What is Relational Model?

- Relational Model represents how data is stored in Relational Databases. A relational database stores data in the form of relations (tables).
  - After designing the conceptual model of Database using ER diagram, we need to convert the conceptual model in the relational model which can be implemented using any RDMBS languages
  - RDMBS languages: Oracle, SQL, MySQL etc.

Introduction to Relational Model

- The **relational model** is the **theoretical** basis of relational databases
- The relational model of data is based on the concept of **relations**
- A "**Relation**" is a mathematical concept based on the ideas of **sets**
- The **Relational Model** was proposed by **E.F. Codd** for IBM in 1970 **to model data in the form of relations or tables.**

What is RDBMS?

- RDBMS stands for: **R**elational **D**atabase **M**anagement **S**ystem. RDBMS is the basis for SQL, and for all modern database systems like MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access.
- A **Relational database management system (RDBMS)** is a database management system (DBMS) that is based on the **relational model** as introduced by E. F. Codd.
- Current popular **RDBMS** include:
  - DB2 & Informix Dynamic Server - *from IBM*
  - Oracle & Rdb - *from Oracle*
  - SQL Server & MS Access - *from Microsoft*

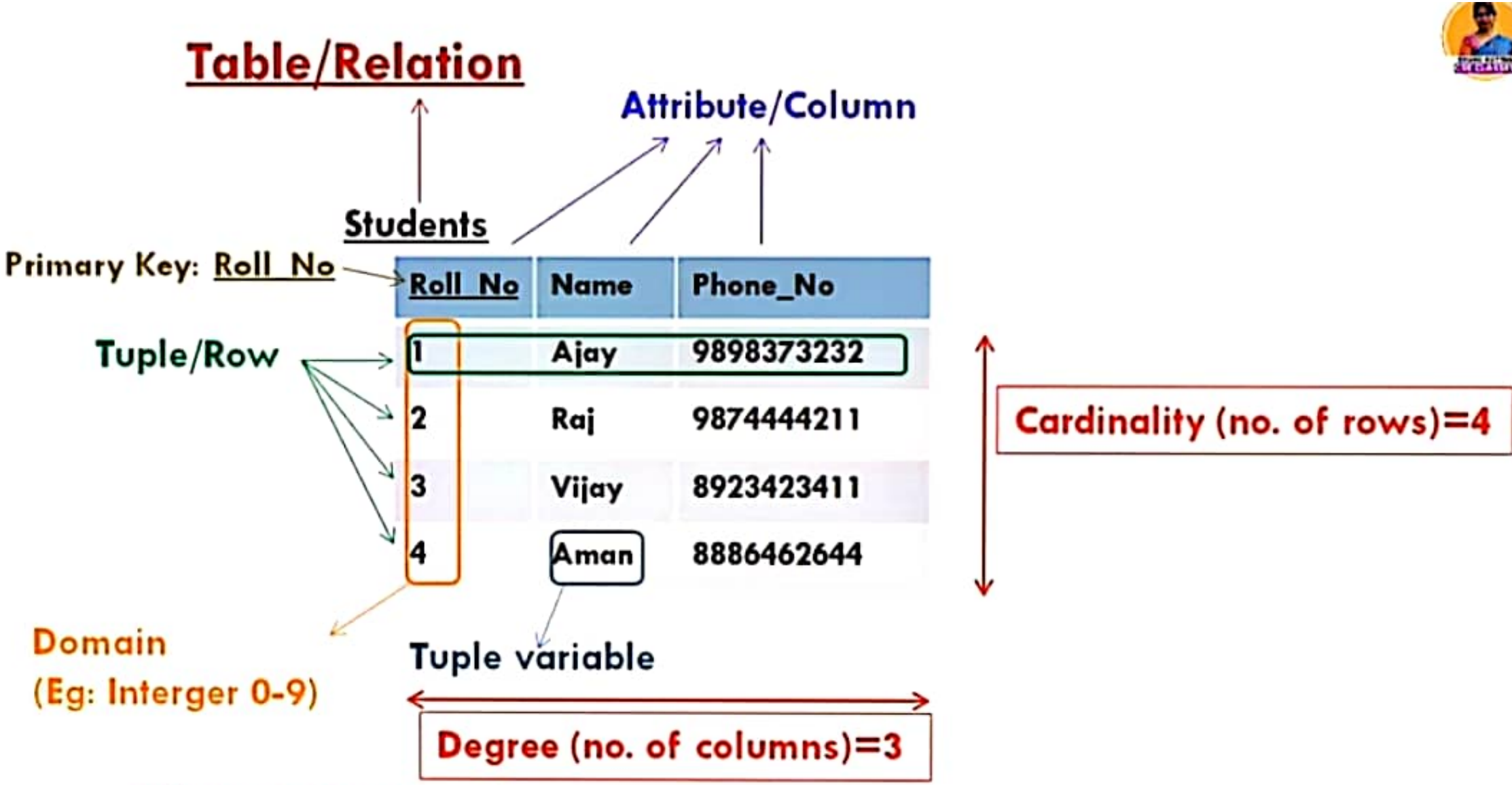
Relational Model concept

Roll_No	Name	Phone_N
1	Ajay	9898373252
2	Raj	9874444211
3	Vijay	8923423411
4	Ar an	8886462644

- **Relation:** A relation is a table with columns and rows.
- **Attribute:** An attribute is a named column of a relation.
- **Domain:** A domain is the set of allowable values for one or more attributes.
- **Tuple:** A tuple is a row of a relation.
- **Relation Schema:** A relation schema represents the name of the relation with its attributes.
- **Relation instance** (State): Relation instance is a finite set of tuples. Relation instances never have duplicate tuples.
- **Degree:** The total number of columns or attributes in the relation
- **Cardinality:** Total number of rows present in the Table.
- **Relation key:** Every row has one or multiple attributes, that can uniquely identify the row in the relation, which is called relation key (Primary key).
- **Tuple Variable:** it is the data stored in a record of the table

Roll_No	Name	Phone_No
1	Ajay	9898373232
2	Raj	9874444211
3	Vijay	8923423411
4	Aman	8886462644

Table/Relation



Relation Schema: Students (Roll_No, Name, Phone_No)

Properties of Relational Model

- Each **Relation** has **unique name**
- Each **tuple/Row** is **unique**: No duplicate row
- Entries in any **column** have the **same domain.**
- Each **attribute/column** has a **unique name**
- **Order** of the columns or rows is **irrelevant** i.e *relations are unordered*
- Each **cell** of relation contains exactly **one value** i.e. *attribute values are required to be atomic*

Students

Roll_No	Name	Phone_No
1	Ajay	9898373232
2	Raj	9874444211
3	Vijay	8923423411
4	Aman	8886462644

Alternative Terminology for Relational Mode

Formal terms	Alternative 1	Alternative 2
Relation	Table	File
Tuple	Row	Record
Attribute	Column	Field

Integrity Constraints over Relation

- **Integrity constraints** are used *to ensure accuracy and consistency of the data in a relational database.*
- **Integrity constraints** are **set of rules** that the database is not permitted to violate.
- **Constraints** may apply to each **attribute** or they may apply to **relationships between tables.**
- **Integrity constraints** ensure that changes (update, deletion, insertion) made to the database by authorized users do not result in a loss of data consistency. Thus, ***integrity constraints guard against accidental damage to the database.***
  - **Example -** A blood group must be 'A' or 'B' or 'AB' or 'O' only (can not any other values else).

TYPES OF INTEGRITY CONSTRAINT

1. Domain Constraint
2. Entity Integrity Constraint
3. Referential Integrity Constraint
4. Key Constraints

1. **Domain Constraint**
2. **Entity Integrity Constraint**
3. **Referential Integrity Constraint**
4. **Key Constraints**

1. Domain Constraints

- **Domain constraints** defines the domain or the valid set of values for an attribute.
- The **data type of domain** includes string, character, integer, time, date, currency, etc. *The value of the attribute must be available in the corresponding domain.*

STUDENT_ID	NAME	SEMESTER	AGE
101	Manish	1st	18
102	Rohit	3rd	19
103	Badal	5th	20
104	Amit	7th	A

Not allowed. Because AGE is an integer value

2. Entity Integrity Constraints

- The **entity integrity constraint** states that **primary key value can't be null.**
- This is because the primary key value is used to identify individual rows in relation and if the primary key has a null value, then we can't identify those rows.
- A table can contain a null value other than the primary key field.

EMP_ID	EMP_NAME	SALARY
111	Mohan	20000
112	Rohan	30000
113	Sohan	35000
	Logan	20000

↑
Not allowed as Primary Key can't contain NULL value

3. Referential Integrity Constraints

- A **referential integrity constraint** is specified between two tables.
- **Referential Integrity constraint** is enforced when a foreign key references the primary key of a table.
- In the **Referential integrity constraints**, if a foreign key in Table 1 refers to the Primary Key of Table 2, then ***either*** every value of the foreign Key in Table 1 must be available in primary key value of Table 2 **or** it must be null.
- The **rules** are:
  - You can't **delete** a record from a *primary table* if matching records exist in a related table.
  - You can't **change** a primary key value in the *primary table* if that record has related records.
  - You can't **insert** a value in the foreign key field of the *related table* that doesn't exist in the primary key of the *primary table*.
  - However, you can enter a **Null** value in the foreign key, specifying that the records are unrelated.

Foreign Key

Relationships

EMP (Table 1): Related /Referencing Table

EMP_ID	EMP_NAME	AGE	DEP_NO
111	Mohan	21	1
112	Rohan	33	2
113	Sohan	27	3
114	Logan	25	5 ✓

Primary Key

DEP (Table 2): Primary/Referenced Table

DEP_NO	LOCATION
1	Mumbai
2	Delhi
3	Noida

Not allowed as DEP_NO 5 is not defined as a Primary Key of Table 2 and In Table 1, DEP_NO is a foreign key defined

4. Key Constraints

- An entity set can have multiple keys or candidate keys (minimal superkey), but out of which one key will be the primary key.
- **Key constraint** specifies that in any relation-
  - All the values of primary key must be **unique.**
  - The value of primary key must **not be null.**

PK

STUDENT_ID	NAME	SEMESTER	AGE
101	Manish	1st	18
102	Rohit	3rd	19
103	Badal	5th	20
102	Amit	7th	21

Not allowed. Because all rows must be **unique**

Keys in DBMS

- ***Definition***: A **key** is an **attribute** or **set of attributes** that **uniquely identifies** any record (or tuple) from the table.
- ***Purpose:***
  - **Key** is used **to uniquely identify any record or row of data from the table.**
  - It is also used to establish and identify **relationships** between **tables.**

Example: Employee Table

Emp_Id	Name	Aadhar_No	Email_Id	Dept_Id
01	Aman	775762540011	aa@gmail.com	1
02	Neha	876834788522	nn@gmail.com	2
03	Neha	996677898677	ss@gmail.com	2
04	Vimal	796454638800	vv@gmail.com	3

Types of keys in DBMS

1. Super Key
2. Candidate Key
3. Primary Key
4. Alternate Key
5. Foreign Key
6. Composite Key

1. Super Key

- A **super key** is a combination of all possible attributes that can uniquely identify the rows (or tuple) in the given relation.
  - Super key is a superset of a candidate key.
  - A table can have many super keys
  - A super key may have additional attribute that are not needed for unique identity

Super Keys

Emp_Id	Name	Aadhar_No	Email_Id	Dept_Id
01	Aman	775762540011	aa@gmail.com	1
02	Neha	876834788522	nn@gmail.com	2
03	Neha	996677898677	ss@gmail.com	2
04	Vimal	796454638800	vv@gmail.com	3

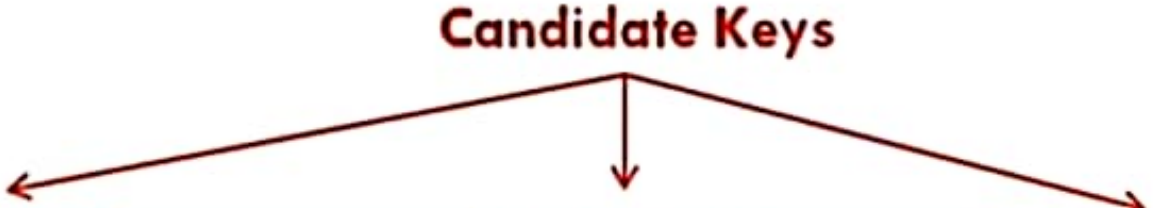
Super Keys:

1. {Emp_Id}
2. {Aadhar_No}
3. {Email_Id}
4. {Emp_Id, Aadhar_No}
5. {Aadhar_No, Email_Id}
6. {Emp_Id , Email_Id}
7. {Emp_Id, Aadhar_No, Email_Id}
8. {Emp_Id, Name}
9. {Emp_id, Name, Dep_Id}
10. {Emp_Id,Name, Aaadhar_No, Email_Id, Dept_Id}, etc....

2. Candidate Key

- A candidate key is an attribute or set of an attribute which can uniquely identify a tuple.
- A candidate key is a **minimal super key;** or a Super key with no redundant attributes.
  - It is called a minimal super key because we select a candidate key from a set of super key such that selected candidate key is the minimum attribute required to uniquely identify the table
- Candidate keys are defined as distinct set of attributes from which **primary key can be selected**

Candidate keys are not allowed to have NULL values



Emp_Id	Name	Aadhar_No	Email_Id	Dept_Id
01	Aman	775762540011	aa@gmail.com	1
02	Neha	876834788522	nn@gmail.com	2
03	Neha	996677898677	ss@gmail.com	2
04	Vimal	796454638800	vv@gmail.com	3

- **Candidate Keys**
  1. {Emp_Id}
  2. {Aadhar_No}
  3. {Email_Id}

3. Primary Key

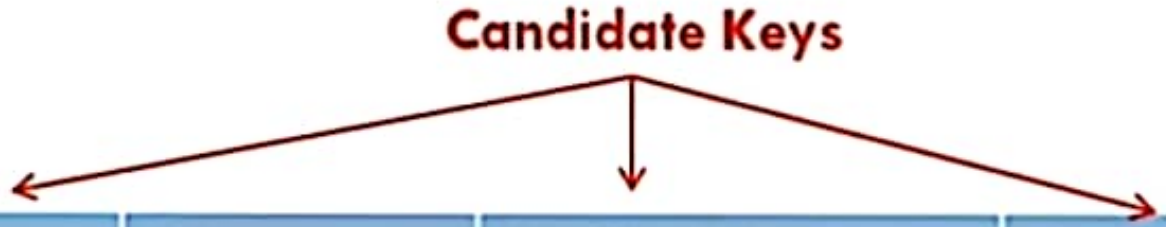
- A **primary key** is one of the candidate key chosen by the database designer to uniquely identify the tuple in the relation
  - The value of primary key can never be NULL.
  - The value of primary key must always be unique (not duplicate).
  - The values of primary key can never be changed i.e. no updation is possible.
  - The value of primary key must be assigned when inserting a record.
  - A relation is allowed to have only one primary key.

Primary Key

Alternate Keys

Emp_Id	Name	Aadhar_No	Email_Id	Dept_Id
01	Aman	775762540011	aa@gmail.com	1
02	Neha	876834788522	nn@gmail.com	2
03	Neha	996677898677	ss@gmail.com	2
04	Vimal	796454638800	vv@gmail.com	3

- **Alternate Keys**
  1. {Aadhar_No}
  2. {Email_Id}



Emp_Id	Name	Aadhar_No	Email_Id	Dept_Id
01	Aman	775762540011	aa@gmail.com	1
02	Neha	876834788522	nn@gmail.com	2
03	Neha	996677898677	ss@gmail.com	2
04	Vimal	796454638800	vv@gmail.com	3

- **Candidate Keys**
  1. {Emp_Id}
  2. {Aadhar_No}
  3. {Email_Id}

Primary Key

Emp_Id	Name	Aadhar_No	Email_Id	Dept_Id
01	Aman	775762540011	aa@gmail.com	1
02	Neha	876834788522	nn@gmail.com	2
03	Neha	996677898677	ss@gmail.com	2
04	Vimal	796454638800	vv@gmail.com	3

- **Primary Key**
  1. {Emp_Id}

3. Primary Key

- A **primary key** is one of the candidate key chosen by the database designer to uniquely identify the tuple in the relation
  - The value of primary key can never be NULL.
  - The value of primary key must always be unique (not duplicate).
  - The values of primary key can never be changed i.e. no updation is possible.
  - The value of primary key must be assigned when inserting a record.
  - A relation is allowed to have only one primary key.

- Out of all candidate keys, only one gets selected as primary key, remaining keys are known as alternate keys.
- In the Employee table
  - **Emp_Id** is best suited for the **primary key.**
  - Rest of the attributes like **Aadhar_No, Email_Id** are considered as a **alternate keys.**

5. Foreign keys

- A **Foreign Key** is:
  - A **key** used to link two tables together.
  - An **attribute (or set of attributes)** in one table that refers to the **Primary Key** in another table.
- The **purpose** of the **foreign key** is
  - to ensure (or maintain) **referential integrity** of the data.

Foreign Keys

Employee Table (Referencing relation)

Emp_Id	Name	Aadhar_No	Email_Id	Dept_Id
01	Aman	775762540011	aa@gmail.com	1
02	Neha	876834788522	nn@gmail.com	2
03	Neha	996677898677	ss@gmail.com	2
04	Vimal	796454638800	vv@gmail.com	3

oreign Key:

In Employee Table

1. Dept_Id

Primary Key

Department Table (Referenced relation)

Dept_Id	Dept_Name
1	Sales
2	Marketing
3	HR

Foreign Key

- Foreign key references the primary key of the table.
- Foreign key can take only those values which are present in the primary key of the referenced relation.
- Foreign key may have a name other than that of a primary key.
- Foreign key can take the NULL value.
- There is no restriction on a foreign key to be unique.
- In fact, foreign key is not unique most of the time.
- Referenced relation may also be called as the master table or primary table.
- Referencing relation may also be called as the foreign table.

6. Composite Key

- A key that has more than one attributes is known as composite key. It is also known as compound key.

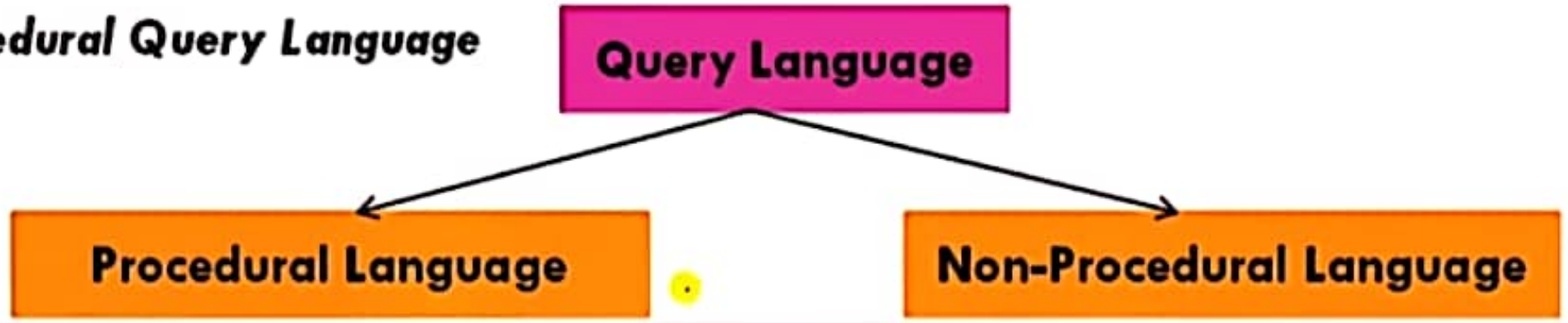
Cust_Id	Order_Id	Product_Code	Product_Count
C01	001	P111	5
C02	012	P111	8
C02	012	P222	6
C01	001	P333	9

- **Composite Key:**

  {Cust_Id, Product_Code}

Relational Query Languages

- Relational database systems are expected be equipped with a *query language* that assist its users to query the database instances.
- **Query Language** is a Language in which user requests information from the database. Eg: **SQL**
- **Query** = "Retrieval Program"
- **Two types of Query Language:**
  1. ***Procedural Query Language***
  2. ***Non-Procedural Query Language***



Procedural vs. Non-Procedural

1. Procedural Query language:

- In ***Procedural query language***, user instructs the system to perform a series of operations to produce the desired results.
- **User** tells ***what data to be retrieved from database and how to retrieve it.***

2. Non-procedural (or Declarative) Query Language:

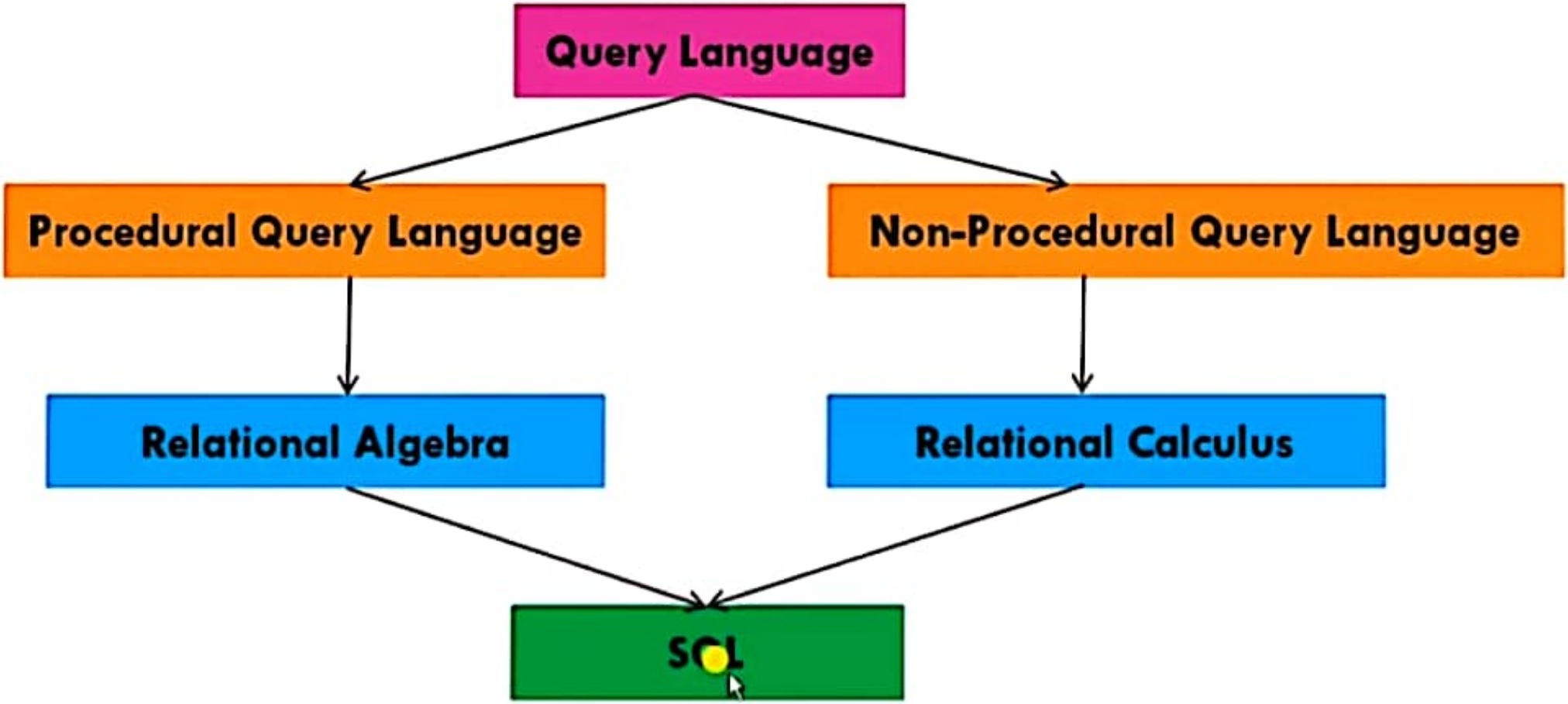
- In ***Non-procedural query language***, user instructs the system to produce the desired result without telling the step by step process.
- **User** tells ***what data to be retrieved from database but doesn't tell how to retrieve it.***

Two "Pure" Query languages

- **Two "Pure" Query languages or Two Mathematical Query Language:**
  1. **Relational Algebra** ✓ Imp
  2. **Relational Calculus** ✓
     1. *Tuple Relational Calculus*
     2. *Domain Relational Calculus*

Relational Algebra vs. Relational Calculus

1. **Relational Algebra:**
   - Relational algebra is a **procedural** query language
   - It is more operational, very useful for representing execution plan.
   - ***Procedural*: *What*** *data is required and* ***How*** *to get those data.*
2. **Relational Calculus:**
   1. *Tuple Relational Calculus*
   2. *Domain Relational Calculus*
   - Relational calculus is a non-procedural query language
   - It is non-operational or declarative
   - ***Non-Procedural*: *What*** *data they want without specifying how to get those data.*


Query Language
Procedural Query Language
Non-Procedural Query Language
Relational Algebra
Relational Calculus
SQL

Relational Algebra, Calculus, RDBMS & SQL:

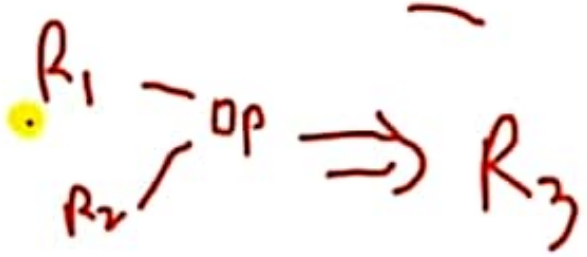
- **Relational Model** is a **theoretical concept.**
- **RDBMS** is a **practical implementation** of **relational model.**
- **SQL** (Structured Query Language) is **used to write query** on **RDBMS**

- ***Relational algebra and calculus*** are the **Mathematical system** or **Query language** used on **relationa model.**
  - Understanding *Relational Algebra and Calculus* is key to understand ***SQL Query Language***
- ***SQL*** is a **practical implementation** of **relational algebra and calculus.**

Relation Model	RDBMS
Relational Algebra, Relational Calculus	SQL
Algorithm	Code
Conceptual	Reality
Theoretical	Practical

Relational Algebra

- **Relational Algebra** is a **procedural query language** which takes a relation as an **input** and generates a relation as an **output**
- Relational Algebra is a language for expressing ***relational database queries***
- It **uses operators** to perform **queries.** An operator can be either **unary** or **binary.**
- **Types of operators in relational algebra:**
  1. Basic/Fundamental Operators
  2. Additional/Derived Operators



- Relational algebra operations work on one or more relations to define another relation without changing the original relations.
- Relational algebra operations are performed recursively on a relation

Example

- In relational algebra, **input is a relation** (table from which data has to be accessed) and **output is also a relation** (a temporary table holding the data asked for by the user).
- Relational algebra is performed recursively on a relation and intermediate results are also considered relations. i.e Relational Algebra works on the whole table at once, so we do not have to use loops etc to iterate over all the row (tuples) of data one by one.
- All we have to do is *to specify the table name* from which we need the data, and in a *single line of command*, relational algebra will traverse the entire given table to fetch data for you.

We can use Relational Algebra to fetch data from this Table(relation)

ID	Name	Age
1	Akon	17
2	Bkon	19
3	Ckon	15
4	Dkon	13

Select Name students with age less than 17

Output

Name
Ckon
Dkon

The output for query is also in form of a table(relation), with results in different columns

Relational Algebra Operations

1. Basic/Fundamental Operations:

1. Selection (σ)
2. Projection (∏)
3. Union (U)
4. Set Difference (–)
5. Cartesian product (X)
6. Rename (ρ)

- ***Select, Project and Rename*** are ***Unary*** operators because they operate on one relation
- ***Union, Difference and Cartesian product*** are ***Binary*** operators because they operate on two relation

2. Additional/Derived Operations:

1. Natural Join (⋈)
2. Left, Right, Full Outer Join (⟕, ⟖, ⟗)
3. Set Intersection (∩)
4. Division (÷)
5. Assignment (←)

- All are Binary operators because they operate on two relations

Selection (σ) Operator

Selection Operator (σ) is a **unary operator** in relational algebra that performs a selection operation.

It selects tuples (or rows) that satisfy the given condition (or predicate) from a relation.

It is denoted by **sigma (σ).**

Notation – $\sigma_p(r)$ **or** $\sigma_{(Condition)}(Relation\ Name)$

- *p is used as a propositional logic formula which may use* ***logical connectives****:* ∧ *(AND),* ∨ *(OR),* ! *(NOT) and* ***relational operators*** *like* =, ≠, <, >, ≤, ≥ *to form the* ***condition****.*

The **WHERE clause** of a SQL command **corresponds** to relational **select** σ( ).

- **SQL:** SELECT * FROM R WHERE C;

Example: Select tuples from student table whose age is greater than 17

$\sigma_{age > 17}$ **(Student)**

Student

roll_no	name	age	address
1	A	20	Bhopal
2	B	17	Mumbai
3	C	16	Mumbai
4	D	19	Delhi
5	E	18	Delhi

Query 1: Select student whose roll no. is 2

$\sigma_{roll\_no=2}$ (Student)

roll_no	name	age	address
2	B	17	Mumbai

Note: *In Selection operation, schema of resulting relation is identical to schema of input relation*

Student

roll_no	name	age	address
1	A	20	Bhopal
2	B	17	Mumbai
3	C	16	Mumbai
4	D	19	Delhi
5	E	18	Delhi

Query 2: Select student whose name is D

$$\sigma_{name="D"}(Student)$$

roll_no	name	age	address
4	D	19	Delhi

roll_no	name	age	address
1	A	20	Bhopal
4	D	19	Delhi
5	E	18	Delhi

Student

roll_no	name	age	address
1	A	20	Bhopal
2	B	17	Mumbai
3	C	16	Mumbai
4	D	19	Delhi
5	E	18	Delhi

Query 4: Select students whose age is greater than 17 and who lives in Delhi

$\sigma_{age > 17 \wedge address = "Delhi"}$ (Student)

roll_no	name	age	address
4	D	19	Delhi
5	E	18	Delhi

Examples

- Select tuples from a relation "**Books**" where subject is "**database**"

$$\sigma_{\text{subject = "database"}}(\text{Books})$$

- Select tuples from a relation "**Books**" where subject is "**database**" and price is "450"

$$\sigma_{\text{subject = "database"} \wedge \text{price = 450}}(\text{Books})$$

- Select tuples from a relation "**Books**" where subject is "**database**" and price is "**450**" or have a publication **year after 2010**

$$\sigma_{\text{subject = "database"} \wedge \text{price = 450} \vee \text{year > 2010}}(\text{Books})$$

and or

PROJECTION OPERATOR

Projection (Π) Operator

- **Projection Operator (Π)** is a **unary operator** in relational algebra that performs a projection operation.
- It **projects** (or displays) the particular **columns (or attributes)** from a relation and
- It delete column(s) that are not in the projection list.
- It is denoted by Π
- **Notation –** $\Pi_{A1, A2, ...An}(r)$ **or** $\Pi_{Attribute\_list}$ **(relation name/table name)**
  - Where A_1, A_2 , A_n are attribute names of relation **r.**
- Duplicate rows are automatically eliminated from result
- The SQL **SELECT command** corresponds to relational **project** Π( ).
  - **SQL:** SELECT A1,A2, . . . An FROM R;
- Example: Display the columns roll_no and name from the relation Student.

$$\Pi_{roll\ no.\ name}(Student)$$

Projection (Π) Operator

- **Projection Operator (Π)** is a **unary operator** in relational algebra that performs a projection operation.
- It **projects** (or displays) the particular **columns (or attributes)** from a relation and
- It delete column(s) that are not in the projection list.
- It is denoted by Π
- **Notation –** $\Pi_{A1, A2, ...An}(r)$ **or** $\Pi_{Attribute\_list}$ **(relation name/table name)**
  - Where A_1, A_2 , A_n are attribute names of relation **r.**
- Duplicate rows are automatically eliminated from result
- The SQL **SELECT command** corresponds to relational **project** Π( ).
  - **SQL:** SELECT A1,A2, . . . An FROM R;
- **Example:** Display the columns roll_no and name from the relation Student.

$\Pi_{roll\ no, name}$ **(*Student*)**

Student

roll_no	name	age
1	A	20
2	B	17
3	C	16
4	D	19
5	E	18
6	F	18

Query 1: Display (or project) the name of students in student table

$$\Pi_{name}(Student)$$

name
A
B
C
D
E
F

Student

roll_no	name	age
1	A	20
2	B	17
3	C	16
4	D	19
5	E	18
6	F	18

Query 2: Display the roll_no and name of students in student table

$$\Pi_{\text{roll\_no, name}}\ (\textit{Student})$$

roll_no	name
1	A
2	B
3	C
4	D
5	E
6	F

Student

roll_no	name	age
1	A	20
2	B	17
3	C	16
4	D	19
5	E	18
6	F	18

Query 3: Display the age of students in student table

$$\Pi_{age}(Student)$$

age
20
17
16
19
18

Note: By default, projection removes duplicate values

Student

roll_no	name	age
1	A	20
2	B	17
3	C	16
4	D	19
5	E	18
6	F	18

Query 4: Display the roll_no and name of students whose age is greater than 17

$$\Pi_{roll\_no,\ name}\ (\sigma_{age\ >\ 17}\ (Student))$$

roll_no	name
1	A
4	D
5	E
6	F

Wrong

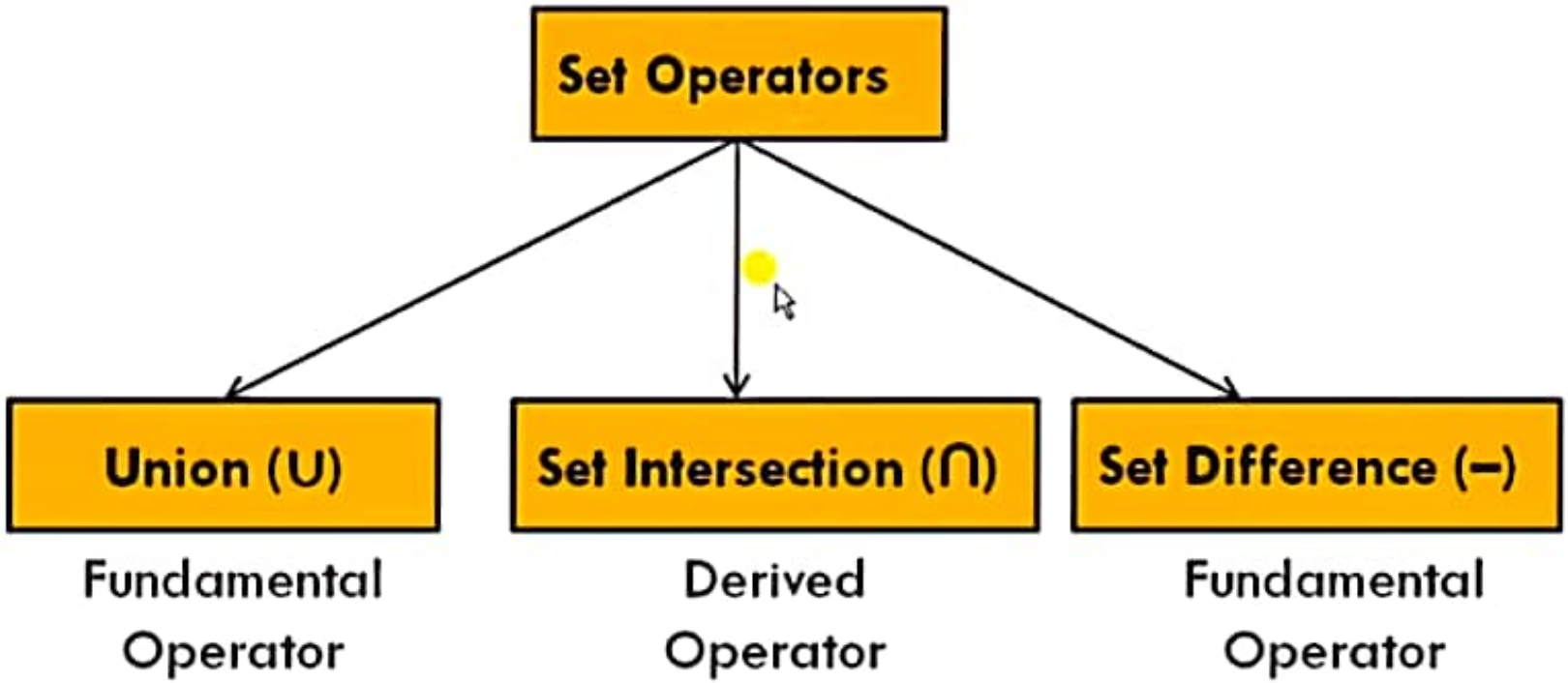
$$\sigma_{age\ >\ 17}\ (\Pi_{roll\ no,\ name}\ (Student))$$

SET OPERATORS: UNION, SET INTERSECTION & SET DIFFERENCE

SHANU KUTTAN
CSE CLASSES

DBMS

Set Operators in Relational Algebra



Set Operators

- **Set operators:** Union, intersection and difference, binary operators as they takes two input relations

- **To use set operators on two relations,**
  - The two relations must be **Compatible**
- **Two relations are Compatible if -**
  1. Both the relations must have **same number of attributes (or columns).**
  2. Corresponding **attribute (or column)** have the **same domain (or type).**
- Duplicate tuples are automatically eliminated

Union (U) Operator

Suppose R and S are two relations. ***The Union operation selects all the tuples that are either in relations R or S or in both relations R & S.***

It eliminates the duplicate tuples.

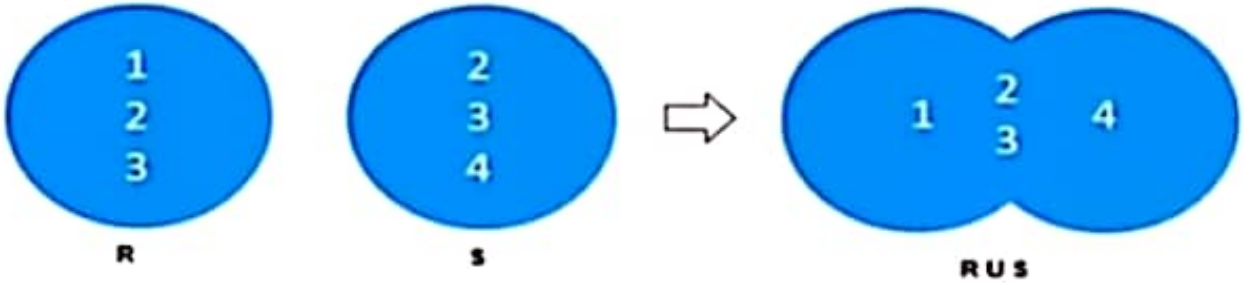
For a **union operation** to be **valid**, the following conditions must hold -

1. Two relations R and S both have **same number of attributes.**
2. Corresponding **attribute (or column)** have the **same domain (or type)..**
   - The attributes of R and S must occur in the same order.
3. Duplicate tuples should be automatically removed

Symbol: U

Notation: **R U S**

- RA: **R U S**
- SQL: **(SELECT * FROM R)**
  UNION
  (SELECT * FROM S);



Example

Student

Roll_no	Name
1	A
2	B
3	C
4	D

Employee

Emp_no	Name
2	B
8	G
9	H

(Student) U (Employee)

Roll_no	Name
1	A
2	B
3	C
4	D
8	G
9	H

Note: Union is commutative: A U B = B U A

Student

Roll_no	Name
1	A
2	B
3	C
4	D

Employee

Emp_no	Name
2	B
8	G
9	H

Π_{Name} (Stud

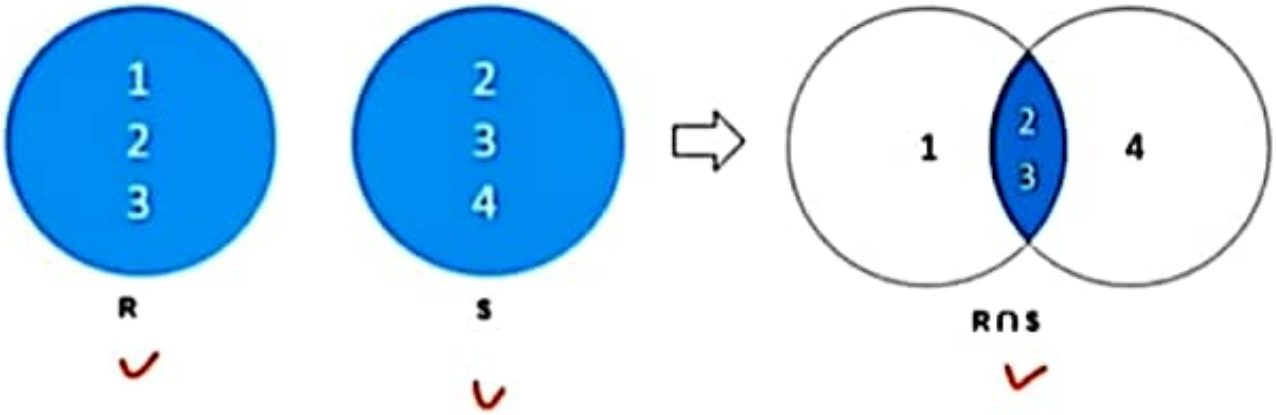
Example

- Find the names of the authors who have either written a book or an article or both.

$$\Pi_{\text{author}}(\text{Books}) \cup \Pi_{\text{author}}(\text{Articles})$$

Set Intersection (∩) Operator

- Suppose R and S are two relations. ***The Set Intersection operation selects all the tuples that are in both relations R & S.***
- For a **Set Intersection** to be **valid**, the following conditions must hold –
  1. Two relations R and S both have **same number of attributes.**
  2. Corresponding **attribute (or column)** have the **same domain (or type).**
     - The attributes of R and S must occur in the same order.
- **Symbol:** ∩
- **Syntax:** R ∩ S
  - RA: **R ∩ S**
  - SQL: **(SELECT * FROM R)**
    INTERSECT
    (SELECT * FROM S);



Example

Student

Roll_no	Name
1	A
2	B
3	C
4	D

Employee

Emp_no	Name
2	B
8	G
9	H

(Student) ∩ (Employee)

Roll_no	Name
2	B

Note: Set Intersection is commutative: $A \cap B = B \cap A$

Example

Student

Roll_no	Name
1	A
2	B
3	C
4	D

Employee

Emp_no	Name
2	B
8	G
9	H

Π_{Name} (Student) $\cap$ Π_{Name} (Employee

Name
B

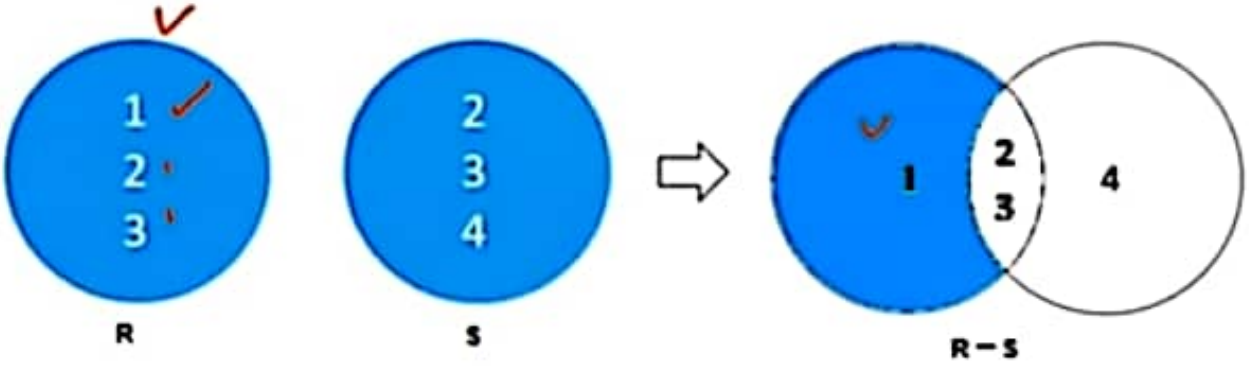
Example

- Find the names of the authors who have written a book and an article both.

$$\Pi_{\text{author}}(\text{Books}) \cap \Pi_{\text{author}}(\text{Articles})$$

Set Difference (–) Operator

- Suppose R and S are two relations. ***The Set Difference operation selects all the tuples that are present in first relation R but not in second relation S.***
- For a **Set Difference** to be **valid**, the following conditions must hold -
  1. Two relations R and S both have **same number of attributes.**
  2. Corresponding **attribute (or column)** have the **same domain (or type).**
     - The attributes of R and S must occur in the same order.
- **Symbol:** –
- **Syntax**: R – S
  - RA: R – S
  - SQL: (SELECT * FROM R)
    EXCEPT
    (SELECT * FROM S);



Example

Student

Roll_no	Name
1	A
2	B
3	C
4	D

Employee

Emp_no	Name
2	B
8	G
9	H

(Student) – (Employee)

Roll_no	Name
1	A
3	C
4	D

Note: **1. Set Difference is non-commutative: $A - B \neq B - A$**

2. $R - (R - S) = R \cap S$

Intersection can be is derived from set difference that's why intersection is derived operator

Example

Student

Roll_no	Name
1	A
2	B
3	C
4	D

Employee

Emp_no	Name
2	B
8	G
9	H

Π_{Name} (Student) – Π_{Name} (Employee)

Name
A
C
D

Example

- Find the names of the authors who have written books but not articles.

$$\Pi_{\text{author}} (\text{Books}) - \Pi_{\text{author}} (\text{Articles})$$

Cartesian Product/Cross Product

- **Cartesian Product** is fundamental operator in relational algebra
- **Cartesian Product combines information of two different relations into one.**
- It is also called **Cross Product.**
  - Generally, a **Cartesian Product** is never a meaningful operation when it is performed alone. However, it becomes **meaningful when it is followed by other operations.**
  - Generally it is followed by select operations.
- **Symbol: x**
- **Notation: R1 x R2**
- **SQL: SELECT * FROM R1, R2**

Example

R1

A	B
α	1
β	2

R2

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

R1 x R2

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

Characteristics

1. If ***relation R1 and R2 have a & b attributes*** respectively, then **resulting relation** will have **a + b attributes** from both the input relations.
2. If ***relation R1 and R2 have n1 & n2 tuples*** respectively, then **resulting relation** will have **n1 x n2 tuples**, combining each possible pair of tuples from both the relations.

	R1	R2	R1 x R2
Attributes	a	b	(a + b)
Tuples	n1	n2	(n1 x n2)

3. If both input relation have **some attribute having same name**, change the name of the attribute with the name of the relation **"relation_name.attribute_name"**

Example

R1

A	B
α	1
β	2

R2

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

R1 x R2

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

Example

R1

A	B
α	1
β	2

2

R2

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

4

R1 x R2

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

R1 x R2
Resulting relation must have

2+3 = 5 attributes ✓
2 x 4= 6 tuples

Characteristics

1. If ***relation R1 and R2 have a & b attributes*** respectively, then **resulting relation** will have **a + b attributes** from both the input relations.
2. If ***relation R1 and R2 have n1 & n2 tuples*** respectively, then **resulting relation** will have **n1 x n2 tuples**, combining each possible pair of tuples from both the relations.

	R1	R2	R1 x R2
Attributes	a	b	(a + b)
Tuples	n1	n2	(n1 x n2)

3. If both input relation have **some attribute having same name**, change the name of the attribute with the name of the relation **"relation_name.attribute_name"**

Example

- If both input relation have **some attribute having same name**, change the name of the attribute with the name of the relation **"relation_name.attribute_name"**

R1

A	B
α	1
β	2

R2

B	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

R1 x R2

A	R1.B	R2.B	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

Example: Composition of Operations

- Can build expressions using multiple operations
- **Example:** $\sigma_{A=C}(R1 \times R2)$

A	B	C	D	E
α	1	α	10	a
β	2	β	10	a
β	2	β	20	b

R1 x R2

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

Example

$$\sigma_{author=\text{“Korth”}}(Books \times Articles)$$

The resulting relation shows all the books and articles written by korth

Composition of Operations

- **Example:** $\Pi_{A}(\sigma_{D=20}(R1 \times R2))$

A
α
β

R1 x R2

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

Rename Operator

- The results of relational algebra are also relations but without any name.
- **The RENAME operator is used to rename the output of a relation.**
- Sometimes it is simple and suitable to break a complicated sequence of operations and rename it as a relation with different names. Reasons to rename a relation can be many, like:
  - We may want to save the result of a relational algebra expression as a relation so that we can use it later.
  - We may want to join (or cartesian product) a relation with itself, in that case, it becomes too confusing to specify which one of the tables we are talking about, in that case, we rename one of the tables and perform join operations on them.
- **Symbol**: rho ρ
- **Notation 1**: $\rho_x(E)$

Where the symbol 'ρ' is used to denote the **RENAME** operator

and **E** is **the result of expression or sequence of operation** which is **saved with the name X**

$R \times (R)_S$

- **SQL**: Use the **AS keyword in the FROM clause**

  (Eg: Students AS Students1 renames Students to Students1)

  ```
  SELECT column_name
  FROM tablename AS new_table_name
  WHERE condition ;
  ```

- **SQL**: Use the **AS keyword in the SELECT clause to rename attributes (columns)**

  (Eg: RollNo AS SNo renames RollNo to SNo)

  ```
  SELECT column_name AS new_column_name
  FROM tablename
  WHERE condition
  ```

Example:

- *Suppose we want to do cartesian producact between same table then* ***one of the table should be renamed with another name***
- $R \times R$

  (Ambiguity will be there)

R

A	B
α	1
β	2

- $R \times \rho_S (R)$

  (Rename R to S)

R x R

R.A	R.B	R.A	R.B
α	1	α	1
α	1	β	2
β	2	α	1
β	2	β	2

$R \times \rho_S (R)$

R.A	R.B	S.A	S.B
α	1	α	1
α	1	β	2
β	2	α	1
β	2	β	2

Rename Operation ...

- **Notation 2:** $\rho_{X(A1, A2, \ldots, An)}(E)$

  It returns the result of expression *E* under the name *X*, and with the attributes renamed to *A1*, *A2*, ..., *An*.

- **Notation 3:** $\rho_{(A1, A2, \ldots, An)}(E)$

  It returns the result of expression *E* with the attributes renamed to *A1*, *A2*, ..., *An*.

- **Example-1:** Query to find the female students from Student relation and rename the relation Student as FemaleStudent and the attributes of Student – RollNo, SName as Sno, Name.

Student

RollNo	SName	Gender
1	Neha	F
2	Suman	F
3	Sohan	M
4	Mohan	M
5	Rohan	M

Rename

FemaleStudent

SNo	Name
1	Neha
2	Suman

$$\rho_{FemaleStudent(Sno,\ Name)}\ (\pi_{RollNo,\ SName}(\sigma_{Gender='F'}(Student)))$$

Examples:

- **Example-2:** Query to rename the attributes Name, Age of table Person to N, A.

$$\rho_{(N, A)}(Person)$$

- **Example-3:** Query to rename the table name Project to Work and its attributes to P, Q, R

$$\rho_{Work(P, Q, R)}(Project)$$

- **Example-4:** Query to rename the first attribute of the table Student with attributes A, B, C to P.

$$\rho_{(P, B, C)}(Student)$$

- **Example-4:** Query to rename the table name Loan to L.

$$\rho_{L}(Loan)$$

Banking Example

branch (*branch-name*, *branch-city*, *assets*)

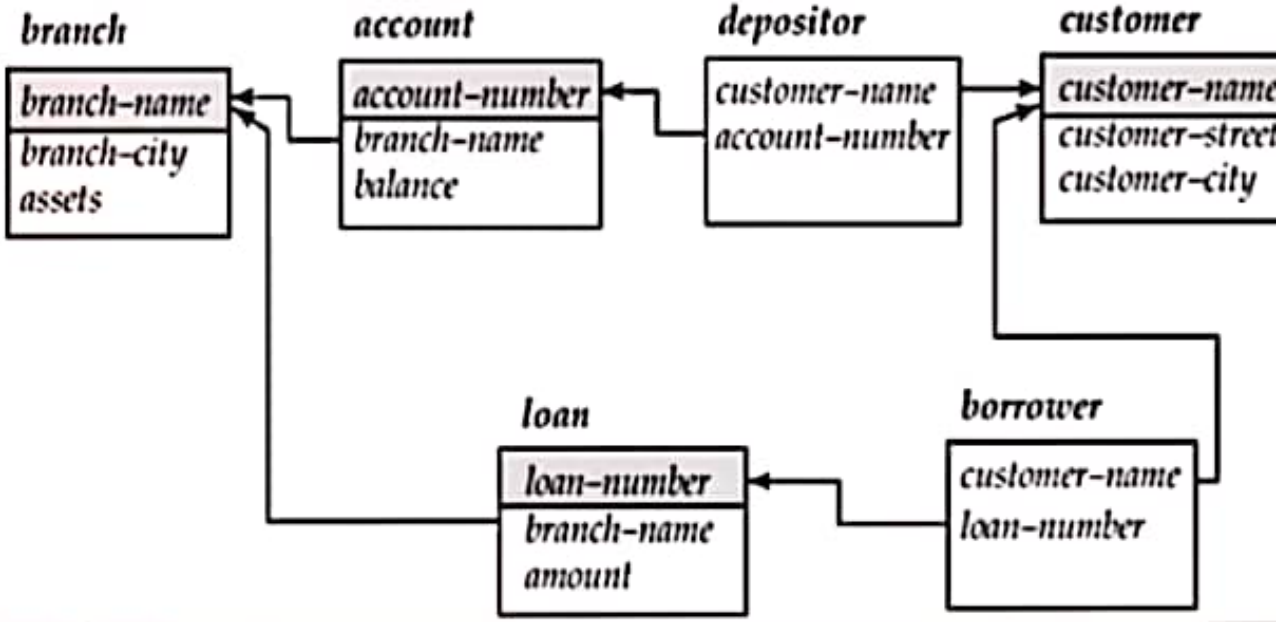
customer (*customer-name*, *customer-street*, *customer-city*)

account (*account-number*, *branch-name*, *balance*)

loan (*loan-number*, *branch-name*, *amount*)

depositor (*customer-name*, *account-number*)

borrower (*customer-name*, *loan-number*)



branch

branch-name	branch-city	assets
Brighton	Brooklyn	7100000
Downtown	Brooklyn	9000000
Mianus	Horseneck	400000
North Town	Rye	3700000
Perryridge	Horseneck	1700000
Pownal	Bennington	300000
Redwood	Palo Alto	2100000
Round Hill	Horseneck	8000000

customer

customer-name	customer-street	customer-city
Adams	Spring	Pittsfield
Brooks	Senator	Brooklyn
Curry	North	Rye
Glenn	Sand Hill	Woodside
Green	Walnut	Stamford
Hayes	Main	Harrison
Johnson	Alma	Palo Alto
Jones	Main	Harrison
Lindsay	Park	Pittsfield
Smith	North	Rye
Turner	Putnam	Stamford
Williams	Nassau	Princeton

depositor

customer-name	account-num
Hayes	A-102
Johnson	A-101
Johnson	A-201
Jones	A-217
Lindsay	A-222
Smith	A-215
Turner	A-305

account

account-number	branch-name	balance
A-101	Downtown	500
A-102	Perryridge	400
A-201	Brighton	900
A-215	Mianus	700
A-217	Brighton	750
A-222	Redwood	700
A-305	Round Hill	350

loan

loan-number	branch-name	amount
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

borrower

customer-name	loan-number
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

Selection (σ): Example 1

- Find all **loans** made at "Perryridge" **branch**

$$\sigma_{\text{branch-name} = \text{"Perryridge"}} (\text{loan})$$

- Find all **loans** of over $1200

$$\sigma_{\text{amount} > 1200} (\text{loan})$$

loan

loan-number	branch-name	amount
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

branch (branch-name, branch-city, assets)

customer (customer-name, customer-street, customer-city)

account (account-number, branch-name, balance)

loan (loan-number, branch-name, amount)

depositor (customer-name, account-number)

borrower (customer-name, loan-number)

Selection (σ): Example 2

- Find all tuples who have taken **loans** of more than $1200 made by the "Perryridge" **branch**

$$\sigma_{\text{branch-name} = \text{"Perryridge"} \wedge \text{amount} > 1200}(\text{loan})$$

loan-number	branch-name	amount
L-15	Perryridge	1500
L-16	Perryridge	1300

loan

loan-number	branch-name	amount
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

Projection (Π): Example 1

- Find all **loan** numbers and the **amount** of the **loans**

$$\Pi_{\text{loan-number, amount}}\ (\text{loan})$$

loan-number	amount
L-11	900
L-14	1500
L-15	1500
L-16	1300
L-17	1000
L-23	2000
L-93	500

loan

loan-number	branch-name	amount
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

branch (branch-name, branch-city, assets)

customer (customer-name, customer-street, customer-city)

account (account-number, branch-name, balance)

loan (loan-number, branch-name, amount)

depositor (customer-name, account-number)

borrower (customer-name, loan-number)

Projection (II): Example 2

- Find the **loan** number for each loan of an amount greater than $1200

$$\Pi_{loan\text{-}number}\ (\sigma_{amount > 1200}\ (loan))$$

loan-number
L-14
L-15
L-16
L-23

loan

loan-number	branch-name	amount
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Projection (∏): Example 3

- Find those **customers** who lives in "Harrison"

$$\Pi_{\text{customer-name}} (\sigma_{\text{customer-city}=\text{"Harrison"}} (\text{customer}))$$

customer-name
Hayes
Jones

ustomer

ustomer-name	customer-street	customer-city
Adams	Spring	Pittsfield
Brooks	Senator	Brooklyn
Curry	North	Rye
Glenn	Sand Hill	Woodside
Green	Walnut	Stamford
Hayes	Main	Harrison
Johnson	Alma	Palo Alto
Jones	Main	Harrison
Lindsay	Park	Pittsfield
Smith	North	Rye
Turner	Putnam	Stamford
Williams	Nassau	Princeton

branch (branch-name, branch-city, assets)

customer(customer-name, customer-street, customer-city)

account(account-number, branch-name, balance)

loan(loan-number, branch-name, amount)

depositor(customer-name, account-number)

borrower(customer-name, loan-number)

Union (∪): Example 1

- Find the names of all customers who have a loan, an account, or both, from the bank

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

customer-name
Adams
Curry
Hayes
Jackson
Jones
Smith
Williams
Lindsay
Johnson
Turner

branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Intersection(∩): Example 1

- Find the names of all customers who have a loan and an account at bank.

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

customer-name
Hayes
Jones
Smith

branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Set Difference (-): Example 1

- Find the names of all customers who have an account but no loan from the bank.

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

customer-name
Johnson
Lindsay
Turner

branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Cartesian Product (x): Example 1

Find the **names** of all customers who have a **loan** at the Perryridge **branch.**

branch (branch-name, branch-city, assets)

customer(customer-name, customer-street, customer-city)

account(account-number, branch-name, balance)

loan(loan-number, branch-name, amount)

depositor(customer-name, account-number)

borrower(customer-name, loan-number)

borrower × loan

customer-name	borrower. loan-number	loan. loan-number	branch-name	amount
Adams	L-16	L-11	Round Hill	900
Adams	L-16	L-14	Downtown	1500
Adams	L-16	L-15	Perryridge	1500
Adams	L-16	L-16	Perryridge	1300
Adams	L-16	L-17	Downtown	1000
Adams	L-16	L-23	Redwood	2000
Adams	L-16	L-93	Mianus	500
Curry	L-93	L-11	Round Hill	900
Curry	L-93	L-14	Downtown	1500
Curry	L-93	L-15	Perryridge	1500
Curry	L-93	L-16	Perryridge	1300
Curry	L-93	L-17	Downtown	1000
Curry	L-93	L-23	Redwood	2000
Curry	L-93	L-93	Mianus	500
Hayes	L-15	L-11		900
Hayes	L-15	L-14		1500
Hayes	L-15	L-15		1500
Hayes	L-15	L-16		1300
Hayes	L-15	L-17		1000
Hayes	L-15	L-23		2000
Hayes	L-15	L-93		500
...	...	...	...	...
...	...	...	...	...
...	...	...	...	...
Smith	L-23	L-11	Round Hill	900
Smith	L-23	L-14	Downtown	1500
Smith	L-23	L-15	Perryridge	1500
Smith	L-23	L-16	Perryridge	1300
Smith	L-23	L-17	Downtown	1000
Smith	L-23	L-23	Redwood	2000
Smith	L-23	L-93	Mianus	500
Williams	L-17	L-11	Round Hill	900
Williams	L-17	L-14	Downtown	1500
Williams	L-17	L-15	Perryridge	1500
Williams	L-17	L-16	Perryridge	1300
Williams	L-17	L-17	Downtown	1000
Williams	L-17	L-23	Redwood	2000
Williams	L-17	L-93	Mianus	500

$$\Pi_{\text{customer-name}} (\sigma_{\text{branch-name}=\text{"Perryridge"}} (\sigma_{\text{borrower. loan-number = loan. loan-number}} (\text{borrower x loan})))$$

borrower × loan

customer-name	borrower. loan-number	loan. loan-number	branch-name	amount
Adams	L-16	L-11	Round Hill	900
Adams	L-16	L-14	Downtown	1500
Adams	L-16	L-15	Perryridge	1500
Adams	L-16	L-16	Perryridge	1300
Adams	L-16	L-17	Downtown	1000
Adams	L-16	L-23	Redwood	2000
Adams	L-16	L-93	Mianus	500
Curry	L-93	L-11	Round Hill	900
Curry	L-93	L-14	Downtown	1500
Curry	L-93	L-15	Perryridge	1500
Curry	L-93	L-16	Perryridge	1300
Curry	L-93	L-17	Downtown	1000
Curry	L-93	L-23	Redwood	2000
Curry	L-93	L-93	Mianus	500
Hayes	L-15	L-11		900
Hayes	L-15	L-14		1500
Hayes	L-15	L-15		1500
Hayes	L-15	L-16		1300
Hayes	L-15	L-17		1000
Hayes	L-15	L-23		2000
Hayes	L-15	L-93		500
...	...	...	...	...
...	...	...	...	...
...	...	...	...	...
Smith	L-23	L-11	Round Hill	900
Smith	L-23	L-14	Downtown	1500
Smith	L-23	L-15	Perryridge	1500
Smith	L-23	L-16	Perryridge	1300
Smith	L-23	L-17	Downtown	1000
Smith	L-23	L-23	Redwood	2000
Smith	L-23	L-93	Mianus	500
Williams	L-17	L-11	Round Hill	900
Williams	L-17	L-14	Downtown	1500
Williams	L-17	L-15	Perryridge	1500
Williams	L-17	L-16	Perryridge	1300
Williams	L-17	L-17	Downtown	1000
Williams	L-17	L-23	Redwood	2000
Williams	L-17	L-93	Mianus	500

$$\Pi_{customer\text{-}name}\ (\sigma_{branch\text{-}name="Perryridge"}\ (\sigma_{borrower.\ loan\text{-}number\ =\ loan.\ loan\text{-}number}\ (borrower\ x\ loan)))$$

customer-name
Adams
Hayes

Cartesian Product (x): Example 1

Find the **names** of all customers who have a **loan** at the Perryridge **branch.**

$$\Pi_{\text{customer-name}} (\sigma_{\text{branch-name}=\text{"Perryridge"}} (\sigma_{\text{borrower. loan-number} = \text{loan. loan-number}} (\textbf{borrower x loan})$$

branch (branch-name, branch-city, assets)
customer(customer-name, customer-street, customer-city)
account(account-number, branch-name, balance)
loan(loan-number, branch-name, amount)
depositor(customer-name, account-number)
borrower(customer-name, loan-number)

Cartesian Product (x): Example 1

Find the **names** of all customers who have a **loan** at the Perryridge **branch.**

$$\Pi_{\text{customer-name}} (\sigma_{\text{branch-name}=\text{"Perryridge"}} (\sigma_{\text{borrower.loan-number} = \text{loan.loan-number}} (\textbf{borrower x loan})))$$

customer-name
Adams
Hayes

branch (<u>branch-name</u>, branch-city, assets)
customer(<u>customer-name</u>, customer-street, customer-city)
account(<u>account-number</u>, branch-name, balance)
loan(<u>loan-number</u>, branch-name, amount)
depositor(<u>customer-name</u>, <u>account-number</u>)
borrower(customer-name, loan-number)

Cartesian Product (x): Example 2

- Find the **names** of all customers who have a **loan** at the Perryridge **branch** but do **not** have an **account** at any **branch** of the bank.

$$\Pi_{customer\text{-}name}\ (\sigma_{branch\text{-}name\ =\ \text{“Perryridge”}}\ (\sigma_{borrower.\ loan\text{-}number\ =\ loan.\ loan\text{-}number}\ (borrower\ x\ loan)))$$

$$-\ \Pi_{customer\text{-}name}\ (depositor)$$

customer-name
Adams

branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Rename Operator (ρ): Example 1

- Find the largest **account** balance in the bank

Rename Operator (ρ): Example 1

- Find the largest **account** balance in the bank
  - *Strategy*:
    1. Find those balances that are not largest (as a temporary relation)
       - Rename ***account*** relation as ***d*** so that we can compare each account balance with all others

$$\Pi_{account.balance}(\sigma_{account.balance < d.balance}(account \times \rho_d(account)))$$

    2. Use set difference to find those account balances that were not found in the earlier step.
       - Take set difference between relation $\Pi_{balance}(account)$ and **temporary relation** just computed, to obtain the result

$$\Pi_{balance}(account) - \Pi_{account.balance}(\sigma_{account.balance < d.balance}(account \times \rho_d account))$$

account

account-number	*branch-name*	*balance*
A-101	Downtown	500
A-102	Perryridge	400
A-201	Brighton	900
A-215	Mianus	700
A-217	Brighton	750
A-222	Redwood	700
A-305	Round Hill	350

$$\Pi_{balance}(account) - \Pi_{account.balance}(\sigma_{account.balance < d.balance}(account \times \rho_{d}\ account))$$

balance
500
400
900
700
750
700
350

−

balance
500
400
700
750
350

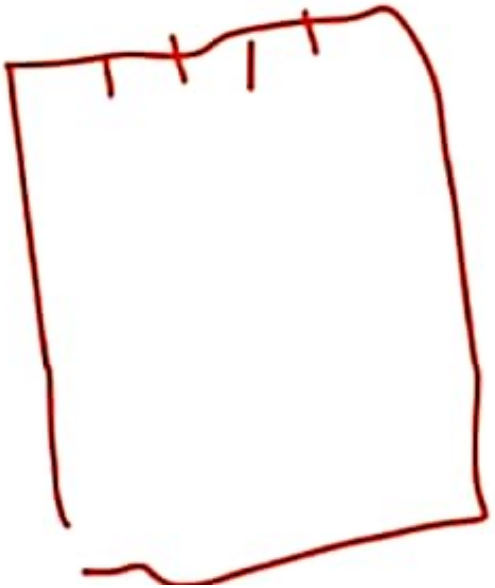
result

balance
900

Introduction

- **Cartesian product** of two relations (A x B), gives us all the possible tuples that are paired together.
  - But it **might not be feasible** in certain cases to take a **Cartesian product** where we **encounter huge relations** with **thousands of tuples** having a considerable **large number of attributes.**

A x B =



Join Operation (⋈)

- **Join** is an **Additional / Derived operator** which simplify the queries, but does not add any new power to the basic relational algebra.
- **Join** is a **combination** of a **Cartesian product** followed by a **selection process**

  Join = Cartesian Product + Selection

- A **Join operation** pairs two tuples from different relations, if and only if a given join condition is satisfied.
- **Symbol:** ⋈
- $A \bowtie_c B = \sigma_c (A \times B)$

$A \bowtie_c B$ =

Difference

Joins (⋈)

- Combination of tuples that satisfy the filtering/matching conditions
- Fewer tuples then cross product, might be able to compute efficiently

Cartesian Product /Cross Product/Cross Join(X)

- All possible combination of tuples from the relations
- Huge number of tuples and costly to manage

R ⋈ S
(Natural Join)

A	B	C
1	a	3
2	b	4

R

A	B
1	a
2	b

S

B	C
a	3
b	4

R x S

A	R.B	S.B	C
1	a	a	3
1	a	b	4
2	b	a	3
2	b	b	4

Types of JOINS

1. Inner Join (Join):

- Theta join
- Equi join
- Natural join

2. Outer join:

(Extension of join)

- Left Outer Join
- Right Outer Join
- Full Outer Join

Types of JOINS

1. Inner Join:

- Contains only those tuples that satisfy the matching condition
  - **Theta(θ) join**
  - **Equi join**
  - **Natural join**

2. Outer join:

- Extension of join
- Contains matching tuples that satisfy the matching condition, along with some or all tuples that do not satisfy the matching condition
- Contains all rows from either one or both relation
  - **Left Outer Join**
  - **Right Outer Join**
  - **Full Outer Join**

Types of JOINS

1. Inner Join:

- Contains only those tuples that satisfy the matching condition

- **Theta(θ) / Conditional Join**
  - $A \bowtie_\theta B$
  - uses all kinds of comparison operators(<,>,<=,>=,=,≠)
- **Equi Join**
  - Special case of theta join
  - uses only equality (=) comparison operator
- **Natural join**
  - $A \bowtie B$
  - Based on common attributes in both relation
  - does not use any comparison operator

2. Outer join:

- Contains matching tuples that satisfy the matching condition, along with some or all tuples that do not satisfy the matching condition
- Contains all rows from either one or both relation

- **Left Outer Join**
  - Left relation tuples will always be in result whether the value is matched or not
- **Right Outer Join**
  - Right relation tuples will always be in result whether the value is matched or not
- **Full Outer Join**
  - Tuples from both relations are present in result, whether the value is matched or not

INNER JOIN:

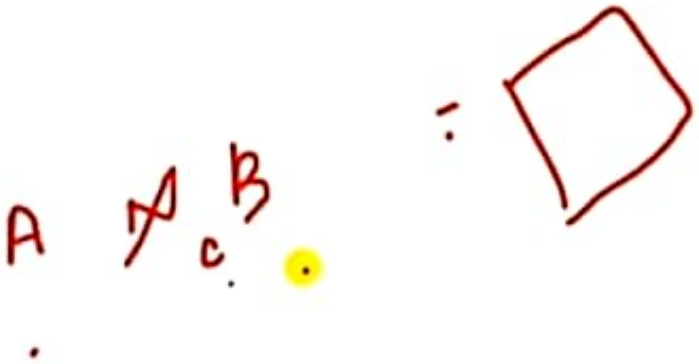
THETA JOIN

EQUI JOIN

NATURAL JOIN

Inner Join

- An **Inner join** includes **only those tuples that satisfy the matching criteria, while the rest of tuples are excluded.**
- Theta Join, Equi join, and Natural Join are called **inner joins.**



1. Theta(θ) / Conditional join

- **Theta join / Conditional Join**
  - It combines tuples from different relations provided they satisfy the **theta (θ) condition.**
  - It is a **general case of join.** And it is **used** when we want to **join** two or more relation based on some **conditions.**
  - The **join condition** is denoted by the symbol **θ.**
  - It **uses** all kinds of **comparison operators** like <, >, <=, >=, =, ≠
- **Notation**: $A \bowtie_{\theta} B$ ✓

  Where **θ** is a **predicate/condition.** It can use any comparison operator (<, >, <=, >=, =, ≠)
- $A \bowtie_{\theta} B = \sigma_{\theta}(A \times B)$

Example: Theta Join

S1

sid	name	rating	age
22	dustin	7	45.0
31	lubber	8	55.0
58	rusty	10	35.0

R1

sid	bid	day
22	101	10/10/96
58	103	11/12/96

$$S1 \bowtie_{S1.sid < R1.sid} R1$$

S1.Sid	sname	rating	age	R1.sid	bid	day
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.0	58	103	11/12/96

Equivalent to: $\sigma_{S1.sid < R1.sid}(S1 \times R1)$

So, $A \bowtie_\theta B = \sigma_\theta (A \times B)$

2. Equi Join

- When a theta join uses only equivalence (=) condition, it becomes a **Equi join**.
- **Equi join** is a **special case of theta (or conditional) join** where condition contains ***equalities*** (=).
- ***Notation:*** $A \bowtie_{A.a1 = B.b1 \wedge \ldots\ldots\ldots\ldots \wedge A.an = B.bn} B$

2. Equi Join

- When a theta join uses only equivalence (=) condition, it becomes a **Equi join**.
- **Equi join** is a **special case of theta (or conditional) join** where condition contains ***equalities*** (=).
- ***Notation:*** $A \bowtie_{A.a1 = B.b1 \wedge \ldots\ldots\ldots\ldots \wedge A.an = B.bn} B$

θ

Example 1: Equi Join

S1

sid	name	rating	age
22	dustin	7	45.0
31	lubber	8	55.0
58	rusty	10	35.0

R1

sid	bid	day
22	101	10/10/96
58	103	11/12/96

$$S1 \bowtie_{S1.sid = R1.sid} R1$$

S1.Sid	sname	rating	age	R1.sid	bid	day
22	dustin	7	45.0	22	101	10/10/96
58	rusty	10	35.0	58	103	11/12/96

Equivalent to $\sigma_{S1.sid = R1.sid}(S1 \times R1)$

Example 2: Equi Join

Student

SID	Name	Std
101	Rohan	11
102	Mira	12

Subjects

Class	Subject
11	Maths
11	Physics
12	English
12	Chemistry

Student ⋈ *Student. Std = Subjects. Class* Subject

SID	Name	Std	Class	Subject
101	Rohan	11	11	Maths
101	Rohan	11	11	Physics
102	Mira	12	12	English
102	Mira	12	12	Chemistry

3. Natural Join

- **Natural join** can only be performed ***if there is at least one common attribute*** (column) that exist between two relations. In addition, the **attributes** must have the **same name and domain.**
- Natural join does not use any comparison operator.
- ***It is same as equi join which occurs implicitly*** *by comparing all the* **common attributes (columns)** *in both relation,* **but** difference is that ***in Natural Join the common attributes appears only once.*** *The resulting schema will change.*
- **Notation**: **A ⋈ B**
- *The* **result of the natural join** *is the* **set of all combinations of tuples** *in two relations* **A** *and* **B** *that are* **equal on their common attribute names**.

Natural Join...

Note:

- The **Natural Join** of two relations can be ***obtained*** *by applying a* ***Projection operation to Equi join*** *of two relations*. In terms of basic operators:

  Natural Join = Cartesian product + Selection + Projection

- **Natural Join (⋈) is by default inner join** because the tuples which does not satisfy the conditions of join does not appear in result set.
- **Natural Join** is very important.

Example 1:

r = (A, B, C, D) s = (B, D, E)

- Resulting schema of r ⋈ s = (A, B, C, D, E)
- r ⋈ s is defined as:

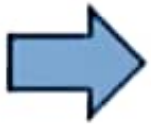
$$\Pi_{r.A,\, r.B,\, r.C,\, r.D,\, s.E}\left(\sigma_{r.B = s.B \,\wedge\, r.D = s.D}\left(r \times s\right)\right)$$

r

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

s

B	D	E
1	a	α
3	a	β
1	a	γ
2	b	δ
3	b	ϵ



r ⋈ s

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

Example 2: Natural Join

Courses

CID	Course	Dept
CS01	Database	CS
ME01	Mechanics	ME
EE01	Electronics	EE

HOD

Dept	Head
CS	Rohan
ME	Sara
EE	Jiya

Courses ⋈ HOD

CID	Course	Dept	Head
CS01	Database	CS	Rohan
ME01	Mechanics	ME	Sara
EE01	Electronics	EE	Jiya

Equivalent to: $\Pi_{CID,\ Course,\ Courses.Dept,\ Head}\ (\sigma_{Courses.Dept\ =\ HOD.Dept}(Courses \times HOD))$

Example 3: Natural Join

S1

sid	name	rating	age
22	dustin	7	45.0
31	lubber	8	55.0
58	rusty	10	35.0

R1

sid	bid	day
22	101	10/10/96
58	103	11/12/96

S1 ⋈ R1

sid	sname	rating	age	bid	day
22	dustin	7	45.0	101	10/10/96
58	rusty	10	35.0	103	11/12/96

Equivalent to: $\Pi_{S1.sid,\ sname,\ rating,\ age,\ bid,\ day}\ (\sigma_{S1.sid\ =\ R1.sid}\ (S1 \times R1))$


NATURAL JOIN

Natural Join

- **Natural join** can only be performed ***if there is at least one common attribute*** (column) that exist between two relations. In addition, the **attributes** must have the **same name and domain.**
- *In **Natural Join** resulting relation the **common attributes appears only once.***
- **Notation**: $A \bowtie B$ (Natural join does not use any comparison operator)
- *The **result of the natural join** is the **set of all combinations of tuples** in two relations **A** and **B** that are **equal on their common attribute names**.*
- The **Natural Join** of two relations can be ***obtained*** *by applying a **projection operation to equi join** of two relations*. And in terms of basic operators:

Natural Join = Cartesian product + Selection + Projection

Example 1:

r = (A, B, C, D) s = (B, D, E)

- Resulting schema of r ⋈ s = (A, B, C, D, E)

r

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

s

B	D	E
1	a	α
3	a	β
1	a	γ
2	b	δ
3	b	ϵ

Example 1:

r = (A, B, C, D) s = (B, D, E)

- Resulting schema of r ⋈ s = (A, B, C, D, E)
- r ⋈ s is defined as:

$$\Pi_{r.A,\, r.B,\, r.C,\, r.D,\, s.E}\left(\sigma_{r.B = s.B \,\wedge\, r.D = s.D}\,(r \times s)\right)$$

r

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

s

B	D	E
1	a	α
3	a	β
1	a	γ
2	b	δ
3	b	ϵ

r ⋈ s

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

Example: Natural Join

Courses

CID	Course	Dept
CS01	Database	CS
ME01	Mechanics	ME
EE01	Electronics	EE

HOD

Dept	Head
CS	Rohan
ME	Sara
EE	Jiya

Courses ⋈ HOD

CID	Course	Dept	Head
CS01	Database	CS	Rohan
ME01	Mechanics	ME	Sara
EE01	Electronics	EE	Jiya

Example: Natural Join

Courses

CID	Course	Dept
CS01	Database	CS
ME01	Mechanics	ME
EE01	Electronics	EE

HOD

Dept	Head
CS	Rohan
ME	Sara
EE	Jiya

Courses ⋈ HOD

CID	Course	Dept	Head
CS01	Database	CS	Rohan
ME01	Mechanics	ME	Sara
EE01	Electronics	EE	Jiya

Equivalent to: $\Pi_{CID,\ Course,\ Courses.Dept,\ Head}\ (\sigma_{Courses1.Dept\ =\ HOD.Dept}(Courses\ x\ HOD))$

Query 1:

- Find the names of all customers who have a loan at the bank, along with the loan number and the loan amount.

- ***Using Cartesian Product:***

$$\Pi_{\text{customer-name, loan. loan-number, amount}} (\sigma_{\text{borrower. loan-number = loan. loan-number}} (\text{borrower x loan}))$$

- ***Using Natural Join:***

$$\Pi_{\text{customer-name, loan.loan-number, amount}} (\text{borrower} \bowtie \text{loan})$$

branch (branch-name, branch-city, assets)
customer(customer-name, customer-street, customer-cit
account(account-number, branch-name, balance)
loan(loan-number, branch-name, amount)
depositor(customer-name, account-number)
borrower(customer-name, loan-number)

Query 2:

- Find the names of all branches with customers who have an account in the bank and who live in Harrison.

$$\Pi_{branch\text{-}name}\ (\sigma_{customer\text{-}city=\text{“Harrison”}}\ (customer \bowtie account \bowtie depositor))$$

branch (branch-name, branch-city, assets)
customer(customer-name, customer-street, customer-city)
account(account-number, branch-name, balance)
loan(loan-number, branch-name, amount)
depositor(customer-name, account-number)
borrower(customer-name, loan-number)

- **Note:** Natural join is associative

$$((customer \bowtie account) \bowtie depositor) = (customer \bowtie (account \bowtie depositor))$$

So we can write it $(customer \bowtie account \bowtie depositor)$

Query 3:

- Find all customers who have ***both a loan and an account*** *at the bank.*
  - ***Using Natural Join:***

    $\Pi_{customer\text{-}name}$ (**borrower** ⋈ **depositor**))

  - ***Using Set intersection:***

    $\Pi_{customer\text{-}name}$ (**borrower**) ∩ $\Pi_{customer\text{-}name}$ (**depositor**)

branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Query 4:

Find the **names** of all customers who have a **loan** at the Perryridge **branch.**

Using Cartesian Product:

$$\Pi_{\text{customer-name}} (\sigma_{\text{branch-name}=\text{"Perryridge"}} (\sigma_{\text{borrower. loan-number} = \text{loan. loan-number}} (\text{borrower x loan})))$$

Using Natural Join:

$$\Pi_{\text{customer-name}} (\sigma_{\text{branch-name}=\text{"Perryridge"}} (\text{borrower} \bowtie \text{loan}))$$

branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Query 5:

- Find the **names** of all customers who have a **loan** at the Perryridge **branch** but do **not** have an **account** at any **branch** of the bank.

$$\Pi_{customer\text{-}name}\ (\sigma_{branch\text{-}name=\text{“}Perryridge\text{”}}\ (borrower \bowtie loan)) - \Pi_{customer\text{-}name}\ (depositor)$$

branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

OUTER JOIN:

LEFT-OUTER JOIN

RIGHT-OUTER JOIN

FULL-OUTER JOIN

SHANU KUTTAN
CSE CLASSES

DBMS

Outer Join

- *An **Inner join** includes only those tuples with matching attributes and the rest are discarded in the resulting relation. Therefore, we need to use **outer joins** to include all the rest of the tuple from the participating relations in the resulting relation.*
- The **outer join** operation is an **extension of the join operation** that **avoids loss of information.**
- **Outer Join** contains matching tuples that satisfy the matching condition, along with some or all tuples that do not satisfy the matching condition.
  - It is based on both matched or unmatched tuple.
  - It contains all rows from either one or both relations are present
- *It uses **NULL** values.*
  - *NULL* signifies that the value is unknown or does not exist

Example: Inner join (Natural Join)

Courses

CID	Course
100	Database
101	Mechanics
102	Electronics

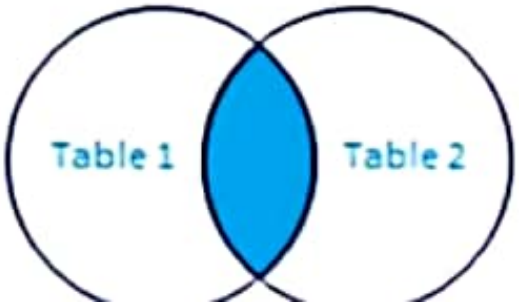
HOD

CID	Name
100	Rohan
102	Sara
104	Jiya

Courses ⋈ HOD

CID	Course	Name
100	Database	Rohan
102	Electronics	Sara

Inner Join



Outer Join Types

- **Outer Join = Natural Join + Extra information**

  (from left table, right table or both table)

- There are **three kinds of outer joins:**

  1. **Left outer join**
  2. **Right outer join**
  3. **Full outer join**

1. Left outer join (R1 ⟕ R2)

When applying join on two relations R1 and R2, some tuples of R1 or R2 does not appear in result set which does not satisfy the join conditions. But..

- In **Left outer join**, ***all the tuples from the Left relation R1 are included in the resulting relation.*** The tuples of R1 which do not satisfy join condition will have values as **NULL** for attributes of R2.
- In short:
  - **All** record from **left** table
  - Only matching records from right table
- **Symbol**: ⟕
- **Notation**: **R1 ⟕ R2**

Example: Left outer join

Courses

CID	Course
100	Database
101	Mechanics
102	Electronics

HOD

CID	Name
100	Rohan
102	Sara
104	Jiya

Courses $\overline{\bowtie}$ HOD

Example: Left outer join

Courses

CID	Course
100	Database
101	Mechanics
102	Electronics

HOD

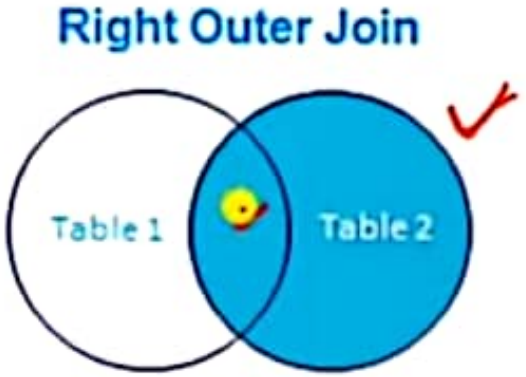
CID	Name
100	Rohan
102	Sara
104	Jiya

Courses ⋈ HOD

CID	Course	Name
100	Database	Rohan
101	Mechanics	NULL
102	Electronics	Sara

2. Right outer join (R1 ⟖ R2)

- In **Right outer join**, ***all the tuples from the right relation R2 are included in the resulting relation.*** The tuples of R2 which do not satisfy join condition will have values as **NULL** for attributes of R1.
- In short:
  - **All** record from **right** table
  - Only matching records from left table
- **Symbol**: ⟖
- **Notation**: **R1 ⟖ R2**



Example: Right outer join

Courses

CID	Course
100	Database
101	Mechanics
102	Electronics

HOD

CID	Name
100	Rohan
102	Sara
104	Jiya

Courses ⟖ HOD

Example: Right outer join

Courses

CID	Course
100	Database
101	Mechanics
102	Electronics

HOD

CID	Name
100	Rohan
102	Sara
104	Jiya

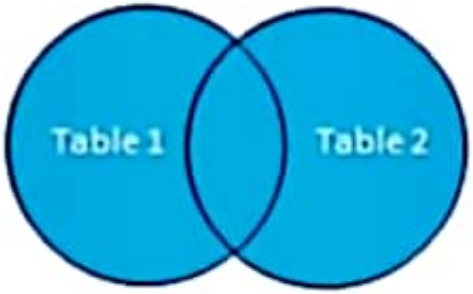
Courses ⟖ HOD

CID	Course	Name
100	Database	Rohan
102	Electronics	Sara
104	NULL	Jiya

3. Full outer join (R1 ⟗ R2)

- In **Full outer join**, ***all the tuples from both Left relation R1 and right relation R2 are included in the resulting relation.*** The tuples of both relations R1 and R2 which do not satisfy join condition, their respective unmatched attributes are made **NULL.**
- In short:
  - All record from **all** table
- **Symbol**: ⟗
- **Notation**: **R1 ⟗ R2**

Full Outer Join



Example: Full outer join

Courses

CID	Course
100	Database
101	Mechanics
102	Electronics

HOD

CID	Name
100	Rohan
102	Sara
104	Jiya

Courses ⋈ HOD

CID	Course	Name
100	Database	Rohan
101	Mechanics	**NULL**
102	Electronics	Sara
104	**NULL**	Jiya

- Relation **Loan**

loan-number	branch-name	amount
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

Inner Join (Natural Join)– Example

loan

loan-number	branch-name	amount
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

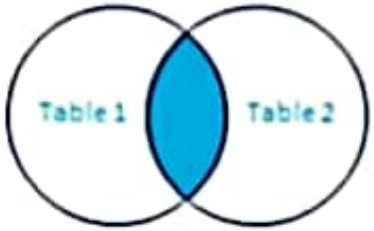
borrower

customer-name	loan-number
Jones	L-170
Smith	L-230
Hayes	L-155

Loan ⋈ borrower

loan-number	branch-name	amount	customer-name
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

Inner Join



Inner Join (Natural Join)– Example

loan

loan-number	branch-name	amount
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

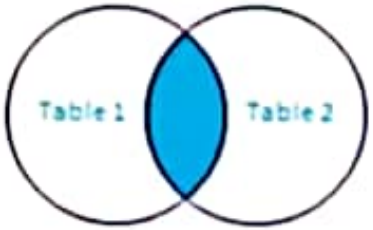
borrower

customer-name	loan-number
Jones	L-170
Smith	L-230
Hayes	L-155

Loan ⋈ borrower

loan-number	branch-name	amount	customer-name
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

Inner Join



Left Outer Join – Example

loan

loan-number	branch-name	amount
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

borrower

customer-name	loan-number
Jones	L-170
Smith	L-230
Hayes	L-155

Loan ⟕ borrower

loan-number	branch-name	amount	customer-name
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	*NULL*

Left Outer Join – Example

loan

loan-number	branch-name	amount
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

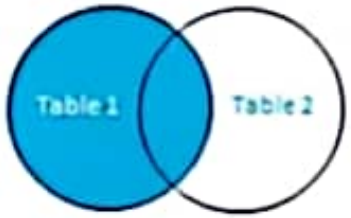
borrower

customer-name	loan-number
Jones	L-170
Smith	L-230
Hayes	L-155

Loan ⟕ borrower

loan-number	branch-name	amount	customer-name
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	*NULL*

Left Outer Join



Right Outer Join – Example

loan

loan-number	branch-name	amount
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

borrower

customer-name	loan-number
Jones	L-170
Smith	L-230
Hayes	L-155

Loan ⟖ borrower

loan-number	branch-name	amount	customer-name
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	NULL	NULL	Hayes

Full Outer Join – Example

loan

loan-number	branch-name	amount
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

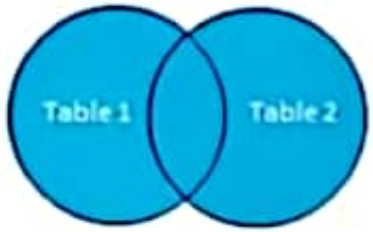
borrower

customer-name	loan-number
Jones	L-170
Smith	L-230
Hayes	L-155

Loan ⟗ borrower

loan-number	branch-name	amount	customer-name
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	*NULL*
L-155	*NULL*	*NULL*	Hayes

Full Outer Join




DIVISION OPERATOR

Division Operator (÷, /)

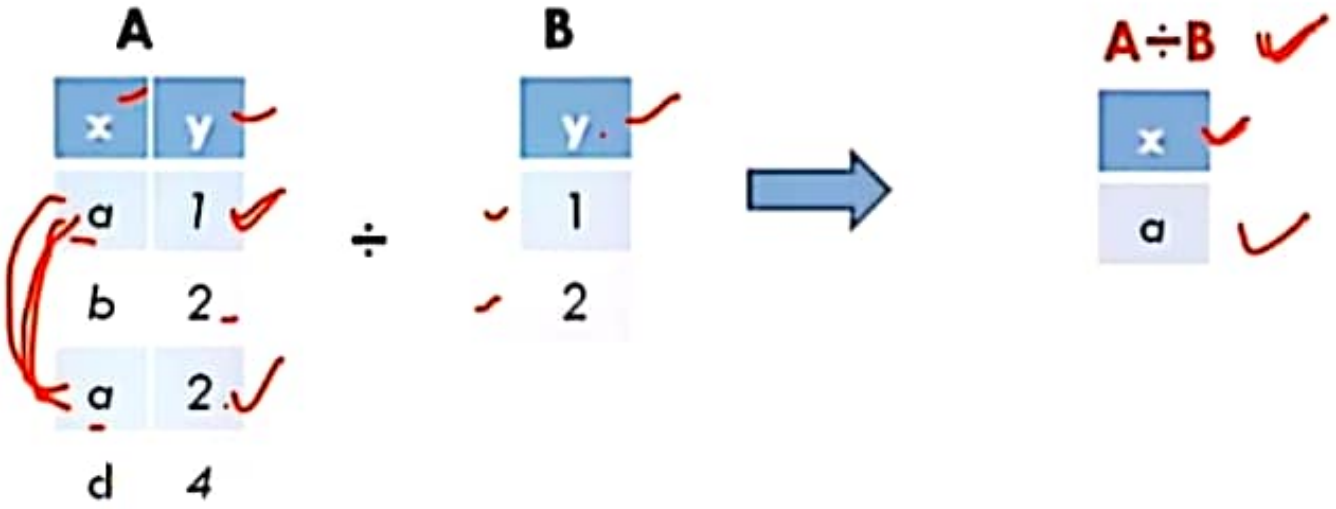
- Division operator is a Derived Operator, not supported as a primitive operator
- Suited to queries that include the keyword "**all**", or "**every**" like "at all", "for all" or "in all", "at every", "for every" or "in every". Eg:
  - *Find the person that has account in **all** the banks of a particular city*
  - *Find sailors who have reserved **all** boats.*
  - *Find employees who works on **all** projects of company.*
  - *Find students who have registered for **every** course.*
- In all these queries, the description after the keyword **"all"** or **"every"** *defines a set which contains some elements* and the **final result** *contains those records who satisfy these requirements.*
- **Notation**: **A÷B** or **A/B** where A and B are two relations

Division Operator (÷)...

Division operator can be applied if and only if:

- Attributes of B is **proper subset** of Attributes of A.
- The relation returned by **division operator** will have **attributes =(All attributes of A – All attributes of B)**
- The relation returned by **division operator** will **return** those **tuples from relation A which are associated t every B's tuple**.

Example 1:



Example 2: shows how it works

A ✓

sno	pno
S1	P1
S1	P2
S1	P3
S1	P4
S2	P1
S2	P2
S3	P2
S4	P2
S4	P4

B1 ✓

pno
P2

B2

pno
P2
P4

B3

pno
P1
P2
P4

Example 2: shows how it works

A

sno	pno
S1	P1
S1	P2
S1	P3
S1	P4
S2	P1
S2	P2
S3	P2
S4	P2
S4	P4

B1

pno
P2

B2

pno
P2
P4

B3

pno
P1
P2
P4

A/B1

sno
S1
S2
S3
S4

A/B2

sno
S1
S4

A/B3

sno
S1

Expressing A/B Using Basic Operators

- Division is a **derived operator** (or additional operator).
- Division can be expressed in terms of **Cross Product, Set-Difference and Projection.**

Idea:

- For A/B, compute all x values that are not 'disqualified' by some y value in B.
  - x value is disqualified if by attaching y value from B, we obtain an **xy tuple** that is not in A.

Disqualified x values: $\Pi_x((\Pi_x(A) \times B) - A)$

So $A/B = \Pi_x(A) -$ all disqualified tuples

$A/B = \Pi_x(A) - \Pi_x((\Pi_x(A) \times B) - A)$

A		B	A÷B
x	y	y	x
a	1	1	a
b	2	2	
a	2		
d	4		

A

sno	pno
S1	P1
S1	P2
S1	P3
S1	P4
S2	P1
S2	P2
S3	P2
S4	P2
S4	P4

B

pno
P2
P4

A/B

sno
S1
S4

A

sno	pno
S1	P1
S1	P2
S1	P3
S1	P4
S2	P1
S2	P2
S3	P2
S4	P2
S4	P4

B

pno
P2
P4

Disqualified x values: $\Pi_{sno}((\Pi_{sno}(A) \times B) - A)$

$\Pi_{sno}(A)$

sno
S1
S2
S3
S4

B

pno
P2
P4

$(\Pi_{sno}(A) \times B)$

sno	pno
S1	P2
S2	P2
S3	P2
S4	P2
S1	P4
S2	P4
S3	P4
S4	P4

A

sno	pno
S1	P1
S1	P2
S1	P3
S1	P4
S2	P1
S2	P2
S3	P2
S4	P2
S4	P4

$((\Pi_{sno}(A) \times B) - A)$

sno	pno
S2	P4
S3	P4

$\Pi_{sno}((\Pi_{sno}(A)$

sno
S2
S3

$A/B = \Pi_{sno}(A) -$ all disqualified tuples

$A/B = \Pi_{sno}(A) - \Pi_{sno}((\Pi_{sno}(A) \times B) - A)$

A/B

sno
S1
S4

A

sno	pno
S1	P1
S1	P2
S1	P3
S1	P4
S2	P1
S2	P2
S3	P2
S4	P2
S4	P4

B

pno
P2
P4

Disqualified x values: $\Pi_{sno}((\Pi_{sno}(A) \times B) - A)$

$\Pi_{sno}(A)$

sno
S1
S2
S3
S4

B

pno
P2
P4

$(\Pi_{sno}(A) \times B)$

sno	pno
S1	P2
S2	P2
S3	P2
S4	P2
S1	P4
S2	P4
S3	P4
S4	P4

A

sno	pno
S1	P1
S1	P2
S1	P3
S1	P4
S2	P1
S2	P2
S3	P2
S4	P2
S4	P4

$((\Pi_{sno}(A) \times B) - A)$

sno	pno
S2	P4
S3	P4

$\Pi_{sno}((\Pi_{sno}(A)$

sno
S2
S3

$A/B = \Pi_{sno}(A) - \text{all disqualified tuples}$

$A/B = \Pi_{sno}(A) - \Pi_{sno}((\Pi_{sno}(A) \times B) - A)$

sno
S1
S2
S3

−

sno
S2
S3

A/B

sno
S1
S4

Example 3:

A

A	B	C	D	E
α	a	α	a	1
α	a	γ	a	1
α	a	γ	b	1
β	a	γ	a	1
β	a	γ	b	3
γ	a	γ	a	1
γ	a	γ	b	1
γ	a	β	b	1

÷

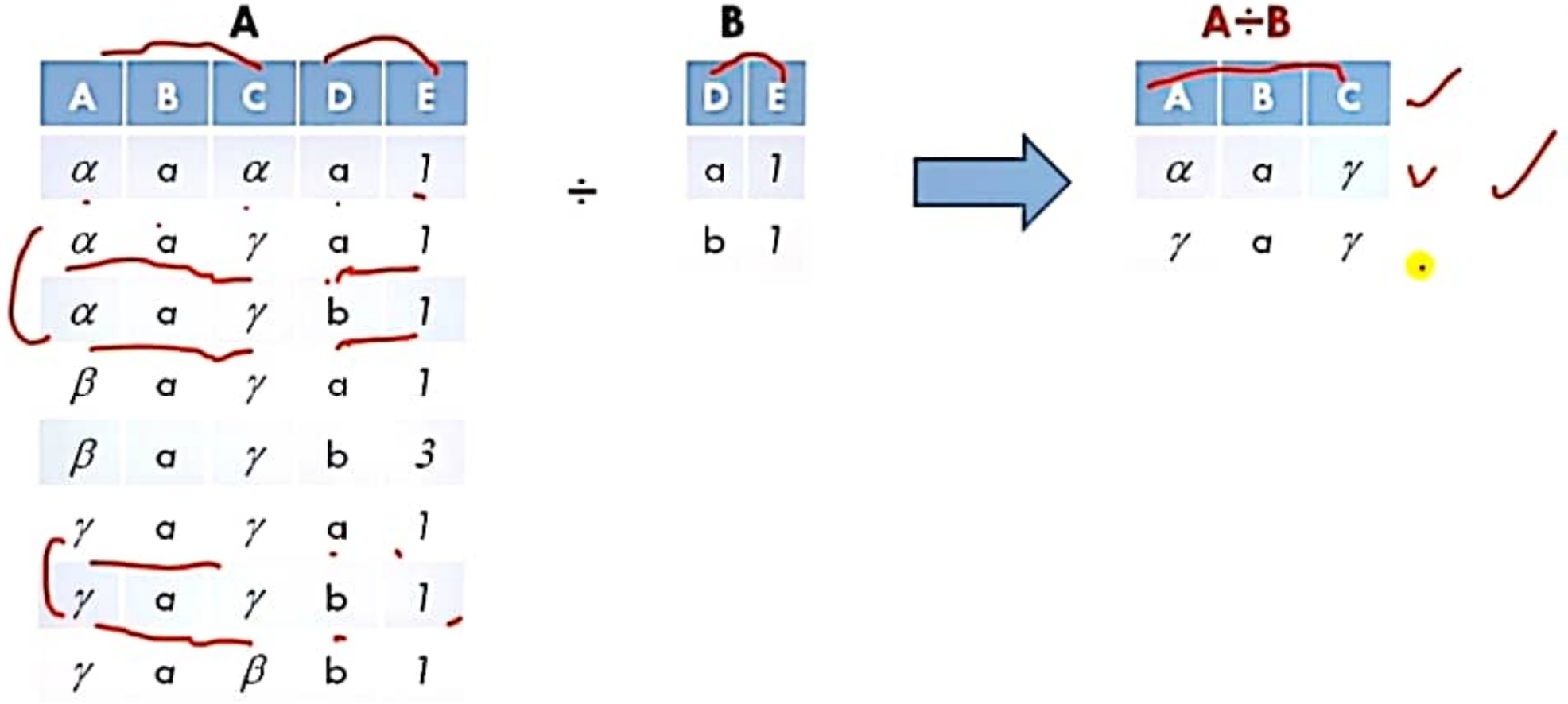
B

D	E
a	1
b	1

A÷B

A	B	C
α	a	γ
γ	a	γ

Example 3:



A

A	B	C	D	E
α	a	α	a	1
α	a	γ	a	1
α	a	γ	b	1
β	a	γ	a	1
β	a	γ	b	3
γ	a	γ	a	1
γ	a	γ	b	1
γ	a	β	b	1

÷

B

D	E
a	1
b	1

A÷B

A	B	C
α	a	γ
γ	a	γ

branch

branch-name	branch-city	assets
Brighton	Brooklyn	7100000
Downtown	Brooklyn	9000000
Mianus	Horseneck	400000
North Town	Rye	3700000
Perryridge	Horseneck	1700000
Pownal	Bennington	300000
Redwood	Palo Alto	2100000
Round Hill	Horseneck	8000000

customer

customer-name	customer-street	customer-city
Adams	Spring	Pittsfield
Brooks	Senator	Brooklyn
Curry	North	Rye
Glenn	Sand Hill	Woodside
Green	Walnut	Stamford
Hayes	Main	Harrison
Johnson	Alma	Palo Alto
Jones	Main	Harrison
Lindsay	Park	Pittsfield
Smith	North	Rye
Turner	Putnam	Stamford
Williams	Nassau	Princeton

depositor

customer-name	account-number
Hayes	A-102
Johnson	A-101
Johnson	A-201
Jones	A-217
Lindsay	A-222
Smith	A-215
Turner	A-305

account

account-number	branch-name	balance
A-101	Downtown	500
A-102	Perryridge	400
A-201	Brighton	900
A-215	Mianus	700
A-217	Brighton	750
A-222	Redwood	700
A-305	Round Hill	350

loan

loan-number	branch-name	amount
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

borrower

customer-name	loan-number
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

Query:

Find all customers who have an account at all branches located 'brooklyn' city

- 1st obtain all branches in Brooklyn by the expression:

$$r1 = \Pi_{branch\text{-}name}(\sigma_{branch\text{-}city=\text{"Brooklyn"}}(branch))$$

branch-name
Brighton
Downtown

- 2nd we can find all (customer-name, branch-name) pairs for which the customer has an account at
- branch by writing

$$r2 = \Pi_{customer\text{-}name,\ branch\_name}(depositor \bowtie account)$$

customer-name	branch-name
Hayes	Perryridge
Johnson	Downtown
Johnson	Brighton
Jones	Brighton
Lindsay	Redwood
Smith	Mianus
Turner	Round Hill

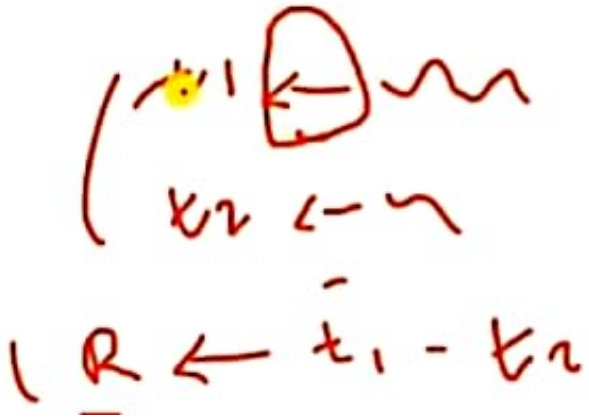
- Now, we need to find customers who appear in r2 with every branch name in r1. So r2 ÷ r1

$$\Pi_{customer\text{-}name,\ branch\_name}(depositor \bowtie account) \div \Pi_{branch\text{-}name}(\sigma_{branch\text{-}city=\text{"Brooklyn"}}(branch))$$

customer-name
Johnson

Assignment Operator

- The **assignment operation** (←) **provides** a **convenient way** to express complex queries.
  - It **writes query** as a sequential program consisting of:
    - a series of **assignments**
    - followed by an expression whose value is displayed as a **result** of the query.
  - **Assignment** must always be made to a **temporary relation variable.**



Example:

- **Division operation** $A \div B = \prod_x(A) - \prod_x((\prod_x(A) \times B) - A)$
- Write $A \div B$ as

$$temp1 \leftarrow \prod_x (A)$$
$$temp2 \leftarrow \prod_x ((temp1 \times B) - A)$$
$$result \leftarrow temp1 - temp2$$

  - The result to the right of the ← is assigned to the relation variable on the left of the ←
  - May use variable in subsequent expressions.

EXTENDED RELATIONAL ALGEBRA OPERATIONS

Extended Relational Algebra

Extended Relational Algebra ***increases power over basic relational algebra.***

1. Generalized Projection
2. Aggregate Functions
3. Outer Join

1. Generalized Projection

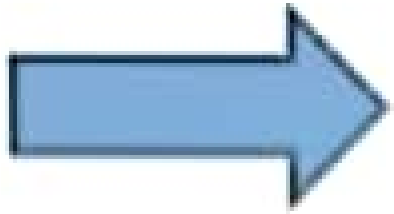
- Normal ***projection*** only projects the columns where as ***generalized projection*** allows arithmetic operations on those projected columns.
- **Generalized Projection** extends the ***projection operation*** by allowing ***arithmetic functions*** to be used in the ***projection list.***

$$\Pi_{F1, F2, \ldots, Fn}(E)$$

  - ***E*** is any relational-algebra **expression**
  - Each of $F_1, F_2, \ldots, F_n$ are **arithmetic expressions** involving ***constants and attributes*** in the schema of *E*.

r

A	B	C
a	1	5
a	2	5
b	3	8
b	4	10



$\Pi_{A,\ 2C-}$

A	C - B
a	9
a	8
b	13
b	16

Example 1:

r

A	B	C
a	1	5
a	2	5
b	3	8
b	4	10

$\Pi_{A,\ C-B}(r)$

A	C - B
a	4
a	3
b	5
b	6

Example Query

- Given relation:

credit_info(customer_name, limit, credit_balance)

customer_name	limit	credit_balance
A	5000	2000
B	6000	4000
C	10000	6000

"Find how much more each person can spend ?"

$$\Pi_{customer\_name,\ limit - credit\_balance}\ (credit\_info)$$

customer_name	Limit - credit_balance
A	3000
B	2000
C	4000

Aggregate Functions and Operations

- **Aggregation function** *takes* ***a collection of values*** *and* ***returns a single value*** *as* ***a result.***

  - **avg:** average value
  - **min:** minimum value
  - **max:** maximum value
  - **sum:** sum of values
  - **count:** number of values

  - These operations can be applied on entire relation or certain groups of tuples.
  - It ignore **NULL** values except count

- *Generalize form (g)* of **Aggregate operation**:

$$G_1, G_2, \ldots, G_n \; g \; F_1(A_1), F_2(A_2), \ldots, F_n(A_n) \; (E)$$

  - *E* is any relational-algebra expression
  - $G_1, G_2 \ldots, G_n$ is a list of attributes on which to group (can be empty)
  - Each F_i is an aggregate function
  - Each A_i is an attribute name

Aggregate Operation – Example 1

Relation r

A	B	C
a	a	5
a	b	5
b	c	2
b	d	3

$$g_{sum(c)}(r)$$

sum (c)
20

$$_{A}g_{sum(c)}(r)$$

A	sum (c)
a	10
b	5

Aggregate Operation – Example 2

Relation '*account*' grouped by *branch-name*:

branch_name	account_number	balance
Perryridge	A-102	400
Perryridge	A-201	900
Brighton	A-217	750
Brighton	A-215	750
Redwood	A-222	700

$$branch\_name\ \mathcal{G}\ sum(balance)\ (account)$$

Aggregate Operation – Example 2

Relation '*account*' grouped by *branch-name*:

branch_name	account_number	balance
Perryridge	A-102	400
Perryridge	A-201	900
Brighton	A-217	750
Brighton	A-215	750
Redwood	A-222	700

$branch\_name\ \mathcal{G}\ sum(balance)\ (account)$

branch_name	sum(balance)
Perryridge	1300
Brighton	1500
Redwood	700

Aggregate Functions ...

- **Result of aggregation *does not have a name***
  - Can use **rename** operation to give it a name
  - For convenience, we permit renaming as part of aggregate operation using **'as'** keyword

$$branch\_name\ \mathcal{G}\ sum(balance)\ as\ sum\_balance\ (account)$$

NULL Values

- *It is possible for* ***tuples*** *to have a* ***null value***, denoted by ***null***, for some of their attributes
- ***null*** signifies
  - **an unknown value/missing**, or
  - **a value that does not exist**

Emp_Id	Name	Phone_No	Passport_No
001	Rahul	7777777777	111 111 111
002	Anil	8888888888	NULL

First_Name	Middle_Name	Last Name
Ranjit	Singh	Thakur
Amit	NULL	Chopra

NULL Values...

- The result of any **arithmetic expression** (+, -, *, /) involving ***null*** must return a ***null***.
  - *Eg:* *5 + null = null*
    *null * 5 = null*
- **Aggregate functions** simply **ignore null values** (as in SQL)
  - Eg: Aggregate functions ***avg, min, max, sum*** ignores *null values* **except** for ***count***
- For **duplication, elimination** and **grouping**, **null** is ***treated like any other value***, and two nulls are assumed to be the same (as in SQL)

NULL Values ...

- **Comparisons** (<, <=, >,>=, =, ≠) with **null values** evaluate to *special value* ***unknown*** (as in SQL)
  - Because we are not sure whether the result is *true* or *false*
  - *Eg:* *5 = null*, *null > 5*, *5 > null*, *null = null* ⟶ ***unknown***
- **Comparisons** in **Boolean expressions** involving **AND, OR, NOT** operations uses ***three-valued logic*** i.e. true (1), false (0), unknown (as in SQL)
- ***Three-valued logic*** using the truth value ***unknown***:
  - **OR:** (*unknown* **or** *true*) = *true*,
    (*unknown* **or** *false*) = *unknown*
    (*unknown* **or** *unknown*) = *unknown*
  - **AND:** (*true* **and** *unknown*) = *unknown*,
    (*false* **and** *unknown*) = *false*,
    (*unknown* **and** *unknown*) = *unknown*
  - **NOT:** (**not** *unknown*) = *unknown*
- Result of **select predicate** is treated as ***false*** if it evaluates to ***unknown***

MODIFICATION OF THE DATABASE

Modification of the Database

- The **content of the database** may be **modified** using the following operations:
  1. **Deletion**
  2. **Insertion**
  3. **Updating**
- All these operations are expressed using the **assignment operator** (←).
  - *Eg:* r_{new} ← **operations on** (r_{old})

1. Deletion

- A **delete** request is expressed **similarly** to a **query**, except *instead of displaying tuples to the user*, ***the selected tuples are removed from the database.***
- ***In Deletion, tuples are deleted from the relation***
- **Can delete only whole tuples**; cannot delete values on only particular attributes
- A **deletion** is expressed in relational algebra by:

$$r \leftarrow r - E$$

where r is a relation and E is a relational algebra expression.

Example: Deletion

$$r \leftarrow r - \sigma_{A=2}(r)$$ ✓

r

A	B	C
1	1	10
1	2	10
2	3	10
2	4	20

Example: Deletion

r

A	B	C
1	1	10
1	2	10
2	3	10
2	4	20

$r \leftarrow r - \sigma_{A=2}(r)$

A	B	C
1	1	10
1	2	10

$r \leftarrow r - \{(1, 2, 10)\}$

A	B	C
1	1	10
2	3	10
2	4	20

Deletion Examples Query

- Delete all account records in the Perryridge branch.

$$account \leftarrow account - \sigma_{branch\text{-}name = \text{"Perryridge"}}(account)$$

- Delete all loan records with amount in the range of 0 to 50

$$loan \leftarrow loan - \sigma_{amount \geq 0 \text{ and } amount \leq 50}(loan)$$

- Delete all accounts at branches located in Needham.

$$r_1 \leftarrow \sigma_{branch\text{-}city = \text{"Needham"}}(account \bowtie branch)$$

$$r_2 \leftarrow \Pi_{account\text{-}number,\ branch\text{-}name,\ balance}(r_1)$$

$$r_3 \leftarrow \Pi_{customer\text{-}name,\ account\text{-}number}(r_2 \bowtie depositor)$$

$$account \leftarrow account - r_2$$

$$depositor \leftarrow depositor - r_3$$

2. Insertion

- ***Similar to deletion**, but **uses ∪ operator** instead of – operator.*
- ***In Insertion, tuples are added to the relation***
- To **insert** data into a relation, we either:
  - **specify a tuple to be inserted**, or
  - **write a query whose result is a set of tuples to be inserted**
- An **insertion** is expressed in relational algebra by:

$$r \leftarrow r \cup E$$

where r is a relation and E is a relational algebra expression.

- The ***insertion of a single tuple*** is expressed by letting E be a *constant relation containing one tuple.*

Example: Insertion

r

A	B	C
1	1	10
1	2	10
2	3	10

$$r \leftarrow r \cup \{(2, 4, 20)\}$$

Example: Insertion

r

A	B	C
1	1	10
1	2	10
2	3	10

$r \leftarrow r \cup \{(2, 4, 20)\}$

A	B	C
1	1	10
1	2	10
2	3	10
2	4	20

Insertion Examples Query

- Insert information in the database specifying that **Smith** has **$1200** in account **A-973** at the **Perryridge** branch.

  - $account \leftarrow account \cup \{(A\text{-}973, \text{“Perryridge”}, 1200)\}$
  - $depositor \leftarrow depositor \cup \{(\text{“Smith”}, A\text{-}973)\}$

We may want to insert tuples on the basis of the result of a query.

- Provide as a gift for **all loan customers** in the **Perryridge** branch, a **$200 savings account.** Let the loan number serve as the account number for the new savings account.

  - $r_1 \leftarrow (\sigma_{branch\text{-}name = \text{“Perryridge”}}(borrower \bowtie loan))$
  - $account \leftarrow account \cup \prod_{loan\text{-}number,\ branch\text{-}name,\ 200}(r_1)$
  - $depositor \leftarrow depositor \cup \prod_{customer\text{-}name,\ loan\text{-}number}(r_1)$

3. Updating

- **Updating** is a mechanism ***to change a value in a tuple*** *without changing* ***all values*** *in the tuple*
- ***Use*** the ***generalized projection*** operator to do this task

$$r \leftarrow \prod_{F1, F2, \ldots, Fn} (r)$$

  - Each F_i can be either:
    - the ***attribute*** of r; or
    - an ***expression***, involving only ***constants and the attributes*** of r, which gives the new value for the attribute
  - **Note**: The schema of the expression resulting from the generalized projection expression must match the original schema of r.

Example: Update

$$r \leftarrow \prod_{A,\ 2*B,\ C} (r)$$

r

A	B	C
1	1	10
1	2	10
2	4	20

Example: Update

r

A	B	C
1	1	10
1	2	10
2	4	20

$$r \leftarrow \prod_{A,\ 2*B,\ C} (r)$$

A	B	C
1	2	10
1	4	10
2	8	20

$$r \leftarrow \prod_{A,\ 2*B,\ C} (\sigma_{A=1} (r))$$

A	B	C
1	2	10
1	4	10

Update Examples Query

- Make interest payments by increasing all balances by 5 percent.

$$account \leftarrow \Pi_{account\text{-}number,\ branch\text{-}name,\ balance * 1.05}(account)$$

10.5 / 10 = 1.05

- Pay all accounts with balances over $10,000 a 6 percent interest and pay all others 5 percent

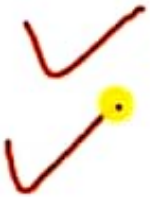
$$account \leftarrow \Pi_{account\text{-}number,\ branch\text{-}name,\ balance * 1.06}(\sigma_{balance > 10000}(account)) \cup \Pi_{account\text{-}number,\ branch\text{-}name,\ balance * 1.05}(\sigma_{balance \leq 10000}(account))$$

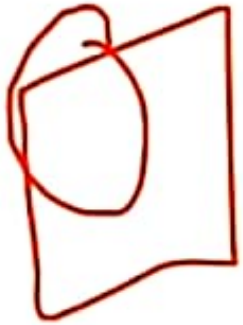
Introduction to Views

- In some cases, it is not desirable for all users to see the ***entire logical model*** i.e. ***all the actual relations stored in the database.***
  - Consider "*a person who needs to know a customer's loan number but has no need to see the loan amount.*" This person should see a relation described, in the relational algebra, by

$$\Pi_{\text{customer-name, loan-number}} (\text{borrower} \bowtie \text{loan})$$

- *Any relation that is made visible to a user as a* ***"virtual relation"*** *is called a* ***VIEW***.
  - It provides **Limited access to DB.**
  - It presents the **Tailored schema.**





What are Views

- **"Views"** are **Virtual Relations (or virtual tables)** , through which a selective portion of the data from one or more relations (or tables) can be seen.
- Views **do not contain data** of their own.
- Views **do not exist physically.**
- **Uses of View**:
  - It **helps** in **query processing** like simplify commands for the user, store complex queries, etc
  - to **restrict access to the database**
  - to **hide data complexity**
- **Database modifications** (like insert, delete and update operations) on views **affects the actual relations** in the database, upon which view is based. (Also in SQL)
- Views are stored in the **data dictionary** in the table called **USER_VIEWS**

View Definition

- A **view** is defined using the **create view** statement:

create view v as <query expression>

where **<query expression>** is any legal relational algebra query expression, and **v** represents the **view name**

Example: *In SQL,*

create view v as
select column-list
from table-name [**where** condition];

- Once a **view** is defined, the **view name** can be used to refer to the **virtual relation** that the view generates.

Example: *In SQL,* **select * from v;**

- ***View definition*** is **not the same as creating a new relation** by evaluating the query expression.
- Rather, **a *view definition* causes the saving of an expression to be substituted into queries using the view.**
  - It means wherever **view v** is used, it is actually replaced by the equivalent **query expression** at run time

View in Relational algebra: Example 1

Creating a view (*loan-customer*) consisting all loan customers and their loan number

create view *loan-customer* as

$$\Pi_{\text{customer-name, loan-number}} (\text{borrower} \bowtie \text{loan})$$

We can find all loan customers and their loan number

loan-customer

- **Note:** So wherever **view *loan-customer*** is used, it is actually replaced by the equivalent **query expression** at run time. This query is evaluated and the entire answer is resulted.

View in Relational algebra: Example 2

- **Creating a view (*all-customer*) consisting of branches and their customers**

  create view *all-customer* as

$$\Pi_{\textit{branch-name, customer-name}} (\textit{depositor} \bowtie \textit{account}) \cup \Pi_{\textit{branch-name, customer-name}} (\textit{borrower} \bowtie \textit{loan})$$

- **We can find all customers of the Perryridge branch**

$$\Pi_{\textit{customer-name}} (\sigma_{\textit{branch-name} = \text{"Perryridge"}} (\textit{all-customer}))$$

- **Note**: So wherever **view *all-customer*** is used, it is actually replaced by the equivalent **query expression** at run time. This query is evaluated and the entire answer is resulted.

View in SQL: Example 1

- **Creating a view *loan-customer*: from multiple tables**

```
create view loan-customer_as
        select customer-name, loan-number
        from borrower natural inner join loan;
```

- **We can find all loan customers and their loan number**

```
select *
from loan-customer
```

View in SQL: Example 2

- **Creating a view *student-view*: from single tables**

```
create view student-view as
    select name, age
    from Students;
```

roll-no	name	age	address
1	Rohan	20	Delhi
2	Sohan	21	Mumbai
3	Mohan	22	Pune

- **To see the student-view**

```
select *
from student-view;
```

name	age
Rohan	20
Sohan	21
Mohan	22

Drop View

- A view can be deleted using the **Drop View** statement.

drop view *viewname*

- Example: **drop view *student-view***

Views: Summary

"Views does not stored **physically.**"

- When we define a **view**, the **database system stores the definition of the view itself, rather than** the **result of evaluation of the relational-algebra expression** that defines the view.
- Wherever a ***view relation*** appears in a query, it is **replaced by the stored query expression.**
- Thus, whenever we evaluate the query, the ***view relation*** gets **recomputed.**
- If the original table used in view changes, it does not affects the view relation because it is evaluated every time

Types of Views:

- ***Read-only View :***
  - Used only to read data.
  - In SQL, it allows only SELECT operations.
- ***Updateable View :***
  - Used to read and update the data.
  - In SQL, it allows SELECT as well as INSERT , UPDATE and DELETE operations.

Materialized Views

For some views, there is a term called ***materialization***. *So some views are materialized.* This depends on the query engine etc., the database engine.

Certain database systems allow view relations to be stored, but they make sure that, if the actual relations used in the view definition change, the view is kept up to date. Such views are **called materialized views.**

- The **process of keeping the view up to date** is called **view maintenance.**
- Applications that use a view frequently in multiple queries benefited from the use of **materialized views** because then query expression is not going to be evaluated at run time
- Of course, the benefits to queries from the materialization of a view must be weighed against the storage costs and the added overhead for updates.

RELATIONAL CALCULUS:

TUPLE RELATIONAL CALCULUS

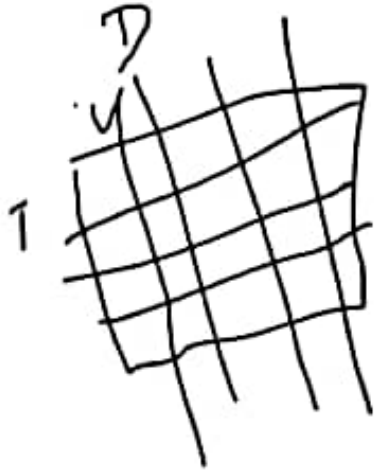
DOMAIN RELATIONAL CALCULUS

Relational Calculus

- **Relational Calculus** is a **non-procedural query language** (or **declarative language**). It uses mathematical **predicate calculus** (or **first-order logic**) instead of algebra.
- **Relational Calculus** tells **what to do** but never explains how to do.
- **Relational Calculus** provides **description about the query** to get the result *where as* ***Relational Algebra*** **gives the method** to get the result.

- When applied to database, it comes in **two flavors**:

1. **Tuple Relational Calculus (TRC):**
   - Proposed by Codd in the year 1972
   - **Works on tuples** (or rows) like SQL
2. **Domain Relational Calculus (DRC):**
   - Proposed by Lacroix & pirotte in the year 1977
   - **Works on domain of attributes** (or Columns) like QBE (Query-By-Example)



Relational Calculus ...

- **Calculus** has *variables, constants, comparison operator, logical connectives* and *quantifiers.*
  - ✓ ***TRC***: **Variables range over *tuples*.**
    - ✓ Like SQL
  - ✓ ***DRC***: **Variables range over *domain elements*.**
    - ✓ Like Query-By-Example (QBE)
- Expressions in the calculus are called ***formulas***.
- **Resulting tuple** is an assignment of constants to variables that make the **formula** evaluate to ***true***.

1. Tuple Relational Calculus (TRC)

- **Tuple relational calculus** is a **non-procedural query language**
- **Tuple relational calculus** is used for **selecting the tuples** in a relation that satisfy the given **condition** (or **predicate**). The **result** of the relation can have one or more tuples.
- A **query** in **TRC** is expressed as:

$$\{t \mid P(t)\}$$

Where t denotes *resulting tuple* and $P(t)$ denotes *predicate (or condition) used to fetch tuple t*

- **Result of Query: It is the set of all tuples t such that predicate P is true for t**
- **Notations** used:
  - t is a *tuple variable*,
  - $t[A]$ denotes the value of tuple t on attribute A
  - $t \in r$ denotes that tuple t is in relation r
  - P is a **formula** similar to that of the ***predicate calculus***

Predicate Calculus Formula

1. **Set of attributes and constants**
2. **Set of comparison operators:** e.g., $<$, $\leq$, $=$, $\neq$, $>$, $\geq$
3. **Set of connectives:** and ($\wedge$), or ($\vee$), not ($\neg$)
4. **Implication ($\Rightarrow$):** $x \Rightarrow y$, if x is true, then y is true $\quad x \Rightarrow y \equiv \neg x \vee y$
5. **Quantifiers:** **Existential Quantifiers ($\exists$) and Universal Quantifier($\forall$).**

- $\exists t \in r\,(Q(t)) \equiv$ "there exists" a tuple in t in relation r such that predicate $Q(t)$ is true
- $\forall t \in r\,(Q(t)) \equiv$ Q is true "for all" tuples t in relation r

Free and Bound variables:

- The use of **quantifiers** $\exists X$ and $\forall X$ in a formula is said to ***bind*** X in the formula.
- A variable that is **not bound** is **free**.
- Let us revisit the definition of a **query**: $\{t \mid P(t)\}$
- ***There is an important restriction***
  - the variable t that appears to the left of `|' must be the ***only free variable*** in the formula $P(t)$.
  - in other words, **all other tuple variables** must be **bound** using a **quantifier**

Example Queries: TRC

- **Find the *loan-number*, *branch-name*, and *amount* for all loans of over $1200.**

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

It selects ***all tuples t*** from *relation* ***loan*** such that the resulting ***loan*** tuples will have ***amount*** greater than $1200

- **Find the *loan number* for each loan of an amount greater than $1200**

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

It selects the ***set of tuples t*** such that there exists a ***tuple s*** in relation ***loan*** for which the values of ***t*** & ***s*** for the *loan-number* attribute are equal and the value of *s* for the *amount* attribute is greater than $1200

branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Example Queries: TRC

- **Find the *loan-number*, *branch-name*, and *amount* for all loans of over $1200.**

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

(Selection)

It selects ***all tuples t*** from *relation* ***loan*** such that the resulting ***loan*** tuples will have ***amount*** greater than $1200

- **Find the *loan number* for each loan of an amount greater than $1200**

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

(Projection)

It selects the ***set of tuples t*** such that there exists a ***tuple s*** in relation ***loan*** for which the values of *t* & *s* for the *loan-number* attribute are equal and the value of *s* for the *amount* attribute is greater than $1200

branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Example Queries: TRC...

branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

- **Find the *names of all customers* having a *loan* at the *Perryridge* branch**

$$\{ t \mid \exists s \in borrower \ (t[customer\text{-}name] = s[customer\text{-}name] \wedge \exists u \in loan \ (u[branch\text{-}name] = \text{“Perryridge”} \wedge u[loan\text{-}number] = s[loan\text{-}number])) \}$$

(Join)

- **Find the *names of all customers* having a loan, an account, or both at the bank**

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

(Union)

- **Find the names of all customers who have a loan and an account at the bank**

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

(Intersection)

2. Domain Relational Calculus (DRC

- **Domain Relational Calculus** is a **non-procedural query language.**
- In **Domain Relational Calculus** the records are filtered based on the **domains.**
- **DRC** uses **list of attributes** to be selected from relation based on the **condition (or predicate).**

2. Domain Relational Calculus (DRC)

- **Domain Relational Calculus** is a **non-procedural query language.**
- In **Domain Relational Calculus** the records are filtered based on the **domains.**
- **DRC** uses **list of attributes** to be selected from relation based on the **condition (or predicate).**
- **DRC** is same as **TRC** but differs by **selecting the attributes** rather than **selecting whole tuples**
- In **DRC**, each **query** is an expression of the form:

$$\{ < a_1, a_2, \ldots, a_n > \mid P(a_1, a_2, \ldots, a_n) \}$$

$a_1, a_2, \ldots, a_n$ represent *domain variables*

P represents a ***predicate*** similar to that of the *predicate calculus*

- **Result of Query:** It is the set of all tuples $a_1, a_2, \ldots, a_n$ such that predicate P is true for $a_1, a_2, \ldots, a_n$ tuples

Predicate Calculus Formula

- **Notations used:**
  - $< a_1, a_2, \ldots, a_n > \in r$, where r is relation on n attributes and $a_1, a_2, \ldots, a_n$ are domain variables or domain constants
  - P is a **formula** similar to that of the ***predicate calculus***
- **Formula:**
  1. **Set of domain variables and constants**
  2. **Set of comparison operators:** e.g., $<, \leq, =, \neq, >, \geq$
  3. **Set of connectives:** and ($\wedge$), or (v), not ($\neg$)
  4. **Implication ($\Rightarrow$):** $x \Rightarrow y$, if x is true, then y is true $x \Rightarrow y \equiv \neg x \vee y$
  5. **Quantifiers:** **Existential Quantifiers ($\exists$) and Universal Quantifier($\forall$).**

     $\exists x\,(P(x))$ *and* $\forall x\,(P(x))$

     x is Free domain variable

Example Queries: DRC

- Find the *loan-number*, *branch-name*, and *amount* for loans of over $1200:

$$\{<l, b, a> \mid <l, b, a> \in loan \wedge a > 1200\}$$

(Selection)

- Find the *loan number* for each loan of an amount greater than $1200:

$$\{<l> \mid \exists\, b, a\ (<l, b, a> \in loan \wedge a > 1200)\}$$

(Selection then Projection)

branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Example Queries: DRC ...

- Find the *names of all customers* who have a loan of over $1200:

$$\{< c > \mid \exists\, l, b, a\, (< c, l > \in borrower \wedge (< l, b, a > \in loan \wedge a > 1200))\}$$

- Find the *names of all customers* having a *loan* at the *Perryridge* branch and find the *loan amount*:

$$\{< c, a > \mid \exists\, l\, (< c, l > \in borrower \wedge \exists\, b\, (< l, b, a > \in loan \wedge b = \text{“Perryridge”}))\}$$

branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)