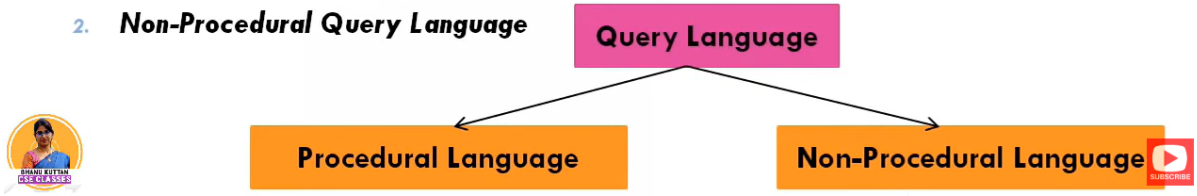
Relational Query Languages

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



Procedural vs. Non-Procedural

1. Procedural Query language:

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

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

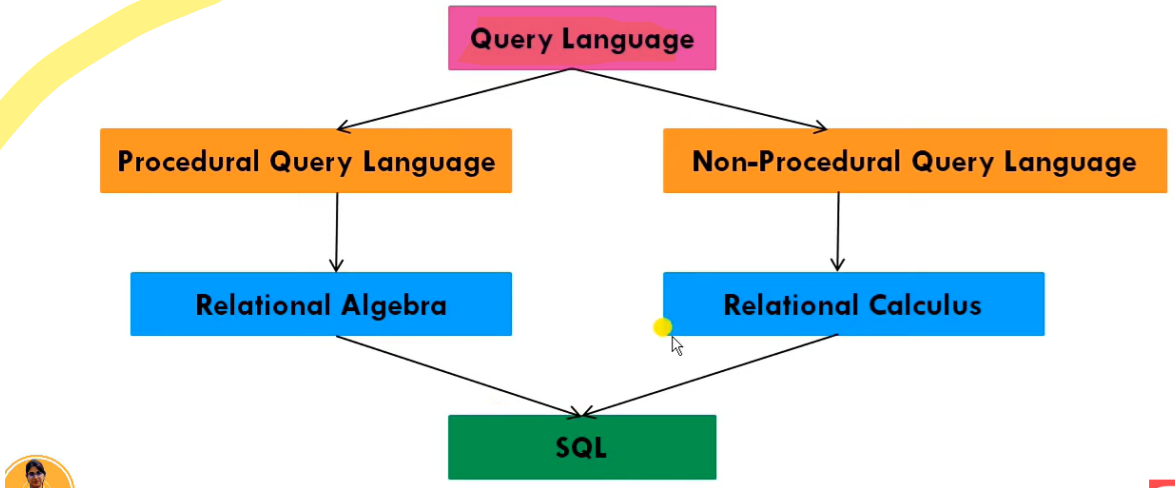
- In ***Non-procedural query language***, user instructs the system to produce the desired result without telling the step by step process.
- **User** tells ***what data to be retrieved from database but doesn't tell how to retrieve it.***

Two "Pure" Query languages

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

Relational Algebra vs. Relational Calculus

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



Relational Algebra, Calculus, RDBMS & SQL:

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

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

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

Relational Algebra

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

  1. Basic/Fundamental Operators
  2. Additional/Derived Operators
- Relational algebra operations work on one or more relations to define another relation without changing the original relations.
- Relational algebra operations are performed recursively on a relation

Relational Algebra Operations

1. Basic/Fundamental Operations:

1. Selection (σ)
2. Projection (Π)
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

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

Examples

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

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

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

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

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

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

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

Student

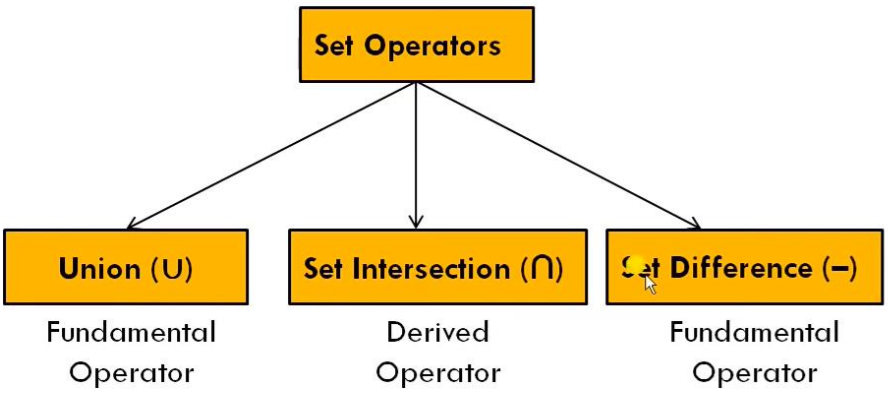
roll_no	name	age
1	A	20
2	B	17
3	C	16
4	D	19
5	E	18
6	F	18

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

Set Operators in Relational Algebra

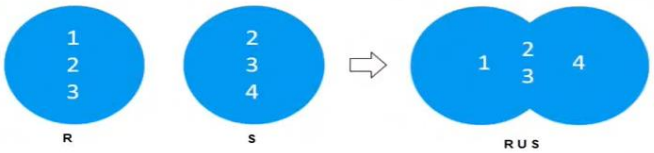


Set Operators

- **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

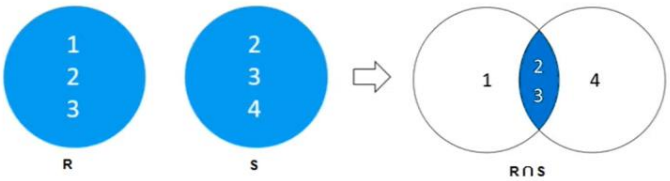
Union (U) Operator

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



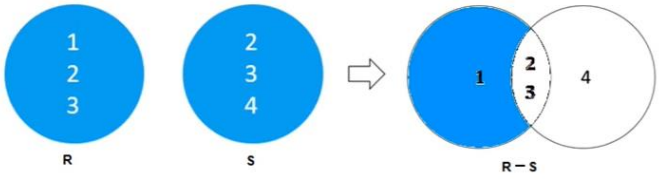
Set Intersection (∩) Operator

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



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);



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**

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

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

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)***

  (Rename R to S)

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-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.

  ρ_{L} (Loan)

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

A	B
1	a
2	b

S

B	C
a	3
b	4

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

2. Outer join:

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

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

Inner Join

- 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

2. Equi Join

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

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

3. Natural Join

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

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

Outer Join Types

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

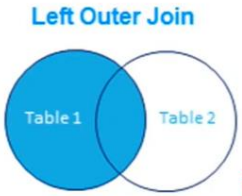
  (from left table, right table or both table)

- 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)

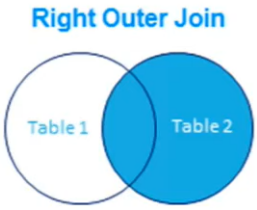
When applying **join** on two relations R1 and R2, some tuples of R1 or R2 does not appear in result set which does not satisfy the join conditions. **But..**

- 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**



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**



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**

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.*

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

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

Expressing A/B Using Basic Operators

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

Idea:

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

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

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

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

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

Assignment Operator

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

Example:

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

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

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

Extended Relational Algebra

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 Query

Suggested: Set Operators in Relational Algebra | Union-Intersection...

- 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 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 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

Modification of the Database

- The **content of the database** may be **modified** using the following operations:
  1. **Deletion**
  2. **Insertion**
  3. **Updating**

1. Deletion

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

$$r \leftarrow r - E$$

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

Example: Deletion

r

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

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

A	B	C
1	1	10
1	2	10

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

Deletion Examples Query

- Delete all account records in the Perryridge branch.

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

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

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

- Delete all accounts at branches located in Needham.

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

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

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

$$account \leftarrow account - r_2$$

$$depositor \leftarrow depositor - r_3$$

2. Insertion

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

$$r \leftarrow r \cup E$$

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

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

3. Updating

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

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

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

Example: Update

r

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

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

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

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

A	B	C
1	2	10
1	4	10

Introduction to Views

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

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

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

What are Views

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

View Definition

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

  create view v as <query expression>

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

  Example: *In SQL,*

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

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

  Example: *In SQL,*

  ```
  select * from v;
  ```

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

View in Relational algebra: Example 1

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

  create view *loan-customer* as

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

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

  loan-customer

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

Drop View

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

drop view *viewname*

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

Views: Summary

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

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

Types of Views:

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

Materialized Views

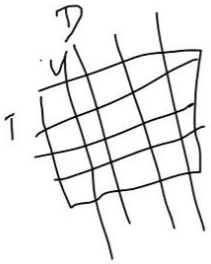
For some views, there is a term called ***materialization***. *So some views are materialized.* This depends on the query engine etc., the database engine.

- **Certain database systems allow view relations to be stored**, but they make sure that, if the actual relations used in the view definition change, the view is kept up to date. Such views are **called materialized views**.
  - The **process of keeping the view up to date** is called **view maintenance**.
  - Applications that use a view frequently in multiple queries benefited from the use of **materialized views** because then query expression is not going to be evaluated at run time
  - Of course, the benefits to queries from the materialization of a view must be weighed against the storage costs and the added overhead for updates.

Relational Calculus

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

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



Relational Calculus ...

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

1. Tuple Relational Calculus (TRC)

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

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

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

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

Predicate Calculus Formula

1. **Set of attributes and constants**
2. **Set of comparison operators:** e.g., $<, \leq, =, \neq, >, \geq$
3. **Set of connectives:** and ($\wedge$), or (v), not ($\neg$)
4. **Implication** ($\Rightarrow$): $x \Rightarrow y$, if x is true, then y is true $\quad x \Rightarrow y \equiv \neg x \vee y$
5. **Quantifiers:** **Existential Quantifiers** ($\exists$) **and Universal Quantifier**($\forall$).
   - $\exists t \in r\,(Q(t)) \equiv$ "there exists" a tuple in t in relation r such that predicate $Q(t)$ is true
   - $\forall t \in r\,(Q(t)) \equiv Q$ is true "for all" tuples t in relation r

Free and Bound variables:

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

Example Queries: TRC

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

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

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

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

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

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

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

Example Queries: TRC...

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

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

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

Join

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

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

Union

OR

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

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

∩

AND

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**

Example Queries: DRC

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

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

(Selection)

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

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

(Selection then Projection

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

Example Queries: DRC …

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

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

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

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

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