

# Video chapters

- **Chapter-1 (Basics):** Data & information, Database System vs File System, Views of Data Base, Data Independence, Instances & Schema, OLAP Vs OLTP, Types of Data Base, DBA, Architecture.
- **Chapter-2 (ER Diagram):** Entity, Attributes, Relationship, Degree of a Relationship, Mapping, Weak Entity set, Conversion from ER Diagram to Relational Model, Generalization, Specification, Aggregation.
- **Chapter-3 (RDBMS & Functional Dependency):** Basics & Properties, Update Anomalies, Purpose of Normalization, Functional Dependency, Closure Set of Attributes, Armstrong's axioms, Equivalence of two FD, Canonical cover, Keys.
- **Chapter-4 (Normalization):** 1NF, 2NF, 3NF, BCNF, Multivalued Dependency, 4NF, Lossy-Lossless Decomposition, 5NF, Dependency Preserving Decomposition.
- **Chapter-5 (Indexing):** Overview of indexing, Primary indexing, Clustered indexing and Secondary Indexing, B-Tree.
- **Chapter-6 (Relational Algebra) :** Query Language, Select, Project, Union, Set Difference, Cross Product, Rename Operator, Additional or Derived Operators.
- **Chapter-7(SQL) :** Introduction to SQL, Classification, DDL Commands, Select, Where, Set Operations, Cartesian Product, Natural Join, Outer Join, Rename, Aggregate Functions, Ordering, String, Group, having, Trigger, embedded, dynamic SQL.
- **Chapter-8 (Relational Calculus) :** Overview, Tuple Relation Calculus, Domain Relation Calculus.
- **Chapter-9 (Transaction) :** What is Transaction, ACID Properties, Transaction Sates, Schedule, Conflict Serializability, View Serializability, Recoverability, Cascade lessness, Strict Schedule.
- **Chapter-10 (Recovery & Concurrency Control) :** Log Based Recovery, Shadow Paging, Data Fragmentation, TIME STAMP ORDERING PROTOCOL, THOMAS WRITE RULE, 2 phase locking, Basic 2pl, Conservative 2pl, Rigorous 2pl, Strict 2pl, Validation based protocol Multiple Granularity.

# Syllabus

- **UNIT-1 : INTRODUCTION** Overview, Database System vs File System, Database System Concept and Architecture, Data Model Schema and Instances, Data Independence and Database Language and Interfaces, Data Definitions Language, DML, Overall Database Structure. **Data Modeling Using the Entity Relationship Model:** ER Model Concepts, Notation for ER Diagram, Mapping Constraints, Keys, Concepts of Super Key, Candidate Key, Primary Key, Generalization, Aggregation, Reduction of an ER Diagrams to Tables, Extended ER Model, Relationship of Higher Degree.
- **UNIT-2 : RELATIONAL DATA MODEL** Relational Data Model Concepts, Integrity Constraints, Entity Integrity, Referential Integrity, Keys Constraints, Domain Constraints, Relational Algebra, Relational Calculus, Tuple and Domain Calculus. **Introduction on SQL:** Characteristics of SQL, Advantage of SQL. SQL Data Type and Literals. Types of SQL Commands. SQL Operators and Their Procedure. Tables, Views and Indexes. Queries and Sub Queries. Aggregate Functions. Insert, Update and Delete Operations, Joins, Unions, Intersection, Minus, Cursors, Triggers, Procedures in SQL/PL SQL.
- **UNIT-3 : DATA BASE DESIGN & NORMALIZATION** Functional dependencies, normal forms, first, second, 3 third normal forms, BCNF, inclusion dependence, loss less join decompositions, normalization using FD, MVD, and JDs, alternative approaches to database design.
- **UNIT-4 : TRANSACTION PROCESSING CONCEPT** Transaction System, Testing of Serializability, Serializability of Schedules, Conflict & View Serializable Schedule, Recoverability, Recovery from Transaction Failures, Log Based Recovery, Checkpoints, Deadlock Handling. Distributed Database: Distributed Data Storage, Concurrency Control, Directory System.
- **UNIT-5 : CONCURRENCY CONTROL TECHNIQUES** Concurrency Control, Locking Techniques for Concurrency Control, Time Stamping Protocols for Concurrency Control, Validation Based Protocol, Multiple Granularity, Multi Version Schemes, Recovery with Concurrent Transaction, Case Study of Oracle.

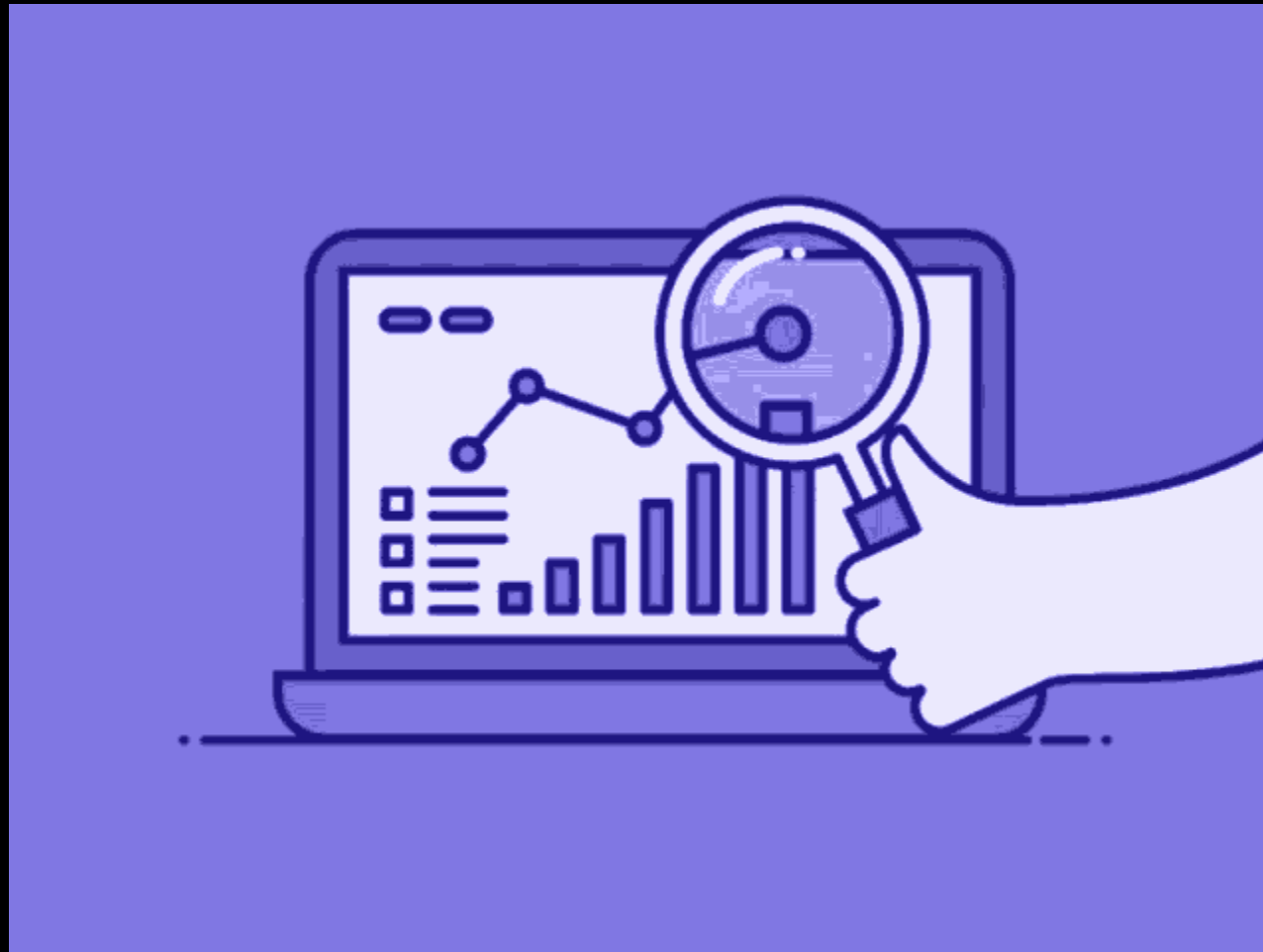
# What is Data ?

- Data are characteristics or attributes, often numerical, collected through observation and can be qualitative(descriptive) or quantitative(numerical). Any facts and figures about an entity is called as Data.
- Data serves a crucial role in various sectors by facilitating analysis, supporting decision-making processes, and being fundamental to research activities.



# What is Information ?

- Data becomes information when analyzed and placed in context, providing a basis for understanding, decision-making, and further analysis. Processed Data is called information.



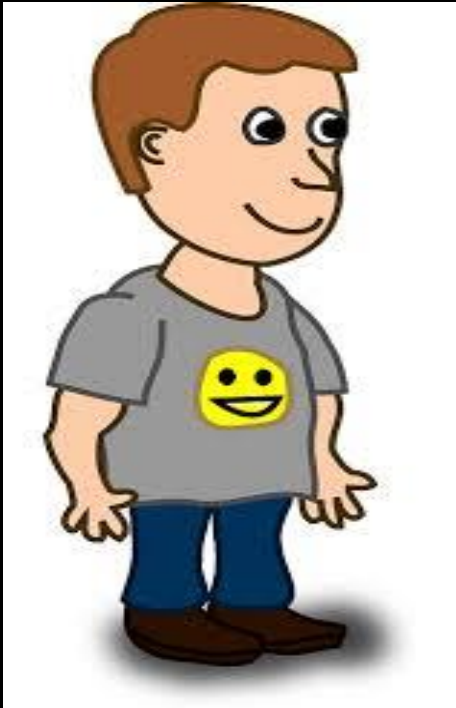
# What is Data Base?

- A database is a structured collection of data, facilitating easy access, management, and updates., generally stored and accessed electronically from a computer system.

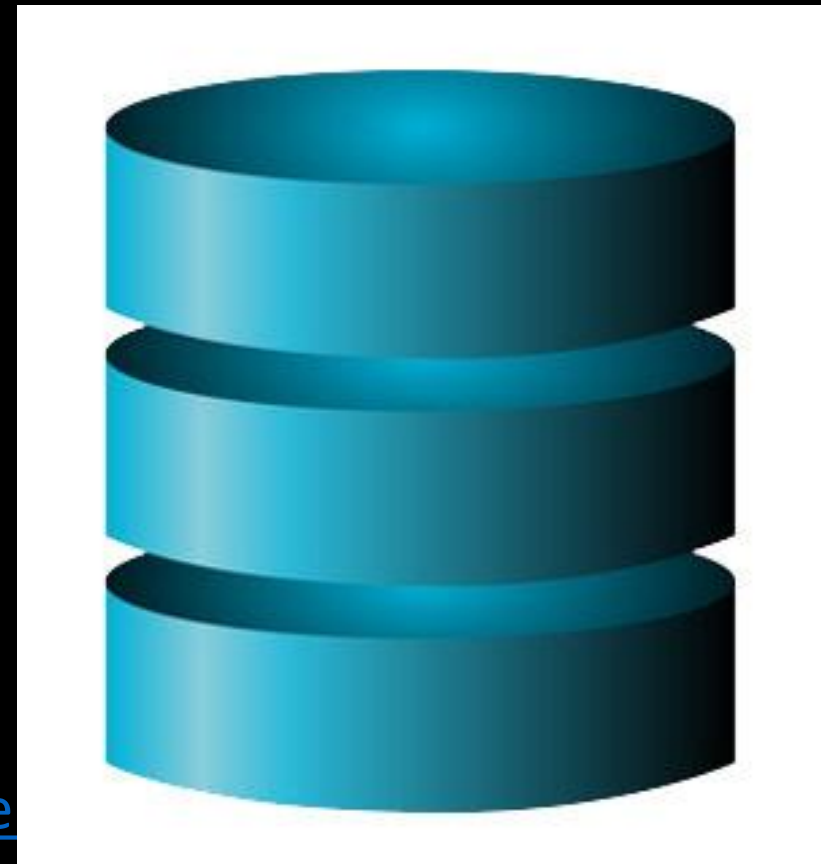


# What is Data Base Management System?

- A DBMS is software facilitating efficient data storage, retrieval, and management in databases.
- Ensures data safety and integrity, while offering accessibility and concurrency control.
- Supports functions like data querying, reporting, and analytics for informed decision-making.



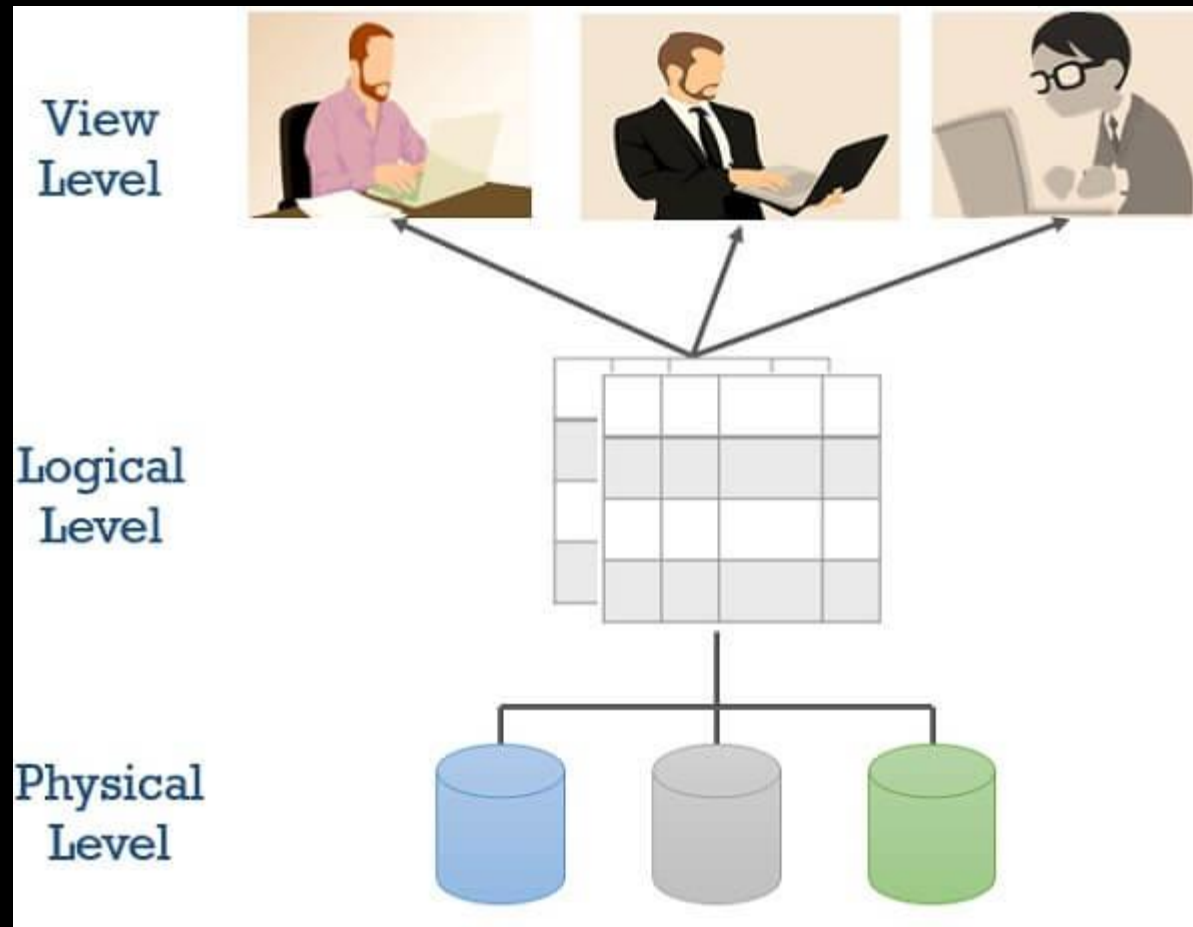
[Knowledge Gate](#)



Aspect	File System	Database Management System
Data Access	Slower data retrieval due to unstructured querying capabilities.	Structured querying capabilities allow for quicker data access.
Data Isolation	Challenges in correlating data across separate files leading to data isolation.	Facilitates data integration, reducing data isolation issues.
Data Integrity	Risk of inadvertent data alterations or deletions creating integrity problems.	Features to prevent unauthorized data alterations, maintaining integrity.
Atomicity Problem	Potential for data inconsistency due to incomplete operations, leading to atomicity problems.	Supports transaction properties like atomicity, ensuring operations are either completed fully or not at all.
Concurrent Access Anomalies	Conflicts and inconsistencies from simultaneous data access/modifications, causing concurrent access anomalies. <a href="https://www.knowledgegate.com">Knowledge Gate Website</a>	Advanced concurrency controls to manage multiple users accessing the database simultaneously, reducing anomalies.

# View of Data Base (Data Abstraction)

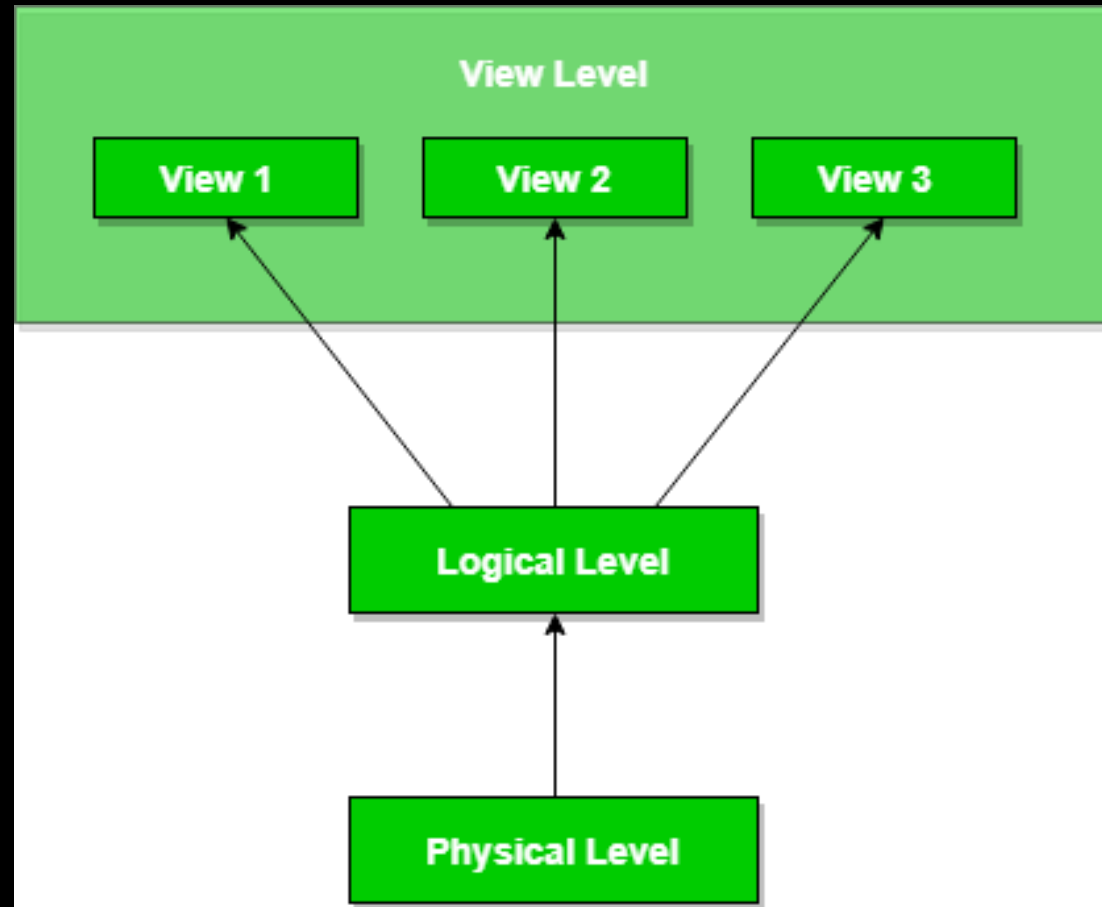
- Physical Level
- Logical Level/ Conceptual Level
- View Level





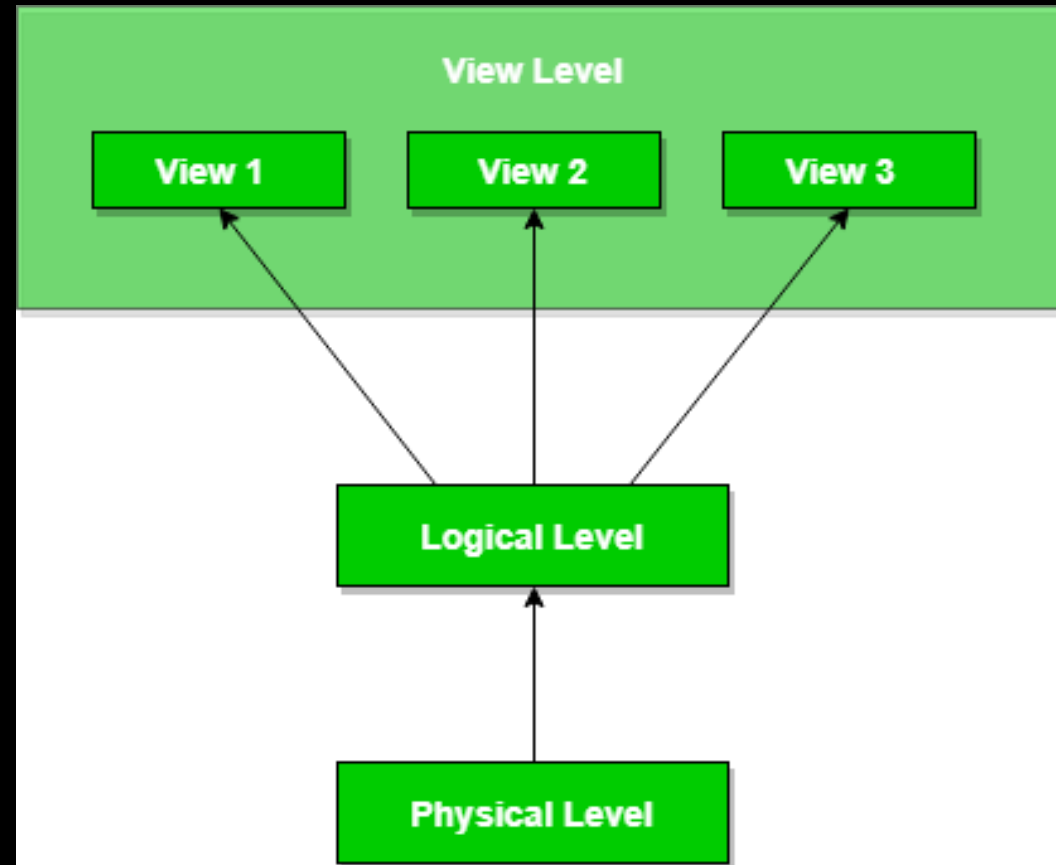
# View of Data Base (Data Abstraction)

- **Physical Level**: The internal schema details data storage and access on hardware, featuring the lowest level of data abstraction with complex structures, predominantly managed by the database administrator.



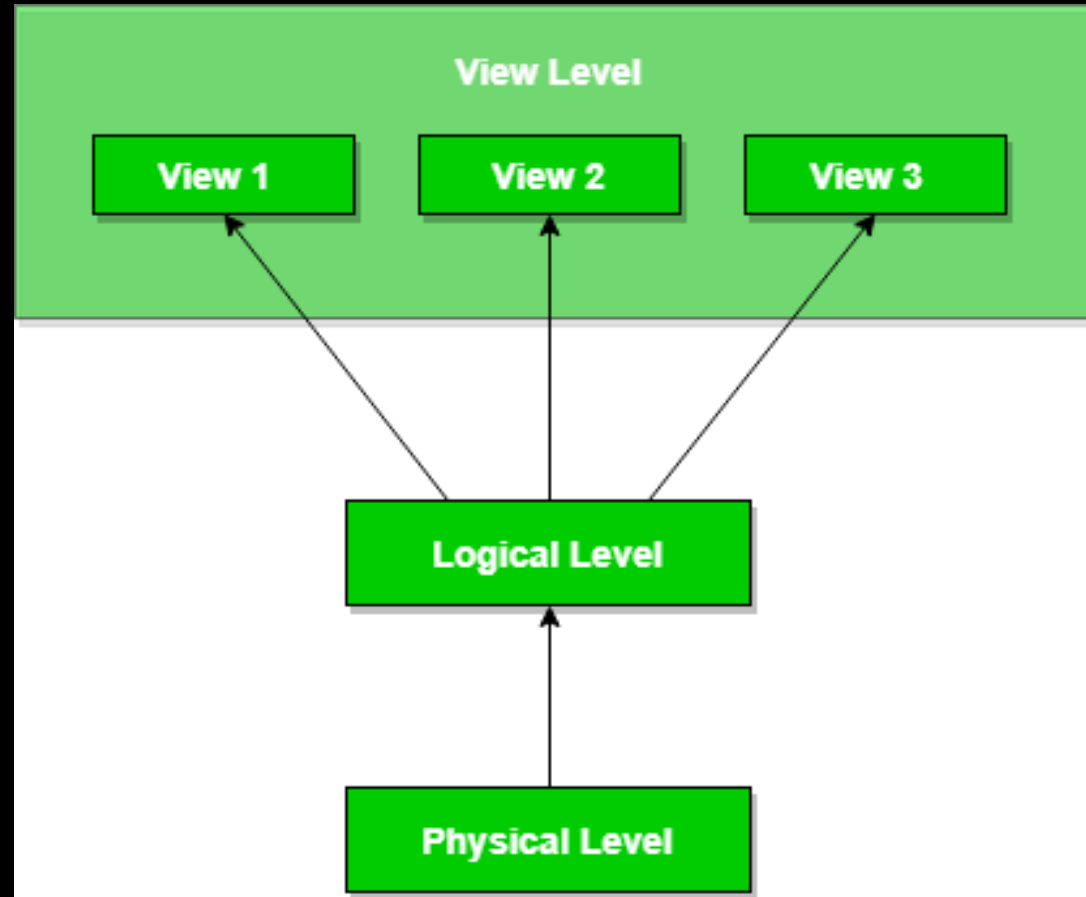
## View of Data Base

- Logical Level/ Conceptual Level: Above the physical level, this level showcases data as entity sets and their relationships, detailing the types and connections between stored data in the database.



## View of Data Base

- **View Level**: This is the pinnacle of data abstraction, displaying only a portion of the entire database focusing on user-interest areas. It can represent multiple views of the same data, allowing users to access information through various applications from the database.



# Data independence

- Data independence is defined as the capacity to change the schema at one level of a database system without having to change the schema at the next higher level.
- Types of data independence :
  - **Physical data independence**: a. Physical data independence is the ability to modify internal schema without changing the conceptual schema. b. Modification at the physical level is occasionally necessary in order to improve performance. c. It refers to the immunity of the conceptual schema to change in the internal schema. d. Examples of physical data independence are reorganizations of files, adding a new access path or modifying indexes, etc.
  - **Logical data independence**: a. Logical data independence is the ability to modify the conceptual schema without having to change the external schemas or application programs. b. It refers to the immunity of the external model to changes in the conceptual model. c. Examples of logical data independence are addition/removal of entities.

# Instance and Schemas

- **Instance of the Database:**
  - The collection of information stored in the database at a specific moment is known as an instance of the database. It is a snapshot of the database that contains live data at that moment, showing the current state of all records and transactions.
- **Database Schema:**
  - The database schema refers to the overall design of the database, illustrating the logical structure and organization of data. It defines how data is organized and how relationships between data are handled, essentially serving as the blueprint for how the database is constructed.

Aspect	OLAP (Online Analytical Processing)	OLTP (Online Transaction Processing)
<b>Primary Function</b>	Designed for complex data analysis and reporting.	Handles daily transactional data processing.
<b>Database Design</b>	Star or snowflake schema, optimizing for read operations.	Normalized schema, optimizing for write operations.
<b>Query Complexity</b>	Complex queries involving aggregations and computations across multiple dimensions.	Simple and standard queries focusing on CRUD operations (Create, Read, Update, Delete).
<b>Data Volume</b>	Deals with large volumes of data for historical analysis.	Processes a high number of small transactions.
<b>Response Time</b>	Slower response time due to complex queries.	Fast response time to support high transaction rates.

## Types of Data Base

- **Commercial Database**: Predominantly used in the business sector to handle large volumes of transactions and customer data. A CRM system like Salesforce which handles large volumes of customer data and transactions.
- **Multimedia Database**: Stores data types such as images, audio, and video files, facilitating the management and retrieval of multimedia content. A digital asset management system like Adobe Experience Manager that facilitates the storage and retrieval of multimedia content.
- **Deductive Database**: Utilizes logic programming to derive information from data stored in a database, allowing for more complex and analytical queries. A database using Datalog (a query language) which allows for complex logical queries and information derivation.

- **Temporal Database**: Keeps track of changing data over time, allowing for queries concerning time-based data. A historical trading database in the financial sector which keeps track of stock prices over time.
- **Geological Information System (GIS)**: Stores, organizes, and analyzes geographical data, aiding in spatial analysis and mapping projects. A system like ArcGIS which enables the storage, analysis, and visualization of geographical data.



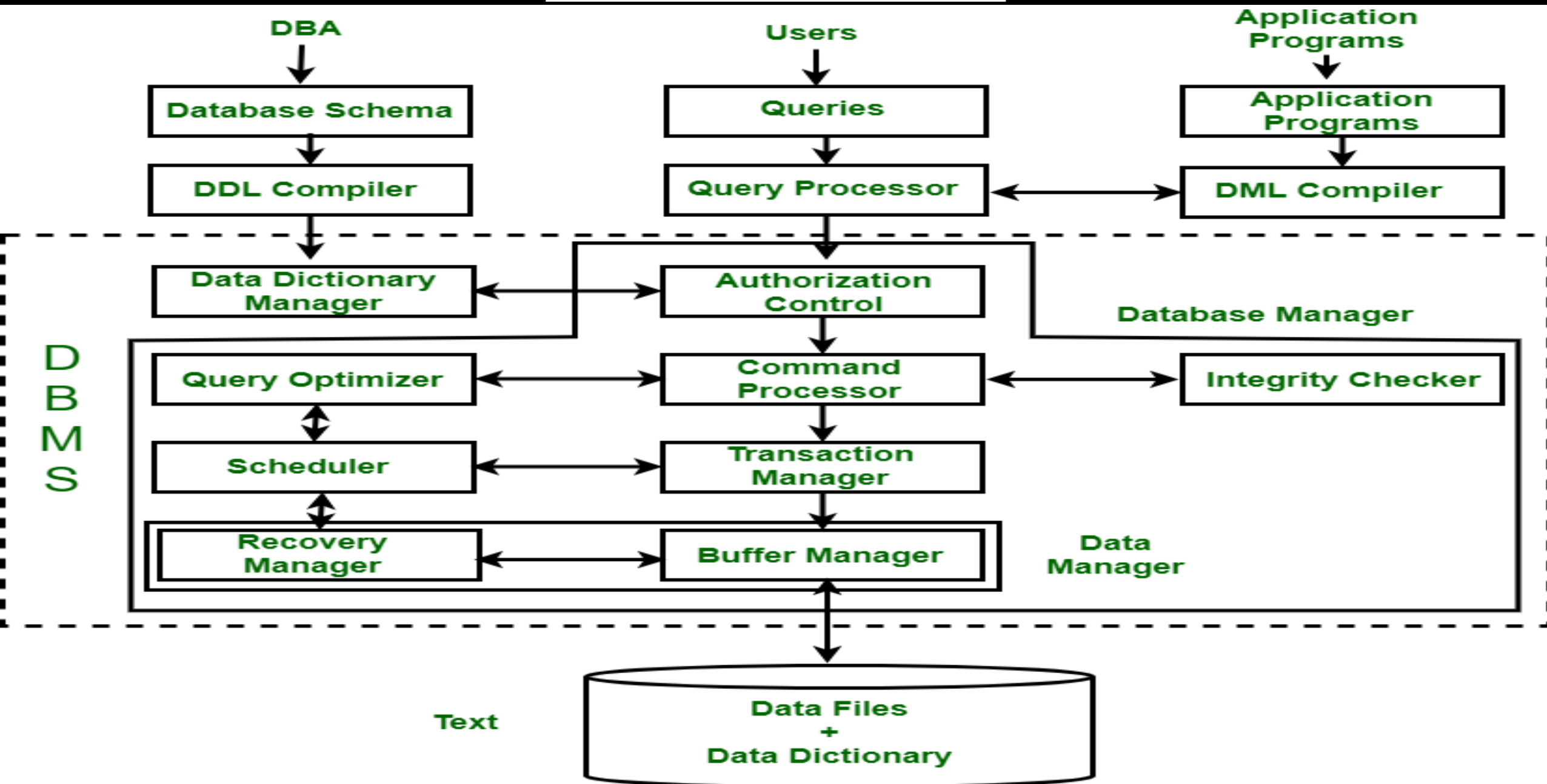
# DBA(Database Administrator)

Database administrators hold authority over data and the programs facilitating data access. Their roles/functions are:

- **Schema Definition**: DBA outlines the original database schema. Achieved through writing definitions translated to permanent labels in the data dictionary by the DDL compiler.
- **Storage Structure and Access Method Definition**: Responsible for forming appropriate storage structures and access methods. Achieved through writing definitions translated by the data storage and definition language compiler.
- **Schema and Physical Organization Modification**: Involves altering the database schema or physical storage organization. Changes are implemented by writing definitions that modify the relevant internal system tables.
- **Granting of Authorization for Data Access**: DBA grants varied types of data access authorization to different database users.
- **Integrity Constraint Specification**: DBA implements and maintains integrity constraints to ensure data accuracy and consistency.



# DBMS Architecture



**1. Query Processor:** This is the component of a DBMS that interprets and executes user queries. It comprises several sub-components including:

1. **DML Compiler:** Processes Data Manipulation Language (DML) statements into low-level instructions that can be executed.
2. **DDL Interpreter:** Processes Data Definition Language (DDL) statements into metadata tables.
3. **Embedded DML Pre-compiler:** Translates DML statements embedded in application programs into procedural calls.
4. **Query Optimizer:** Determines the most efficient way to execute a query by evaluating different query plans.

**2. Storage Manager:** Also known as the Database Control System, it is responsible for managing the data stored in the database, ensuring its consistency and integrity. It includes the following sub-components:

1. **Authorization Manager:** Manages access controls and privileges.
2. **Integrity Manager:** Ensures that data modifications adhere to integrity constraints.
3. **Transaction Manager:** Manages concurrent access to the database and maintains database consistency during transactions.
4. **File Manager:** Manages file space and data structures representing information in the database.
5. **Buffer Manager:** Manages data cache and data transfer between main memory and secondary storage.

**3. Disk Storage:** Represents the storage aspect of a DBMS, encompassing the following components:

1. **Data Files:** Files where the actual data is stored.
2. **Data Dictionary:** Repository containing information about the structure and characteristics of database objects.
3. **Indices:** Data structures that facilitate faster data retrieval.

A Database Management System (DBMS) consists of three primary components:

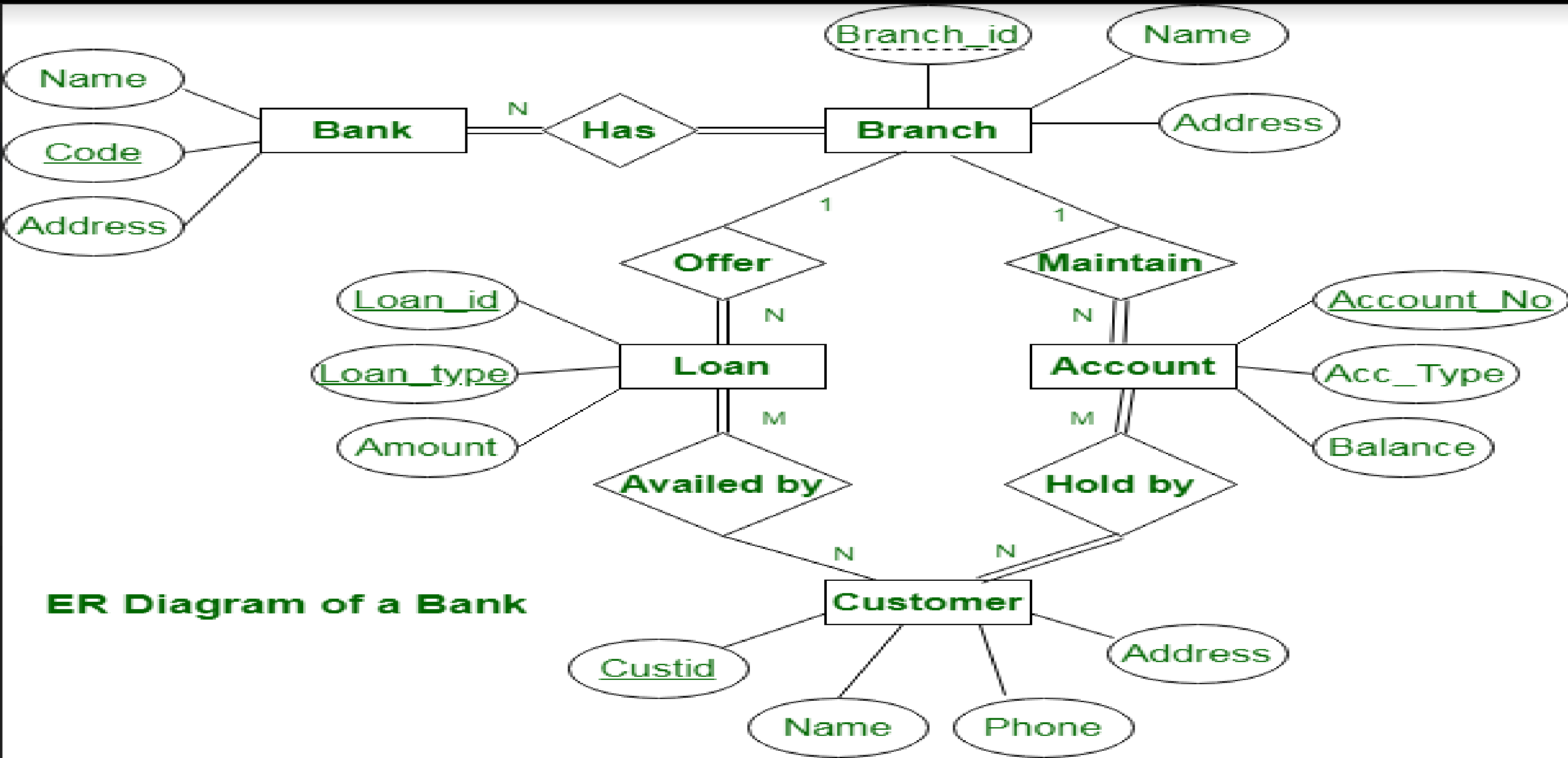
1. **Internal Level**: Concerns the physical storage of data in databases, overseeing data storage on hardware devices, and managing low-level aspects like data compression and indexing.
2. **Conceptual Level**: Represents the logical layout of the database, detailing the schema with tables and attributes and their interrelations. It's independent of specific DBMS implementations, focusing on organizing and connecting data elements.
3. **External Level**: Embodies the user interface of the database, facilitating data access and interaction through user-friendly views and interfaces tailored to various user groups.

# ER Diagram

- Developed by Dr. Peter Chen in 1976, this conceptual level method, grounded in real-world perceptions, facilitates diagrammatic data representation, simplifying comprehension for non-technical users.
- The E-R data model, central to database design, encapsulates entities and their attributes within an enterprise schema, serving as a clear, standardized tool for translating real-world enterprise interactions into a conceptual schema.

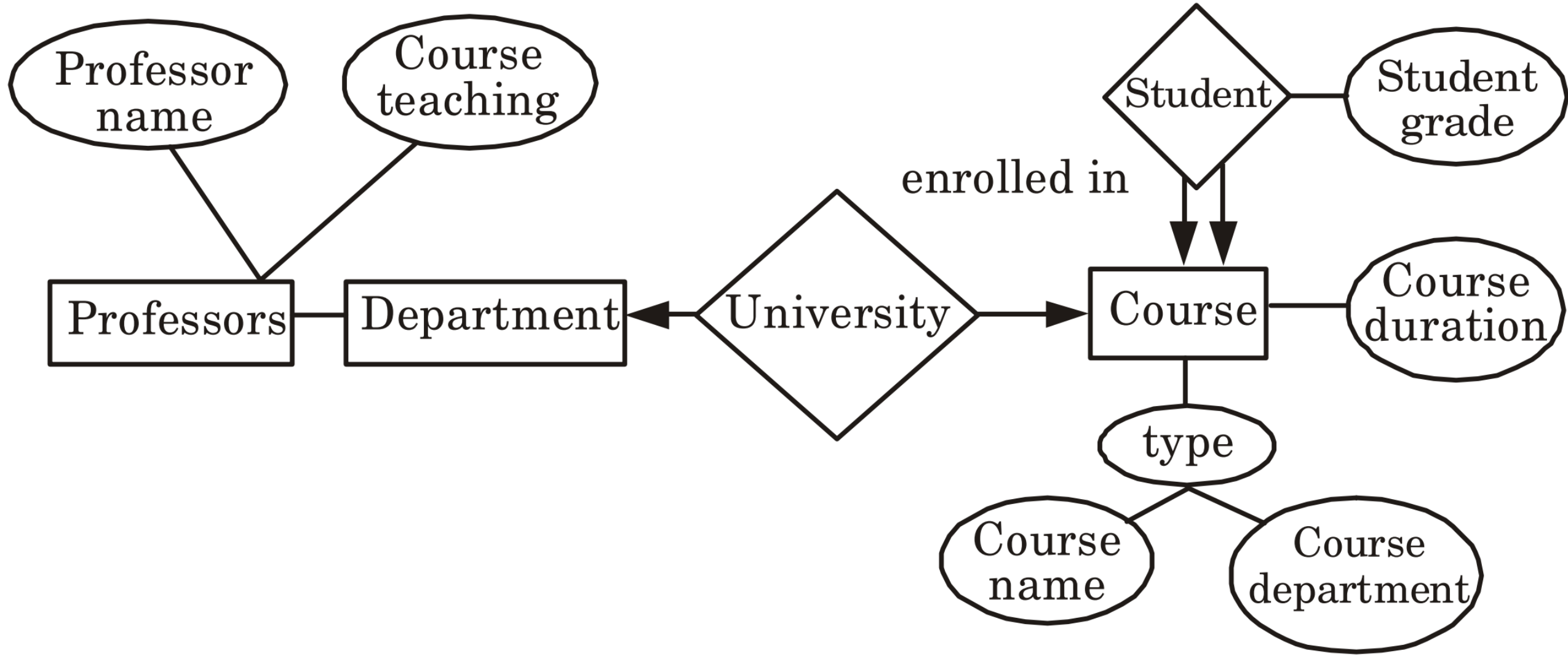


# ER diagram for bank

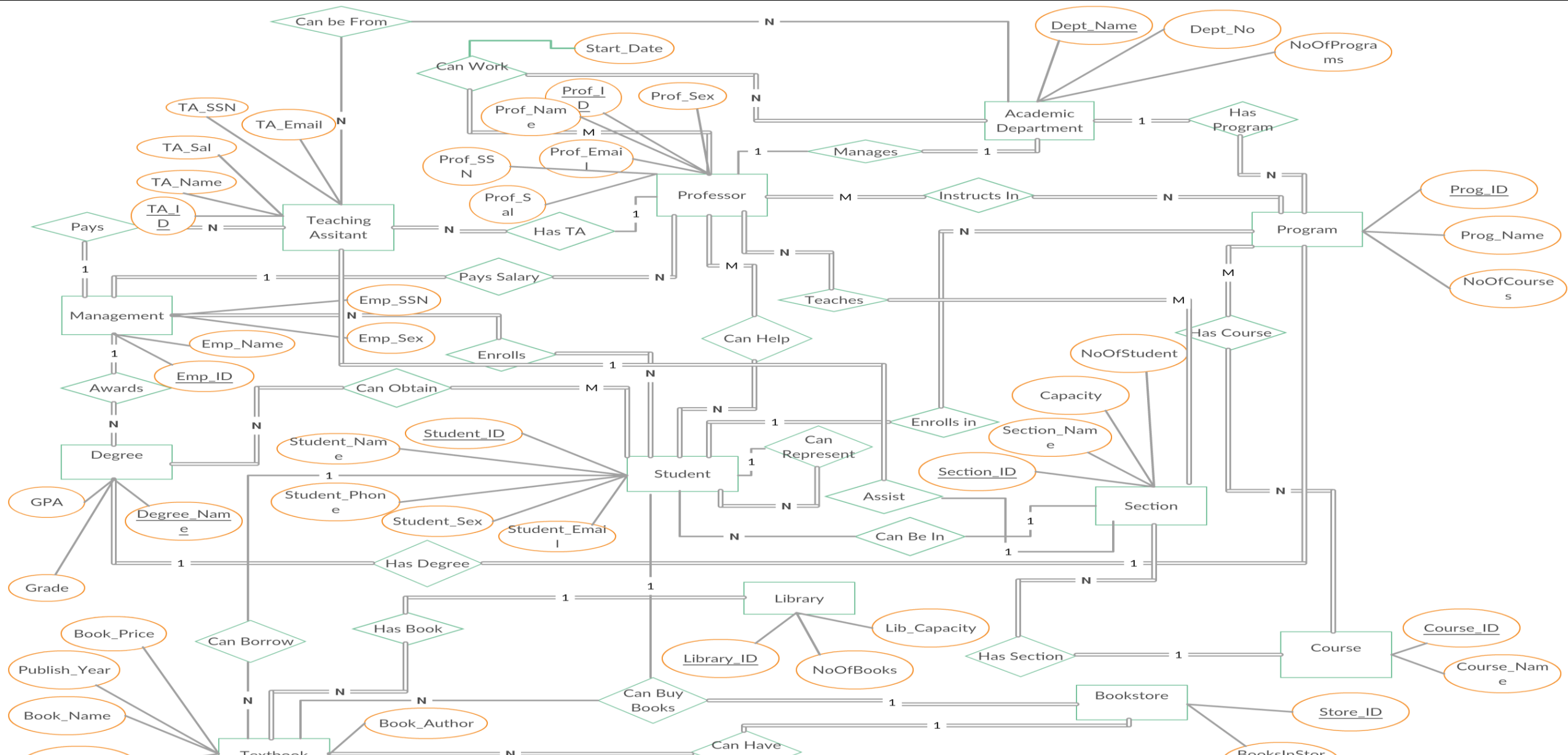


**ER Diagram of a Bank**

# ER diagram for University System

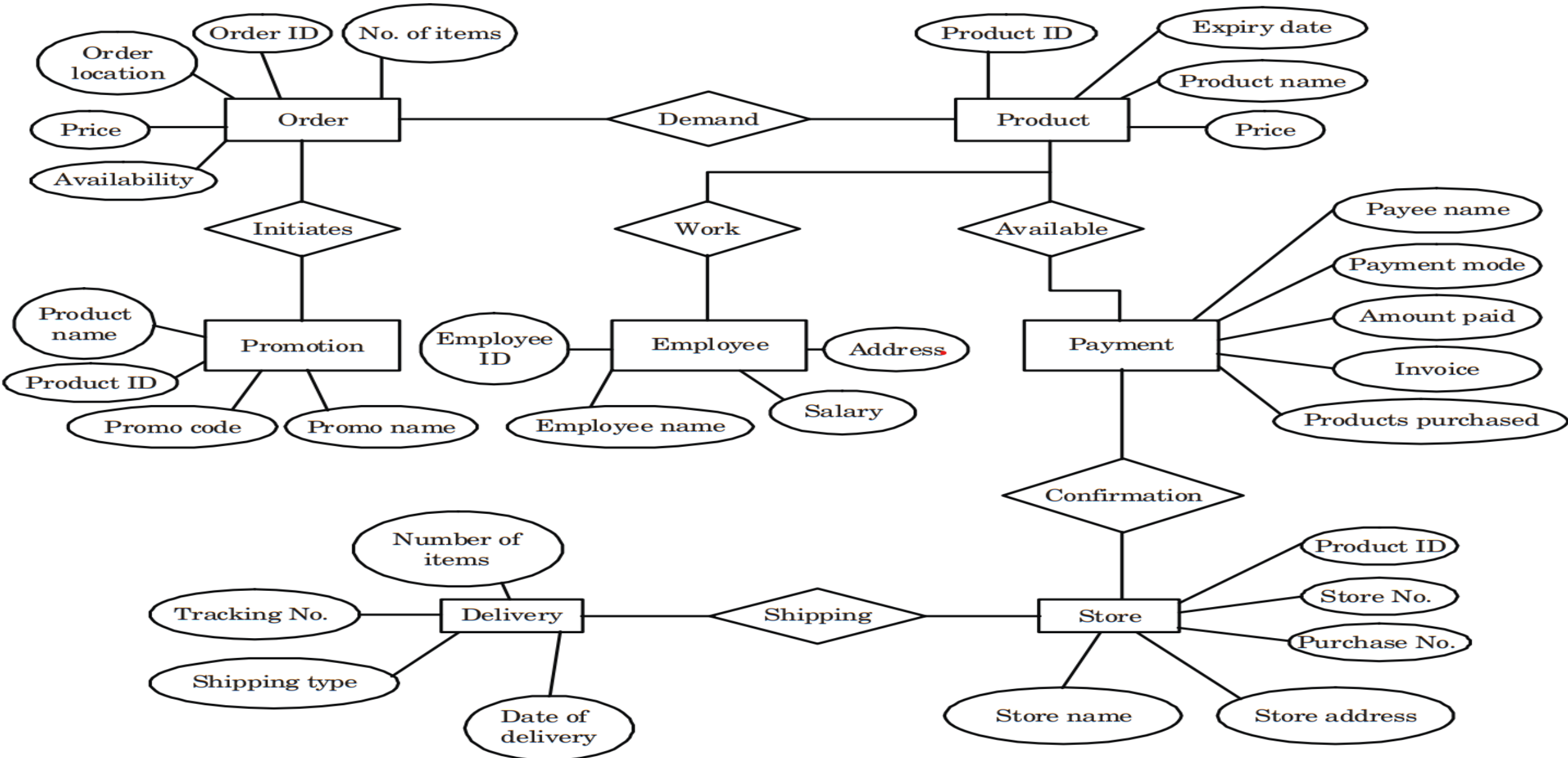


# ER diagram for University System





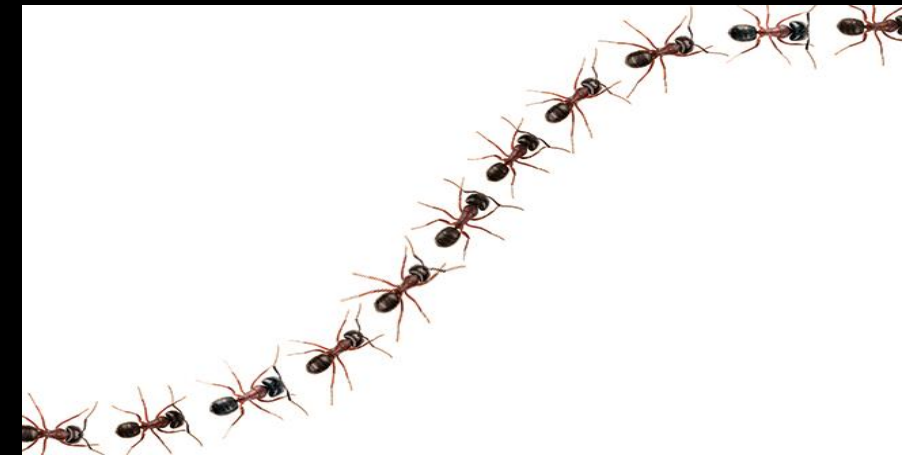
# ER diagram for Marketing Company



# ENTITY

- An entity is a thing or an object in the real world that is distinguishable from other object based on the values of the attributes it possesses.
- An entity may be **concrete**, such as a person or a book, or it may be **abstract**, such as a course, a course offering, or a flight reservation.

Name	FName	City	Age	Salary
Smith	John	3	35	\$280
Doe	Jane	1	28	\$325
Brown	Scott	3	41	\$265
Howard	Shemp	4	48	\$359
Taylor	Tom	2	22	\$250





- In ER diagram we cannot represent an entity, as entity is an instant not schema, and ER diagram is designed to understand schema
- In a relational model entity is represented by a row or a tuple or a record in a table.

CustomerID	FirstName	LastName	Birthdate
XY001	John	Doe	April 18, 1929
BR092	Mary	Green	March 4, 1980
PD500	Francesca	de la Gillebert	September 12, 1959
WI308	John	Green	March 4, 1980

- **ENTITY SET**- Collection of same type of entities that share the same properties or attributes.
- In an ER diagram an entity set is represented by a rectangle
- In a relational model it is represented by a separate table

Name	FName	City	Age	Salary
Smith	John	3	35	\$280
Doe	Jane	1	28	\$325
Brown	Scott	3	41	\$265
Howard	Shemp	4	48	\$359
Taylor	Tom	2	22	\$250



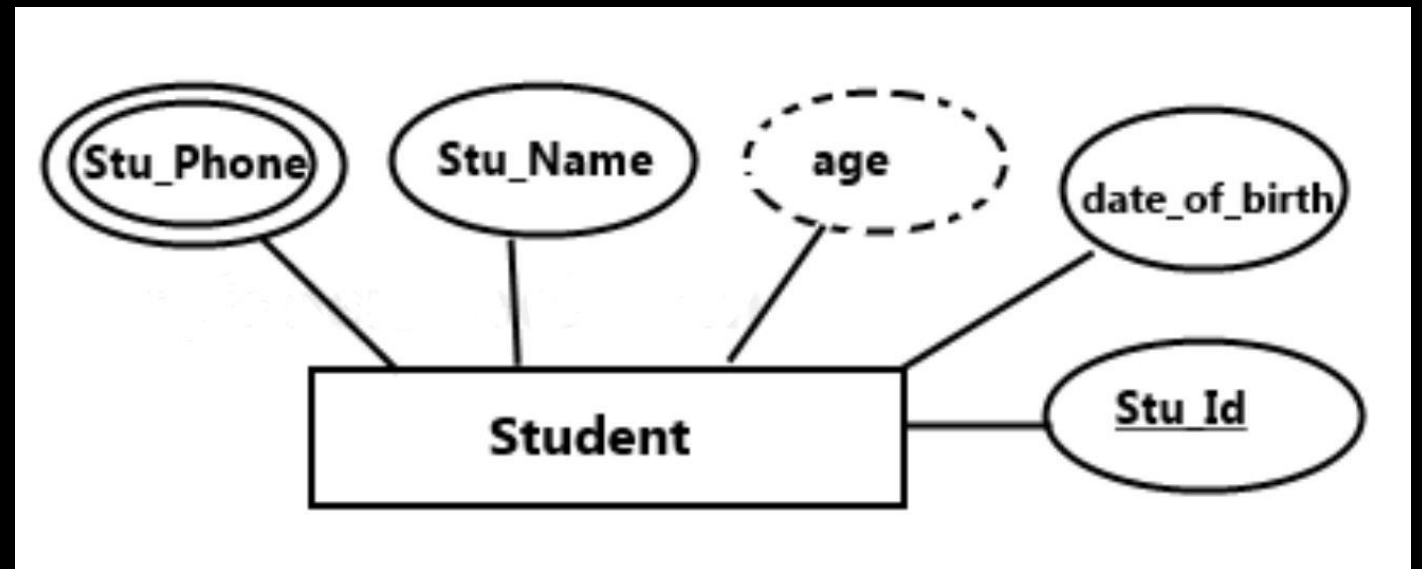
# ATTRIBUTES

- Attributes are the units defines and describe properties and characteristics of entities.
- Attributes are the descriptive properties possessed by each member of an entity set.  
for each attribute there is a set of permitted values called domain.

Name	FName	City	Age	Salary
Smith	John	3	35	\$280
Doe	Jane	1	28	\$325
Brown	Scott	3	41	\$265
Howard	Shemp	4	48	\$359
Taylor	Tom	2	22	\$250

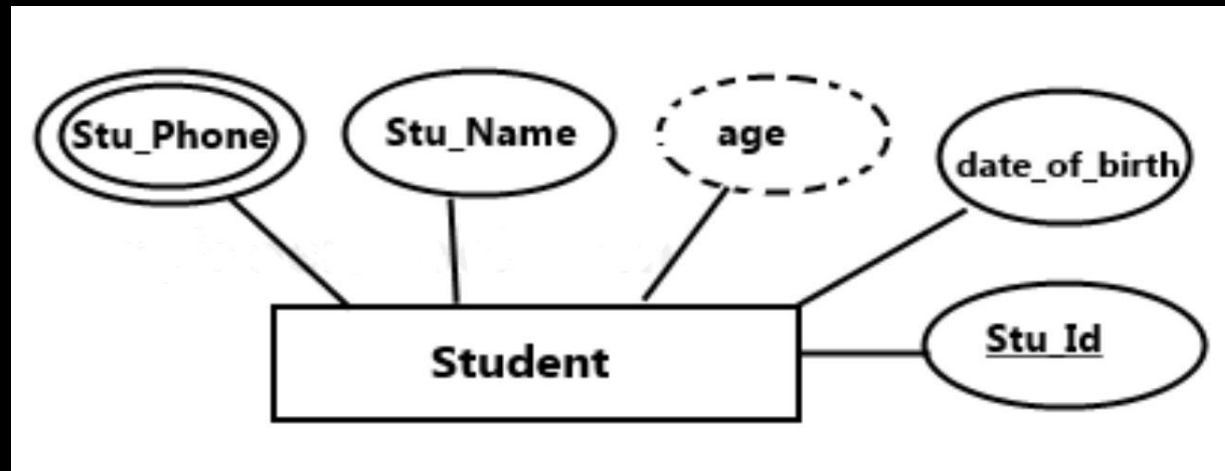
- In an ER diagram attributes are represented by ellipse or oval connected to rectangle.
- While in a relational model they are represented by independent column. e.g. Instructor (ID, name, salary, dept\_name)

Name	FName	City	Age	Salary
Smith	John	3	35	\$280
Doe	Jane	1	28	\$325
Brown	Scott	3	41	\$265
Howard	Shemp	4	48	\$359
Taylor	Tom	2	22	\$250



## Types of Attributes

- **Single valued**- Attributes having single value at any instance of time for an entity. E.g. – Aadhar no, dob.
- **Multivalued** - Attributes which can have more than one value for an entity at same time. E.g.
  - Phone no, email, address.
  - A multivalued attribute is represented by a double ellipse in an ER diagram and by an independent table in a relational model.
  - Separate table for each multivalued attribute, by taking mva and pk of main table as fk in new table





Customer			
Customer ID	First Name	Surname	Telephone Number
123	Rabri Devi	Singh	555-861-2025, 192-122-1111
456	Imarti Devi	Zhang	(555) 403-1659 Ext. 53; 182-929-2929
789	Barfi Devi	Doe	555-808-9633

Customer			
Customer ID	First Name	Surname	Telephone Number
123	Pooja	Singh	555-861-2025, 192-122-1111
456	San	Zhang	(555) 403-1659 Ext. 53; 182-929-2929
789	John	Doe	555-808-9633

जुगाड़ technology



Customer				
Customer ID	First Name	Surname	Telephone Number1	Telephone Number2
123	Pooja	Singh	555-861-2025	192-122-1111
456	San	Zhang	(555) 403-1659 Ext. 53	182-929-2929
789	John	Doe	555-808-9633	

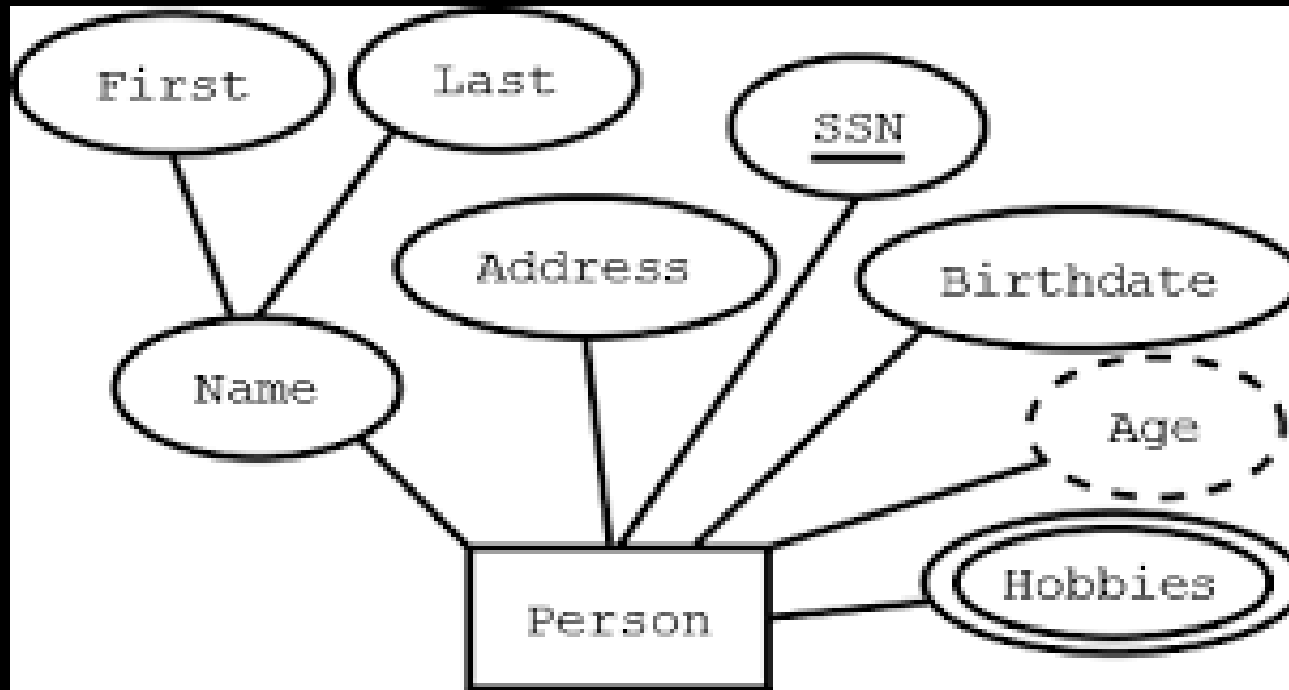
Customer			
<u>Customer ID</u>	First Name	Surname	Telephone Number
123	Rabri Devi	Singh	555-861-2025, 192-122-1111
456	Imarti Devi	Zhang	(555) 403-1659 Ext. 53; 182-929-2929
789	Barfi Devi	Doe	555-808-9633



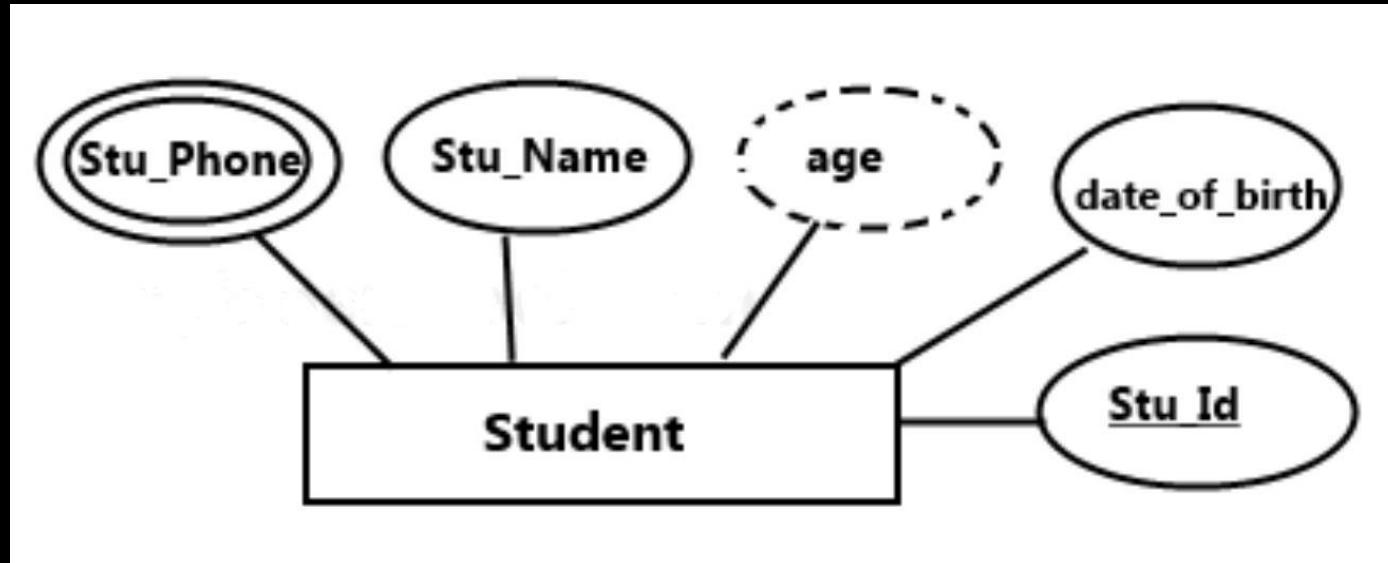
Customer Name		
<u>Customer ID</u>	First Name	Surname
123	Pooja	Singh
456	San	Zhang
789	John	Doe

Customer Phone Number	
<u>Customer ID</u>	<u>Telephone Number</u>
123	555-861-2025
123	192-122-1111
456	(555) 403-1659 Ext. 53
456	182-929-2929
789	555-808-9633

- **Simple** - Attributes which cannot be divided further into sub parts. E.g. Age
- **Composite** - Attributes which can be further divided into sub parts, as simple attributes. A composite attribute is represented by an ellipse connected to an ellipse and in a relational model by a separate column.

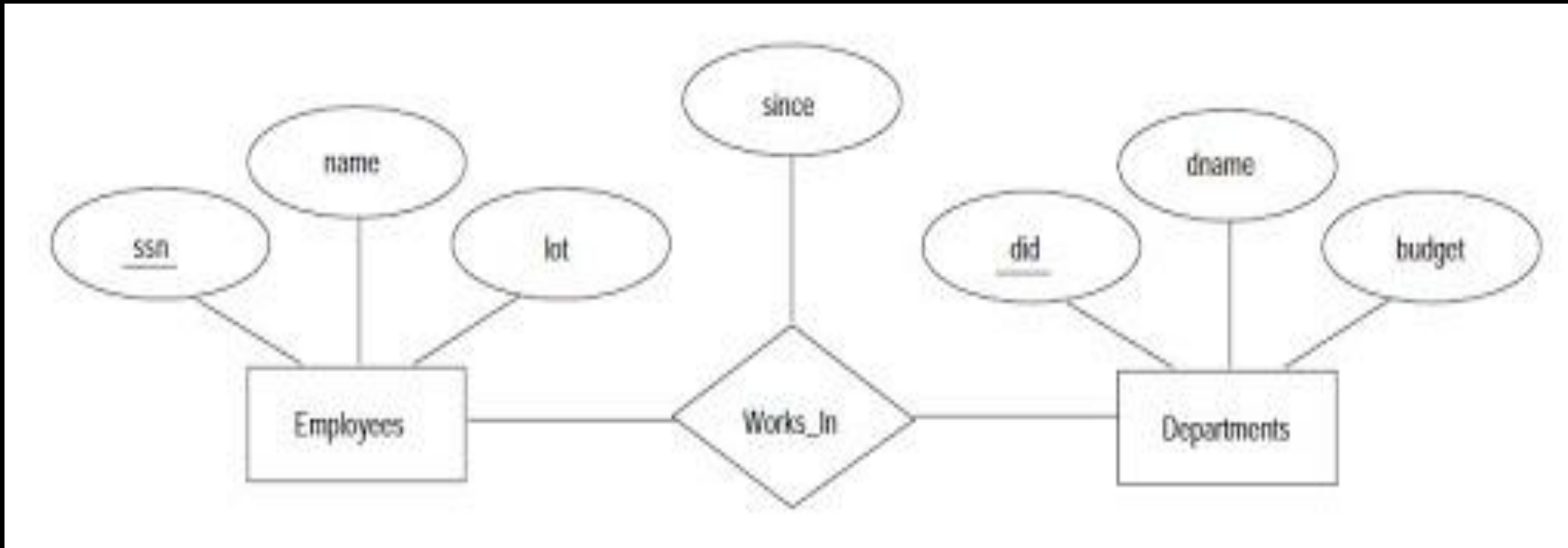


- **Stored** - Main attributes whose value is permanently stored in database. E.g. date\_of\_birth
- **Derived** -The value of these types of attributes can be derived from values of other Attributes. E.g. - Age attribute can be derived from date\_of\_birth and Date attribute.



**Descriptive attribute** - Attribute of relationship is called descriptive attribute.

- An attribute takes a null value when an entity does not have a value for it. The null value may indicate “not applicable” — that is, that the value does not exist for the entity.



# Relationship / Association

Relationship Status:  

Family:

Featured Friends: 

[Create new list - Add an existing list or group](#)

## Relationship / Association

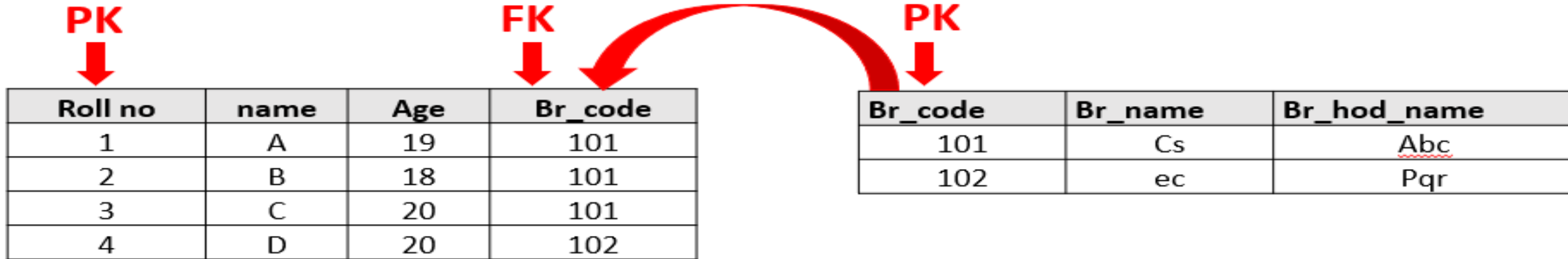
- Is an association between two or more entities of same or different entity set.
- In ER diagram we cannot represent individual relationship as it is an instance or data.





- In an ER diagram it is represented by a diamond, while in relational model sometimes through foreign key and other time by a separate table.

Roll no	name	Age	Br_code	Br_name	Br_hod_name
1	A	19	101	Cs	Abc
2	B	18	101	Cs	Abc
3	C	20	101	Cs	Abc
4	D	20	102	Ec	Pqr



- Every relationship type has three components.
  - Name
  - Degree
  - Structural constraints (cardinalities ratios, participation)

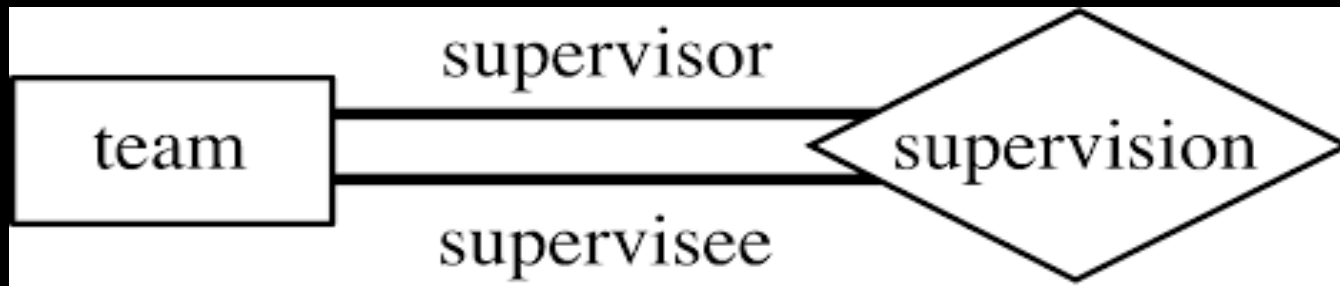
- NAME- every relation must have a unique name.

## Degree of a relationship/relationship set

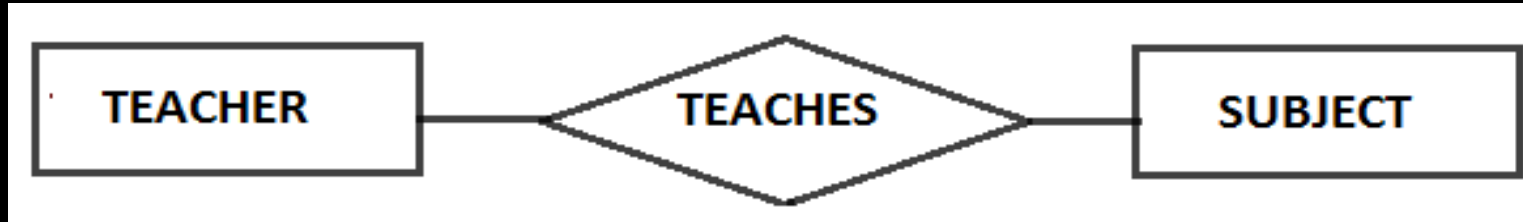
- Means number of entities set(relations/tables) associated(participate) in the relationship set.
- Most of the relationship sets in a data base system are binary.
- Occasionally however relationship sets involve more than two entity sets.
- Logically, we can associate any number of entity set in a relationship called N-ary Relationship.



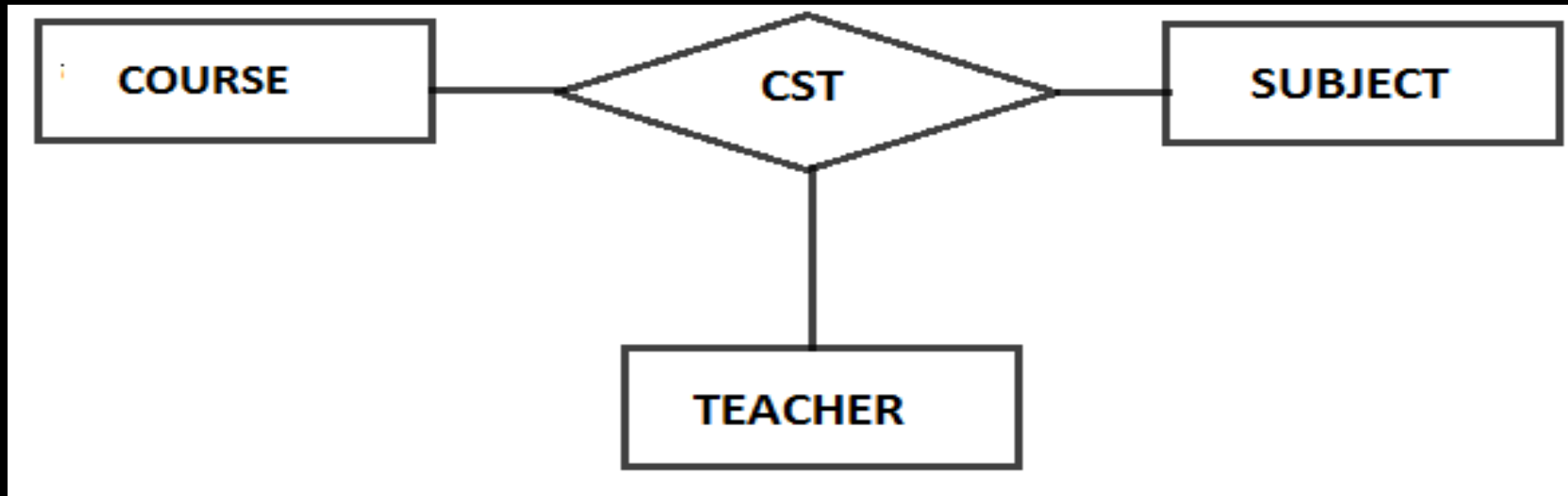
- **Unary Relationship** - One single entity set participate in a Relationship, means two entities of the same entity set are related to each other.
- These are also called as self-referential Relationship set.
- E.g.- A member in a team maybe supervisor of another member in team.



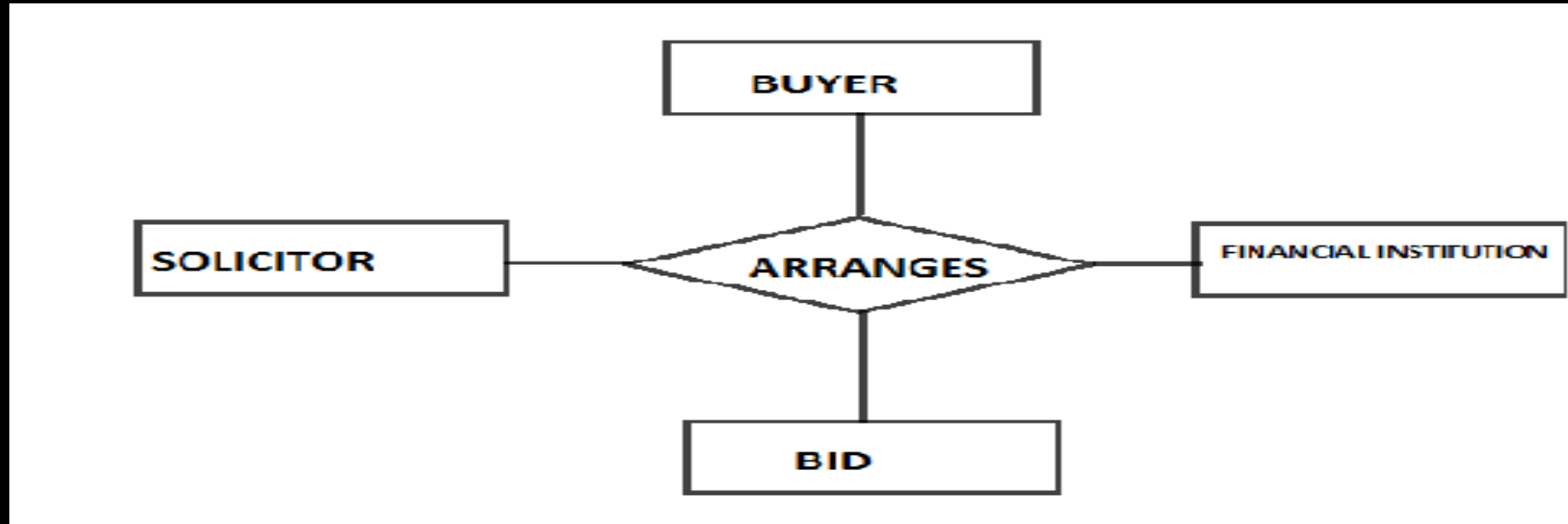
- **Binary Relationship** - Two entity sets participate in a Relationship. It is most common Relationship.



- **Ternary Relationship** - When three entities participate in a Relationship. E.g. The University might need to record which teachers taught which subjects in which courses.

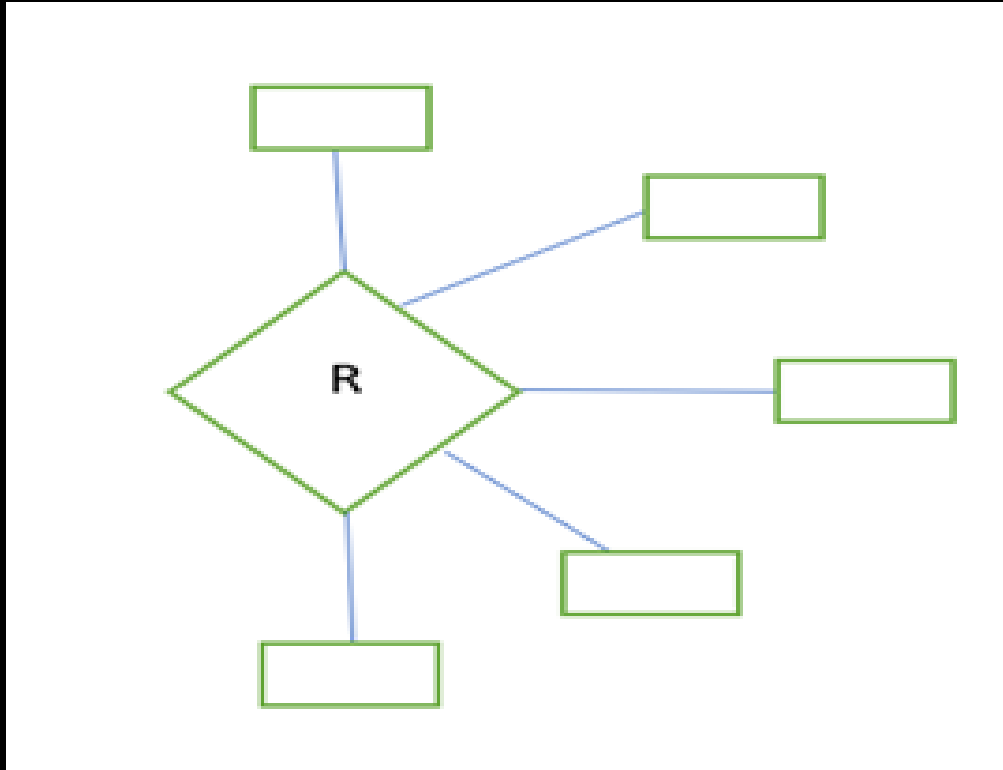


- Quaternary Relationship - When four entities participate in a Relationship.





- N-ary relationship – where n number of entity set are associated



- But the most common relationships in ER models are ***Binary***.

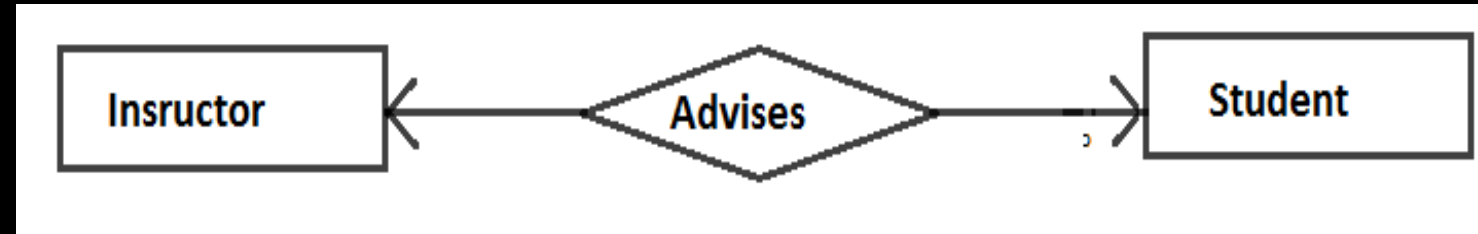
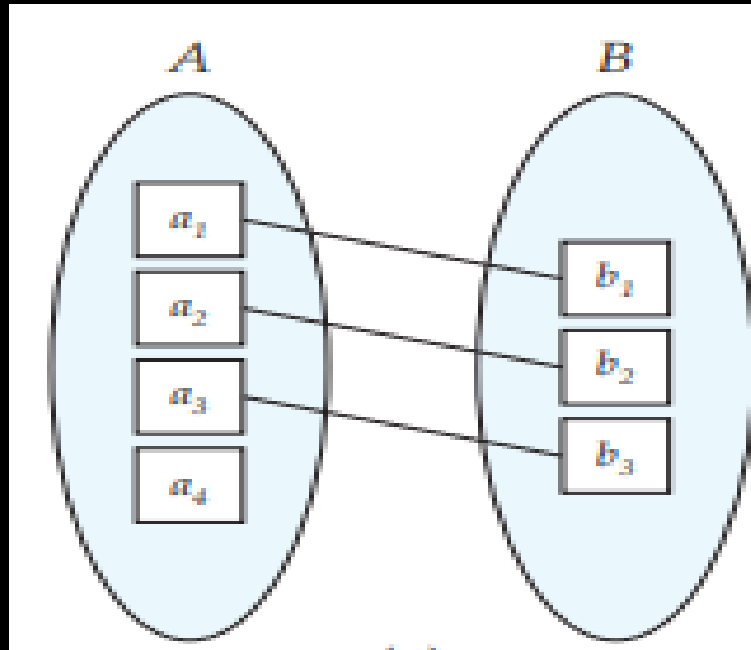
## Structural constraints (Cardinalities Ratios, Participation)

- An E-R enterprise schema may define certain constraints to which the contents of a database must conform.

## MAPPING CARDINALITIES / CARNINALITY RATIOS

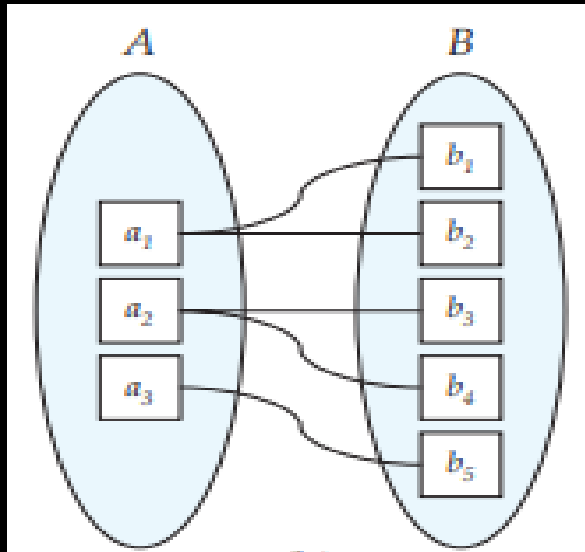
- Express the number of entities to which another entity can be associated via a relationship set. Four possible categories are-
  - One to One (1:1) Relationship.
  - One to Many (1: M) Relationship.
  - Many to One (M: 1) Relationship.
  - Many to Many (M: N) Relationship.

**One to One (1:1) Relationship** - An entity in A is associated with at most one entity in B, and an entity in B is associated with at most one entity in A.



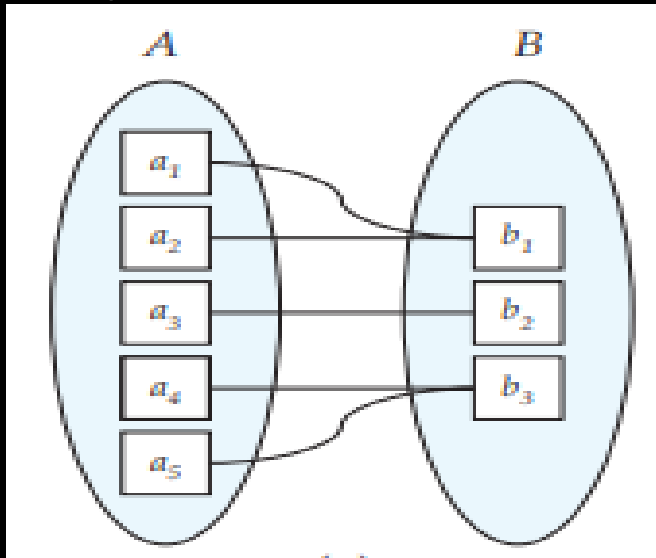
E.g.- The directed line from relationship set advisor to both entities set indicates that 'an instructor may advise at most one student, and a student may have at most one advisor'.

**One to Many (1: M) Relationship** - An entity in A is associated with any number (zero or more) of entities in B. An entity in B, however, can be associated with at most one entity in A.



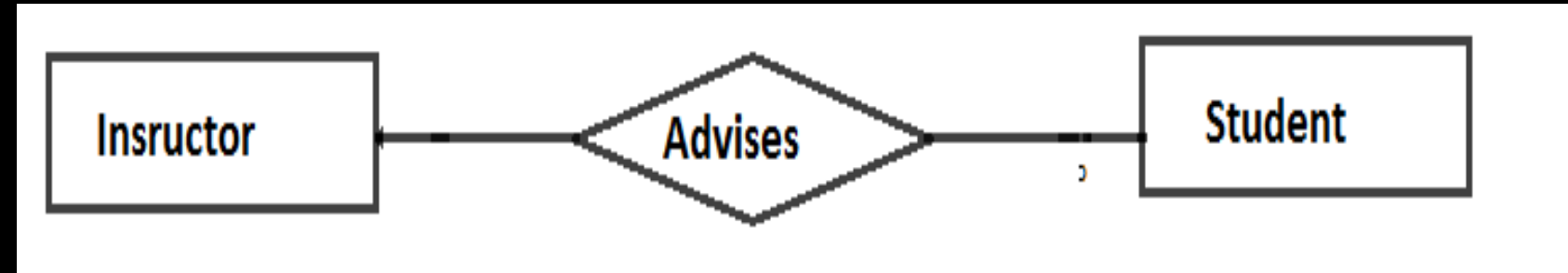
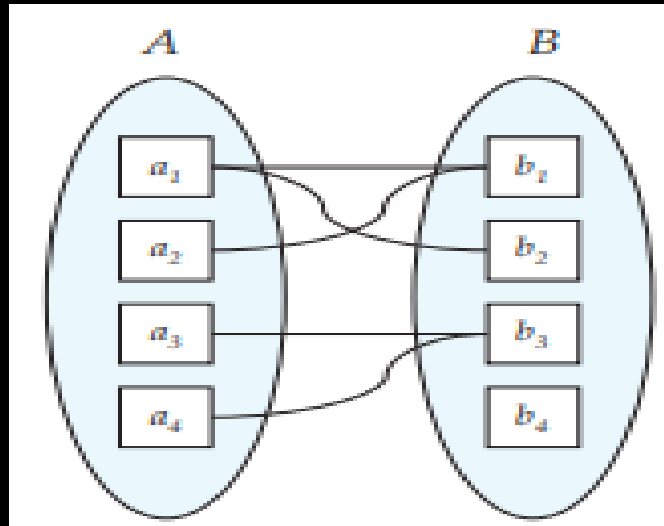
E.g.- This indicates that an instructor may advise many students, but a student may have at most one advisor.

**Many to One (M: 1) Relationship** - An entity in A is associated with at most one entity in B. An entity in B, however, can be associated with any number (zero or more) of entities in A.



E.g.- This indicates that student may have many instructors but an instructor can advise at most one student.

**Many to Many(M:N) Relationship** - An entity in A is associated with any number (zero or more) of entities in B, and an entity in B is associated with any number (zero or more) of entities in A.

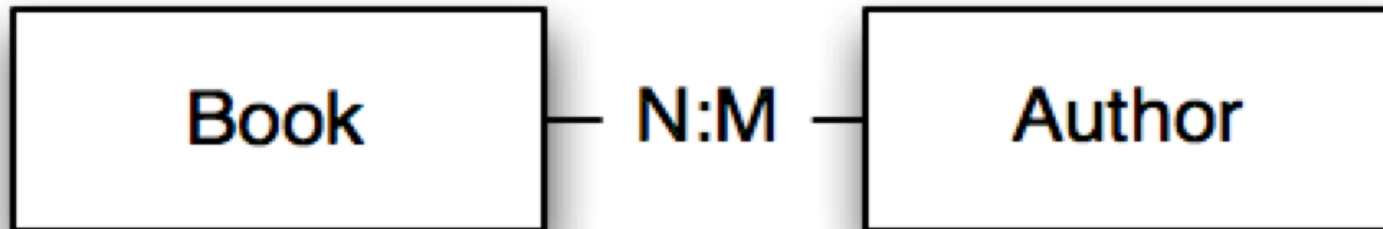


E.g.- This indicates a student may have many advisors and an instructor may advise many students.

- **PARTICIPATION CONSTRAINTS**- it defines participations of entities of an entity type in a relationship.

- **Partial participation**

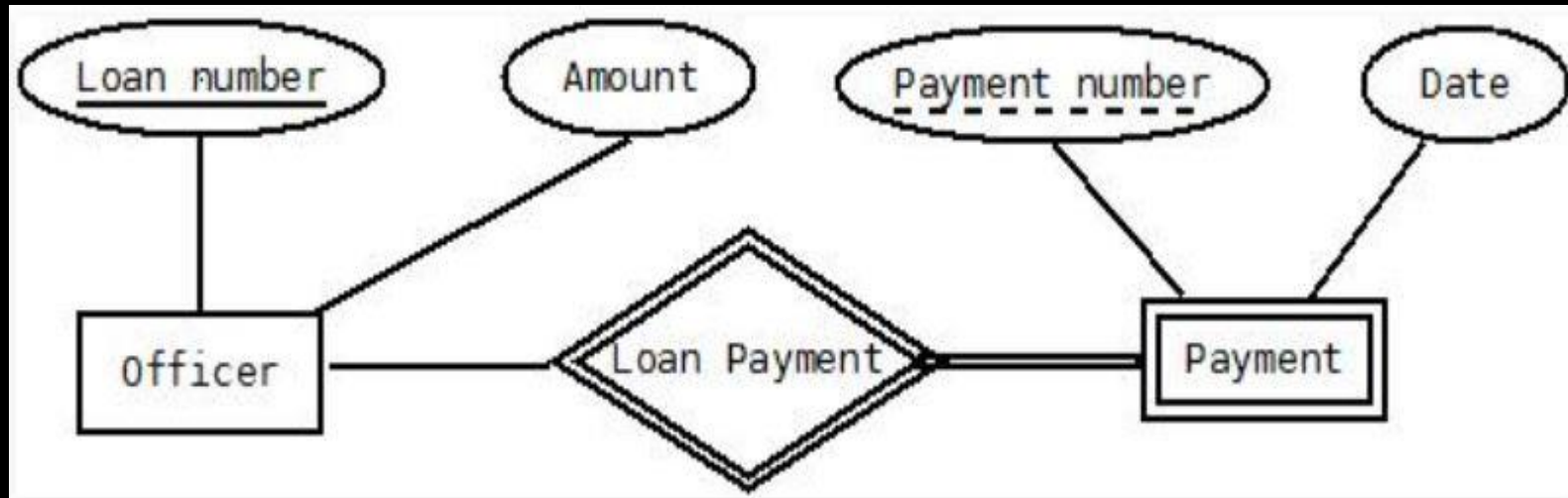
- **Total Participation**



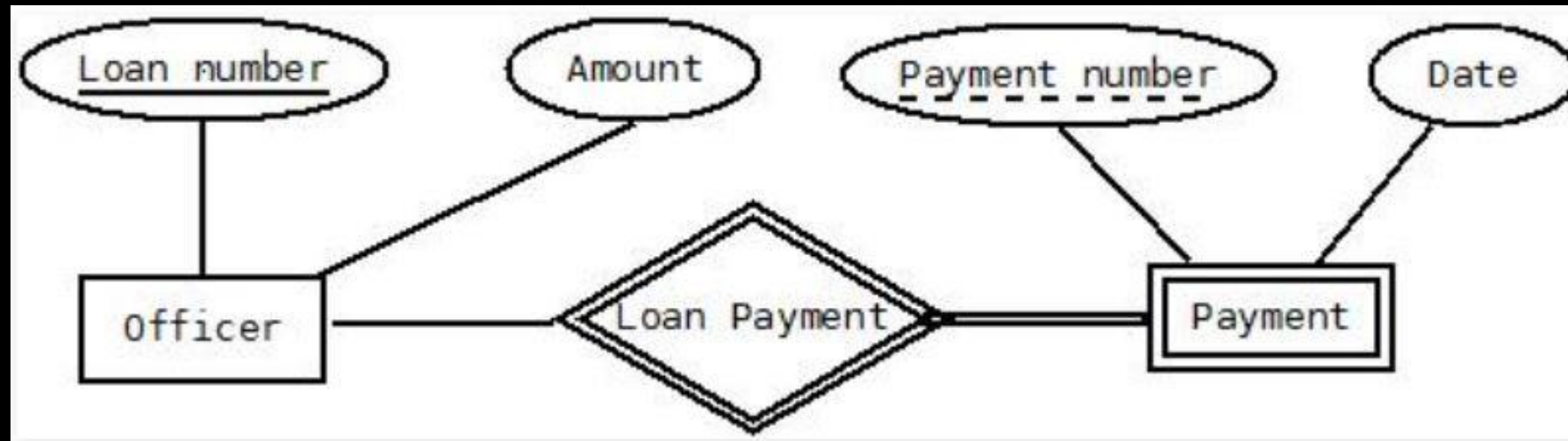


## STRONG AND WEAK ENTITY SET

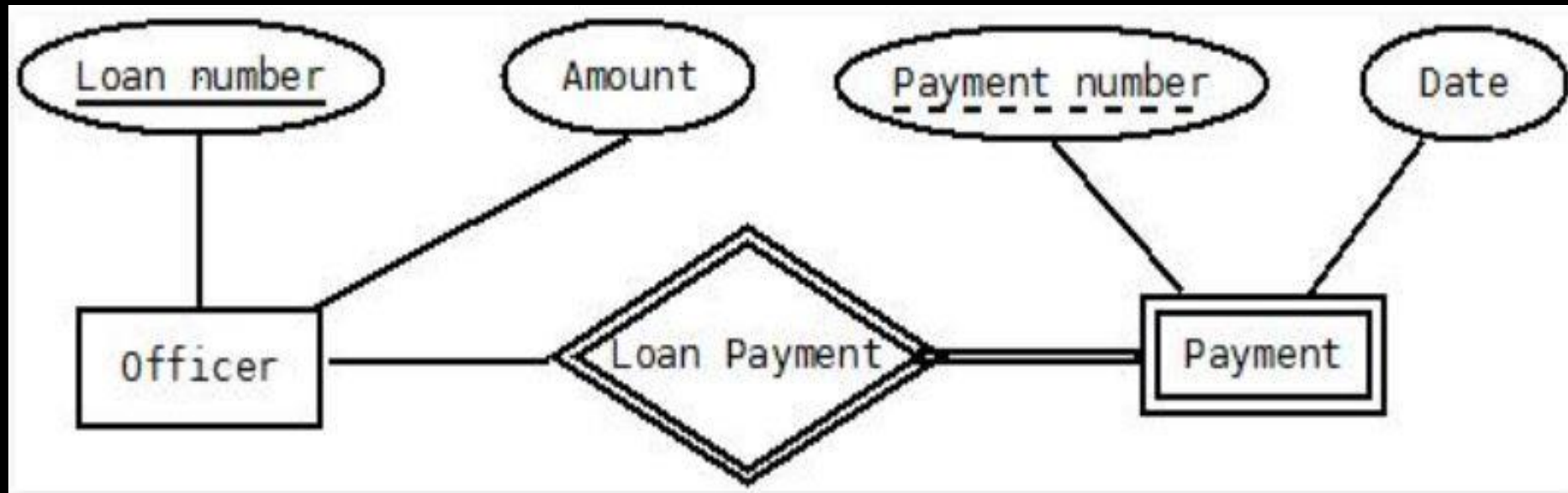
- An entity set is called strong entity set, if it has a primary key, all the tuples in the set are distinguishable by that key.
- An entity set that does not possess sufficient attributes to form a primary key is called a weak entity set.
- It contains discriminator attributes (partial key) which contain partial information about the entity set, but it is not sufficient enough to identify each tuple uniquely. Represented by double rectangle.



- For a weak entity set to be meaningful and converted into strong entity set, it must be associated with another strong entity set called the **identifying or owner entity set** i.e. weak entity set is said to be **existence dependent** on the identity set.
- The identifying entity set is said to own weak entity set that it identifies.
- A weak entity set may participate as owner in an identifying relationship with another weak entity set.



- The relationship associating the weak entity set with the identifying entity set is called the **identifying relationship (double diamonds)**.
- The identifying relationship is many to one from the weak entity set to identifying entity set, and the participation of the weak entity set in relationship is always total.
- The primary key of weak entity set will be the union of primary key and discriminator attributes.



## REASONS TO HAVE WEAK ENTITY SET

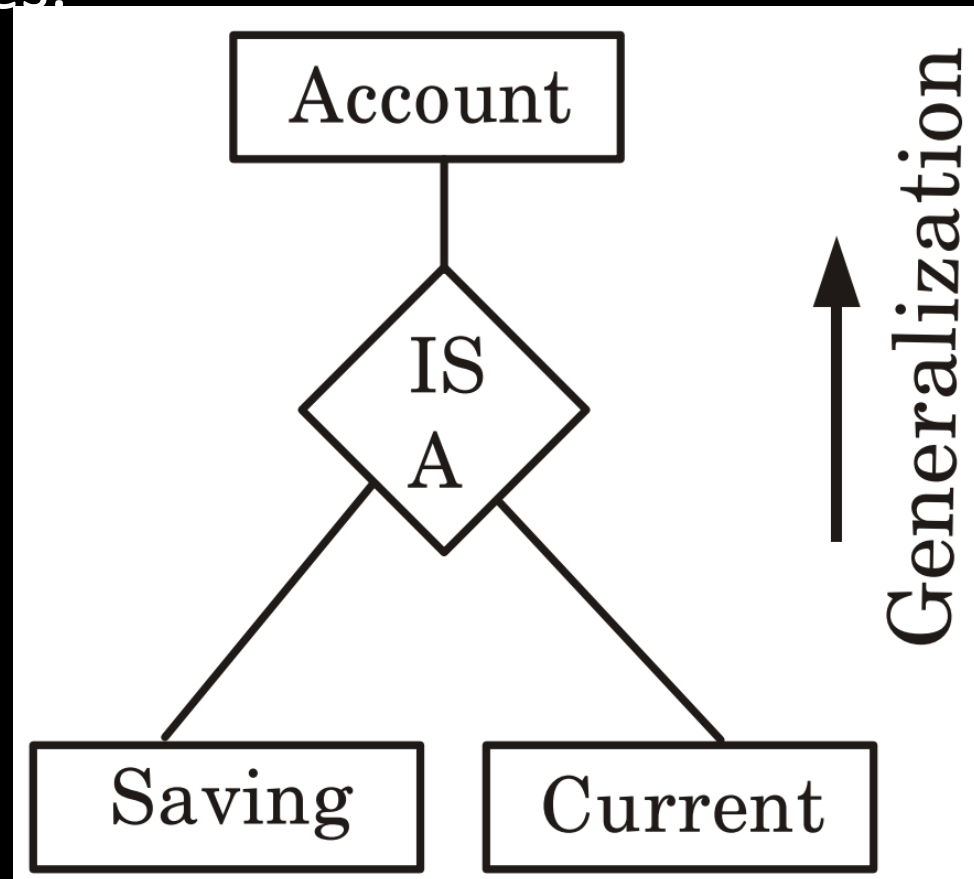
- Weak entities reflect the logical structure of an entity being dependent on another.
- Weak entity can be deleted automatically when their strong entity is deleted.
- Without weak entity set it will lead to duplication and consequent possible inconsistencies.

## Conversion From ER Diagram To Relational Model

- Entity Set
  - Convert every strong, weak entity set into a separate table. In weak entity set we make it dependent onto one strong entity set (**identifying or owner entity set**).
- Relationship
  - If Unary: No separate table is required, add a new column as fk which refer the pk of the same table.
  - if 1:1 No separate table is required, take pk of one side and put it as fk on other side, priority must be given to the side having total participation.
  - if 1:n or n:1 No separate table is required, modify n side by taking pk of 1 side a foreign key on n side.
  - If m-n Separate table is required take pk of both table and declare their combination as a pk of new table
  - (3 or More) Take the pk of all participating entity sets as fk and declare their combinations as pk in the new table.
- Attributes
  - Multivalued-A separate table must be taken for all multivalued attributes, where we take pk of the main table as fk and declare combination of fk and multivalued attribute are pk in the new table.
  - Composite Attributes-A separate column must be taken for all simple attributes of the composite attribute.

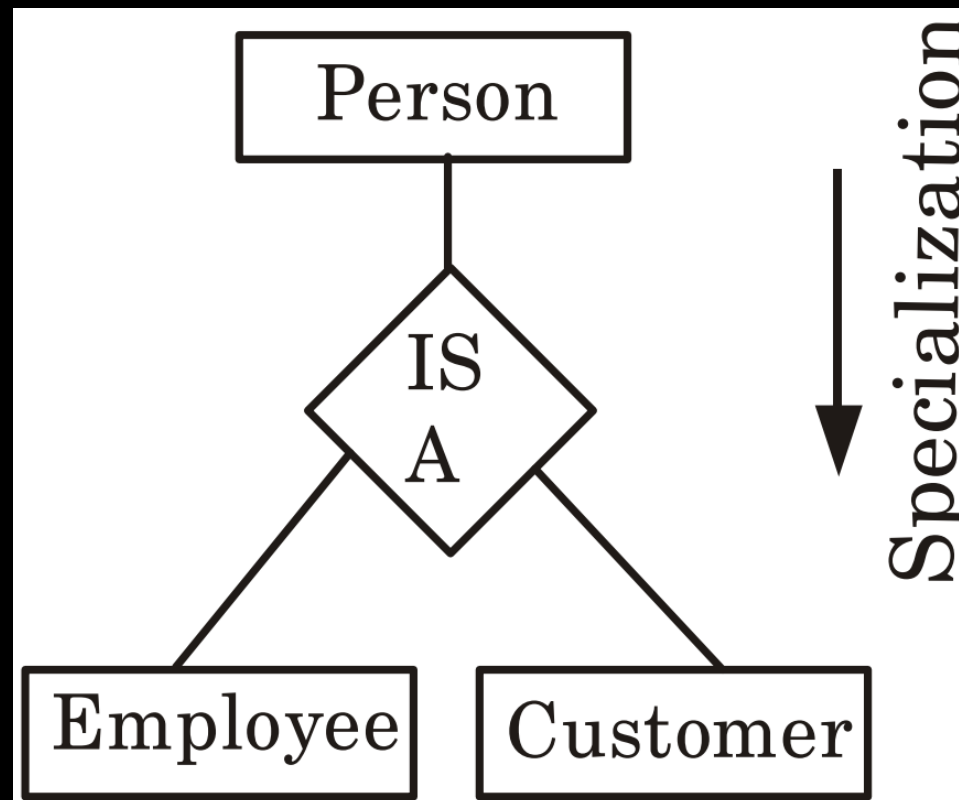
# Generalization

- Involves merging two lower-level entities to create a higher-level entity.
- A bottom-up approach that builds complexity from simpler components.
- Highlights similarities among lower-level entity sets while hiding differences.
- Leads to a simplified, structured data representation, aiding in database design and querying processes.



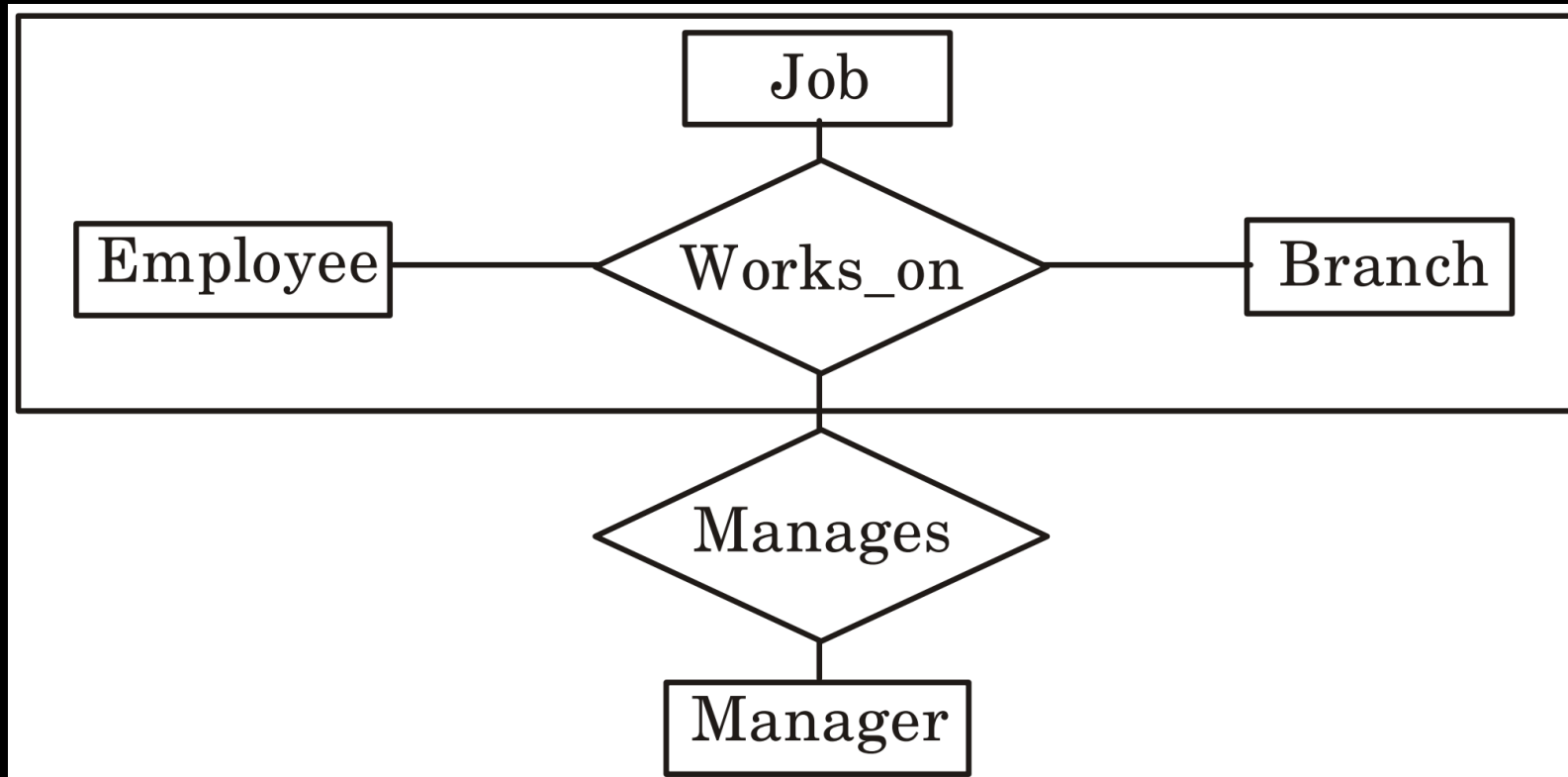
# Specialization

- A process where a higher-level entity is broken down into more specific, lower-level entities.
- This top-down approach delineates complexity into simpler components.
- Acts as the converse of the generalization process, focusing on differentiating properties rather than similarities.



# Aggregation

- A concept wherein relationships are abstracted to form higher-level entities, enabling a more organized representation of complex relationships.





## ADVANTAGES OF E-R DIGRAM

- Constructs used in the ER diagram can easily be transformed into relational tables.
- It is simple and easy to understand with minimum training.

## DISADVANTAGE OF E-R DIGRAM

- Loss of information content
- Limited constraints representation
- It is overly complex for small projects

# RELATIONAL DATABASE MANAGEMENT SYSTEM

- A relational database management system (RDBMS), conceptualized by Edgar F. Codd in 1970, serves as the foundation for most contemporary commercial and open-source database applications.
- Central to its design is the utilization of tables for data storage, where it maintains and enforces specific data relationships, marking a significant evolution in database design.

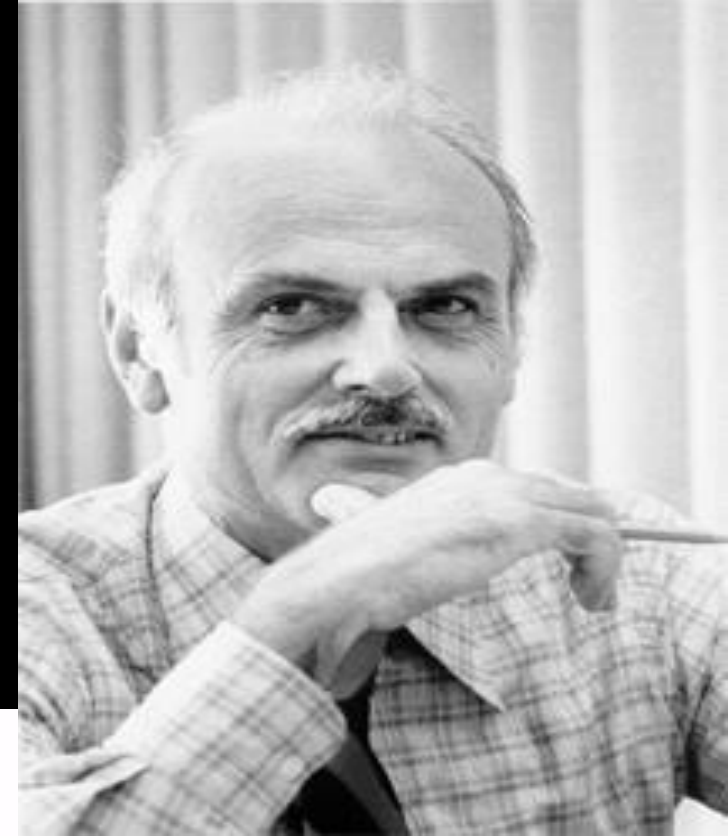


Diagram illustrating a table structure with annotations:


- Column (attribute)
- Table (relation)
- Row (tuple)
- Primary key
- Data value

CustomerID	FirstName	LastName	Birthdate
XY001	John	Doe	April 18, 1929
BR092	Mary	Green	March 4, 1980
PD500	Francesca	de la Gillebert	September 12, 1959
WI308	John	Green	March 4, 1980

# BASICS OF RDBMS

- **Domain (set of permissible value in particular column)** is a set of atomic values.
- By **atomic** we mean that each value in the domain is indivisible as far as the formal relational model is concerned.
- A common method of specifying a domain is to specify a data type from which the data values forming the domain are drawn.
  - E.g. Names: The set of character strings that represent names of persons.

**Domain/  
Field/  
Column/  
Arity/  
Degree**



NAME	ID	CITY	COUNTRY	HOBBY
NISHA	1	AGRA	INDIA	PLAYING
NIKITA	2	DELHI	INDIA	DANCING
AJAY	3	AGRA	INDIA	CHESS
ARPIT	4	PATNA	INDIA	READING

- **Table (Relation)** - A Relation is a set of tuples/rows/entities/records.
- **Tuple** - Each row of a Relation/Table is called Tuple.
- **Arity/Degree** - No. of columns/attributes of a Relation. E.g. - Arity is 5 in Table Student.
- **Cardinality** - No of rows/tuples/record of a Relational instance. E.g. - Cardinality is 4 in table Student.

NAME	ID	CITY	COUNTRY	HOBBY
NISHA	1	AGRA	INDIA	PLAYING
NIKITA	2	DELHI	INDIA	DANCING
AJAY	3	AGRA	INDIA	CHESS
ARPIT	4	PATNA	INDIA	READING

**Rows/Tuples/Record/** →  
**Cardinality**

# Properties of Relational tables

1. Cells contains atomic values
2. Values in a column are of the same kind
3. Each row is unique
4. No two tables can have the same name in a relational schema.
5. Each column has a unique name
6. The sequence of rows is insignificant
7. The sequence of columns is insignificant.

## Problems in relational database

- Update Anomalies- Anomalies that cause redundant work to be done during insertion into and Modification of a relation and that may cause accidental loss of information during a deletion from a relation
  - Insertion Anomalies
  - Modification Anomalies
  - Deletion Anomalies

- **Insertion anomalies:** An independent piece of information cannot be recorded into a relation unless an irrelevant information must be inserted together at the same time.

Roll no	name	Age	Br_code	Br_name	Br_hod_name
1	A	19	101	Cs	Abc
2	B	18	101	Cs	Abc
3	C	20	101	Cs	Abc
4	D	20	102	Ec	Pqr

- **Modification anomalies:** The update of a piece of information must occur at multiple locations.

Roll no	name	Age	Br_code	Br_name	Br_hod_name
1	A	19	101	Cs	Abc
2	B	18	101	Cs	Abc
3	C	20	101	Cs	Abc
4	D	20	102	Ec	Pqr



- **Deletion Anomalies:** The deletion of a piece of information unintentionally removes other information.

Roll no	name	Age	Br_code	Br_name	Br_hod_name
1	A	19	101	Cs	Abc
2	B	18	101	Cs	Abc
3	C	20	101	Cs	Abc
4	D	20	102	Ec	Pqr

Roll no	name	Age	Br_code	Br_name	Br_hod_name
1	A	19	101	Cs	Abc
2	B	18	101	Cs	Abc
3	C	20	101	Cs	Abc
4	D	20	102	Ec	Pqr

**PK**  
↓

**FK**  
↓

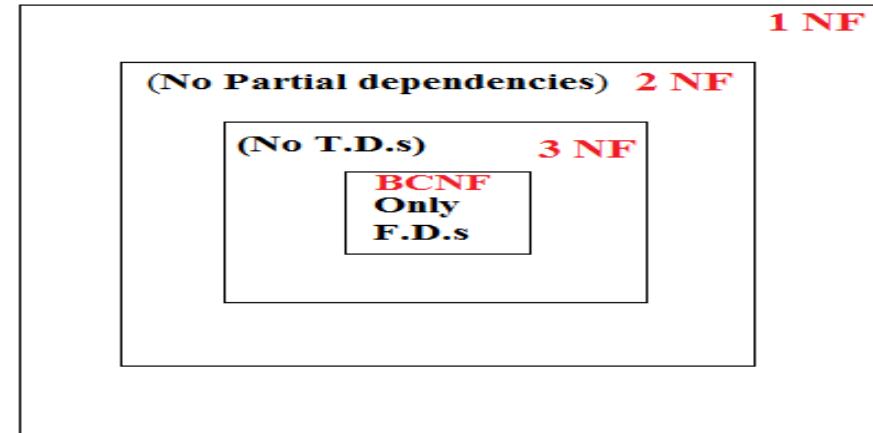
**PK**  
↓

Roll no	name	Age	Br_code
1	A	19	101
2	B	18	101
3	C	20	101
4	D	20	102

Br_code	Br_name	Br_hod_name
101	Cs	Abc
102	ec	Pqr

# Purpose of Normalization

- Normalization may be simply defined as refinement process. Which includes creating tables and establishing relationships between those tables according to rules designed both to protect data and make the database more flexible by eliminating two factors;
  - Redundancy
  - Inconsistent Dependency
- With out normalization data base system may be inaccurate, slow and inefficient and they might not produce the data we expect. A series of normal form tests that can be carried out on individual relation schemas so that the relational database can be **normalized** to any desired degree.
- 1NF>>>2NF>>3NF>>BCNF



Hierarchy of Normal forms

## Conclusion

1. Like every paragraph must have a single idea similarly every table must have a single idea and if a table contains more than one idea then that table must be decomposed until each table contains only one idea.
2. We need some tools to approach this decomposition or normalization on large database which contains a number of table, and that tool is functional dependencies.

Roll no	name	Age	Br_code	Br_name	Br_hod_name
1	A	19	101	Cs	Abc
2	B	18	101	Cs	Abc
3	C	20	101	Cs	Abc
4	D	20	102	Ec	Pqr

Roll no	name	Age	Br_code
1	A	19	101
2	B	18	101
3	C	20	101
4	D	20	102

Br_code	Br_name	Br_hod_name
101	Cs	Abc
102	ec	Pqr

# Functional Dependency



**Br\_code**



**Br\_hod\_name**

**Br\_code**



**Br\_hod\_name**



Functional Dependency कोई बताता नहीं, इसकी feel आ जाती है



## FUNCTIONAL DEPENDENCY

- A formal tool for analysis of relational schemas.
- In a Relation R, if ' $\alpha$ '  $\subseteq$  R AND ' $\beta$ '  $\subseteq$  R, then attribute or a Set of attribute ' $\alpha$ ' Functionally derives an attribute or set of attributes ' $\beta$ ', iff each ' $\alpha$ ' value is associated with precisely one ' $\beta$ ' value.
- For all pairs of tuples  $t_1$  and  $t_2$  in R such that
  - If  $T_1[\alpha] = T_2[\alpha]$
  - Then,  $T_1[\beta] = T_2[\beta]$

X	Y	Z
1	4	2
1	4	3
2	6	3
3	2	2

**Q** Consider the following relation instance, which of the following dependency doesn't hold

**A)**  $A \rightarrow b$

**B)**  $BC \rightarrow A$

**C)**  $B \rightarrow C$

**D)**  $AC \rightarrow B$

A	B	C
1	2	3
4	2	3
5	3	3

- Trivial Functional dependency - If  $\beta$  is a subset of  $\alpha$ , then the functional dependency  $\alpha \rightarrow \beta$  will always hold.

जिसका होना न होना बराबर हो



X	Y	Z
1	4	2
1	4	3
2	6	3
3	2	2

# ATTRIBUTES CLOSURE/CLOSURE ON ATTRIBUTE SET/ CLOSURE SET OF ATTRIBUTES

- Attribute closure of an attribute set can be defined as set of attributes which can be functionally determined from F.
  - DENOTED BY  $F^+$

R (A, B, C, D)

$A \rightarrow B$

$B \rightarrow C$

$AB \rightarrow D$

$A^+ =$

R(ABCDEFGG)

$A \rightarrow B$

$BC \rightarrow DE$

$AEG \rightarrow G$

$(AC)^+ = ?$

Q R(ABCDE)

$A \rightarrow BC$

$CD \rightarrow E$

$B \rightarrow D$

$E \rightarrow A$

$(B)^+ =$

# ARMSTRONG'S AXIOMS

1. An **axiom** or **postulate** is a statement that is taken to be true, to serve as a premise or starting point for further reasoning and arguments.
2. **Armstrong's axioms** are a set of axioms (or, more precisely, inference rules) used to infer all the functional dependencies on a relational database. They were developed by William W. Armstrong in his 1974 paper.
3. The axioms are sound in generating only functional dependencies in the closure of a set of functional dependencies (denoted as  $F^+$ ) when applied to that set (denoted as  $F$ ).



# Armstrong Axioms

- **Reflexivity:** If  $Y$  is a subset of  $X$ , then  $X \rightarrow Y$
- **Augmentation:** If  $X \rightarrow Y$ , then  $XZ \rightarrow YZ$
- **Transitivity:** If  $X \rightarrow Y$  and  $Y \rightarrow Z$ , then  $X \rightarrow Z$

From these rules, we can derive these secondary rules-

- **Union:** If  $X \rightarrow Y$  and  $X \rightarrow Z$ , then  $X \rightarrow YZ$
- **Decomposition:** If  $X \rightarrow YZ$ , then  $X \rightarrow Y$  and  $X \rightarrow Z$
- **Pseudo transitivity:** If  $X \rightarrow Y$  and  $WY \rightarrow Z$ , then  $WX \rightarrow Z$
- **Composition:** If  $X \rightarrow Y$  and  $Z \rightarrow W$ , then  $XZ \rightarrow YW$



## Why Armstrong axioms refers to the Sound and Complete

- By sound, we mean that given a set of functional dependencies  $F$  specified on a relation schema  $R$ , any dependency that we can infer from  $F$  by using the primary rules of Armstrong axioms holds in every relation state  $r$  of  $R$  that satisfies the dependencies in  $F$ .
- By complete, we mean that using primary rules of Armstrong axioms repeatedly to infer dependencies until no more dependencies can be inferred results in the complete set of all possible dependencies that can be inferred from  $F$ .

## Equivalence of Two FD sets-

Two FD sets  $F_1$  and  $F_2$  are equivalent if –

$$F_1^+ = F_2^+$$

Or

$$F_1 \subseteq F_2^+ \text{ and } F_2 \subseteq F_1^+$$

Q Consider the following set of fd  $R(ACDEH)$

F	G
$A \rightarrow C$	$A \rightarrow CD$
$AC \rightarrow D$	
$E \rightarrow AD$	$E \rightarrow AH$
$E \rightarrow H$	

## To find the MINIMAL COVER / CANONICAL COVER / IRREDUCIBLE SET

- A **canonical cover** (also known as a minimal cover) for a set of functional dependencies in a database is a minimal set of functional dependencies that is equivalent to the original set, but with redundant dependencies and extraneous attributes removed. It is used in the normalization process of database design to simplify the set of functional dependencies and to find a good set of relations.
- There may be any following type of redundancy in the set of functional dependencies: -
  - Complete production may be Redundant.
  - One or more than one attributes may be redundant on right hand side of a production.
  - One or more than one attributes may be redundant on Left hand side of a production.

$Q \rightarrow R(ABCD)$

$A \rightarrow B$

$C \rightarrow B$

$D \rightarrow ABC$

$AC \rightarrow D$

$R(A,B,C)$

$A \rightarrow B$

$B \rightarrow C$

$A \rightarrow C$

$AB \rightarrow B$

$AB \rightarrow C$

$AC \rightarrow B$

# Key



## Super key

- Set of attributes using which we can identify each tuple uniquely is called Super key.
- Let  $X$  be a set of attributes in a Relation  $R$  , if  $X^+$ (Closure of  $X$ ) determines all attributes of  $R$  then  $X$  is said to be Super key of  $R$  .
- There should be at least one Super key in every relation.



## Candidate key

- Minimal set of attributes using which we can identify each tuple uniquely is called Candidate key. A super key is called candidate key if it's No proper subset is a super key. Also called as **MINIMAL SUPER KEY**.
- There should be at least one candidate key.

**Prime attribute** - Attributes that are member of at least one candidate Keys are called Prime attributes.

## Primary key

- One of the candidate keys is selected by database administrator as a Primary means to identify tuple is called primary Key. Primary Key attribute are not allowed to have Null values. Exactly one Primary Key per table in RDMS.
- Candidate key which are not chosen as primary key is alternate key.

# Foreign Keys

- A foreign key is a column or group of columns in a relational database table that refers the primary key of the same table or some other table to represent relationship.
- The *concept of referential integrity* is derived from foreign key theory.

Roll no	name	Age	Br_code	Br_name	Br_hod_name
1	A	19	101	Cs	Abc
2	B	18	101	Cs	Abc
3	C	20	101	Cs	Abc
4	D	20	102	Ec	Pqr

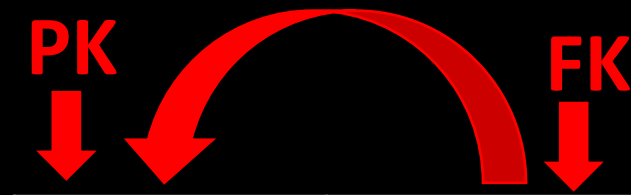
**PK**  
↓

**FK**  
↓

**PK**  
↓

Roll no	name	Age	Br_code
1	A	19	101
2	B	18	101
3	C	20	101
4	D	20	102

Br_code	Br_name	Br_hod_name
101	Cs	Abc
102	ec	Pqr



Roll no	CR
1	1
2	2
3	1
4	2
5	1
6	2
7	1
8	2
9	1
10	2
11	1
12	2
13	1
14	2
15	null

- **Composite key** – Composite key is a key composed of more than one column sometimes it is also known as concatenated key.
- **Secondary key** – Secondary key is a key used to speed up the search and retrieval contrary to primary key, a secondary key does not necessarily contain unique values.

# NORMAL FORM

---



## FIRST NORMAL FORM

- 1NF is the initial step of database normalization.
- Implications of first normal form
  - **Atomic Values**: Each cell in a table contains indivisible, atomic values. Means a Relation should not contain any multivalued or composite attributes.
  - **Unique Columns**: Each column must have a distinct name to identify the data it contains.
  - **Primary Key**: A table in 1NF should have a primary key that uniquely identifies each record.
  - **Eliminating Duplicates**: Duplicate rows are removed to prevent data redundancy.

**Prime attribute:** - A attribute is said to be prime if it is part of any of the candidate key

**Non-Prime attribute:** - A attribute is said to be non-prime if it is not part of any of the candidate key

**Eg** R(ABCD)

AB → CD

Here candidate key is AB so, A and B are prime attribute, C and D are non-prime attributes.

**PARTIAL DEPENDENCY**- When a non – prime attribute is dependent only on a part (Proper subset) of candidate key then it is called partial dependency. (PRIME > NON-PRIME)

**Full DEPENDENCY**- When a non – prime attribute is dependent on the entire candidate key then it is called Full dependency.

e.g. R(ABCD)

$AB \rightarrow D$

$A \rightarrow C$

## SECOND NORMAL FORM

- Relation R is in 2NF if,
  - R should be in 1 NF.
  - R should not contain any Partial dependency. (that is every non-prime attribute should be fully dependent upon candidate key)

Q R(A, B, C)  $B \rightarrow C$

A	B	C
a	1	X
b	2	Y
a	3	Z
C	3	Z
D	3	Z
E	3	Z

A	B
A	1
B	2
A	3
C	3
D	3
E	3

B	C
1	X
2	Y
3	Z

#उन\_हवाओं\_से\_भी\_जल्द\_सामना\_होगा  
#जो\_हमारे\_खिलाफ\_चल\_रही\_है\_ 🇮🇳 🇮🇳



**TRANSITIVE DEPENDENCY** – A functional dependency from non-Prime attribute to non-Prime attribute is called transitive

E.g.- R(A, B, C, D) with A as a candidate key

$A \rightarrow B$

$B \rightarrow C$  [ transitive dependency]

$C \rightarrow D$  [transitive dependency]

## THIRD NORMAL FORM

- Let R be the relational schema, it is said to be in 3 NF
  - R should be in 2NF
  - It must not contain any transitive dependency



## THIRD NORMAL FORM DIRECT DEFINATION

- A relational schema R is said to be 3 NF if every functional dependency in R from  $\alpha \twoheadrightarrow \beta$ , either  $\alpha$  is super key or  $\beta$  is the prime attribute

$R(A, B, C)$

$A \rightarrow B$

$B \rightarrow C$

A	B	C
A	1	P
B	2	Q
C	2	Q
D	2	Q
E	3	R
F	3	R
G	4	S

A	B
A	1
B	2
C	2
D	2
E	3
F	3
G	4

B	C
1	P
2	Q
3	R
4	S

## BCNF (BOYCE CODD NORMAL FORM)

- A relational schema R is said to be BCNF if every functional dependency in R from
  - $\alpha \rightarrow \beta$
  - $\alpha$  must be a super key

$R(A, B, C)$

$AB \rightarrow C$

$C \rightarrow B$

A	B	C
A	C	B
B	B	C
B	A	D
A	A	E
C	C	B
D	C	B
E	C	B
F	C	B

A	B
A	B
B	B
B	A
A	A
C	C
D	C
e	C
f	c

C	B
B	C
C	B
D	A
E	A

## Some important note points on Normalization:

- A Relation with two attributes is always in BCNF.
- A Relation schema R consist of only prime attributes then R is always in 3NF, but may or may not be in BCNF.

**Q** Consider the universal relational schema R (A, B, C, D, E, F, G, H, I, J) and a set of following functional dependencies.

$F = \{AB \rightarrow C, A \rightarrow DE, B \rightarrow F, F \rightarrow GH, D \rightarrow IJ\}$  determine the keys for R ?

Decompose R into 2nd normal form.

**Q** Find the normal form of relation  $R(A, B, C, D, E)$  having FD set  $F = \{A \rightarrow B, BC \rightarrow E, ED \rightarrow A\}$ .

# Multivalued Dependency

- Denoted by,  $A \twoheadrightarrow B$ , Means, for every value of A, there may exist more than one value of B.
- E.g.  $S\_name \twoheadrightarrow Club\_name$

<b>S_Name</b>	<b>Club_name</b>
Kamesh	Dance
Kamesh	Guitar



- A **trivial multivalued dependency**  $X \twoheadrightarrow Y$  is one where either  $Y$  is a subset of  $X$ , or  $X$  and  $Y$  together form the whole set of attributes of the relation.

$S\_name \twoheadrightarrow Club\_name$

S_Name	Club_name
Kamesh	Dance
Kamesh	Guitar

$S\_name \twoheadrightarrow Club\_name$   
 $S\_name \twoheadrightarrow P\_no$

S_Name	Club_name	P_no
Kamesh	Dance	123
Kamesh	Guitar	123
Kamesh	Dance	789
Kamesh	Guitar	789

**E.g.** let the constraint specified by MVD in relation **Student** as

$S\_name \twoheadrightarrow Club\_name$

$S\_name \twoheadrightarrow P\_no$

S_Name	Club_name
Kamesh	Dance
Kamesh	Guitar

S_Name	P_no
Kamesh	123
Kamesh	789

S_Name	Club_name	P_no
Kamesh	Dance	123
Kamesh	Guitar	123
Kamesh	Dance	789
Kamesh	Guitar	789

NOTE: The above Student schema is in BCNF as no functional dependency holds on EMP, but still redundancy due to MVD.

- Each row indicates that a given restaurant can deliver a given variety. The table has no non-key attributes because its only key is {Restaurant, Variety, Delivery Area}. Therefore, it meets all normal forms up to BCNF.

- If we assume, however, that **Variety** offered by a restaurant are not affected by delivery area (i.e. a restaurant offers all **Variety** it makes to all areas it supplies), then it does not meet 4NF. The problem is that the table features two non-trivial multivalued dependencies on the {Restaurant} attribute (which is not a super key). The dependencies are:

- {Restaurant}      {Variety}
- {Restaurant}      {Delivery Area}

Restaurant Delivery Permutations		
Restaurant	Variety	Delivery Area
Chatora Sweets	Samosa	Hatibagan Market
Chatora Sweets	Samosa	Chandni Chowk
Chatora Sweets	Samosa	Koramangala
Chatora Sweets	Dosa	Hatibagan Market
Chatora Sweets	Dosa	Chandni Chowk
Chatora Sweets	Dosa	Koramangala
Moolchand	Ladoo	Koramangala
Moolchand	Dosa	Koramangala
Thaggu	Samosa	Hatibagan Market
Thaggu	Samosa	Chandni Chowk
Thaggu	Ladoo	Hatibagan Market
Thaggu	Ladoo	Chandni Chowk

- If we have two or more multivalued *independent* attributes in the same relation schema, we get into a problem of having to repeat every value of one of the attributes with every value of the other attribute to keep the relation state consistent and to maintain the independence among the attributes involved. This constraint is specified by a multivalued dependency.

Delivery Areas By Restaurant	
Restaurant	Delivery Area
Chatora Sweets	Hatibagan Market
Chatora Sweets	Chandni Chowk
Chatora Sweets	Koramangala
Moolchand	Koramangala
Thaggu	Hatibagan Market
Thaggu	Chandni Chowk

Varieties By Restaurant	
Restaurant	Pizza Variety
Chatora Sweets	Samosa
Chatora Sweets	Dosa
Moolchand	Ladoo
Moolchand	Dosa
Thaggu	Samosa
Thaggu	Ladoo

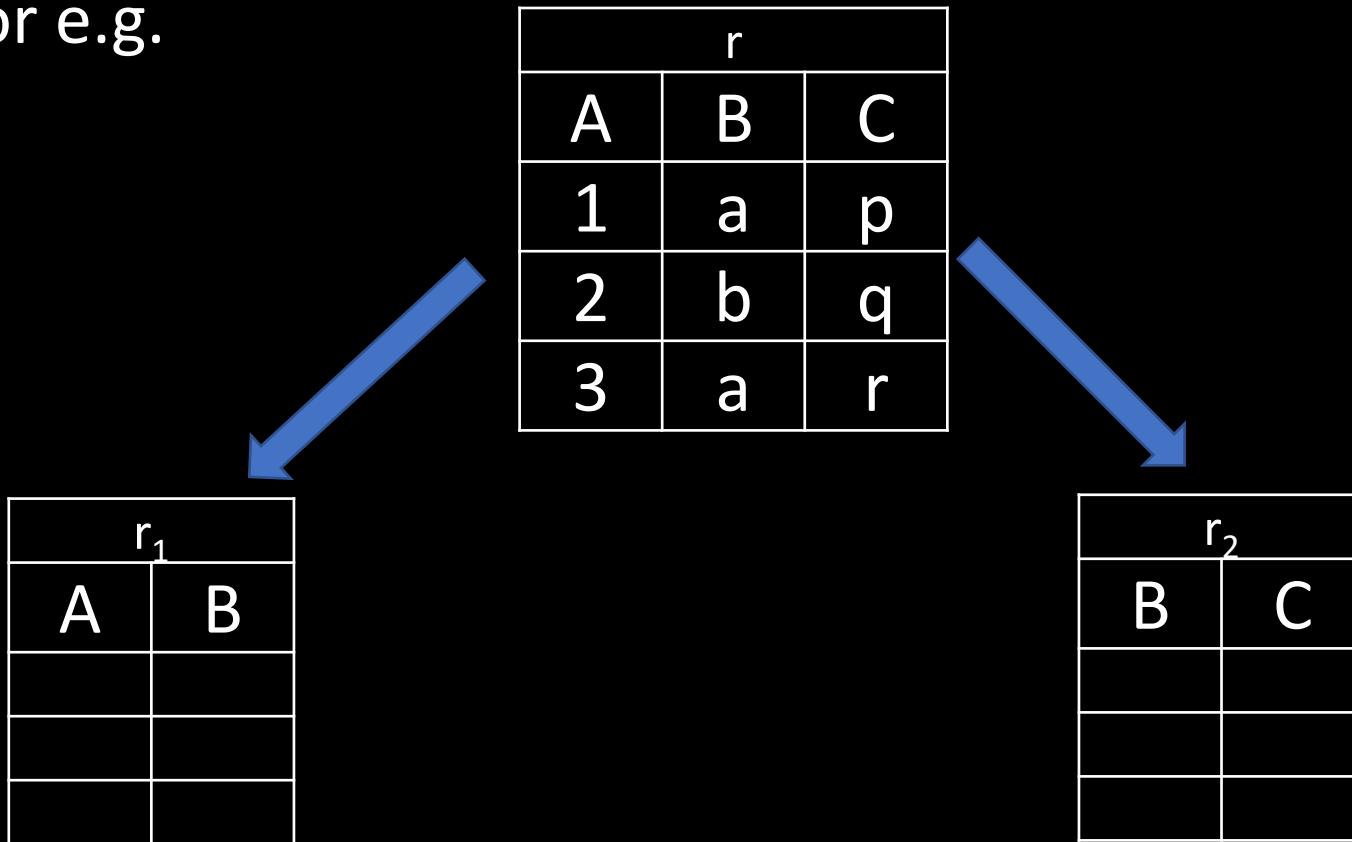
- A relation is in 4NF iff
  - It is in BCNF
  - There must not exist any non-trivial multivalued dependency.
- Each MVD is decomposed in separate table, where it becomes trivial MVD.

## Lossy/Lossless-Dependency Preserving Decomposition

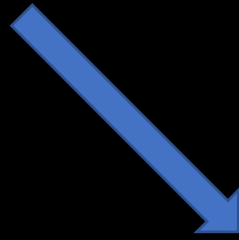
- Because of a normalization a table is Decomposed into two or more tables, but during this decomposition we must ensure satisfaction of some properties out of which the most important is lossless join property / decomposition.

- if we decompose a table  $r$  into two tables  $r_1$  and  $r_2$  because of normalization then at some later stage if we want to join (combine) (natural join) these tables  $r_1$  and  $r_2$ , then we must get back the original table  $r$ , without any extra or less tuple. But some information may be lost during retrieval of original relation or table.

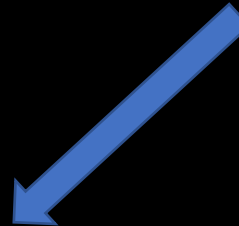
For e.g.



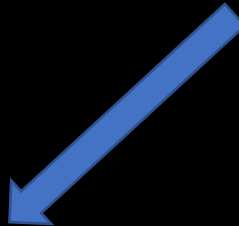
r		
A	B	C
1	a	p
2	b	q
3	a	r



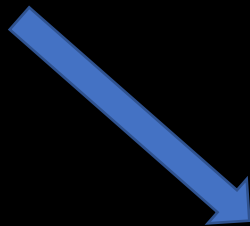
r <sub>2</sub>	
B	C
a	p
b	q
a	r



A	B	C



r <sub>1</sub>	
A	B
1	a
2	b
3	a





- Decomposition is lossy if  $R_1 \bowtie R_2 \supset R$
- Decomposition is lossy if  $R \supset R_1 \bowtie R_2$

- Decomposition is lossless if  $R_1 \bowtie R_2 = R$  "The decomposition of relation R into  $R_1$  and  $R_2$  is lossless when the join of  $R_1$  and  $R_2$  yield the same relation as in R." which guarantees that the spurious (extra or less) tuple generation problem does not occur with respect to the relation schemas created after decomposition.
- This property is extremely critical and must be achieved at any cost.
- lossless Decomposition / NonAdditive Join Decomposition

A	B	C	D	E
A	122	1	W	A
E	236	4	X	B
A	199	1	Y	C
B	213	2	Z	D

## How to check for lossless join decomposition using FD set, following conditions must hold:

- Union of Attributes of  $R_1$  and  $R_2$  must be equal to attribute of  $R$ . Each attribute of  $R$  must be either in  $R_1$  or in  $R_2$ .  $\text{Att}(R_1) \cup \text{Att}(R_2) = \text{Att}(R)$
- Intersection of Attributes of  $R_1$  and  $R_2$  must not be NULL.  $\text{Att}(R_1) \cap \text{Att}(R_2) \neq \Phi$
- Common attribute must be a key for at least one relation ( $R_1$  or  $R_2$ )
  - $\text{Att}(R_1) \cap \text{Att}(R_2) \rightarrow (R_1)$  or  $\text{Att}(R_1) \cap \text{Att}(R_2) \rightarrow (R_2)$

Q R (A, B, C, D)

$A \rightarrow B, B \rightarrow C, C \rightarrow D, D \rightarrow A$

$R_1(A, B), R_2(B, C)$  AND  $R_3(C, D)$

Q R(ABCDE)(NF)  $R_1(AB)$   $R_2(BC)$   $R_3(ABCD)$   $R_4(EG)$

$A \rightarrow BC$

$C \rightarrow DE$

$D \rightarrow E$

## 5 NF/Project-Join Normal Form

- A Relational table R is said to be in 5<sup>th</sup> normal form if
  - it is in 4 NF
  - it cannot be further non-loss decomposed

## Dependency Preserving Decomposition

- Let relation R be decomposed into Relations  $R_1, R_2, R_3, \dots, R_N$  with their respective functional Dependencies set as  $F_1, F_2, F_3, \dots, F_N$ , then the Decomposition is Dependency Preserving iff

- $\{F_1 \cup F_2 \cup F_3 \cup F_4 \dots \cup F_N\}^+ = F^+$

- Dependency preservation property, although desirable, is sometimes sacrificed.





# Indexing

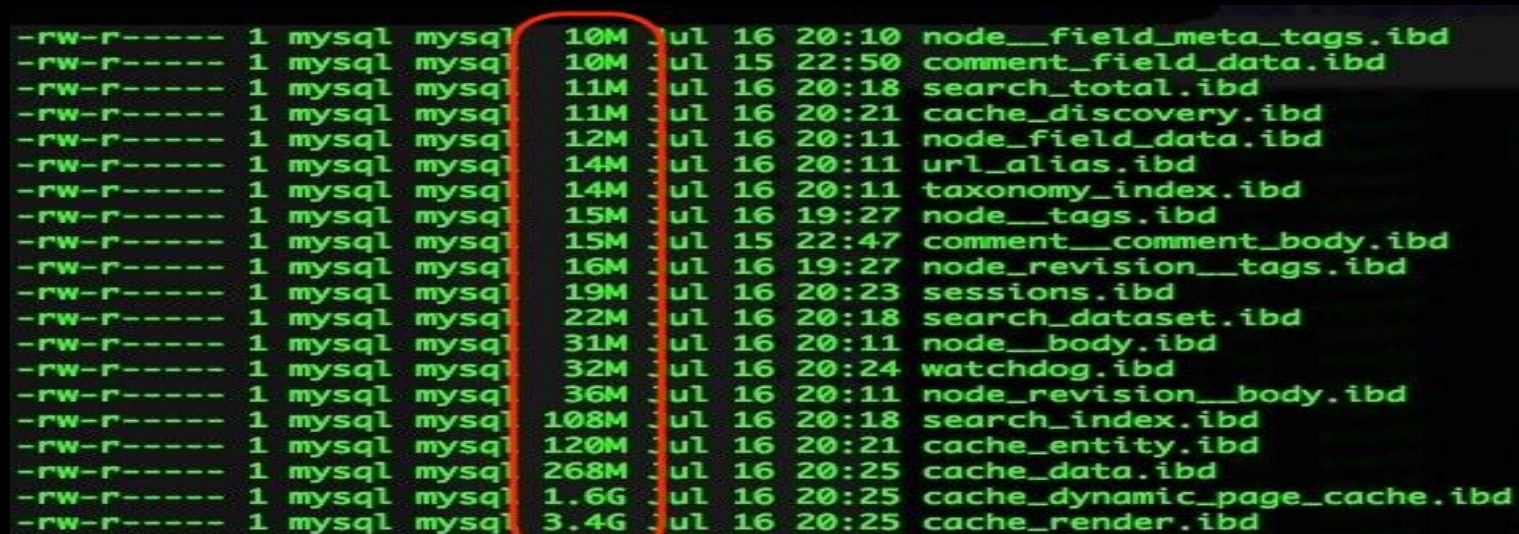
Relational databases are based on set theory.

- In set theory, the order of elements is unimportant, similarly in database tables.
- However, in practical implementation, element order in tables is often specified.
- Various operations such as search, insertion, and deletion are influenced by the order of elements in the tables.
- Elements in a table can be stored in two ways: sorted (ordered) or unsorted (unordered).

# File organization/ organization of records in a file

## Ordered file organization

- All the records in the file are ordered on some search key field.
- Here binary search is possible. (give example of book page searching)
- Maintenance (insertion & deletion) is costly, as it requires reorganization of entire file.
- Notes that we will get binary search only if we are using that key for searching on which indexing is done, otherwise it will behave as unsorted file.
- if file is unordered then no of block accesses required to reach correct block which contain the desired record is  $O(\log_2 n)$ , where n is the number of blocks.



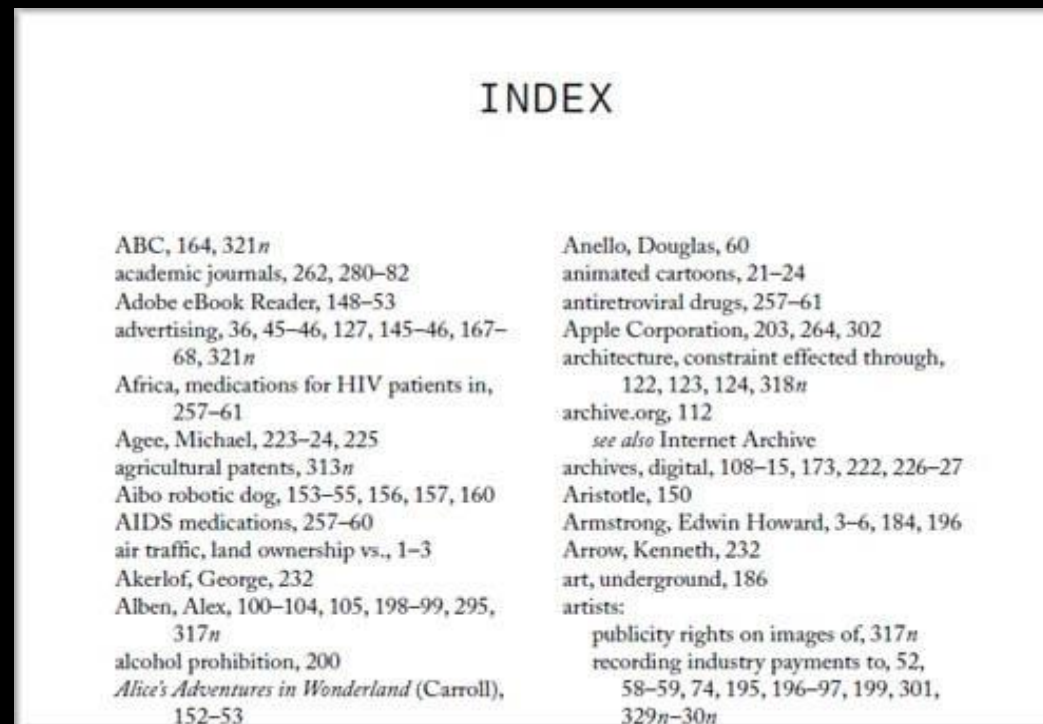
```
-rw-r----- 1 mysql mysql 10M ul 16 20:10 node__field_meta_tags.ibd
-rw-r----- 1 mysql mysql 10M ul 15 22:50 comment_field_data.ibd
-rw-r----- 1 mysql mysql 11M ul 16 20:18 search_total.ibd
-rw-r----- 1 mysql mysql 11M ul 16 20:21 cache_discovery.ibd
-rw-r----- 1 mysql mysql 12M ul 16 20:11 node_field_data.ibd
-rw-r----- 1 mysql mysql 14M ul 16 20:11 url_alias.ibd
-rw-r----- 1 mysql mysql 14M ul 16 20:11 taxonomy_index.ibd
-rw-r----- 1 mysql mysql 15M ul 16 19:27 node__tags.ibd
-rw-r----- 1 mysql mysql 15M ul 15 22:47 comment__comment_body.ibd
-rw-r----- 1 mysql mysql 16M ul 16 19:27 node_revision__tags.ibd
-rw-r----- 1 mysql mysql 19M ul 16 20:23 sessions.ibd
-rw-r----- 1 mysql mysql 22M ul 16 20:18 search_dataset.ibd
-rw-r----- 1 mysql mysql 31M ul 16 20:11 node__body.ibd
-rw-r----- 1 mysql mysql 32M ul 16 20:24 watchdog.ibd
-rw-r----- 1 mysql mysql 36M ul 16 20:11 node_revision__body.ibd
-rw-r----- 1 mysql mysql 108M ul 16 20:18 search_index.ibd
-rw-r----- 1 mysql mysql 120M ul 16 20:21 cache_entity.ibd
-rw-r----- 1 mysql mysql 268M ul 16 20:25 cache_data.ibd
-rw-r----- 1 mysql mysql 1.6G ul 16 20:25 cache_dynamic_page_cache.ibd
-rw-r----- 1 mysql mysql 3.4G ul 16 20:25 cache_render.ibd
```

# Unordered file organization

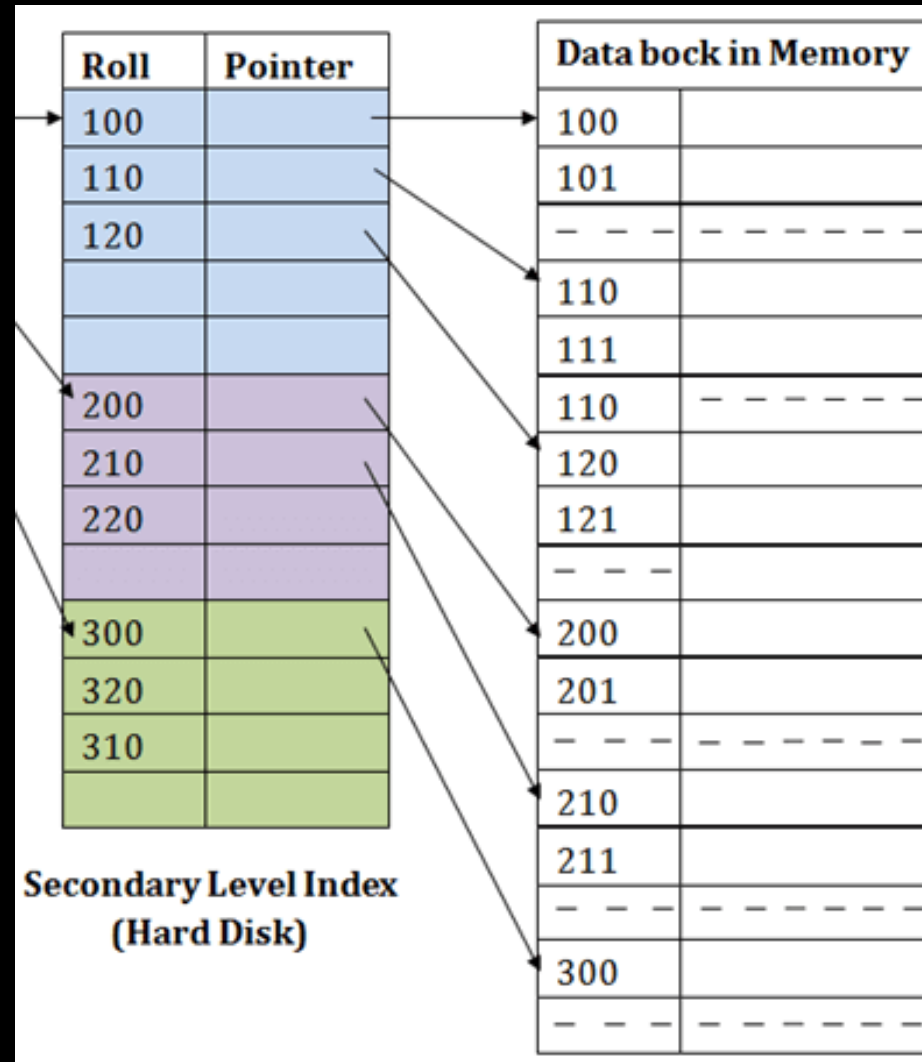
- Records are typically added at the end of the file, without following any specific order.
- This insertion method allows only linear search, resulting in slower search times.
- Despite slow searches, maintenance including insertion and deletion is simpler.
- No reorganization of the entire file is needed, making maintenance easier.
- If file is unordered then no of block accesses required to reach correct block which contain the desired record is  $O(n)$ , where  $n$  is the number of blocks.

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 2	A-215	Mianus	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

- Indexes are supplementary structures in databases, aiding in swift record retrieval.
- They enable quick data access based on particular attributes identified for indexing.
- This technique is similar to the index sections seen in books.
- Indexes provide secondary pathways to access records without changing their physical position in the main file.



- The size of index file is way smaller than that of the main file, as index file record contain only two columns key (attribute in which searching is done) and block pointer (base address of the block of main file which contains the record holding the key), while main file contains all the columns.

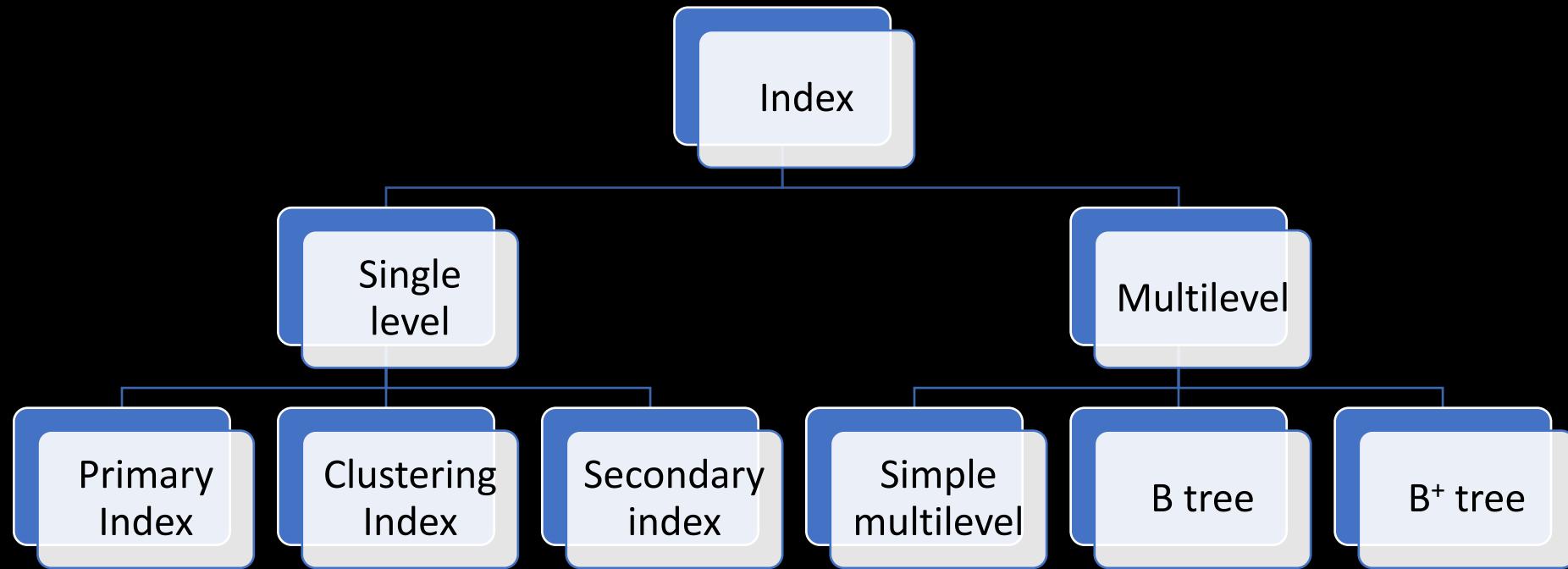


Q Suppose we have ordered file with records stored  $r = 30,000$  on a disk with Block Size  $B = 1024$  B. File records are of fixed size and are unspanned with record length  $R = 100$  B. Suppose that ordering key field of file is 9 B long and a block pointer is 6 B long, Implement primary indexing?

1. Indexes can be established on any relation field, be it primary key or non-key.
2. Each attribute can have a dedicated index file, meaning multiple index files may exist for one main file.
3. Index files are always organized, allowing for the utilization of binary search advantages, irrespective of the main file's order.
4. Indexing accelerates data retrieval time but also introduces space overhead for storing the index file.
5. The correct block in the main file can be located with  $\log_2(\text{number of blocks in index file}) + 1$  accesses.



# TYPES OF INDEXING



- In single-level indexing, an index file is created for the main file, marking the end of the indexing process.
- Multiple-level indexing, on the other hand, involves creating an index for the index file and continually repeating this procedure until only a single block remains.

## PRIMARY INDEXING

- Main file is always sorted according to primary key.
- Indexing is done on Primary Key, therefore called as primary indexing
- Index file have two columns, first primary key and second anchor pointer (base address of block)
- It is an example of Sparse Indexing.
- Here first record (anchor record) of every block gets an entry in the index file
- No. of entries in the index file = No of blocks acquired by the main file.

# CLUSTERED INDEXING

- Main file will be ordered on some non-key attributes
- No of entries in the index file = no of unique values of the attribute on which indexing is done.
- It is the example of Sparse as well as dense indexing

**Q** Suppose we have ordered file with records stored  $r = 30,000$  on a disk with Block Size  $B = 1024$  B. File records are of fixed size and are unspanned with record length  $R = 100$  B. Suppose that ordering key field of file is 9 B long and a block pointer is 6 B long, Implement Secondary indexing?

## SECONDARY INDEXING

- Most common scenarios, suppose that we already have a primary indexing on primary key, but there is frequent query on some other attributes, so we may decide to have one more index file with some other attribute.
- Main file is ordered according to the attribute on which indexing is done(unordered).
- Secondary indexing can be done on key or non-key attribute.
- No of entries in the index file is same as the number of entries in the main file.
- It is an example of dense indexing.

## Dense Vs Sparse

- Dense Index In dense index, there is an entry in the index file for every search key value in the main file. This makes searching faster but requires more space to store index records itself. Note that it is not for every record, it is for every search key value. Sometime number of records in the main file  $>$  number of search keys in the main file, for example if search key is repeated.
- Sparse Index-If an index entry is created only for some records of the main file, then it is called sparse index. No. of index entries in the index file  $<$  No. of records in the main file. Note: - dense and sparse are not complementary to each other, sometimes it is possible that a record is both dense and sparse.

## B tree

- A B-tree of order  $m$  if non-empty is an  $m$ -way search tree in which.
  - The root has at least zero child nodes and at most  $m$  child nodes.
  - The internal nodes except the root have at least  $\lceil m/2 \rceil$  child nodes and at most  $m$  child nodes.
  - The number of keys in each internal node is one less than the number of child nodes and these keys partition the subtrees of the nodes in a manner similar to that of  $m$ -way search tree.
  - All leaf nodes are on the same level(perfectly balanced).

### Root

Rules	MAX	MIN
CHILD	$m$	0
DATA	$m-1$	1

### Internal except Root

Rules	MAX	MIN
CHILD	$m$	$\lceil m/2 \rceil$
DATA	$m-1$	$\lceil m/2 \rceil - 1$

### Leaf

Rules	MAX	MIN
CHILD	0	0
DATA	$m-1$	$\lceil m/2 \rceil - 1$

## Insertion in B-TREE

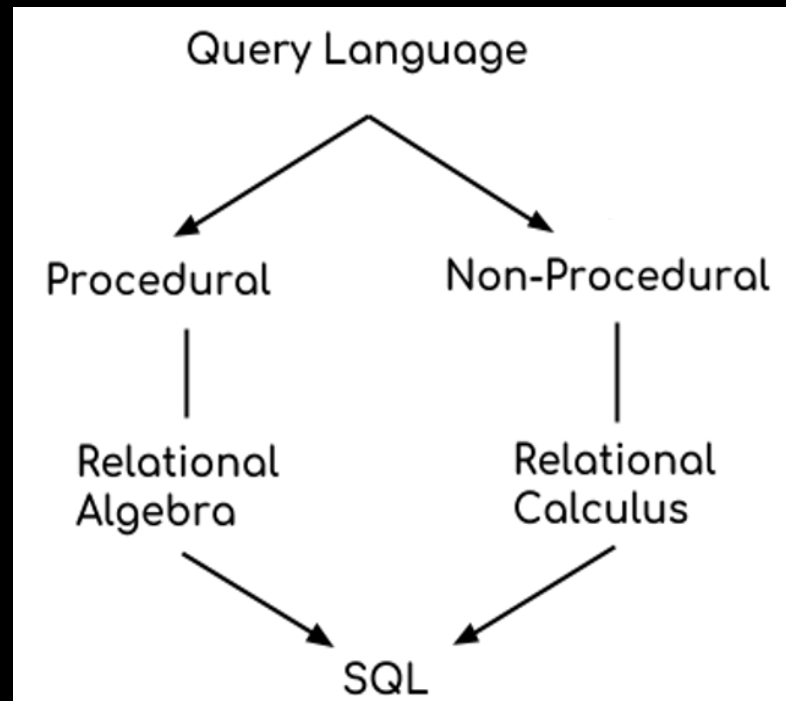
- A B-tree starts with a single root node (which is also a leaf node) at level 0 (zero). Once the root node is full with  $m - 1$  search key values and we attempt to insert another entry in the tree, the root node splits into two nodes at level 1.
- Only the middle value is kept in the root node, and the rest of the values are split evenly between the other two nodes. When a non-root node is full and a new entry is inserted into it, that node is split into two nodes at the same level, and the middle entry is moved to the parent node along with two pointers to the new split nodes.
- If the parent node is full, it is also split. Splitting can propagate all the way to the root node, creating a new level if the root is split.



**Q** Consider the following elements 5, 10, 12, 13, 14, 1, 2, 3, 4  
insert them into an empty b-tree of order = 3.

# Query Language

- After designing a data base, that is ER diagram followed by conversion in relational model followed by normalization and indexing, now next task is how to store, retrieve and modify data in the data base.
- So here we will be concentrating more on the retrieval part. Query languages are used for this purpose. **Query languages, data query languages or database query languages (DQLs)** are computer languages using which user request some information from the database. A well known example is the **Structured Query Language (SQL)**.



# Procedural Query Language

- Here users instruct the system to performs a sequence of operations on the data base in order to compute the desired result.
- Means user provides both what data to be retrieved and how data to be retrieved. e.g. Relational Algebra.

## Non-Procedural Query Language

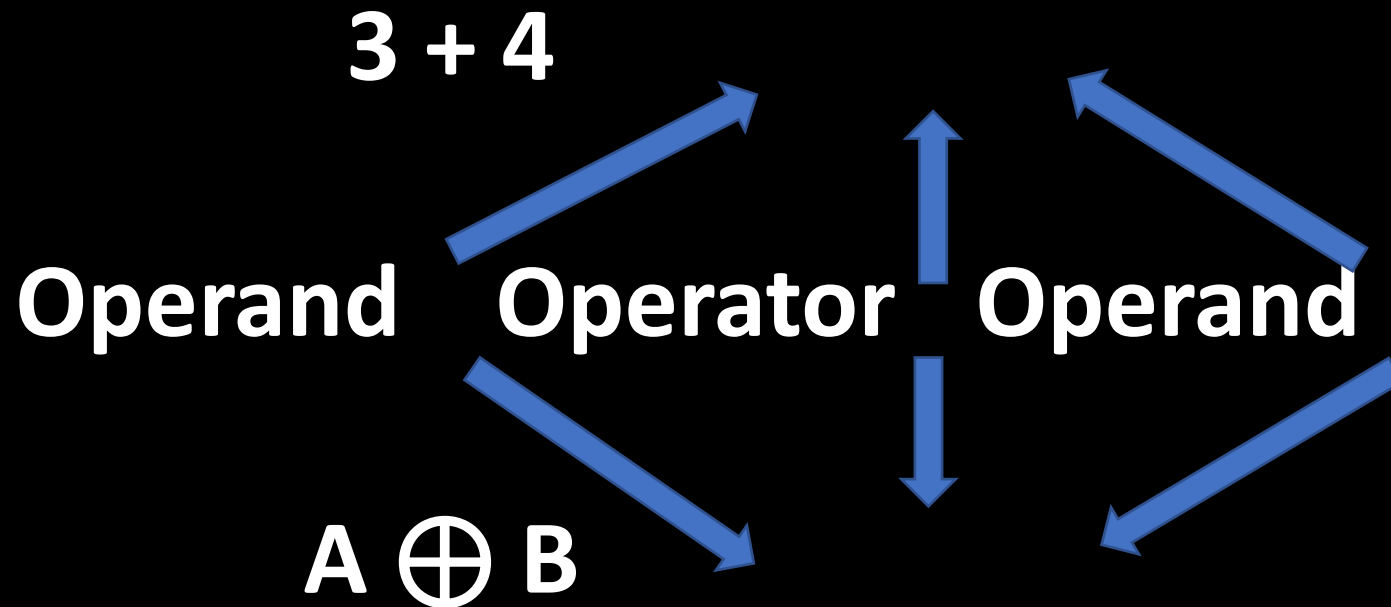
- In nonprocedural language, the user describes the desired information without giving a specific procedure for obtaining that information.
- What data to be retrieved e.g. Relational Calculus. **Tuple relational calculus, Domain relational calculus** are declarative query languages based on mathematical logic

- Relational Algebra (Procedural) and Relational Calculus (non-procedural) are mathematical system/ query languages which are used for query on relational model.
- RA and RC are not executed in any computer they provide the fundamental mathematics on which SQL is based.
- SQL (structured query language) works on RDBMS, and it includes elements of both procedural or non-procedural query language.

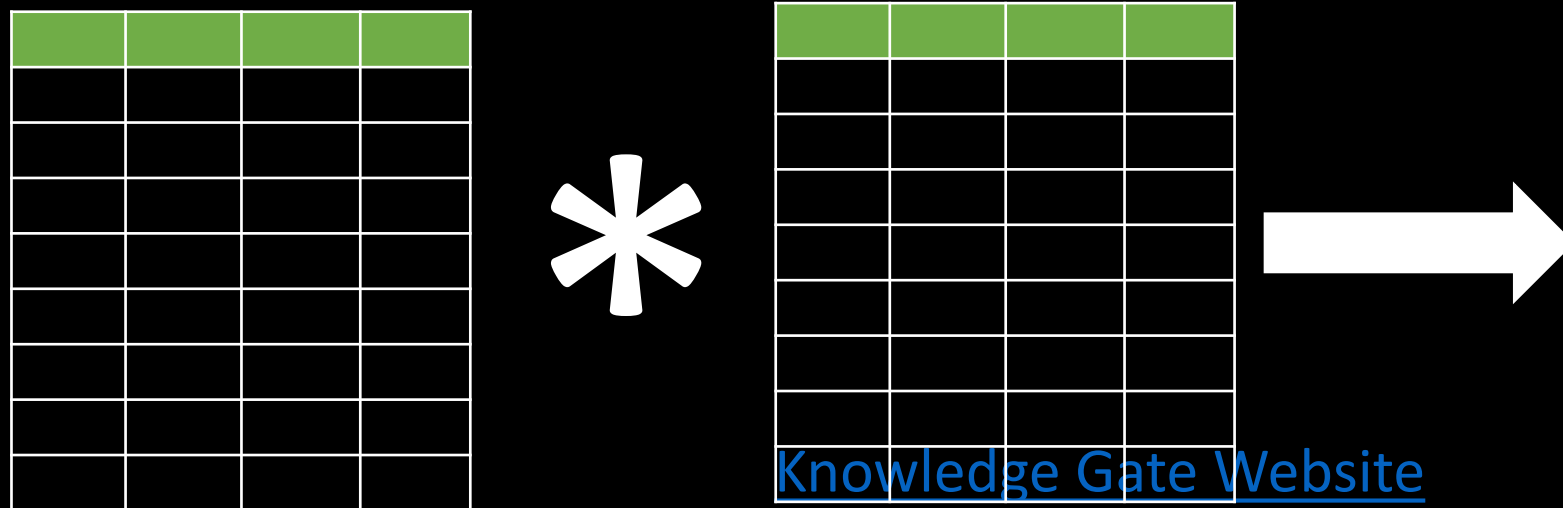
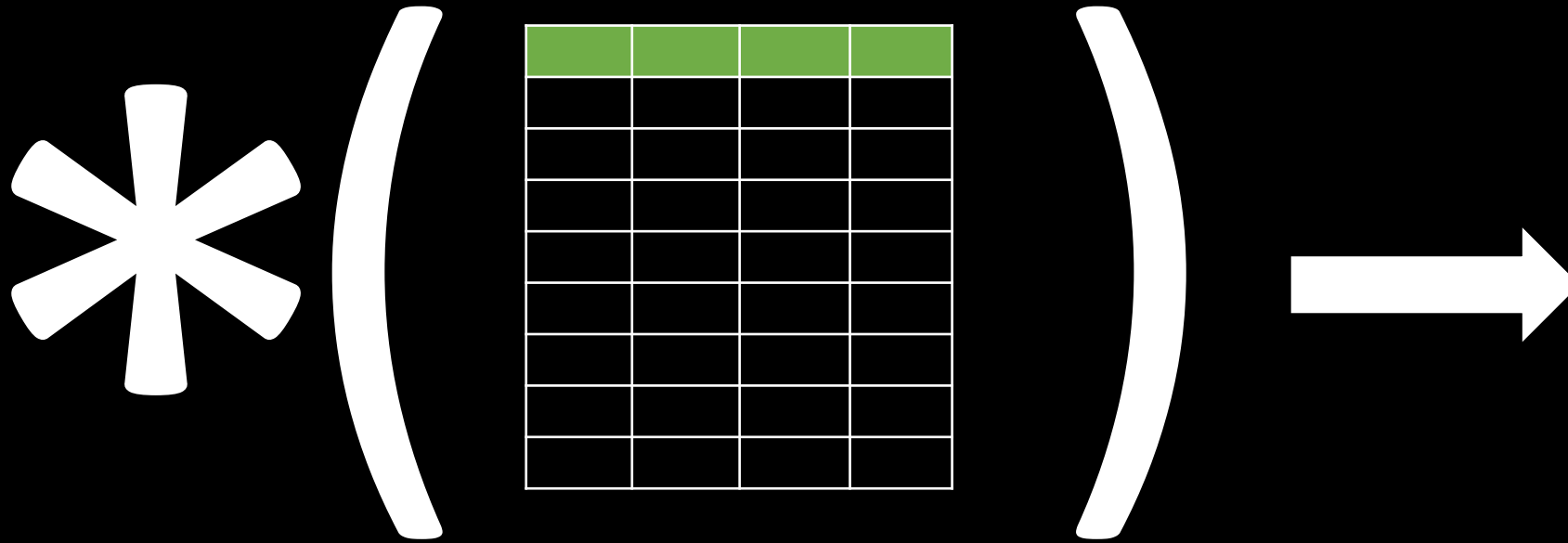
<u>Relational model</u>	<u>RDBMS</u>
RA, RC	SQL
Algo	Code
Conceptual	Reality
Theoretical	Practical

# RELATIONAL ALGEBRA

- RA like any other mathematical system provides a number of operators and use relations (tables) as operands and produce a new relation as their result.



- Every operator in the RA accepts (one or two) relation/table as input arguments and returns always a single relation instance as the result without a name.



- It also does not consider duplicity by default as it is based on set theory. Same query is written in RA and SQL the result may be different as SQL considers duplication.
- As it is pure mathematics no use of English keywords. Operators are represented using symbols.



## BASIC / FUNDAMENTAL OPERATORS

- The fundamental operations in the relational algebra are **select**, **project**, **union**, **set difference**, **Cartesian product**, and **Rename**.

Name	Symbol
Select	$(\sigma)$
Project	$(\Pi)$
Union	$(\cup)$
Set difference	$(-)$
Cross product	$(X)$
Rename	$(\rho)$

## DERIVED OPERATORS

- There are several other operations namely: **set intersection, natural join, and assignment.**

Name	Symbol	DERIVED FROM
Join	$(\bowtie)$	$(X)$
Intersection	$(\cap)$	$(-)$ $A \cap B = A - (A - B)$
Division	$(\div)$	$(X, -, \Pi)$
Assignment	$(=)$	

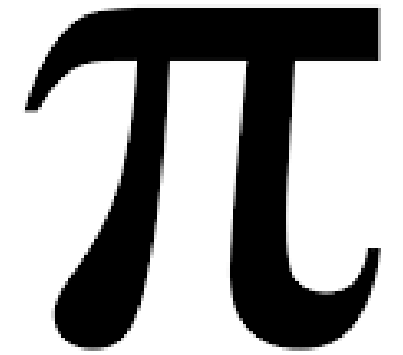
- The **select, project, and rename** operations are called **unary operations**, because they operate on one relation.
- **Union, Cartesian product and set difference** operate on pairs of relations and are, therefore, called **binary operations**.
- Relational algebra also provides the framework for query optimization.

**Relational schema** - A relation schema R, denoted by R ( $A_1, A_2, \dots, A_n$ ), is made up of a relation name R and a list of attributes,  $A_1, A_2, \dots, A_n$ . Each **attribute**  $A_i$  is the name of a role played by some domain D in the relation schema R. It is use to describe a Relation.

E.g. Schema representation of Table **Student** is as –  
**STUDENT (NAME, ID, CITY, COUNTRY, HOBBY).**

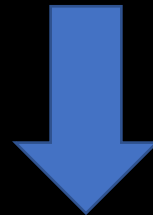
**Relational Instance** - Relations with its data at particular instant of time.

# The Project Operation (Vertical Selection)



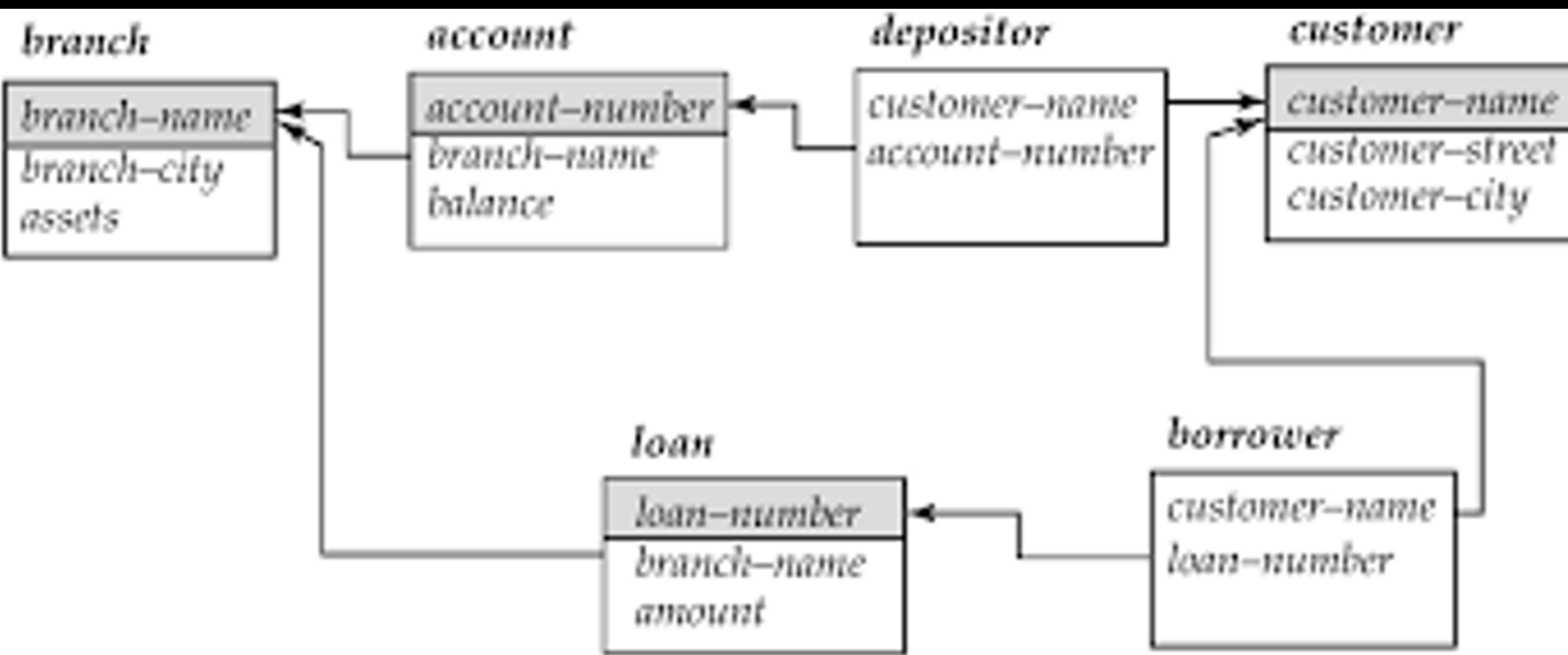
- Main idea behind project operator is to select desired columns
- The project operation is a unary operation that returns its argument relation, with certain attributes left out.
- Projection is denoted by the uppercase Greek letter pi ( $\Pi$ ). Maximum columns selected can be 1, Maximum selected Columns can be  $n - 1$ .

•  $\Pi_{\text{column\_name}}(\text{table\_name})$



Q Write a RELATIONAL ALGEBRA query to find the name of all customer without duplication having bank account?

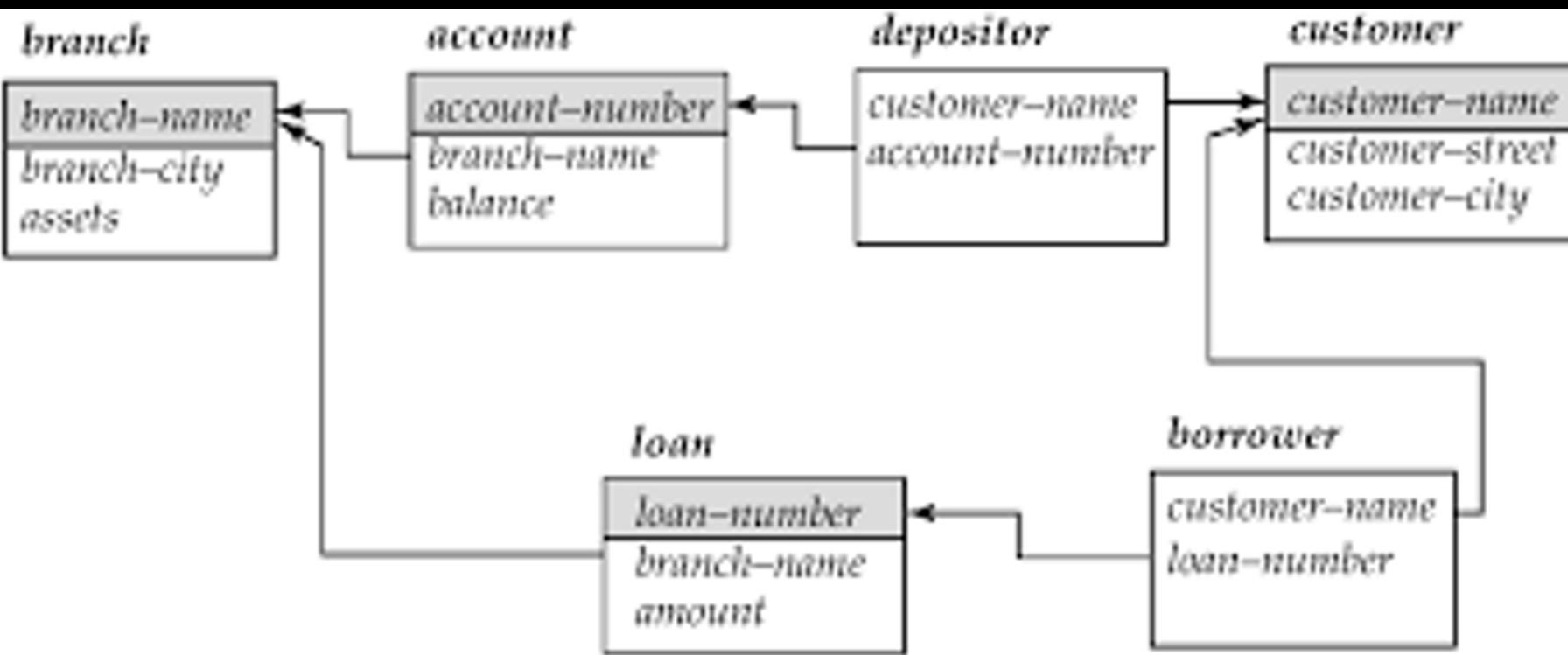
Q Write a RELATIONAL ALGEBRA query to find all the details of bank branches?



Q Write a RELATIONAL ALGEBRA query to find the name of all customer without duplication having bank account?

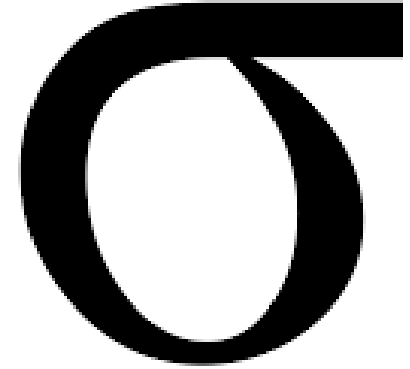
$\Pi_{\text{customer\_name}}(\text{depositor})$

Q Write a RELATIONAL ALGEBRA query to find all the details of bank branches?  
**(branch)**



## The Select Operation (Horizontal Selection)

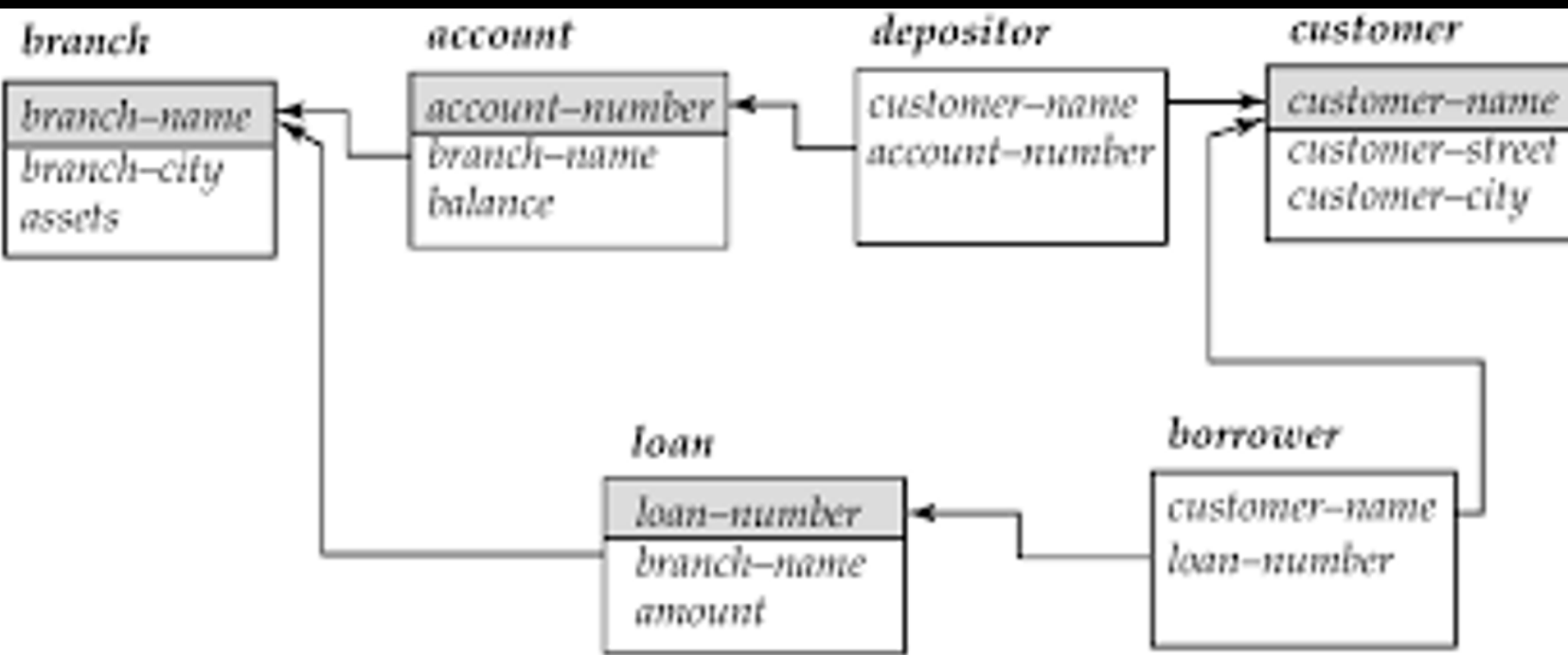
- The select operation selects tuples that satisfy a given predicate/Condition p.
- Lowercase Greek letter sigma ( $\sigma$ ) is used to denote selection.
- It is a unary operator.
- Eliminates only tuples/rows.
- $\sigma_{\text{condition}}(\text{table\_name})$





Q Write a RELATIONAL ALGEBRA query to find all account\_no where balance is less the 1000?

Q Write a RELATIONAL ALGEBRA query to find branch name which is situated in Delhi and having assets less than 1,00,000?

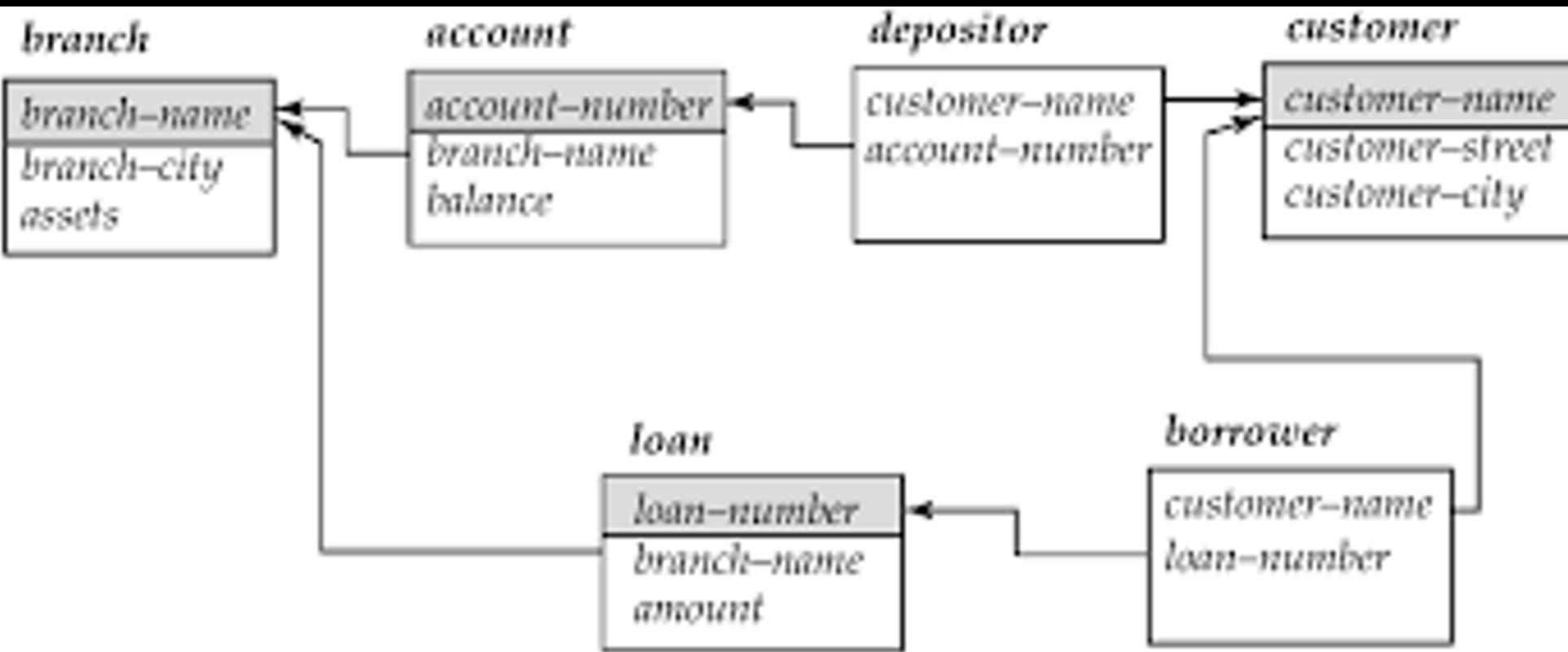


Q Write a RELATIONAL ALGEBRA query to find all account\_no where balance is less the 1000?

$\Pi_{\text{account\_number}}(\sigma_{\text{balance} < 1000}(\text{account}))$

Q Write a RELATIONAL ALGEBRA query to find branch name which is situated in Delhi and having assets less than 1,00,000?

$\Pi_{\text{branch\_name}}(\sigma_{(\text{branch\_city} = \text{delhi}) \wedge (\text{assets} < 1000)}(\text{branch}))$

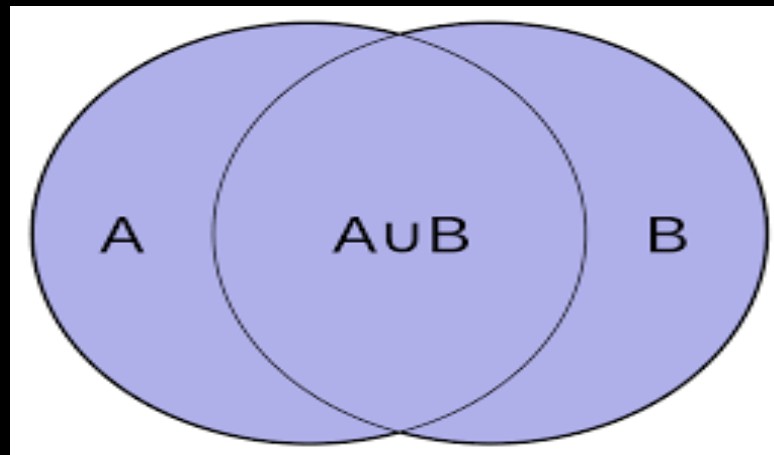
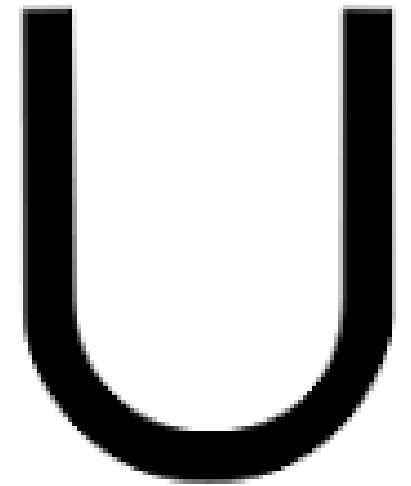


- Commutative in Nature,  $\sigma_{p1 \wedge p2}(r) = \sigma_{p2 \wedge p1}(r) = \sigma_{p1}(\sigma_{p2}(r)) = \sigma_{p2}(\sigma_{p1}(r))$
- We allow comparisons using  $=$ ,  $\neq$ ,  $<$ ,  $>$ ,  $\leq$  and  $\geq$  in the selection predicate.
- Using the connectives and ( $\wedge$ ), or ( $\vee$ ), and not ( $\neg$ ), we can combine several predicates into a larger predicate.
- Minimum number of tuples selected can be 0, Maximum selected tuples can be all.



## The Union Operation

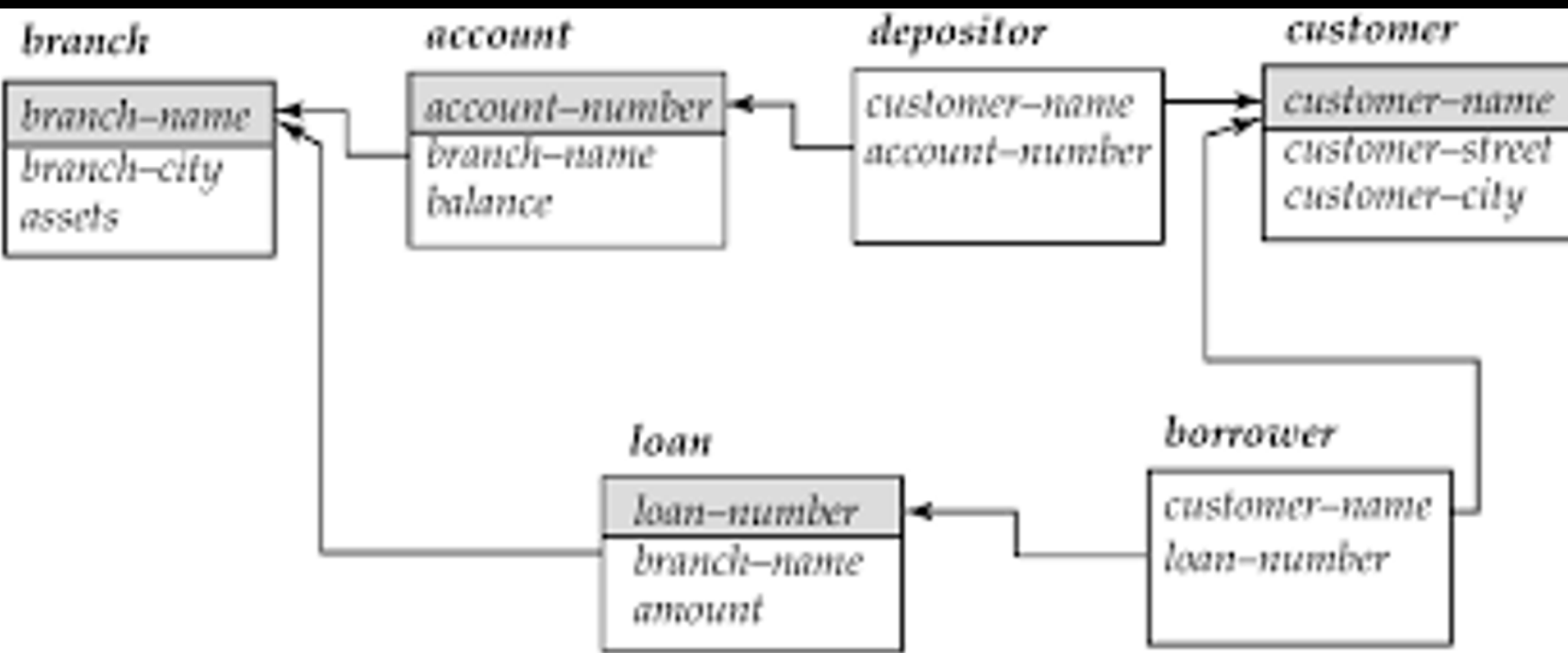
- It is a binary operation, denoted, as in set theory, by  $\cup$ .
- Written as,  $\text{Expression}_1 \cup \text{Expression}_2$ ,  $r \cup s = \{t \mid t \in r \text{ or } t \in s\}$
- For a union operation  $r \cup s$  to be valid, we require that two conditions are satisfied:
  - The relations  $r$  and  $s$  must be of the same arity. That is, they must have the same number of attributes.
  - The domains of the  $i^{\text{th}}$  attribute of  $r$  and the  $i^{\text{th}}$  attribute of  $s$  must be the same, for all  $i$ .



- Some points to remember
  - $\text{Deg}(R \cup S) = \text{Deg}(R) = \text{Deg}(S)$
  - $\text{Max}(IRI, ISI) \leq IRUSI \leq (IRI+ISI)$

**Q Write a RELATIONAL ALGEBRA query to find all the customer name who have a loan or an account or both?**

**Q Write a RELATIONAL ALGEBRA query to find all the customer name who have a loan but do not have an account?**

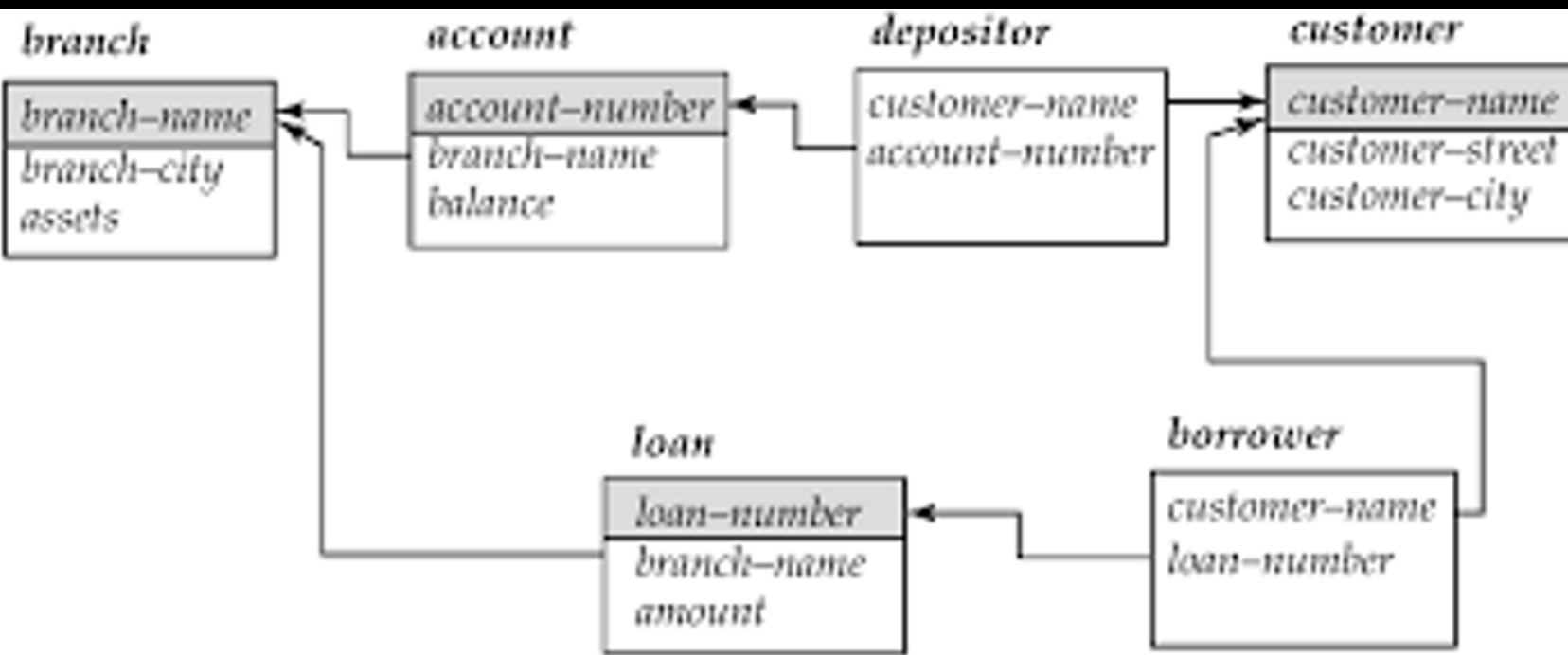


Q Write a RELATIONAL ALGEBRA query to find all the customer name who have a loan or an account or both?

$\Pi_{\text{customer\_name}}(\text{depositor}) \cup \Pi_{\text{customer\_name}}(\text{borrower})$

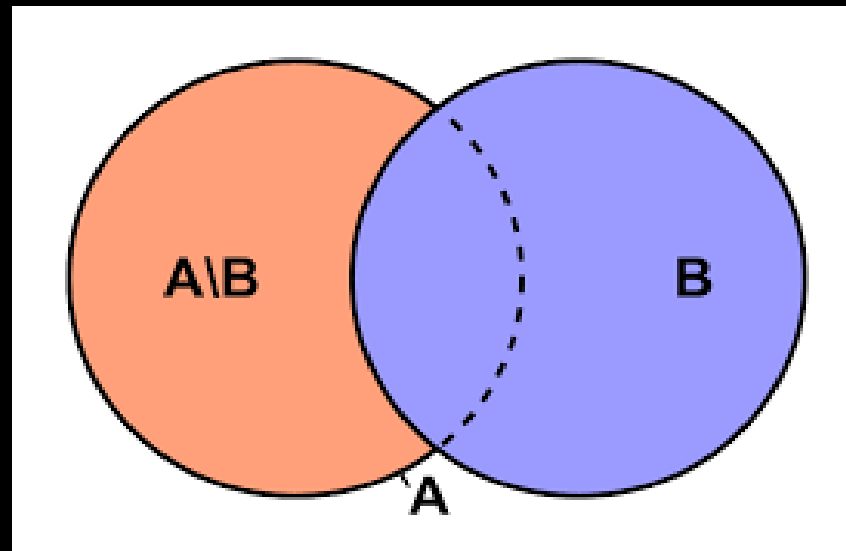
Q Write a RELATIONAL ALGEBRA query to find all the customer name who have a loan but do not have an account?

$\Pi_{\text{customer\_name}}(\text{borrower}) - (\Pi_{\text{customer\_name}}(\text{depositor}))$



## The Set-Difference Operation

- The set-difference operation, denoted by  $-$ , allows us to find tuples that are in one relation but are not in another. It is a binary operator.
- The expression  $r - s$  produces a relation containing those tuples in  $r$  but not in  $s$ .
- We must ensure that set differences are taken between compatible relations.
- For a set-difference operation  $r - s$  to be valid, we require that the relations  $r$  and  $s$  be of the same arity, and that the domains of the  $i$ th attribute of  $r$  and the  $i$ th attribute of  $s$  be the same, for all  $i$ .
  - $0 \leq |R - S| \leq |R|$





# The Cartesian-Product Operation

- The Cartesian-product operation, denoted by a cross ( $\times$ ), allows us to combine information from any two relations.
- It is a binary operator; we write the Cartesian product of relations  $R_1$  and  $R_2$  as  $R_1 \times R_2$ .
- Cartesian-product operation associates every tuple of  $R_1$  with every tuple of  $R_2$ .
  - $R_1 \times R_2 = \{rs \mid r \in R_1 \text{ and } s \in R_2\}$ , contains one tuple  $\langle r, s \rangle$  (concatenation of tuples  $r$  and  $s$ ) for each pair of tuples  $r \in R_1, s \in R_2$ .

$R_1$	
A	B
1	P
2	Q
3	R

$R_2$	
B	C
Q	X
R	Y
S	Z

$R_1 \times R_2$			
A	$R_1.B$	$R_2.B$	C

# The Cartesian-Product Operation

- The Cartesian-product operation, denoted by a cross ( $\times$ ), allows us to combine information from any two relations.
- It is a binary operator; we write the Cartesian product of relations  $R_1$  and  $R_2$  as  $R_1 \times R_2$ .
- Cartesian-product operation associates every tuple of  $R_1$  with every tuple of  $R_2$ .
  - $R_1 \times R_2 = \{rs \mid r \in R_1 \text{ and } s \in R_2\}$ , contains one tuple  $\langle r, s \rangle$  (concatenation of tuples  $r$  and  $s$ ) for each pair of tuples  $r \in R_1, s \in R_2$ .

$R_1$	
A	B
1	P
2	Q
3	R

$R_2$	
B	C
Q	X
R	Y
S	Z

$R_1 \times R_2$			
A	$R_1.B$	$R_2.B$	C
1	P	Q	X
1	P	R	Y
1	P	S	Z
2	Q	Q	X
2	Q	R	Y
2	Q	S	Z
3	R	Q	X
3	R	R	Y
3	R	S	Z

# The Cartesian-Product Operation

- The Cartesian-product operation, denoted by a cross ( $\times$ ), allows us to combine information from any two relations.
- It is a binary operator; we write the Cartesian product of relations  $R_1$  and  $R_2$  as  $R_1 \times R_2$ .
- Cartesian-product operation associates every tuple of  $R_1$  with every tuple of  $R_2$ .
  - $R_1 \times R_2 = \{rs \mid r \in R_1 \text{ and } s \in R_2\}$ , contains one tuple  $\langle r, s \rangle$  (concatenation of tuples  $r$  and  $s$ ) for each pair of tuples  $r \in R_1, s \in R_2$ .

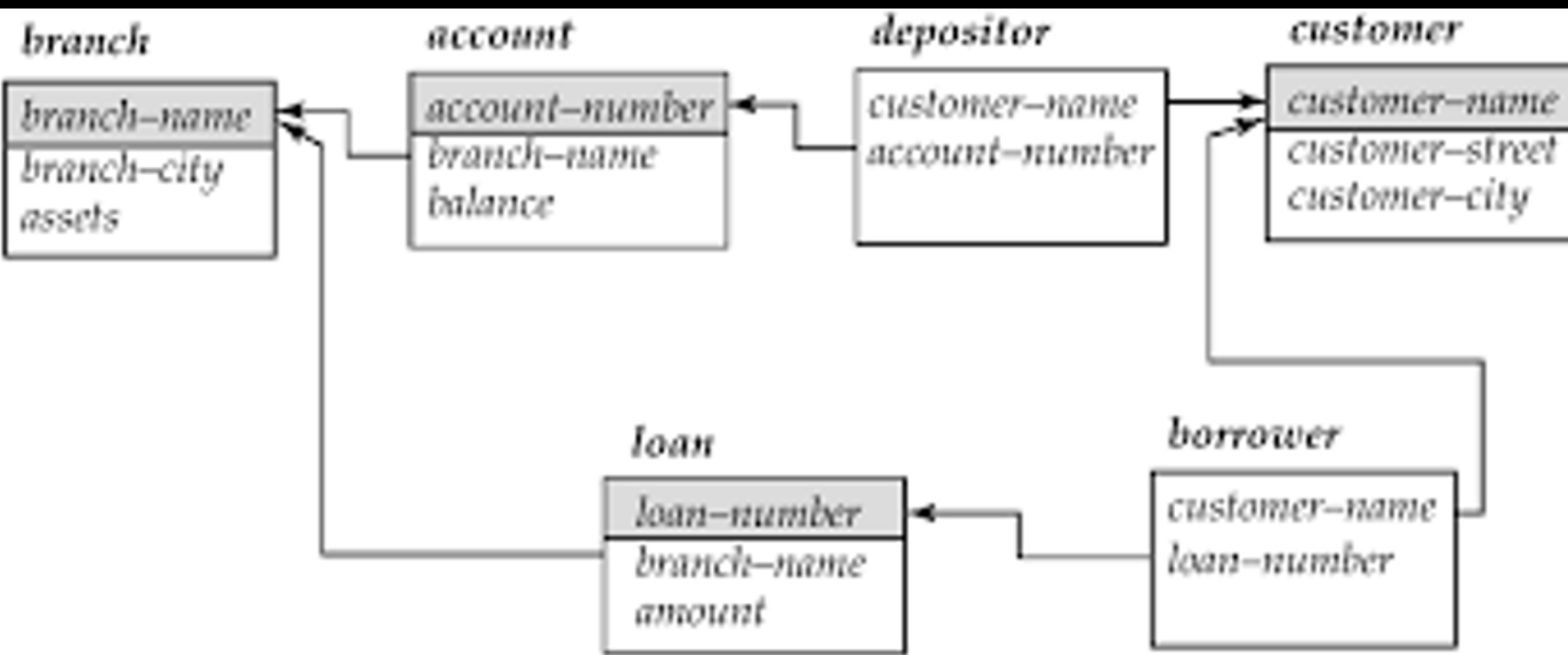
$R_1$	
A	B
1	P
2	Q
3	R

$R_2$	
B	C
Q	X
R	Y
S	Z

$R_1 \times R_2$			
A	$R_1.B$	$R_2.B$	C
1	P	Q	X
1	P	R	Y
1	P	S	Z
2	Q	Q	X
2	Q	R	Y
2	Q	S	Z
3	R	Q	X
3	R	R	Y
3	R	S	Z

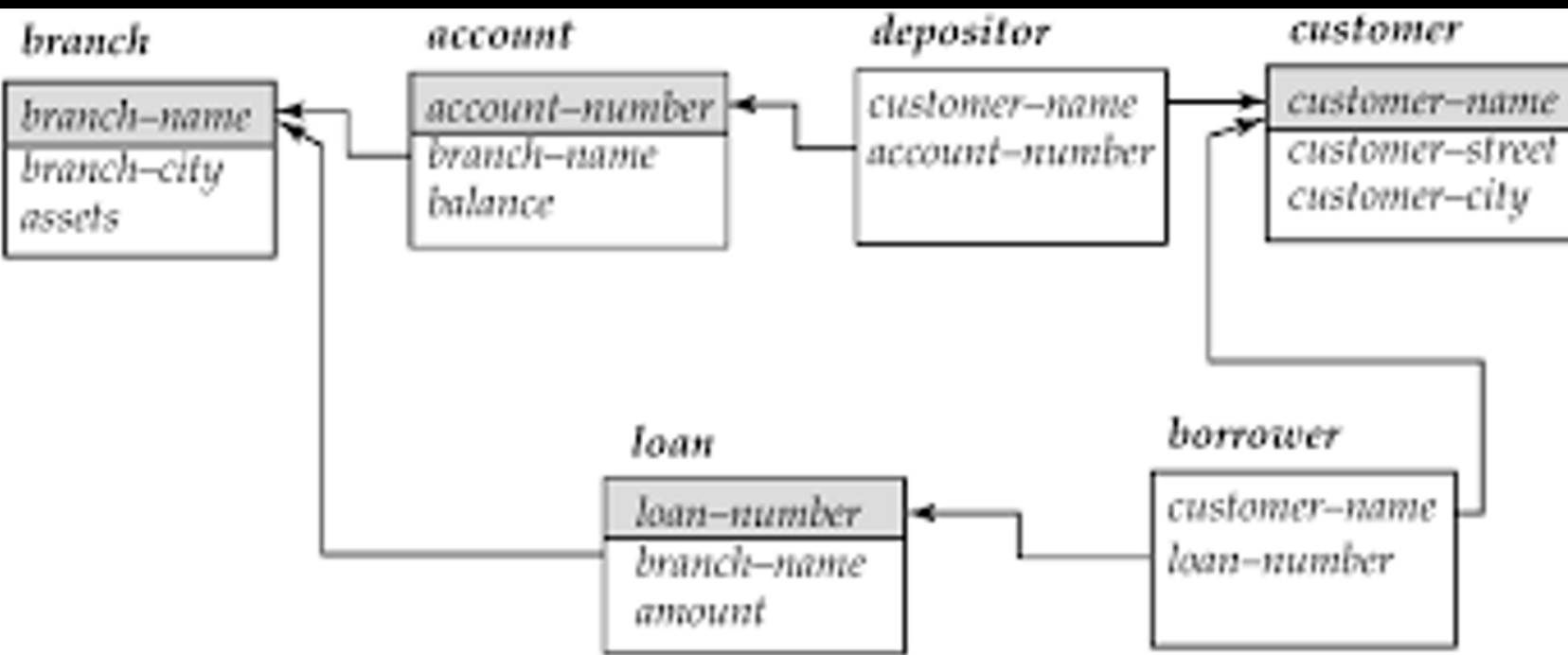
- $R_1 \times R_2$  returns a relational instance whose schema contains all the fields of  $R_1$  (in order as they appear in  $R_1$ ) and all fields of  $R_2$  (in order as they appear in  $R_2$ ).
- If  $R_1$  has  $m$  tuples and  $R_2$  has  $n$  tuples the result will be having  $= m * n$  tuples.
- Same attribute name may appear in both  $R_1$  and  $R_2$ , we need to devise a naming schema to distinguish between these attributes.

**Q Write a RELATIONAL ALGEBRA query to find the name of all the customers along with account balance, who have an account in the bank?**



Q Write a RELATIONAL ALGEBRA query to find the name of all the customers along with account balance, who have an account in the bank?

$\Pi_{\text{customer\_name, balance}} (\sigma_{\text{account.account\_number=depositor.account\_number}} (\text{account X depositor}))$



# Rename Operation

- The results of relational algebra are also relations but without any name. This Query do not change the name of the table in the original data base, but create a new copy of the table.
- The rename operation allows us to rename the output relation. It is denoted with small Greek letter rho  $\rho$ . Where the result of expression  $E$  is saved with name of  $x$ .
- $\rho_{x(A1, A2, A3, A4, \dots, AN)}(E)$
- $\rho_{Learner}(\text{Student})$
- $\rho_{Learner(\text{Stu\_ID, User\_Name, Age})}(\text{Student}(\text{Roll\_No, Name, Age}))$

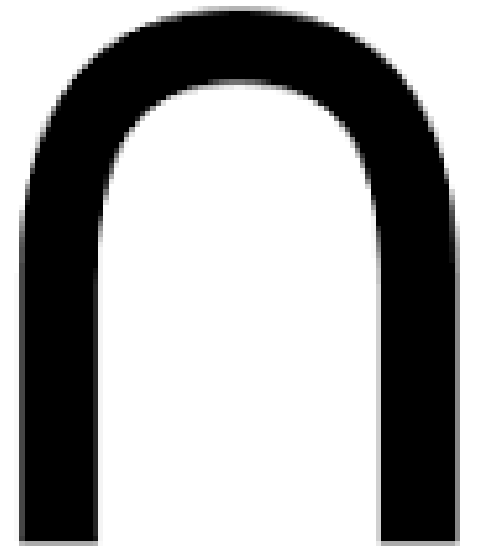
## Additional/Derived Relational-Algebra Operations

- If we restrict ourselves to just the fundamental operations, certain common queries are lengthy to express. Therefore, we use additional operations.
- These additional operations do not add any power to the algebra.
- They are used to simplify the queries.



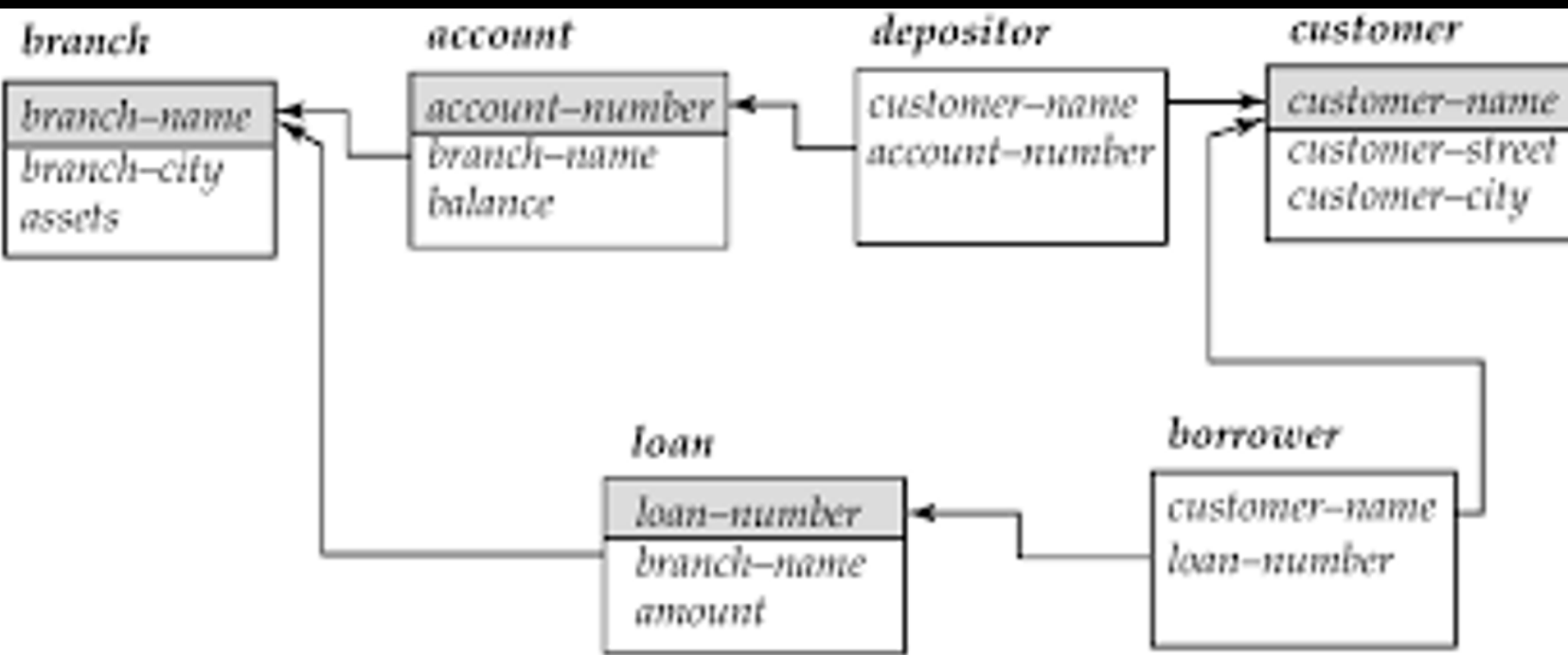
## Set-Intersection Operator

- We will be using  $\cap$  symbol to denote set intersection.
- $r \cap s = r - (r - s)$
- Set intersection is not a fundamental operation in relational algebra.
- $r \cap s = \{t \mid t \in r \text{ and } t \in s\}$
- $0 \leq |R \cap S| \leq \min(|R|, |S|)$



Q Write a RELATIONAL ALGEBRA query to find all the customer name who have both a loan and an account?

$\Pi_{\text{customer\_name}}(\text{depositor}) \cup \Pi_{\text{customer\_name}}(\text{borrower})$




# The Natural-Join Operation \*

- The natural join is a binary operation that allows us to combine certain selections and a Cartesian product into one operation.

The natural join is a Lossy operator.

R <sub>1</sub>	
A	B
1	P
2	Q
3	R

R <sub>2</sub>	
B	C
Q	X
R	Y
S	Z

R <sub>1</sub>		R <sub>2</sub>
A	B	C
1	Q	X
2	Q	X
3	R	Y

- In general, the DIVISION operation is applied when we have query like student who have completed both database1 and data base2 tasks.

<i>Completed</i>		<i>DBProject</i>
<b>Student</b>	<b>Task</b>	<b>Task</b>
Fred	Database1	Database1
Fred	Database2	Database2
Fred	Compiler1	
Eugene	Database1	
Eugene	Compiler1	
Sarah	Database1	
Sarah	Database2	

$$\pi_{\text{Student}}(R) - \{\pi_{\text{Student}}[(\pi_{\text{Student}}(R) \times S) - \pi_{\text{Student,Task}}(R)]\}$$

*Completed*

Student	Task
Fred	Database1
Fred	Database2
Fred	Compiler1
Eugene	Database1
Eugene	Compiler1
Sarah	Database1
Sarah	Database2

*DBProject*

Task
Database1
Database2

*Completed*

÷

*DBProject*

Student
Fred
Sarah

**R**

A	B
a1	b1
a2	b1
a3	b1
a4	b1
a1	b2
a3	b2
a2	b3
a3	b3
a4	b3
a1	b4
a2	b4
a3	b4

**S**

A
a1
a2
a3

**T**

B
b1
b4

# Introduction to SQL

- Structured Query Language is a domain-specific language (not general purpose) used in programming and design for managing data held in a relational database management system (RDBMS).
- Although we refer to the SQL language as a “query language,” it can do much more than just query a database. It can define the structure of the data base, modify data in the database, specify security constraints and number of other tasks.
- Originally based upon relational algebra(procedural) and tuple relational calculus (Non-procedural) mathematical model.

# Overview of the SQL Query Language

1. IBM developed the original version of SQL, originally called Sequel (*Structured English Query Language*), as part of the System R project in the early 1970s.
2. The Sequel language has evolved since then, and its name has changed to SQL (Structured Query Language) (some other company has trademark on the word sequel). SQL has clearly established itself as *the* standard relational database language.
3. In 1986, the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO) published an SQL standard, called SQL-86.
4. The next version of the standard was SQL-89, SQL-92, SQL:1999, SQL:2003, SQL:2006, SQL:2008, SQL:2011, SQL: 2016, SQL: 2019 and most recently SQL:2023.



# Classification of database languages

## 1. Data Definition Language (DDL) :

1. a. DDL is set of SQL commands used to create, modify and delete database structures but not data.
2. b. They are used by the DBA to a limited extent, a database designer, or application developer.
3. c. Create, drop, alter, truncate are commonly used DDL command. CREATE, ALTER, DROP, TRUNCATE, COMMENT, GRANT, REVOKE statement

## 2. Data Manipulation Language (DML) :

1. a. A DML is a language that enables users to access or manipulates data as organized by the appropriate data model.
2. b. There are two types of DMLs :
  1. i. Procedural DMLs : It requires a user to specify what data are needed and how to get those data.
  2. ii. Declarative DMLs (Non-procedural DMLs) : It requires a user to specify what data are needed without specifying how to get those data.
3. c. Insert, update, delete, query are commonly used DML commands. INSERT, UPDATE, DELETE statement

### **3. Data Control Language (DCL) :**

1. a. It is the component of SQL statement that control access to data and to the database.
2. b. Commit, rollback command are used in DCL. GRANT and REVOKE statement

### **4. Data Query Language (DQL) :**

1. a. It is the component of SQL statement that allows getting data from the database and imposing ordering upon it.
2. b. It includes select statement. SELECT statement

### **5. View Definition Language (VDL) :**

1. 1. VDL is used to specify user views and their mapping to conceptual schema.
2. 2. It defines the subset of records available to classes of users.
3. 3. It creates virtual tables and the view appears to users like conceptual level.
4. 4. It specifies user interfaces. SQL is a DML language.

```
CREATE TABLE table_name (  
    column1 data_type [constraints],  
    column2 data_type [constraints],  
    column3 data_type [constraints],  
    ...  
);
```

```
CREATE TABLE Students (  
    StudentID INT PRIMARY KEY,  
    FirstName VARCHAR(50),  
    LastName VARCHAR(50),  
    Age INT,  
    Email VARCHAR(100)  
);
```

- list of some common data types supported by SQL along with a brief description of each:
- Numeric Data Types:
  - 1. `INT`: For storing integer values.
  - 2. `SMALLINT`: A smaller range of integers compared to INT.
  - 3. `BIGINT`: For storing larger integers.
  - 4. `DECIMAL(p, s)`: For storing exact numerical values, where `p` is the precision and `s` is the scale.
  - 5. `FLOAT`: For storing floating-point numbers.
  - 6. `REAL`: A data type that can store floating-point numbers, generally with less precision compared to FLOAT.
- String Data Types:
  - 7. `VARCHAR(n)`: Variable-length character string, where `n` is the maximum length.
  - 8. `CHAR(n)`: Fixed-length character string, where `n` is the length.
  - 9. `TEXT`: For storing long text strings.

## **Adding a New Column**

```
ALTER TABLE Employees  
ADD PhoneNumber VARCHAR(15);
```

## **Dropping a Column**

```
ALTER TABLE Employees  
DROP COLUMN PhoneNumber;
```

## **Modifying an Existing Column**

```
ALTER TABLE Employees  
MODIFY COLUMN PhoneNumber VARCHAR(20);
```

```
ALTER TABLE Employees  
ALTER COLUMN PhoneNumber VARCHAR(20);
```

## **Renaming a Column**

```
ALTER TABLE Employees  
RENAME COLUMN PhoneNumber TO ContactNumber;
```

## **Renaming a table**

```
ALTER TABLE Employees  
RENAME TO Staff;
```

```
DROP TABLE table_name;
```

```
CREATE TABLE Orders (  
    OrderID INT PRIMARY KEY,  
    CustomerID INT,  
    OrderDate DATE,  
    FOREIGN KEY (CustomerID) REFERENCES Customers (CustomerID)  
);
```

```
ALTER TABLE Orders  
ADD FOREIGN KEY (CustomerID) REFERENCES Customers (CustomerID);
```

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

```
INSERT INTO Students (StudentID, FirstName, LastName, Age)
VALUES
(1, 'Amit', 'Sharma', 20),
(2, 'Priya', 'Gupta', 22),
(3, 'Ravi', 'Kumar', 19);
```

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

```
DELETE FROM Students  
WHERE StudentID = 1;
```

```
DELETE FROM table_name;
```



# Basic Structure of SQL Queries

- For any SQL as query, input and output both are relations. Number of relations inputs to a query will be at least one, but output will always be a single relation without any name unless specified, but columns will have names from input tables.
- The basic structure of an SQL query consists of three clauses: select, from, and where. The query takes it's input the relations listed in the from clause, operates on them as specified in the where and select clauses, and then produces a relation as the result without any name unless specified.
- A typical SQL query has the form.

<b>Select</b> $A_1, A_2, \dots, A_n$	(Column name)
<b>from</b> $r_1, r_2, \dots, r_m$	(Relation/table name)
<b>Where</b> P;	(Condition)

<b>Select</b> $A_1, A_2, \dots, A_n$	(Column name)
<b>from</b> $r_1, r_2, \dots, r_m$	(Relation/table name)
<b>Where</b> $P;$	(Condition)

1. It is to be noted only select and from are mandatory clauses, and if not required then it is not essential to write where. If the **where** clause is omitted, the predicate  $P$  is **true**.
2. SQL in general is not case sensitive i.e. it doesn't matter whether we write query in upper or lower case.
3. In practice, duplicate elimination is time-consuming. Therefore, SQL allows duplicates in relations as well as in the results of SQL expressions. In those cases where we want to force the elimination of duplicates, we insert the keyword **distinct** after **select**, will discuss in detail later. SQL allows us to use the keyword **all** to specify explicitly that duplicates are not removed, Since duplicate retention is the default, we shall not use **all** in our examples.

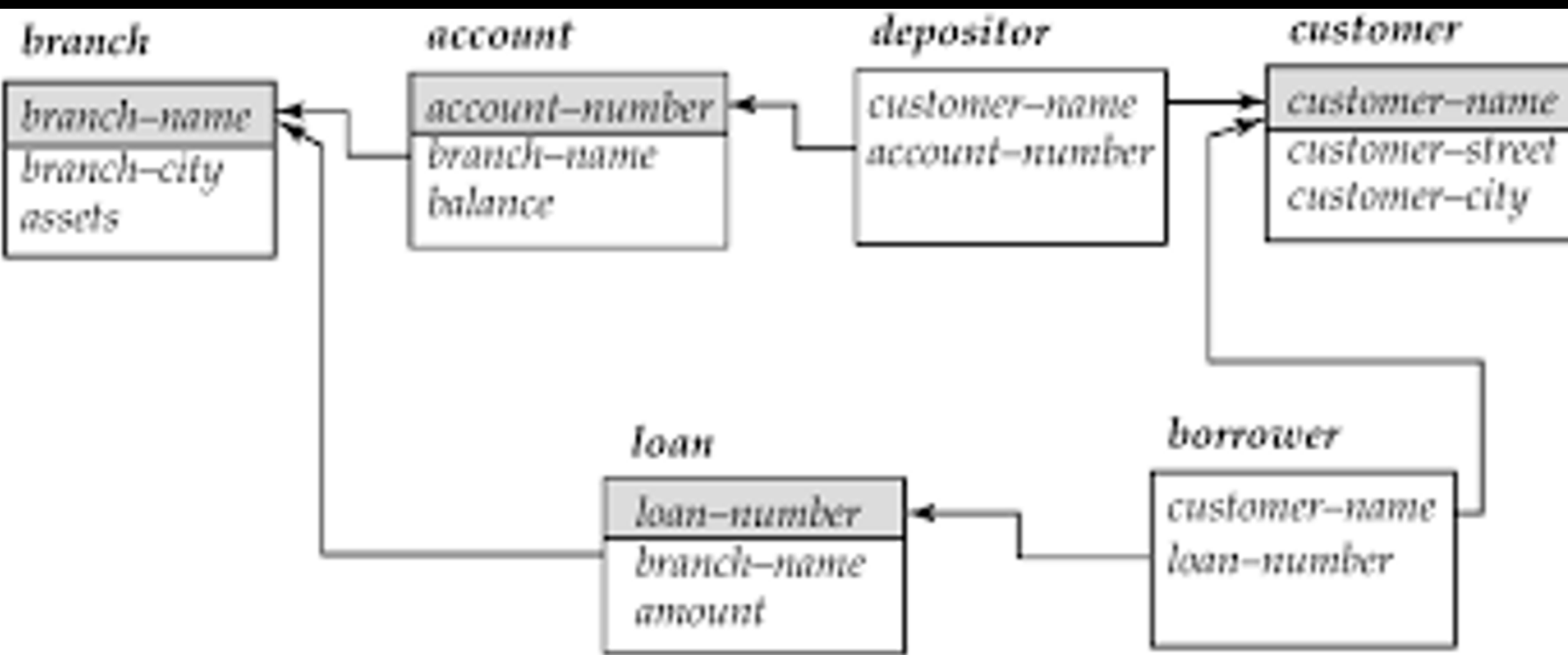


Q Write a SQL query to find all the details of bank branches?

Q Write a SQL query to find each loan number along with loan amount?

Q Write a SQL query to find the name of all customer without duplication having bank account?

Q Write a SQL query to find all account\_no and balance with 6% yearly interest added to it?



**Q find all the details of bank branches?**

```
Select *  
from branch
```

**Q find each loan number along with loan amount?**

```
Select loan_number, amount  
from loan
```

**Q find the name of all customer without duplication having bank account?**

```
Select distinct customer_name  
from depositor
```

**Q find all account\_no and balance with 6% yearly interest added to it?**

```
Select account_number, balance*1.06  
from account
```

# Select Clause with where clause

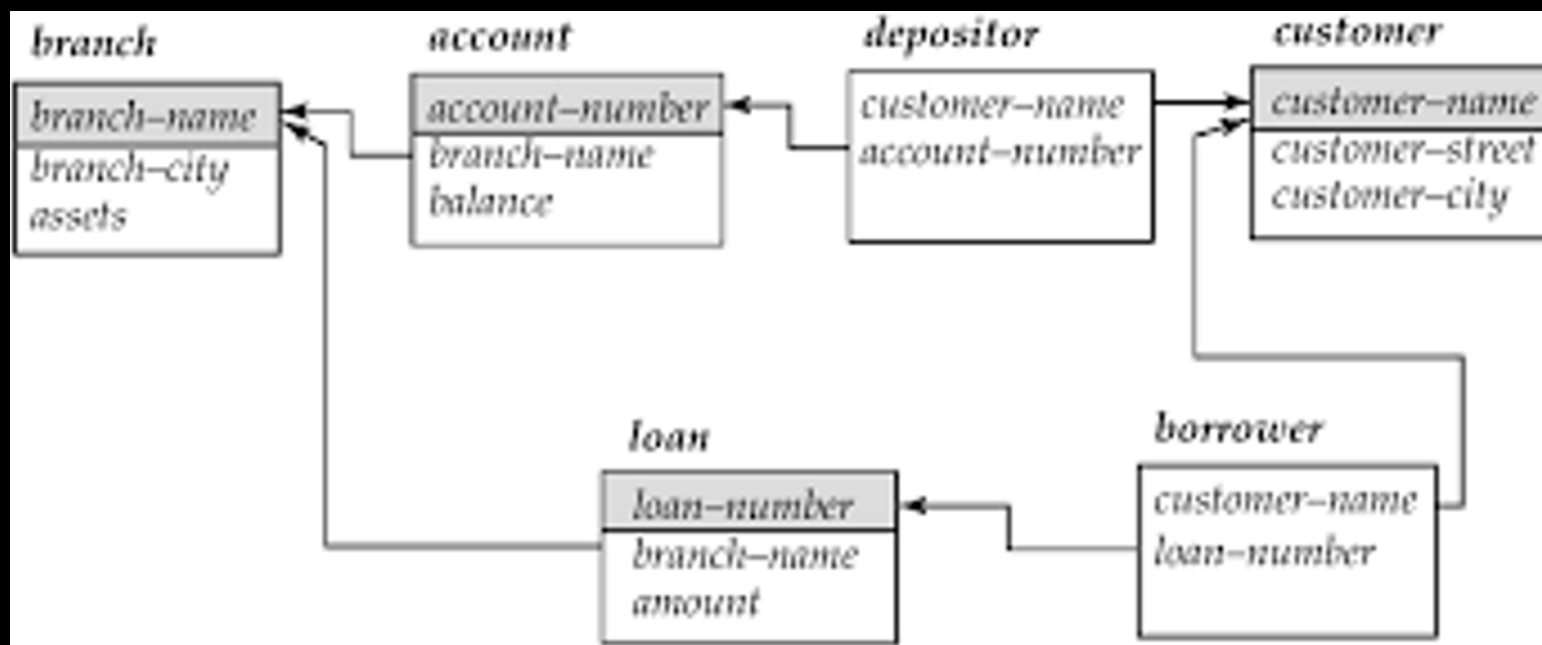
1. Where clause in SQL is same as ' $\sigma$ ' sigma of relational algebra where we specify the conditions/Predicate (horizontal filtering).
2. Where clause can have expressions involving the comparison operators  $<$ ,  $<=$ ,  $>$ ,  $>=$ ,  $=$  and  $\neq$ . SQL allows us to use the comparison operators to compare strings and arithmetic expressions.
3. SQL allows the use of the logical connectives **and**, **or**, and **not** in the **where** clause.
4. SQL includes a **between** comparison operator to simplify **where** clauses that specify that a value be less than or equal to some value and greater than or equal to some other value.
5. Similarly, we can use the **not between** comparison operator.




Q Write a SQL query to find all account\_no where balance is less the 1000?

Q Write a SQL query to find branch name which is situated in Delhi and having assets less than 1,00,000?

Q Write a SQL query to find branch name and account\_no which has balance greater than equal to 1,000 but less than equal to 10,000?



**Q find all account\_no where balance is less the 1000?**

```
Select account_number  
from account  
Where balance < 1000
```

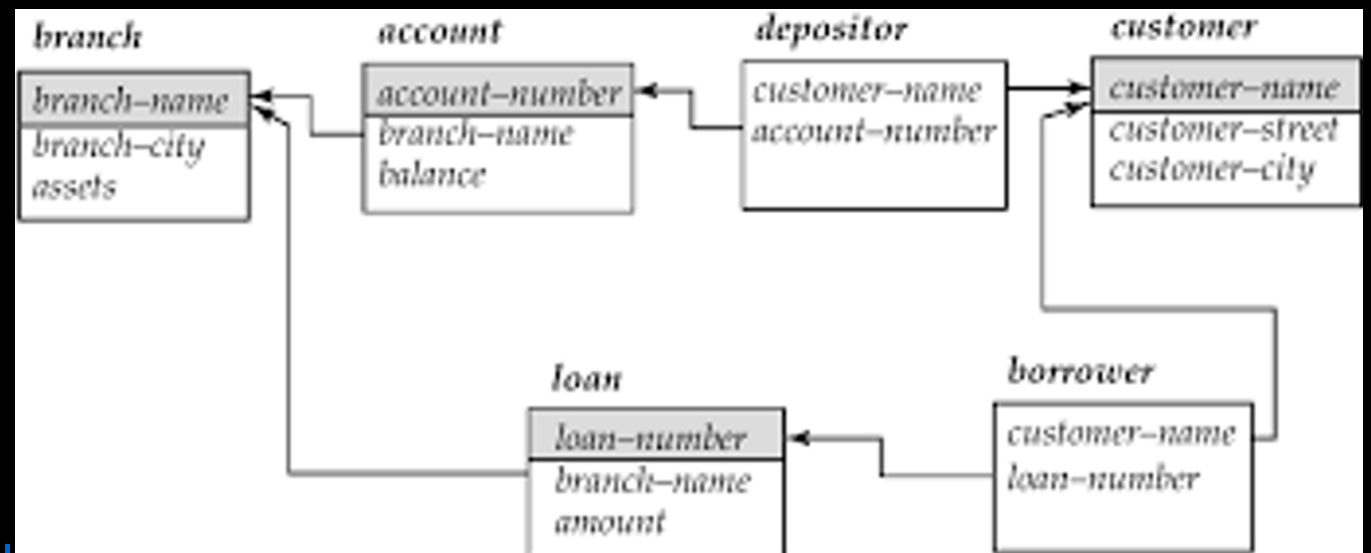
**Q Write a SQL query to find branch name which is situated in Delhi and having assets less than 1,00,000?**

```
Select branch_name  
from branch  
Where branch_city = 'delhi' and assets < 1,00,000
```

**Q Write a SQL query to find branch name and account\_no which has balance greater than equal to 1,000 but less than equal to 10,000?**

```
Select branch_name, account_number  
from account  
Where balance between 1000 and 10000
```

```
Select branch_name, account_number  
from account  
Where balance >= 1000 and balance<=10000
```





# Set Operation

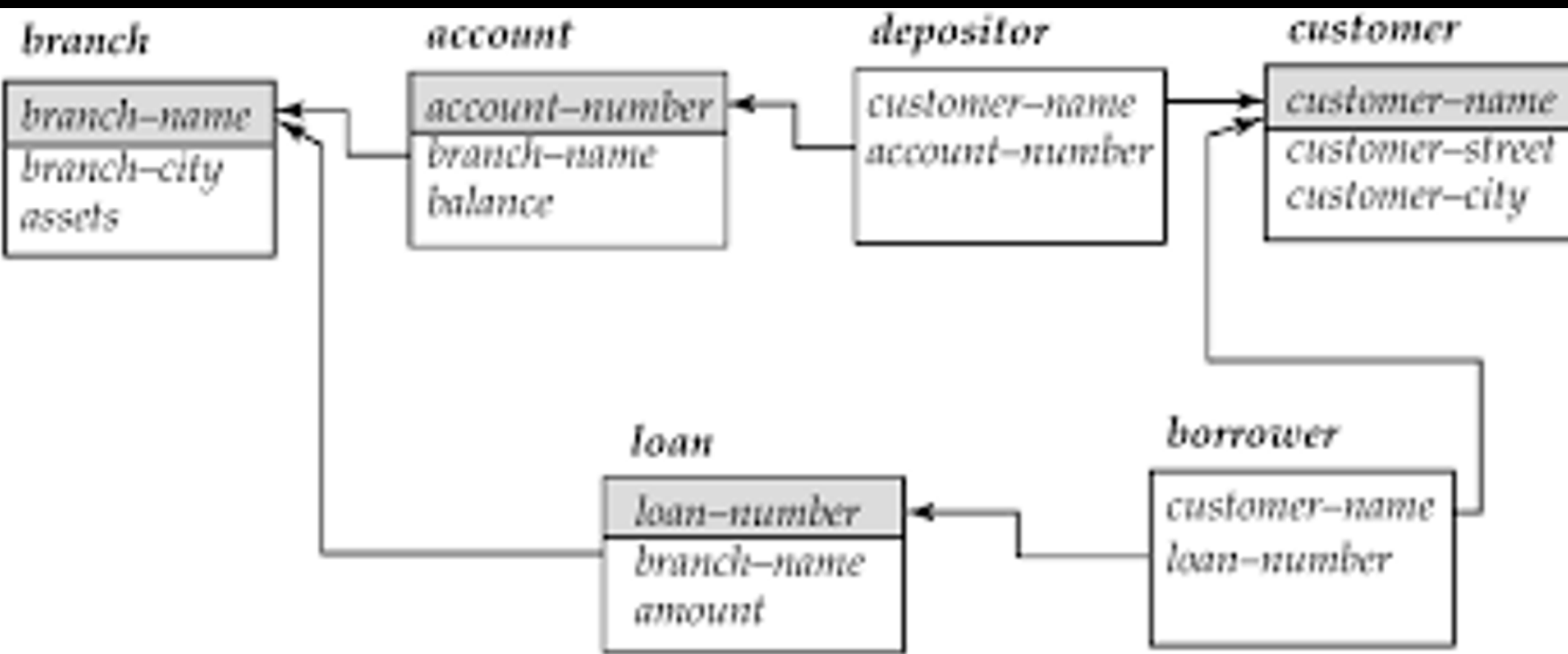
1. The SQL operations **union**, **intersect**, and **except/minus** operate on relations and corresponds to the mathematical set-theory operations  $\cup$ ,  $\cap$  and  $-$  respectively.
2. The **union** operation automatically eliminates duplicates, unlike the **select** clause, If we want to retain all duplicates, we must write **union all** in place of **union**.
3. The **intersect** operation automatically eliminates duplicates. If we want to retain all duplicates, we must write **intersect all** in place of intersect.
4. If we want to retain duplicates, we must write **except all** in place of **except**.

**Q** Write a SQL query to find all the customer name

**a)** who have a loan or an account or both ?

**b)** who have both a loan and an account?

**c)** who have a loan but do not have an account?



**Q** Write a SQL query to find all the customer name  
**a)** who have a loan or an account or both ?

Select customer\_name  
From depositor

**Union**

Select customer\_name  
From borrower

**b)** who have both a loan and an account?

Select customer\_name  
From depositor

**intersect**

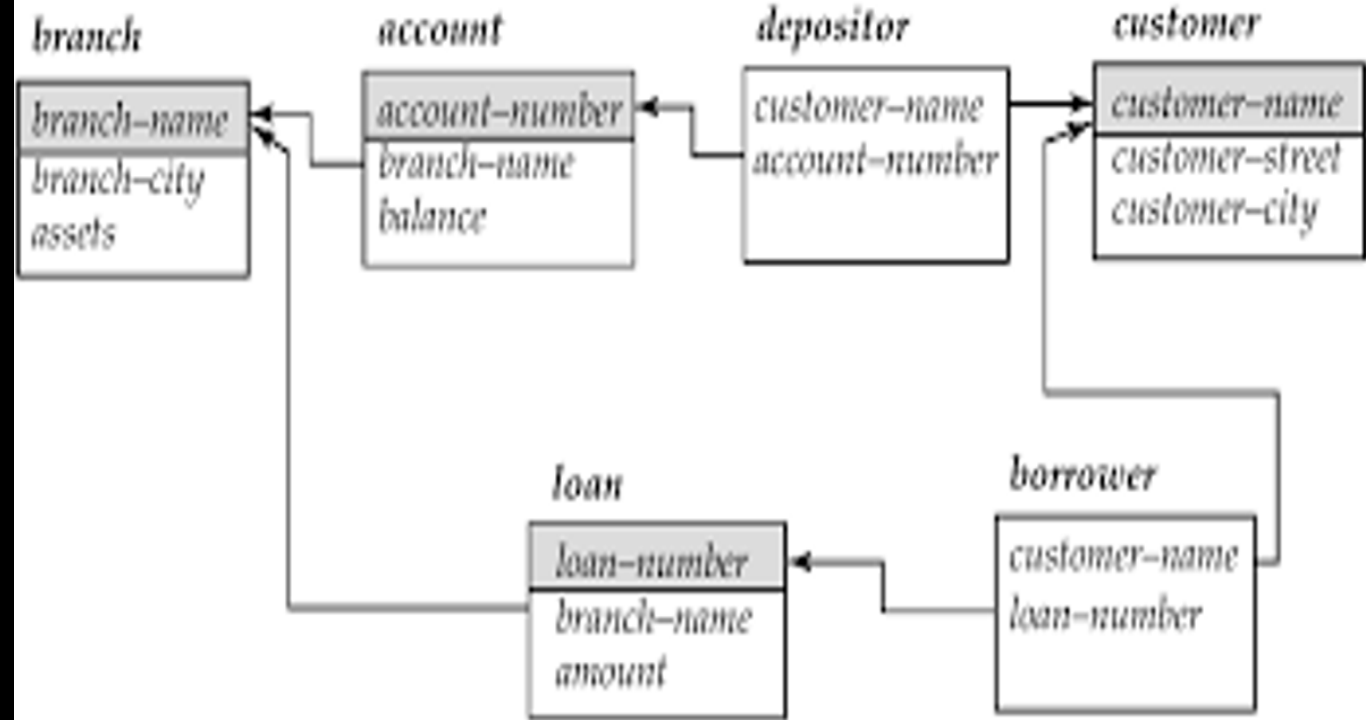
Select customer\_name  
From borrower

**c)** who have a loan but do not have an account?

Select customer\_name  
From borrower

**Except**

Select customer\_name  
From depositor



# Queries on Multiple Relations

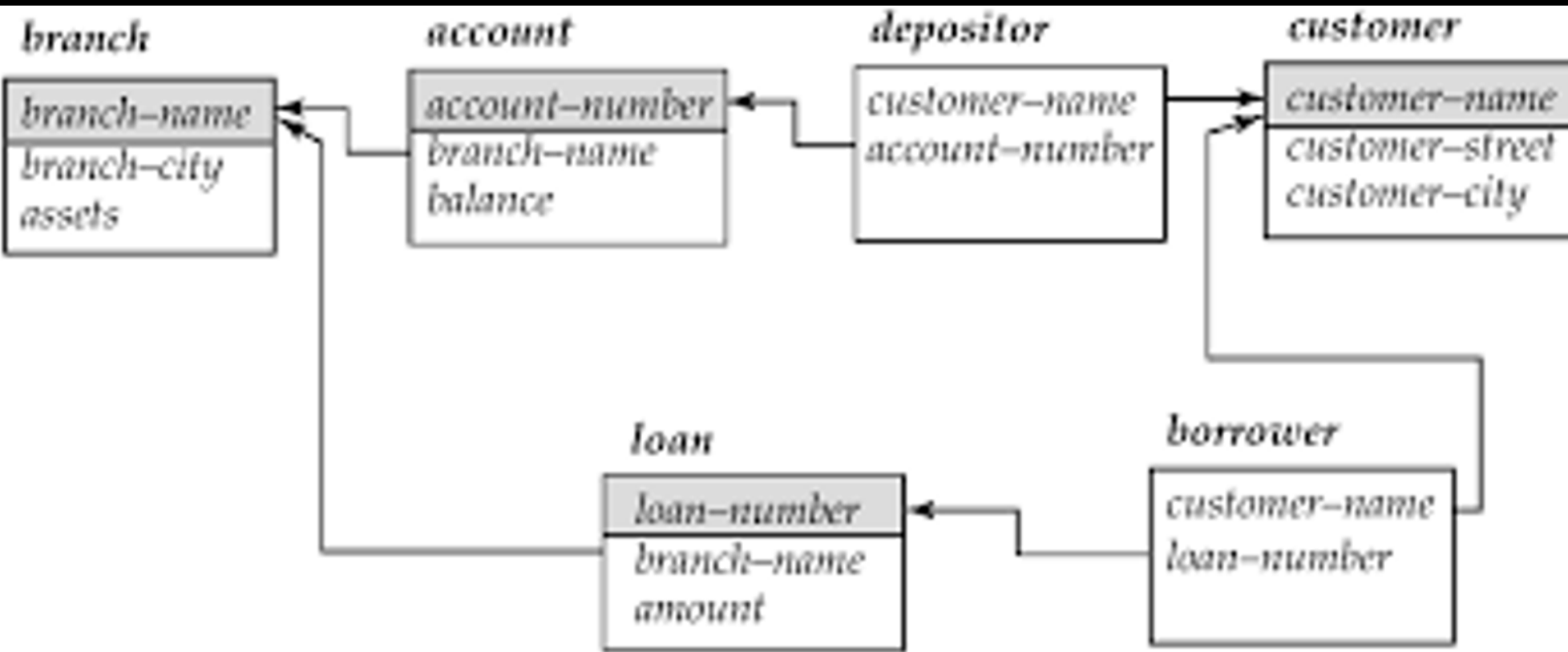
- The **from** clause by itself defines a Cartesian product of the relations listed in the clause. Cartesian product of two relations, which concatenates each tuple of the first relation with every tuple of the second
- Since the same attribute name may appear in both  $r_1$  and  $r_2$ , we prefix the name of the relation from which the attribute originally came, before the attribute name. For those attributes that appear in only one of the two schemas, we shall usually drop the relation-name prefix. This simplification does not lead to any ambiguity.
- Cartesian Product is commutative in nature

R <sub>1</sub>	
A	B
1	P
2	Q
3	R

R <sub>2</sub>	
B	C
Q	X
R	Y
S	Z

R <sub>1</sub> X R <sub>2</sub>			
A	R <sub>1</sub> .B	R <sub>2</sub> .B	C
1	P	Q	X
1	P	R	Y
1	P	S	Z
2	Q	Q	X
2	Q	R	Y
2	Q	S	Z
3	R	Q	X
3	R	R	Y
3	R	S	Z

Q Write a RELATIONAL ALGEBRA query to find the name of all the customers along with account balance, who have an account in the bank?

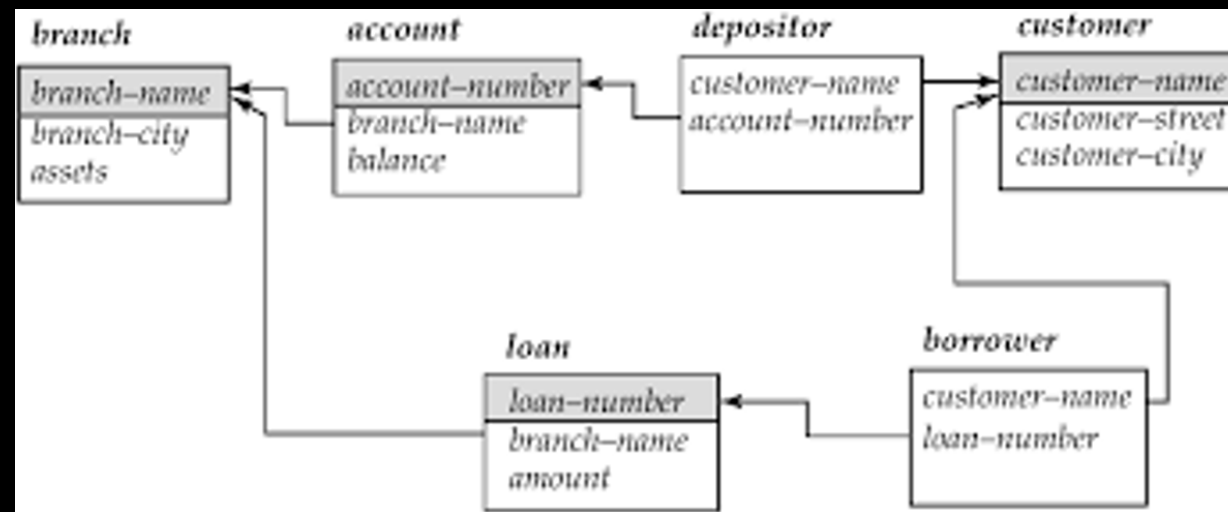


**Q** find the name of all the customers with account balance, who have an account in the bank?

Select customer\_name, balance

From account, depositor

Where account.account\_number = depositor.account\_number



# Natural Join

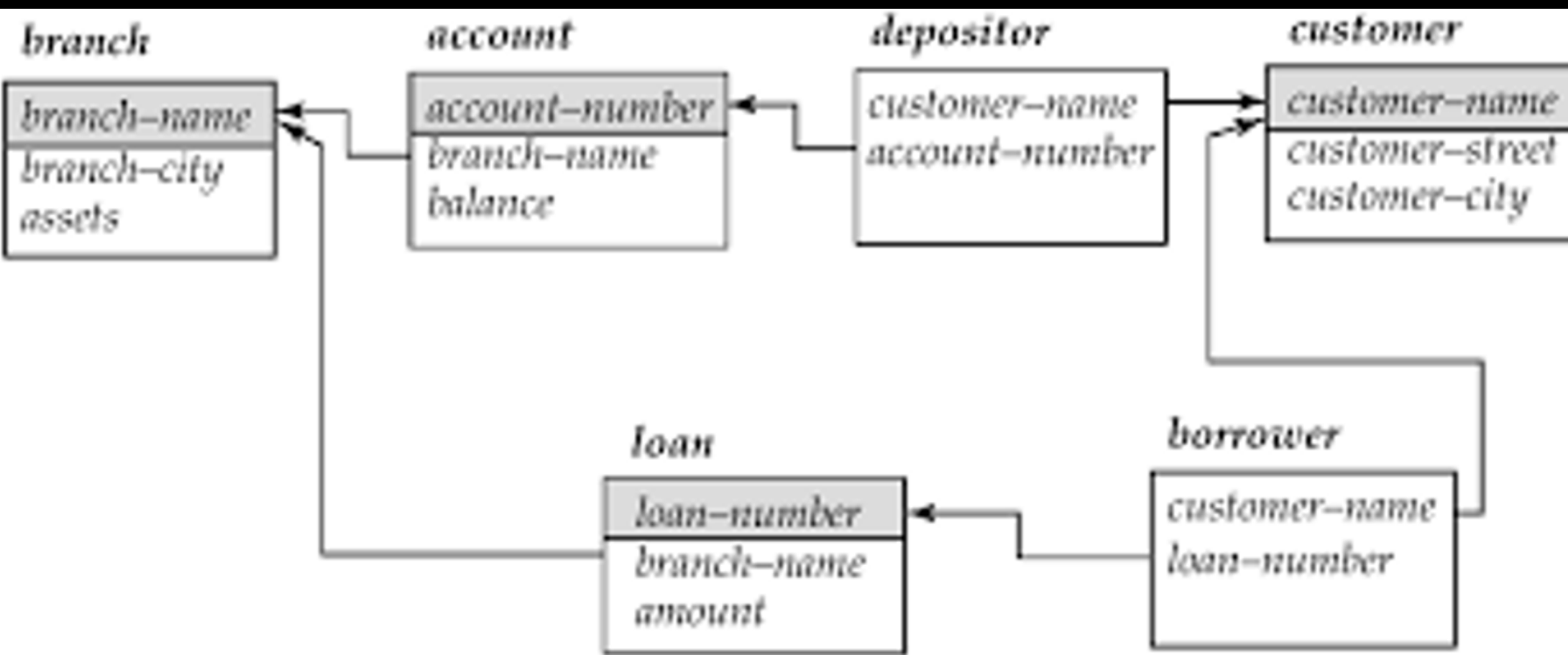
- To make the life of an SQL programmer easier for this common case, SQL supports an operation called the *natural join*. The **natural join** operation like cartesian product operates on two relations and produces a relation as the result.
- Natural join considers only those pairs of tuples with the same value on those attributes that appear in the schemas of both relations. Notice that we do not repeat those attributes that appear in the schemas of both relations; rather they appear only once.
- Notice also the order in which the attributes are listed: first the attributes common to the schemas of both relations, second those attributes unique to the schema of the first relation, and finally, those attributes unique to the schema of the second relation. Commutative in nature.

R <sub>1</sub>	
A	B
1	P
2	Q
3	R

R <sub>2</sub>	
B	C
Q	X
R	Y
S	Z

R <sub>1</sub> ⋈ R <sub>2</sub>		
A	B	C
2	Q	X
3	R	Y

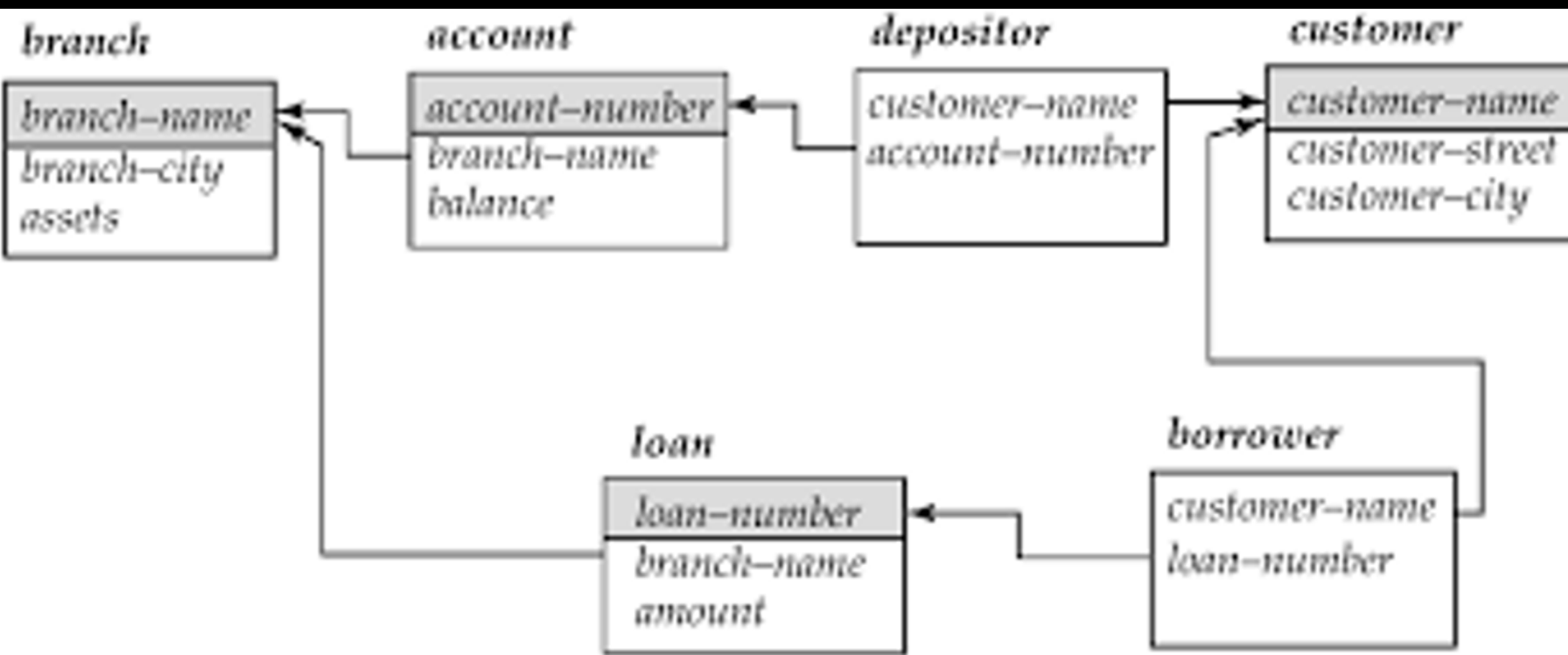
Q Write a SQL query to find the name of all the customers along with account balance, who have an account in the bank?





**Q** Write a SQL query to find the name of all the customers along with account balance, who have an account in the bank?

Select customer\_name, balance  
From account natural join depositor



## Outer Join

- The problem with natural join or join or inner join is only those values that appears in both relations will manage to reach final table, but if some value is explicitly in table one or in second table then that information will be lost, and that will be loss of information.
- The outer join operation works in a manner similar to the join operations we have already studied, but preserve those tuples that would be lost in a join, by creating tuples in the result containing null values.
- There are in fact three forms of outer join:
  - The **left outer join** (left join) preserves tuples only in the relation named before (to the left of) the left outer join operation.
  - The **right outer join** (right join) preserves tuples only in the relation named after (to the right of) the right outer join operation.
  - The **full outer** join preserves tuples in both relations.

$R_1$	
A	B
1	P
2	Q
3	R

$R_2$	
B	C
Q	X
R	Y
S	Z

$R_1 * R_2$			
A	$R_1.B$	$R_2.B$	C
1	P	Q	X
1	P	R	Y
1	P	S	Z
2	Q	Q	X
2	Q	R	Y
2	Q	S	Z
3	R	Q	X
3	R	R	Y
3	R	S	Z

$R_1 \bowtie R_2$		
A	B	C
2	Q	X
3	R	Y

$R_1 \bowtie R_2$		
A	B	C
1	P	null
2	Q	X
3	R	Y

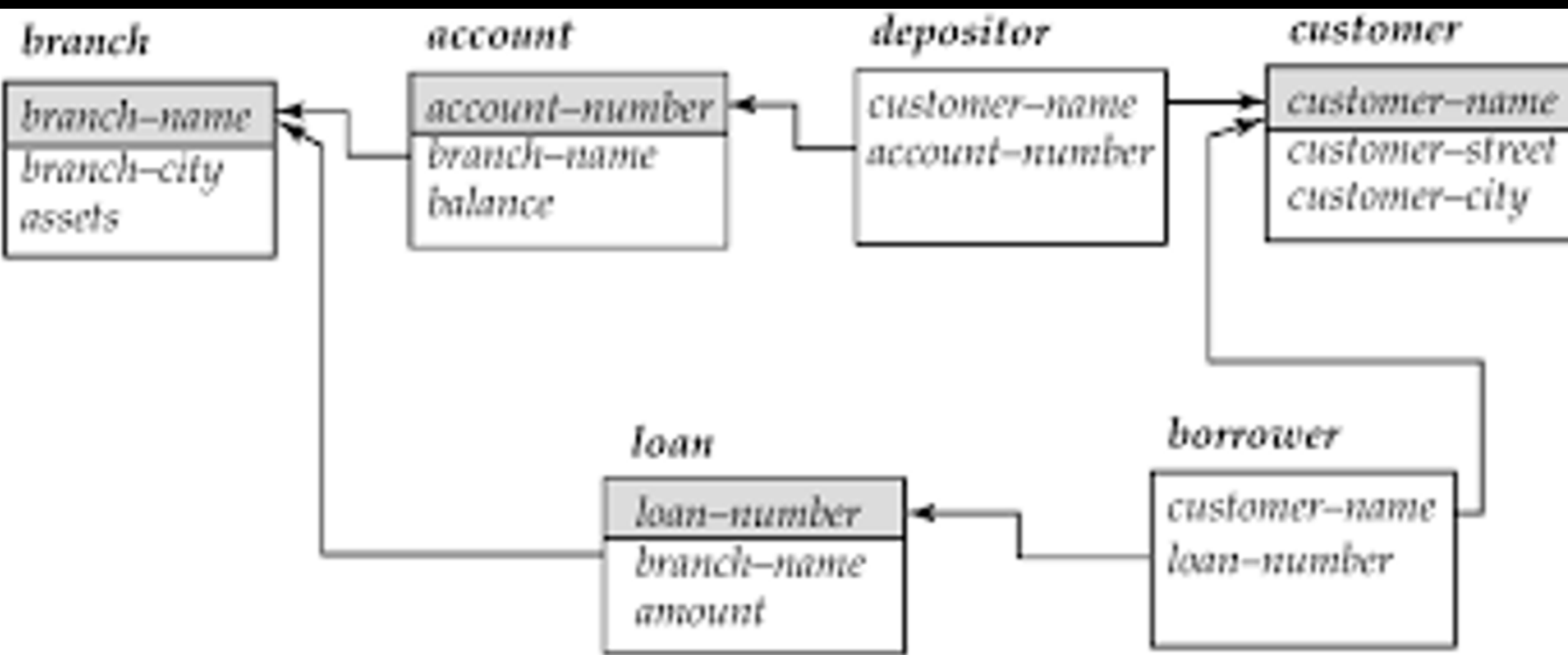
$R_1 \bowtie R_2$		
A	B	C
2	Q	X
3	R	Y
null	S	Z

$R_1 \bowtie R_2$		
A	B	C
1	P	null
2	Q	X
3	R	Y
null	S	Z

# Alias Operation/rename

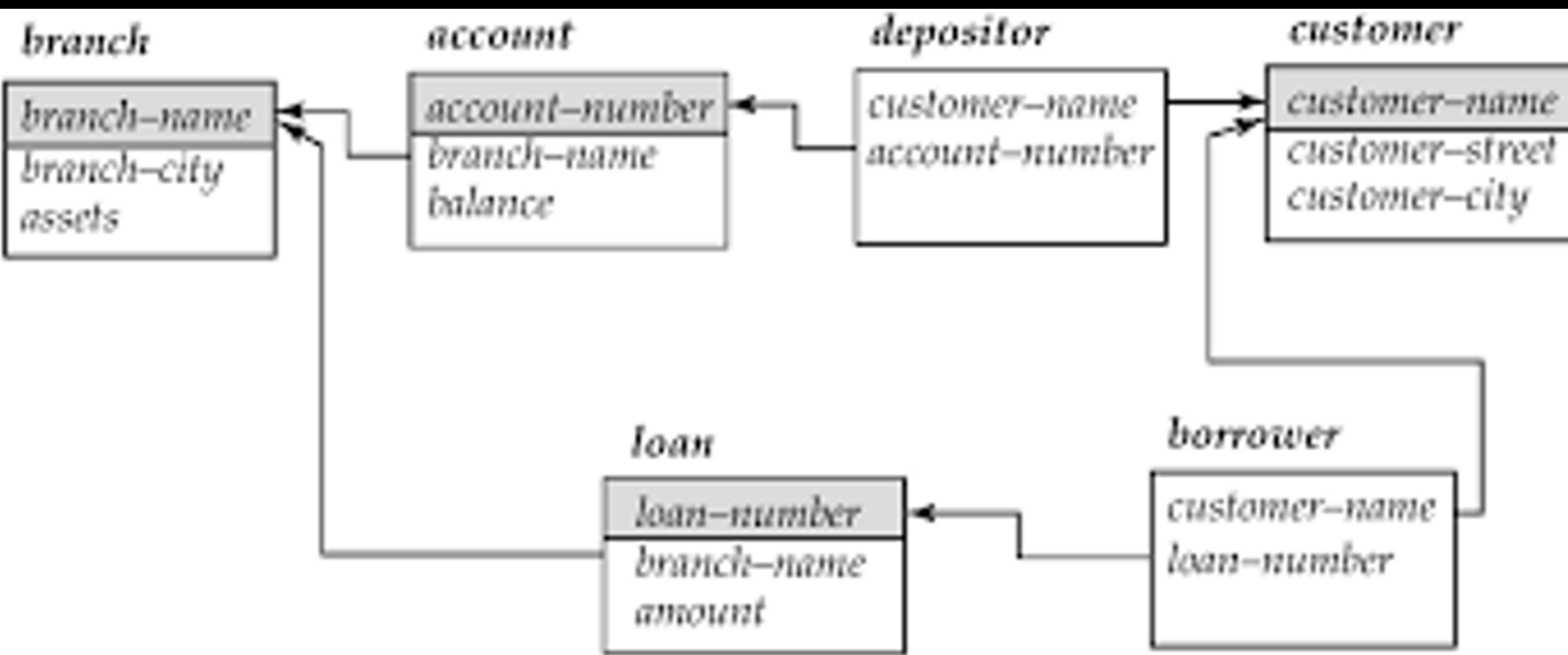
- SQL aliases are used to give a table, or a column in a table, a temporary name. Just create a new copy but do not change anything in the data base. An alias only exists for the duration of the query.
- Aliases are often used to make column names more readable.
- It uses the as clause, taking the form: old-name as new-name. The as clause can appear in both the select and from clauses.

Q Write a SQL query to find the account\_no along and balance with 8% interest, as Account, total\_balance?



**Q Write a SQL query to find the account\_no along and balance with 8% interest, as Account, total\_balance?**

Select account\_number, balance\*1.06 as total\_balance  
From account



**Q Write a SQL query to find the loan\_no with maximum loan amount?**

Select balance

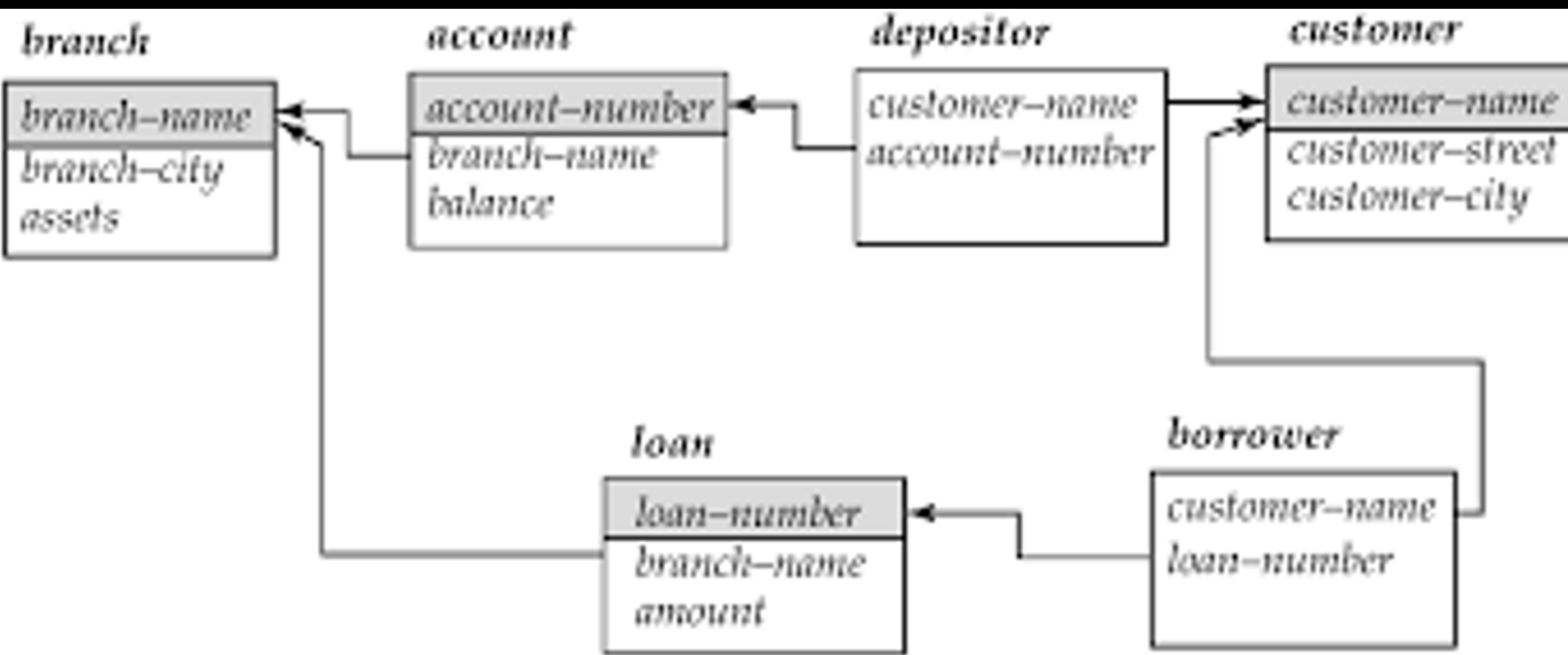
From account

**Except**

Select A.balance

From account as A, account as B

Where A.balance < B.balance



# Aggregate Functions

- *Aggregate functions* are functions that take a collection (a set or multiset) of values as input and return a single value. SQL offers five built-in aggregate functions:
  - Average: **avg**
  - Minimum: **min**
  - Maximum: **max**
  - Total: **sum**
  - Count: **count**
- The input to **sum** and **avg** must be a collection of numbers, but the other operators can operate on collections of nonnumeric data types, such as strings, as well. Count is the only aggregate function which can work with null, all other aggregate functions simply ignore null.
- We use the aggregate function **count** frequently to count the number of tuples in a relation. The notation for this function in SQL is **count (\*)**.



Q find the number of accounts in the bank?

Select count(\*)

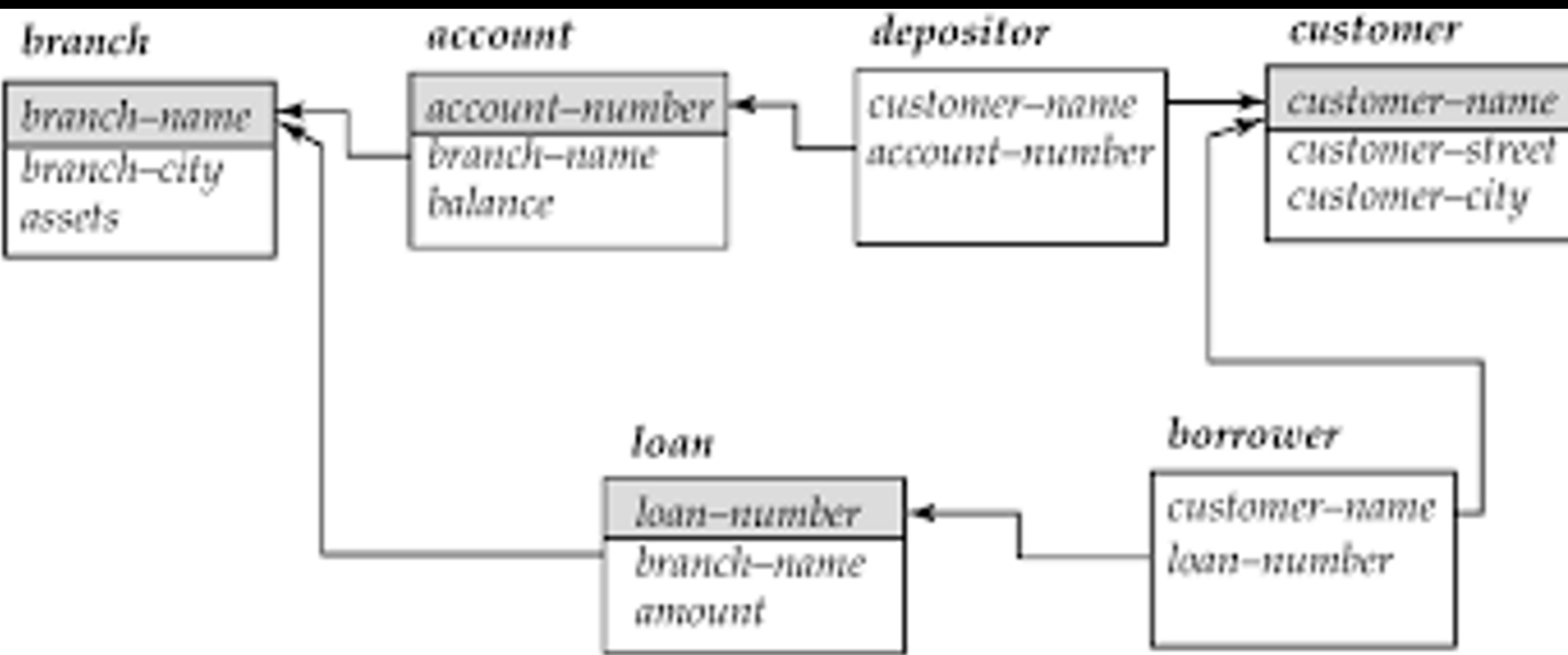
from account

Q find the average balance of every account in the banks from south\_delhi branch?

Select avg(balance)

from account

Where branch\_name = 'south\_delhi'



## Q Consider a table along with two query?

**Select avg (balance)  
from account**

Account_no	balance	Branch_name
Abc123	100	N_delhi
Pqr123	500	S_mumbai
Wyz123	null	S_delhi

**Select sum(balance)/count(balance)  
from account**

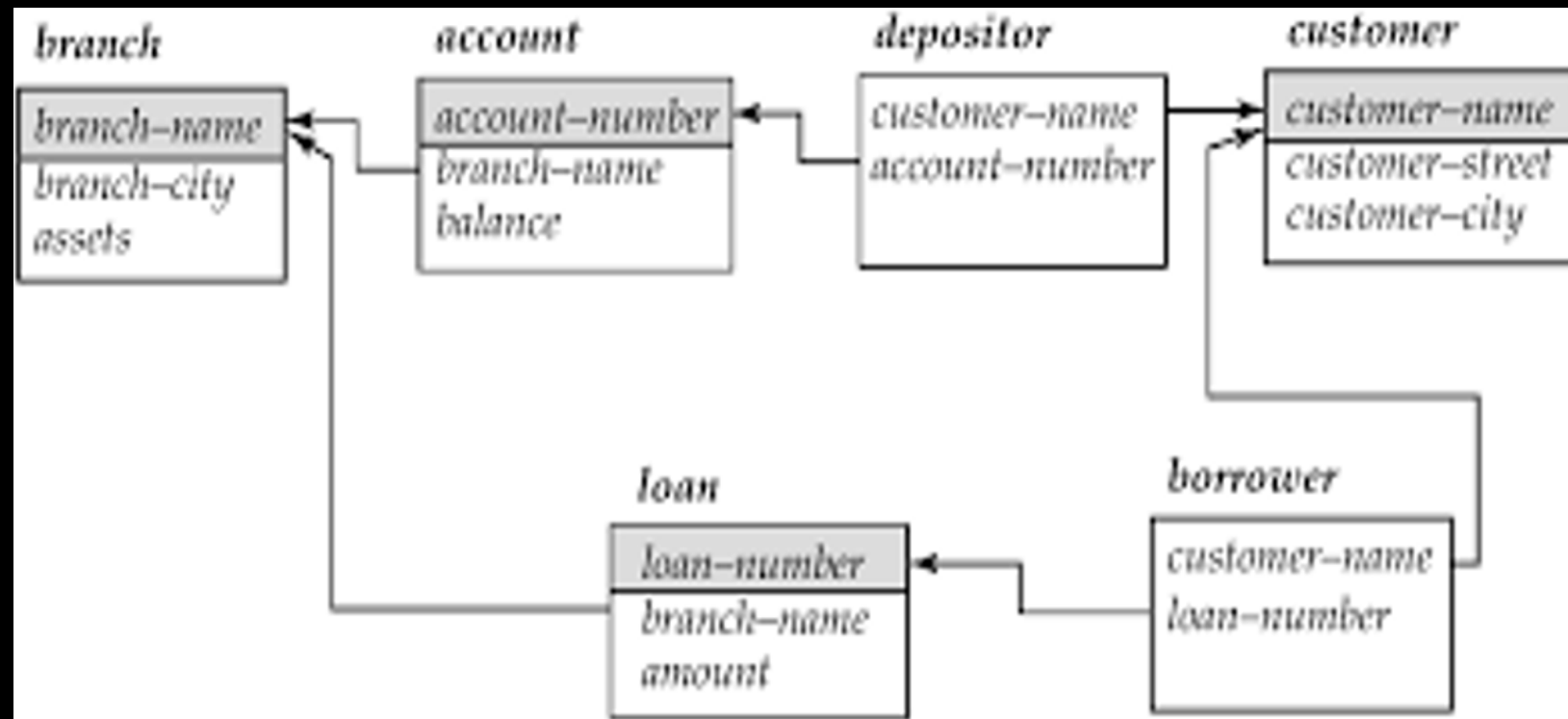
# Ordering the Display of Tuples

- SQL offers the user some control over the order in which tuples in a relation are displayed. The **order by** clause causes the tuples in the result of a query to appear in sorted order.

**Q** find all the **branch\_name** which are situated in Delhi in alphabetic order?

```
Select distinct branch_name  
from branch  
where branch_city = 'Delhi'  
Order by branch_name aesc;
```

```
Select distinct branch_name  
from branch  
where branch_city = 'Delhi'  
Order by branch_name desc;
```



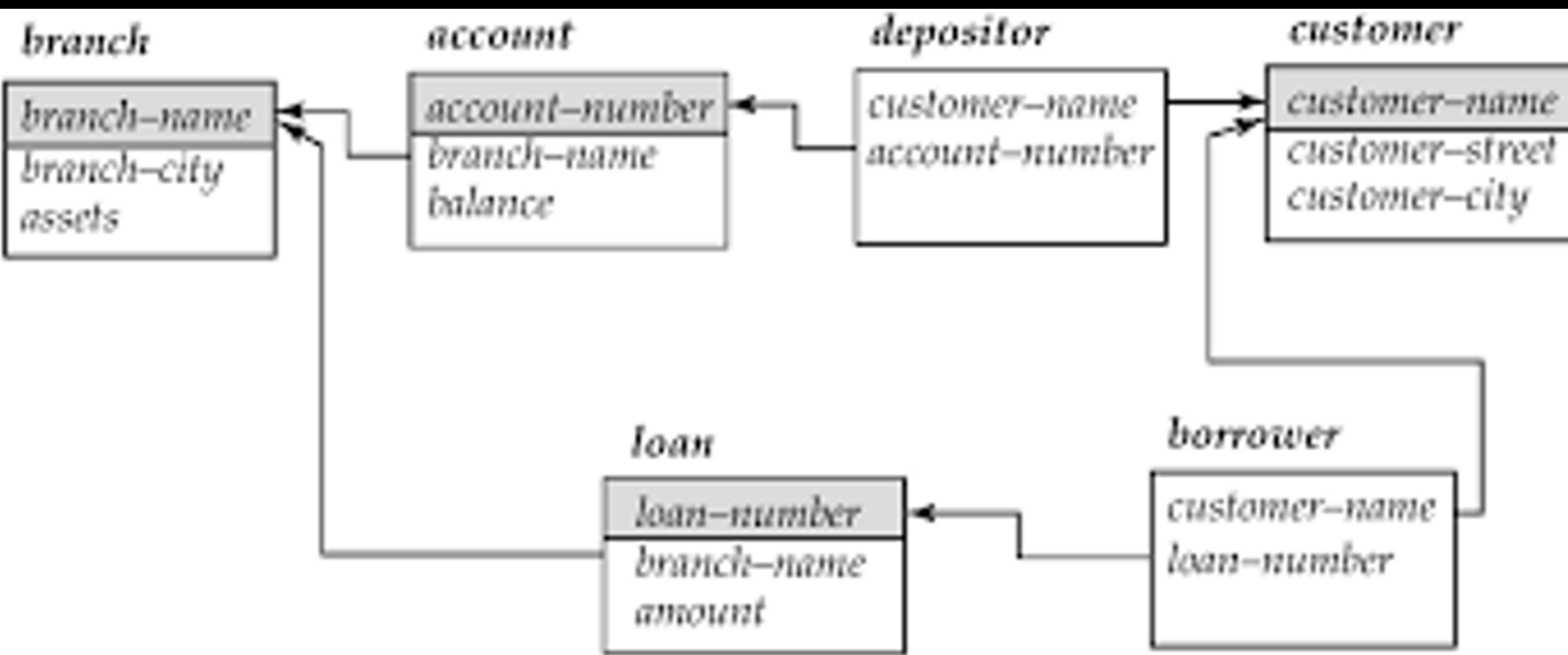
# String Operations

- SQL specifies strings by enclosing them in single quotes, for example, 'Computer'. The SQL standard specifies that the equality operation on strings is case sensitive; as a result the expression 'Computer' = 'computer' evaluates to false.
- However, some database systems, such as MySQL and SQL Server, do not distinguish uppercase from lowercase when matching strings; as a result, would evaluate to true on these databases. This default behavior can, however, be changed, either at the database level or at the level of specific attributes.
- SQL also permits a variety of functions on character strings, such as concatenating, extracting substrings, finding the length of strings, converting strings to uppercase and lowercase, removing spaces at the end of the string and so on. There are variations on the exact set of string functions supported by different database systems.

- Pattern matching can be performed on strings, using the operator **like**. We describe patterns by using two special characters:
  - Percent (%): The % character matches any substring.
  - Underscore (\_): The \_ character matches any character.
- '%Comp%' matches any string containing "Comp" as a substring, for example, 'Intro to Computer Science', and 'Computational Biology'.
  - '\_\_\_' matches any string of exactly three characters.
  - '\_\_\_%' matches any string of at least three characters.

Q find all the branch name who have exactly 5 character in their name ?

Q find all the customer name who have 'kumar' in their name ?

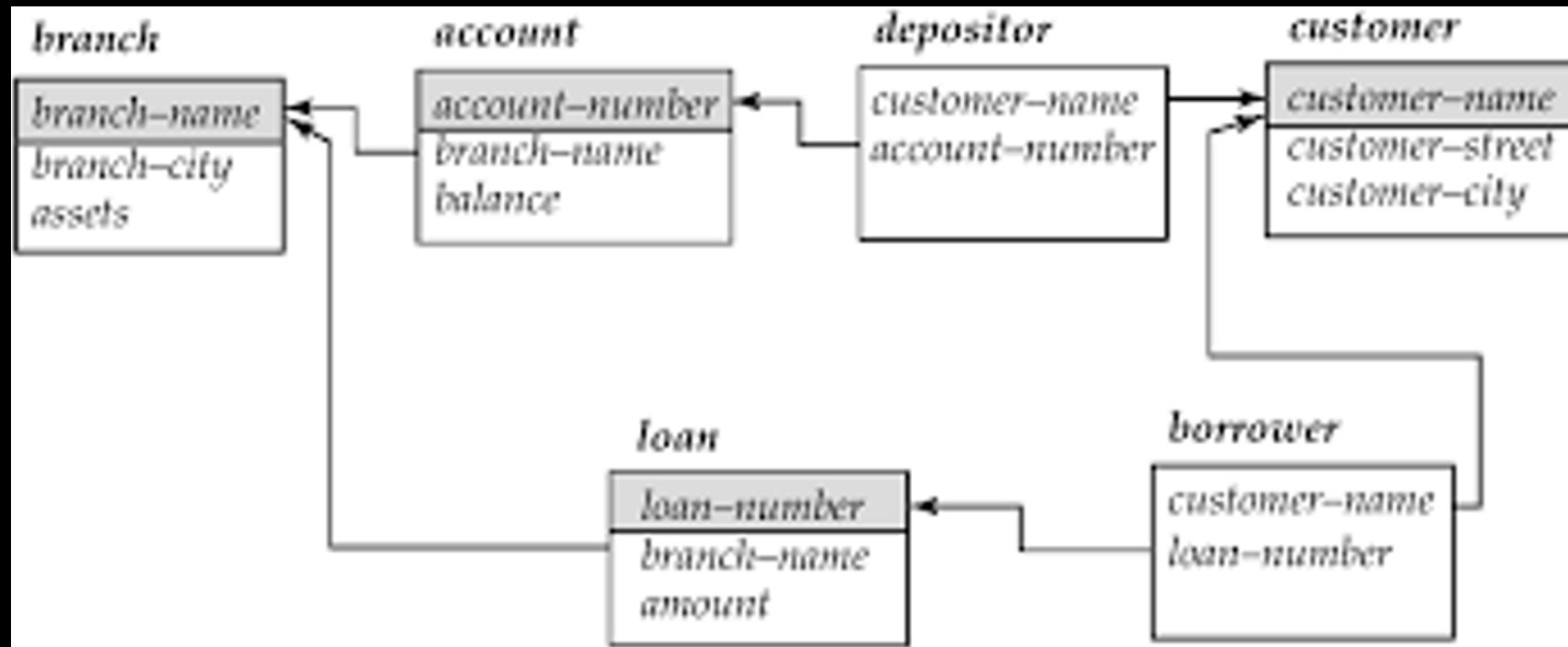


**Q find all the branch name who have exactly 5 character in their name ?**

```
Select branch_name  
from branch  
where branch_name like '_____'
```

**Q find all the customer name who have 'kumar' in their name ?**

```
Select customer_name  
from customer  
where customer_name like '%kumar%'
```



- We define the escape character for a **like** comparison using the **escape** keyword. To illustrate, consider the following patterns, which use a backslash (\) as the escape character:
- **like** 'ab\%cd%' **escape** '\' matches all strings beginning with "ab%cd".
- **like** 'ab\\cd%' **escape** '\' matches all strings beginning with "ab\cd".



## Group by clause

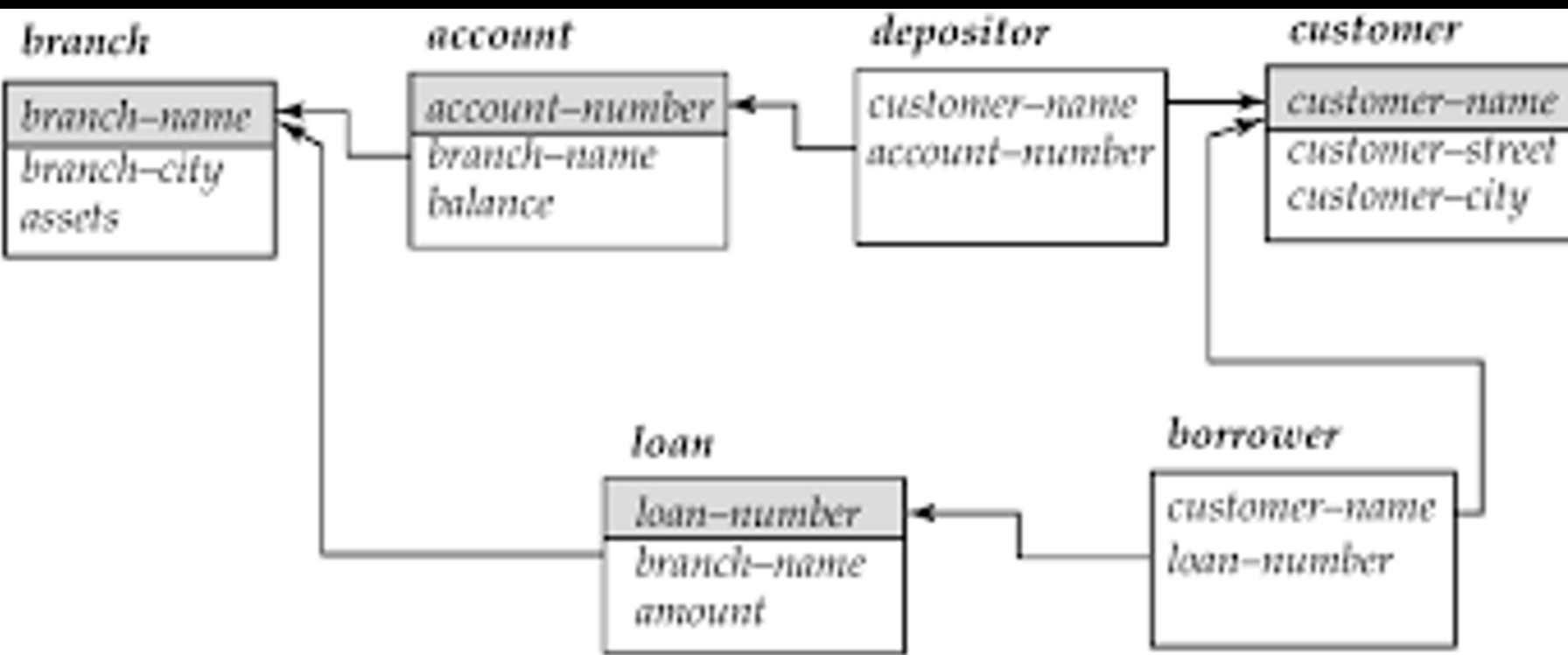
1. There are circumstances where we would like to work on a set of tuples in a relation rather than working on the whole table as one unit.
2. The attribute or attributes given in the **group by** clause are used to form groups. Tuples with the same value on all attributes in the **group by** clause are placed in one group.

**Q find the average account balance of each branch?**

Select branch\_name, avg(balance)

from account

Group by branch\_name



**Q find the branch name of Gwalior city with average balance more than 1500?**

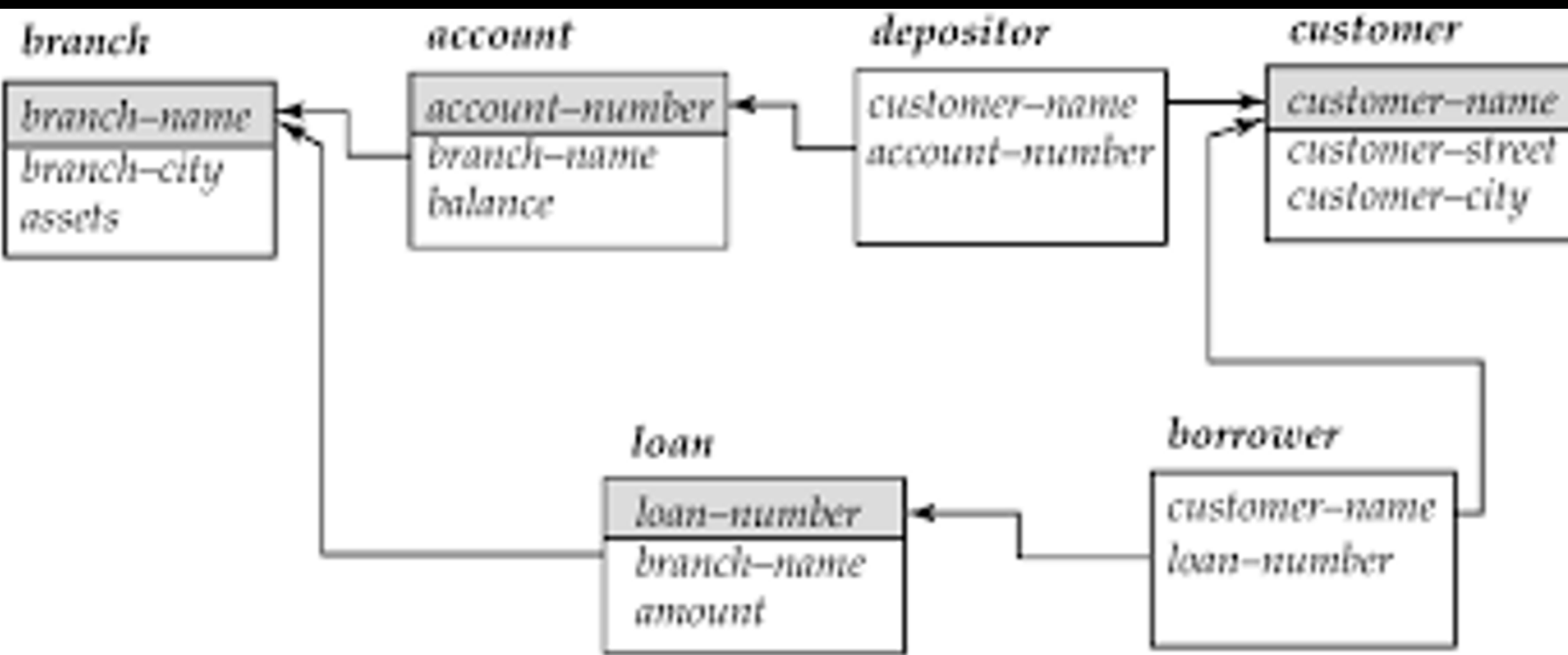
Select branch.branch\_name, avg(balance)

from branch, account

Where branch.branch\_name = account.branch\_name and branch\_city = 'gwalior'

Group by branch\_name

Having avg(balance) > 1500

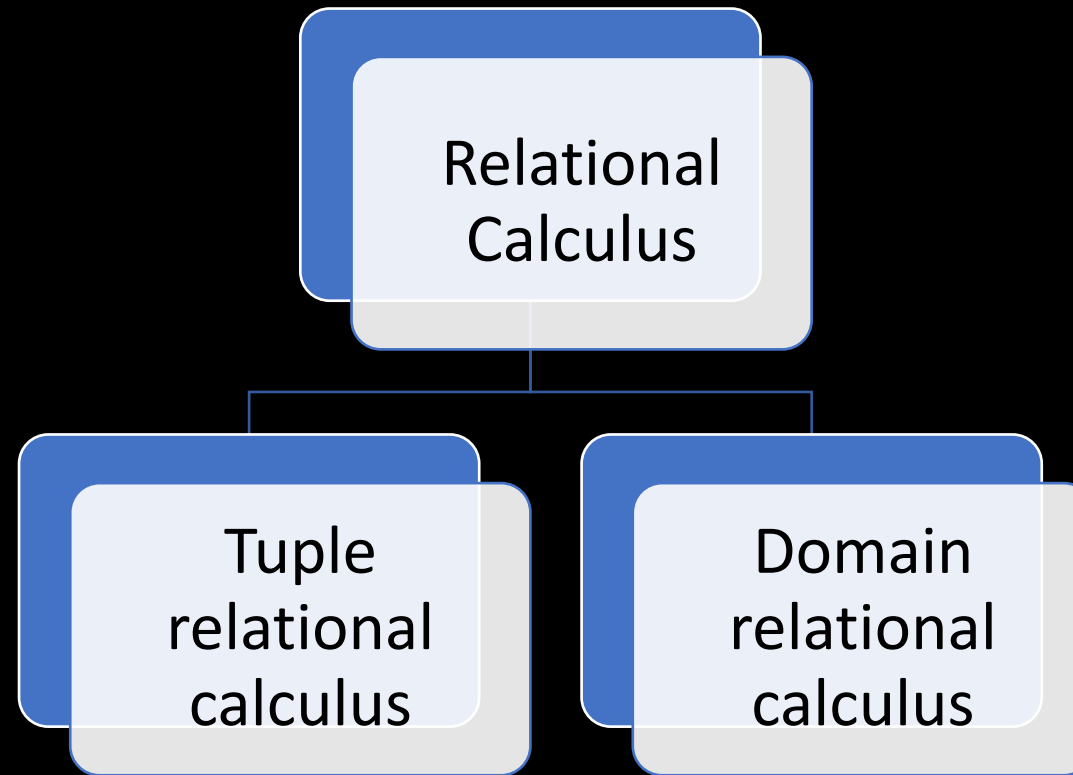


- **Trigger Definition**
  - A trigger is a special procedure that automatically activates in response to modifications on a table or view in a database, aiding in maintaining data integrity.
- **Usage of Triggers**
  - Triggers help maintain database integrity by automatically enforcing conditions or modifying data during database operations. They are crucial for applying business rules, auditing database alterations, and supporting data replication, ensuring compliance before any data insertions, updates, or deletions.

- **Embedded SQL**
  - Embedded SQL is a method where SQL statements are incorporated directly into a procedural programming language, such as C or Java. It allows the programmers to integrate SQL queries within their code, facilitating the interaction between the database and the application. Embedded SQL statements are static and defined at compile time.
- **Dynamic SQL**
  - Dynamic SQL, on the other hand, enables the construction of SQL statements dynamically at runtime. It allows the creation of more flexible and adaptable applications, where SQL statements can be generated and executed based on changing conditions or user inputs. This makes it possible to create more complex and adaptable database operations, though it might be more susceptible to SQL injection attacks if not handled carefully.

## Relational Calculus

- Relational calculus is non-procedural query language, where we have to define what to get and not how to get it



## Tuple Relational Calculus

- TRC is based on specifying a number of tuple variables. Each tuple variable usually range over a particular database relation, meaning that the variable may takes as its value from any individual tuple of a relation.
- A simple tuple relational calculus query is of the form.
  - $\{t \mid \text{Condition}(t)\}$
- Where  $t$  is a tuple variable and  $\text{condition}(t)$  is a conditional expression involving. The result of such a query is the set of all tuple  $t$  that satisfy condition ( $t$ ).

## Student(Roll No, Name, Branch)

**Q** Find the details of all computer science students?

SQL: select \* from student where branch = CSE

RA:  $\{\sigma_{\text{branch} = \text{CSE}}(\text{Student})\}$

TRC:  $\{t \mid \text{Student}(t) \wedge t.\text{branch} = \text{CSE}\}$

DRC:



## Student(Roll No, Name, Branch)

Q Find the Roll No of all computer science students?

SQL: select Roll No from student where branch = CSE

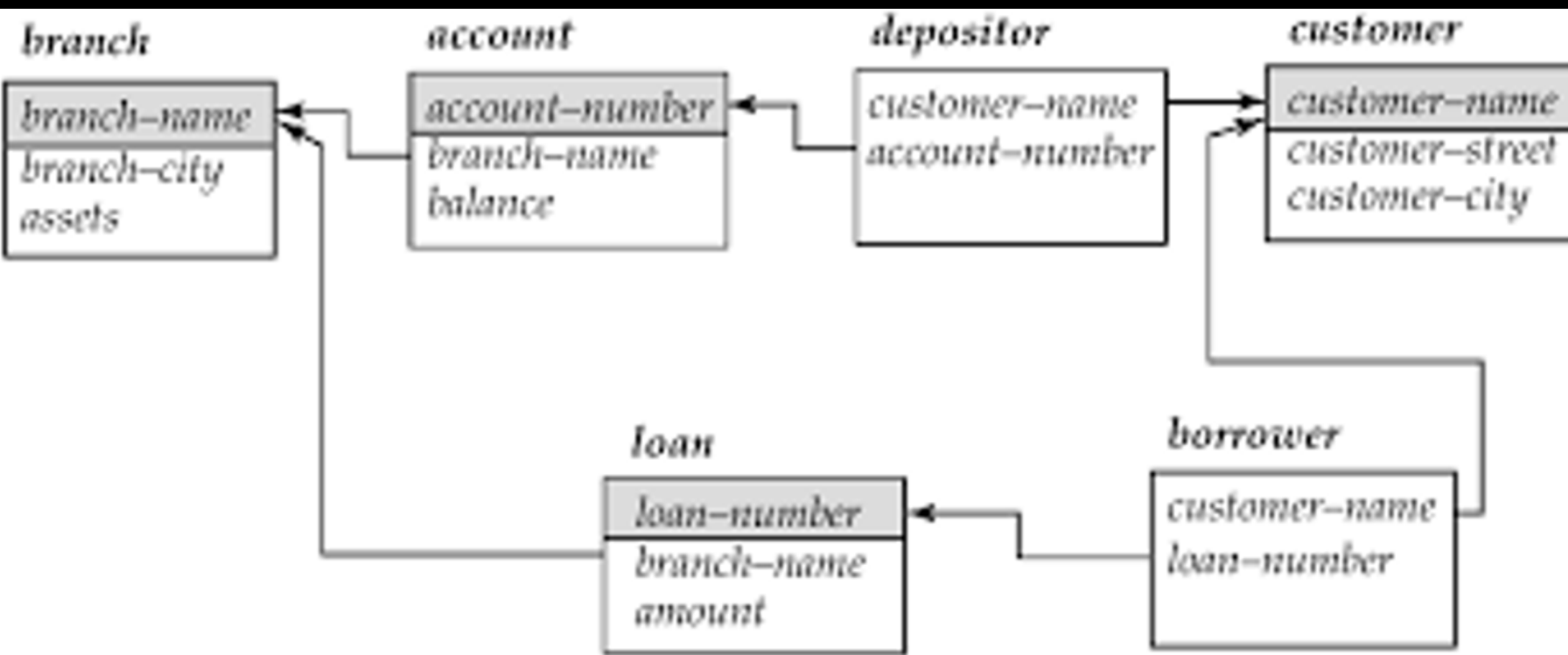
RA:  $\{\Pi_{\text{roll no}} (\sigma_{\text{branch} = \text{CSE}} (\text{Student}))\}$

TRC:  $\{t. \text{Roll No} \mid \text{Student}(t) \cap t. \text{branch} = \text{CSE}\}$

DRC:

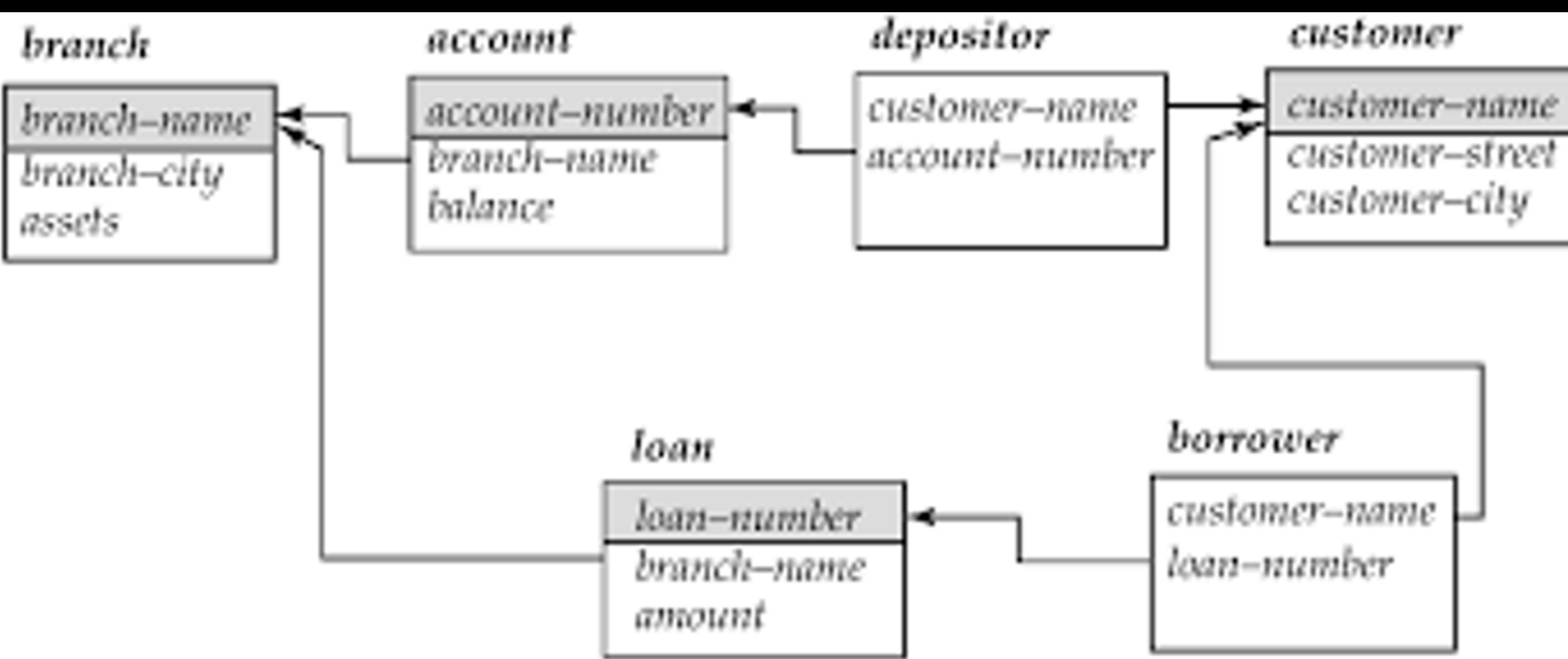
Q Find all the details of loan for amount over 1200?

$\{t \mid t \in \text{loan} \wedge t[\text{amount}] > 1200\}$



**Q Find the loan number for each loan of amount over 1200?**

$\{t \mid \exists s \in \text{loan} (t[\text{loan number}] = s[\text{loan number}] \wedge s[\text{amount}] > 1200) \}$

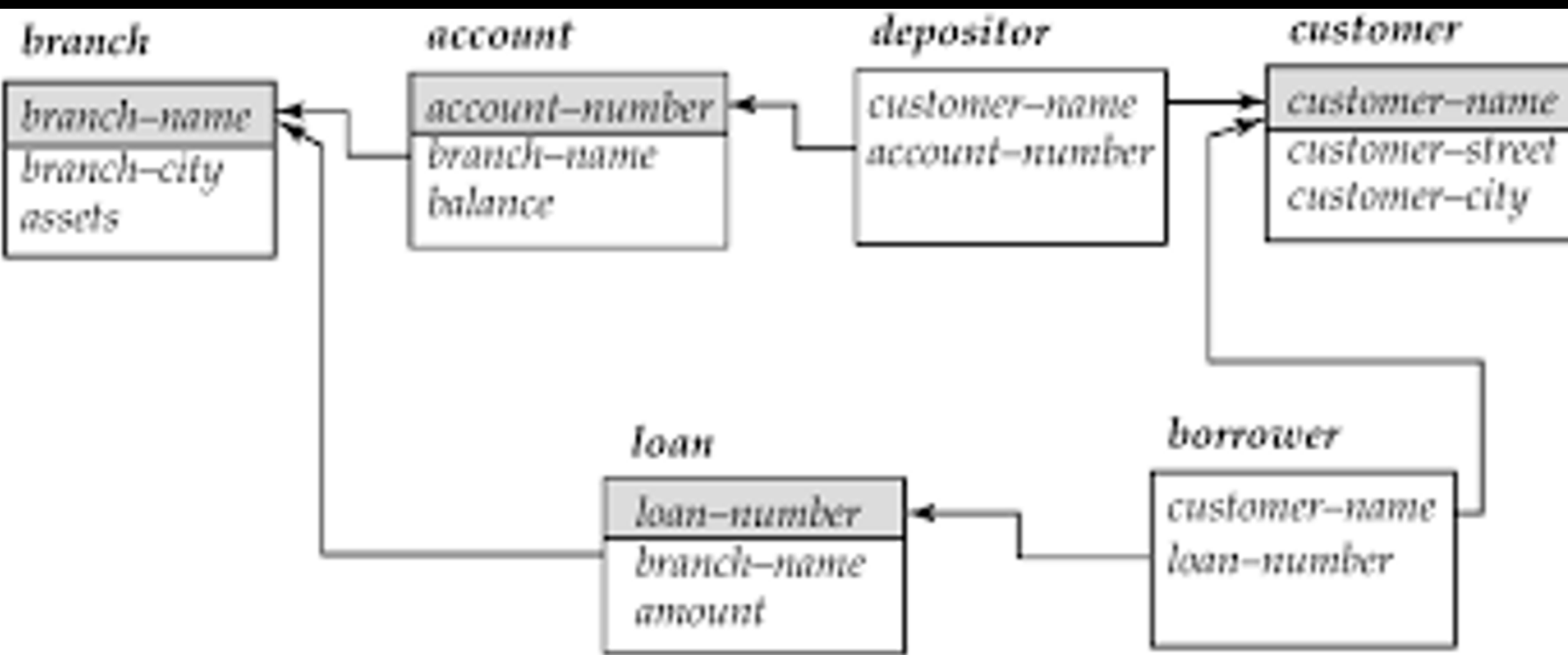


**Q Find the name of all the customers who have a loan from Noida branch?**

$\{t \mid \exists s \in \text{borrower} (t[\text{customer name}] = s[\text{customer name}])$

$\wedge \exists u \in \text{loan} (u[\text{customer name}] = s[\text{customer name}])$

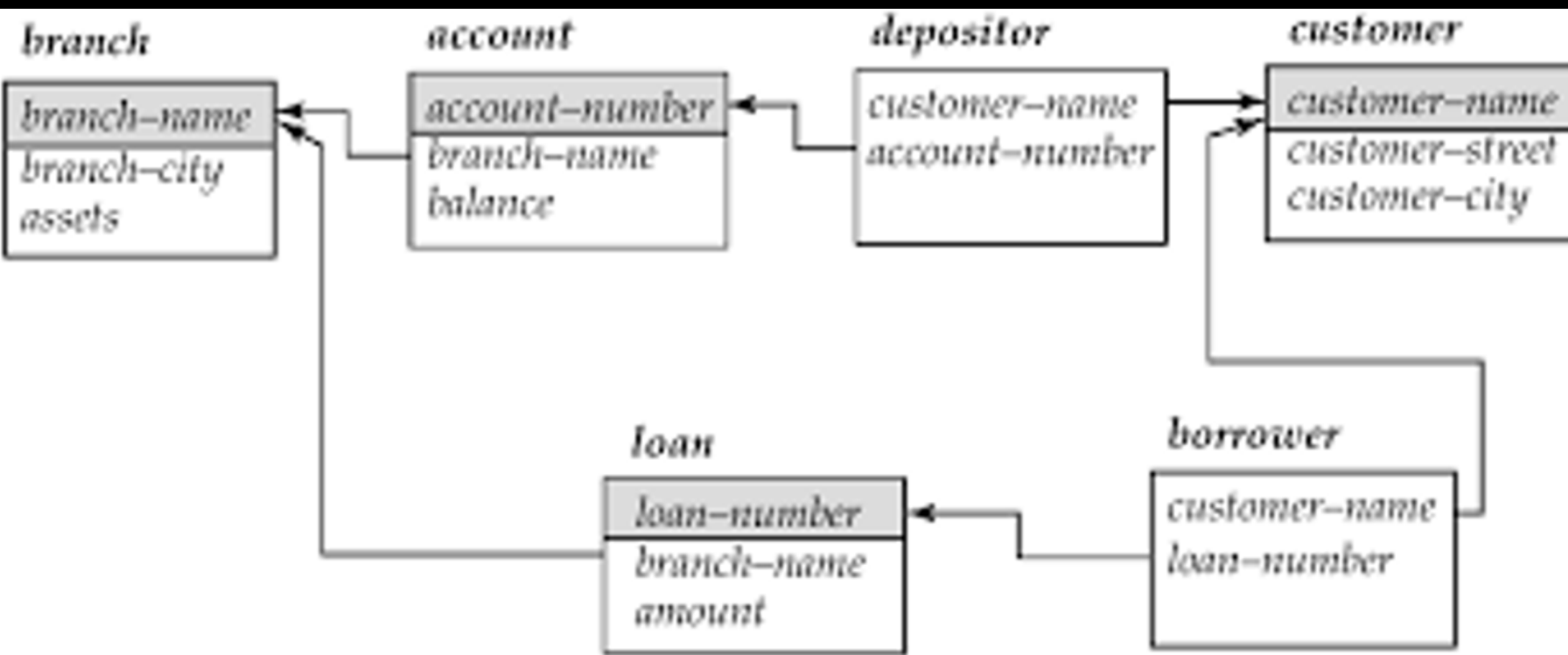
$\wedge u[\text{branch name}] = \text{'Noida'} \}$



**Q Find the name of all the customers who have a loan or account or both at the bank?**

$\{t \mid \exists s \in \text{borrower } (t[\text{customer name}] = s[\text{customer name}])$

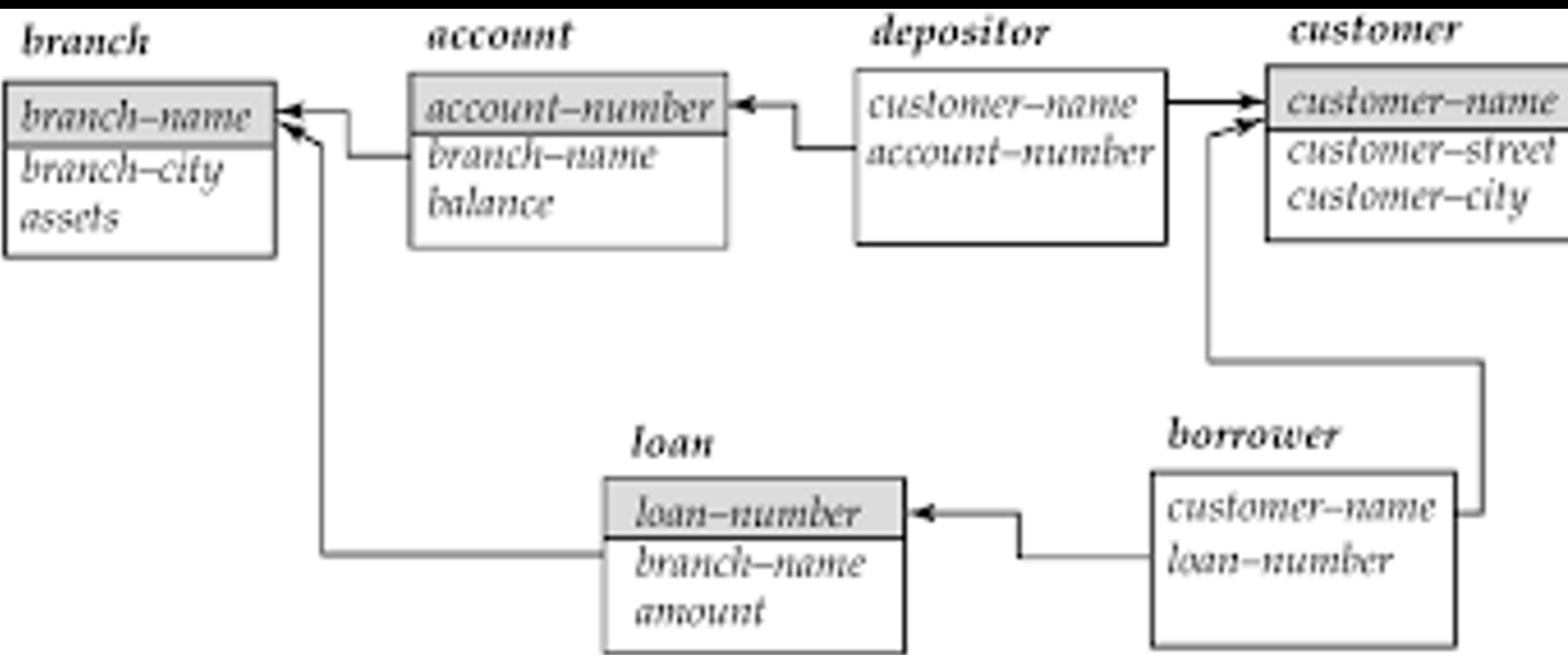
$\vee \exists u \in \text{depositor } (t[\text{customer name}] = u[\text{customer name}]) \}$



**Q** Find the name of all the customers who have a loan and account both at the bank?

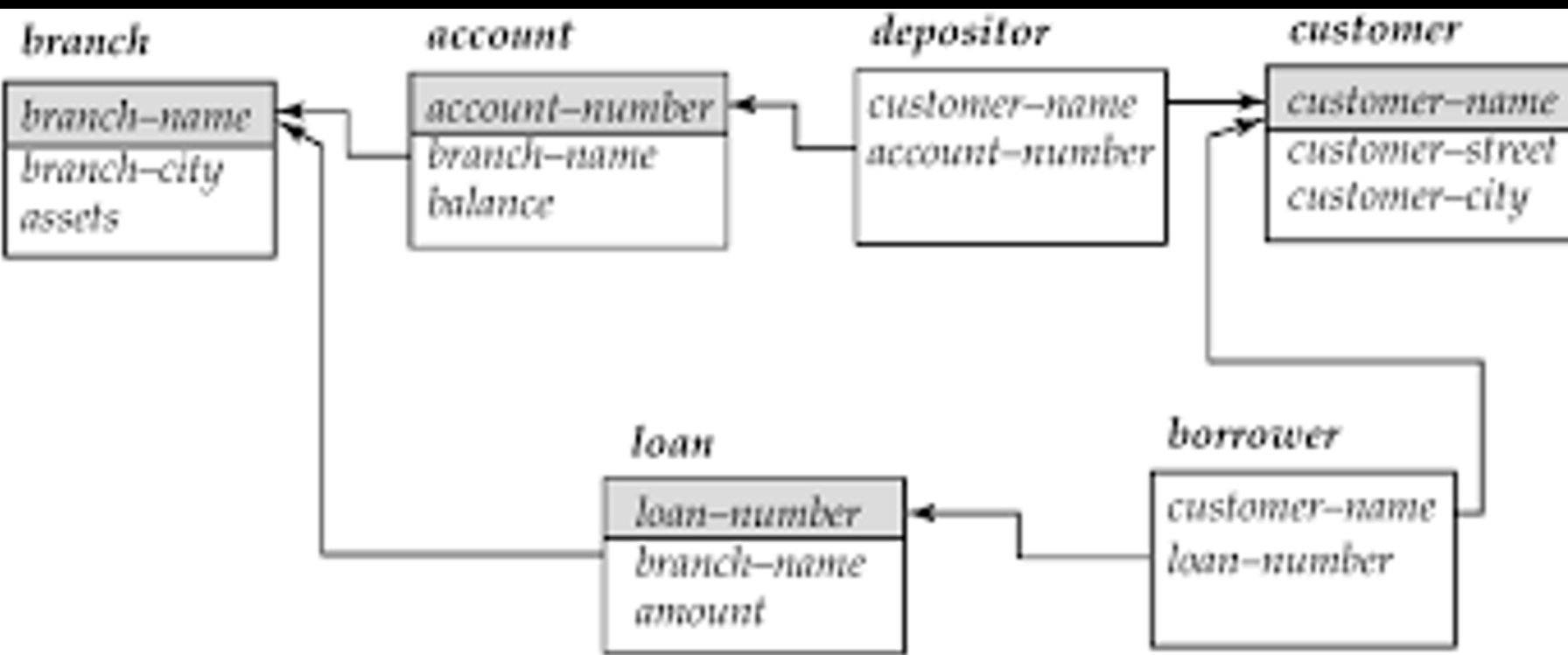
$\{t \mid \exists s \in \text{borrower} (t[\text{customer name}] = s[\text{customer name}])$

$\wedge \exists u \in \text{depositor} (t[\text{customer name}] = u[\text{customer name}]) \}$



**Q** Find the name of all the customers who have a loan from the bank and do not have a account?

$\{t \mid \exists u \in \text{depositor borrower } (t[\text{customer name}] = u[\text{customer name}])$   
 $\wedge \neg \exists s \in \text{borrower } (t[\text{customer name}] = s[\text{customer name}]) \}$









## Domain Relational Calculus

- Domain calculus differs from the tuple calculus in the type of variables used in formulas: rather than having variables range over tuples.
- The variable range over single values from domains of attributes. To form a relation of degree  $n$  for a query result, we must have  $n$  of these domain variables. One for each attribute.
- An expression of the domain calculus is of the form
- $(x_1, x_2, \dots, x_n \mid \text{COND}(x_1, x_2, \dots, x_n, x_{n+1}, x_{n+2}, \dots, x_{n+m}))$
- Where  $x_1, x_2, \dots, x_n$  are domain variables that range over domains and COND is a condition of the domain relational calculus.

## Student(Roll No, Name, Branch)

Q Find the details of all computer science students?

SQL: select \* from student where branch = CSE

RA:  $\{\sigma_{\text{branch} = \text{CSE}}(\text{Student})\}$

TRC:  $\{t \mid \text{Student}(t) \cap t.\text{branch} = \text{CSE}\}$

DRC:  $\{(\text{Roll No, Name, Branch}) \mid \text{Student}(\text{Roll no, Name, Branch}) \cap \text{branch} = \text{CSE}\}$

## Student(Roll No, Name, Branch)

Q Find the Roll No of all computer science students?

SQL: select Roll No from student where branch = CSE

RA:  $\{\Pi_{\text{sname}} (\sigma_{\text{branch} = \text{CSE}} (\text{Student}))\}$

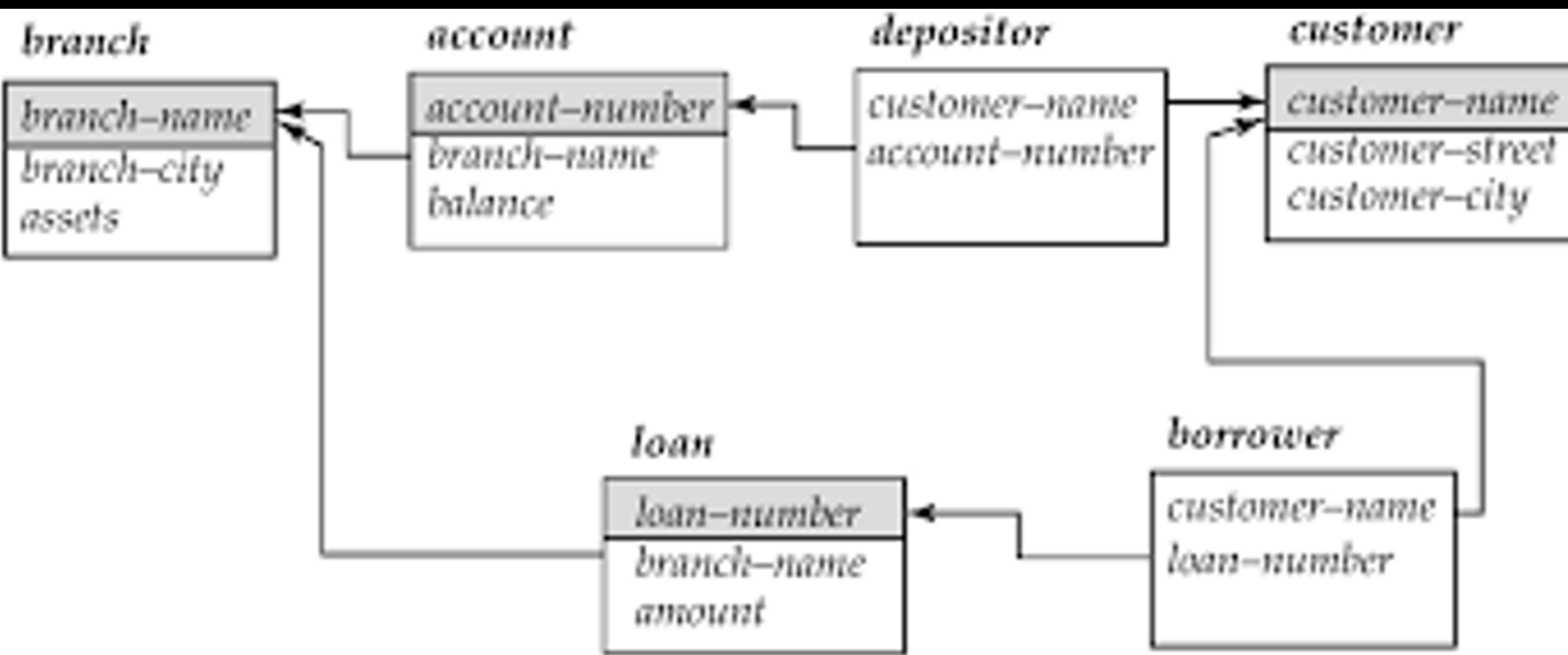
TRC:  $\{t. \text{Roll No} \mid \text{Student}(t) \cap t. \text{branch} = \text{CSE}\}$

DRC:  $\{(\text{Roll No}) \mid \text{Student}(\text{Roll no, Name, Branch}) (\text{Name, Branch}) \cap \text{branch} = \text{CSE}\}$

**Example:** Find all details of instructors whose salary is greater than \$80,000

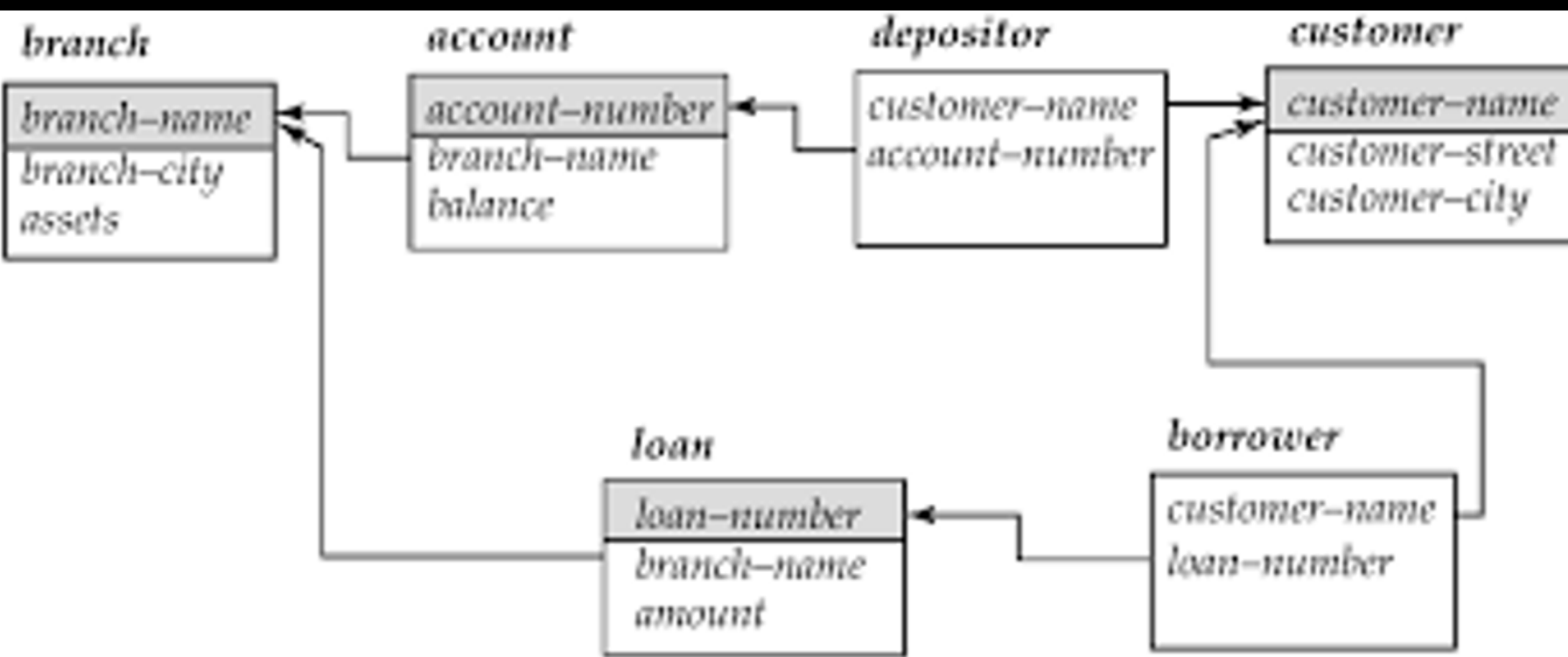
Q Find the branch name, loan number and amount for loan of amount over 1200?

$\{ \langle l, b, a \rangle \mid \langle l, b, a \rangle \in \text{loan} \wedge a > 1200 \}$



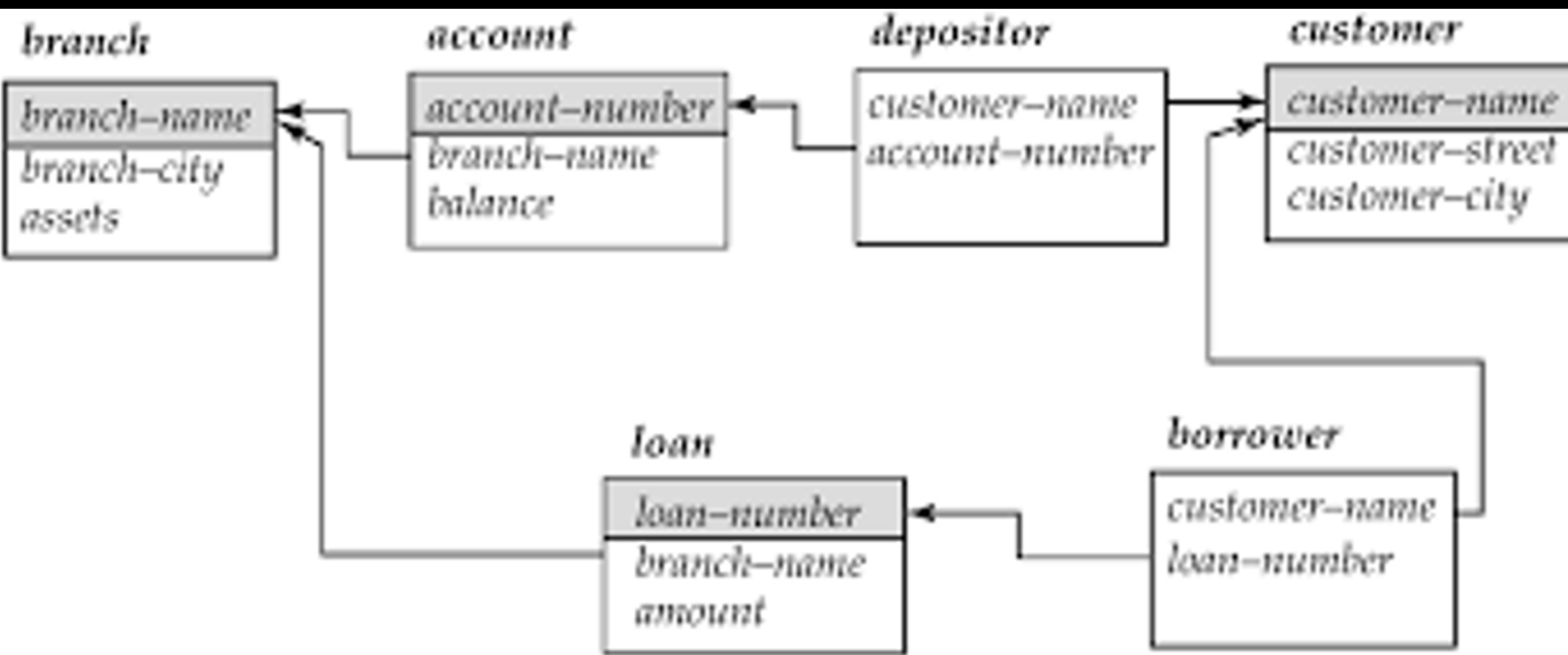
**Q** Find the loan number for each loan of amount over 1200?

$\{ \langle l \rangle \mid \exists_{b,a} \langle l, b, a \rangle \in \text{loan} \wedge a > 1200 \}$



Q Find the name of all the customers who have a loan from Noida branch with loan amount?

$\{ \langle c, a \rangle \mid \exists_l (\langle c, l \rangle \in \text{borrower} \wedge \exists_b (\langle l, b, a \rangle \in \text{loan} \wedge b = \text{'Noida'})) \}$

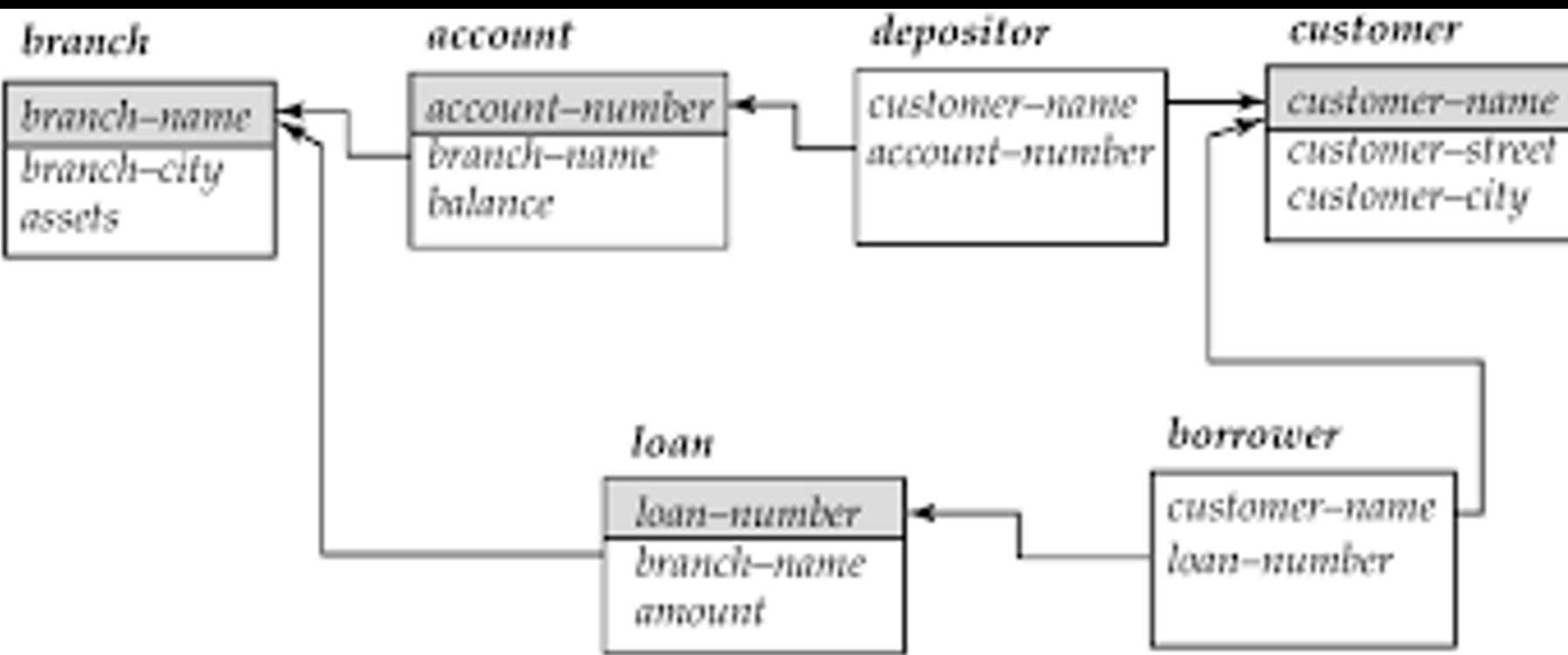




**Q** Find the name of all the customers who have a loan or account or both at the bank?

$\{ \langle c \rangle \mid \exists_l (\langle c, l \rangle \in \text{borrower} \wedge \exists_{b,a} (\langle l, b, a \rangle \in \text{loan} \wedge b = \text{'Noida'}))$

$\vee \exists_a (\langle c, a \rangle \in \text{depositor} \wedge \exists_{b,a} (\langle a, bn, bal \rangle \in \text{account} \wedge bn = \text{'Noida'})) \}$







# TRANSACTION

- Why we study transaction?
  - According to general computation principle (operating system) we may have partially executed program, as the level of atomicity is instruction i.e. either an instruction is executed completely or not
  - But in DBMS view, user perform a logical work(operation) which is always atomic in nature i.e. either operation is execute or not executed, there is no concept like partial execution. For example, Transaction  $T_1$  which transfer 100 units from account A to B.
  - In this transaction if a failure occurs after Read(B) then the final statue of the system will be inconsistent as 100 units are debited from account A but not credited in account B, this will generate inconsistency. Here for 'consistency' before  $(A + B) ==$  after  $(A + B)$ ".

$T_1$
<b>Read(A)</b>
<b>A = A-100</b>
<b>Write(A)</b>
<b>Read(B)</b>
<b>B = B+100</b>
<b>Write(B)</b>

# What is transaction

- To remove this partial execution problem, we increase the level of atomicity and bundle all the instructions of a logical operation into a unit called transaction.
- So formally 'A transaction is a Set of logically related instructions to perform a logical unit of work'.

$T_1$
<b>Read(A)</b>
<b><math>A = A - 100</math></b>
<b>Write(A)</b>
<b>Read(B)</b>
<b><math>B = B + 100</math></b>
<b>Write(B)</b>



- As here we are only concerned with DBMS so we will only two basic operations on database.
- **READ (X)** - Accessing the database item x from disk (where database stores data) to memory variable also named as X.
- **WRITE (X)** - Writing the data item from memory variable X to disk.

## Desirable properties of transaction

- Now as the smallest unit which have atomicity in DBMS view is transaction, so if want that our data should be consistent then instead of concentrating on data base, we must concentrate on the transaction for our data to be consistent.

<b><math>T_1</math></b>
<b>Read(A)</b>
<b><math>A = A - 100</math></b>
<b>Write(A)</b>
<b>Read(B)</b>
<b><math>B = B + 100</math></b>
<b>Write(B)</b>



- Transactions should possess several properties, often called the **ACID** properties; to provide integrity and consistency of the data in the database. The following are the ACID properties:

**A = Atomicity**

**C = Consistency**

**I = Isolation**

**D = Durability**



- **Atomicity** - A transaction is an atomic unit of processing; it should either be performed in its entirety or not performed at all.

$T_1$
Read(A)
$A = A - 100$
Write(A)
Read(B)
$B = B + 100$
Write(B)

- **Consistency** - A transaction should be consistency preserving, meaning that if it is completely executed from beginning to end without interference from other transactions, it should take the database from one consistent state to another.

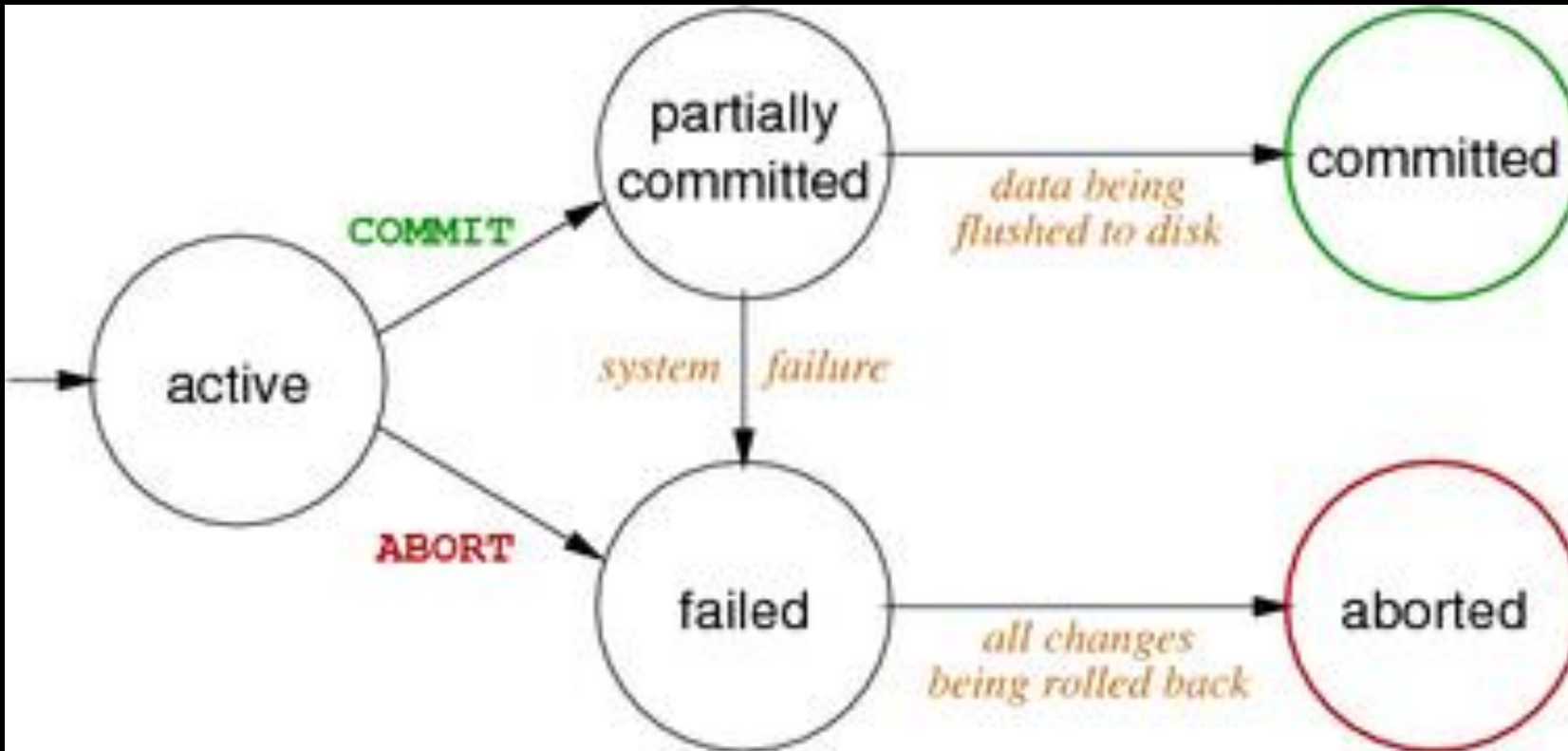


- **Isolation** - A transaction should appear as though it is being executed in isolation from other transactions, even though many transactions are executing concurrently.
- That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.

- **Durability** - The changes applied to the database by a committed transaction must persist in the database.

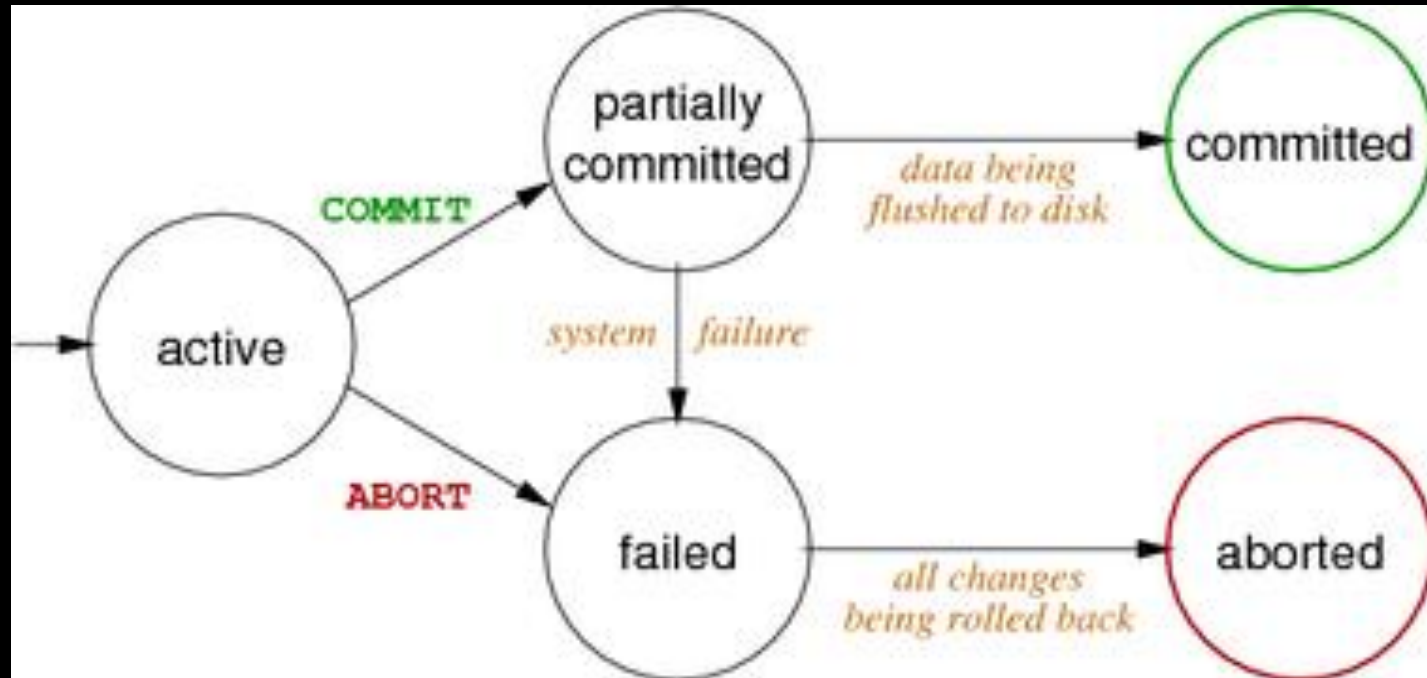
# Transaction states

- ACTIVE - It is the initial state. Transaction remains in this state while it is executing operations.



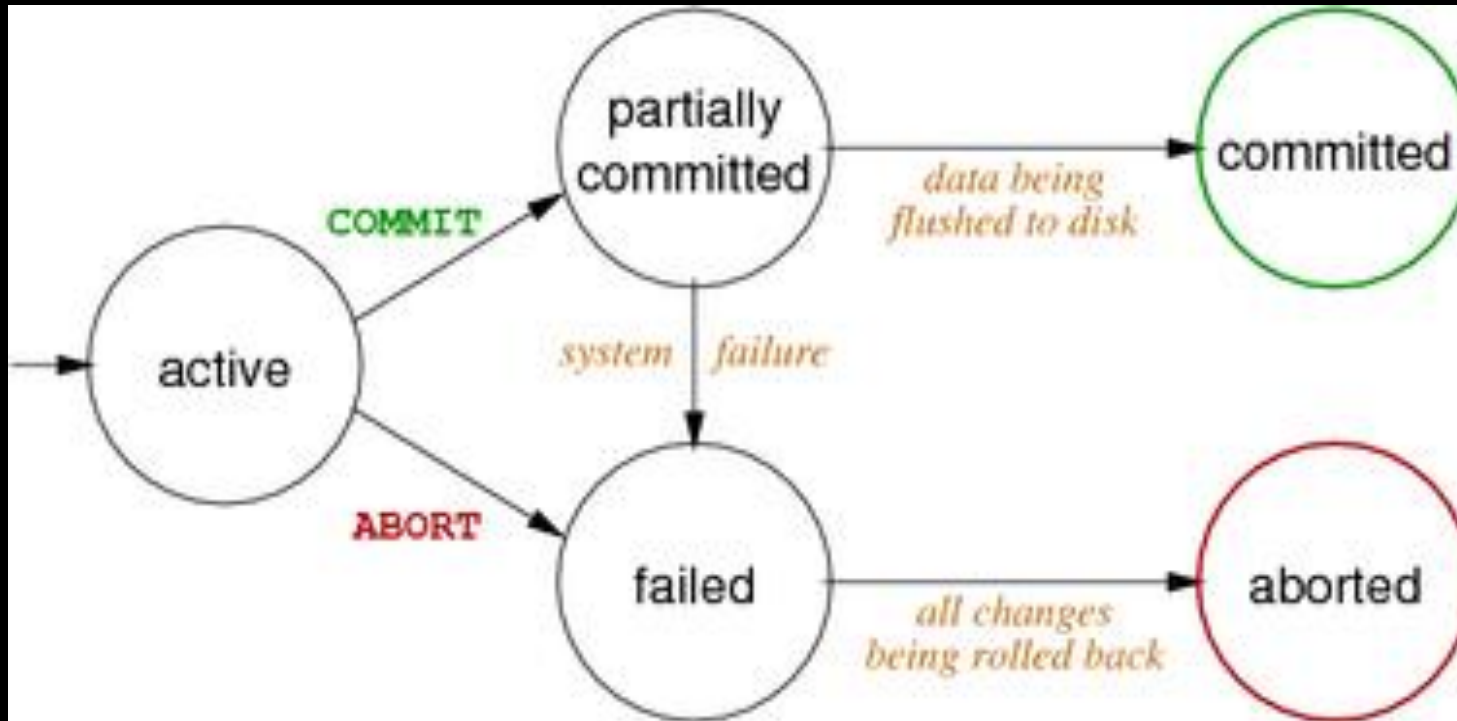
# Transaction states

- PARTIALLY COMMITTED - After the final statement of a transaction has been executed, the state of transaction is partially committed as it is still possible that it may have to be aborted (due to any failure) since the actual output may still be temporarily residing in main memory and not to disk.



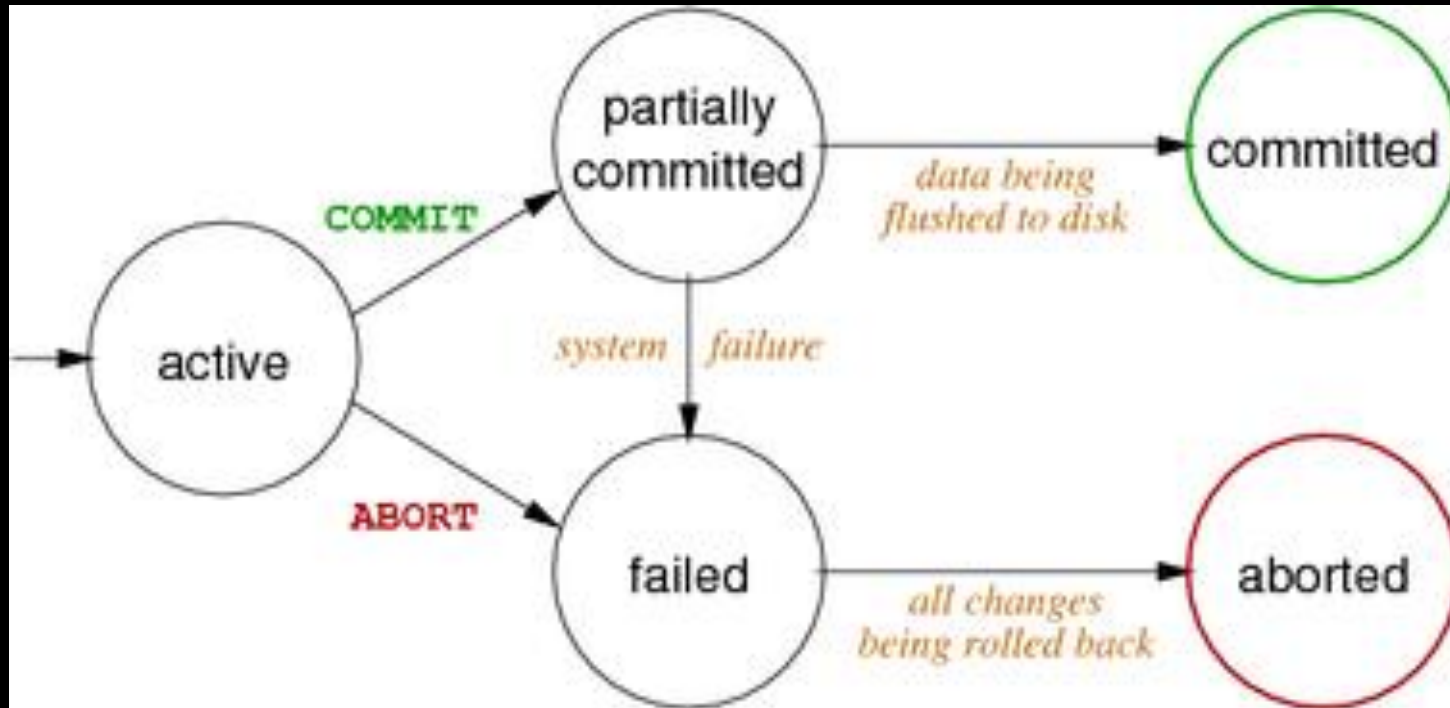
# Transaction states

- **FAILED** - After the discovery that the transaction can no longer proceed (because of hardware /logical errors). Such a transaction must be rolled back.



# Transaction states

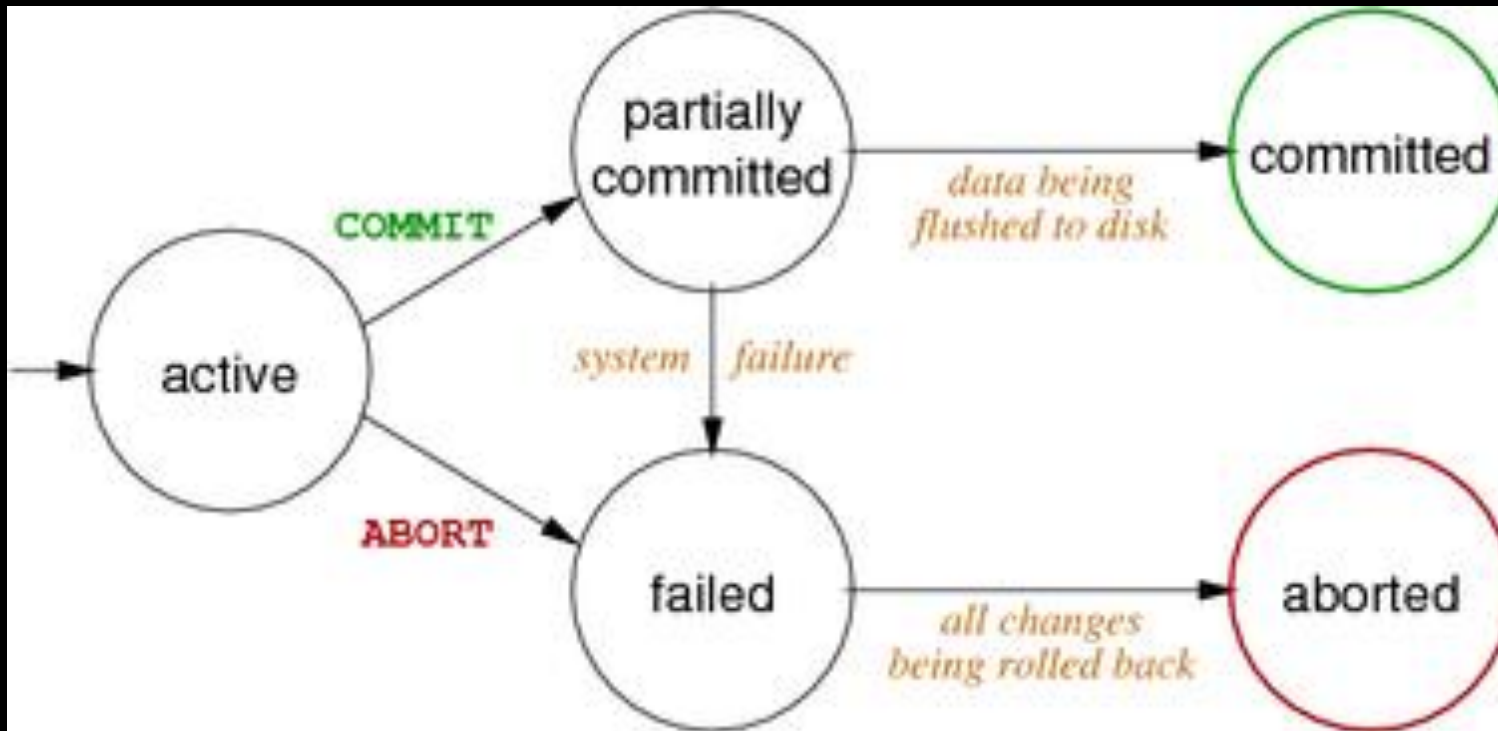
- **ABORTED** - A transaction is said to be in aborted state when the when the transaction has been rolled back and the database has been restored to its state prior to the start of execution.





# Transaction states

- COMMITTED - A transaction enters committed state after successful completion of a transaction and final updation in the database.



# Why we need concurrent execution

- Concurrent execution is necessary because-
  - It leads to good database performance , less weighting time.
  - Overlapping I/O activity with CPU increases throughput and response time.



## PROBLEMS DUE TO CONCURRENT EXECUTION OF TRANSACTION

- But interleaving of instructions between transactions may also lead to many problems that can lead to inconsistent database.
- Sometimes it is possible that even though individual transaction are satisfying the acid properties even though the final statues of the system will be inconsistent.

## Solution is Schedule

- When two or more transaction executed together or one after another then they can be bundled up into a higher unit of execution called schedule.

$T_0$	$T_1$
read( $A$ )	
write( $A$ )	
read( $B$ )	
write( $B$ )	
	read( $A$ )
	write( $A$ )
	read( $B$ )
	write( $B$ )

$T_2$	$T_3$
read( $B$ )	
	read( $B$ )
	write( $B$ )
read( $A$ )	
	read( $A$ )
	write( $A$ )

- Serial schedule - A serial schedule consists of sequence of instruction belonging to different transactions, where instructions belonging to one single transaction appear together. Before complete execution of one transaction another transaction cannot be started. Every serial schedule lead database into consistent state.

$T_0$	$T_1$
read( $A$ )	
write( $A$ )	
read( $B$ )	
write( $B$ )	
	read( $A$ )
	write( $A$ )
	read( $B$ )
	write( $B$ )

- **Non-serial schedule** - A schedule in which sequence of instructions of a transaction appear in the same order as they appear in individual transaction but the instructions may be interleaved with the instructions of different transactions i.e. concurrent execution of transactions takes place.

$T_2$	$T_3$
read( $B$ )	read( $B$ )
	write( $B$ )
read( $A$ )	read( $A$ )
	write( $A$ )

- **Conclusion of schedules**
  - We do not have any method to proof that a schedule is consistent, but from the above discussion we understand that a serial schedule will always be consistent.
  - So if somehow we proof that a non-serial schedule will also have same effects as of a serial schedule than we can proof that, this particular non-serial schedule will also be consistent.

On the basis of  
SERIALIZABILITY

```
graph TD; A[On the basis of SERIALIZABILITY] --- B[Conflict serializable]; A --- C[View serializable];
```

Conflict  
serializable

View  
serializable



T <sub>1</sub>	T <sub>2</sub>
R(A)	
R(B)	

T <sub>1</sub>	T <sub>2</sub>
	R(B)
R(A)	

T <sub>1</sub>	T <sub>2</sub>
R(A)	
W(A)	

T <sub>1</sub>	T <sub>2</sub>
	W(A)
R(A)	

T <sub>1</sub>	T <sub>2</sub>
R(A)	
W(B)	

T <sub>1</sub>	T <sub>2</sub>
	W(B)
R(A)	

T <sub>1</sub>	T <sub>2</sub>
	R(A)
W(A)	

T <sub>1</sub>	T <sub>2</sub>
W(A)	
R(A)	

T <sub>1</sub>	T <sub>2</sub>
R(A)	
R(A)	

T <sub>1</sub>	T <sub>2</sub>
	R(A)
R(A)	

T <sub>1</sub>	T <sub>2</sub>
	W(A)
W(A)	

T <sub>1</sub>	T <sub>2</sub>
W(A)	
W(A)	

**Conflict equivalent** – if one schedule can be converted to another schedule by swapping of non-conflicting instruction then they are called conflict equivalent schedule.

$T_1$	$T_2$
R(A)	
A=A-50	
	R(B)
	B=B+50
R(B)	
B=B+50	
	R(A)
	A=A+10

$T_1$	$T_2$
	R(B)
	B=B+50
R(A)	
A=A-50	
R(B)	
B=B+50	
	R(A)
	A=A+10

# SERIALIZABILITY

- **Conflicting instructions** - Let I and J be two consecutive instructions belonging to two different transactions  $T_i$  and  $T_j$  in a schedule S, the possible I and J instruction can be as-
  - I= READ(Q), J=READ(Q) ->*Non-conflicting*
  - I= READ(Q), J=WRITE(Q) ->*Conflicting*
  - I= WRITE(Q), J=READ(Q) ->*Conflicting*
  - I= WRITE(Q), J=WRITE(Q) ->*Conflicting*
- So, the instructions I and J are said to be conflicting, if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation.

## CONFLICT SERIALIZABLE

- The schedules which are conflict equivalent to a serial schedule are called conflict serializable schedule.
- If a schedule  $S$  can be transformed into a schedule  $S'$  by a series of swaps of non-conflicting instructions, we say that  $S$  and  $S'$  are conflict equivalent. A schedule  $S$  is **conflict serializable**, if it is conflict equivalent to a serial schedule.

$T_1$	$T_2$
read( $A$ ) write( $A$ )	read( $A$ ) write( $A$ )
read( $B$ ) write( $B$ )	read( $B$ ) write( $B$ )

$T_1$	$T_2$
read( $A$ ) write( $A$ ) read( $B$ ) write( $B$ )	read( $A$ ) write( $A$ ) read( $B$ ) write( $B$ )

**Q** Consider the following schedule for transactions  $T_1$ ,  $T_2$  and  $T_3$ : what is the correct serialization of the above?

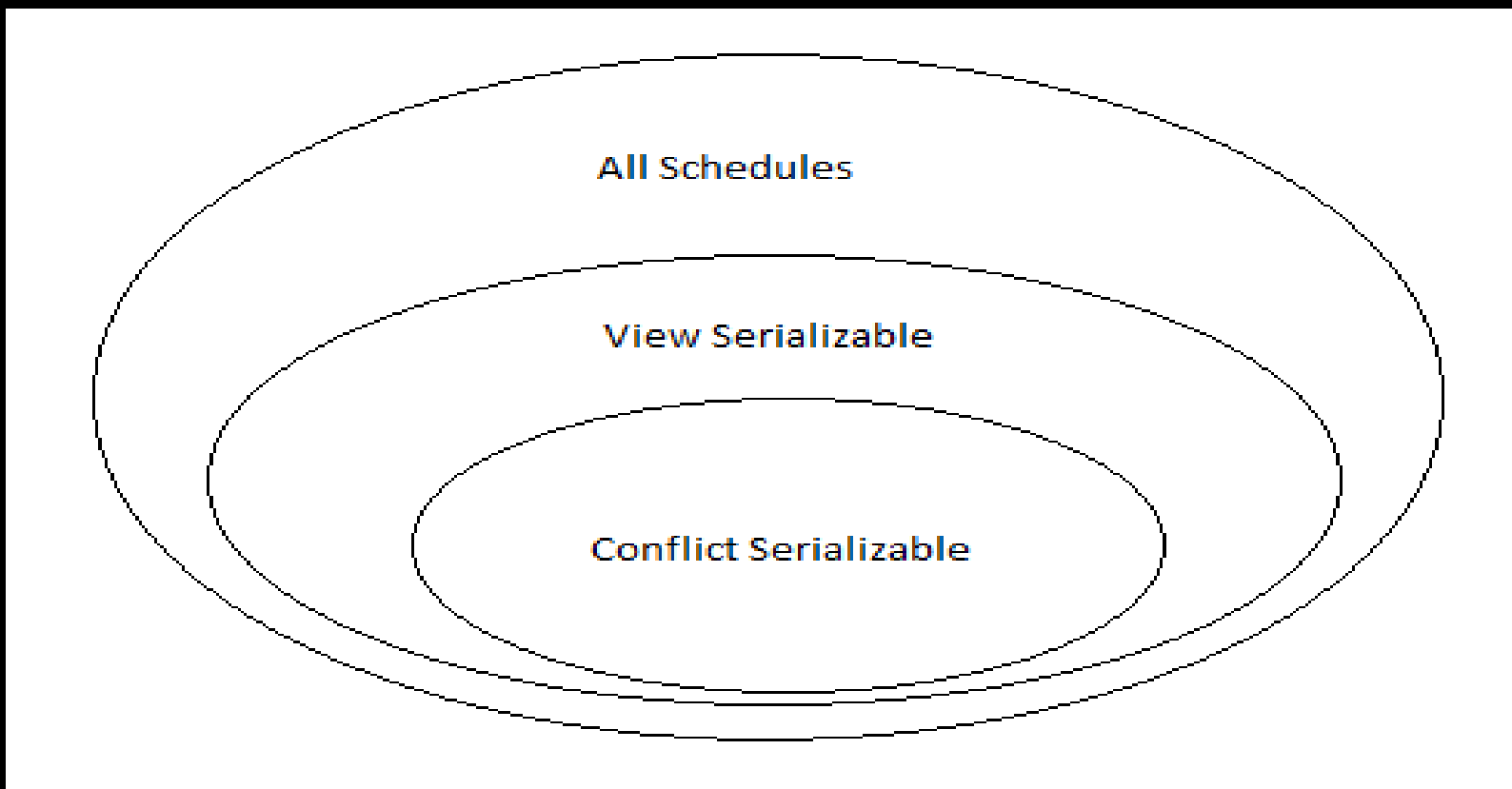
<u>T1</u>	<u>T2</u>	<u>T3</u>
Read ( X )		
	Read ( Y )	
		Read ( Y )
	Write ( Y )	
Write ( X )		
		Write ( X )
	Read ( X )	
	Write ( X )	

## Procedure for determining conflict serializability of a schedule

- It can be determined using PRECEDENCE GRAPH method:
- A precedence graph consists of a pair  $G (V, E)$
- $V$ = set of vertices consisting of all the transactions participating in the schedule.
- $E$ = set of edges consists of all edges  $T_i \rightarrow T_j$ , for which one of the following conditions holds:
  - $T_i$  executes write(Q) before  $T_j$  executes read(Q)
  - $T_i$  executes read(Q) before  $T_j$  executes write(Q)
  - $T_i$  executes write(Q) before  $T_j$  executes write(Q)
- If an edge  $T_i \rightarrow T_j$  exists in the precedence graph, then in any serial schedule  $S'$  equivalent to  $S$ ,  $T_i$  must appear before  $T_j$ .
- **If the precedence graph for  $S$  has no cycle, then schedule  $S$  is conflict serializable, else it is not.**

## VIEW SERIALIZABLE

- If a schedule is not conflict serializable, still it can be consistent, so let us study a weaker form of serializability called View serializability, and even if a schedule is view serializable still it can be consistent.
- If a schedule is conflict serializable then it will also be view serializable, so we must check view serializability only if a schedule is not conflict serializable.



- Two schedules S and S' are view equivalent, if they satisfy following conditions –
  - For each data item Q, if the transaction  $T_i$  reads the initial value of Q in schedule S, then the transaction  $T_i$  must, in schedule S', also read the initial value of Q.
  - If a transaction  $T_i$  in schedule S reads any data item Q, which is updated by transaction  $T_j$ , then a transaction  $T_i$  must in schedule S' also read data item Q updated by transaction  $T_j$  in schedule S'.
  - For each data item Q, the transaction (if any) that performs the final write(Q) operation in schedule S, then the same transaction must also perform final write(Q) in schedule S'.

$T_1$	$T_2$
read(A) write(A)	read(A) write(A)
read(B) write(B)	read(B) write(B)

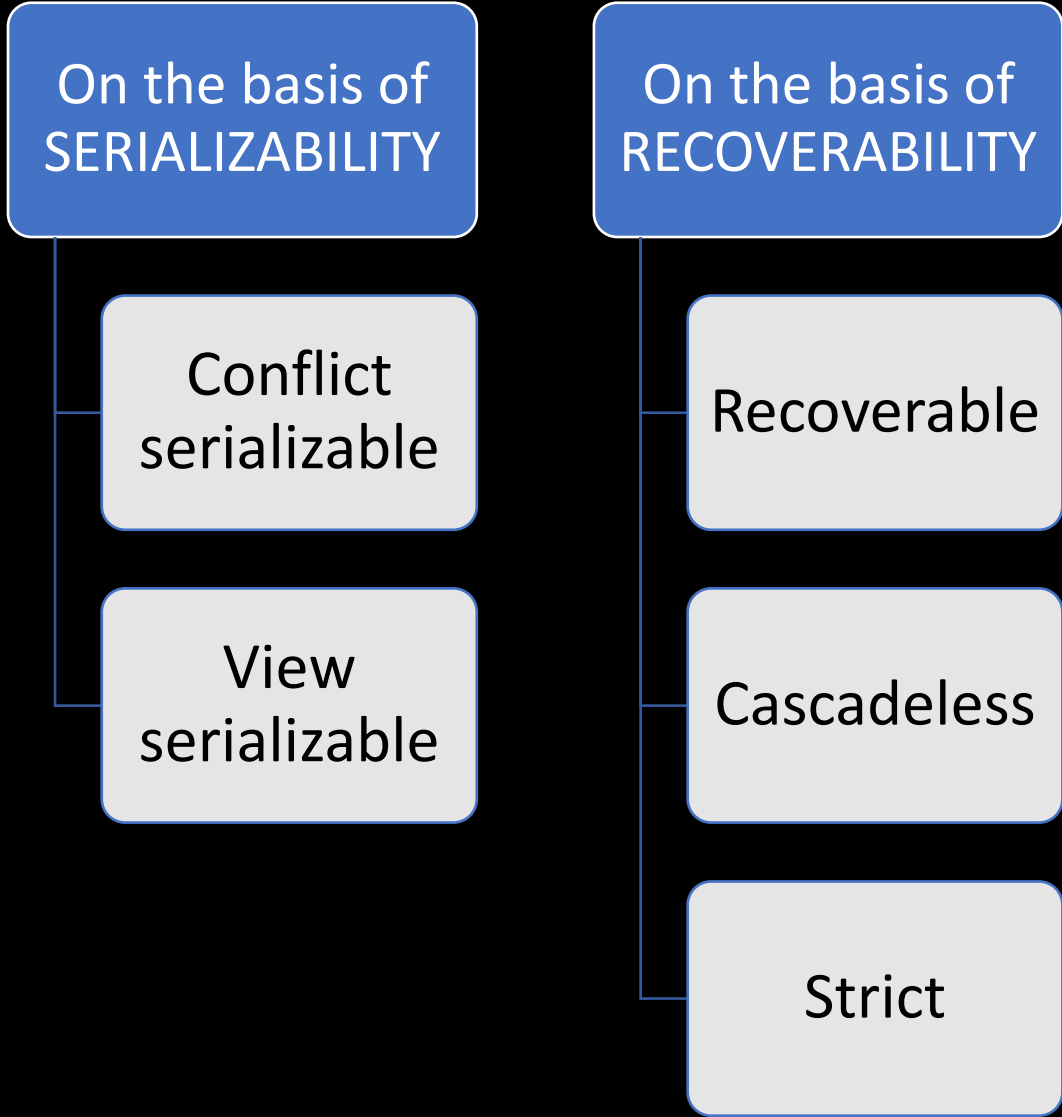
$T_1$	$T_2$
read(A) write(A) read(B) write(B)	read(A) write(A) read(B) write(B)



## View Serializable

A schedule S is view serializable, if it is view equivalent to a serial schedule.

Schedule A			Serial schedule <T3,T4,T6>		
T3	T4	T6	T3	T4	T6
read(Q)			read(Q)		
	write(Q)		write(Q)		
write(Q)				write(Q)	
		write(Q)			write(Q)



## NON- RECOVERABLE Vs RECOVERABLE SCHEDULE

- A schedule in which for each pair of transaction  $T_i$  and  $T_j$ , such that if  $T_j$  reads a data item previously written by  $T_i$ , then the commit or abort operation of  $T_i$  appears before  $T_j$ . Such a schedule is called Non- Recoverable schedule.
- A schedule in which for each pair of transaction  $T_i$  and  $T_j$ , such that if  $T_j$  reads a data item previously written by  $T_i$ , then the commit or abort of  $T_i$  must appear before  $T_j$ . Such a schedule is called Recoverable schedule.

S	
$T_1$	$T_2$
R(X)	
W(X)	
	R(X)
	C
C	

S	
$T_1$	$T_2$
R(X)	
W(X)	
	R(X)
C	
	C

## CASCADING ROLLBACK Vs CASCADELESS SCHEDULE

- It is a phenomenon, in which even if the schedule is recoverable, the a single transaction failure leads to a series of transaction rollbacks, is called cascading rollback. Cascading rollback is undesirable, since it leads to undoing of a significant amount of work. Uncommitted reads are not allowed in cascade less schedule.
- A schedule in which for each pair of transactions  $T_i$  and  $T_j$ , such that if  $T_j$  reads a data item previously written by  $T_i$  then the commit or abort of  $T_i$  must appear before read operation of  $T_j$ . Such a schedule is called cascade less schedule.

S		
$T_1$	$T_2$	$T_3$
R(X)		
W(X)		
	R(X)	
	W(X)	
		R(X)
C		
	C	
		C

S		
$T_1$	$T_2$	$T_3$
R(X)		
W(X)		
C		
	R(X)	
	W(X)	
	C	
		R(X)
		C

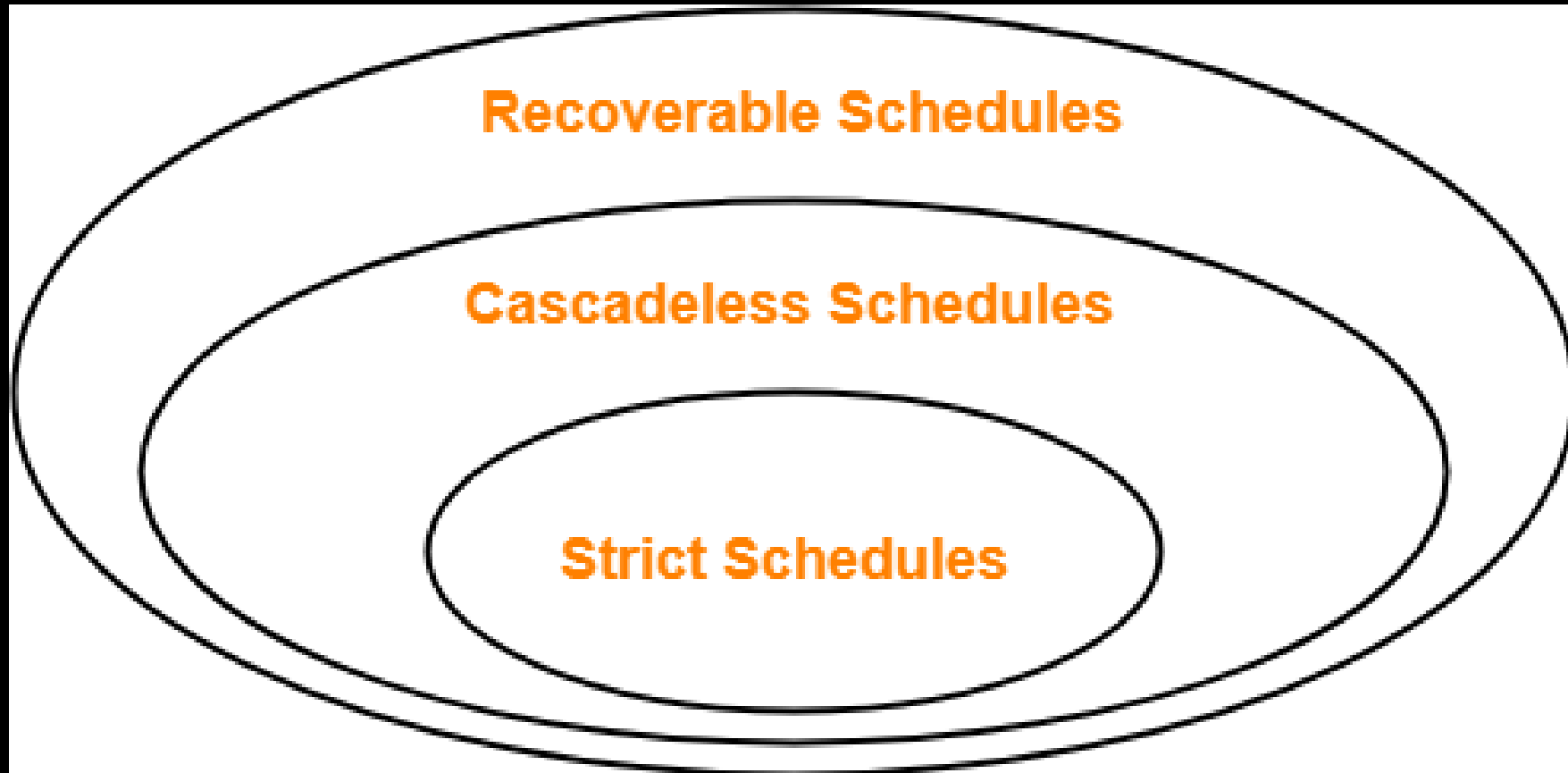
## Strict Schedule

- A schedule in which for each pair of transactions  $T_i$  and  $T_j$ , such that if  $T_j$  reads a data item previously written by  $T_i$  then the commit or abort of  $T_i$  must appear before read and write operation of  $T_j$ .

$S_1$	
$T_1$	$T_2$
R(a)	
W(a)	
	W(a)
C	
	R(a)
	C

$S_2$	
$T_1$	$T_2$
R(a)	
W(a)	
C	
	W(a)
	R(a)
	C

$S_3$	
$T_1$	$T_2$
R(a)	
	R(b)
W(a)	
	W(b)
C	
	R(a)
	C



# Log based Recovery

- The system log, or transaction log, records all the changes or activities happening in the database, ensuring that transactions are durable and can be recovered in the event of a failure.
- Different types of log records represent various stages or actions in a transaction.
  - $\langle T_i \text{ start} \rangle$ : This log entry indicates that the transaction  $T_i$  has started its execution.
  - $\langle T_i, X_j, V_1, V_2 \rangle$ : This log entry documents a write operation. It states that transaction  $T_i$  has changed the value of data item  $X_j$  from  $V_1$  to  $V_2$ .
  - $\langle T_i \text{ commit} \rangle$ : This log entry marks the successful completion of transaction  $T_i$ .
  - $\langle T_i \text{ abort} \rangle$ : This log entry denotes that the transaction  $T_i$  has been aborted, either due to an error or a rollback operation.
- Before any write operation modifies the database, a log record of that operation needs to be created. This is to ensure that in case of a failure, the system can restore the database to a consistent state using the log records.

<b>Aspect</b>	<b>Deferred Database Modification</b>	<b>Immediate Database Modification</b>
<b>Write Operation Timing</b>	Only at the commit point.	As soon as changes occur.
<b>I/O Operations</b>	Reduced, as changes are batched and written at once during the commit, saving on intermediate I/O operations.	Increased, as each change triggers immediate write operations, leading to more frequent I/O operations.
<b>Recovery Complexity</b>	Simpler, as uncommitted changes are not reflected in the database, making rollback easier in case of failures.  <a href="http://www.knowledgegate.com">Knowledge Gate Website</a>	More complex, as it may require undoing changes from uncommitted transactions that have been written to the database.



## Shadow Paging Recovery Technique

In the shadow paging recovery technique, the database maintains two page tables during a transaction: the current page table (reflecting the state before the transaction began) and the shadow page table (which tracks changes made during the transaction). Here's how it operates:

- **Initialization:** When a transaction begins, the database creates a shadow copy of the page table. The database pages themselves are not duplicated; only the page table entries are duplicated.
- **Modifications:** As the transaction progresses, any changes are reflected in the current page table, while the shadow page table retains the original state. If a page is modified, a new copy of the page is created, and the current page table is updated to point to this new version, thereby ensuring that the shadow page table still points to the original unmodified page.

- **Commit:** Upon transaction commit, the shadow page table is discarded, and the current page table becomes the new committed state of the database. The database atomically switches to using the current page table, ensuring that changes are installed all at once.
- **Recovery:** In case of a system failure before the transaction commits, the database can easily recover by discarding the current page table and reverting back to the shadow page table, thereby restoring the database to its state before the transaction began.

# Data fragmentation

- Data fragmentation in the context of a Database Management System (DBMS) refers to the process of breaking down a database into smaller, manageable pieces, and distributing these pieces across a network of computers. This is a significant component in distributed database systems.
  - **Horizontal Fragmentation**: In DBMS, horizontal fragmentation divides a table into sections based on rows. Each fragment contains a subset of rows, usually segmented based on certain conditions or attributes. This can enhance performance by facilitating parallel processing and quicker data retrieval, especially useful in distributed database systems.
  - **Vertical Fragmentation**: This fragmentation type divides a database table by columns, with different sets of columns stored in separate fragments. This approach is employed when different users require access to varied data attributes, promoting efficient data handling and minimizing data transfer times, which is advantageous in scenarios where queries only necessitate a subset of attributes.
  - **Hybrid Fragmentation**: Hybrid or mixed fragmentation in DBMS combines both horizontal and vertical strategies, optimizing data storage and retrieval for complex access patterns. This method can potentially offer performance improvements by utilizing the merits of both horizontal and vertical fragmentation, and is typically implemented in databases with complex, multifaceted data access requirements.

## Distributed database

- A distributed database is a database in which storage devices are not all attached to a common processor. It may be stored in multiple computers, located in the same physical location, or dispersed over a network of interconnected computers.
- **Data Replication**
  - *Advantages:* Increased Availability, Improved Performance, Enhanced Reliability
  - *Disadvantages:* Storage Costs, Maintenance Complexity, Write Complexity.
- **Data Fragmentation**
  - *Advantages:* Efficient Data Access, Distributed Processing, Localized Management.
  - *Disadvantages:* Complexity, Dependency on Network, Reconstruction Issues.

## CONCURRENCY CONTROL

Here we will study those protocols which guarantee to generate schedules which always satisfy desirable properties like conflict serializability. Along with we desire the following properties from schedule generating protocols

- Concurrency should be as high as possible, as this is our ultimate goal because of which we are making all the effort.
- The time taken by a transaction should also be less.
- Easy to understand and implement.

- **Time stamping based method:** - Where before entering the system, a specific order is decided among the transaction, so in case of a clash we can decide which one to allow and which to stop.
- **Lock based method:** - where we ask a transaction to first lock a data item before using it. So that no different transaction can use a data at the same time, removing any possibility of conflict.
  - 2 phase locking
    1. Basic 2pl
    2. Conservative 2pl
    3. Rigorous 2pl
    4. Strict 2pl
  - Graph based protocol
- **Validation based protocol** – Majority of transactions are read only transactions, the rate of conflicts among the transaction may be low, thus many of transaction, if executed without the supervision of a concurrency control scheme, would nevertheless leave the system in a consistent state.

## TIME STAMP ORDERING PROTOCOL

- Basic idea of time stamping is to decide the order between the transaction before they enter in the system using a stamp (time stamp), in case of any conflict during the execution order can be decided using the time stamp.
- Let's understand how this protocol works, here we have two idea of timestamping, one for the transaction, and other for the data item.

- Time stamp for transaction,
  - With each transaction  $t_i$ , in the system, we associate a unique fixed timestamp, denoted by  $TS(t_i)$ .
  - This timestamp is assigned by database system to a transaction at time transaction enters into the system.
  - If a transaction has been assigned a timestamp  $TS(t_i)$  and a new transaction  $t_j$ , enters into the system with a timestamp  $TS(t_j)$ , then always  $TS(t_i) < TS(t_j)$ .



- Two things are to be noted
  1. First time stamp of a transaction remain fixed throughout the execution
  2. Second it is unique means no two transaction can have the same timestamp.

- Time stamp with data item, in order to assure such scheme, the protocol maintains for each data item Q two timestamp values:
  1. W-timestamp(Q) is the largest time-stamp of any transaction that executed write(Q) successfully.
  2. R-timestamp(Q) is the largest time-stamp of any transaction that executed read(Q) successfully.
- These timestamps are updated whenever a new read(Q) or write(Q) instruction is executed.

- Suppose a transaction  $T_i$  request a *read(Q)*
  1. If  $TS(T_i) < W\text{-timestamp}(Q)$ , then  $T_i$  needs to read a value of  $Q$  that was already overwritten. Hence, the read operation is rejected, and  $T_i$  is rolled back.
  2. If  $TS(T_i) \geq W\text{-timestamp}(Q)$ , then the read operation is executed, and  $R\text{-timestamp}(Q)$  is set to the maximum of  $R\text{-timestamp}(Q)$  and  $TS(T_i)$ .

- Suppose that transaction  $T_i$  issues *write(Q)*.
  1. If  $TS(T_i) < R\text{-timestamp}(Q)$ , then the value of  $Q$  that  $T_i$  is producing was needed previously, and the system assumed that that value would never be produced. Hence, the write operation is rejected, and  $T_i$  is rolled back.
  2. If  $TS(T_i) < W\text{-timestamp}(Q)$ , then  $T_i$  is attempting to write an obsolete value of  $Q$ . Hence, this write operation is rejected, and  $T_i$  is rolled back.
  3. If  $TS(T_i) \geq R\text{-timestamp}(Q)$ , then the write operation is executed, and  $W\text{-timestamp}(Q)$  is set to  $\max(W\text{-timestamp}(Q), TS(T_i))$ .
  4. If  $TS(T_i) \geq W\text{-timestamp}(Q)$ , then the write operation is executed, and  $W\text{-timestamp}(Q)$  is set to  $\max(W\text{-timestamp}(Q), TS(T_i))$ .

	Conflict Serializability	View Serializability	Recoverability	Cascadelessness	Deadlock Freedom
Time Stamp Ordering	<b>YES</b>	<b>YES</b>	<b>NO</b>	<b>NO</b>	<b>YES</b>
Thomas Write Rule					
Basic 2PL					
Conservative 2PL					
Rigorous 2PL					
Strict 2PL					

## THOMAS WRITE RULE

- Thomas write is an improvement in time stamping protocol, which makes some modification and may generate those protocols that are even view serializable, because it allows greater potential concurrency.
- It is a Modified version of the timestamp-ordering protocol in which Blind write operations may be ignored under certain circumstances.
- The protocol rules for read operations remain unchanged. while for write operation, there is slightly change in Thomas write rule than timestamp ordering protocol.

**When  $T_i$  attempts to write data item  $Q$ ,**

- **if  $TS(T_i) < W\text{-timestamp}(Q)$ , then  $T_i$  is attempting to write an obsolete value of  $\{Q\}$ . Rather than rolling back  $T_i$  as the timestamp ordering protocol would have done, this {write} operation can be ignored.**

- This modification is valid as the any transaction with  $TS(T_i) < W\text{-timestamp}(Q)$ , the value written by this transaction will never be read by any other transaction performing  $Read(Q)$  ignoring such obsolete write operation is considerable.
- Thomas' Write Rule allows greater potential concurrency. Allows some view-serializable schedules that are not conflict serializable.

$T_3$	$T_4$	$T_6$
read(Q)	write(Q)	
write(Q)		
		write(Q)



	Conflict Serializability	View Serializability	Recoverability	Cascadelessness	Deadlock Freedom
Time Stamp Ordering	<b>YES</b>	<b>YES</b>	<b>NO</b>	<b>NO</b>	<b>YES</b>
Thomas Write Rule	<b>NO</b>	<b>YES</b>	<b>NO</b>	<b>NO</b>	<b>YES</b>
Basic 2PL					
Conservative 2PL					
Rigorous 2PL					
Strict 2PL					

## Lock Based Protocols

- To ensure isolation is to require that data items be accessed in a mutually exclusive manner i.e. while one transaction is accessing a data item, no other transaction can modify that data item. Locking is the most fundamental approach to ensure this.
- Lock based protocols ensure this requirement. Idea is first obtain a lock on the desired data item then if lock is granted then perform the operation and then unlock it.

- In general, we support two modes of lock because, to provide better concurrency.
- **Shared mode**
  - If transaction  $T_i$  has obtained a shared-mode lock (denoted by S) on any data item Q, then  $T_i$  can read, but cannot write Q, any other transaction can also acquire a shared mode lock on the same data item (this is the reason we called this shared mode).
- **Exclusive mode**
  - If transaction  $T_i$  has obtained an exclusive-mode lock (denoted by X) on any data item Q, then  $T_i$  can both read and write Q, any other transaction cannot acquire either a shared or exclusive mode lock on the same data item. (this is the reason we called this exclusive mode)

## Lock –Compatibility Matrix

- Conclusion shared is compatible only with shared while exclusive is not compatible either with shared or exclusive.
- To access a data item, transaction  $T_i$  must first lock that item, if the data item is already locked by another transaction in an incompatible mode, or some other transaction is already waiting in non-compatible mode, then concurrency control manager will not grant the lock until all incompatible locks held by other transactions have been released. The lock is then granted.

**Current State of lock of data items**

		Exclusive	Shared	Unlocked
Requested Lock	Exclusive	N	N	Y
	Shared	N	Y	Y
	Unlock	Y	Y	-

[Knowledge Gate Website](#)

- Lock based protocol do not ensure serializability as granting and releasing of lock do not follow any order and any transaction any time may go for lock and unlock. Here in the example below we can see, that even this transaction in using locking but neither it is conflict serializable nor independent from deadlock.

T <sub>1</sub>	T <sub>2</sub>
LOCK-X(A)	
READ(A)	
WRITE(A)	
UNLOCK(A)	
	LOCK-S(B)
	READ(B)
	UNLOCK(B)
LOCK-X(B)	
READ(B)	
WRITE(B)	
UNLOCK(B)	
	LOCK-S(A)
	READ(A)
	UNLOCK(A)

- If we do not use locking, or if we unlock data items too soon after reading or writing them, we may get inconsistent states, as there exists a possibility of dirty read. On the other hand, if we do not unlock a data item before requesting a lock on another data item, concurrency will be poor.
- We shall require that each transaction in the system follow a set of rules, called a **locking protocol**, indicating when a transaction may lock and unlock each of the data items for e.g. 2pl or graph based locking.
- Locking protocols restrict the number of possible schedules.

## Two phase locking protocol(2PL)

- The protocol ensures that each transaction issue lock and unlock requests in two phases, note that each transaction will be 2 phased not schedule.
- Growing phase- A transaction may obtain locks, but not release any locks.
- Shrinking phase- A transaction may release locks, but may not obtain any new locks.

- Initially a transaction is in growing phase and acquires lock as needed and in between can perform operation reach to lock point and once a transaction releases a lock, it can issue no more lock requests i.e. it enters the shrinking phase.

T <sub>1</sub>	T <sub>2</sub>
LOCK-X(A)	
READ(A)	
WRITE(A)	
	LOCK-S(B)
	READ(B)
LOCK-X(B)	
READ(B)	
WRITE(B)	
	LOCK-S(A)
	READ(A)
	UNLOCK(B)
UNLOCK(A)	
UNLOCK(B)	
	UNLOCK(A)



	Conflict Serializability	View Serializability	Recoverability	Cascadelessness	Deadlock Freedom
Time Stamp Ordering	<b>YES</b>	<b>YES</b>	<b>NO</b>	<b>NO</b>	<b>YES</b>
Thomas Write Rule	<b>NO</b>	<b>YES</b>	<b>NO</b>	<b>NO</b>	<b>YES</b>
Basic 2PL	<b>YES</b>	<b>YES</b>	<b>NO</b>	<b>NO</b>	<b>NO</b>
Conservative 2PL					
Rigorous 2PL					
Strict 2PL					

## Properties

- 2PL ensures conflict serializability, and the ordering of transaction over lock points is itself a serializability order of a schedule in 2PL.
- If a schedule is allowed in 2PL protocol then definitely it is always conflict serializable. But it is not necessary that if a schedule is conflict serializable then it will be generated by 2pl. Equivalent serial schedule is based on the order of lock points.
- View serializability is also guaranteed.
- Does not ensure freedom from deadlock
- May cause non-recoverability.
- Cascading rollback may occur.

## Conservative 2PL

- The idea is there is no growing phase transaction start directly from lock point, i.e. transaction must first acquire all the required locks then only it can start execution. If all the locks are not available then transaction must release the acquired locks and must wait.
  - Shrinking phase will work as usual, and transaction can unlock any data item anytime.
  - we must have a knowledge in future to understand what is data required so that we can use it

	Conflict Serializability	View Serializability	Recoverability	Cascadelessness	Deadlock Freedom
Time Stamp Ordering	YES	YES	NO	NO	YES
Thomas Write Rule	NO	YES	NO	NO	YES
Basic 2PL	YES	YES	NO	NO	NO
Conservative 2PL	YES	YES	NO	NO	YES
Rigorous 2PL					
Strict 2PL					

## RIGOROUS 2PL

- Requires that all locks be held until the transaction commits.
- This protocol requires that locking be two phase and also all the locks taken be held by transaction until that transaction commit.
- Hence there is no shrinking phase in the system.

	Conflict Serializability	View Serializability	Recoverability	Cascadelessness	Deadlock Freedom
Time Stamp Ordering	YES	YES	NO	NO	YES
Thomas Write Rule	NO	YES	NO	NO	YES
Basic 2PL	YES	YES	NO	NO	NO
Conservative 2PL	YES	YES	NO	NO	YES
Rigorous 2PL	YES	YES	YES	YES	NO
Strict 2PL					

## STRICT 2PL

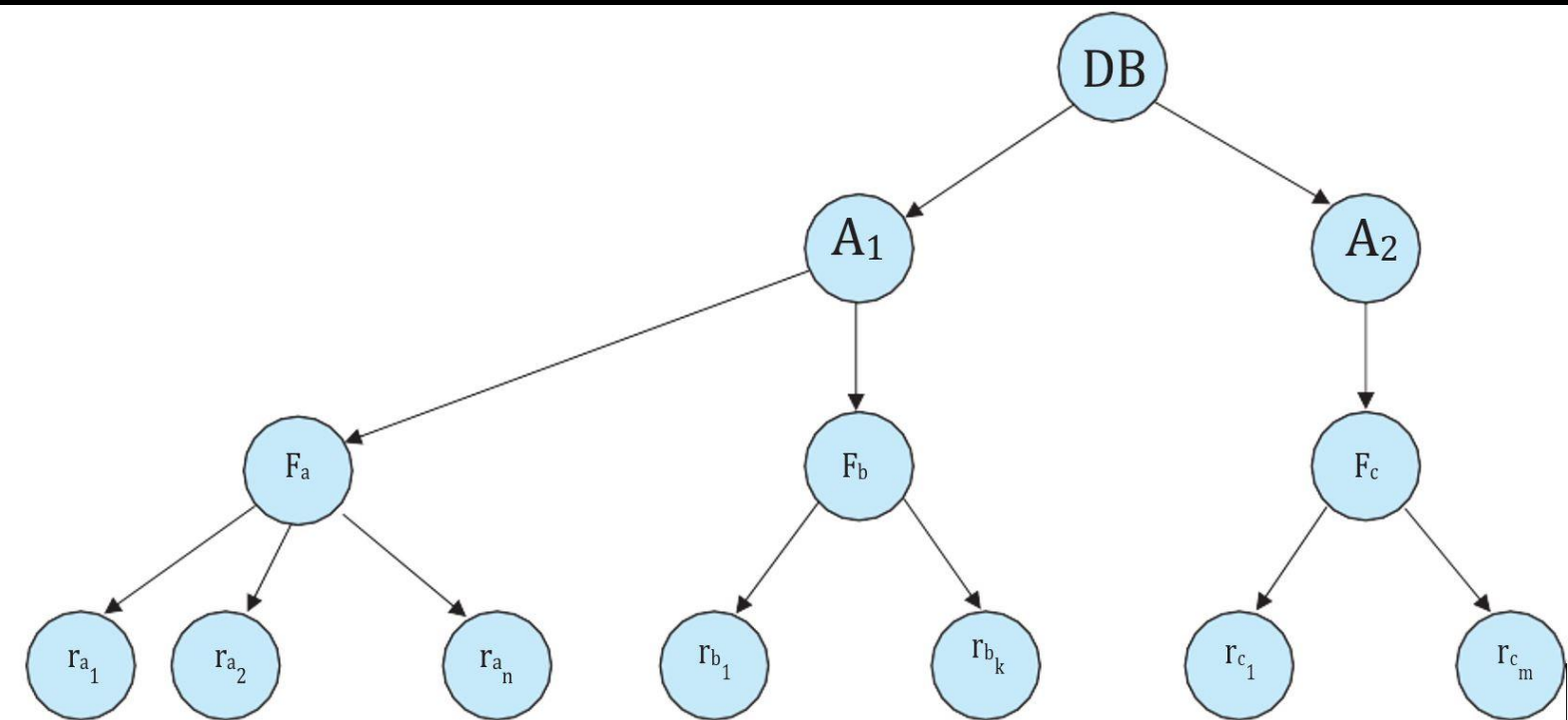
- that all exclusive-mode locks taken by a transaction be held until that transaction commits. This requirement ensures that any data written by an uncommitted transaction are locked in exclusive mode until the transaction commits, preventing any other transaction from reading the data.
- This protocol requires that locking be two phase and also that exclusive –mode locks taken by transaction be held until that transaction commits.
- So it is simplified form of rigorous 2pl

	Conflict Serializability	View Serializability	Recoverability	Cascadelessness	Deadlock Freedom
Time Stamp Ordering	YES	YES	NO	NO	YES
Thomas Write Rule	NO	YES	NO	NO	YES
Basic 2PL	YES	YES	NO	NO	NO
Conservative 2PL	YES	YES	NO	NO	YES
Rigorous 2PL	YES	YES	YES	YES	NO
Strict 2PL	YES	YES	YES	YES	NO



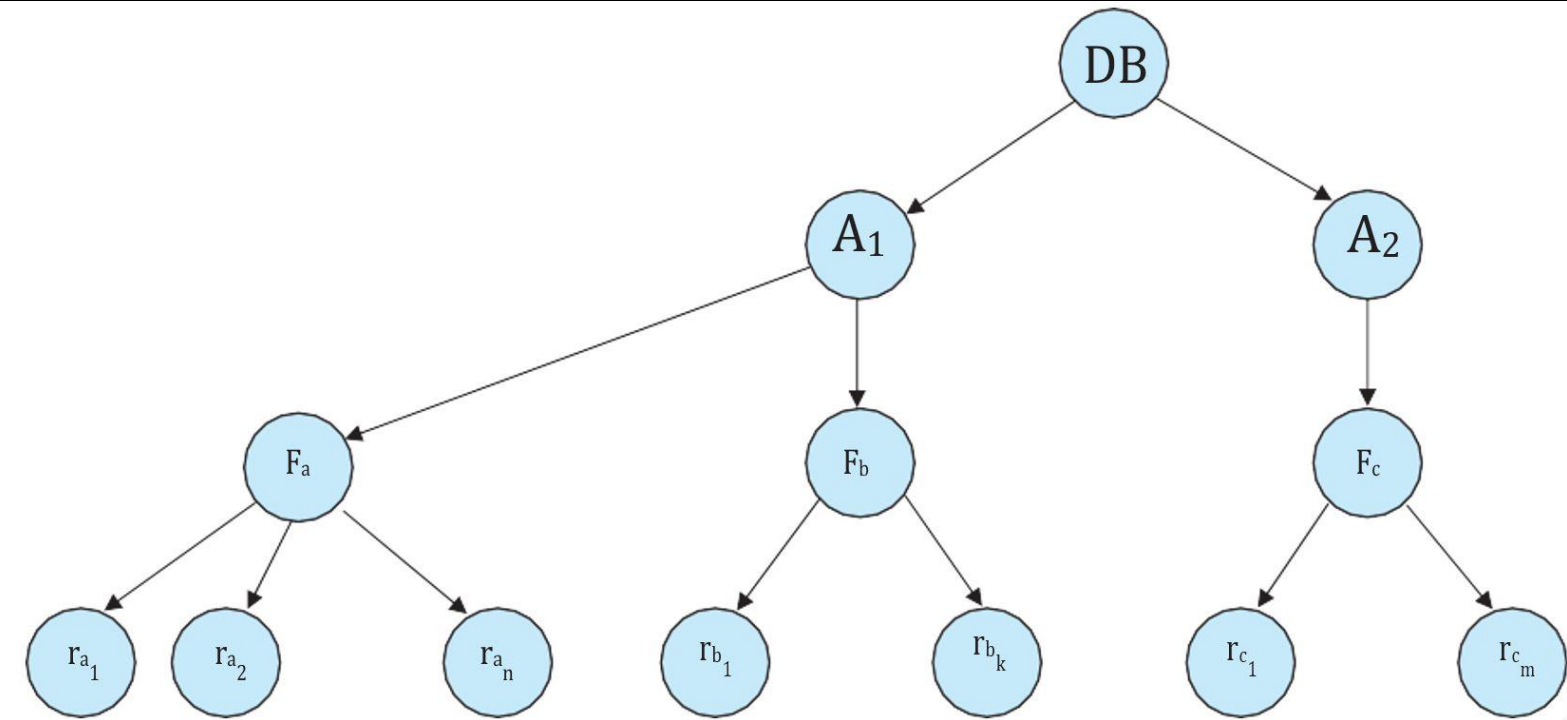
# Multiple Granularity

	IS	IX	S	X
IS				
IX				
S				
X				



## Multiple Granularity

	<b>IS</b>	<b>IX</b>	<b>S</b>	<b>X</b>
<b>IS</b>	true	true	true	false
<b>IX</b>	true	true	false	false
<b>S</b>	true	false	true	false
<b>X</b>	false	false	false	false



# Validation-Based Protocols

- In cases where a majority of transactions are read-only transactions, the rate of conflicts among transactions may be low.
- Thus, many of these transactions, if executed without the supervision of a concurrency-control scheme, would nevertheless leave the system in a consistent state.
- A concurrency-control scheme imposes overhead of code execution and possible delay of transactions. It may be better to use an alternative scheme that imposes less overhead.
- A difficulty in reducing the overhead is that we do not know in advance which transactions will be involved in a conflict. To gain that knowledge, we need a scheme for *monitoring* the system.
- The **validation protocol** requires that each transaction  $T_i$  executes in two or three different phases in its lifetime, depending on whether it is a read-only or an update transaction. The phases are, in order:

- **Read phase.** During this phase, the system executes transaction  $T_i$ . It reads the values of the various data items and stores them in variables local to  $T_i$ . It performs all write operations on temporary local variables, without updates of the actual database.
- **Validation phase.** The validation test (described below) is applied to transaction  $T_i$ . This determines whether  $T_i$  is allowed to proceed to the write phase without causing a violation of serializability. If a transaction fails the validation test, the system aborts the transaction.
- **Write phase.** If the validation test succeeds for transaction  $T_i$ , the temporary local variables that hold the results of any write operations performed by  $T_i$  are copied to the database. Read-only transactions omit this phase.