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 rows (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

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 3: Select students whose age is greater than 17

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

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

$$\sigma_{age > 17 \wedge address = \text{“Delhi”}} (Student)$$

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**

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

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

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

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 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

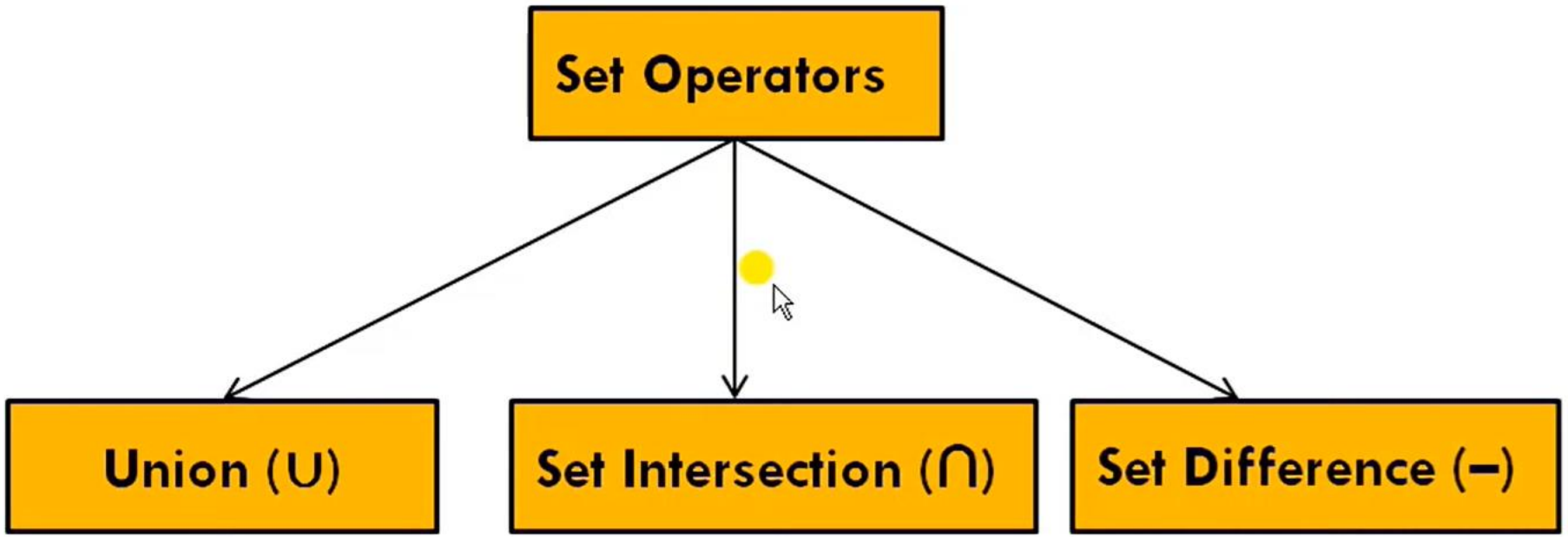
$$\Pi_{roll\_no,\ name}\ (\sigma_{age\ >\ 17}\ (Student))$$

roll_no	name
1	A
4	D
5	E
6	F

Wrong

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

Set Operators in Relational Algebra



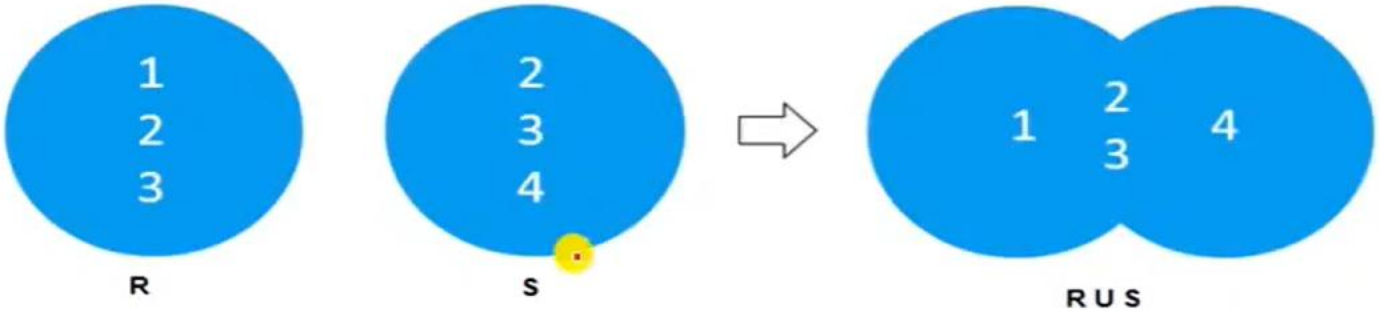
Set Operators

- **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

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) U Π_{Name} (Employee)

Name
A
B
C
D
G
H

Example

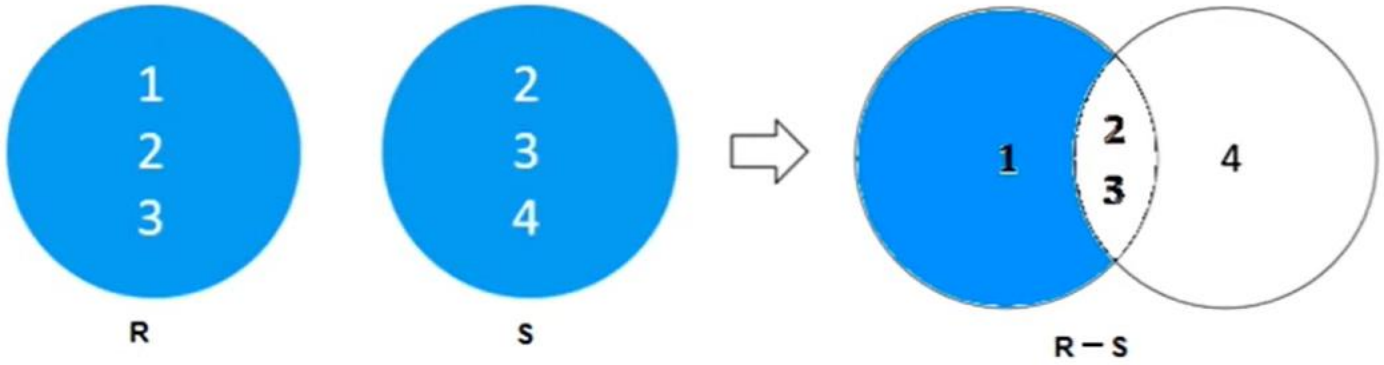
- Find the names of the authors who have written a book and an article both.

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

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

$$\Pi_{\text{Name}}(\text{Student}) - \Pi_{\text{Name}}(\text{Employee})$$

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

R1 x R2
Resulting relation must have

2+3 = 5 attributes
2 x 4= 6 tuples

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

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

Example

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

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

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**

- **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 product between same table then* ***one of the table should be renamed with another name***
- ***R x R***

  (Ambiguity will be there)

R

A	B
α	1
β	2

- ***R x ρ_s (R)***

  (Rename R to S)

RxR

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 x ρ_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*.

Examples:

- **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)

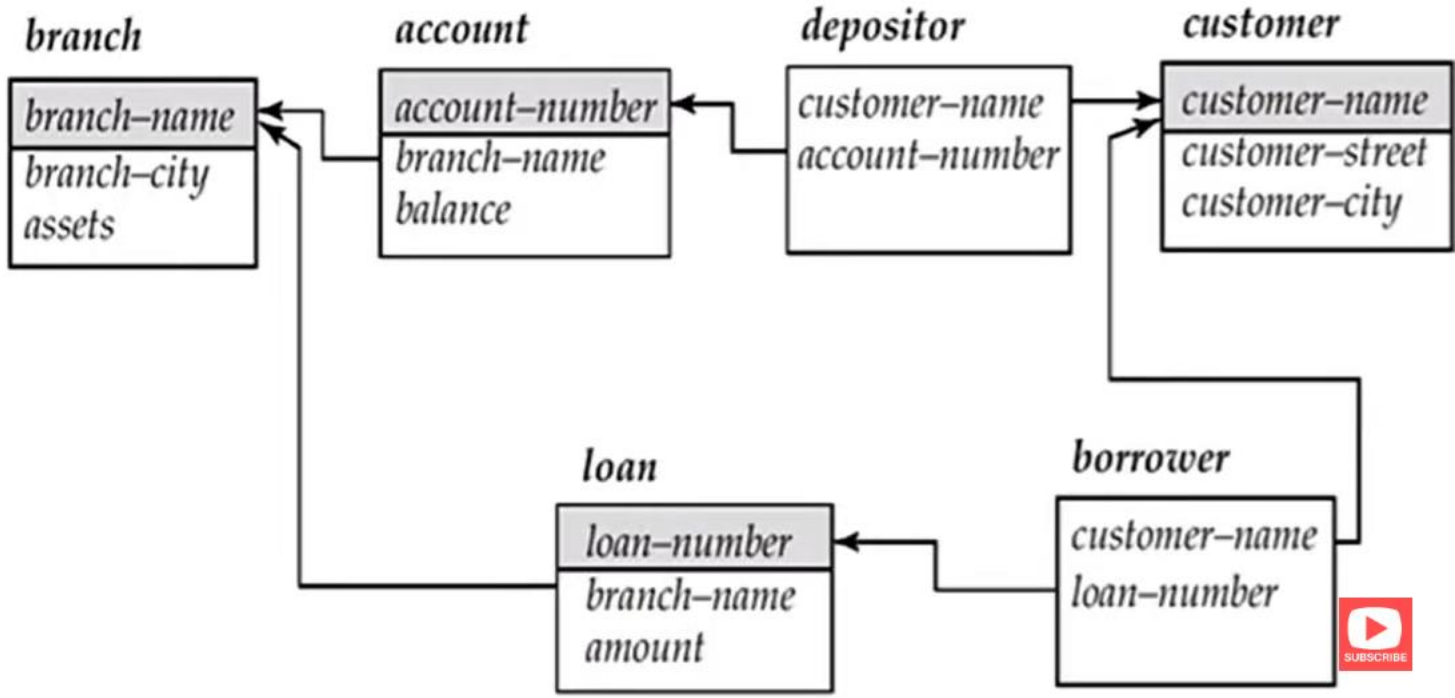
customer (customer-name, customer-street, customer-city)

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 (<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(<u>customer-name</u>, <u>loan-number</u>)

Selection (σ): Example 2

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

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

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 (Π): 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 (Π): Example 3

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

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

customer-name
Hayes
Jones

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

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.

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

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)

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

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

customer-name
Adams
Hayes

Cartesian Product (x): Example 1

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

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

customer-name
Adams
Hayes

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 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
  - *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.**

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**: ⋈
- $A \bowtie_C B = \sigma_C (A \times B)$

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

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**: ⋈
- $A \bowtie_c B = \sigma_c (A \times B)$

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:*** $A \bowtie_{A.a1 = B.b1 \wedge \ldots\ldots\ldots\ldots\ldots \wedge A.an = B.bn} B$

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* **Subjects**

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

* imp

- **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**: $A \bowtie B$
- *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***.

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	δ

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

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** 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 ⋈ 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

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-city)
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 $(customer \bowtie account \bowtie depositor)$

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 tuples 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

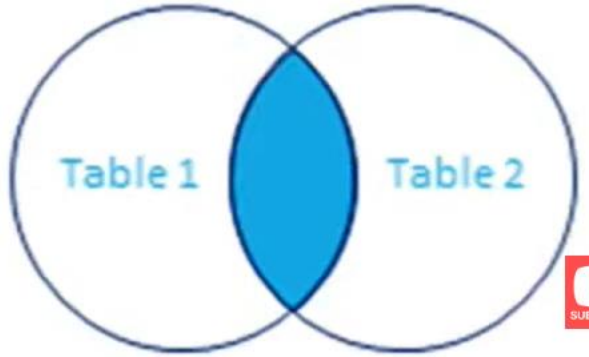
HOD

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)

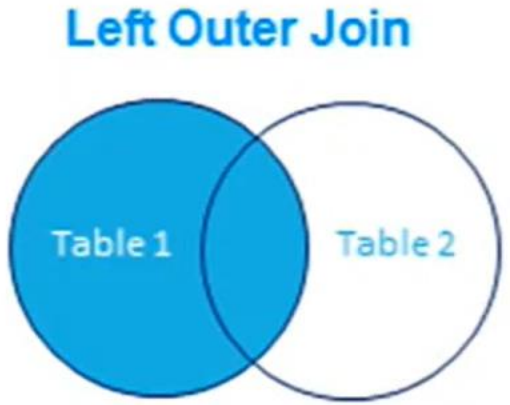
- There are **three kinds of outer joins:**

  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

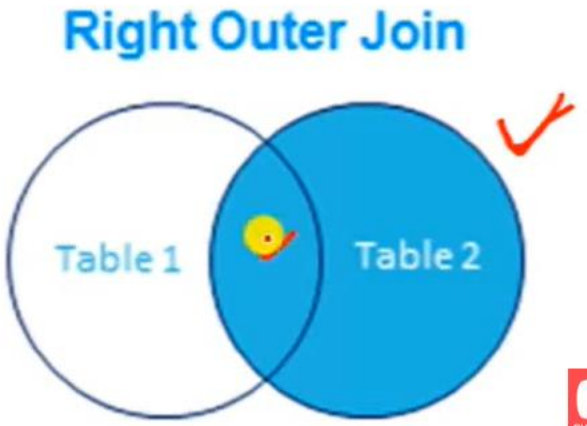
CID	Name
100	Rohan
102	Sara
104	Jiya

Courses ⟕ 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

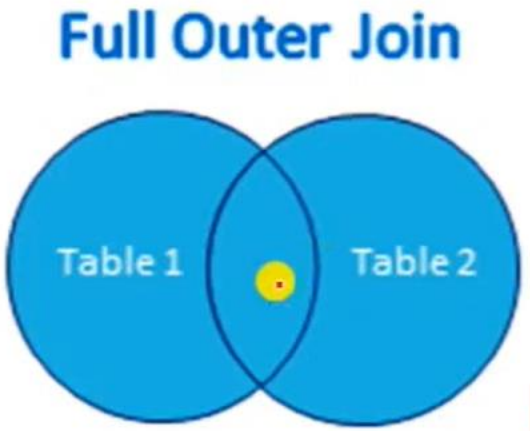
CID	Name
100	Rohan
102	Sara
104	Jiya

Courses ⟖ HOD

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**



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

Outer Join – Example

- Relation **Loan**

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

- Relation **borrower**

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

Inner Join (Natural Join)– Example

loan

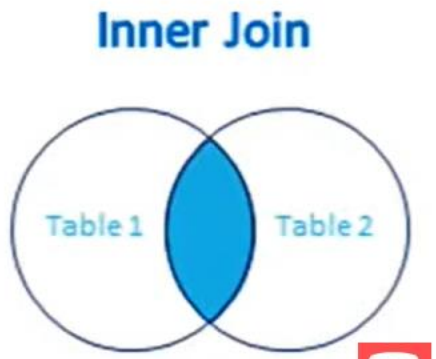
loan-number	branch-name	amount
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

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



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*

Right Outer Join – Example

loan

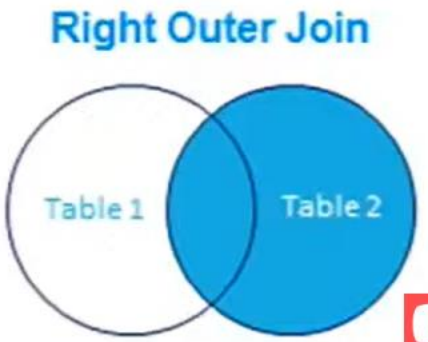
loan-number	branch-name	amount
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

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

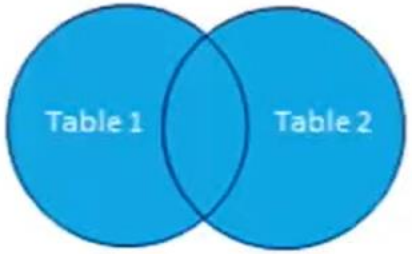
borrower

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 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 to every B's tuple**.

Example 1:

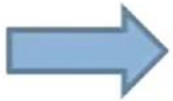
A

x	y
a	1
b	2
a	2
d	4

÷

B

y
1
2



A÷B

x
a

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 **a, b, c, d** **b, d**

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

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

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

$\Pi_{sno}(A)$

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

sno
S1
S2
S3
S4

−

sno
S2
S3

A/B

sno
S1
S4

BHANU KUTTAN CSE CLASSES

SUBSCRIBE

Example 3:

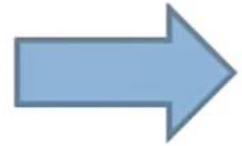
A

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

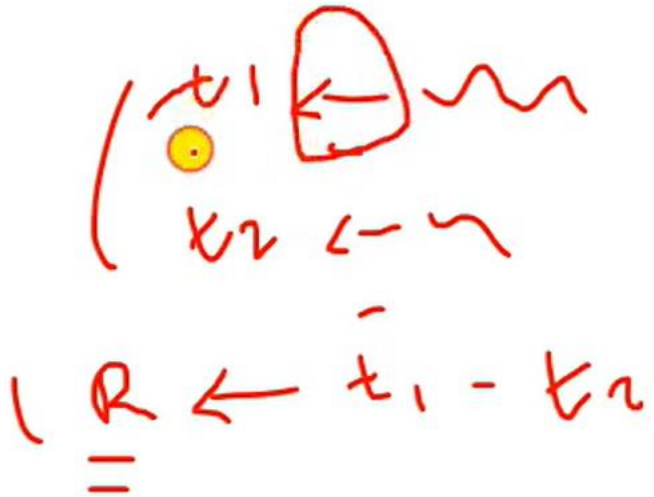


A÷B

A	B	C
α	a	γ
γ	a	γ

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.**



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*.

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**:

$$G1, G2, \ldots, Gn \; g_{F1(A1), F2(A2), \ldots, Fn(An)} (E)$$

  - ***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 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

Suggested: Relational Model & Relational Algebra

- *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***

SQL (Structured Query Language)

- **SQL** is a **Domain-Specific Language.**
- **SQL** is **Declarative language i.e.** a **non-procedural language.**
  - **SQL** allows to declare **what we want to do**, but not how to do.
    - In **SQL,** we can specify through queries that **what data we want** but does not specify how to get those data.
    - Whereas, **C** is procedural language as it uses functions, procedures, loops, etc. **to specify what to do and how to do it.**

Question: How the query is actually performed in SQL?
Answer: The database/SQL engine is very powerful. When we write an SQL query, it first parses it; then it figures out its internal algorithms and it tries to select an algorithm which will be the best for that particular query. So, then it applies that algorithm, finds the answer and returns it.

- **SQL** is based on ***Relational Algebra*** and ***Tuple Relational Calculus***.

What Can SQL do?

- SQL can **execute queries** against a database
- SQL can **retrieve data** from a database
- SQL can **insert records** in a database
- SQL can **update records** in a database
- SQL can **delete records** from a database
- SQL can **create** new **databases**
- SQL can **create** new **tables** in a **database**
- SQL can **create** stored **procedures** in a **database**
- SQL can **create views** in a **database**
- SQL can **set permissions** on tables, procedures, and views

Types of SQL Commands

- **Data Definition Language (DDL)** commands are used to define the structure and schema of the database.
  - **CREATE** - to create new table or database
  - **ALTER** - to alter the structure of table
  - **DROP** - to delete a table from database
  - **TRUNCATE** - to delete all records from table
  - **RENAME** - to rename a table
- **Data Manipulation Language (DML)** commands are used for accessing and manipulating the data stored in the database.
  - **SELECT** - to retrieve data from the database
  - **INSERT** - to insert a new row into a table
  - **UPDATE** - to update existing row in a table
  - **DELETE** - to delete a row from a table

Programmers call these DML operations "CRUD".
CRUD** stands for:* ***Create, Read, Update, Delete
Performed by: ***INSERT, SELECT, UPDATE, DELETE***

Types of SQL Commands...

- **Data Control Language (DCL)** are the commands to control the access of the data stored in the database
  - **GRANT** - grant permission to user for database access
  - **REVOKE** - take back granted permission from user
- **Transaction Control Language (TCL)** commands are used to manage the changes made by the DML statement.
  - **COMMIT** - to permanently save the transaction
  - **ROLLBACK** - to undo transaction
  - **SAVEPOINT** - to temporarily save a transaction so that you can rollback to that point whenever necessary
- **Note:** TCL Commands are used for only DML commands while DDL and DCL commands are auto-commited

SQL Rules

- **SQL** is **NOT case sensitive.**
  - For example: select is the same as SELECT
- But ***names*** of **databases, tables and columns** are **case sensitive.**
  - For example: In given SQL query, ACCOUNT is different from account

```
SELECT * FROM ACCOUNT;

SELECT * FROM account;
```

- **SQL statements** can use **multiple lines**
  - End each SQL statement with a **semi-colon** ;

SQL: Data Definition Language (DDL)

- **Data Definition Language (DDL)** is used for creating and modifying the database objects such as tables, indices, views and users.
- **DDL Commands** are used to define the structure and schema of the database
- All the command of DDL are *auto-committed* that means it permanently save all the changes in the database.
- **Data Definition Language (DDL) Commands:**
  - **CREATE** - to create new table or database
  - **ALTER** - to alter (or modify) the structure of table
  - **DROP** - to delete a table from database
  - **TRUNCATE** - to delete all records from table
  - **RENAME** - to rename a table

1. CREATE

- **CREATE** statement is used to create database schema and to define the type and structure of the data to be stored in the database.
- **CREATE** statement can be used for
  - **Creating a Database,**
  - **Creating a Table**, etc.

I. Creating a DATABASE:

- **CREATE DATABASE** statement is used **to create a database in RDBMS.**

Syntax:

```
CREATE DATABASE database_name;
```

Example:

```
CREATE DATABASE my_db;
```

Note:

- **SHOW statement:** To see existing **databases** and **tables**

```
SHOW databases;
SHOW tables;
```

- **USE statement:** To use or select any existing database

```
Syntax      USE database_name;
Example:    USE my_db;
```

II. Creating a TABLE:

- **CREATE TABLE** statement is used **to create a new table in a database.**
  - It specifies ***column names*** of the table, its ***data types*** (e.g. varchar, integer, date, etc.) and can also specify ***integrity constraints*** (e.g. Primary key, foreign key, not null, unique).

Syntax:

```
CREATE TABLE  table_name
(
        column_name1    datatype,
        column_name2    datatype,
             …                …
        column_name n   datatype,
        (Integrity Constraints)
);
```

Example: Create table

Example 1:

```
CREATE TABLE Emp(
    Emp_ID int,
    Name varchar(20),
    Age int,
    Address varchar(100),
    Salary numeric(10,2)
);
```

Emp_ID	Name	Age	Address	Salary

Example 2:

```
CREATE TABLE branch(
    branch-name varchar(15),
    branch-city varchar(30),
    assets integer,
    primary key (branch-name),
    check(assets>0)
);
```

branch-name	branch-city	assets

Note:

DESC (or DESCRIBE) statement: To describe the details of the table structure

Syntax: **DESC *table_name*;**

Example: **DESC *Emp*;**

SQL DATA TYPES

- **Each column** in a database **table** is required to have a ***name*** and a ***data type***.
- The **data type** is a guideline for SQL to understand *what type of data is expected inside of each column*, and it also *identifies how SQL will interact with the stored data.*

- **Note**: Data types might have different names in different database. And even if the name is the same, the size and other details may be different! Always check the documentation!

Data type	Description
CHAR(size)	A FIXED length string (can contain letters, numbers, and special characters), with user-specified length *size*.
VARCHAR(size)	A VARIABLE length string (can contain letters, numbers, and special characters), with user-specified length *size*. Full From Character Varying
INT / INTEGER	A medium Integer (Signed range is from -2,147,483,648 to 2,147,483,647. Unsigned range is from 0 to 4294967295). 4 bytes
SMALLINT	A small integer (Signed range is from -32,768 to 32,767. Unsigned range is from 0 to 65,535.) 2 bytes
BIGINT	A large integer (Signed range is from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,808. Unsigned range is from 0 to 18446744073709551615). 8 bytes
NUMERIC(size,d)/ DECIMAL(size,d)/ FLOAT(size,d)	A small floating point number. Its *size* parameter specifies the total number of digits. The number of digits after the decimal point is specified by *d* parameter. For NUMRIC(3,1), 46.2 can be stored but not 466.0 or 0.32
REAL, DOUBLE PRECISION	Floating point and double-precision floating point numbers, with machine-dependent precision.
FLOAT(n)	A floating point number, with user-specified precision of at least *n* digits.
DOUBLE(size, d)	A large floating point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter

Data type	Description
DATE	A date. Format: YYYY-MM-DD. The supported range is from '1000-01-01' to '9999-12-31'
TIME	A time. Format: hh:mm:ss. The supported range is from '-838:59:59' to '838:59:59'
DATETIME / TIMESTAMP	A date and time combination. Format: YYYY-MM-DD hh:mm:ss.
YEAR	A year in four-digit format. Values allowed in four-digit format: 1901 to 2155, and 0000.

Note: There are many other data types.....

SQL Constraints

- **SQL constraints are used to specify rules for the data in a table.**
- **Constraints** are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the table. If there is any violation between the constraint and the data action, the action is aborted.
- **Constraints** can be **column level** or **table level**.
  - ***Column level*** constraints apply to a column
  - ***Table level*** constraints apply to the whole table.
- **Constraints** can be specified:
  - when a table is created with the **CREATE TABLE** statement, or
  - we can use the **ALTER TABLE** statement to create constraints even after the table is created.

SQL Constraints:

- ***NOT NULL***: Ensures that a column cannot have NULL value.
- ***DEFAULT***: Provides a default value for a column when none is specified.
- ***UNIQUE***: Ensures that all values in a column are different.
- ***PRIMARY KEY***: A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
- ***FOREIGN KEY***: Uniquely identifies a row/record in another table.
- ***CHECK***: Ensures that all the values in a column satisfies certain conditions
- ***INDEX***: Used to create and retrieve data from the database very quickly

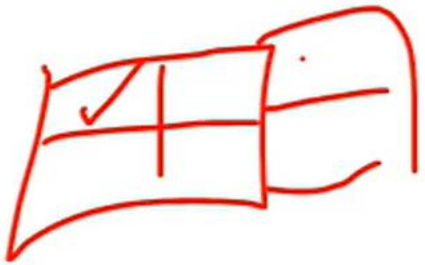
Example:

```
CREATE TABLE  Emp
(
        Emp_ID int NOT NULL,
        Name varchar(20) NOT NULL,
        Age int NOT NULL,
        Address varchar(100) DEFAULT 'India',
        Salary numeric(10,2) NOT NULL,
        DeptID int,

        PRIMARY KEY(Emp_ID),
        FORIEGN KEY (Dept_ID) REFERENCES Dept(Dept_ID),
        CHECK(Age>=18)
);
```

2. ALTER

- **ALTER TABLE** statement is **used to modify the structure of the existing table**
- It is used to add, delete, modify or rename columns in an existing table.
- It is also used to add and drop various constraints on an existing table.



1. **ALTER TABLE – ADD COLUMN**
   - For adding new columns in a table
2. **ALTER TABLE - DROP COLUMN**
   - For removing existing columns in table
3. **ALTER TABLE - MODIFY COLUMN**
   - To modify existing columns in a table
4. **ALTER TABLE – RENAME COLUMN**
   - To rename an existing column in a table

1. ALTER TABLE – ADD COLUMN

- For adding new columns in a table
- *Syntax:*

```
ALTER TABLE table_name
ADD column_name(s)  datatype(s);
```

Example:

```
ALTER TABLE Emp
ADD Email varchar(100);
```

2. ALTER TABLE - DROP COLUMN

- For removing existing columns in a table
- *Syntax*:

```
ALTER TABLE table_name
DROP column_name(s);
```

Example:

```
ALTER TABLE Emp
DROP Email;
```

3. ALTER TABLE - MODIFY COLUMN

- **To modify existing columns in a table**
  - To change **data type** of any column or to modify its **size.**
- ***Syntax*:**

```
ALTER TABLE table_name
MODIFY column_name  datatype;
```

Example:

```
ALTER TABLE Emp
MODIFY Name varchar(100);
```

4. ALTER TABLE – RENAME COLUMN

- To rename an existing column in a table
- *Syntax*:

```
ALTER TABLE table_name
RENAME old_column_name To new_column_name;
```

Example:

```
ALTER TABLE Emp
MODIFY Address To Location;
```

3. DROP

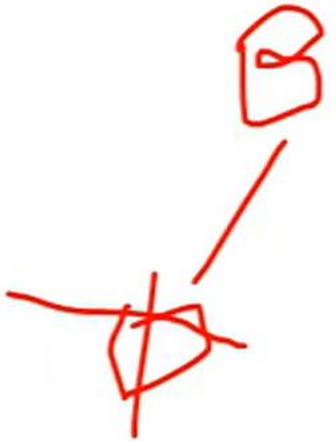
- **DROP TABLE** statement **completely removes a table from the database.**
  - This command will destroy the table structure and the data stored in it.

Syntax:

```
DROP TABLE table_name;
```

Example:

```
DROP TABLE Emp;
```



DROP command is also be used on Databases.

DROP DATABASE statement is used to delete the complete database.

Syntax: `DROP DATABASE database_name;`

Example: `DROP DATABASE my_db;`

4. TRUNCATE

- **TRUNCATE TABLE** statement is used **to remove all rows (complete data) from a table.** It is similar to the DELETE statement with no WHERE clause.
- **TRUNCATE TABLE Vs DROP TABLE**
  - DROP TABLE command can also be used to delete complete table but it deletes table structure too. TRUNCATE TABLE doesn't delete the structure of the table.
- **Syntax:**

```
TRUNCATE TABLE table_name;
```

Emp

Emp_ID	Name	Age	Location	Salary
1	Rahul	32	Delhi	2000
2	Kamal	25	Pune	1500

Example:

```
TRUNCATE TABLE Emp;
```

Emp_ID	Name	Age	Location	Salary

SQL: Data Manipulation Language (DML)

- **Data Manipulation Language (DML)** commands are used **for accessing and manipulating the data stored in the database.**
- The DML commands are *not auto-committed* that means it can't permanently save all the changes in the database. They can be rollback.
- **Data Manipulation Language (DML)**
  - **SELECT** - to retrieve data from the database ⟸ **Data Query language (DQL)**
  - **INSERT** - to insert a new row into a table
  - **UPDATE** - to update existing row in a table
  - **DELETE** - to delete a row from a table

  } **Modification of Database**

R C U D

Programmers call these DML operations "CRUD".

***CRUD** stands for:* **Create, Read, Update, Delete**

***Performed** by:* **INSERT, SELECT, UPDATE, DELETE**

1. SELECT (mostly used SQL command/query)

- **SELECT** statement is **used to select a set of data from a database table.** Or Simply **SELECT** statement is **used to retrieve data from a database table.**
  - It **returns** data in the form of a **result table.** These result tables are called **result-sets.**
- **SELECT** is also called **DQL** because it is used to **query** information from a database table
- **SELECT** statement specifies **column names**, and **FROM** specifies **table name**
- **SELECT** command is used with different **Conditions** and **CLAUSES**.

Basic Syntax:

To retrieve selected fields from the table:

```
SELECT column1, column2, ... columnN
FROM table_name(s);
```

To retrieve all the fields from the table:

```
SELECT *
FROM table_name(s);
```

Example Table: Emp

Columns/Fields/Attributes

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

Records/ Rows/ Tuples

Example: SELECT

Example 1:

To retrieve columns *Emp_ID, Name* from *Emp* table

```
SELECT Emp_ID, Name
FROM Emp;
```

Emp_ID	Name
1	Rahul
2	Kamal
3	Karan
4	Chirag
5	Harsh
6	Kajal
7	Mahi

Example 2:

To retrieve all the fields from the Emp table

```
SELECT *
FROM Emp;
```

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

WHERE CLAUSE

- **WHERE Clause** is used **to select a particular record based on a condition**. It is used to filter records.
- **WHERE Clause is used to specify a condition while fetching the data from a single table or by joining with multiple conditions**
- The **WHERE** clause is not only used in the **SELECT** statement, but it is also used in the **UPDATE**, **DELETE** statement.
- ***Syntax: "SELECT with WHERE Clause"***

```
SELECT column1, column2, ... columnN
FROM table_name(s)
WHERE condition;
```

Example:

To retrieve *Emp_ID, Name* from *Emp* table whose *Salary* is greater than 2000

```
SELECT Emp_ID, Name
FROM Emp
WHERE Salary>2000;
```

Emp_ID	Name
4	Chirag
5	Harsh
6	Kajal
7	Mahi

Basic Query Structure

Suggested: Relational Model & Relational Algebra

- **SQL** is based on **set and relational operations** with certain modifications and enhancements
- A **typical SQL query** has the form:

  select $A_1, A_2, \ldots, A_n$
  from $r_1, r_2, \ldots, r_m$
  where P

  Select command is equivalent to Projection (Π)

  - A_is represent ***attributes***
  - r_is represent ***relations***
  - P is a ***predicate (or condition)***.
- This **SQL query** is **equivalent** to the **relational algebra expression**.

$$\Pi_{A1, A2, \ldots, An}(\sigma_P(r_1 \times r_2 \times \ldots \times r_m))$$

- The **result** of an **SQL query** is a **relation (or table)**.

Optional Clauses in SELECT statement

Suggested: Relational Model & Relational Algebra

- [**WHERE Clause**] : It specifies which rows to retrieve by specifying **conditions.**
- [**GROUP BY Clause**] : **Groups rows** that share a **property** so that the **aggregate function** can be applied to each group.
- [**HAVING Clause**] : It **selects** among the **groups** defined by the **GROUP BY** clause by specifying **conditions. .**
- [**ORDER BY**] : It specifies an order in which to return the rows.

- [**DISTINCT Clause**]: It is used to remove duplicates from results set of a **SELECT** statement. (**SELECT DISTINCT**)

1. SQL Arithmetic Operators

Operator	Description
+	Add
–	Subtract
*	Multiply
/	Divide
%	Modulo

Example:

To retrieve *Emp_ID, Name, Salary plus 3000 from Emp* table

```
SELECT Emp_ID, Name, Salary + 3000
FROM Emp;
```

Emp_ID	Name	Salary
1	Rahul	5000.00
2	Kamal	4500.000
3	Karan	5000.00
4	Chirag	9500.00
5	Harsh	11500.00
6	Kajal	7500.00
7	Mahi	13000.00

2. SQL Comparison Operators

Operator	Description
=	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<> or !=	Not equal to

Example:

To retrieve *Emp_ID, Name* from *Emp* table whose *Salary* is greater than 2000

```
SELECT Emp_ID, Name, Salary
FROM Emp
WHERE Salary >2000;
```

Emp_ID	Name	Salary
4	Chirag	6500.00
5	Harsh	8500.00
6	Kajal	4500.00
7	Mahi	10000.00

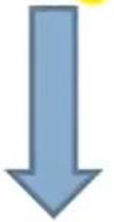
3. SQL Logical Operators

Operator	Description
AND	It displays a record if all the conditions separated by AND are TRUE.
OR	It displays a record if any of the conditions separated by OR is TRUE
NOT	• It displays a record if the condition(s) is NOT TRUE. • It reverses the meaning of any operator with which it is used. This is a negate operator. Eg: NOT EXISTS, NOT BETWEEN, NOT IN, IS NOT NULL, etc.

Example:

To retrieve all records from *Emp* table whose *salary* is greater than 2000 and age is less than 25

```
SELECT *
FROM Emp
WHERE Salary>2000 AND Age<25;
```



Emp_ID	Name	Age	Address	Salary
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

4. SQL Special Operators

Operator	Description
ALL	It compares a value to all values in another value set (in subqueries)
ANY	It compares the values in the list according to the condition (in subqueries).
BETWEEN	It is used to search for values that are within a set of values (given minimum and maximum values).
EXISTS	It is used to search for the presence of a row in a specified table (in subqueries).
IN	It compares a value to that specified list value (and in subquries).
LIKE	It compares a value to similar values using wildcard operator (% , _).
IS NULL	It checks for missing data or NULL values

Example:

To retrieve all records from *Emp* table with salary between 2000 and 5000

```
SELECT *
FROM Emp
WHERE Salary BETWEEN 2000 AND 5000;
```

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
3	Karan	23	Pune	2000.00
6	Kajal	22	Bilaspur	4500.00

2. INSERT

- The **INSERT INTO** statement is **used to insert new records in a table.**
- It is possible to write the **INSERT INTO** statement in **two ways.**

Syntax 1: **Specify both** the **column names** and the **values** to be inserted

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

Syntax 2: **Specify only values** to be inserted. But needed to **remember** the **column order**

```
INSERT INTO table_name
VALUES (value1, value2, value3,...valueN);
```

Example: INSERT INTO

```
INSERT INTO Emp (Emp_ID, Name, Age, Address, Salary) VALUES (1, 'Rahul', 32, 'Delhi',2000);
INSERT INTO Emp (Emp_ID, Name, Age, Address, Salary) VALUES (2, 'Kamal', 25, 'Pune',1500);
INSERT INTO Emp (Emp_ID, Name, Age, Address, Salary) VALUES (3, 'Karan', 23, 'Pune', 2000);
```

1st way

```
INSERT INTO Emp VALUES (4, 'Chirag', 25, 'Mumbai', 6500);
INSERT INTO Emp VALUES (5, 'Harsh', 32, 'Mumbai', 8500);
INSERT INTO Emp VALUES (6, 'Kajal', 22, 'Bilaspur', 4500);
INSERT INTO Emp VALUES (7, 'Mahi', 24, 'Patna', 10000);
```

2nd way

To Check:

```
SELECT *
FROM Emp;
```

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

3. UPDATE

- The **UPDATE** statement is **used to modify the existing records in a table.**
- **Note**: ***Always use the WHERE clause with the UPDATE statement to update the selected rows, otherwise all the rows would be affected.***

Syntax:

```
UPDATE table_name
SET column1 = value1, column2 = value2, ..... columnN = valueN
WHERE condition;
```

Example: UPDATE

Example:

To update the address of Emp_ID: 3 to *Chennai*

```
UPDATE Emp
SET Address='Chennai'
WHERE Emp_ID=3;
```

To Check:

```
SELECT *
FROM Emp;
```

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Chennai	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

4. DELETE

- The **DELETE** statement is **used to delete existing records in a table.**
- **Note**: ***Always use the WHERE clause with a DELETE statement to delete the selected rows, otherwise all the records would be deleted.***

Syntax:

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

Example: DELETE

Example:

To delete the employee records of ***Pune*** location

```
DELETE FROM Emp
WHERE Address='Pune';
```

To Check:

```
SELECT *
FROM Emp;
```

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

SELECT DISTINCT: "DISTINCT Clause"

- The **SELECT DISTINCT** statement is **used to return only distinct (different) values.**

  Or **simply it is used to return only unique values.**

- **DISTINCT Clause** is used **to remove duplicates** from results set of a **SELECT** statement.
  - Inside a table, a **column** often **contains many duplicate values**; and sometimes you only want to list the different (distinct) values.

Syntax:

```
SELECT DISTINCT column1, column2, ... columnN
FROM table_name;
```

SELECT DISTINCT....

Note:

- When only **one column** (expression) is **provided** in the **DISTINCT clause**, the query will return the **unique values** for that **column.**
- When **more than one column** (expression) is **provided** in the **DISTINCT clause**, the query will retrieve **unique combinations** for the **columns** listed.
- In SQL, the **DISTINCT clause** doesn't **ignore NULL values**. So when using the **DISTINCT** clause in your SQL statement, your result set will include **NULL** as a distinct value.

SQL: AND, OR, NOT OPERATOR

- The **WHERE** clause can be combined with **AND**, **OR**, and **NOT** operators.
  - To make more precise conditions for fetching data from database by combining more than one condition together.
- **The AND and OR operators are used to filter records based on more than one condition**
  - The **AND** operator displays a **record** if **all the conditions** separated by **AND** are **TRUE.**
  - The **OR** operator displays a **record** if **any of the conditions** separated by **OR** is **TRUE.**
- **The NOT operator displays a record if the condition(s) is NOT TRUE**
  - **NOT** reverses the meaning of any operator with which it is used. This is a **negate operator.**
  - **Eg: NOT IN, NOT BETWEEN, NOT EXISTS, IS NOT NULL,** etc.

Example Table: Emp

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

SQL: AND OPERATOR

- ***AND operator (or AND condition)*** is used to combine multiple conditions with **WHERE** clause, in **SELECT**, **UPDATE** or **DELETE** statements.
- All conditions must be satisfied for a record to be selected.
- ***Syntax:***

```
SELECT column1, column2, ... columnN
FROM table_name
WHERE condition1 AND condition2.....AND conditionN;
```

Example:

Selects all records from *Emp* table whose *salary* is greater than 2000 and age is less than 25

```
SELECT *
FROM Emp
WHERE Salary>2000 AND Age<25;
```

Emp_ID	Name	Age	Address	Salary
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

SQL: OR OPERATOR

Suggested: Relational Model & Relational Algebra

- **OR operator (or OR condition)** is used to combine multiple conditions with **WHERE** clause, in **SELECT, UPDATE** or **DELETE** statements.
- Any one of the conditions must be satisfied for a record to be selected.
- ***Syntax*:**

```
SELECT column1, column2, ... columnN
FROM table_name
WHERE condition1 OR condition2.....OR conditionN;
```

Example:

Selects all records from *Emp* table whose *salary* is greater than 2000 or whose age is less than 25

```
SELECT *
FROM Emp
WHERE Salary>2000 OR Age<25;
```

Emp_ID	Name	Age	Address	Salary
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

SQL: NOT OPERATOR

- **NOT operator (or NOT condition)** is used to negate the conditions with the **WHERE** clause, in **SELECT, UPDATE** or **DELETE** statements.
- It displays a record if the condition(s) is NOT TRUE
- **NOT** reverses the meaning of any operator with which it is used. This is a **negate operator.**
  - **Eg: NOT IN, NOT BETWEEN, NOT EXISTS, IS NOT NULL, etc.**
- ***Syntax*:**

```
SELECT column1, column2, ... columnN
FROM table_name
WHERE NOT condition;
```

Example:

Selects all records from where address is *not* *Mumbai*

```
SELECT *
FROM Emp
WHERE NOT Address='Mumbai';
```

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

Example: Combining AND and OR Conditions

- The SQL **AND condition** and **OR condition** can be combined to test for multiple conditions with the **WHERE** clause, in **SELECT, UPDATE or DELETE** statements.
- When combining these conditions, it is **important** to **use parentheses ()**.
  - The **parentheses determine the order** that the AND and OR conditions are evaluated.
- **Precedence order: NOT, AND, OR**

Example:

Selects all records from *Emp* table whose *salary* is greater than 2000 and age is less than 25 or whose Emp ID is 2.

```
SELECT *
FROM Emp
WHERE (Salary>2000 AND Age<25) OR (Emp_ID=2);
```

Emp_ID	Name	Age	Address	Salary
2	Kamal	25	Pune	1500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

SQL: IN OPERATOR

- The **IN** operator allows us **to specify multiple values in a WHERE clause.**
  - It allows us to easily test if an expression matches any value in a list of values
- The **IN** operator is **a shorthand for multiple OR conditions.** So it **used to replace many OR conditions.**
- **Syntax:**

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN(value1, value2, ... valueN);
```

OR (In Subqueries)

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN(SELECT STATEMENT);
```

Subquery

A subquery is a *select-from-where* expression that is nested within another query.

Example: IN OPERATOR

Example 1:

It selects all Employees located in Delhi, Bilaspur and Patna

```
SELECT *
FROM Emp
WHERE Address IN ('Delhi', 'Bilaspur','Patna);
```

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

Example 2:

It selects all Employees whose salary is either 2000, 4500 or 10000

```
SELECT *
FROM Emp
WHERE Salary IN (2000, 4500,10000);
```

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
3	Karan	23	Pune	2000.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

Example: IN OPERATOR using Subquery

Example 3:

It selects all employees that are from the same address as the customers

```
SELECT *
FROM Emp
WHERE Address IN(SELECT Address FROM Customers);
```

Address
Delhi
Pune
Pune
Mumbai
Mumbai
Bilaspur
Patna

Address
Delhi
Bhopal
Bangaluru
Chennai
Bilaspur

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
6	Kajal	22	Bilaspur	4500.00

SQL: NOT IN OPERATOR

- **NOT IN operator** is just opposite of **IN operator**
- **Syntax**:

```
SELECT column_name(s)
FROM table_name
WHERE column_name NOT IN (value1, value2, ... valueN);
```

***Example*:**

It selects all Employees *not* located in Delhi, Bilaspur and Patna

```
SELECT *
FROM Emp
WHERE Address NOT IN ('Delhi','Bilaspur','Patna');
```

Emp_ID	Name	Age	Address	Salary
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00

SQL: BETWEEN OPERATOR

- **BETWEEN** operator (**BETWEEN...AND**) is used **to select values within given range** (given minimum and maximum values).
- The values can be numbers, text, or dates.
- The **BETWEEN** operator is **inclusive**: **begin** and **end** values are **included.**
- **Syntax**:

```
SELECT column_name(s)
FROM table_name
WHERE column_name BETWEEN value1 AND value2;
```

Example:

It selects all Employees with the Salary between 2000 to 7000

```
SELECT *
FROM Emp
WHERE Salary BETWEEN 2000 AND 7000;
```

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
6	Kajal	22	Bilaspur	4500.00

SQL: NOT BETWEEN OPERATOR

- **NOT BETWEEN operator** is opposite of **BETWEEN operator**
- **Syntax**:

```
SELECT column_name(s)
FROM table_name
WHERE column_name NOT BETWEEN value1 AND value2;
```

Example:

It selects all Employees with the salary not between 2000 to 7000

```
SELECT *
FROM Emp
WHERE Salary NOT BETWEEN 2000 AND 7000;
```

Emp_ID	Name	Age	Address	Salary
2	Kamal	25	Pune	1500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

SQL: LIKE OPERATOR

- The **LIKE operator** is used in a **WHERE clause** to search for a specified **pattern** in a column.
- The **LIKE operator** is **used to match string pattern values using two wildcard characters.**
- ***Wildcard Character***: *It is used to substitute one or more characters in a string.*
- There are **two wildcards used** in conjunction with the **LIKE operator.**
  - **The percent sign (%):** It represents **zero, one or multiple characters.**
  - **The underscore (_):** It represents only **a single character.**
- The **percent sign (%)** and the **underscore (_)** can also be used in **combinations**!

Syntax:

```
SELECT column1, column2, ...
FROM table_name
WHERE columnN LIKE pattern;
```

Example 1:

It selects all Employees with a *Name* starting with "K"

```
SELECT * FROM Emp
WHERE Name LIKE "K%";
```

Emp_ID	Name	Age	Address	Salary
2	Kamal	25	Pune	1500
3	Karan	23	Pune	2000.00
6	Kajal	22	Bilaspur	4500.00

Example 2:

It selects all Employees with a *Name* that have "a" in second position

```
SELECT * FROM Emp
WHERE Name LIKE "_a%";
```

Emp_ID	Name	Age	Address	Salary
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

Example 3:

It selects all Employees with a *Name* starting with "M" and are at least 4 characters in length

```
SELECT * FROM Emp
WHERE Name LIKE "M_ _ _%";
```

Emp_ID	Name	Age	Address	Salary
7	Mahi	24	Patna	10000.00

Example 4:

It selects all records from Emp where salary starts with 200

```
SELECT * FROM Emp
WHERE Salary LIKE "200%";
```

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
3	Karan	23	Pune	2000.00

Examples : LIKE operator with wildcard operator (% , _):

LIKE Operator	Description
WHERE Name LIKE 'A%'	Finds any values that start with "**A**", Eg: Aman, Aakriti
WHERE Name LIKE '%a'	Finds any values that end with "**a**", Eg: Neha, Soma
WHERE Name LIKE '%oh%'	Finds any values that have "**oh**" in any position, Eg: Sohan, Mohan
WHERE Name LIKE '_r%'	Finds any values that have "**r**" in the second position, Eg: Priya, Priyesh
WHERE Name LIKE 'K_ _ _ _%'	Finds any values that start with "**K**" and are at least **5** characters in length, Eg: Kamal, Karan, Kajal
WHERE Name LIKE 'A_ _%'	Find, Eg: s any values that start with "**A**" and are at least **3** characters in length, Eg: Aki, Anu
WHERE Name LIKE 'A%N'	Finds any values that start with "**A**" and ends with "**N**", Eg: Aman, Raman

SQL NULL Values

- Attributes can have **Null values**, if permitted by the schema definition for a table (i.e., **no *NOT NULL Constraint***),
- **NULL** represents a ***missing value, unknown value, non-existent or non-applicable value.***
- A **NULL value** in a table is a value in a field that appears to be **blank (empty)** i.e. with **no value.**

➢ **Note:** A **NULL value** is **different** than a **zero** value or a field that contains **spaces.**

➢ **A field with a NULL value is one that has been left blank during record creation!**

Emp_ID	Name	Age	Address	Salary	Email
1	Rahul	32	Delhi	2000.00	rr@gmail.com
2	Kamal	25	Pune	1500.00	kk@gmail.com
3	Karan	23	Pune	2000.00	NULL
4	Chirag	25	Mumbai	6500.00	cc@gmail.com
5	Harsh	32	Mumbai	8500.00	hh@gmail.com
6	Kajal	22	Bilaspur	4500.00	NULL
7	Mahi	24	Patna	10000.00	mm@gmail.com

SQL NULL Values...

- Result of any **arithmetic expression** involving ***null*** is ***null***
- Result of **where** clause condition is ***false*** if it evaluates to ***null***.
- **and, or, not operators** handles ***null*** as follows:

and	true	false	null
true	true	false	null
null	null	false	null
false	false	false	false

or	true	false	null
true	true	true	true
null	true	null	null
false	true	false	null

not	
true	false
null	null
false	true

SQL: IS NULL OPERATOR

- The **IS NULL operator** is **used to check for NULL values (or empty values).**
- It allows us to find out the set of records in which value for a particular column is **NULL**
- It returns **TRUE** if a NULL value is found, otherwise it returns **FALSE.**
- It can be used in a **SELECT**, **UPDATE**, or **DELETE** statement with **Where** Clause.

➢ **IS NULL Syntax:**

```
SELECT column_names
FROM table_name
WHERE column_name IS NULL;
```

Example: IS NULL OPERATOR

List employees who have no associated email Ids

```
SELECT *
FROM Emp
WHERE Email IS NULL;
```

Emp Table

Emp_ID	Name	Age	Address	Salary	Email
1	Rahul	32	Delhi	2000.00	rr@gmail.com
2	Kamal	25	Pune	1500.00	kk@gmail.com
3	Karan	23	Pune	2000.00	NULL
4	Chirag	25	Mumbai	6500.00	cc@gmail.com
5	Harsh	32	Mumbai	8500.00	hh@gmail.com
6	Kajal	22	Bilaspur	4500.00	NULL
7	Mahi	24	Patna	10000.00	mm@gmail.com

Emp_ID	Name	Age	Address	Salary	Email
3	Karan	23	Pune	2000.00	NULL
6	Kajal	22	Bilaspur	4500.00	NULL

SQL: IS NOT NULL OPERATOR

- **IS NOT NULL** operator is **opposite** of **IS NULL** operator
- The **IS NOT NULL operator** is **used to check for NOT NULL values** (or **non-empty values**
- It returns **TRUE** if a NULL value is not found, otherwise it returns **FALSE**.
- It can be used in a **SELECT**, **UPDATE**, or **DELETE** statement with **Where** Clause.
- **IS NOT NULL Syntax:**

```
SELECT column_names
FROM table_name
WHERE column_name IS NOT NULL;
```

SQL: IS NOT NULL OPERATOR

List employees who have associated email Ids

```
SELECT *
FROM Emp
WHERE Email IS NOT NULL;
```

Emp_ID	Name	Age	Address	Salary	Email
1	Rahul	32	Delhi	2000.00	rr@gmail.com
2	Kamal	25	Pune	1500.00	kk@gmail.com
4	Chirag	25	Mumbai	6500.00	cc@gmail.com
5	Harsh	32	Mumbai	8500.00	hh@gmail.com
7	Mahi	24	Patna	10000.00	mm@gmail.com

Emp Table

Emp_ID	Name	Age	Address	Salary	Email
1	Rahul	32	Delhi	2000.00	rr@gmail.com
2	Kamal	25	Pune	1500.00	kk@gmail.com
3	Karan	23	Pune	2000.00	NULL
4	Chirag	25	Mumbai	6500.00	cc@gmail.com
5	Harsh	32	Mumbai	8500.00	hh@gmail.com
6	Kajal	22	Bilaspur	4500.00	NULL
7	Mahi	24	Patna	10000.00	mm@gmail.com

SQL Aliases

- **SQL ALIASES** can be used to create a temporary name for **columns** or **tables.**
- **SQL Aliases** is used to give an alias name to a table or a column
  - **COLUMN ALIASES** are used to make column headings in result set easier to read.
  - **TABLE ALIASES** are used to shorten SQL query to make it easier to read or when there are more than one table is involved.
- SQL Aliases:
  - Aliases are created to make table or column names easier to read
  - An **alias** only exists for the duration of the query. i.e. The renaming is just a temporary change and the actual table name does not change in the database.
  - Aliases are useful when table or column names are big or not very readable.
  - These are preferred when there are **more than one** table involved in a query.

Syntax: SQL Aliases

- **Column Alias Syntax:**

```
SELECT column_name AS alias_name
FROM table_name;
```

- **Table Alias Syntax:**

```
SELECT column_name(s)
FROM table_name AS alias_name;
```

Example: Column Alias

Emp

Emp_ID	Name	Age	Address	Salary	Dept_Id
1	Rahul	32	Delhi	2000.00	3
2	Kamal	25	Pune	1500.00	3
3	Karan	23	Pune	2000.00	3
4	Chirag	25	Mumbai	6500.00	1
5	Harsh	32	Mumbai	8500.00	1
6	Kajal	22	Bilaspur	4500.00	1
7	Mahi	24	Patna	10000.00	1

```
SELECT Emp_ID AS ID, Name
FROM Emp;
```

ID	Name
1	Rahul
2	Kamal
3	Karan
4	Chirag
5	Harsh
6	Kajal
7	Mahi

Example: Table Alias

Emp

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	3
2	Kamal	25	Pune	1500.00	NULL
3	Karan	23	Pune	2000.00	3
4	Chirag	25	Mumbai	6500.00	1
5	Harsh	32	Mumbai	8500.00	1
6	Kajal	22	Bilaspur	4500.00	NULL
7	Mahi	24	Patna	10000.00	NULL

Dept

Dept_ID	D_Name
1	Sales
2	HR
3	Finance
4	Marketing

```
SELECT E. Emp_ID, E.Name,D.Dept_ID, D.D_Name
FROM Emp AS E, Dept AS D
WHERE E. Dept_ID=D.Dept_ID;
```

Emp_ID	Name	Dept_Id	D_Name
1	Rahul	3	Finance
3	Karan	3	Finance
4	Chirag	1	Sales
5	Harsh	1	Sales

SQL Functions

- SQL provides many built-in functions to perform operations on data.
- These functions are useful while performing mathematical calculations, string concatenations, sub-strings etc.
- SQL functions are divided into two categories:
  1. **Aggregate Functions**
  2. **Scalar Functions**
- **Aggregate functions and Scalar functions both** return a *single value.*
- **Aggregate functions** operate on *many records* while **Scalar functions** operate on *each record independently*

Aggregate Functions in SQL

- **Aggregate functions** performs calculations on set of values and **returns a single value.**
  1. **COUNT()**
  2. **SUM()**
  3. **AVG()**
  4. **MAX()**
  5. **MIN()**

- **Aggregate functions** are often used by **GROUP BY** clause of the **SELECT** statement ***to group multiple rows together as input to form a single value output***
- **Aggregate functions** ignore NULL values, except count(*)

1. COUNT() Function

- **COUNT()**: **Count function** returns the number of rows that matches the specified criterion
- **Syntax**:

```
SELECT COUNT(column_name)
FROM table_name
WHERE condition;
```

Example: COUNT()

Student Table

ID	Name	Marks
1	A	90
2	B	40
3	C	70
4	D	60
5	E	70
6	F	NULL

- ***Find the total number of student records***

  COUNT(*) *returns total no. of records*

```
SELECT COUNT(*)
FROM Student;
```

COUNT(*)
6

- ***Find the count of entered marks***

  COUNT(Marks) *returns no. of non null values over column Marks*

```
SELECT COUNT(Marks)
FROM Student;
```

COUNT(Marks)
5

- ***Find the count of distinct entered marks***

  COUNT(DISTINCT Marks) *returns no. of distinct non null values over column Marks*

```
SELECT COUNT(DISTINCT Marks)
FROM Student;
```

COUNT(DISTINCT Marks)
4

2. SUM() Function

- **SUM(): Sum function** returns total sum of a selected **numeric** columns.
- **Syntax:**

```
SELECT SUM(column_name)
FROM table_name
WHERE condition;
```

Example: SUM()

Student Table

ID	Name	Marks
1	A	90
2	B	40
3	C	70
4	D	60
5	E	70
6	F	NULL

- ***Find the sum of student marks***

SUM(Marks) *sum all non null values of column marks*

```
SELECT SUM(Marks)
FROM Student;
```

SUM(Marks)
330

- ***Find the distinct sum of student marks***

SUM(DISTINCT Marks) *sum all distinct non null values of column marks*

```
SELECT SUM(DISTINCT Marks)
FROM Student;
```

SUM(DISTINCT Marks)
260

Using 'AS' for Renaming the column

```
SELECT SUM(Marks) AS 'Sum Marks'
FROM Student;
```

Sum Marks
330

3. AVG() Function

- **AVG(): Average function** returns the average value of selected **numeric** column.
- **Syntax:**

```
SELECT AVG(column_name)
FROM table_name
WHERE condition;
```

Example: AVG()

Student Table

ID	Name	Marks
1	A	90
2	B	40
3	C	70
4	D	60
5	E	70
6	F	NULL

- *Find the average marks of students*

```
SELECT AVG(Marks)
FROM Student;
```

AVG(Marks)
66

AVG(Marks) = SUM(Marks)/COUNT(Marks)= 330/5 =66

- *Find the distinct average marks of students*

```
SELECT AVG(DISTINCT Marks)
FROM Student;
```

AVG(DISTINCT Marks)
65

AVG(DISTINCT Marks) = SUM(DISTINCT Marks)/COUNT(DISTINCT Marks)= 260/4 =65

Using 'AS' for Renaming the column

```
SELECT AVG(Marks) AS 'Average Marks'
FROM Student;
```

Average Marks
66

4. MAX() Function

- **MAX(): Maximum function** returns maximum value of selected column
- **Syntax:**

```
SELECT MAX(column_name)
FROM table_name
WHERE condition;
```

Example: MAX()

- *Find the maximum student marks:* **MAX(Marks)**

```
SELECT MAX(Marks)
FROM Student;
```

MAX(Marks)
90

Student Table

ID	Name	Marks
1	A	90
2	B	40
3	C	70
4	D	60
5	E	70
6	F	NULL

5. MIN() Function

- **MIN(): Minimum function** returns minimum value of selected column.
- **Syntax:**

```
SELECT MIN(column_name)
FROM table_name
WHERE condition;
```

Example: MIN()

- *Find the minimum students marks:* MIN(Marks)

```
SELECT MIN(Marks)
FROM Student;
```

MIN(Marks)
40

Student Table

ID	Name	Marks
1	A	90
2	B	40
3	C	70
4	D	60
5	E	70
6	F	NULL

Scalar Functions

- Scalar functions return a single value from an input value.

1. UCASE()
2. LCASE()
3. MID()
4. LENGTH()
5. ROUND()
6. NOW()
7. FORMAT()

1. UCASE() Function

- **UCASE(): Upper case function** is used to convert value of string column to uppercase characters.
- **Syntax:**

```
SELECT UCASE(column_name)
FROM table_name;
```

Example: UCASE()

- *Converting string values of name column to upper case*

```
SELECT UCASE(Name)
FROM Emp;
```

UCASE(Name)
RAHUL
KAMAL
KARAN
CHIRAG
HARSH
KAJAL
MAHI

Emp Table

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

2. LCASE() Function

- **LCASE(): Lower case function** is used to convert value of string column to lower case characters.
- **Syntax:**

```
SELECT LCASE(column_name)
FROM table_name;
```

Example: LCASE()

- *Converting string values of name column to lower case*

```
SELECT LCASE(Name)
FROM Emp;
```

LCASE(Name)
rahul
kamal
karan
chirag
harsh
kajal
mahi

Emp Table

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

3. MID() Function

- **MID(): MID function** is used to extract substrings from column values of string type in a table.
- **Syntax:**

```
SELECT MID(column_name, start, length)
FROM table_name;
```

Note:

Length is optional

Start signifies the starting position of string

Example: MID()

- *Extract sub-string from name column from second position having length of 3*

```
SELECT MID(Name,2,3)
FROM Emp;
```

MID(Name,2,3)
ahu
ama
ara
hir
ars
aja
ahi

Emp Table

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

4. LENGTH() Function

- **LENGTH(): Length function** returns the length of a string in the column.
- **Syntax:**

```
SELECT LENGTH(column_name)
FROM table_name;
```

Example: LENGTH()

Find the length of string of name column

```
SELECT LENGTH(Name)
FROM Emp;
```

LEN(Name)
5
5
5
6
5
5
4

Emp Table

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

5. ROUND() Functio..

- **ROUND():** The **ROUND() function** is used to round a **numeric** column to the number of decimals specified.
- **Syntax:**

```
SELECT ROUND(column_name, decimals)
FROM table_name;
```

Note:
Decimals represents **number of decimals to be fetched.**

Example: ROUND()

- *Find the round-off salary of employees*

```
SELECT ROUND(Salary)
FROM Emp;
```

or

```
SELECT ROUND(Salary,0)
FROM Emp;
```

ROUND(Salary)
2001
1500
2000
6501
8500
4500
10000

Emp Table

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.50
2	Kamal	25	Pune	1500.20
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.80
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

Example: ROUND()

- *Find round-off salary of employees (upto 1 decimal place)*

```
SELECT ROUND(Salary,1)
FROM Emp;
```

ROUND(Salary,1)
2001.5
1500.2
2000.0
6501.8
8500.0
4500.0
10000.0

Emp Table

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.50
2	Kamal	25	Pune	1500.20
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.80
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

6. NOW() Function

- **NOW()**: The **NOW() function** returns the **current system date and time.**
- **Syntax:**

```
SELECT NOW()
FROM table_name;
```

Example: NOW()

- *Fetching current system date & time*

```
SELECT NOW() ;
```

NOW()
2021-01-25 13:40:25

- *Fetching current system date & time in a column*

```
SELECT Name, NOW() AS 'Date & Time'
FROM Emp;
```

Name	Date & Time
Rahul	2021-01-25 13:40:25
Kamal	2021-01-25 13:40:25
Karan	2021-01-25 13:40:25
Chirag	2021-01-25 13:40:25
Harsh	2021-01-25 13:40:25
Kajal	2021-01-25 13:40:25
Mahi	2021-01-25 13:40:25

7. FORMAT() Function

- **FORMAT():** The **FORMAT()** function is used to format how a column is to be displayed.
- **Syntax:**

```
SELECT FORMAT(column_name, format)
FROM table_name;
```

Example: FORMAT()

- *Formatting current date as 'YYYY-MM-DD'*

```
SELECT Name, FORMAT(NOW(),'YYYY-MM-DD') AS 'Date'
FROM Emp;
```

Name	Date
Rahul	2021-01-25
Kamal	2021-01-25
Karan	2021-01-25
Chirag	2021-01-25
Harsh	2021-01-25
Kajal	2021-01-25
Mahi	2021-01-25

ORDER BY Clause

- The output of the **SELECT** queries do not have any specific order. *The* ***ORDER BY Clause allows us to specify order for the query output.***
- **ORDER BY Clause** is used with **SELECT statement** for arranging retrieved data in sorted order.
- The **ORDER BY Clause** by default sorts the retrieved data in **ascending order** [**ASC**].
- To sort the data in **descending order**, **DESC** keyword is used with ORDER BY clause.
- **Syntax**:

```
SELECT column-list
FROM table_name
[WHERE conditions]
ORDER BY column1, column2, .. columnN [ASC | DESC];
```

Example Table: Emp

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
7	Mahi	24	Patna	10000.00

Query 1 : (Eg: Ascending Order - Default)

- To display details of Emp table in ascending order by *salary*

```
SELECT *
FROM Emp
ORDER BY Salary;
```

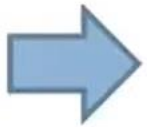
ASC

Emp_ID	Name	Age	Address	Salary
2	Kamal	25	Pune	1500.00
1	Rahul	32	Delhi	2000.00
3	Karan	23	Pune	2000.00
6	Kajal	22	Bilaspur	4500.00
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
7	Mahi	24	Patna	10000.00

Query 2: (Eg: Ascending Order with multiple columns)

- To sort details of Emp table in ascending order by *name* and *salary*

```
SELECT *
FROM Emp
ORDER BY Name, Salary;
```



Emp_ID	Name	Age	Address	Salary
4	Chirag	25	Mumbai	6500.00
5	Harsh	32	Mumbai	8500.00
6	Kajal	22	Bilaspur	4500.00
2	Kamal	25	Pune	1500.00
3	Karan	23	Pune	2000.00
7	Mahi	24	Patna	10000.00
1	Rahul	32	Delhi	2000.00

In ORDER BY Clause multiple columns can be used.

In this case:

1st column is used as **primary ordering** field,

2nd column is used as **secondary ordering** field,

and so on

Query 3: (Eg: Descending Order)

- To sort details of Emp table in descending order by name

```
SELECT *
FROM Emp
ORDER BY Name DESC;
```

Emp_ID	Name	Age	Address	Salary
1	Rahul	32	Delhi	2000.00
7	Mahi	24	Patna	10000.00
3	Karan	23	Pune	2000.00
2	Kamal	25	Pune	1500.00
6	Kajal	22	Bilaspur	4500.00
5	Harsh	32	Mumbai	8500.00
4	Chirag	25	Mumbai	6500.00

GROUP BY Clause

- In SQL, the **GROUP BY** statement is used for organizing similar data into groups.
  - For Eg: "find the number of customers in each **country**".
- The **GROUP BY** statement is often used with **aggregate functions** (COUNT, MAX, MIN, SUM, AVG) to group the result-set by one or more columns.
- **Imp Points:**
  - **GROUP BY** clause is **used with** the **SELECT** statement in the **SQL** query.
  - **GROUP BY** clause is **placed after** the **WHERE** clause in **SQL**.
  - **GROUP BY** clause is **placed before** the **ORDER BY** clause in **SQL**.

Syntax:

```
SELECT column_name(s), aggregate_function(column_name)
FROM table_name
[WHERE condition]
GROUP BY column_name(s)
[ORDER BY column_name(s)];
```

Example Table: Emp

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Query 1:

(GROUP BY with COUNT aggregate function)

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Write a query to find the number of employee in each department

```
SELECT Dept_id, COUNT(*)
FROM Emp
GROUP BY Dept_id
```

Dept_ID	COUNT(*)
500	3
501	2
502	2

Query 2:

(GROUP BY with COUNT and ORDER BY)

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Write a query to find the number of employee in each department, sorted low to high

```
SELECT Dept_id, COUNT(*)
FROM Emp
GROUP BY Dept_id
ORDER BY COUNT(*);
```

Dept_ID	COUNT(*)
501	2
502	2
500	3

Query 3:

(GROUP BY with SUM aggregate function)

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Write a query to find the sum of salaries of employees from each department

```
SELECT Dept_id, SUM(Salary) AS 'SumSalary'
FROM Emp
GROUP BY Dept_id;
```

Dept_ID	SumSalary
500	5500.00
501	15000.00
502	14500.00

Query 4:

(GROUP BY with AVG aggregate function)

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Write a query to find the average of salaries of employees from each department

```
SELECT Dept_id, ROUND(AVG(Salary))
FROM Emp
GROUP BY Dept_id;
```

Dept_ID	AVG(Salary)
500	1833
501	7500
502	7250

Query 5:

(GROUP BY with MIN aggregate function)

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Write a query to find the minimum salary of employee from each department

```
SELECT Dept_id, MIN(Salary)
FROM Emp
GROUP BY Dept_id;
```

Dept_ID	MIN(Salary)
500	1500.00
501	6500.00
502	4500.00

Query 6:

(GROUP BY with MAX aggregate function)

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Write a query to find the maximum salary of employee from each department

```
SELECT Dept_id, MAX(Salary)
FROM Emp
GROUP BY Dept_id;
```

Dept_ID	MAX(Salary)
500	2000.00
501	8500.00
502	10000.00

Query 7:

(GROUP BY with MIN & MAX both)

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Write a query to find the minimum and maximum salary of employee from each department

```
SELECT Dept_id, MIN(Salary), MAX(Salary)
FROM Emp
GROUP BY Dept_id;
```

Dept_ID	MIN(Salary)	MAX(Salary)
500	1500.00	2000.00
501	6500.00	8500.00
502	4500.00	10000.00

Query 8:

(GROUP BY with multiple columns)

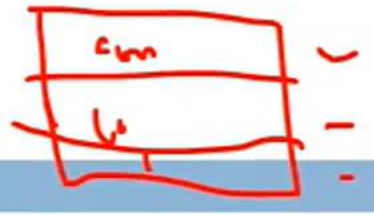
Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Write a query to find the number of employees in each department location wise

```
SELECT Dept_id, Address, COUNT(*)
FROM Emp
GROUP BY Dept_id, Address;
```

Dept_ID	Address	COUNT(*)
500	Delhi	1
500	Pune	2
501	Mumbai	2
502	Bilaspur	1
502	Patna	1

HAVING Clause



- **HAVING** clause is used with **GROUP BY** clause & filter the groups created by the **GROUP BY** clause
- The **HAVING clause** enables you to specify **conditions** that filter which **group** results appear in the results.
- The **HAVING** clause was added to SQL because the *aggregate functions like* ***SUM, AVG, MIN, MAX,COUNT*** cannot be used with **WHERE** clause.
- The **WHERE clause** places conditions on the **selected columns**, whereas the **HAVING** clause places conditions on **groups** created by the **GROUP BY clause**
- In SQL query, **HAVING** clause is placed after the **GROUP BY** clause

Syntax:

```
SELECT column_name(s), aggregate_function(column_name)
FROM table_name
[WHERE condition]
GROUP BY column_name(s)
HAVING condition
[ORDER BY column_name(s)];
```

Order of Clauses(in SQL):

1. SELECT
2. FROM
3. WHERE
4. GROUP BY
5. HAVING
6. ORDER BY

Example Table: Emp

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Query 1:

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Write a query to display departments where the number of employee is greater than 2

```
SELECT Dept_id, COUNT(*)
FROM Emp
GROUP BY Dept_id
HAVING COUNT(*) >2;
```

Dept_ID	COUNT(*)
500	3

Query 2:

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Write a query to find the department for which the sum of salaries of employees is more than 10000

```
SELECT Dept_ID, SUM(Salary)
FROM Emp
GROUP BY Dept_id
HAVING SUM(Salary)>10000;
```

Dept_ID	SUM(Salary)
501	15000.00
502	14500.00

Query 3:

Emp_ID	Name	Age	Address	Salary	Dept_ID
1	Rahul	32	Delhi	2000.00	500
2	Kamal	25	Pune	1500.00	500
3	Karan	23	Pune	2000.00	500
4	Chirag	25	Mumbai	6500.00	501
5	Harsh	32	Mumbai	8500.00	501
6	Kajal	22	Bilaspur	4500.00	502
7	Mahi	24	Patna	10000.00	502

Write a query to find the sum of salaries of employee that lives at same location

```
SELECT Address, SUM(Salary)
FROM Emp
GROUP BY Address
HAVING COUNT(Address)>=2;
```

Address	SUM(Salary)
Mumbai	15000.00
Pune	3500.00

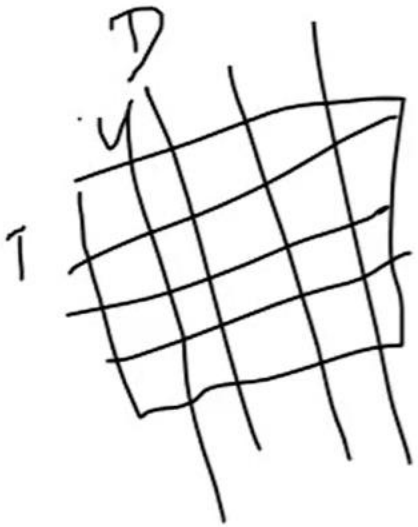
Relational Calculus

Suggested: Relational Algebra | Relational Algebra Operators | DB...

- **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
  - $\mathbf{P}$ 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 (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**($\forall$).

- $\exists t \in r\,(Q(t)) \equiv$ "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.**

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

(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**

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

OR

Union

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

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

∩

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:

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

(Join)

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

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

(Join)

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)

Database Users

- **Database Users** are the persons who interact with the database and take the benefits of database.
- Users are differentiated by the way they expect to interact with the system.
- Four types of DB users:
  1. **Naive users/Native users/End users**
  2. **Application programmers**
  3. **Sophisticated users**
  4. **Specialized users**

Types of Database Users

1. **Naive Users/Native Users/End Users**: They are the unsophisticated users who use the existing application to interact with the database.
   - *Example: People accessing database over the web, bank tellers, clerical staffs, etc.*
2. **Application Programmers**: They are the computer professionals who write the application programs. They interact with system through DML queries.
   - *For example: Writing a C program to generate the report of employees who are working in particular department, will involve a query to fetch the data from database. It will include a embedded SQL query in the C Program.*

Types of Database Users …

3. **Sophisticated Users**: They interact with the system by writing SQL queries directly through the query processor (like SQL) without writing application programs.
   - *Example: Analysts, who submits SQL queries to explore data in the DBMS.*
4. **Specialized Users**: They are also sophisticated users who write specialized database applications that do not fit into the traditional data processing framework. They are the developers who develop the complex programs to the requirement.
   - *Example: Computer-Aided Design (CAD) Systems, Expert Systems, Knowledge Based System, etc. that store complex data types (graphics and audio data) & environment modelling systems*

Database Administrators (DBAs)

- DBA is a person or group that is responsible for supervising both the database and the use of the DBMS.
- Database Administrators (DBAs) coordinate all the activities of the database system.
- They use specialized software to store and organize data
- They have all the permissions

DBAs Tasks

- Schema definition
- Storage structure and access method definition
- Schema and physical organization modification
- Specifying integrity constraints
- Granting user authority to access database
- Monitoring performance & responding to changes in requirements
- Routine Maintenance
- Acting as liaison with users
- Backing up and restoring databases

Transaction Management

- ***What if system fails?***
- ***What if more than one user is concurrently updating the same data?***
- A ***Transaction*** is a collection of operations that performs a single logical function in a database application
  - Each **transaction** is a unit of both atomicity and consistency. Thus, we require that *transaction* do not violate any database consistency constraints
  - For example: Lets say your account is A and your friend's account is B and you are transferring 10000 from A to B, the set of operations or the steps of the transaction are

1. Read your account balance	R(A);
2. Deduct the amount from your balance	A = A - 10000;
3. Write the remaining balance to your account	W(A);
4. Read your friend's account balance	R(B);
5. Add the amount to his account balance	B = B + 10000;
6. Write the new updated balance to his account	W(B);

  - This whole set of operations can be called a **transaction.** R is Read and W is Write

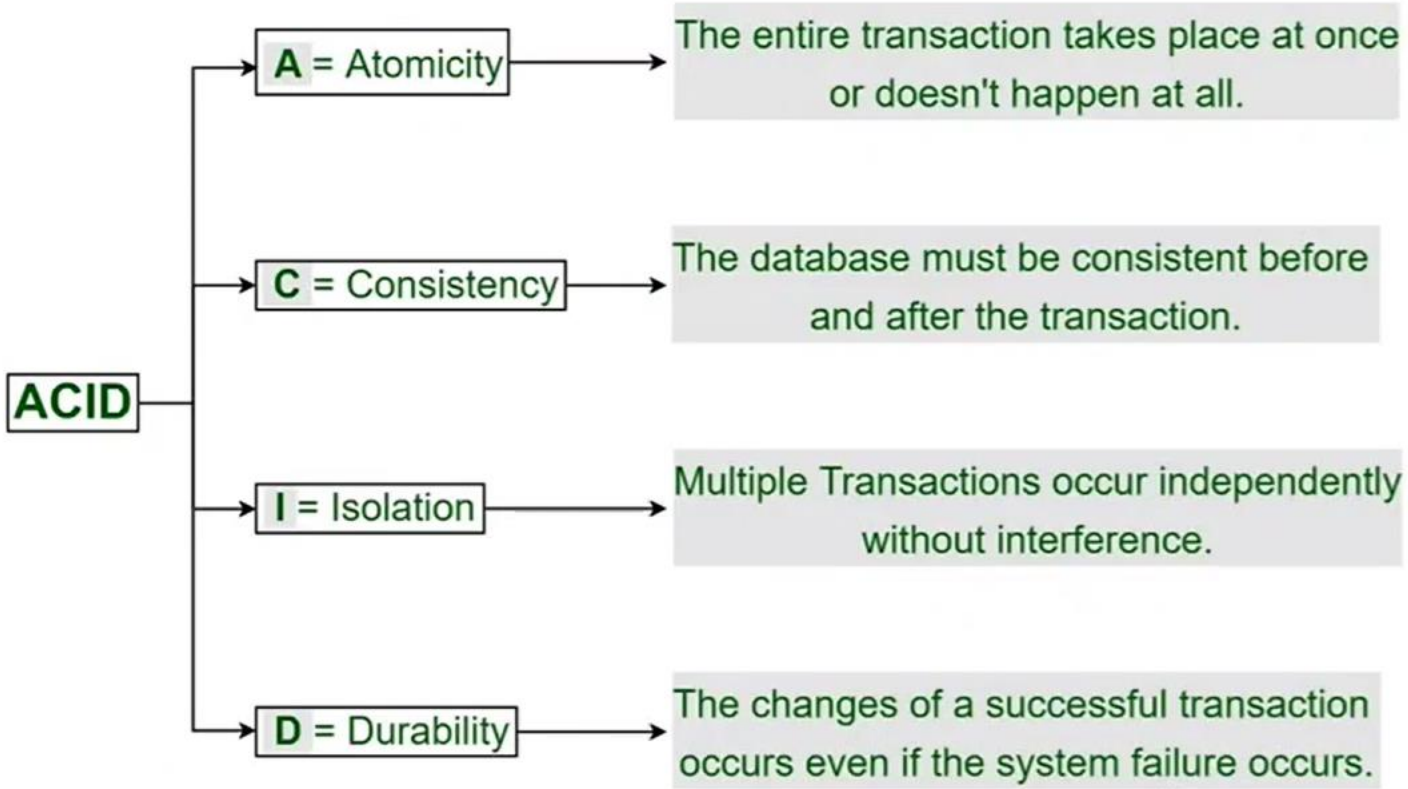
Transaction Management...

- **Transaction-management component** ensures that the database remains in a consistent (correct) state despite system failures (e.g., power failures and operating system crashes) and transaction failures.
- **Concurrency-control manager** controls the interaction among the concurrent transactions, to ensure the consistency of the database.

Transaction Management...

- In order to maintain consistency in a database, before and after the transaction, certain properties are followed. These are called **ACID** properties

ACID Properties in DBMS



Storage Management

- **Storage manager** is a program module that provides the interface between the *low-level data stored in the database* and *the application programs and queries submitted to the system.*
- The storage manager is responsible to the following tasks:
  - Interaction with the OS file manager
  - Efficient storing, retrieving and updating of data
- Issues:
  - Storage access
  - File organization
  - Indexing and hashing

Database Access from Application Programs

- **SQL**: widely used non-procedural, commercial language
- Example: Find the name of the instructor with ID 22222

```
select   name
from     instructor
where    instructor.ID = '22222'
```

- **SQL** is NOT a Turing equivalent language which means that everything that need to be computed cannot be computed in SQL
- **SQL** alone is not very powerful; because there are some computations that cannot be obtained by any SQL query. Such computations must be written in a host language, such as C, C++, Java with embedded SQL queries that access the data in the database
  - Application Programs are programs that are used to interact with the database in this fashion. Eg: Banking system are programs that generate: Payroll checks, Debit accounts, Credit Accounts or transfer funds between accounts
  - To access the database, DML statements need to be executed from the host language
- **Application Programs generally access the database through one of the 2 ways:**
  1. APIs (ODBC/JDBC) which allows SQL Queries to be sent to database
  2. Language extensions to allow embedded SQL

Application Programs generally access the DB through one of the 2 ways

1. *By providing an Application Programming Interface i.e API (the set of procedures) that can be used to send DDL & DML statements (SQL Queries) to the database and retrieve the results* i.e. **APIs (ODBC/JDBC) which allows SQL Queries to be sent to database**

   - Eg: 1. ODBC (Open Database Connectivity) standard
     - It is created by Microsoft in 1992 that is used by Windows software applications, using C/C++ language, to access databases via SQL

     2. JDBC (Java Database Connectivity) standard
     - It is created by Sun Microsystems in 1997 that is used by JAVA application, using only JAVA Language, to access databases via SQL

2. *By extending the host language syntax to embed DML calls within the host language program* i.e **Language extensions to allow embedded SQL**

   - Eg: C,C+, JAVA with embedded SQL queries
   - Usually special character (like EXEC) prefaces DML Calls, and a Preprocessor called the DML Pre-Compiler, that converts the DML statements to normal procedure calls in the host language