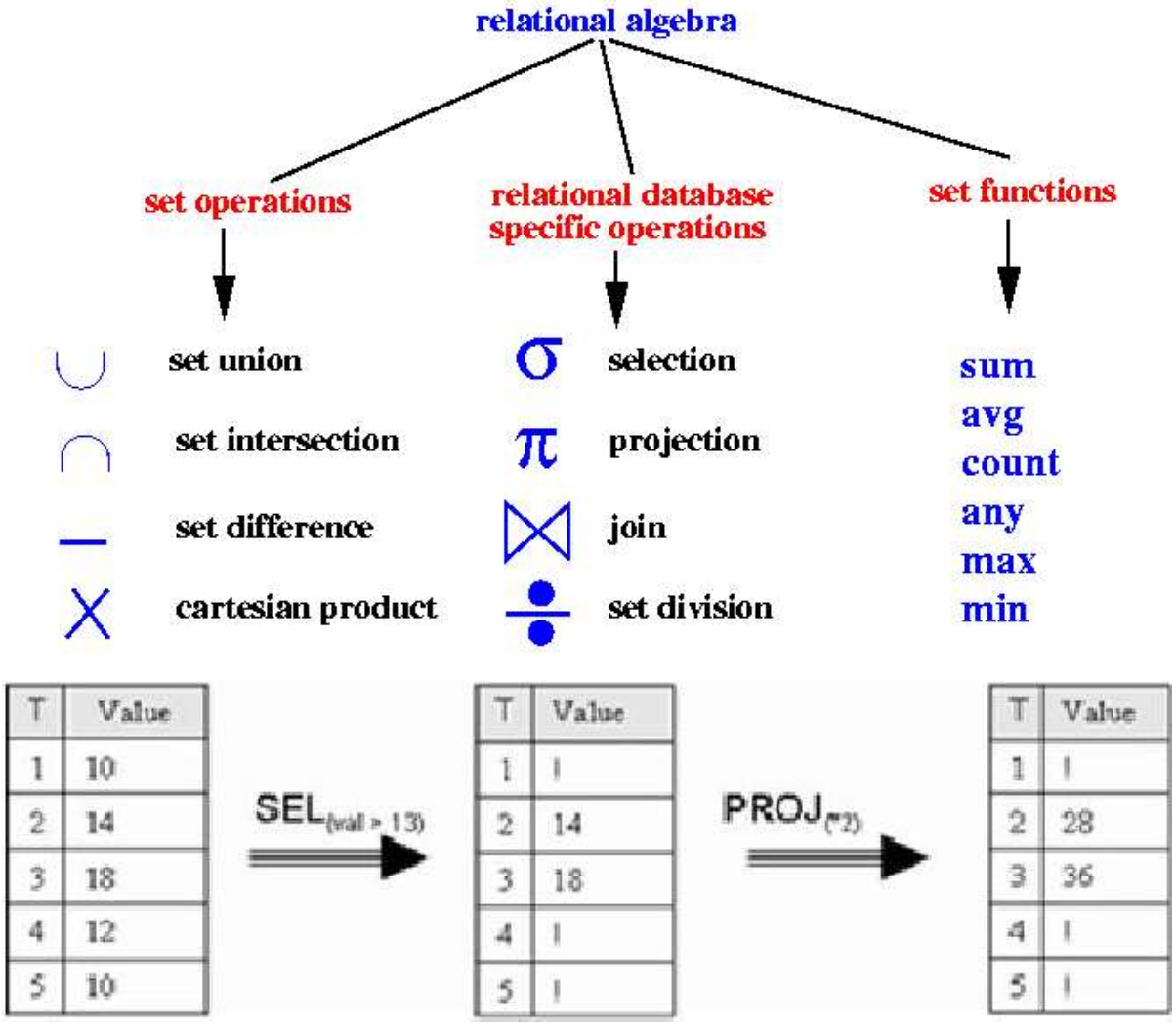
Aspect	Strong Entity Set	Weak Entity Set
Primary Key	Has its own	Does not have
Dependency	Independent	Depends on strong entity
Existence	Can exist alone	Cannot exist alone
Representation	Single rectangle	Double rectangle
Key Type	Primary key	Partial key
Example	Student, Employee	Dependent, Order_Item

Conclusion

- **Strong entities** represent independent real-world objects.
- **Weak entities** depend on strong entities for identification and existence. Understanding this distinction is essential for **accurate ER diagram design and database integrity**.

16

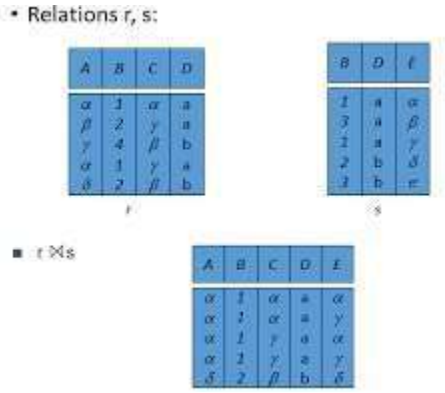
Fundamental Operations in Relational Algebra & Queries



T	Value
1	10
2	14
3	18
4	12
5	10

T	Value
1	!
2	14
3	18
4	!
5	!

T	Value
1	!
2	28
3	36
4	!
5	!



Introduction

Relational Algebra is a formal, procedural query language used to retrieve and manipulate data stored in relational databases. It consists of a set of **fundamental operations** that form the basis for more complex queries in DBMS.

Fundamental Operations in Relational Algebra

The basic (fundamental) relational algebra operations are:

1. **Selection (σ)**
   – Selects tuples (rows) that satisfy a given condition
2. **Projection (π)**
   – Selects specific attributes (columns) from a relation
3. **Union (∪)**
   – Combines tuples from two relations
4. **Set Difference (−)**
   – Returns tuples present in one relation but not in another
5. **Cartesian Product (×)**
   – Combines each tuple of one relation with every tuple of another
6. **Rename (ρ)**
   – Renames relations or attributes

Relational Algebra Queries

Assumed Relations

EMPLOYEE(Emp_ID, Name, Salary, Dept)
STUDENT(Sid, Name)
COURSE(Cid, Cname)
ENROLL(Sid, Cid)

---

a) Retrieve names of employees earning more than 50,000

Relational Algebra Expression

$$\pi_{Name}(\sigma_{Salary>50000}(EMPLOYEE))$$

Explanation

- **σ** selects employees with salary > 50,000
- **π** projects only the Name attribute

---

b) Find all students enrolled in "Database Systems"

Step 1: Select course "Database Systems"

$\sigma_{Cname='Database\ S}$ $_{ystems'}(COURSE)$

Step 2: Join with ENROLL and STUDENT relations

Final Relational Algebra Expression

$\pi_{Name}(STUDENT \bowtie ENROLL$
$\bowtie \sigma_{Cname='Database\ S}$ $_{ystems'}(COURSE))$

Explanation

- Select the required course
- Join COURSE with ENROLL using Cid
- Join result with STUDENT using Sid
- Project student names

Summary Table

Operation	Symbol	Purpose
Selection	σ	Select rows
Projection	π	Select columns
Union	∪	Combine relations
Difference	−	Subtract relations

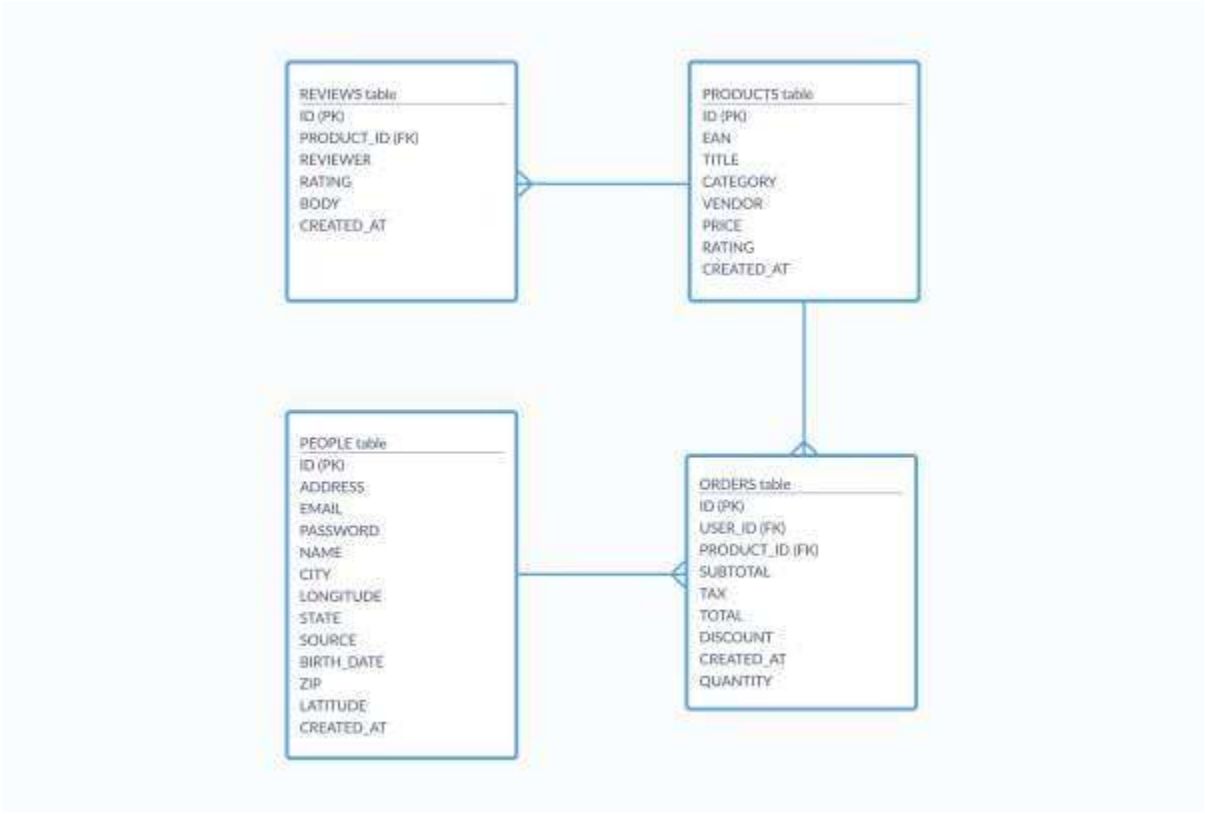
Operation	Symbol	Purpose
Cartesian Product	×	Pair all tuples
Rename	ρ	Rename relation

Conclusion

Relational algebra provides a **theoretical foundation** for query processing in DBMS. Using its **fundamental operations**, complex database queries—such as filtering employees by salary or finding students enrolled in a specific course—can be expressed in a precise and structured manner.

17

Schema, Instance, and Relation with Example s Difference between Relational Algebra and Relational Calculus



Part A: Schema, Instance, and Relation

Introduction

In a relational database system, data is described at different levels of abstraction. The terms **schema**, **instance**, and **relation** are fundamental concepts used to understand the structure and state of a database.

---

1. Schema

Definition

A **Schema** is the **logical structure or blueprint** of the database.
It defines:

- Tables
- Attributes
- Data types
- Constraints

–Schema is **static** (does not change frequently).

Example

STUDENT (Roll_No, Name, Dept, Marks)

This defines the structure of the STUDENT table.

---

2. Instance

Definition

An **Instance** is the **actual data stored** in the database at a particular moment of time.

– Instance is **dynamic** (changes whenever data is inserted, deleted, or updated).

Example (Instance of STUDENT table at a given time)

Roll_No	Name	Dept	Marks
1	Amit	CSE	85
2	Riya	IT	78

---

3. Relation

Definition

A **Relation** is a **table** in a relational database consisting of:

- Rows (tuples)
- Columns (attributes)

It follows a defined schema and contains a set of tuples.

Example

The **STUDENT** table itself is a **relation**, where:

- Attributes → Roll_No, Name, Dept, Marks
- Tuples → Each row of data

Summary (Schema, Instance, Relation)

Term	Meaning	Nature
Schema	Database structure	Static
Instance	Database data at a time	Dynamic
Relation	Table of rows C columns	Logical

Part B: Difference between Relational Algebra and Relational Calculus

Relational Algebra

Definition

Relational Algebra is a **procedural query language**.
It specifies **how to retrieve data** by applying a sequence of operations.

Key Characteristics

- Uses operators like σ, π, ∪, ×, −
- Query result is always a relation
- Foundation for query optimization

Example

Retrieve names of CSE students:

$$\pi_{Name}(\sigma_{Dept='CSE'}\ (STUDENT))$$

Relational Calculus

Definition

Relational Calculus is a **non-procedural (declarative) query language**.
It specifies **what data to retrieve**, not how to retrieve it.

Types

- Tuple Relational Calculus (TRC)
- Domain Relational Calculus (DRC)

Example (Tuple Relational Calculus)

{ t.Name | STUDENT(t) ∧ t.Dept = 'CSE' }

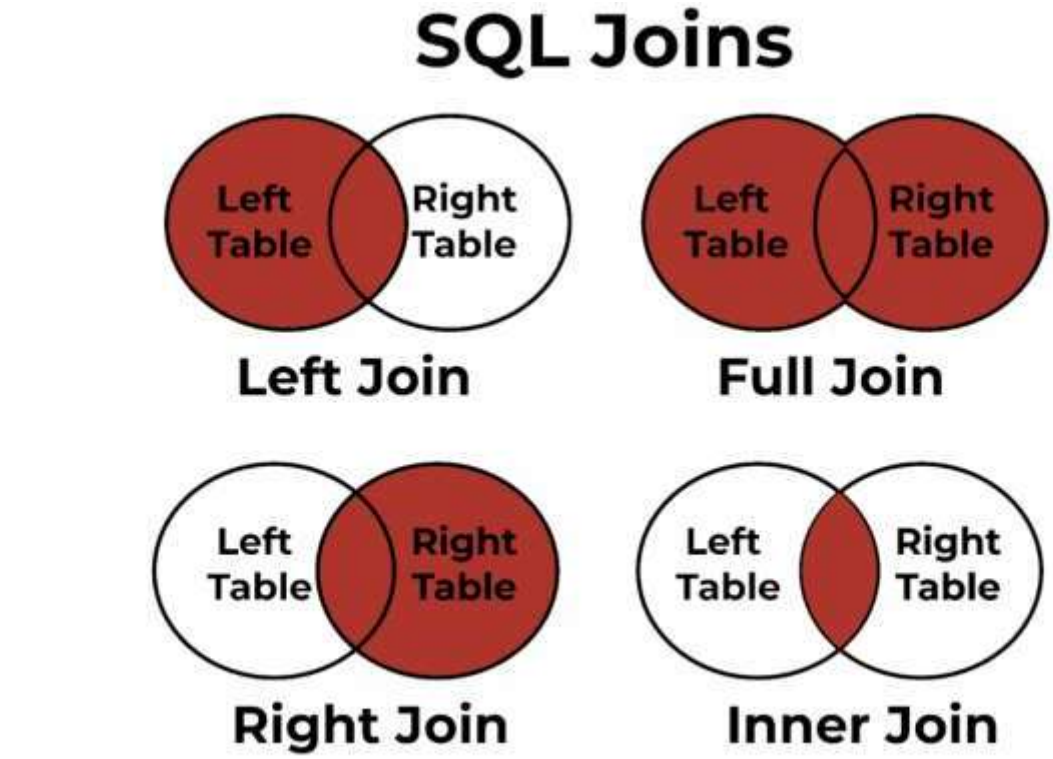
Difference between Relational Algebra and Relational Calculus

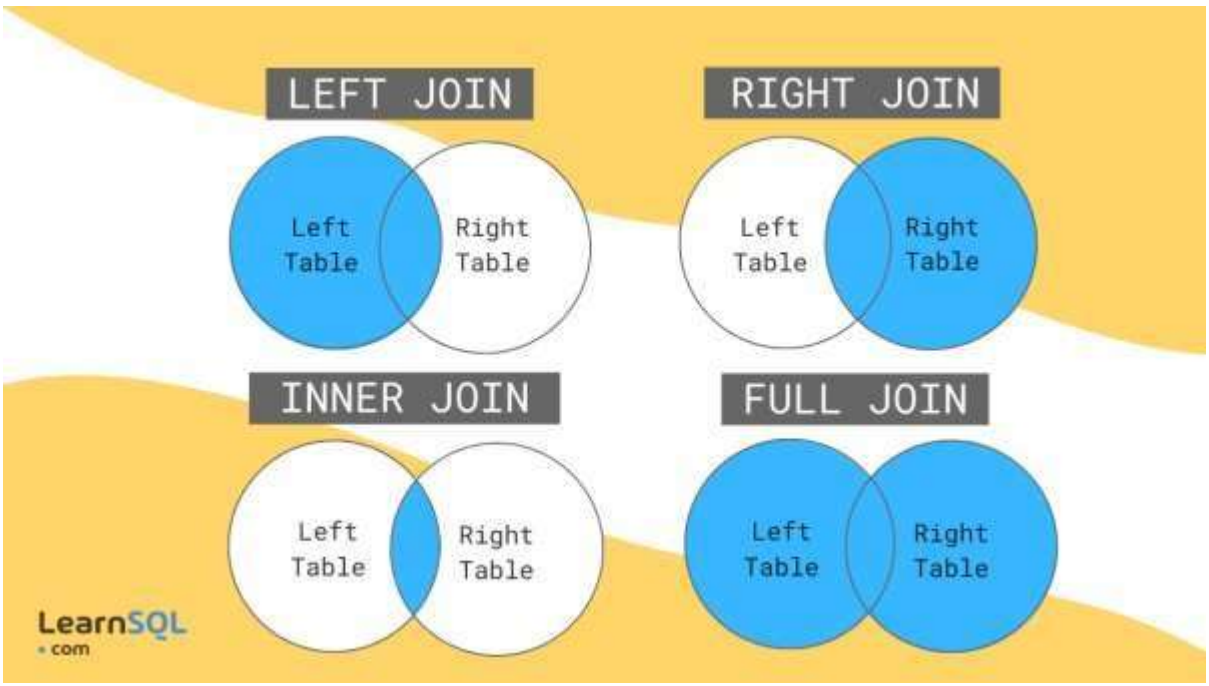
Aspect	**Relational Algebra**	**Relational Calculus**
Type	Procedural	Non-procedural
Focus	How to retrieve data	What data to retrieve
Language Style	Operational	Declarative
Ease of Understanding	Moderate	More abstract
Use in DBMS	Query execution C optimization	Query specification
SQL Relation	Basis of execution	Basis of SQL logic

Conclusion

- **Schema** defines the structure, **instance** represents the current data, and a **relation** is a table storing tuples.
- **Relational Algebra** focuses on *how* queries are executed, while **Relational Calculus** focuses on *what* result is required.
  Both are essential theoretical foundations of modern database systems.

Types of SQL Commands s SQL Joins (with Syntax and Examples)





Part A: Types of SQL Commands

Introduction

SQL (Structured Query Language) is used to **define, manipulate, control, and query data** in relational databases. SQL commands are broadly classified into different categories based on their functionality.

---

1. Data Definition Language (DDL)

Purpose

Used to **define and modify database structure**.

Commands

- CREATE
- ALTER
- DROP
- TRUNCATE

Example

```
CREATE TABLE STUDENT (
 Roll_No INT,
 Name VARCHAR(30),
 Dept VARCHAR(10)
);
```

---

2. Data Manipulation Language (DML)

Purpose

Used to **insert, update, and delete data** in tables.

Commands

- INSERT
- UPDATE
- DELETE

Example

```
INSERT INTO STUDENT VALUES (1, 'Amit', 'CSE');
```

---

3. Data Query Language (DQL)

Purpose

Used to **retrieve data** from database.

Command

- SELECT

Example

SELECT Name FROM STUDENT WHERE Dept = 'CSE';

---

4. Data Control Language (DCL)

Purpose

Used to **control access permissions**.

Commands

- GRANT
- REVOKE

Example

GRANT SELECT ON STUDENT TO user1;

---

5. Transaction Control Language (TCL)

Purpose

Used to **manage transactions**.

Commands

- COMMIT
- ROLLBACK
- SAVEPOINT

Example

COMMIT;

Summary Table: SQL Command Types

Type	Purpose	Commands
DDL	Define structure	CREATE, ALTER
DML	Modify data	INSERT, UPDATE
DQL	Retrieve data	SELECT
DCL	Control access	GRANT, REVOKE

Type	Purpose	Commands
TCL	Transaction control	COMMIT, ROLLBACK

Part B: SQL Joins

Definition of SQL Join

A **JOIN** is used to **combine rows from two or more tables** based on a **related column** (usually a primary key–foreign key relationship).

Assumed Tables

STUDENT

Sid	Name	Dept

ENROLL

Sid	Course

Types of SQL Joins

---

1. INNER JOIN

Definition

Returns **only matching rows** from both tables.

Syntax

```
SELECT columns
FROM table1
INNER JOIN table2
ON condition;
```

Example

```
SELECT STUDENT.Name, ENROLL.Course
FROM STUDENT
INNER JOIN ENROLL
ON STUDENT.Sid = ENROLL.Sid;
```

---

2. LEFT OUTER JOIN

Definition

Returns **all rows from left table** and matching rows from right table.

Syntax

```
SELECT columns
FROM table1
LEFT JOIN table2
ON condition;
```

---

3. RIGHT OUTER JOIN

Definition

Returns **all rows from right table** and matching rows from left table.

Syntax

```
SELECT columns
FROM table1
RIGHT JOIN table2
ON condition;
```

---

4. FULL OUTER JOIN

Definition

Returns **all rows from both tables**, matched or unmatched.

Syntax

```
SELECT columns
FROM table1
FULL OUTER JOIN table2
ON condition;
```

---

5. CROSS JOIN

Definition

Returns **Cartesian product** of both tables.

Syntax

```
SELECT *
```

```
FROM table1
CROSS JOIN table2;
```

---

6. SELF JOIN

Definition

A table is joined with **itself**.

Syntax

```
SELECT A.Name, B.Name
FROM EMP A, EMP B
WHERE A.Manager_ID = B.Emp_ID;
```

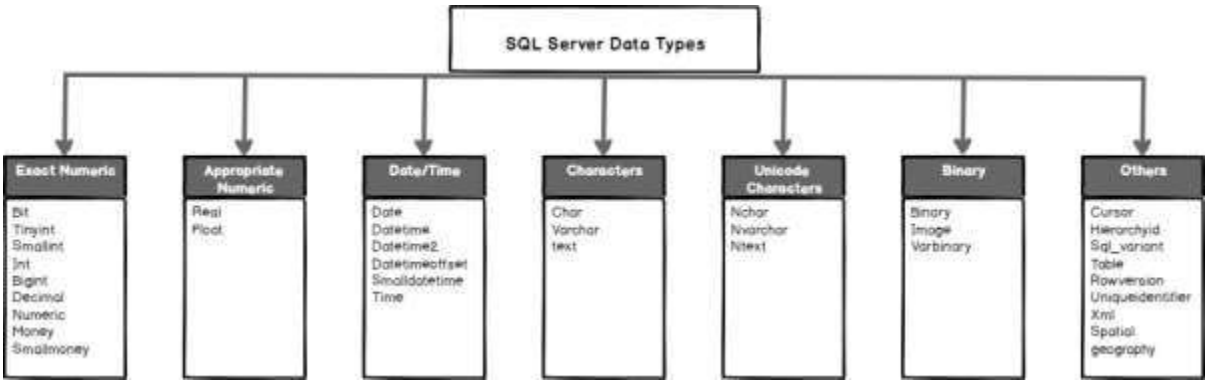
Difference Summary of Joins

Join Type	Result
INNER JOIN	Matching rows only
LEFT JOIN	All left + matching right
RIGHT JOIN	All right + matching left
FULL JOIN	All rows
CROSS JOIN	Cartesian product
SELF JOIN	Table joined with itself

Conclusion

SQL commands are categorized based on their role in **database definition, manipulation, querying, control, and transaction management**.
SQL joins are essential for **retrieving related data from multiple tables**, enabling powerful and meaningful queries in relational databases.

Built-in Data Types and Aggregate Functions in SQL (with Examples)

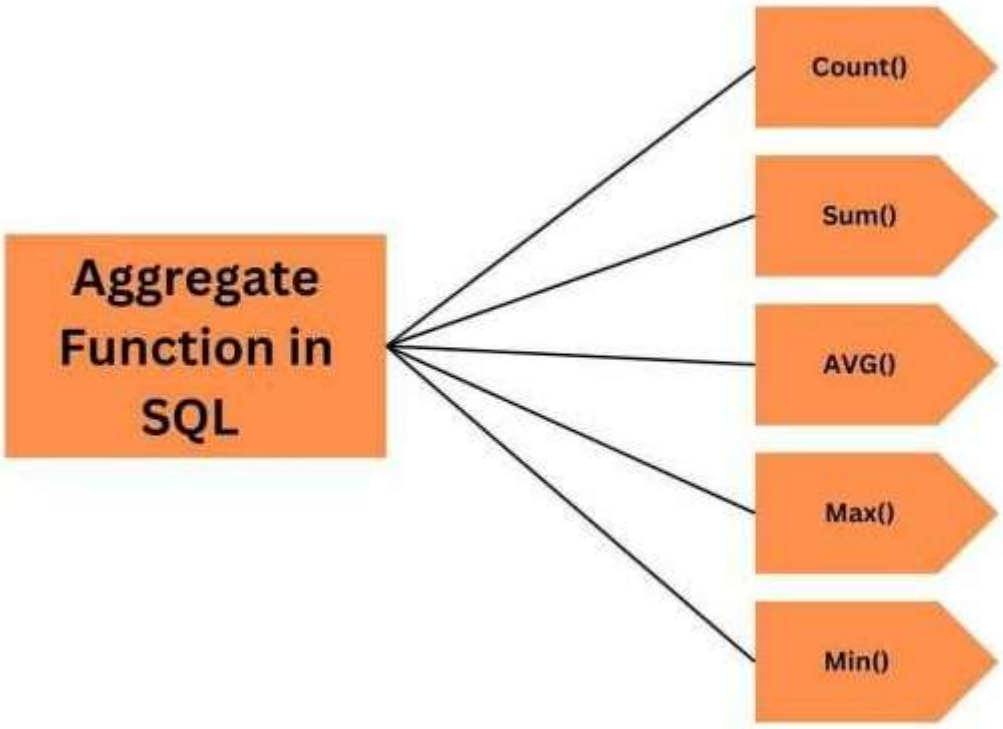



```sql
SELECT
    country.country_name,
    COUNT(city.lat) AS lat_count,
    SUM(city.lat) AS lat_sum,
    AVG(city.lat) AS lat_avg,
    MIN(city.lat) AS lat_min,
    MAX(city.lat) AS lat_max
FROM city
INNER JOIN country ON city.country_id = country.id
GROUP BY country.id, country.country_name;
```

100 %

Results | Messages

	country_name	lat_count	lat_sum	lat_avg	lat_min	lat_max
1	Deutschland	1	52.520008	52.520008	52.520008	52.520008
2	Hrvatska	1	45.815399	45.815399	45.815399	45.815399
3	Polska	1	52.237049	52.237049	52.237049	52.237049
4	Srbija	1	44.787197	44.787197	44.787197	44.787197
5	United States of America	2	74.782845	37.391422	34.052235	40.730610



Part A: Built-in Data Types in SQL (Any Four)

Introduction

SQL provides several **built-in data types** to store different kinds of data efficiently. Choosing the correct data type improves **storage efficiency, data integrity, and query performance**.

---

1. INT (Integer)

- Used to store **whole numbers**
- Can be positive or negative

Example

```
Emp_ID INT
```

Used for employee numbers, roll numbers, etc.

---

2. VARCHAR (Variable Character)

- Used to store **character strings of variable length**
- Saves memory compared to CHAR

Example

Name VARCHAR(50)

Used for names, addresses, department names.

---

3. CHAR (Character)

- Stores **fixed-length character strings**
- Faster access than VARCHAR

Example

Gender CHAR(1)

Used for values like 'M' or 'F'.

---

4. DATE

- Used to store **date values**

Example

Join_Date DATE

Stores date in YYYY-MM-DD format.

(Other common types – for reference)

- FLOAT / DOUBLE
- DECIMAL
- BOOLEAN

Part B: Aggregate Functions in SQL

Definition

Aggregate functions perform **calculations on a group of rows** and return a **single result**. They are commonly used with the **GROUP BY** clause.

Common Aggregate Functions

Function	Purpose
COUNT()	Counts rows
SUM()	Calculates total

Function	Purpose
AVG()	Calculates average
MAX()	Finds maximum
MIN()	Finds minimum

Examples of Aggregate Functions

SELECT COUNT(*) FROM EMPLOYEE;

SELECT AVG(Salary) FROM EMPLOYEE;

SELECT MAX(Salary) FROM EMPLOYEE;

Part C: SQL Queries (Employee Table)

Assumed EMPLOYEE Table

Emp_ID Name Department Salary

---

a) Find names of employees working in "Sales" department

SELECT Name

FROM EMPLOYEE

WHERE Department = 'Sales';

---

b) List the average salary by department

SELECT Department, AVG(Salary) AS Average_Salary

FROM EMPLOYEE

GROUP BY Department;

Explanation

- AVG(Salary) calculates average salary
- GROUP BY Department groups employees department-wise

Summary

Concept	Key Point
Built-in data types	Define nature of data
Aggregate functions	Operate on multiple rows
GROUP BY	Used with aggregates
WHERE	Filters rows

Conclusion

SQL **built-in data types** help in efficient data storage, while **aggregate functions** enable meaningful data analysis such as totals, averages, and counts. Together, they play a crucial role in **database querying and reporting**.

20

Write SQL queries for the table consists of Employee: a) Find names of employees working in "Sales" department. b) List the average salary by department.

Below are the **correct SQL queries** written in **exam-ready format**.

Assumed Table: EMPLOYEE

Emp_ID Name Department Salary

---

a) Find names of employees working in "Sales" department

```
SELECT Name
FROM EMPLOYEE
WHERE Department = 'Sales';
```

Explanation:

- SELECT Name → retrieves employee names
- WHERE Department = 'Sales' → filters employees from Sales department

---

b) List the average salary by department

```
SELECT Department, AVG(Salary) AS Average_Salary
FROM EMPLOYEE
GROUP BY Department;
```

Explanation:

- AVG(Salary) → calculates average salary
- GROUP BY Department → groups employees department-wise

21

The following are SQL queries based on a typical "university schema," which commonly includes tables such as course, department, instructor, student, takes, and section. a) Find the titles of courses in the Comp. Sci. department that have 3 credits. b) Find the IDs of all students who were taught by an instructor named Einstein; make sure there are no duplicates in the result. c) Find the highest salary of any instructor. d) Find all instructors earning the highest salary (there may be more than one with the same salary).

Below are **standard SQL queries** written for a **typical University schema** (as used in DBMS textbooks like Silberschatz–Korth).
These are **exam-ready, correct, and concise**.

Assumed University Schema (standard)

- **course(course_id, title, dept_name, credits)**
- **instructor(ID, name, dept_name, salary)**
- **student(ID, name, dept_name, tot_cred)**
- **takes(ID, course_id, sec_id, semester, year)**
- **teaches(ID, course_id, sec_id, semester, year)**
- **section(course_id, sec_id, semester, year, building, room_number)**

---

a) Find the titles of courses in the Comp. Sci. department that have 3 credits

```
SELECT title
FROM course
WHERE dept_name = 'Comp. Sci.'
 AND credits = 3;
```

---

b) Find the IDs of all students who were taught by an instructor named Einstein (no duplicates)

```
SELECT DISTINCT takes.ID
FROM takes
JOIN teaches
 ON takes.course_id = teaches.course_id
AND takes.sec_id = teaches.sec_id
```

```
 AND takes.semester = teaches.semester
 AND takes.year = teaches.year
JOIN instructor
 ON teaches.ID = instructor.ID
WHERE instructor.name = 'Einstein';
```

---

c) Find the highest salary of any instructor

```
SELECT MAX(salary)
FROM instructor;
```

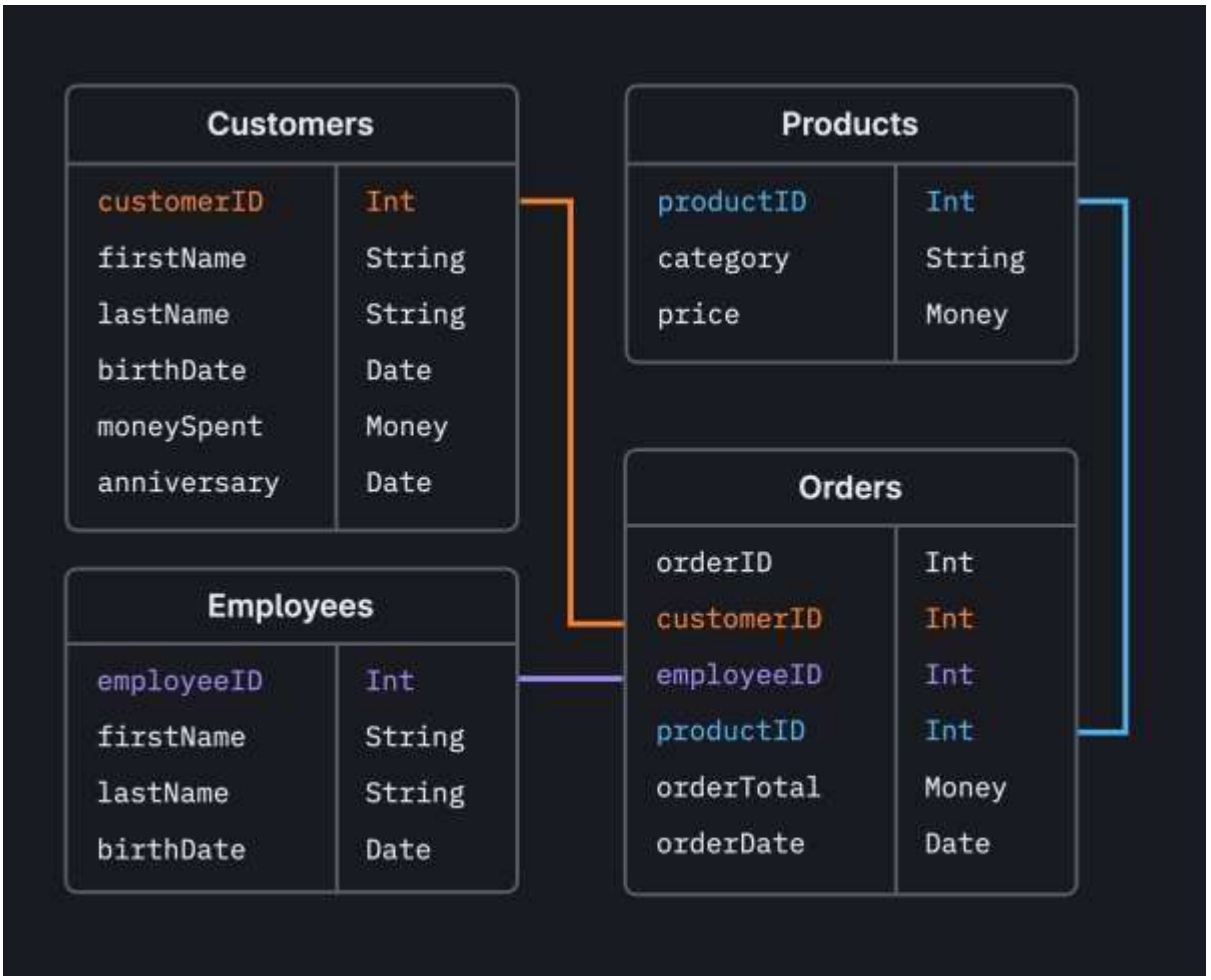
---

d) Find all instructors earning the highest salary

```
SELECT name, salary
FROM instructor
WHERE salary = (
   SELECT MAX(salary)
   FROM instructor
);
```

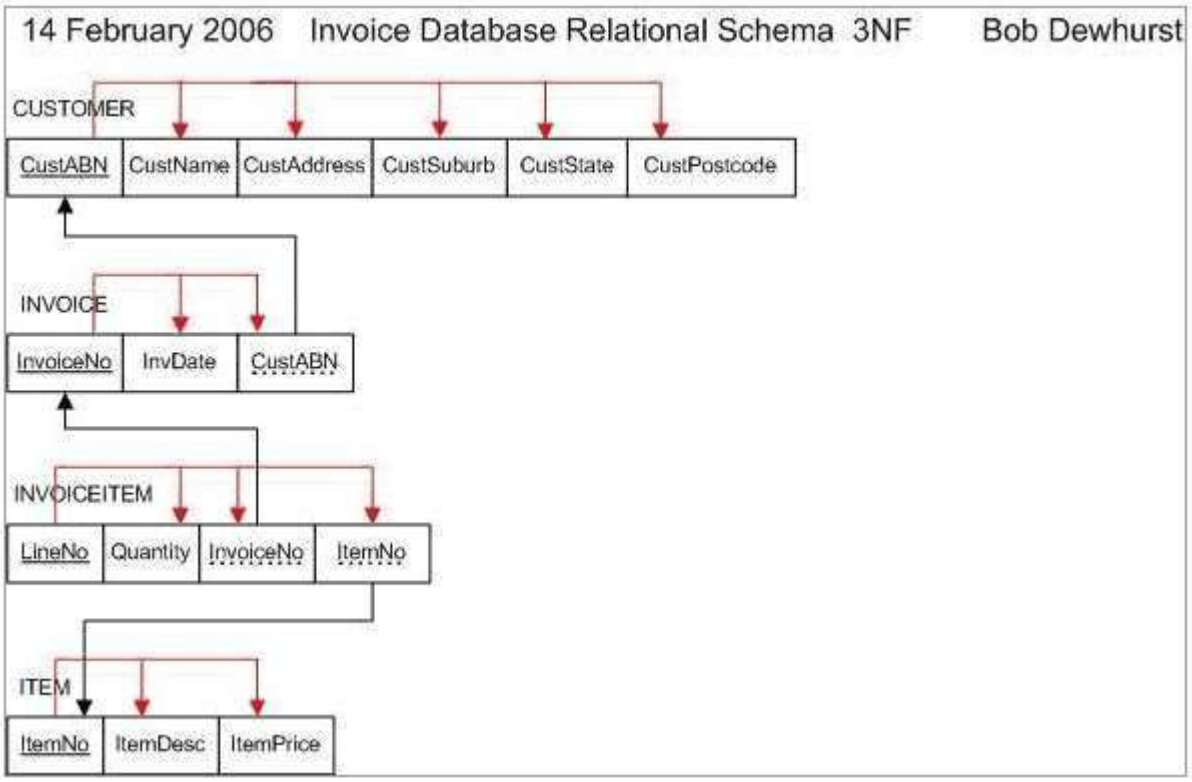
22

Features of a Good Relational Design s Functional Dependency (with Example)



The table below lists customer/car hire data. Each customer may hire cars from various outlets throughout Glasgow. A car is registered at a particular outlet and can be hired out to a customer on a given date.

CarReg	hireDate	Make	model	custNo	custName	outletNo	outletLoc
MS54 0GD	14/5/03	Ford	Focus	C100	Smith, J	01	Bearsden
MS54 0GD	15/5/03	Ford	Focus	C201	Hay, P	01	Bearsden
NS54 TPR	16/5/03	Nissan	Sunny	C100	Smith, J	01	Bearsden
MH54 BRP	14/5/03	Ford	Ka	C313	Blair, O	02	Kelvinbridge
MH54 BRP	20/5/03	Ford	Ka	C100	Smith, J	02	Kelvinbridge
MD510PQ	20/5/03	Nissan	Sunny	C295	Pen, T	02	Kelvinbridge



Introduction

A **good relational database design** ensures that data is stored efficiently, accurately, and consistently. Proper design reduces redundancy, avoids anomalies, and improves maintainability. One of the core concepts used to evaluate and achieve good design is **Functional Dependency (FD)**.

Features of a Good Relational Design

1. **Minimal Data Redundancy**
    - Each fact is stored only once.
    - Saves storage and avoids inconsistent updates.
2. **Elimination of Update Anomalies**
    - Avoids **insertion**, **deletion**, and **update** anomalies.
    - Achieved through normalization.
3. **Data Integrity**
    - Enforced using **primary keys**, **foreign keys**, and constraints.
    - Ensures correctness and validity of data.
4. **Proper Use of Keys**

- Every relation has a well-defined **primary key**.
- Relationships are maintained using **foreign keys**.

5. **Normalized Structure**
   - Relations are normalized at least up to **Third Normal Form (3NF)**.
   - Removes partial and transitive dependencies.
6. **Clear and Simple Schema**
   - Attributes have meaningful names.
   - Relations are easy to understand and maintain.
7. **Logical Data Independence**
   - Changes in schema minimally affect applications.
   - Enhances flexibility and scalability.

Functional Dependency (FD)

Definition

A **Functional Dependency** is a relationship between two sets of attributes in a relation such that:

An attribute **Y** is functionally dependent on attribute **X** if each value of **X** uniquely determines a value of **Y**.

It is denoted as:
X → Y

Example

Consider the relation **STUDENT**

Roll_No	Name	Dept
1	Amit	CSE
2	Riya	IT

Here:

- **Roll_No → Name**
- **Roll_No → Dept**

This means that **Roll_No uniquely determines Name and Dept**.
So, Name and Dept are **functionally dependent** on Roll_No.

Importance of Functional Dependency

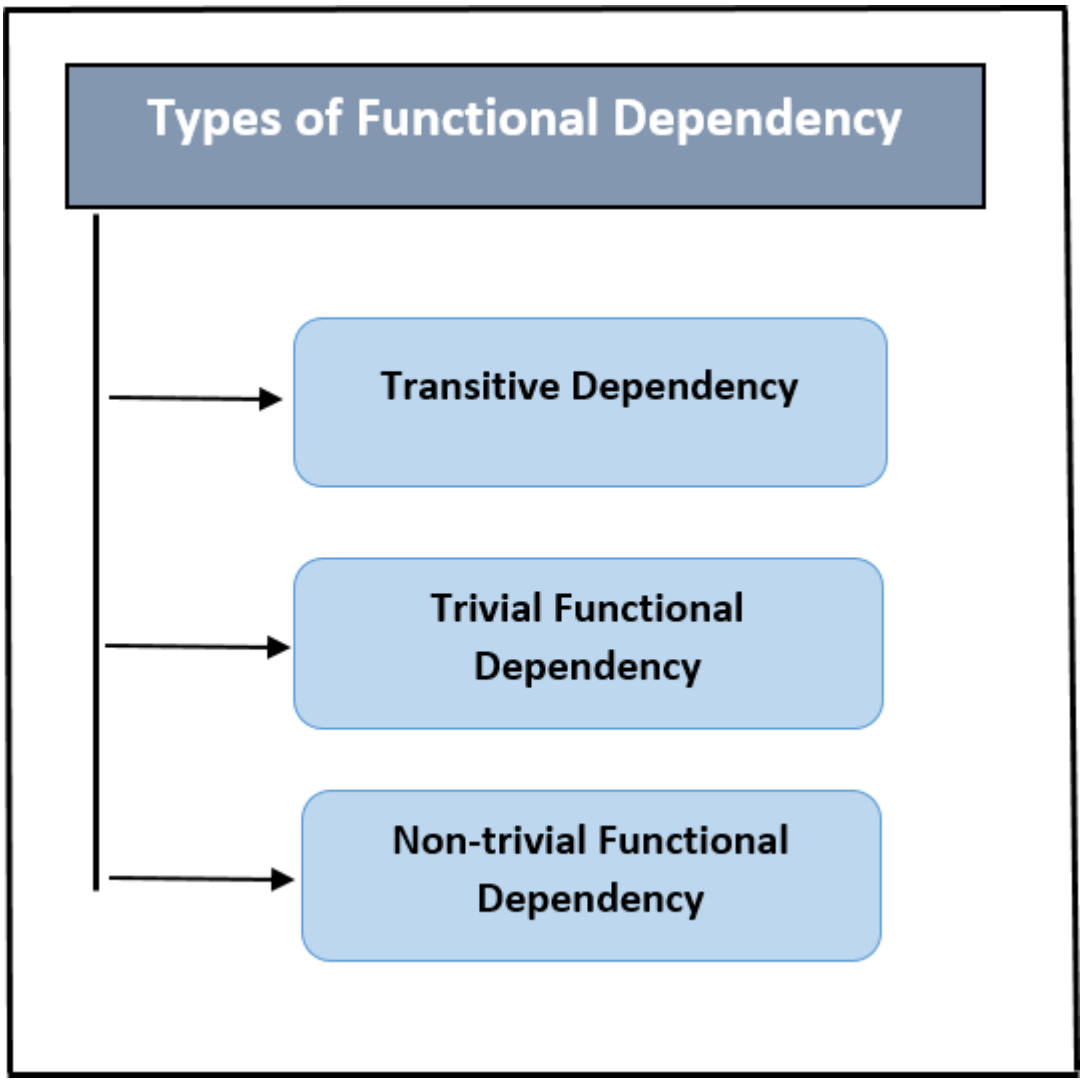
- Helps in **database normalization**
- Identifies **redundancy and anomalies**
- Used to decompose tables into well-structured relations

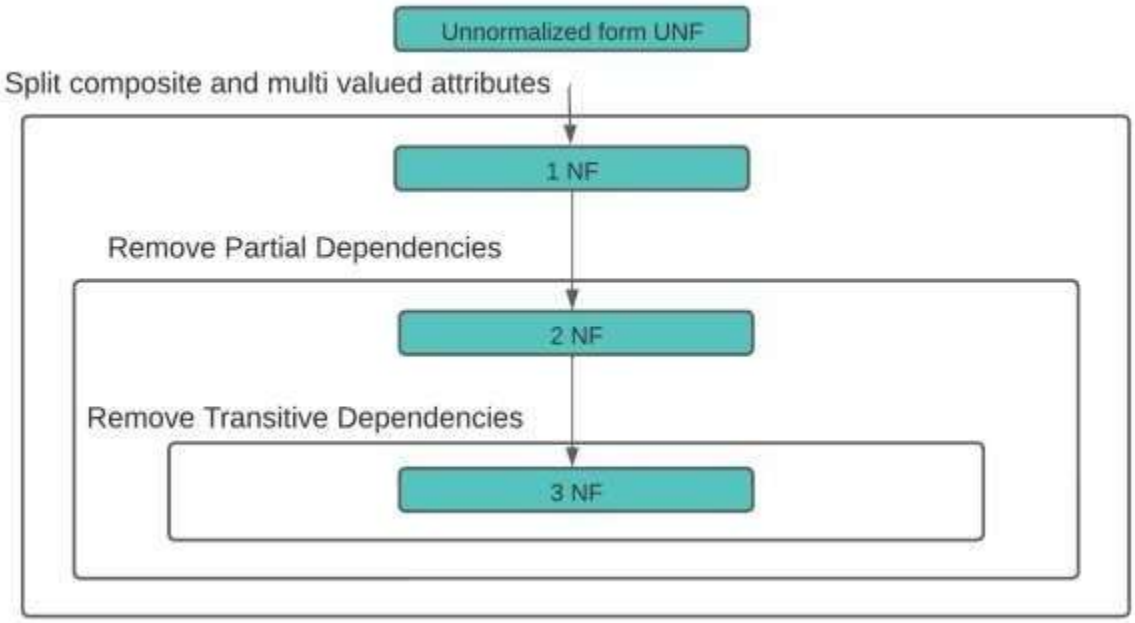
Conclusion

A **good relational design** minimizes redundancy, preserves data integrity, and avoids anomalies.
Functional dependency is a fundamental concept that explains attribute relationships and guides the normalization process, leading to efficient and reliable database design.

23

Trivial Functional Dependencies s Normalization up to BCNF





Part A: Why are certain Functional Dependencies called Trivial Functional Dependencies?

Definition

A **functional dependency (FD)** of the form

$$X \rightarrow Y$$

is called a **trivial functional dependency** if **Y is a subset of X**, i.e.,

$$Y \subseteq X$$

Explanation

- In trivial FDs, the dependency is **always true**
- It does **not provide any new information**
- Such dependencies hold in **every relation by default**

Examples

1. $A \rightarrow A$
2. $\{A, B\} \rightarrow A$
3. $\{A, B, C\} \rightarrow B$

✓ These are trivial because the right-hand side attribute already exists on the left-hand side.

Why they are called "Trivial"?

- They do **not cause redundancy**
- They do **not violate normalization rules**
- They are **ignored during normalization analysis**

Part B: Normalize the Relation R(A, B, C, D, E) up to BCNF

Given

Relation:

$$R(A, B, C, D, E)$$

Functional Dependencies:

- $A \rightarrow B$
- $B \rightarrow C$
- $C \rightarrow D$
- $D \rightarrow E$

Step 1: Find the Candidate Key

Start with **A**:

$$A \rightarrow B \rightarrow C \rightarrow D \rightarrow E$$

So,

$$A^{+} = \{A, B, C, D, E\}$$

🟩 **Candidate Key = A**

Step 2: Check for BCNF

BCNF Condition

A relation is in **BCNF** if for every non-trivial FD

$$X \rightarrow Y$$

X must be a super key

Check Each FD

FD	Is LHS a Super Key?	BCNF Status
$A \rightarrow B$	Yes (A is key)	OK
$B \rightarrow C$	No	✗ Violates BCNF
$C \rightarrow D$	No	✗ Violates BCNF
$D \rightarrow E$	No	✗ Violates BCNF

So, **R is NOT in BCNF**.

Step 3: BCNF Decomposition

We decompose step by step using violating FDs.

Decompose using $B \rightarrow C$

- $R_1(B, C)$
- $R_2(A, B, D, E)$

Now check $R_2(A, B, D, E)$

Remaining FDs:

- $A \rightarrow B$
- $D \rightarrow E$

FD **$D \rightarrow E$** violates BCNF.

Decompose R_2 using $D \rightarrow E$

- $R_3(D, E)$
- $R_4(A, B, D)$

Now check $R_4(A, B, D)$

FD: $A \rightarrow B$
A is the key → ✓ BCNF satisfied

Final BCNF Relations

Relation	Attributes	Key
R_1	(B, C)	B
R_3	(D, E)	D
R_4	(A, B, D)	A

Verification

- All relations satisfy **BCNF**
- No non-trivial FD violates BCNF
- Dependency chain preserved logically

Conclusion

- **Trivial functional dependencies** are those where the right-hand side is a subset of the left-hand side and hence are always true.
- The given relation **R(A, B, C, D, E)** violates BCNF due to non-key dependencies.
- After systematic decomposition, the relation is successfully normalized **up to BCNF**, ensuring **minimal redundancy and strong data integrity**.

24

Closure of Functional Dependencies s Candidate Keys

STUDENT

STUD_NO	STUD_NAME	STUD_PHONE	STUD_STATE	STUD_COUNTRY	STUD_AGE
1	RAM	9716271721	Haryana	India	20
2	RAM	9898291281	Punjab	India	19
3	SUJIT	7898291981	Rajsthan	India	18
4	SURESH		Punjab	India	21

Table 1

Example of Attribute Set Closure

- $R = (A, B, C, G, H, I)$
- $F = \{A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H\}$
- $(AG)^+$
  1. *result* = AG
  2. *result* = $ABCG$ $(A \rightarrow C$ and $A \rightarrow B)$
  3. *result* = $ABCGH$ $(CG \rightarrow H$ and $CG \subseteq AGBC)$
  4. *result* = $ABCGHI$ $(CG \rightarrow I$ and $CG \subseteq AGBCH)$
- Is AG a candidate key? (see definition)
  1. Is AG a super key?
     1. Does $AG \rightarrow R$? == Is $(AG)^+ \supseteq R$
  2. Is any subset of AG a superkey?
     1. Does $A \rightarrow R$? == Is $(A)^+ \supseteq R$
     2. Does $G \rightarrow R$? == Is $(G)^+ \supseteq R$

Given

Relation schema:

$$R(A, B, C, D, E)$$

Functional Dependencies (FDs):

$$F = \{A \rightarrow BC,\ CD \rightarrow E,\ B \rightarrow D,\ E \rightarrow A\}$$

Part A: Compute the Closure of F (Conceptually via Attribute Closures)

To find **candidate keys**, we compute **attribute closures** of relevant attribute sets.

---

1. Closure of A (A^+)

Start with:

$$A^+ = \{A\}$$

Apply FDs:

- $A \rightarrow BC \rightarrow$ add **B, C**
- $B \rightarrow D \rightarrow$ add **D**
- $CD \rightarrow E$(C and D present) $\rightarrow$ add **E**
- $E \rightarrow A$(already present)

$$A^+ = \{A, B, C, D, E\}$$

🟩 **A is a candidate key**

---

2. Closure of E (E^+)

Start with:

$$E^+ = \{E\}$$

Apply FDs:

- $E \rightarrow A \rightarrow$ add **A**
- $A \rightarrow BC \rightarrow$ add **B, C**
- $B \rightarrow D \rightarrow$ add **D**
- $CD \rightarrow E$(already present)

$$E^+ = \{A, B, C, D, E\}$$

🟩 **E is a candidate key**

---

3. Closure of BC (BC^+)

Start with:

$$BC^+ = \{B, C\}$$

Apply FDs:

- $B \rightarrow D \rightarrow$ add **D**
- $CD \rightarrow E \rightarrow$ add **E**
- $E \rightarrow A \rightarrow$ add **A**
- $A \rightarrow BC$(already present)

$$BC^+ = \{A, B, C, D, E\}$$

🟩 **BC is a candidate key**

---

4. Check Smaller Subsets

- $B^+ = \{B, D\}+$
- $C^+ = \{C\}+$
- $AB, AC, AD$are **not minimal** (contain A which alone is key)

Part B: Candidate Keys of R

A **candidate key** is a **minimal attribute set** whose closure contains **all attributes** of R.

Candidate Keys

$$\boxed{\{A\},\ \{E\},\ \{B, C\}}$$

Summary Table

Attribute Set	Closure	Key?
A	A B C D E	■
E	A B C D E	■
BC	A B C D E	■
B	B D	+
C	C	+

Conclusion

- The closure computations show that **A, E, and BC** can each uniquely determine all attributes of relation **R(A, B, C, D, E)**.
- Hence, these are the **candidate keys**.
- Attribute closure is a powerful tool for identifying keys and analyzing normalization.

Locking Mechanism	Concurrency	Blocking Behavior	Benefit
Shared Lock	Multiple transactions can read data simultaneously.	Blocks writes but allows reads.	Ensures data consistency while allowing concurrent reads.
Exclusive Lock	Only one transaction can modify the data.	Blocks both reads and writes from others.	Prevents data corruption by ensuring exclusive access during modifications.
FOR UPDATE	Ensures that only one transaction modifies a row.	Blocks other transactions until the lock is released.	Guarantees that data being modified is not concurrently accessed by others.
SKIP LOCKED	Allows parallel transaction processing without waiting for locked rows.	Skips rows that are locked by other transactions.	Increases throughput by avoiding unnecessary waits.
NO WAIT	Prevents transactions from waiting for locks.	Fails if the row is locked.	Avoids deadlocks and improves system responsiveness by failing quickly when resources are locked.

Part A: Lock-Based Concurrency Control

Introduction

In a multi-user database system, several transactions may execute **simultaneously**. If they access the same data items concurrently, it can lead to **inconsistency**.
Lock-based concurrency control is a technique used by DBMS to **control simultaneous access** to data items using locks, ensuring database consistency and isolation.

What is a Lock?

A **lock** is a variable associated with a data item that **controls access** to that item.

Before a transaction can **read or write** a data item, it must first acquire the appropriate lock.

Types of Locks

1. Shared Lock (S-lock)

- Used for **read operations**
- Multiple transactions can hold a shared lock simultaneously
- No transaction can write while S-lock is held

Example:
Two users reading the same account balance.

---

2. Exclusive Lock (X-lock)

- Used for **write operations**
- Only one transaction can hold an exclusive lock
- No other transaction can read or write the data

Example:
Updating an employee's salary.

Lock Compatibility Table

Lock Held	S-Lock Request	X-Lock Request
S-Lock	Allowed	Not allowed
X-Lock	Not allowed	Not allowed

Two-Phase Locking Protocol (2PL)

Definition

A transaction follows the **Two-Phase Locking (2PL)** protocol if:

1. **Growing Phase** – Locks are acquired, none are released
2. **Shrinking Phase** – Locks are released, none are acquired

This guarantees **conflict serializability**.

Example of Lock-Based Control

Transaction **T1**:

1. Acquire X-lock on A
2. Read A
3. Write A
4. Release lock on A

Transaction **T2** must **wait** until T1 releases the lock before accessing A.

✓ This prevents **lost updates** and **dirty reads**.

Advantages of Lock-Based Concurrency Control

- Ensures data consistency
- Supports serializability
- Widely used in real DBMS

Disadvantages

- Can lead to **deadlocks**
- Lock overhead reduces performance

Part B: ACID Properties of a Transaction

Introduction

A **transaction** is a logical unit of database operations. To ensure reliability, every transaction must satisfy the **ACID properties**.

ACID Properties

1. Atomicity

- Transaction is **all-or-nothing**
- Partial execution is not allowed

Example:
Money debited but not credited → rollback occurs.

---

2. Consistency

- Transaction moves database from one **valid state to another**
- Integrity constraints must be satisfied

Example:
Total bank balance remains unchanged after transfer.

---

3. Isolation

- Concurrent transactions do not interfere
- Intermediate results are hidden

Example:
Another transaction cannot see uncommitted balance changes.

---

4. Durability

- Once committed, changes are **permanent**
- Survive crashes or failures

Example:
After commit, updated salary remains stored even after system crash.

Summary Table (ACID)

Property	Meaning
Atomicity	All or nothing
Consistency	Preserves rules
Isolation	Independent execution
Durability	Permanent storage

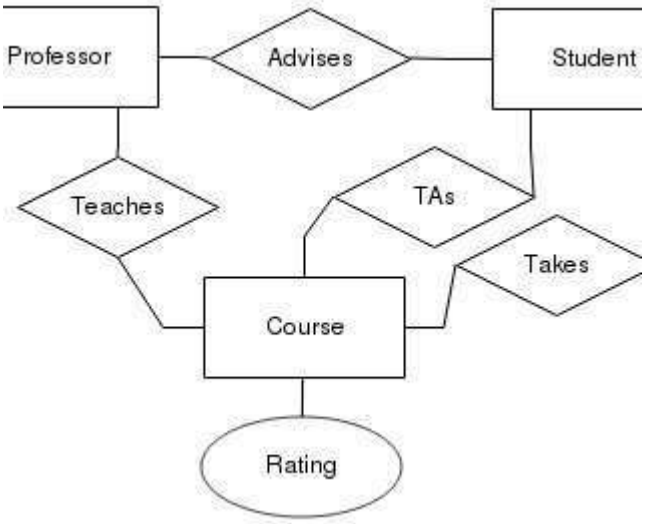
Conclusion

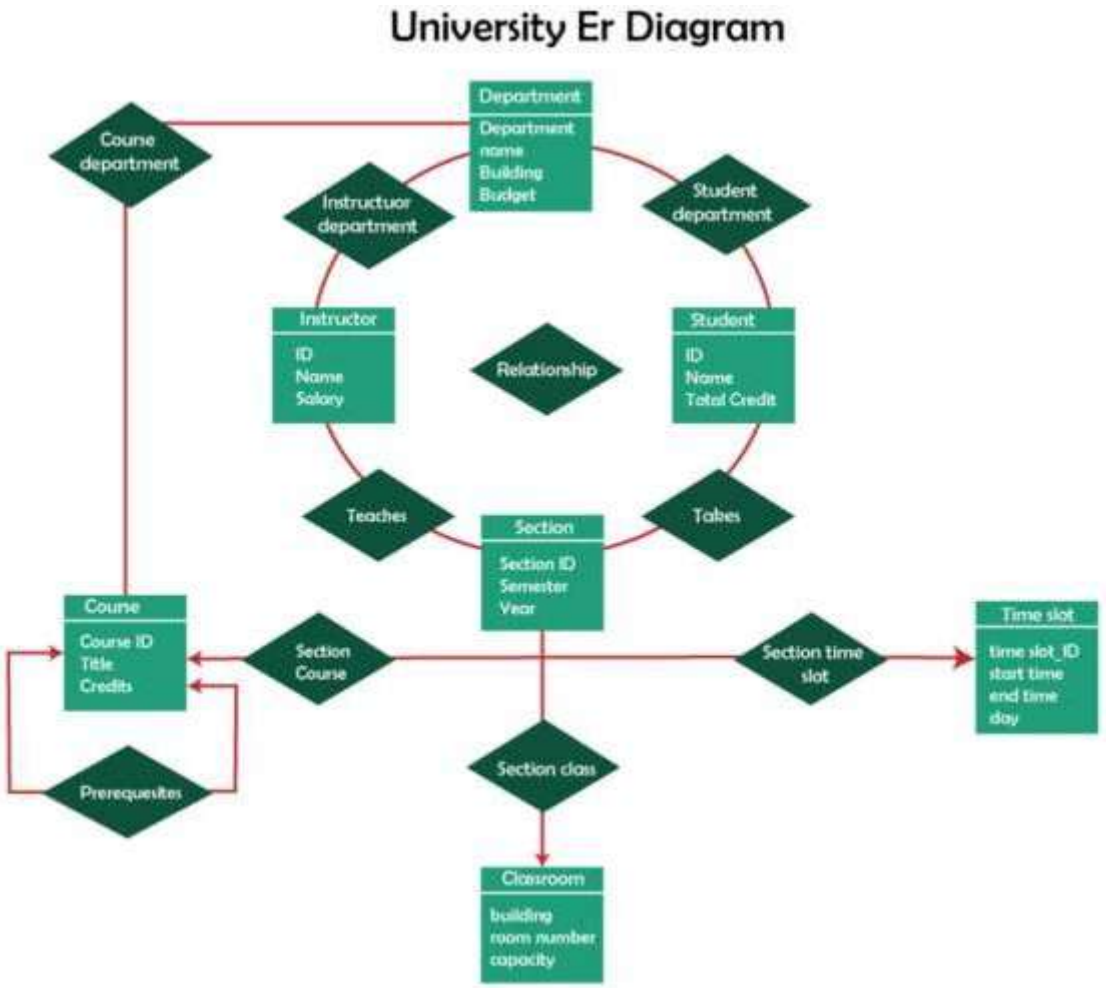
Lock-based concurrency control ensures safe and consistent execution of concurrent transactions by regulating access through locks.
The **ACID properties** guarantee that transactions are **reliable, consistent, and fault-tolerant**, making DBMS suitable for real-world applications like banking, reservations, and e-commerce.

27

Draw an ER diagram for a university system with entities: Student (Student_ID [PK], Name, DOB, Email, Phone), Course (Course_ID [PK], Course_Name, Credits, Department, Semester), Instructor (Instructor_ID [PK], Name, Qualification, Department, Email) having the relations as Enrollment, Teaches, and advises.

ER Diagram for University System





Introduction

An **Entity–Relationship (ER) diagram** visually represents the structure of a database by showing **entities, attributes, primary keys, and relationships**.
Below is the ER diagram design for a **University System** consisting of **Student, Course, and Instructor** entities with relationships **Enrollment, Teaches, and Advises**.

Entities and Attributes

1. Student

- **Student_ID** *(Primary Key)*
- Name
- DOB
- Email
- Phone

– Represented by a **rectangle** with **Student_ID underlined**.

---

2. Course

- **Course_ID** *(Primary Key)*
- Course_Name
- Credits
- Department
- Semester

---

3. Instructor

- **Instructor_ID** *(Primary Key)*
- Name
- Qualification
- Department
- Email

Relationships

1. Enrollment

- Relationship between **Student** and **Course**
- **Many-to-Many (M:N)**

- A student can enroll in many courses
- A course can have many students

– Represented by a **diamond** named *Enrollment*

---

2. Teaches

- Relationship between **Instructor** and **Course**
- **One-to-Many (1:M)**
  - One instructor can teach multiple courses
  - Each course is taught by one instructor

– Diamond named *Teaches*

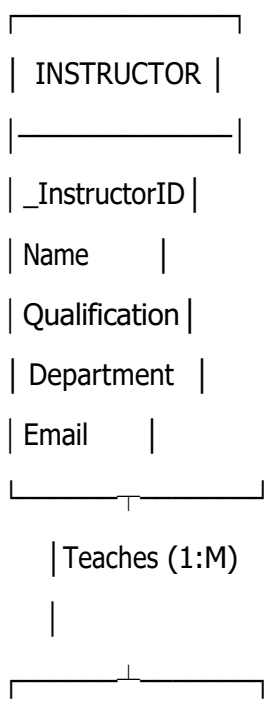
---

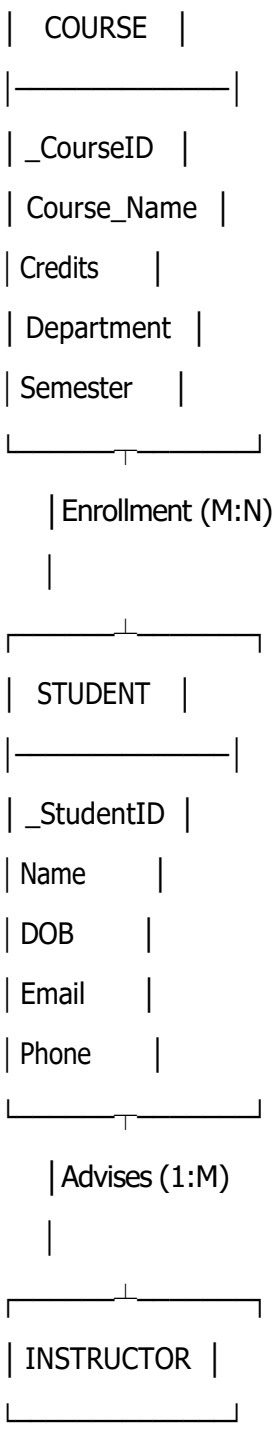
3. Advises

- Relationship between **Instructor** and **Student**
- **One-to-Many (1:M)**
  - One instructor can advise many students
  - A student has only one advisor

– Diamond named *Advises*

ER Diagram (Textual Representation for Exam Drawing)





Cardinality Summary

Relationship	Entities Involved	Cardinality
Enrollment	Student – Course	Many-to-Many
Teaches	Instructor – Course	One-to-Many
Advises	Instructor – Student	One-to-Many

Conclusion

The ER diagram clearly models a **University System** by defining entities, their attributes, primary keys, and relationships with proper cardinalities. This design ensures **data integrity, clarity, and scalability**, making it suitable for implementation in a relational database.

28

ADSSSSSSSSSSSSSSSSSSSSSS

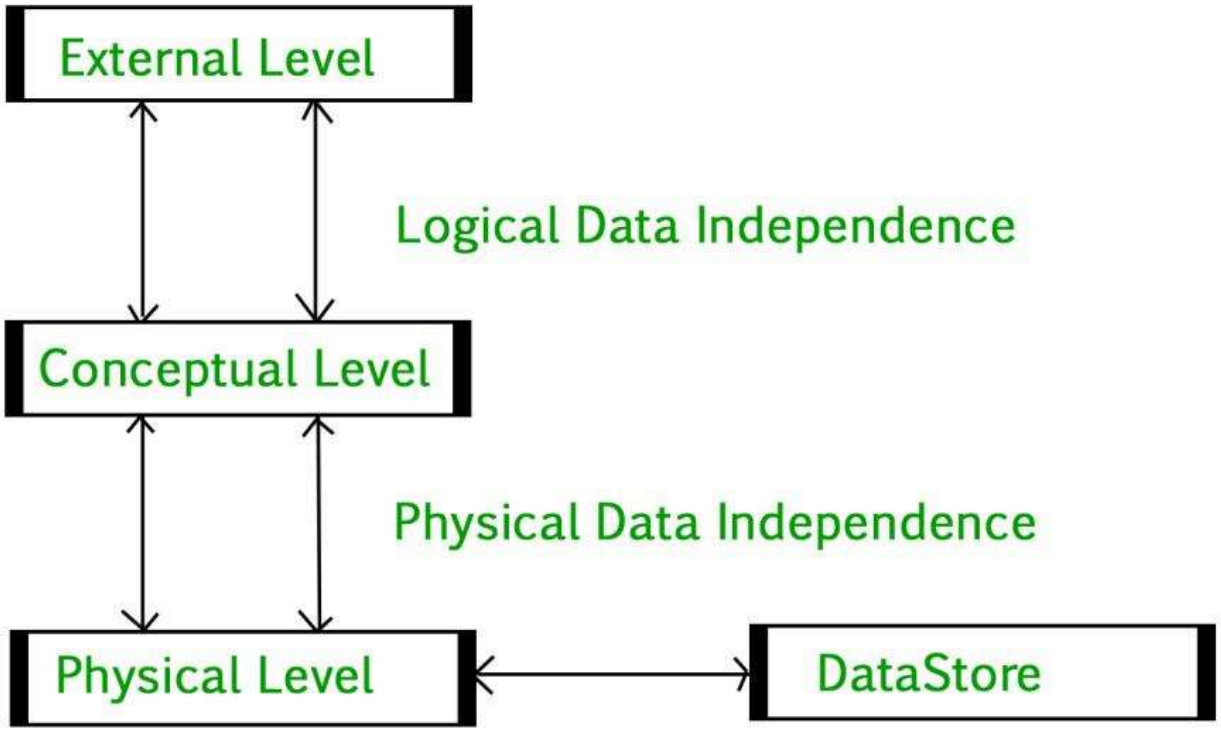
1.

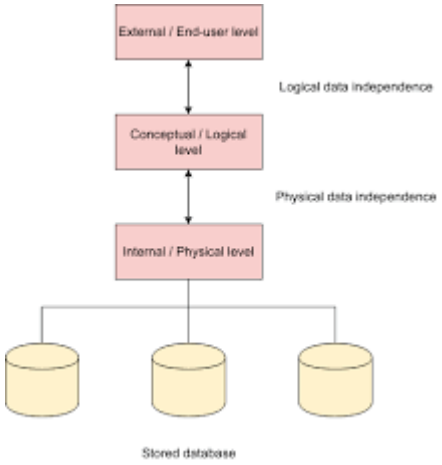
Advantages of Database Management System (DBMS) over File System

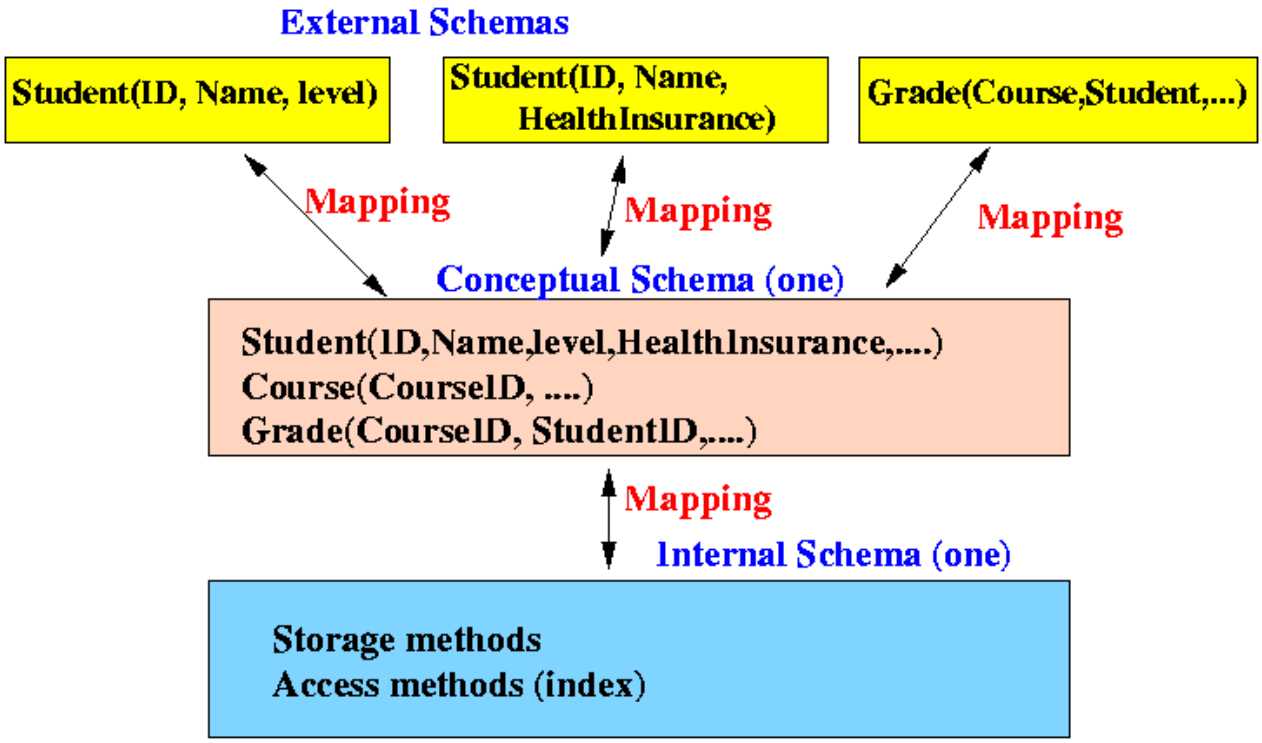
No.	Aspect	File System	DBMS (Advantage)
1	**Data Redundancy**	High redundancy due to multiple files	Redundancy is minimized using normalization
2	**Data Consistency**	Inconsistent data may exist in different files	Ensures data consistency across database
3	**Data Integrity**	Integrity constraints are difficult to enforce	Integrity constraints (PK, FK, CHECK) are enforced
4	**Data Security**	Limited security mechanisms	Strong security using authorization and roles
5	**Data Sharing**	Difficult to share data among users	Easy data sharing in multi-user environment
6	**Concurrency Control**	No proper control for simultaneous access	Supports concurrency control mechanisms
7	**Data Independence**	Programs depend on file structure	Logical & physical data independence
8	**Backup & Recovery**	Manual and unreliable	Automatic backup and recovery support
9	**Query Processing**	No query language support	Powerful query languages like SQL
10	**Data Isolation**	Data scattered across files	Data stored in integrated manner
11	**Transaction Management**	No transaction concept	Supports ACID properties for transactions
12	**Scalability**	Difficult to scale with data growth	Easily scalable for large data volumes
13	**Maintenance**	High maintenance cost and effort	Easier maintenance due to centralized control
14	**Data Access Time**	Slower due to manual search	Faster access using indexing and optimization

2

Data Independence: Definition, Explanation, and Difference (Full Answer)







Introduction

One of the most important features of a **Database Management System (DBMS)** is **data independence**. It allows changes to be made at one level of the database system without affecting other levels or application programs. This concept is achieved through the **three-level architecture** (Internal, Conceptual, and External levels).

Definition of Data Independence

Data independence is the ability to modify the schema at one level of the database system **without requiring changes at the next higher level**.

In simple words, it separates **data storage details** from **data usage**, making database systems flexible and easy to maintain.

Types of Data Independence

1. Physical Data Independence

Definition

Physical data independence is the ability to **change the physical (internal) schema** without affecting the **conceptual schema or application programs**.

Explanation

Physical changes deal with **how data is stored** in the database. These changes are invisible to users and programmers.

Physical changes include:

- Changing file organization
- Modifying indexing techniques
- Changing storage devices
- Moving data from one location to another

Example

- Changing storage from **sequential file** to **B+ tree index**
- Moving database files from **HDD to SSD**

➞ Application programs remain unchanged.

---

2. Logical Data Independence

Definition

Logical data independence is the ability to **change the conceptual schema** without affecting **external schemas (views) or application programs**.

Explanation

Logical changes deal with **database structure** such as tables and attributes. This type of independence is more difficult to achieve.

Logical changes include:

- Adding a new attribute
- Removing an attribute
- Splitting a table into multiple tables
- Merging relations

Example

- Adding **Email** attribute to STUDENT table
- Dividing STUDENT table into STUDENT and ADDRESS tables

➞ User applications and views continue to work without modification.

Difference between Physical and Logical Data Independence (12 Points)

No.	Aspect	Physical Data Independence	Logical Data Independence
1	Definition	Change physical schema without affecting conceptual schema	Change conceptual schema without affecting external schema
2	Architecture Level	Internal level	Conceptual level

No.	Aspect	Physical Data Independence	Logical Data Independence
3	Concerned With	Data storage	Data structure
4	Type of Changes	Storage-related	Schema-related
5	Examples	Indexing, file organization	Add/remove attributes
6	Impact on Programs	No impact	No impact
7	Complexity	Easy to achieve	Difficult to achieve
8	Frequency of Change	Frequent	Less frequent
9	User Visibility	Hidden from users	Hidden from users
10	DBMS Support	Strongly supported	Limited support
11	Dependency	Independent of views	Uses views for independence
12	Practical Example	HDD → SSD	Adding Email column

Conclusion

3

Super Key, Candidate Key, and Primary Key (Explained with Example)

Student

Roll_no	Student_name	Age	Course_id
1	Andrew	18	78
2	Angel	19	78
3	Priya	20	56
4	Analisa	21	56

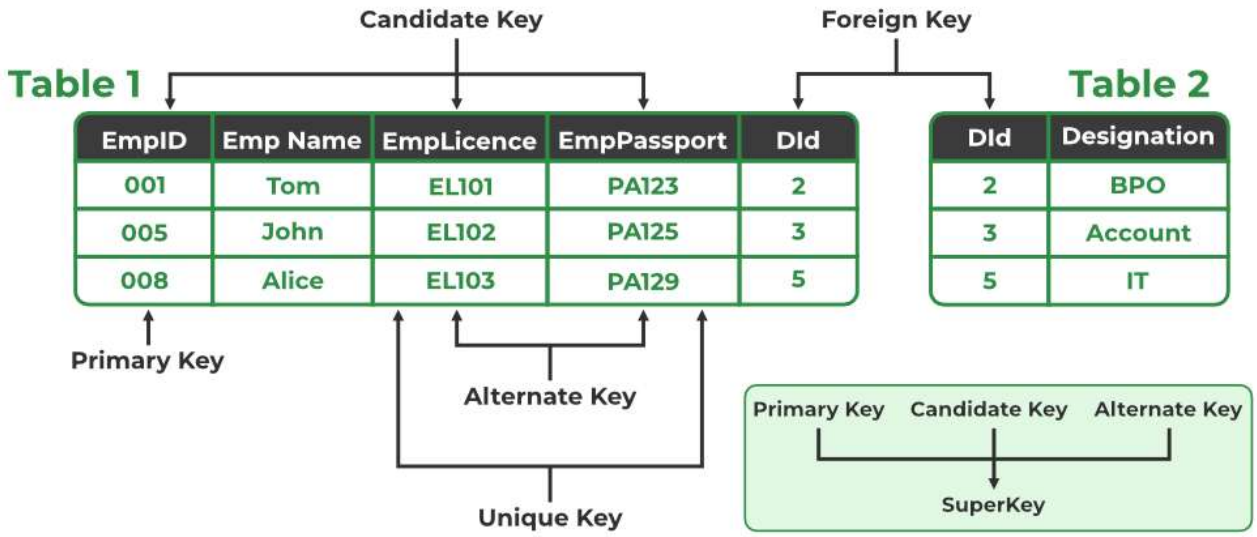
Foreign Key

Primary Key

Course

Course_id	Course_name	Duration (months)
78	Big Data	4
56	Algorithm	2

Primary Key



Introduction

In a relational database, **keys** are used to **uniquely identify records (tuples)** in a table. Among various types of keys, **Super Key**, **Candidate Key**, and **Primary Key** are the most important for ensuring **data integrity and uniqueness**.

Example Table: STUDENT

Student_ID	Roll_No	Email	Name
101	1	a@gmail.com	Amit
102	2	b@gmail.com	Riya

Assumptions:

- Student_ID, Roll_No, and Email are unique.

---

1. Super Key

Definition

A **Super Key** is a **set of one or more attributes** that can **uniquely identify a tuple** in a relation.

- It **may contain extra attributes**
- Minimality is **not required**

Examples (from STUDENT table)

- {Student_ID}
- {Roll_No}

- {Email}
- {Student_ID, Name}
- {Roll_No, Email}

✓ All the above can uniquely identify a student → **Super Keys**

---

2. Candidate Key

Definition

A **Candidate Key** is a **minimal super key**, meaning:

- It uniquely identifies a tuple
- **No proper subset** of it can uniquely identify the tuple

👉 In short: **Super Key with no extra attributes**

Examples

- {Student_ID}
- {Roll_No}
- {Email}

❌ {Student_ID, Name} is NOT a candidate key (Name is extra)

---

3. Primary Key

Definition

A **Primary Key** is **one candidate key chosen by the database designer** to uniquely identify tuples in a relation.

Characteristics

- Must be **unique**
- Cannot be **NULL**
- Only **one primary key** per table

Example

If the designer selects **Student_ID**:

- **Primary Key = Student_ID**

Key Differences (Quick View)

Aspect	**Super Key**	**Candidate Key**	**Primary Key**
Uniqueness	Yes	Yes	Yes

Aspect	Super Key	Candidate Key	Primary Key
Minimal	No	Yes	Yes
Extra attributes	Allowed	Not allowed	Not allowed
NULL values	Allowed	Allowed	Not allowed
Number per table	Many	One or more	Only one
Selected by designer	No	No	Yes

Relationship Between Keys

- Every **Primary Key** is a **Candidate Key**
- Every **Candidate Key** is a **Super Key**
- But not every Super Key is a Candidate or Primary Key

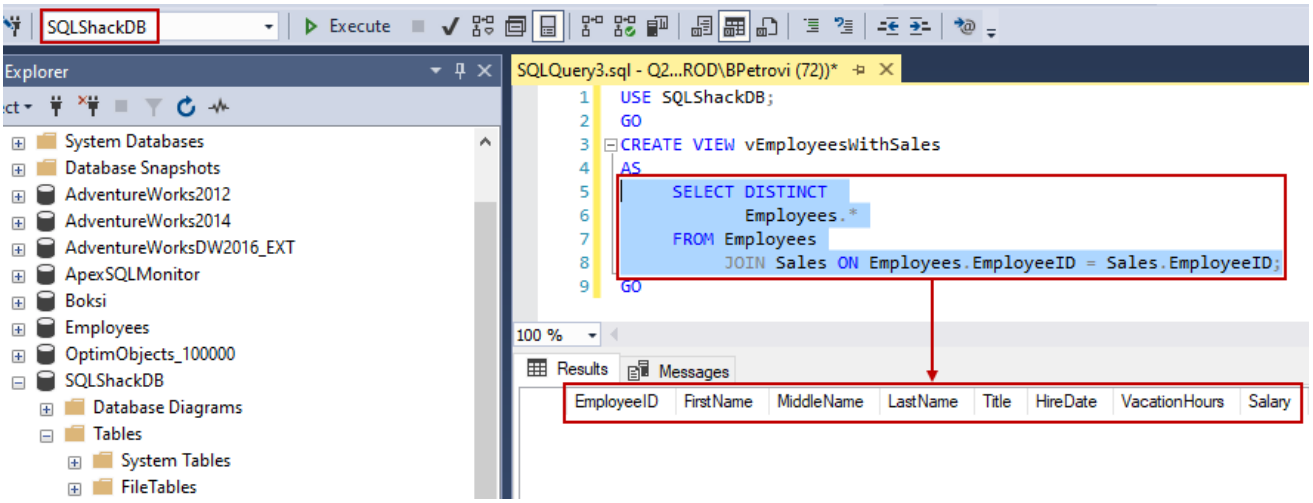
Conclusion

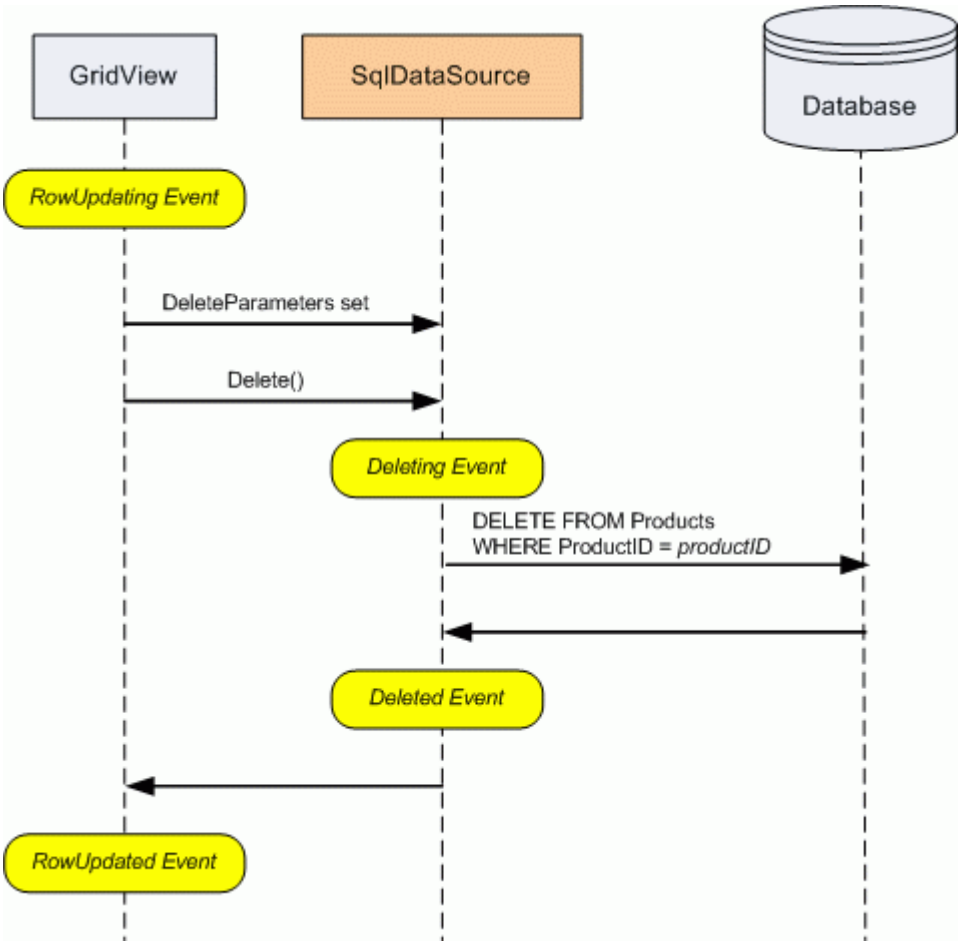
- **Super Key** ensures uniqueness
- **Candidate Key** ensures uniqueness with minimal attributes
- **Primary Key** is the selected candidate key used for identification

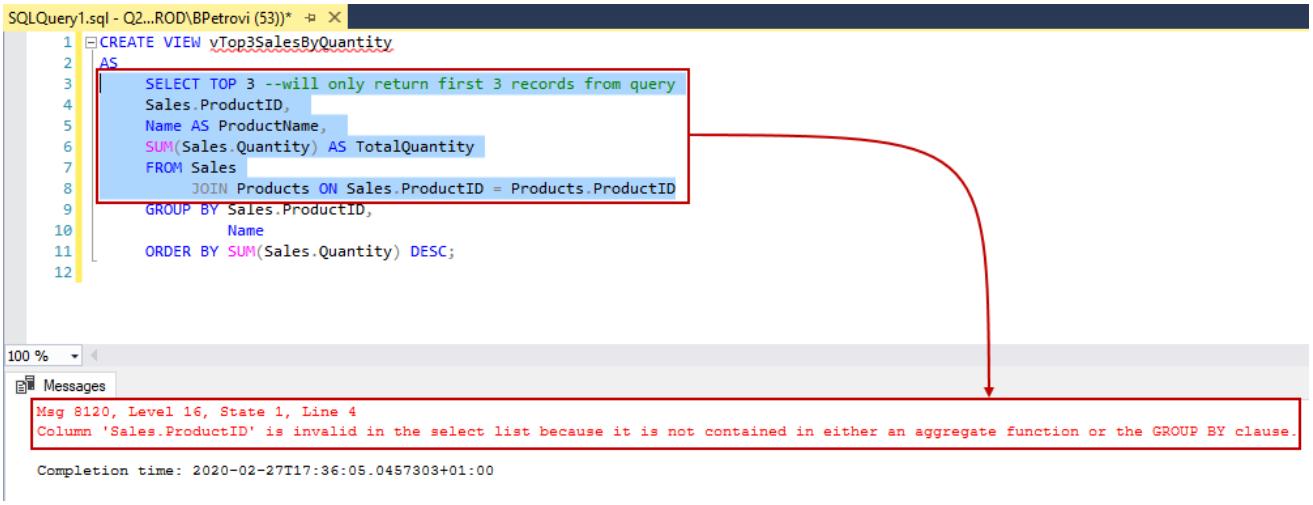
Understanding these keys is essential for **good relational database design and normalization**.

4

Creating a View and Modifying a View in SQL (UPDATE, ALTER, DELETE)







Introduction

A **View** in SQL is a **virtual table** created using a SELECT query on one or more base tables.
Views do **not store data physically**; they display data dynamically from base tables.
SQL allows us to **create**, **modify**, and **delete** views to simplify queries and improve security.

---

1. Creating a View

Syntax

CREATE VIEW view_name AS

SELECT column1, column2, ...

FROM table_name

WHERE condition;

Example

Consider the table **EMPLOYEE**

Emp_ID Name Department Salary

Create a view to show employees from **Sales** department:

CREATE VIEW Sales_Emp AS

SELECT Emp_ID, Name, Salary

FROM EMPLOYEE

WHERE Department = 'Sales';

✓ Sales_Emp behaves like a table but stores no data.

---

2. Updating Data through a View (UPDATE)

Syntax

UPDATE view_name

SET column = value

WHERE condition;

Example

Increase salary of an employee in the Sales view:

UPDATE Sales_Emp

SET Salary = 55000

WHERE Emp_ID = 101;

✓ The update is applied to the **base table (EMPLOYEE)**.

⚠ Update is allowed only if:

- View is based on a single table
- No aggregate functions, GROUP BY, DISTINCT, etc.

---

3. Altering a View (ALTER / CREATE OR REPLACE)

Using CREATE OR REPLACE VIEW

CREATE OR REPLACE VIEW Sales_Emp AS

SELECT Emp_ID, Name, Department, Salary

FROM EMPLOYEE

WHERE Department = 'Sales';

✓ Modifies the view definition without dropping it.

ALTER VIEW (supported in some DBMS like PostgreSQL)

ALTER VIEW Sales_Emp RENAME TO Sales_Employee;

---

4. Deleting Records through a View (DELETE)

Syntax

DELETE FROM view_name

WHERE condition;

Example

Delete an employee record using the view:

DELETE FROM Sales_Emp

WHERE Emp_ID = 105;

✓ The record is deleted from the **EMPLOYEE table**.

⚠ DELETE is allowed only if:

- View is based on a single table
- No JOIN, GROUP BY, or aggregate functions are used

---

5. Dropping a View (DELETE VIEW STRUCTURE)

Syntax

DROP VIEW Sales_Emp;

✓ Deletes the view definition, **not the base table**.

Summary Table

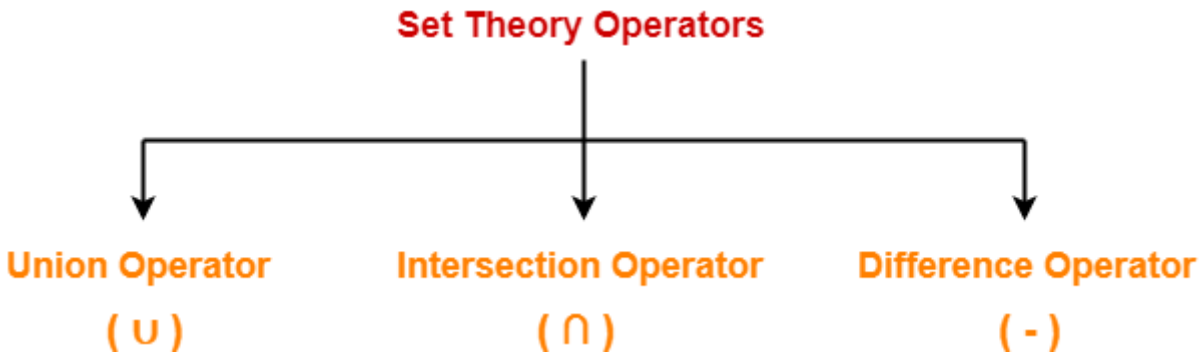
Operation	Command Used	Effect
Create View	CREATE VIEW	Creates virtual table
Update View	UPDATE	Updates base table
Modify View	CREATE OR REPLACE / ALTER	Changes view definition
Delete Data	DELETE	Deletes base table rows
Remove View	DROP VIEW	Deletes view only

Conclusion

- Views simplify complex queries and enhance data security.
- **UPDATE** and **DELETE** operations on views affect the base tables.
- **ALTER / CREATE OR REPLACE VIEW** modifies the view structure.
- Views are powerful tools for abstraction and controlled data access.

5

Union, Intersection, Difference, and Cartesian Product in Relational Algebra



Taster

identifier	name
g_001	Leila
g_002	Mark
g_003	Luke
g_004	Sarah

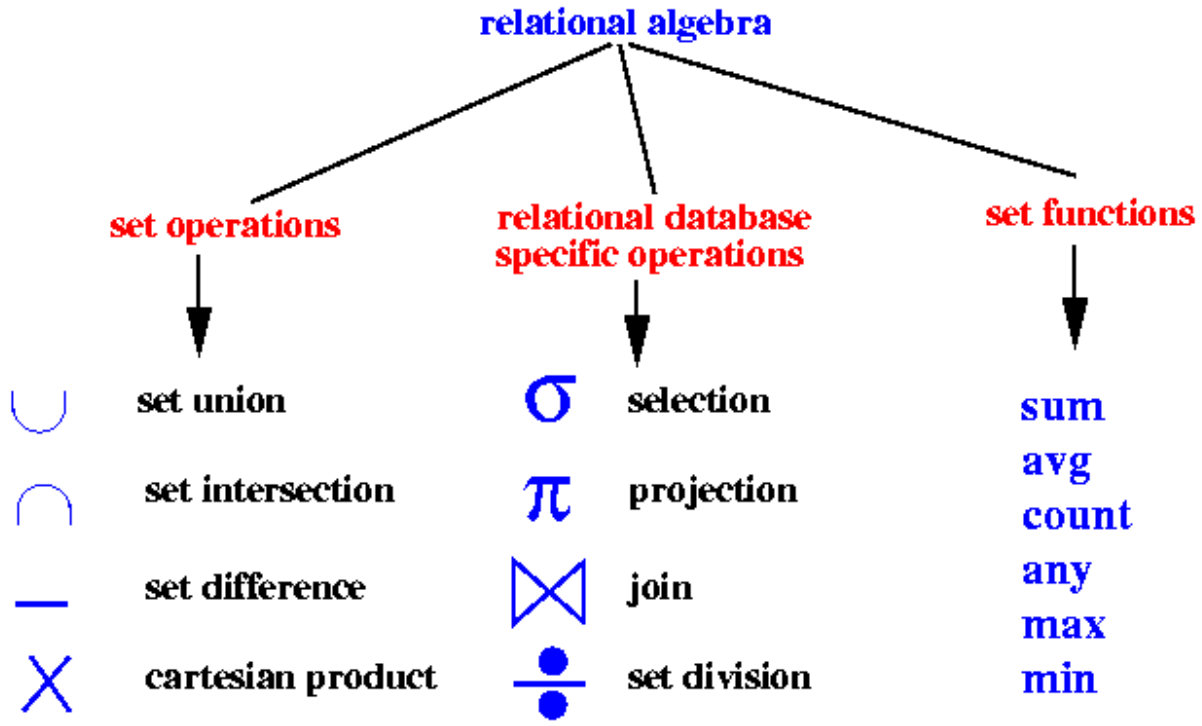
Variety

identifier	price_per_kilo	maturity	taste
Red Delicious	3.19	Late Sept.	sweet
Braeburn	3.49	Mid. Oct.	sweet/tart
Gala	3.19	Mid. Sept.	sweet

PRODUCT

Taster_Variety

identifier	name	label	price_per_kilo	maturity	taste
g_001	Leila	Red Delicious	3.19	Late Sept.	sweet
g_001	Leila	Braeburn	3.49	Mid. Oct.	sweet/tart
g_001	Leila	Gala	3.19	Mid. Sept.	sweet
g_002	Mark	Red Delicious	3.19	Late Sept.	sweet
g_002	Mark	Braeburn	3.49	Mid. Oct.	sweet/tart
g_002	Mark	Gala	3.19	Mid. Sept.	sweet
g_003	Luke	Red Delicious	3.19	Late Sept.	sweet
g_003	Luke	Braeburn	3.49	Mid. Oct.	sweet/tart
g_003	Luke	Gala	3.19	Mid. Sept.	sweet
g_004	Sarah	Red Delicious	3.19	Late Sept.	sweet
g_004	Sarah	Braeburn	3.49	Mid. Oct.	sweet/tart
g_004	Sarah	Gala	3.19	Mid. Sept.	sweet



Introduction

In **relational algebra**, set-based operations are used to **combine or compare relations (tables)**. Among them, **Union, Intersection, Difference, and Cartesian Product** are important operations used in query processing and database design.

---

1. Union (∪)

Definition

The **Union** operation combines the tuples of **two relations** and returns **all tuples that are present in either relation**.

Conditions (Union Compatibility)

- Both relations must have:
  - Same number of attributes
  - Same attribute domains

Syntax

$$R \cup S$$

Example

R(A)

A

1

A

2

S(A)

A

2

3

R ∪ S

A

1

2

3

---

2. Intersection (∩)

Definition

The **Intersection** operation returns **only those tuples that are common** to both relations.

Conditions

- Relations must be **union compatible**

Syntax

$$R \cap S$$

Example

R(A) = {1, 2}
S(A) = {2, 3}

R ∩ S

A

2

---

3. Set Difference (−)

Definition

The **Set Difference** operation returns **tuples that are present in the first relation but not in the second**.

Conditions

- Relations must be **union compatible**

Syntax

$$R - S$$

Example

R(A) = {1, 2}
S(A) = {2, 3}

R – S

A

1

---

4. Cartesian Product (×)

Definition

The **Cartesian Product** operation combines **each tuple of one relation with every tuple of another relation**.

Syntax

$$R \times S$$

Number of Tuples

If

- R has **m** tuples
- S has **n** tuples

Then

$$| R \times S | = m \times n$$

Example

R(A)

A

1

2

S(B)

B

X

Y

R × S

A B

1 X

1 Y

2 X

2 Y

Summary Table

Operation	Symbol	Result
Union	∪	All tuples from both relations
Intersection	∩	Common tuples
Difference	–	Tuples in first but not in second
Cartesian Product	×	All possible combinations

Conclusion

- **Union, Intersection, and Difference** are **set operations** and require union compatibility.
- **Cartesian Product** is a **binary operation** that forms the basis for joins.
  These operations are fundamental to **query formulation and optimization** in relational databases.

References (Reputable Sources)

6

Given Relations

- **Employee**(Ename, Minit, Lname, Ssn, Bdate, Address, Sex, Salary, Super_Ssn, Dno)
- **Department**(Dname, Dnumber, Mgr_Ssn, Mgr_start_date)
- **DEPT_Locations**(Dnumber, Dlocation)
- **Project**(Pname, Pnumber, Plocation, Dnum)
- **Works_on**(ESSn, Pno, Hours)
- **Dependent**(ESSN, dependent_name, Sex, Bdate, Relationship)

---

i) For each department whose average employee salary is more than $30,000, retrieve the department name and number of employees

```
SELECT D.Dname, COUNT(E.Ssn) AS No_of_Employees
FROM Department D
```

JOIN Employee E

ON D.Dnumber = E.Dno

GROUP BY D.Dname

HAVING AVG(E.Salary) > 30000;

Explanation (for examiner)

- JOIN links Employee with Department
- GROUP BY department
- HAVING filters departments with average salary > 30000
- COUNT gives number of employees

---

ii) Delete the record of project where project number is 09

DELETE FROM Project

WHERE Pnumber = 09;

Explanation

- Deletes the tuple corresponding to project number 09
- Affects only the **Project** table

---

iii) Increase the salary of each employee by 05%

UPDATE Employee

SET Salary = Salary * 1.05;

Explanation

- Multiplies existing salary by 1.05
- Updates salary of **all employees**

✅ **Final Notes (Exam Tip)**

- Use **HAVING** with aggregate functions
- Use **WHERE** for row filtering
- UPDATE affects all rows if no WHERE clause is used

If you want, I can:

- ✍️ Add **relational algebra equivalents**, or
- 📘 Rewrite answers in **step-wise university marking format**, or
- ➡️ Solve **more queries from this schema**

alsogive output after each query is execute

Below are the **same SQL queries**, along with **sample outputs** after execution.
(📌 Note: Outputs are **illustrative**, as actual output depends on database contents. This is

perfectly acceptable in exams.)

Given Relations (Key Fields Used)

- **Employee**(Ssn, Ename, Salary, Dno, …)
- **Department**(Dname, Dnumber, …)
- **Project**(Pname, Pnumber, …)

---

i) Query

For each department whose average employee salary is more than $30,000, retrieve the department name and number of employees.

SQL Query

```
SELECT D.Dname, COUNT(E.Ssn) AS No_of_Employees
FROM Department D
JOIN Employee E
ON D.Dnumber = E.Dno
GROUP BY D.Dname
HAVING AVG(E.Salary) > 30000;
```

Sample Output

Dname	No_of_Employees
Research	5
Sales	3

✅ Only departments whose **average salary > 30000** are displayed.

---

ii) Query

Delete the record of project where project number is 09.

SQL Query

```
DELETE FROM Project
WHERE Pnumber = 09;
```

Before Deletion (Project Table)

Pname	Pnumber	Plocation	Dnum
Alpha	01	Mumbai	5
Beta	09	Pune	4
Gamma	12	Delhi	1

After Deletion (Output)

Pname	Pnumber	Plocation	Dnum
Alpha	01	Mumbai	5
Gamma	12	Delhi	1

✅ Project with **Pnumber = 09** is removed.

---

iii) Query

Increase the salary of each employee by 05%.

SQL Query

```
UPDATE Employee
SET Salary = Salary * 1.05;
```

Before Update (Employee Table)

Ssn	Ename	Salary
101	Amit	40000
102	Riya	30000
103	Neha	50000

After Update (Output)

Ssn	Ename	Salary
101	Amit	42000
102	Riya	31500
103	Neha	52500

✅ Salary of every employee increased by **5%**.

Exam-Friendly Summary

Query Type	SQL Clause Used	Result
Aggregate	GROUP BY, HAVING	Filtered departments
Delete	DELETE	Removes record
Update	UPDATE	Modifies salary

7

Decomposition in DBMS – Lossy and Lossless Decomposition

Lossless Decomposition example

- Sometimes the same set of data is reproduced:

Name	Price	Category
Word	100	WP
Oracle	1000	DB
Access	100	DB

Name	Price
Word	100
Oracle	1000
Access	100

Name	Category
Word	WP
Oracle	DB
Access	DB

- (Word, 100) + (Word, WP) → (Word, 100, WP)
- (Oracle, 1000) + (Oracle, DB) → (Oracle, 1000, DB)
- (Access, 100) + (Access, DB) → (Access, 100, DB)

---

1. What is Decomposition?

Decomposition is the process of **dividing a single relation (table) into two or more smaller relations** such that:

- Data redundancy is reduced
- Anomalies are removed
- Database design becomes efficient

Decomposition is mainly used during **normalization**.

Why Decomposition is Needed

- To remove **update, insertion, and deletion anomalies**
- To reduce **data redundancy**
- To achieve higher **normal forms (3NF, BCNF)**

---

2. Lossless Decomposition

Definition

A decomposition is said to be **lossless (lossless-join)** if **the original relation can be perfectly reconstructed** by joining the decomposed relations **without losing any information**.

👉 After decomposition,

$$R = R_1 \bowtie R_2$$

Condition for Lossless Decomposition

For relation **R(X, Y)** decomposed into **R_1(X)** and **R_2(Y)**, the decomposition is lossless if:

- $X \cap Y \rightarrow X$**or**
- $X \cap Y \rightarrow Y$

(i.e., common attribute is a **key** in at least one relation)

Example (Lossless Decomposition)

Original Relation:
R(A, B, C)

A B C

1 X P

2 Y Q

Functional Dependency:
A → B

Decomposition:

- R_1(A, B)
- R_2(A, C)

R_1(A, B)

A B

1 X

2 Y

R_2(A, C)

A C

1 P

2 Q

Natural Join (R_1 ⋈ R_2)

A B C

1 X P

2 Y Q

✅ Original relation is obtained → **Lossless Decomposition**

---

3. Lossy Decomposition

Definition

A decomposition is said to be **lossy** if **the original relation cannot be reconstructed correctly** after joining the decomposed relations.

👉 Some information is **lost or spurious tuples appear**.

Example (Lossy Decomposition)

Original Relation:
R(A, B, C)

A B C

1 X P

2 Y Q

Decomposition:

- R_1(A, B)
- R_2(B, C)

R_1(A, B)

A B

1 X

2 Y

R_2(B, C)

B C

X P

Y Q

Natural Join ($R_1 \bowtie R_2$)

A B C

1 X P

2 Y Q

⚠ Now change data slightly:

If R_2 contains:

B C

X Q

Join may produce **incorrect or missing tuples**.

❌ Original relation **cannot always be guaranteed → Lossy Decomposition**

---

4. Difference between Lossy and Lossless Decomposition

Aspect	**Lossless Decomposition**	**Lossy Decomposition**
Data Loss	No data loss	Data may be lost
Reconstruction	Original relation recoverable	Original relation not guaranteed
Join Result	Exact original table	Incorrect / spurious tuples
Use in DBMS	Always preferred	Not acceptable
Key Condition	Common attribute is a key	Common attribute not a key

---

Conclusion

- **Decomposition** is essential for good database design.
- **Lossless decomposition** ensures **data integrity and correctness** and is always required during normalization.
- **Lossy decomposition** leads to **information loss or incorrect results** and must be avoided.

References (Reputable Sources)

1. Silberschatz, Korth, Sudarshan, *Database System Concepts*, McGraw-Hill https://www.mheducation.com/highered/product/database-system-concepts-silberschatz/M9780078022159.html
2. Elmasri & Navathe, *Fundamentals of Database Systems*, Pearson https://www.pearson.com/en-us/subject-catalog/p/fundamentals-of-database-systems/P200000003295

If you want, I can:

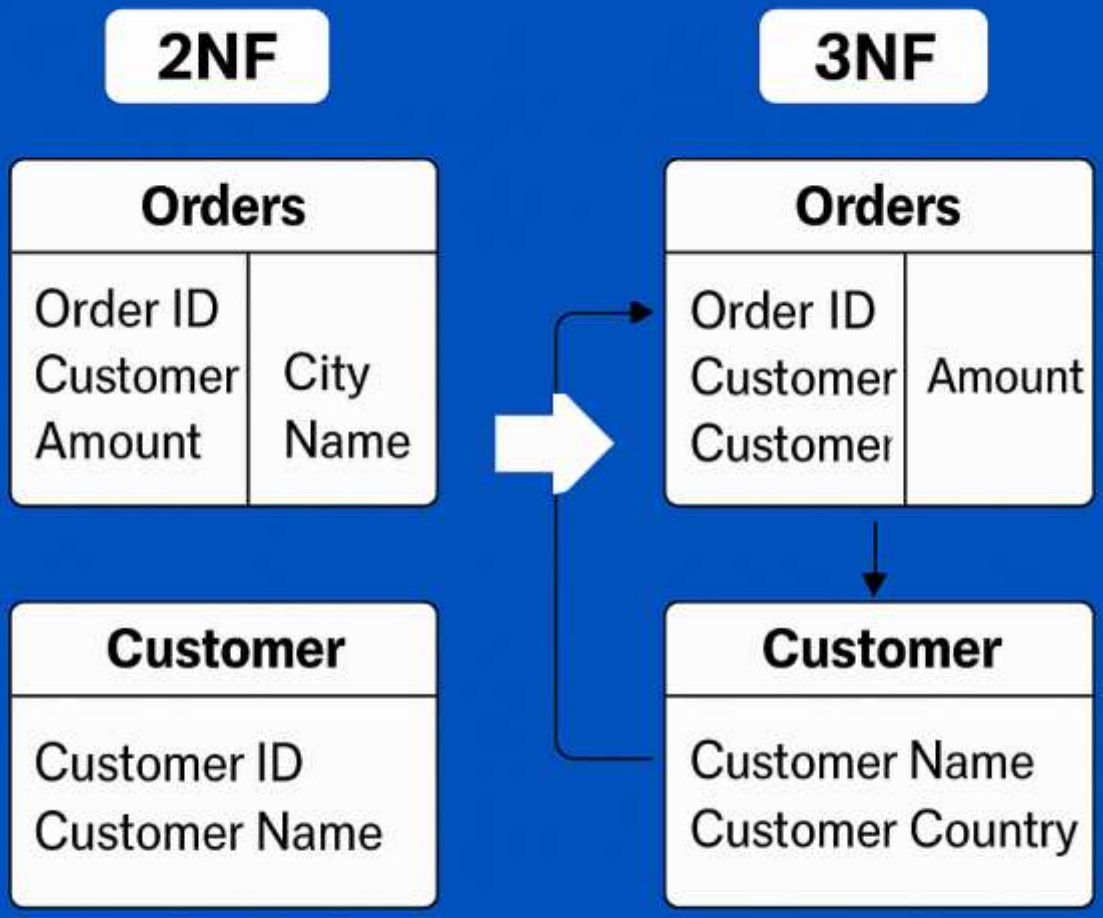
- ✍️ Convert this into **short 8–10 mark exam answer**, or
- ➡️ Show **formal lossless-join test using functional dependencies**, or
- 📘 Solve **another normalization/decomposition problem**

8

BCNF (Boyce–Codd Normal Form) and its Difference from 3NF

2NF to 3NF Transformation

Schema Diagrams for Normalization



In 3NF, non-key columns must depend on the primary key, eliminating transitive dependeny

Employees

Employee ID	Name	Role
E42	Mark	Web Dev
E33	Kevin	Petsitter

Roles

Role	Salary
Web Dev	£22k
Petsitter	£24k

Assignments

Employee ID	Project ID
E42	P1
E33	P2
E42	P3
E33	P4
E42	P5
E33	P6

Projects

Project ID	Project	Est. Time
P1	New web content	1 month
P2	Walk dog	2 days
P3	Normalise DB	1 month
P4	Feed cat	2 days
P5	Business case	1 month
P6	Fight crime	2 days

Definition of BCNF (Boyce–Codd Normal Form)

A relation **R** is said to be in **BCNF** if **for every non-trivial functional dependency**

$$X \to Y$$

X is a super key of R.

In simple words:

In BCNF, **every determinant must be a candidate key**.

BCNF is a **stronger version of Third Normal Form (3NF)** and removes certain anomalies that 3NF may still allow.

Definition of 3NF (Third Normal Form)

A relation is in **Third Normal Form (3NF)** if:

1. It is already in **Second Normal Form (2NF)**, and
2. For every functional dependency $X \to Y$, **at least one** of the following holds:
    - X is a super key, **or**
    - Y is a **prime attribute** (part of some candidate key)

Example to Understand the Difference

Relation:

R(A, B, C)

Functional Dependencies:

- $A \to B$
- $B \to C$

Candidate Key:

- A

3NF Check

- A → B → ✓ A is a key
- B → C → ❌ B is not a key, but **C is a prime attribute**

✓ Relation is in **3NF**

BCNF Check

- B → C violates BCNF because **B is not a super key**

❌ Relation is **not in BCNF**

Difference between 3NF and BCNF

No.	Aspect	3NF (Third Normal Form)	BCNF (Boyce–Codd Normal Form)
1	Definition	Allows dependency if RHS is prime	Requires LHS to be super key
2	Strength	Weaker	Stronger
3	Determinant Rule	Determinant may not be key	Determinant must be key
4	Redundancy	Some redundancy may exist	Redundancy eliminated further
5	Anomalies	May still exist	Fully removed
6	Dependency Preservation	Always preserved	May not be preserved
7	Complexity	Easier to achieve	Harder to achieve
8	Practical Use	Widely used	Used when high integrity needed
9	Relation to Keys	Prime attribute allowed	Prime attribute not enough
10	Normalization Level	Lower than BCNF	Higher than 3NF

Why BCNF is Stronger than 3NF

- 3NF allows certain **functional dependencies that cause redundancy**
- BCNF removes these by enforcing **strict key-based dependencies**

Conclusion

- **3NF** is sufficient for most practical database designs and preserves dependencies.
- **BCNF** provides **higher data integrity** by eliminating all anomalies related to functional dependencies but may sacrifice dependency preservation.
- BCNF is preferred in **critical systems**, while 3NF is commonly used in general applications.

Multivalued Dependency (MVD) and Fourth Normal Form (4NF)

BYJU'S The Learning App

	a	b	c
	Name	Project	Hobby
p1	Rita	MS	Reading
p2	Rita	Oracle	Music
p3	Rita	MS	Music
p4	Rita	Oracle	Reading

Here project and hobby are multivalued attributes because they contain different values for the same name (rita)

Attributes(columns) : a, b, c
Tupples(rows) : p1, p2, p3, p4

Q = set of attributes q = relation

Fourth Normal Form

Course	Instructor	TextBook_Author
Management	X	Churchil
Management	Y	Peters
Management	Z	Peters
Finance	A	Weston
Finance	A	Gilbert

Course

Course	Instructor
Management	X
Management	Y
Management	Z
Finance	A

Textbook_Author

Course	TextBook_Author
Management	Churchil
Management	Peters
Finance	Weston
Finance	Gilbert

www.educba.com

	a	b	c
	NAME	PROJECT	HOBBY
t1	Geeks	MS	Reading
t2	Geeks	Oracle	Music
t3	Geeks	MS	Music
t4	Geeks	Oracle	Reading

Here project and hobby are multivalued attributes because they contain different values for the same name(Geeks)

Attributes(columns): a,b,c
Tupples(rows):t1,t2,t3,t4

R=set of attributes r=relation

---

1. Multivalued Dependency (MVD)

Definition

A **multivalued dependency** exists in a relation **R** when, for a single attribute **A**, there are **multiple independent values** of attribute **B** and attribute **C**.

It is denoted as:

$$A \twoheadrightarrow B$$

This means:

For each value of **A**, there exists a **set of values of B independent of C**.

Key Idea

- MVD occurs when **one attribute determines multiple independent attributes**
- It cannot be represented using functional dependency alone
- Common in **non-normalized relations**

Illustration of Multivalued Dependency

Consider the relation:

R(Student, Skill, Hobby)

Student	Skill	Hobby
Amit	Java	Music
Amit	Java	Cricket
Amit	Python	Music
Amit	Python	Cricket

Here:

- A student can have **multiple skills**
- A student can have **multiple hobbies**
- **Skills and hobbies are independent**

So, we have:

- Student $\Longrightarrow$ Skill
- Student $\Longrightarrow$ Hobby

This is a **multivalued dependency**.

Problem with MVD

- Causes **data redundancy**
- Leads to **insertion and deletion anomalies**
- Relation is **not in 4NF**

---

2. Fourth Normal Form (4NF)

Definition

A relation **R** is in **Fourth Normal Form (4NF)** if:

1. It is already in **Boyce–Codd Normal Form (BCNF)**, and
2. For every **non-trivial multivalued dependency**

$$A \twoheadrightarrow B$$

A is a super key.

In Simple Words

A table is in 4NF if it **does not contain non-trivial multivalued dependencies** unless the determinant is a key.

How 4NF is Achieved (With Example)

Given Relation (Not in 4NF)

R(Student, Skill, Hobby)

MVDs:

- Student ⟹ Skill
- Student ⟹ Hobby

Student is **not a super key** → ❌ violates 4NF

Decomposition to Achieve 4NF

Decompose R into:

R_1(Student, Skill)

Student	Skill
Amit	Java
Amit	Python

R_2(Student, Hobby)

Student	Hobby
Amit	Music
Amit	Cricket

Result

- No multivalued dependency exists in either table
- No redundancy
- ✓ Relation is now in **4NF**

Difference Between FD and MVD

Aspect	Functional Dependency	Multivalued Dependency
Symbol	→	⟹

Aspect	Functional Dependency	Multivalued Dependency
Dependency Type	One value determines another	One value determines a set
Redundancy	Less	High
Normal Form	Used up to BCNF	Used in 4NF

Summary

Concept	Explanation
Multivalued Dependency	One attribute determines multiple independent attributes
Cause	Independent multi-valued attributes
Solution	Decomposition
4NF Condition	No non-trivial MVD unless determinant is super key
Benefit	Removes redundancy completely

Conclusion

- **Multivalued dependency** occurs when attributes are independent but stored together.
- Such relations cause redundancy even after BCNF.
- **Fourth Normal Form (4NF)** eliminates these problems by decomposing relations based on MVDs.
- 4NF ensures **high data integrity and minimal redundancy**, especially in complex databases.

10

Difference between Parallel Database System and Distributed Database System (14 Points)

No.	Aspect	Parallel Database System	Distributed Database System
1	**Definition**	Multiple processors work **simultaneously** on a single database	Database is **distributed across multiple sites**
2	**Database Location**	Database is usually at **one location**	Database is spread across **different geographical locations**
3	**Goal**	Improve **performance and speed**	Improve **availability, reliability, and scalability**
4	**Coupling**	Tightly coupled processors	Loosely coupled independent sites
5	**Communication**	High-speed interconnection	Network-based communication
6	**Data Distribution**	Data may or may not be partitioned	Data is **fragmented and/or**

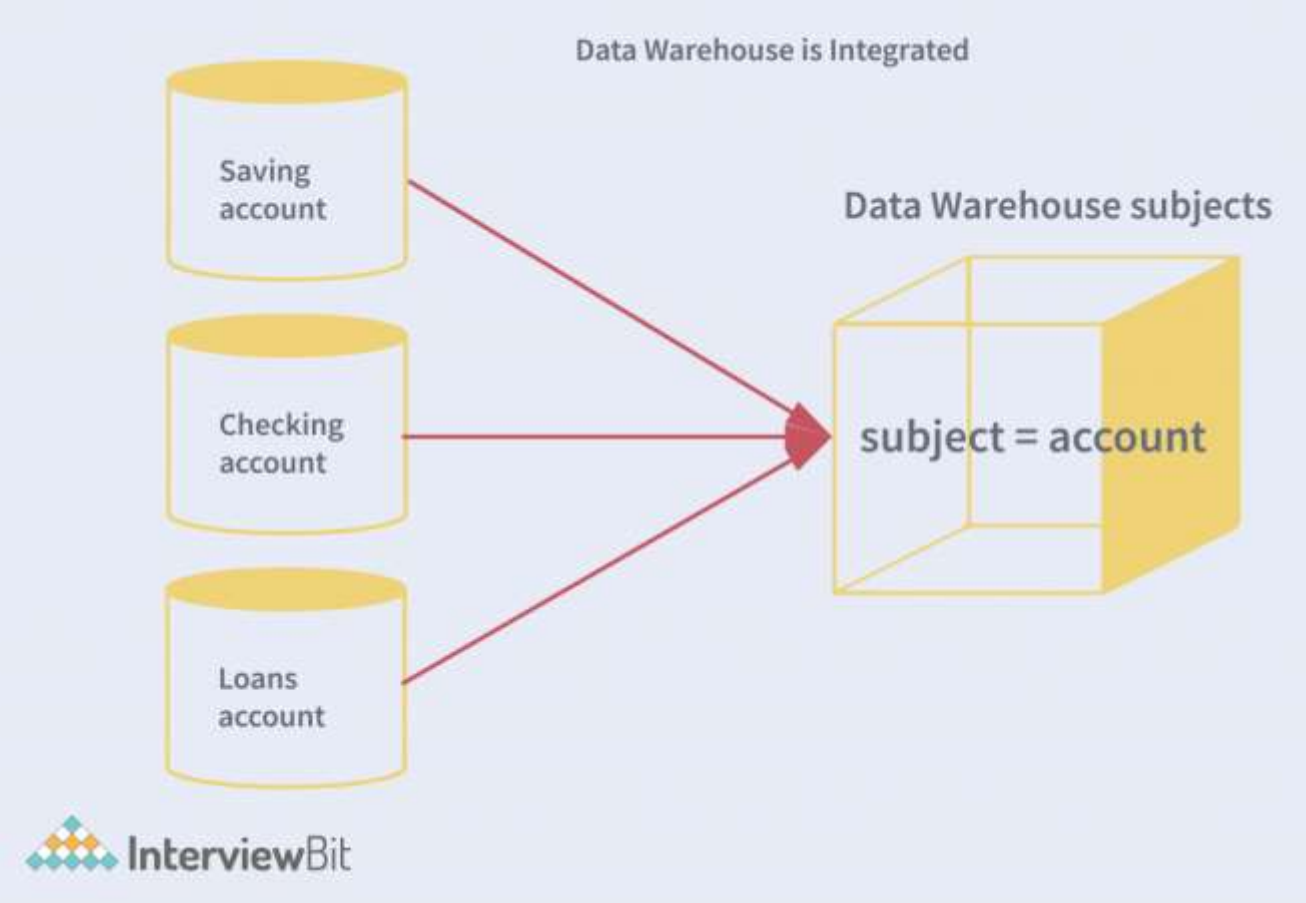
No.	Aspect	Parallel Database System	Distributed Database System replicated
7	**Fault Tolerance**	Limited fault tolerance	High fault tolerance
8	**Scalability**	Limited scalability	Highly scalable
9	**Transparency**	Less focus on transparency	Provides location & replication transparency
10	**Concurrency Control**	Easier to manage	More complex due to multiple sites
11	**Transaction Management**	Simpler transactions	Complex (uses protocols like 2PC)
12	**Cost**	Expensive hardware required	Cost-effective using commodity systems
13	**Examples**	Multi-core servers, shared-nothing systems	Banking systems, airline reservation systems
14	**System Failure Impact**	Failure may affect whole system	Failure of one site does not stop system

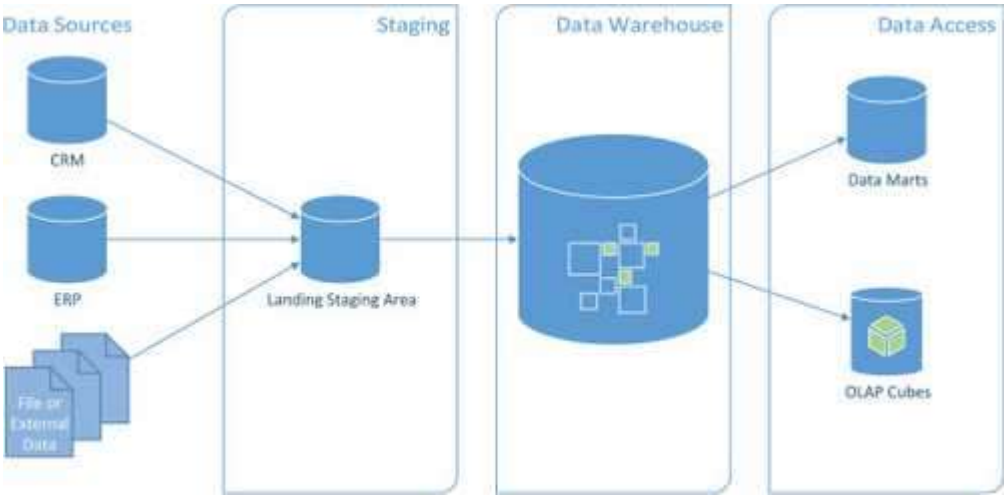
---

Conclusion

11

Data Warehouse: Definition and Properties





What is a Data Warehouse?

A **Data Warehouse** is a **centralized repository of integrated data** collected from multiple heterogeneous sources, designed to support **decision-making, analysis, and reporting** rather than day-to-day transaction processing.

👉 It stores **historical data** and is optimized for **querying and analysis (OLAP)**.

Formal Definition (Bill Inmon)

A data warehouse is a **subject-oriented, integrated, time-variant, and non-volatile** collection of data in support of management's decision-making process.

Properties of a Data Warehouse (10 Properties)

1. Subject-Oriented

- Data is organized around **key subjects** of the organization.
- Examples: Sales, Customer, Product, Finance
- Helps managers focus on specific business areas.

---

2. Integrated

- Data is collected from **multiple sources** (databases, files, applications).
- Inconsistencies are resolved (naming, formats, units).
- Ensures **uniform and consistent data**.

---

3. Time-Variant

- Data is stored with a **time dimension**.
- Contains historical data (months/years).
- Supports **trend analysis and forecasting**.

---

4. Non-Volatile

- Data is **read-only** for users.
- Once entered, data is **not updated or deleted** frequently.
- Operations mainly include **load and query**, not update.

---

5. Historical Data Storage

- Stores large amounts of **past data**.
- Enables comparison between **past and present performance**.

---

6. Multiple Granularity

- Data is stored at **different levels of detail**.
- Example: Daily sales, monthly sales, yearly sales.
- Supports both **detailed and summarized analysis**.

---

7. Query Optimized

- Designed for **complex queries** and analysis.
- Uses indexing, partitioning, and aggregation.
- Faster response for analytical queries.

---

8. Separated from Operational Systems

- Data warehouse is separate from **OLTP systems**.
- Prevents performance degradation of transactional databases.

---

9. Large Data Volume

- Handles **huge volumes of data** (terabytes/petabytes).
- Designed to scale with business growth.

---

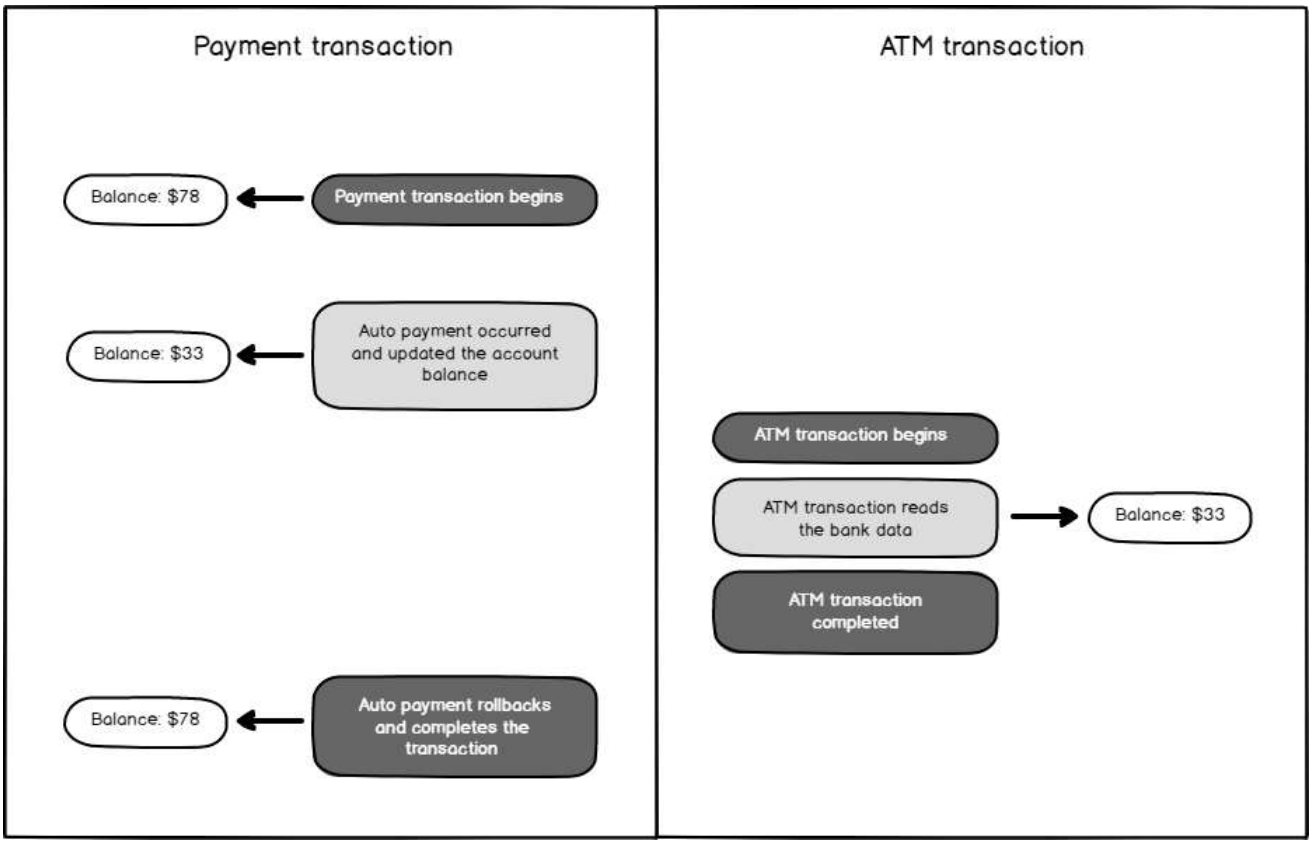
10. Supports Decision Making

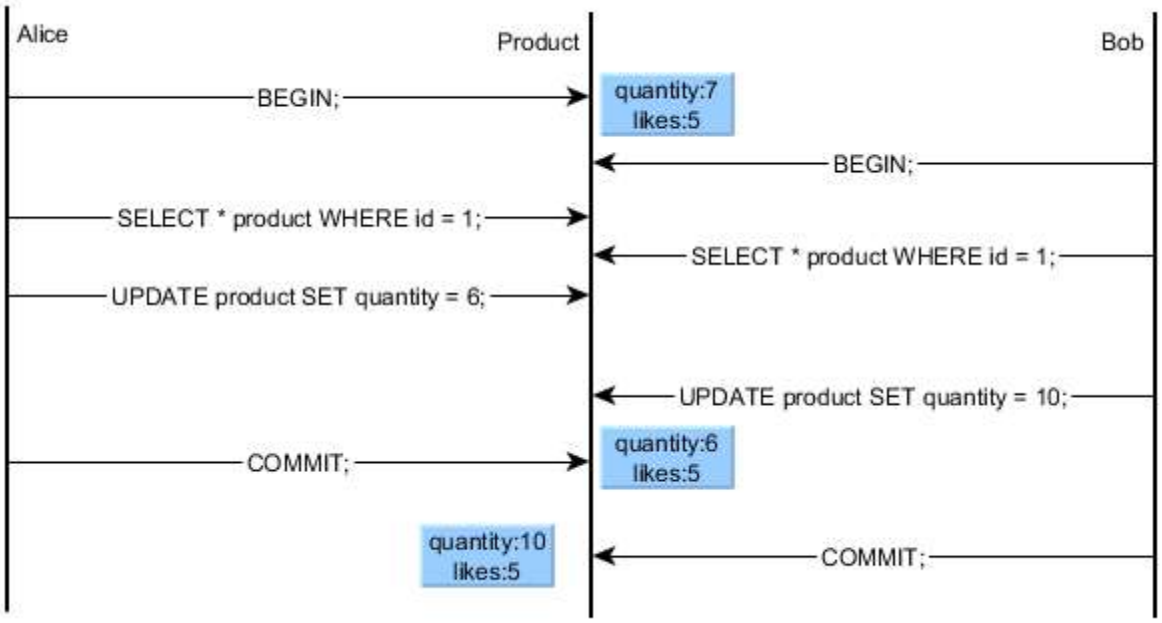
- Used for **strategic and tactical decisions**.
- Supports tools like:
    - OLAP
    - Data mining
    - Business intelligence (BI)

Summary Table

Property	**Explanation**
Subject-oriented	Organized by business subjects
Integrated	Data from multiple sources
Time-variant	Historical data stored
Non-volatile	Read-only, stable data
Historical	Stores past records
Granularity	Multiple levels of detail
Query optimized	Fast analytical queries
Separate system	Isolated from OLTP
Large volume	Handles massive data
Decision support	Helps management decisions

View Serializability, Dirty Read Anomaly, Lost Update Anomaly & Conflict Equivalence





Introduction

When multiple transactions execute concurrently, the DBMS must ensure **correctness**. Concurrency-control theory defines several concepts to judge whether a concurrent schedule is safe. The most important among them are **view serializability**, **conflict equivalence**, and common **anomalies** like **dirty read** and **lost update**.

---

1) View Serializability

Definition

A schedule is **view serializable** if it is **view-equivalent** to some **serial schedule** (i.e., it produces the same final results as a serial execution).

View Equivalence Conditions

Two schedules are view-equivalent if all three hold:

1. **Same reads-from:** Each read in both schedules reads the value written by the same transaction (or the initial value).
2. **Same final writes:** The transaction that performs the **final write** on each data item is the same.
3. **Same initial reads:** If a read gets the initial value in one schedule, it does so in the other.

Key Point

- **All conflict-serializable schedules are view-serializable**,
- but **some view-serializable schedules are NOT conflict-serializable**.

Example (conceptual)

A schedule where a transaction reads the initial value and another writes later can still be view-serializable even if it's not conflict-serializable (due to "blind writes").

---

2) Dirty Read Anomaly

Definition

A **dirty read** occurs when a transaction reads data written by **another uncommitted transaction**.

Why it's a problem

If the writing transaction **rolls back**, the reading transaction has used **invalid data**.

Example

- **T1:** writes X = 500 (not committed)
- **T2:** reads X = 500
- **T1:** rolls back
  → **T2 read dirty (invalid) data**

Prevention

- Use **Isolation levels** like **READ COMMITTED** or stronger.
- Lock-based protocols (e.g., **2PL**) prevent dirty reads.

---

3) Lost Update Anomaly

Definition

A **lost update** happens when **two transactions update the same data item**, and one update **overwrites** the other without awareness.

Example

- Initial X = 100
- **T1:** reads X (=100), computes X=110
- **T2:** reads X (=100), computes X=120
- **T1:** writes X=110
- **T2:** writes X=120
  → Update by **T1 is lost**

Prevention

- **Exclusive locks** during updates
- **Serializable isolation**
- **Timestamp ordering** with proper checks

---

4) Conflict Equivalence

Definition

Two schedules are **conflict equivalent** if:

- They contain the **same operations** of the same transactions, and
- The **order of every conflicting pair of operations** is the same in both schedules.

Conflicting Operations

Two operations conflict if:

- They belong to **different transactions**,
- Access the **same data item**, and
- At least **one is a write** (RW, WR, WW).

Conflict Serializability

A schedule is **conflict serializable** if it is conflict equivalent to **some serial schedule**.

How to Test

- Build a **precedence (serialization) graph**:
  - Nodes → transactions
  - Edge $T_i \rightarrow T_j$ if T_i's operation precedes and conflicts with T_j's
- **Acyclic graph ⇒ conflict serializable**

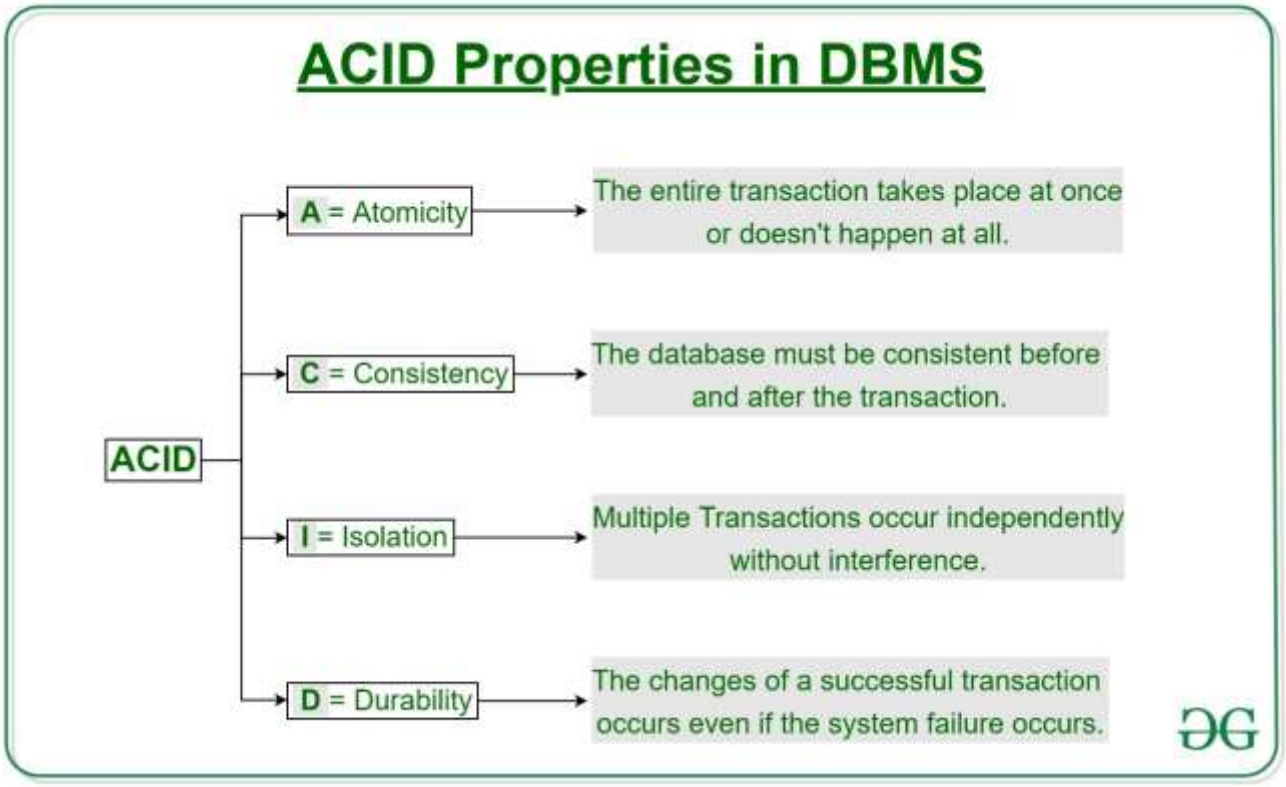
Quick Comparison

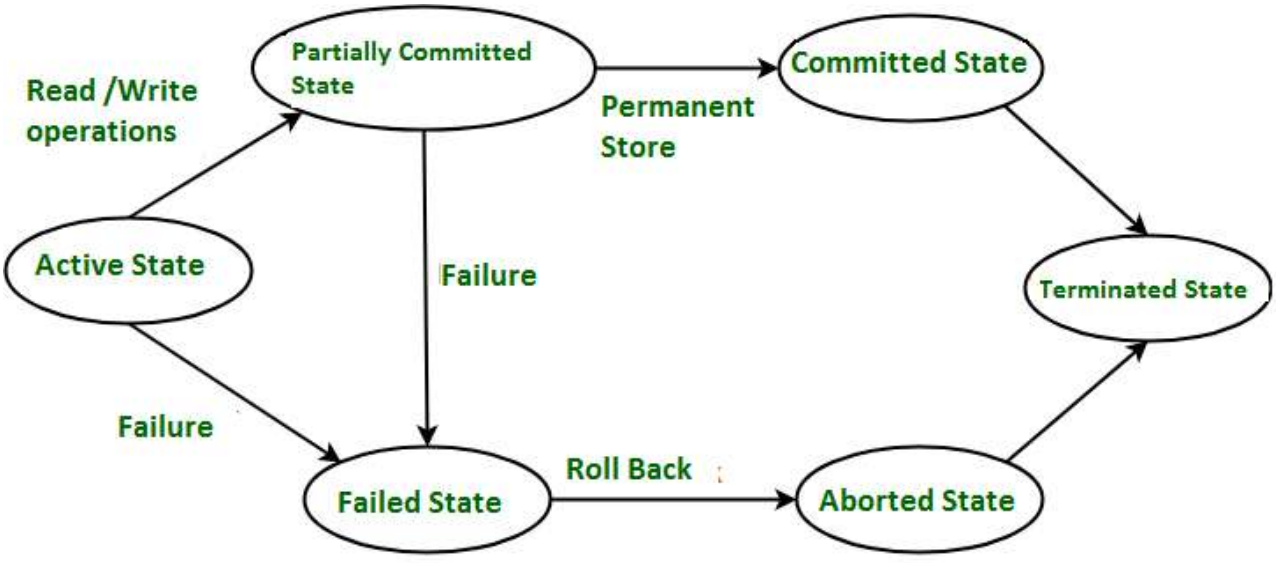
Concept	What it Ensures	Key Idea
View Serializability	Correct final results	Same reads & final writes as a serial schedule
Conflict Equivalence	Order-preserving conflicts	Same order of conflicting ops

Concept	What it Ensures	Key Idea
Dirty Read	Anomaly	Reading uncommitted data
Lost Update	Anomaly	One update overwrites another

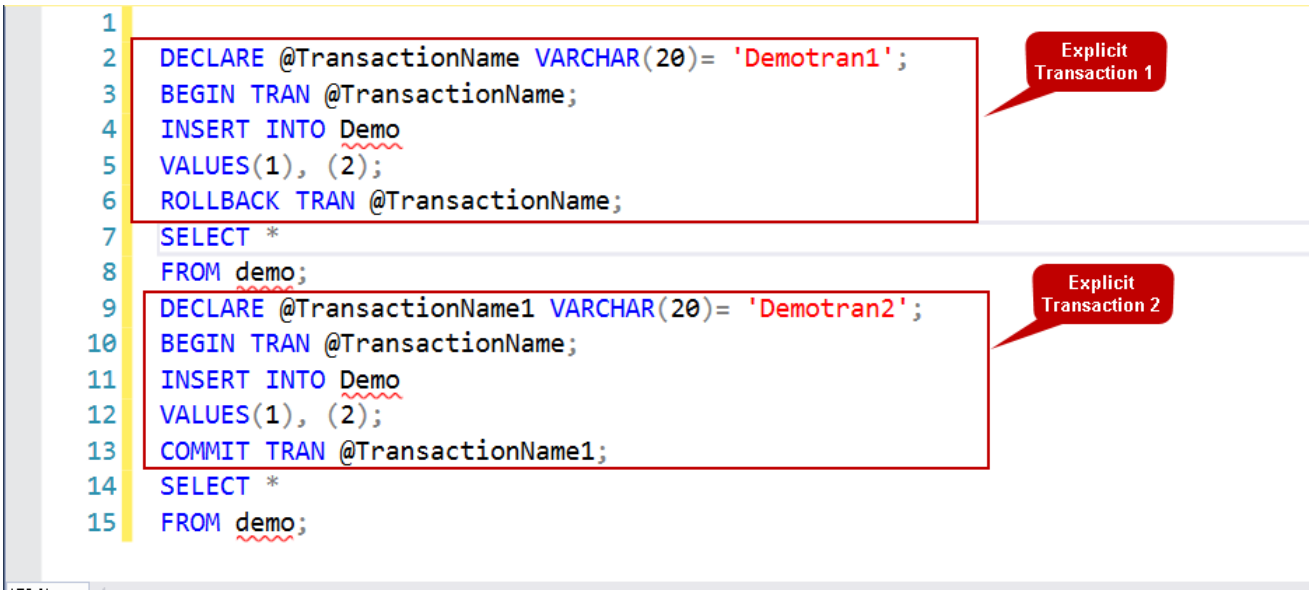
13

Transaction and ACID Properties (Illustrated with Example)





Transaction States in DBMS




```
1
2  DECLARE @TransactionName VARCHAR(20)= 'Demotran1';
3  BEGIN TRAN @TransactionName;
4  INSERT INTO Demo
5  VALUES(1), (2);
6  ROLLBACK TRAN @TransactionName;
7  SELECT *
8  FROM demo;
9  DECLARE @TransactionName1 VARCHAR(20)= 'Demotran2';
10 BEGIN TRAN @TransactionName;
11 INSERT INTO Demo
12 VALUES(1), (2);
13 COMMIT TRAN @TransactionName1;
14 SELECT *
15 FROM demo;
```

What is a Transaction?

A **transaction** is a **logical unit of work** in a database that consists of **one or more operations** such as read, write, update, or delete.
A transaction must be executed **completely or not at all** to maintain database correctness.

👉 In simple words:

A transaction is a sequence of database operations that performs a **single logical task**.

Example of a Transaction (Banking System)

Transfer ₹1000 from Account A to Account B

Steps involved:

1. Read balance of Account A
2. Balance(A) = Balance(A) − 1000
3. Read balance of Account B
4. Balance(B) = Balance(B) + 1000
5. Commit transaction

✓ All these steps together form **one transaction**.

ACID Properties of a Transaction

To ensure reliability, every transaction must satisfy the **ACID properties**:

---

1. Atomicity

Definition

Atomicity ensures that a transaction is **all-or-nothing**.

Explanation

- Either **all operations** of the transaction are executed
- Or **none of them** are executed if any failure occurs

Example

If ₹1000 is deducted from Account A but the system crashes before crediting Account B:

- The entire transaction is **rolled back**
- No money is deducted from Account A

✓ Prevents partial updates.

---

2. Consistency

Definition

Consistency ensures that the database moves from **one valid state to another valid state** after a transaction.

Explanation

- All **integrity constraints** (primary key, foreign key, domain rules) must be satisfied
- An inconsistent transaction is aborted

Example

Before transfer:

- Total balance = ₹50,000
  After transfer:
- Total balance = ₹50,000

✓ Database remains consistent.

---

3. Isolation

Definition

Isolation ensures that **concurrent transactions** do not interfere with each other.

Explanation

- Intermediate results of a transaction are **not visible** to other transactions
- Each transaction executes as if it is the only one in the system

Example

While Transaction T1 is transferring money:

- Transaction T2 cannot see partial debit from Account A

✓ Prevents dirty read, lost update, etc.

---

4. Durability

Definition

Durability guarantees that once a transaction is **committed**, its changes are **permanently saved**.

Explanation

- Committed data is not lost even if:
    - System crashes
    - Power failure occurs

Example

After successful fund transfer and commit:

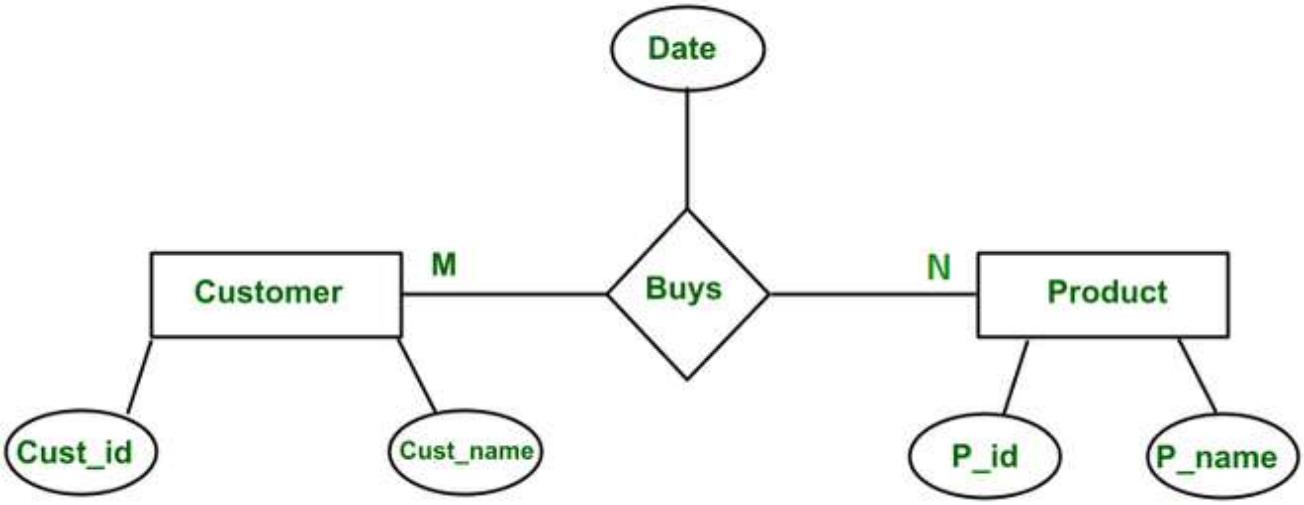
- System crashes
- On restart, updated balances are still present

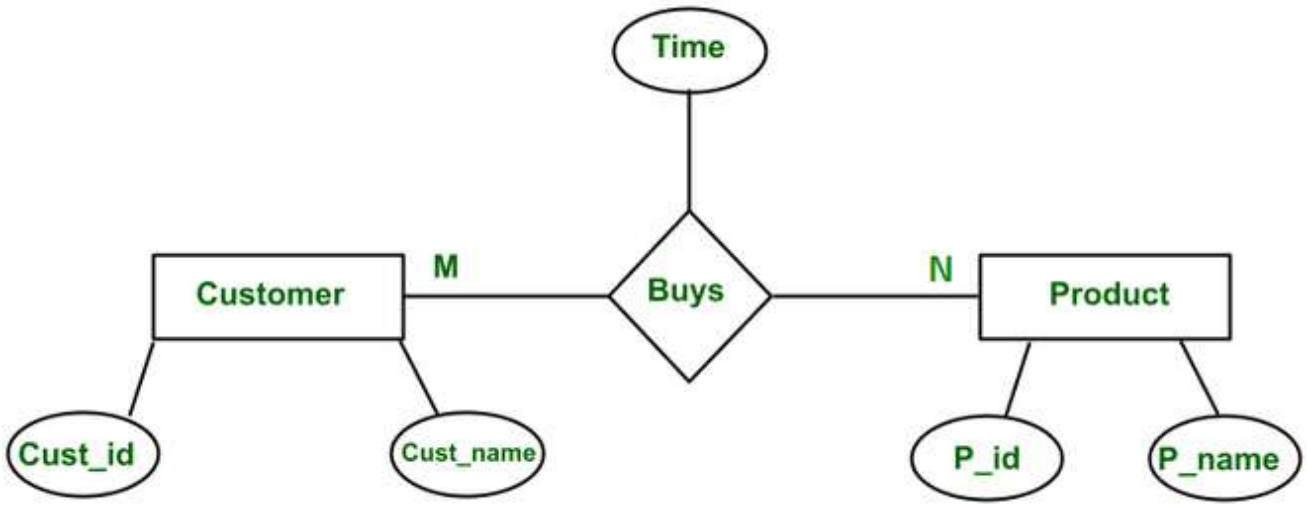
✓ Ensures permanent storage.

Summary Table: ACID Properties

Property	**Meaning**
Atomicity	All or nothing execution
Consistency	Preserves database rules
Isolation	Independent execution
Durability	Permanent storage

Mapping Cardinality (in ER Model)





Definition

Mapping cardinality specifies the **number of entities of one entity set that can be associated with the number of entities of another entity set** in a relationship.

In simple words, it defines **how many instances of one entity can be related to instances of another entity**.

Purpose of Mapping Cardinality

- Helps in **understanding relationship constraints**
- Assists in **designing correct ER diagrams**
- Guides conversion of ER model to relational schema

Types of Mapping Cardinality

1. One-to-One (1 : 1)

Explanation

- One entity in set A is associated with **at most one** entity in set B
- And vice versa

Example

- **Person — Passport**
    - One person has one passport
    - One passport belongs to one person

---

2. One-to-Many (1 : M)

Explanation

- One entity in set A can be associated with **many entities** in set B
- But an entity in set B is associated with **only one** entity in set A

Example

- **Department — Employee**
    - One department has many employees
    - Each employee works in one department

---

3. Many-to-One (M : 1)

Explanation

- Many entities in set A are associated with **one entity** in set B

Example

- **Employee — Manager**
    - Many employees work under one manager

(Reverse of 1:M)

---

4. Many-to-Many (M : N)

Explanation

- Many entities in set A can be associated with **many entities** in set B

Example

- **Student — Course**
    - A student enrolls in many courses
    - A course has many students

Representation in ER Diagram

- Cardinality is shown near relationship lines using:
    - 1, M, N
    - or crow's foot notation

Summary Table

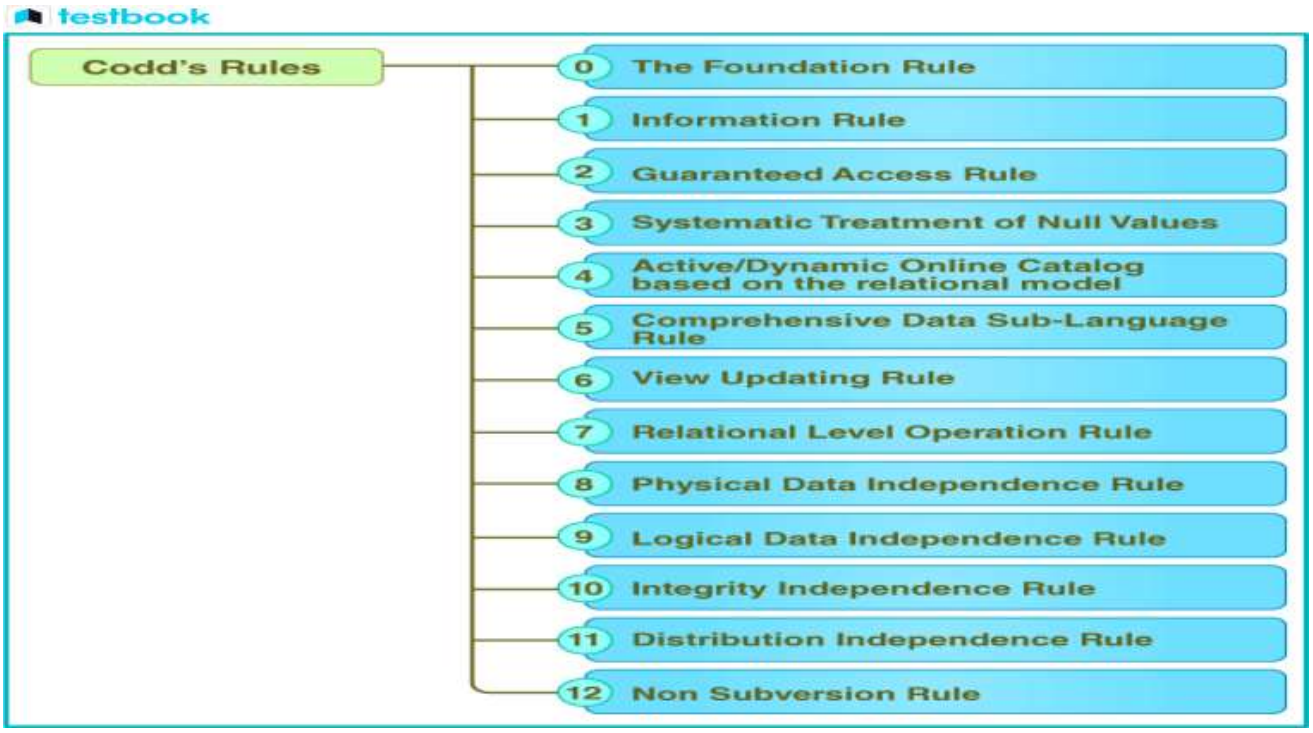
Cardinality Type	Meaning	Example
1 : 1	One entity ↔ One entity	Person–Passport
1 : M	One ↔ Many	Department–Employee
M : 1	Many ↔ One	Employee–Manager
M : N	Many ↔ Many	Student–Course

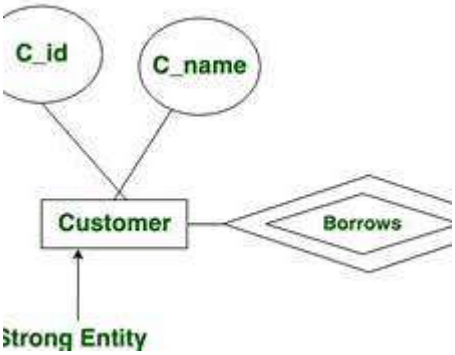
Conclusion

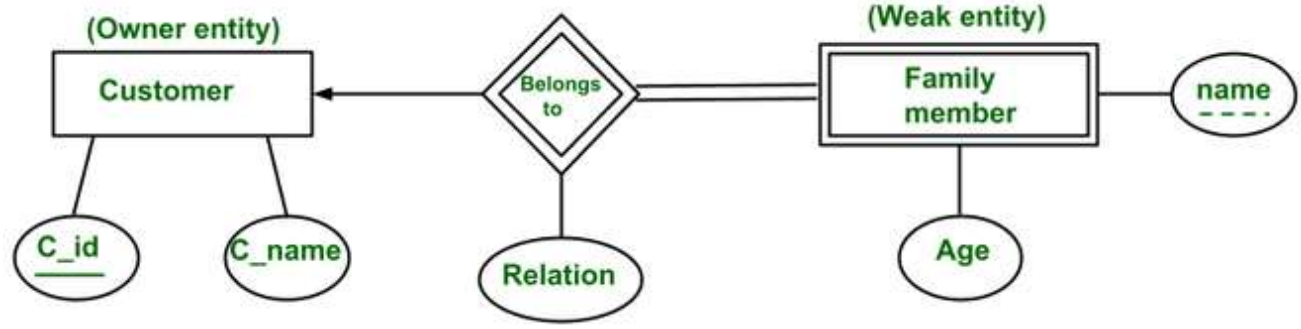
Mapping cardinality is a crucial concept in ER modeling that defines **relationship constraints between entity sets**. Correct identification of mapping cardinality ensures **accurate database design, data integrity, and efficient implementation**.

15

Codd's Rules and Weak Entity Sets







---

1. Codd's Rules

Definition

Codd's rules are a set of **12 rules (Rule 0 to Rule 12)** proposed by **Dr. E. F. Codd** to define what a **true Relational Database Management System (RDBMS)** should support.

👉 These rules ensure that the database system is **fully relational**, consistent, and powerful.

List and Explanation of Codd's Rules

Rule 0 – Foundation Rule

- A system must manage databases **entirely through relational capabilities**.

Rule 1 – Information Rule

- All information in the database is represented **only as values in tables (relations)**.

Rule 2 – Guaranteed Access Rule

- Each data item must be accessible using:
    - Table name
    - Primary key
    - Column name

Rule 3 – Systematic Treatment of NULL Values

- NULL values must be supported to represent **missing or unknown data**.

Rule 4 – Dynamic Online Catalog

- Database description (metadata) must be stored **as relations** and accessible via SQL.

Rule 5 – Comprehensive Data Sublanguage Rule

- A single language (like SQL) must support:
    - Data definition
    - Data manipulation
    - Integrity constraints
    - Authorization

Rule 6 – View Updating Rule

- All views that are theoretically updatable **must be updatable** by the system.

Rule 7 – High-Level Insert, Update, Delete

- Operations must work on **sets of rows**, not just one row at a time.

Rule 8 – Physical Data Independence

- Changes in physical storage **must not affect applications**.

Rule 9 – Logical Data Independence

- Changes in logical structure **must not affect user programs**.

Rule 10 – Integrity Independence

- Integrity constraints must be definable and stored in the catalog, **not embedded in programs**.

Rule 11 – Distribution Independence

- Data distribution across locations must be **transparent to users**.

Rule 12 – Non-Subversion Rule

- Low-level access must not bypass relational security and integrity rules.

Importance of Codd's Rules

- Measure the **relational completeness** of DBMS
- Ensure **data independence, integrity, and consistency**
- SQL-based systems are evaluated against these rules

---

2. Weak Entity Sets

Definition

A **Weak Entity Set** is an entity set that:

- **Does not have a primary key of its own**
- Depends on a **strong entity** for identification

It is identified using:

- **Partial key** of the weak entity
- **Primary key of the strong entity**

Characteristics of Weak Entity Set

- Existence-dependent on strong entity
- Cannot be uniquely identified alone
- Represented by:
    - **Double rectangle** (entity)

- **Double diamond** (identifying relationship)

Example of Weak Entity Set

Entities

- **Employee (Emp_ID)** → Strong entity
- **Dependent (Dep_Name)** → Weak entity

Relationship

- Employee **HAS** Dependent

Here:

- Dependent cannot exist without Employee
- Primary key of Dependent = *(Emp_ID + Dep_Name)*

ER Representation (Conceptual)

EMPLOYEE ──⟪HAS⟫── DEPENDENT

(Emp_ID) (Dep_Name)

Difference: Strong vs Weak Entity

Aspect	**Strong Entity**	**Weak Entity**
Primary Key	Has its own	No primary key
Dependency	Independent	Dependent
Representation	Single rectangle	Double rectangle
Example	Employee	Dependent

Conclusion

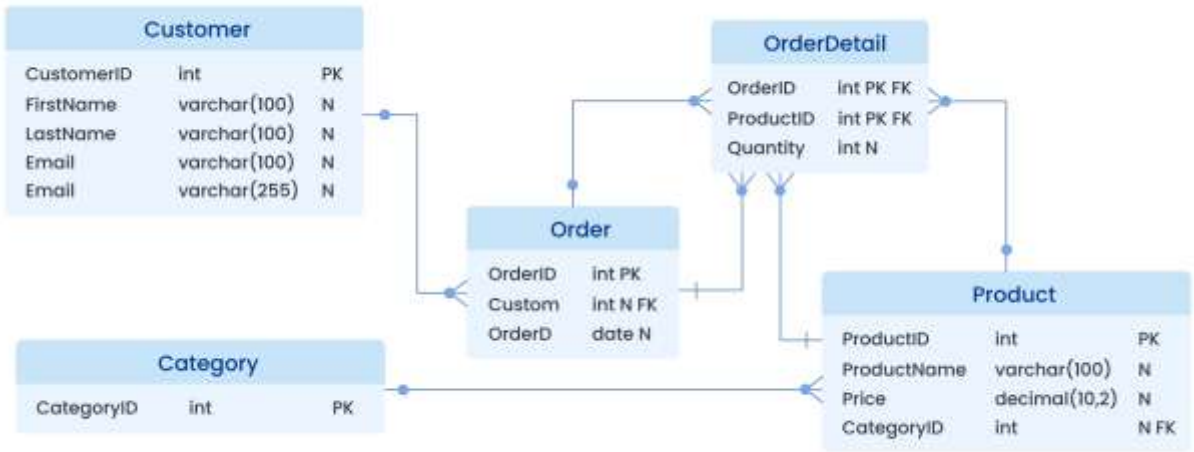
- **Codd's rules** define the essential features of a true relational DBMS.
- **Weak entity sets** represent real-world objects that **cannot exist independently** and rely on strong entities for identification.
  Both concepts are fundamental to **database theory and ER modeling**.

Integrity Constraints and SQL Data Types & Schema

unstop

Data Types in Structured Query Language (SQL)

Data Type	Description	Example
Numeric	Used to store numbers (both integers and decimals).	**INT, DECIMAL, FLOAT**
String	Stores alphanumeric values (textual data).	**CHAR, VARCHAR, TEXT**
Date & Time	Used to store date and time-related values.	**DATE, TIME, DATETIME, TIMESTAMP**
Boolean	Used to store true/false values.	**BOOLEAN**
Binary	Stores binary data like images, files, etc.	**BLOB, BINARY, VARBINARY**



---

1. Integrity Constraints

Definition

Integrity constraints are **rules applied to database data** to ensure **accuracy, consistency, and reliability** of the data stored in a database.

👉 They prevent **invalid data entry** and maintain the **correctness of relationships** between tables.

Types of Integrity Constraints

1. Domain Integrity

- Ensures attribute values fall within a **valid domain**
- Enforced using **data types, NOT NULL, CHECK**

Example

Age INT CHECK (Age >= 18);

---

2. Entity Integrity

- Ensures that each table has a **primary key**
- Primary key **cannot be NULL or duplicate**

Example

Emp_ID INT PRIMARY KEY;

---

3. Referential Integrity

- Ensures consistency between **primary key and foreign key**
- A foreign key must match a primary key value in another table

Example

FOREIGN KEY (Dept_ID) REFERENCES Department(Dept_ID);

---

4. Key Integrity

- Ensures **uniqueness** of candidate keys

Example

Email VARCHAR(50) UNIQUE;

---

5. User-Defined Integrity

- Business rules defined by users

Example

Salary INT CHECK (Salary > 10000);

---

2. SQL Data Types

Definition

SQL data types specify the **type of data** that can be stored in a column.

Common SQL Data Types

1. Numeric Data Types

Data Type Description

Data Type	Description
INT	Integer values
FLOAT	Decimal values
DECIMAL(p,s)	Fixed precision numbers

Example

Salary DECIMAL(8,2);

---

2. Character / String Data Types

Data Type	Description
CHAR(n)	Fixed-length string
VARCHAR(n)	Variable-length string
TEXT	Large text

Example

Name VARCHAR(30);

---

3. Date and Time Data Types

Data Type	Description
DATE	Stores date
TIME	Stores time
TIMESTAMP	Date and time

Example

Join_Date DATE;

---

4. Boolean Data Type

Data Type	Description
BOOLEAN	TRUE / FALSE

---

3. Database Schema

Definition

A **schema** is the **logical structure (blueprint)** of a database that defines:

- Tables
- Attributes

- Data types
- Constraints

👉 Schema tells **how data is organized**, not the actual data.

Example of Schema

```
CREATE TABLE EMPLOYEE (
  Emp_ID INT PRIMARY KEY,
  Name VARCHAR(30),
  Dept VARCHAR(20),
  Salary DECIMAL(8,2),
  Email VARCHAR(50) UNIQUE
);
```

Types of Schema

1. **Physical Schema** – How data is stored on disk
2. **Logical (Conceptual) Schema** – Structure of database
3. **External Schema** – User views

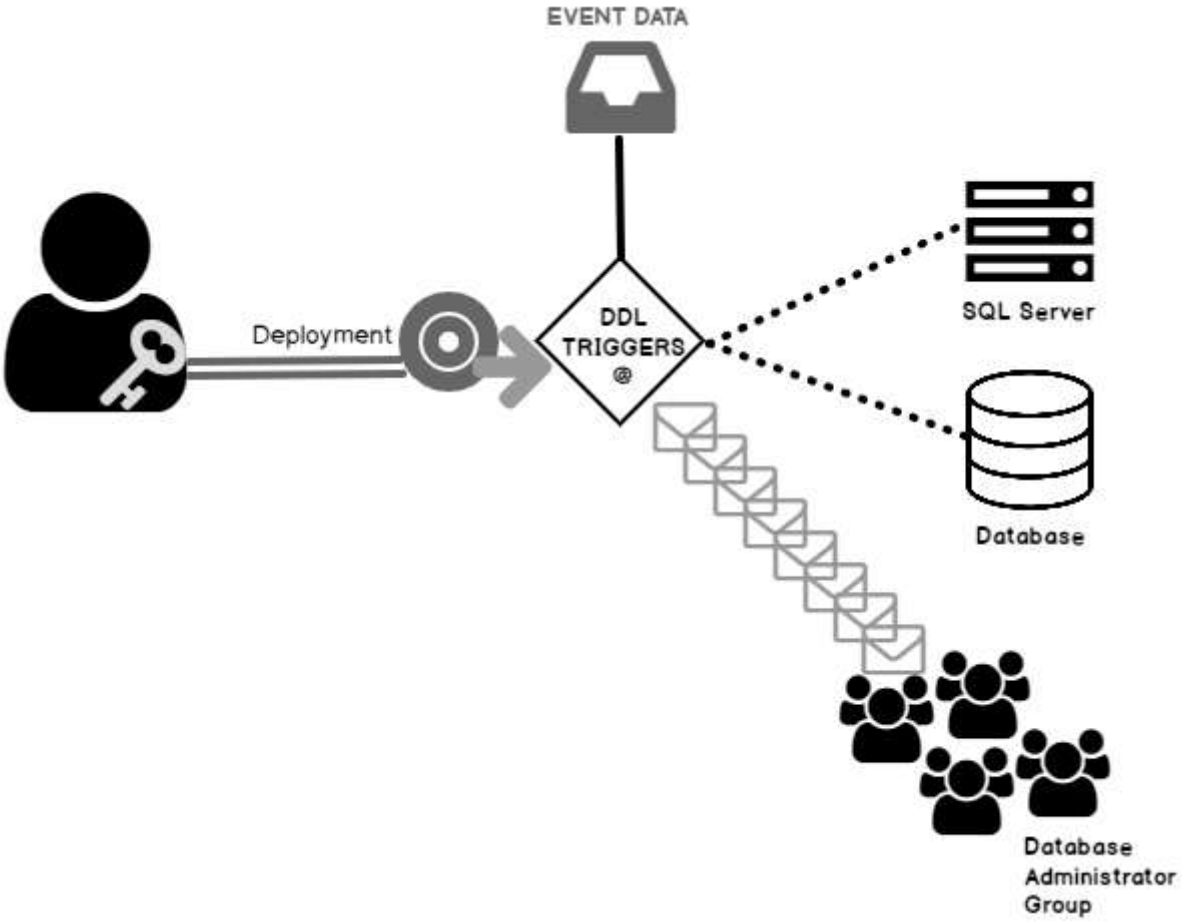
Summary Table

Concept	Explanation
Integrity Constraints	Rules to maintain data correctness
SQL Data Types	Define type of stored data
Schema	Logical design of database

Conclusion

17

Functions, Procedures, and Triggers in a Good Relational Database



BEFORE INSERT

```
INSERT INTO VEHICLE
VALUES('VH010','Honda','10
0','Kanpur');
END;
```

AFTER INSERT

```
INSERT INTO VEHICLE
VALUES(:new.vehicle_id,
:new.vehicle_name, :new.sale,
:new.city);
END;
```

AFTER DELETE

```
delete from vehicle where
vehicle_id = :old.vehicle_id;
END;
```

DROP A TRIGGER

```
DROP TRIGGER trigger_name;
Now below is an example for
the same
DROP TRIGGER
after_delete_employee;
```

Functions	Stored procedures
A function has a return type and returns a value.	A procedure does not have a return type. But it returns values using the OUT parameters.
You cannot use a function with Data Manipulation queries. Only Select queries are allowed in functions.	Using DML queries such as insert, update, select etc… with procedures is possible.
A function does not allow output parameters	A procedure allows both input and output parameters.
Managing transactions inside a function is not allowed.	Managing transactions inside a procedure is possible.
You cannot call stored procedures from a function	Calling a function from a stored procedure is possible
Calling a function using a select statement is possible.	Calling a procedure using select statements is not possible.

Introduction

A **good relational database** supports programmability features that help enforce business rules, improve performance, and maintain data integrity. The three key database objects used for this purpose are **Functions**, **Procedures**, and **Triggers**.

---

1. Function

Definition

A **function** is a stored database object that **accepts input parameters**, performs a computation, and **returns a single value**.

Key Characteristics

- Must return a value
- Can be used inside SQL statements (SELECT, WHERE)
- Cannot modify database data (in most DBMS)

Example

```
CREATE FUNCTION CalcBonus(salary INT)
RETURNS INT
RETURN salary * 0.10;
```

Usage

```
SELECT Name, CalcBonus(Salary) FROM Employee;
```

Purpose

- Reusable calculations
- Cleaner and readable queries

---

2. Procedure (Stored Procedure)

Definition

A **procedure** is a stored database program that **performs a set of operations**.
It **may or may not return a value**.

Key Characteristics

- Can accept input/output parameters
- Can perform INSERT, UPDATE, DELETE
- Called explicitly using CALL or EXEC

Example

```
CREATE PROCEDURE IncreaseSalary(IN empid INT)
BEGIN
  UPDATE Employee
  SET Salary = Salary + 5000
  WHERE Emp_ID = empid;
END;
```

Execution

```
CALL IncreaseSalary(101);
```

Purpose

- Encapsulates business logic
- Improves performance by reducing network traffic

---

3. Trigger

Definition

A **trigger** is a special database object that **automatically executes** in response to a database event.

Events

- INSERT
- UPDATE
- DELETE

Timing

- BEFORE
- AFTER

Example

```
CREATE TRIGGER Salary_Check
BEFORE UPDATE ON Employee
FOR EACH ROW
BEGIN
  IF NEW.Salary < 0 THEN
    SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = 'Salary cannot be negative';
  END IF;
END;
```

Purpose

- Enforces integrity rules automatically
- Maintains audit logs
- Prevents invalid data entry

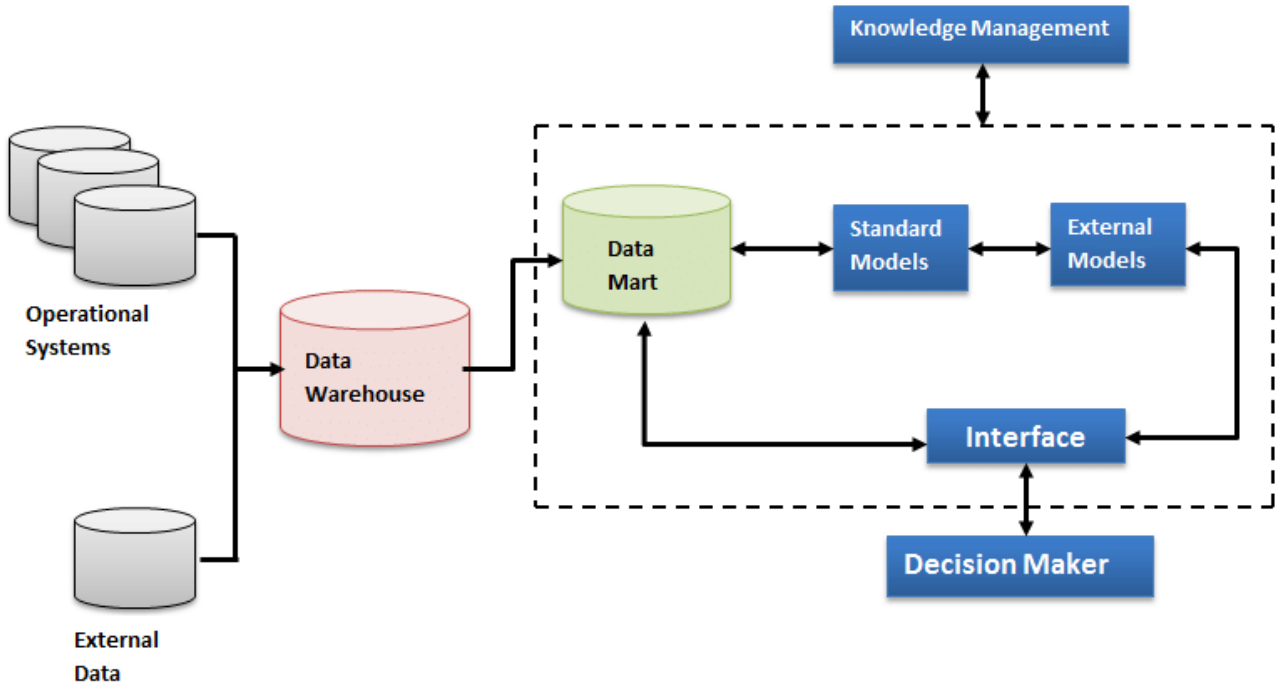
Difference between Function, Procedure, and Trigger

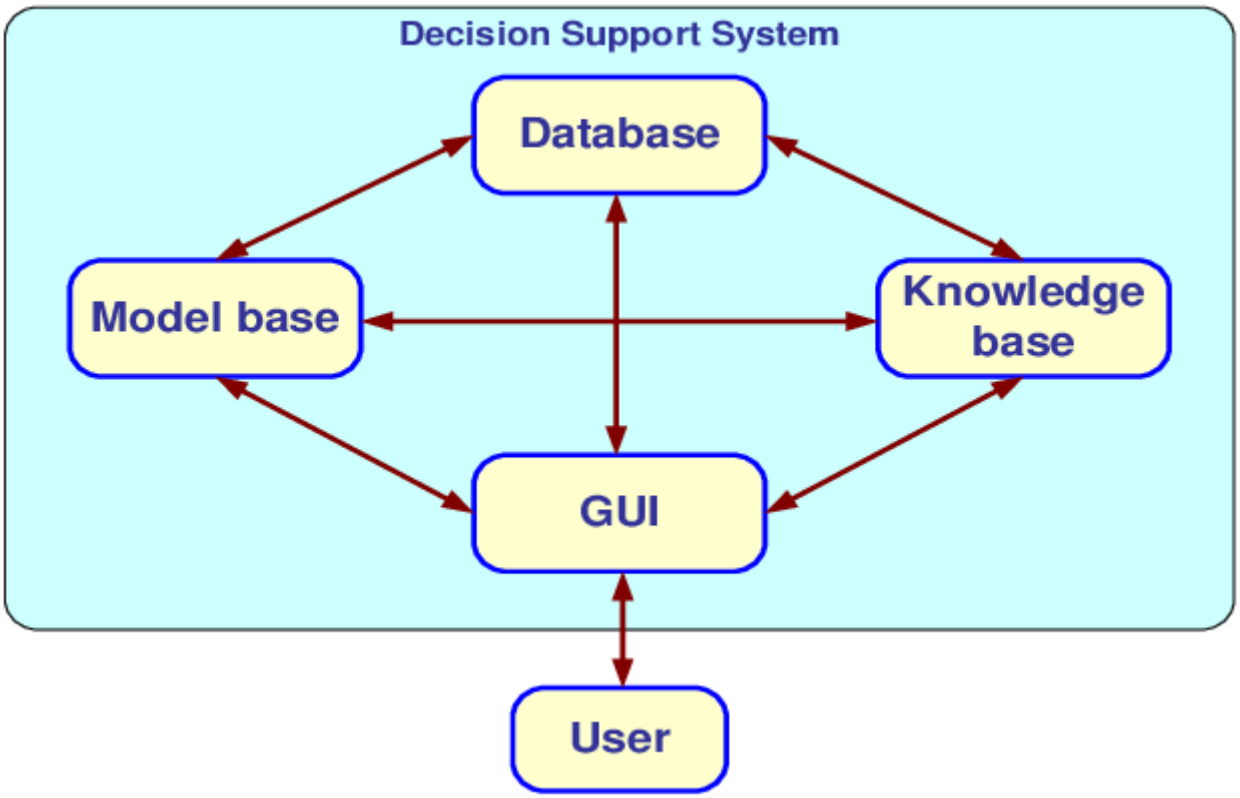
Aspect	Function	Procedure	Trigger
Returns Value	Yes	Optional	No
Called By	SQL query	CALL / EXEC	Automatically
Parameters	Input only	IN / OUT	No parameters
Data Modification	Usually No	Yes	Yes
Execution	Explicit	Explicit	Implicit
Use Case	Calculations	Business logic	Integrity enforcement

Role in a Good Relational Database

Feature	Contribution
Functions	Code reuse & query simplicity
Procedures	Performance & logic control
Triggers	Automatic integrity enforcement

Decision Support System (DSS)


Knowledge Management
Operational Systems
Data Warehouse
Data Mart
Standard Models
External Models
Interface
Decision Maker
External Data


Decision Support System
Database
Model base
Knowledge base
GUI
User

Definition

A **Decision Support System (DSS)** is a **computer-based information system** that supports **managerial decision-making** by combining **data**, **analytical models**, and **user-friendly interfaces**.
It helps decision makers analyze **semi-structured and unstructured problems** rather than routine transactions.

Objectives of DSS

- Support **strategic, tactical, and operational** decisions
- Improve **decision quality and speed**
- Enable **what-if analysis**, forecasting, and simulation
- Assist managers with **insight**, not replace them

Key Characteristics

- **Interactive** and user-driven
- **Flexible** and adaptable to changing scenarios
- Handles **large volumes of data**
- Focuses on **analysis**, not just reporting
- Supports **what-if, goal-seeking, sensitivity analysis**

Components of a DSS

1. Data Management Subsystem

- Stores and manages data from:
    - Databases
    - Data warehouses
    - External sources
- Uses DBMS for querying and retrieval

2. Model Management Subsystem

- Contains analytical models such as:
    - Statistical models
    - Optimization models
    - Forecasting and simulation models
- Enables scenario evaluation

3. User Interface (UI)

- Provides interaction between user and system
- Includes dashboards, reports, charts, and forms
- Ensures ease of use for non-technical users

4. Knowledge Base (optional)

- Stores rules, policies, and expert knowledge
- Enhances decision intelligence

Types of DSS

1. **Data-driven DSS**
    - Focuses on large databases and data analysis
    - Example: Sales trend analysis
2. **Model-driven DSS**
    - Emphasizes mathematical and analytical models
    - Example: Inventory optimization
3. **Knowledge-driven DSS**
    - Uses expert rules and AI techniques
    - Example: Medical diagnosis systems
4. **Document-driven DSS**
    - Manages unstructured documents
    - Example: Policy and legal document analysis
5. **Communication-driven DSS**

- Supports collaboration and group decisions
- Example: Video conferencing with shared analytics

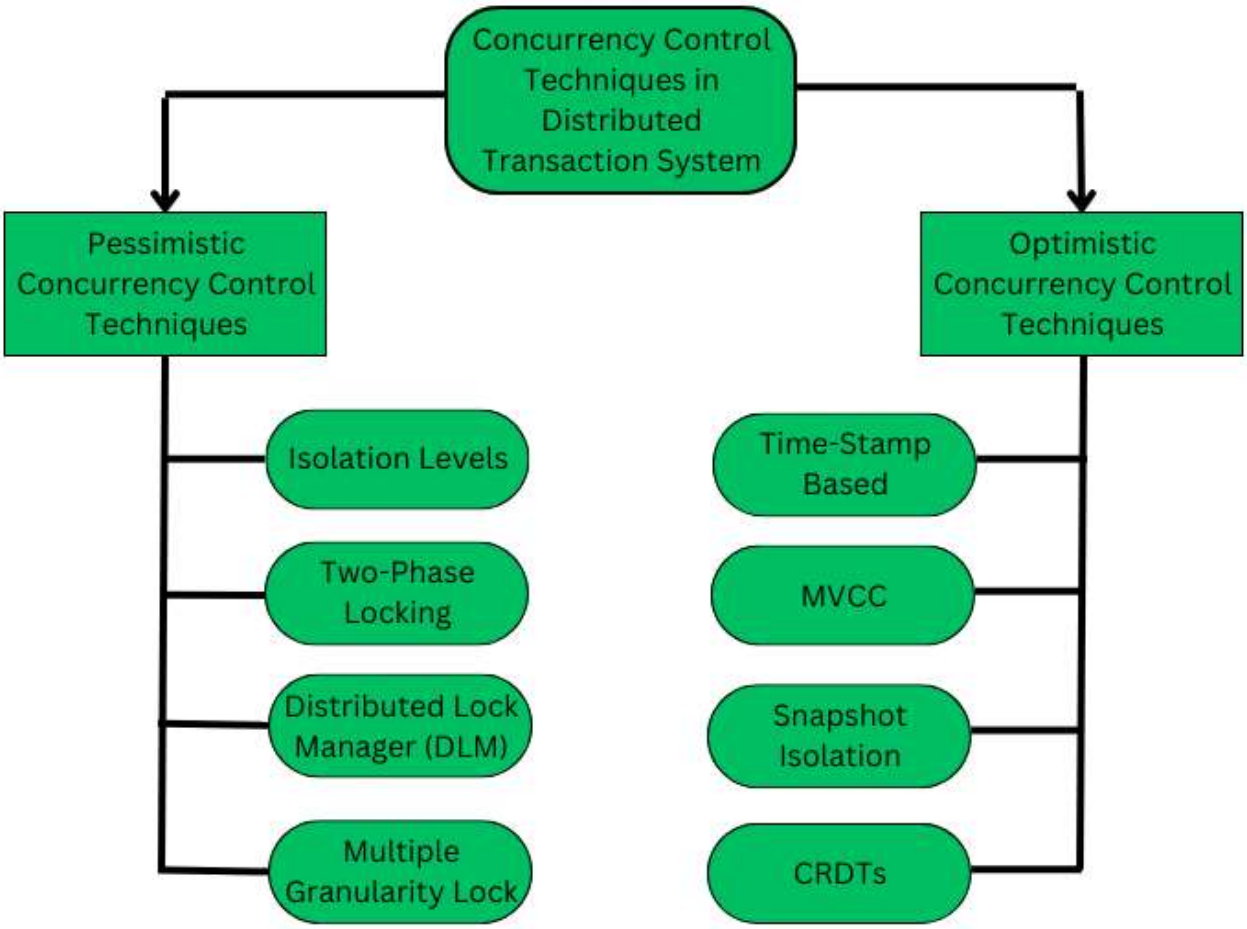
Examples of DSS Applications

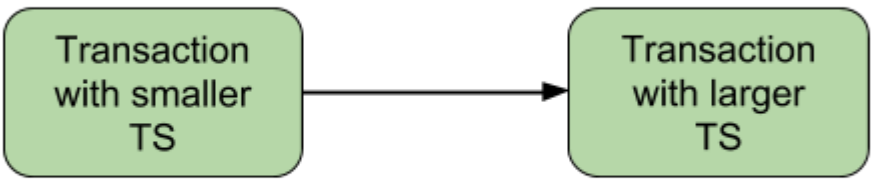
- **Business:** Profit forecasting, market analysis
- **Finance:** Portfolio management, risk assessment
- **Healthcare:** Treatment planning, diagnosis support
- **Manufacturing:** Production planning, capacity analysis
- **Government:** Policy analysis, resource allocation

Difference: DSS vs Transaction Processing System (TPS)

Aspect	DSS	TPS
Purpose	Decision support	Routine operations
Data	Historical & aggregated	Current & detailed
Users	Managers, analysts	Clerks, operators
Queries	Complex, analytical	Simple, repetitive

Concurrency Control Techniques (DBMS)





Definition

Concurrency control techniques are methods used by a **Database Management System (DBMS)** to ensure that **simultaneous execution of transactions** does not lead to

inconsistency in the database.
They guarantee **serializability** and preserve the **ACID properties**, especially **Isolation** and **Consistency**.

Why Concurrency Control is Needed

Without proper control, concurrent transactions may cause anomalies such as:

- **Dirty read**
- **Lost update**
- **Unrepeatable read**
- **Inconsistent retrieval**

Major Concurrency Control Techniques

1. Lock-Based Protocols

- Transactions must **acquire locks** before accessing data.
- **Shared lock (S)** → for read
- **Exclusive lock (X)** → for write

Two-Phase Locking (2PL):

- **Growing phase:** acquire locks
- **Shrinking phase:** release locks
- Guarantees **conflict serializability**

Pros: Strong consistency
Cons: Deadlocks possible

---

2. Timestamp-Based Protocols

- Each transaction gets a **timestamp**.
- Operations are ordered by timestamps.
- If a transaction violates order → **rollback**

Pros: No deadlocks
Cons: More rollbacks, possible starvation

---

3. Optimistic Concurrency Control (OCC)

- Assumes **conflicts are rare**
- Transactions execute without locks
- Checked only at **commit time**

Phases:

1. Read phase

2. Validation phase
3. Write phase

Pros: High concurrency
Cons: Costly rollbacks if conflicts occur

---

4. Multiversion Concurrency Control (MVCC)

- Maintains **multiple versions** of data items
- Readers access old versions; writers create new versions

Pros: High read performance, no blocking
Cons: More storage overhead

---

5. Validation-Based Protocol

- Similar to optimistic approach
- Validates transactions before commit to ensure serializability

---

6. Graph-Based Protocols

- Uses **precedence graph** to control execution order
- Ensures no cycles are formed

Comparison Summary

Technique	Locks Used	Deadlock	Rollback	Concurrency
Lock-based (2PL)	Yes	Possible	Low	Medium
Timestamp-based	No	No	High	Medium
Optimistic	No	No	Medium	High
MVCC	No (for reads)	No	Low	Very High

Conclusion

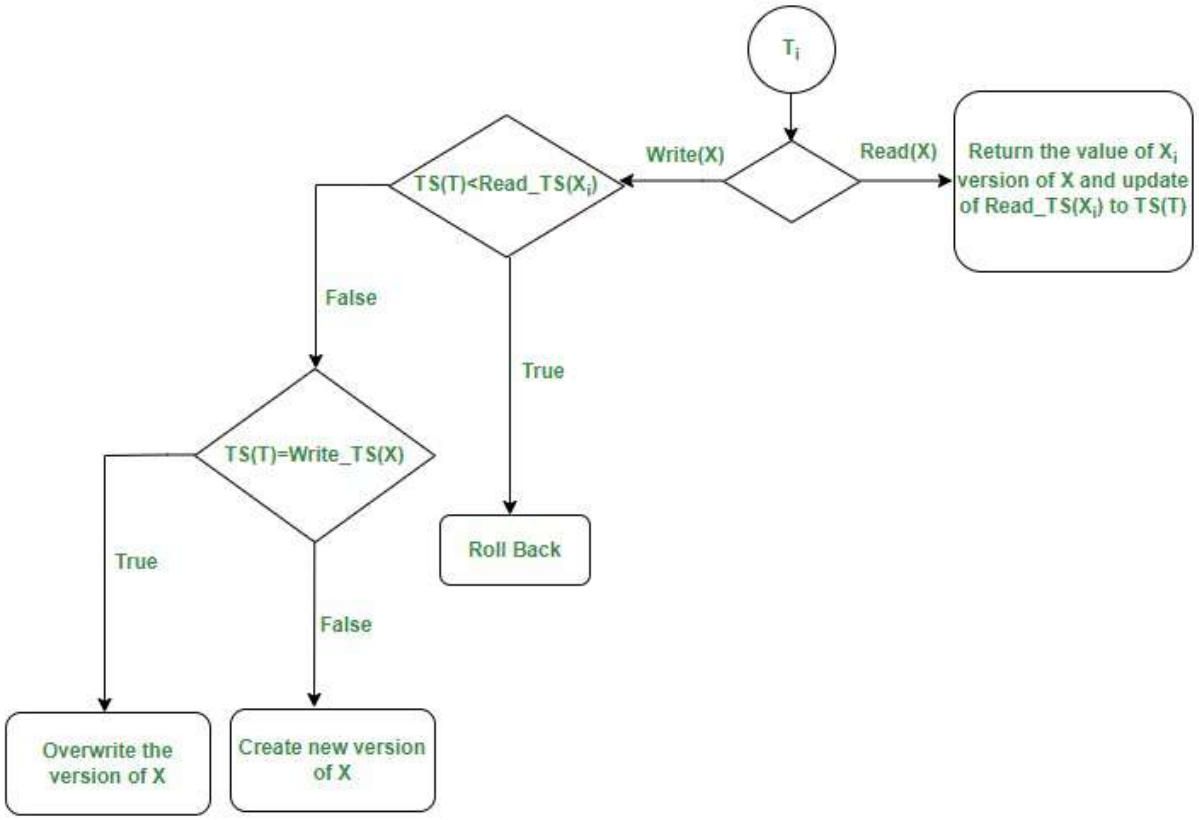
Concurrency control techniques ensure **safe and correct execution of transactions** in a multi-user environment. Different techniques are chosen based on system needs:

- **2PL** for strong consistency
- **Timestamp/OCC** for deadlock-free execution
- **MVCC** for high-performance systems

They are fundamental to reliable and efficient **database systems**.

20

Timestamp-Based Concurrency Control Protocol



T1 (Timestamp =100)	T2 (Timestamp =200)	T3 (Timestamp =300)
R(A)		R(B)
	R(A)	W(B)
R(B)	W(A)	R(A)
W(B)		

Definition

A **Timestamp-Based Protocol** is a **non-locking concurrency control technique** in DBMS that ensures **serializability** by executing transactions in the **order of their timestamps**.
Each transaction is assigned a unique timestamp when it starts, and all operations must respect this order.

Basic Idea

- Every transaction **T** gets a timestamp **TS(T)**.
- Older transaction → **smaller timestamp**
- Younger transaction → **larger timestamp**
- Conflicts are resolved by **rollback**, not by waiting (so **no deadlocks**).

Timestamps Maintained

For each data item **X**, the DBMS maintains:

- **RTS(X)** – *Read Timestamp*: largest TS of any transaction that successfully read X
- **WTS(X)** – *Write Timestamp*: largest TS of any transaction that successfully wrote X

Rules of the Timestamp Ordering Protocol

1) Read Rule

Transaction **T** wants to read **X**:

- If **TS(T) < WTS(X)** → ❌ **Reject read** (value is newer than T) → **Rollback T**
- Else → ✓ **Allow read** and set
  RTS(X) = max(RTS(X), TS(T))

---

2) Write Rule

Transaction **T** wants to write **X**:

- If **TS(T) < RTS(X)** → ❌ **Reject write** → **Rollback T**
- If **TS(T) < WTS(X)** → ❌ **Reject write** → **Rollback T**
- Else → ✓ **Allow write** and set
  WTS(X) = TS(T)

Illustrative Example

Assume:

- **T1** (older): TS(T1) = 10
- **T2** (younger): TS(T2) = 20
  Initially: RTS(X)=0, WTS(X)=0

Steps

1. **T1 reads X** → allowed → RTS(X)=10
2. **T2 writes X** → allowed → WTS(X)=20
3. **T1 writes X** → TS(T1)=10 < WTS(X)=20 → ❌ reject → **T1 rolled back**

✓ The protocol preserves the timestamp order (T1 before T2).

Variants

- **Basic Timestamp Ordering**: strict enforcement; may cause more rollbacks.
- **Thomas' Write Rule**: ignores obsolete writes to reduce unnecessary rollbacks.

Advantages

- **Deadlock-free** (no locks, no waiting)
- Guarantees **serializability**
- Suitable for **real-time systems**

Disadvantages

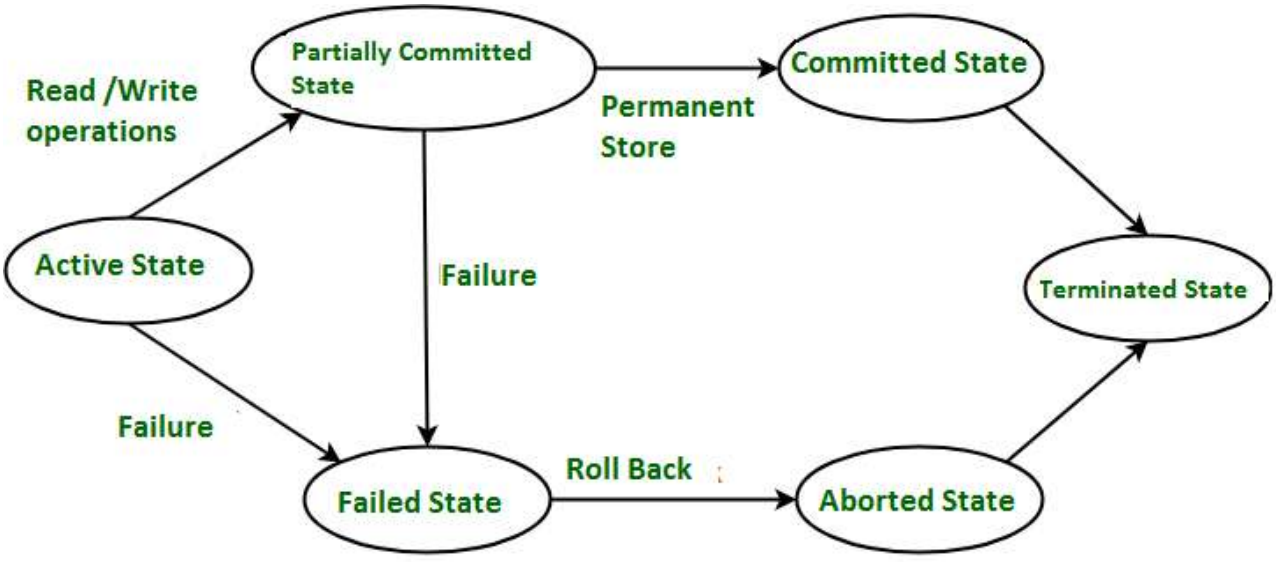
- **Higher rollback rate** under contention
- Possible **starvation** of older transactions
- Overhead of maintaining timestamps

Comparison (Quick)

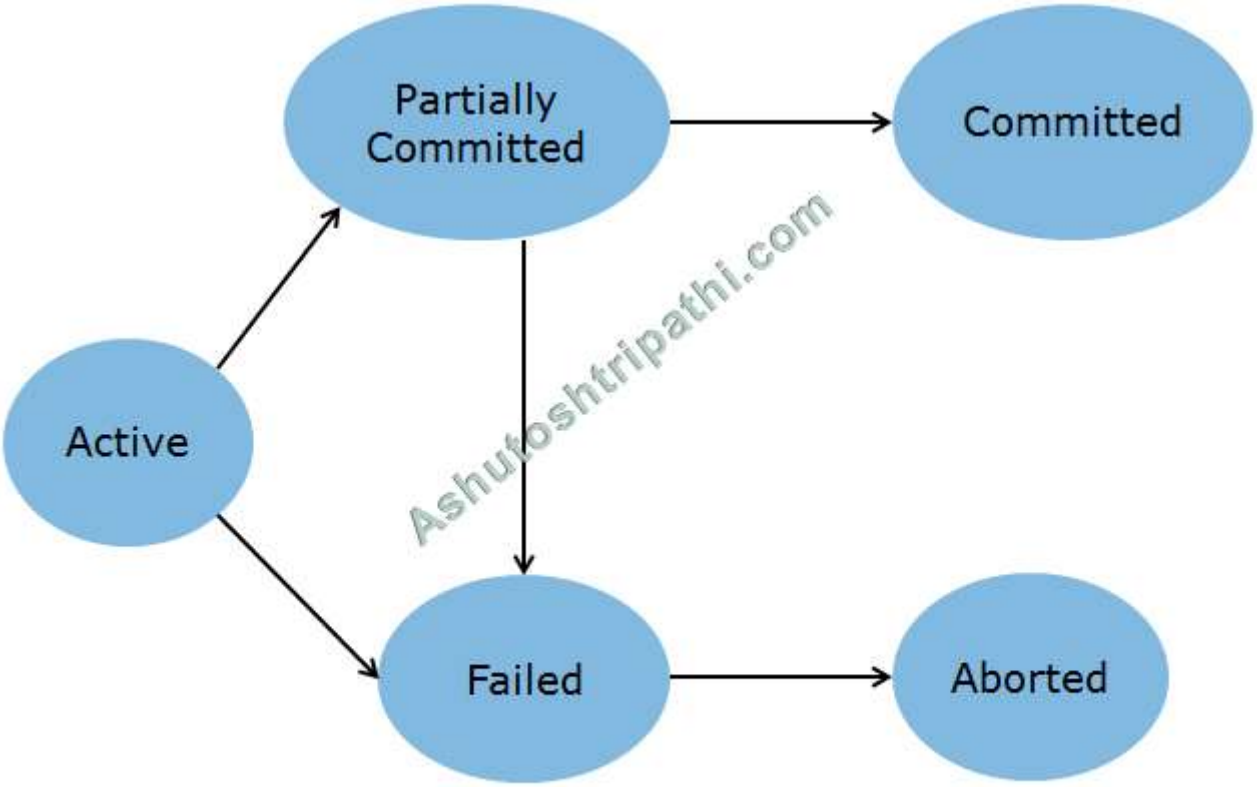
Aspect	**Timestamp-Based**
Locks	Not used
Deadlock	Impossible
Waiting	None
Conflict handling	Rollback
Serializability	Guaranteed

21

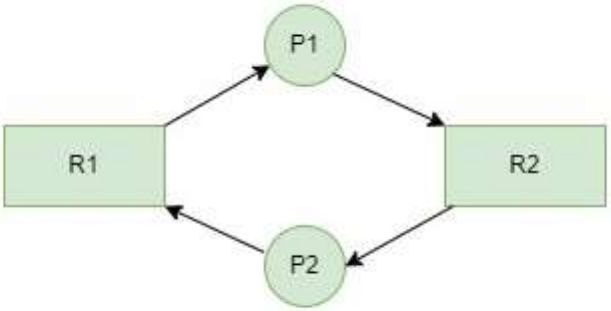
Simple Transaction Model & Deadlock Handling (DBMS)

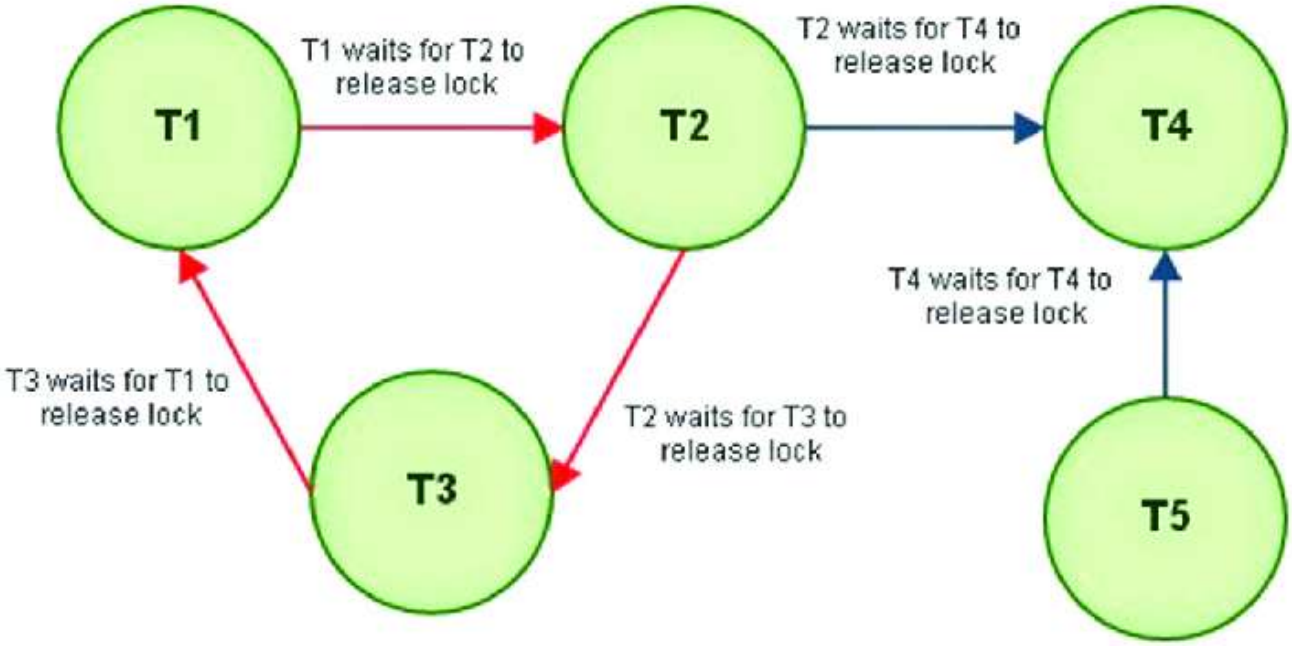


Transaction States in DBMS



Transaction States





---

1) Simple Transaction Model

What is a Transaction?

A **transaction** is a sequence of database operations (read/write) that performs a **single logical unit of work**. It must satisfy **ACID** properties.

Transaction States (Simple Model)

1. **Active**
    - Transaction is executing its operations.
2. **Partially Committed**
    - All statements executed; commit not yet finalized.
3. **Committed**
    - Changes are permanently saved.
4. **Failed**
    - An error occurs; transaction cannot proceed.
5. **Aborted (Rolled Back)**
    - Effects are undone; database restored to previous state.
6. **Terminated**
    - Transaction finishes after commit or abort.

State Flow (Typical):
Active → Partially Committed → Committed → Terminated
Active → Failed → Aborted → Terminated

Example

Bank Transfer (₹1000 from A to B):

- Read A, deduct 1000 → Read B, add 1000 → Commit
  If a crash happens before commit → **Abort & Rollback**.

---

2) Deadlock Handling

What is a Deadlock?

A **deadlock** occurs when two or more transactions **wait indefinitely** for resources held by each other.

Example:

- T1 holds lock on **X**, waits for **Y**
- T2 holds lock on **Y**, waits for **X**
  → Neither can proceed.

Deadlock Handling Techniques

A) Deadlock Prevention (Static Handling)

Prevent deadlocks **before they occur** by restricting how locks are requested.

Techniques:

- **Wait–Die**: Older waits; younger aborts.
- **Wound–Wait**: Older aborts younger; younger waits.

Pros: No deadlocks
Cons: More rollbacks, possible starvation

B) Deadlock Avoidance

Decide dynamically whether granting a lock may lead to deadlock.

- Requires knowing future resource needs.
- **Banker's Algorithm** (theoretical in DBMS).

Pros: Fewer rollbacks than prevention
Cons: High overhead, impractical often

C) Deadlock Detection (Dynamic Handling)

Allow deadlocks, then **detect and resolve** them.

Method:

- Build a **Wait-For Graph (WFG)**
- **Cycle ⇒ Deadlock**

Resolution:

- Select a **victim transaction**

- Roll it back to break the cycle

Pros: Better concurrency
Cons: Detection overhead

D) Deadlock Recovery

After detection:

- **Rollback** victim (partial or full)
- **Restart** transaction later

Quick Comparison

Technique	When Applied	Deadlock	Overhead
Prevention	Before execution	Impossible	Low–Medium
Avoidance	During execution	Avoided	High
Detection	After occurrence	Possible	Medium
Recovery	After detection	Resolved	Medium

Q. 3	**Solve Any Two of the following.**		
A)	What is decomposition? Illustrate lossy and lossless decomposition.	**Apply/CO 3**	
B)	Define Boyce-Codd normal form and third normal form. How does BCNF differ from 3NF?	**Understanding/CO3**	
C)	Illustrate multivalued dependency and explain how fourth normal form is achieved.	**Apply/CO 3**	
Q.4	**Solve Any Two of the following.**		
A)	Discuss Parallel Systems Vs. Distributed Systems.	**Understanding/CO4**	
B)	What is data ware house? List and explain the properties of data ware house.	**Understanding/CO4**	
C)	Write short Note on: JDBC, ODBC and Embedded SQL.	**Understanding/CO4**	
Q. 5	**Solve Any Two of the following.**		
A)	What is transaction? Illustrate ACID Properties of a transaction for a banking application.	**Apply/CO 5**	
B)	What do you mean by Lock, shared Lock& Exclusive lock? Illustrate Two-phase locking protocol.	**Apply/CO 5**	
C)	Explain concept of view serializability, dirty read anomaly, lost update anomaly and conflict equivalent.	**Understanding/CO5**	
	***** End *****		

The grid and the borders of the table will be hidden before final printing.