Unit-2 Relational Model

→ A relational database is a collection of tables, each of which is assigned a unique name.

→ A row in table represents a relationship among a set of values.

→ The concept of table is closely related with concept of relation in mathematics, from which the relational data model name has come up. Following to that, concept of relation has been introduced.

* Basic structure : Consider table named as account.

	AC_NO	branch	balance
	101	Baroda	500
Account ⇒ table	102	Ahmedabad	300
	103	Surat	100
	104	Rajkot	200

→ This table has 3 columns : AC_NO, branch & balance. These column headings such as ac_no, balance etc. are called as attribute.

→ for each attribute there is a permitted set of values, called the domain of that attribute.
example → For domain of branch attribute, we consider set of all branch names.

→ Let D_1 denote set of all account numbers, D_2 the set of all branch names and D_3 the set of all balance. Any row of account must consist of triplet (v_1, v_2, v_3) where v_1 is account no, v_2 is a branch name, v_3 is a balance. In general, account will contain only a subset of set of all possible rows. Thus, account is a subset of $D_1 \times D_2 \times D_3$.

In general, a table of n attributes is a subset of → $D_1 \times D_2 \times D_3 \ldots \times D_{n-1} \times D_n$

Relation :- It is a subset of Cartesian product of a list of domains.

→ We can say that table is a relation and row is a tuple. A tuple variable is a variable that stands for a tuple; i.e. a tuple variable is a variable whose domain is the set of all tuples.

→ For, tuple variable t and relation r, we can say that $t \in r$ to denote that tuple t is in relation r. The order in which tuples appear in a relation is irrelevant, since relation is set of tuples.

Type of domain :-

1) atomic domain :- If elements of domain are indivisible units then domain is called as atomic.
2) non-atomic domain :- If elements of domain are further divisible then domain is called as non-atomic

example → set of integers is an atomic domain, but set of all sets of integers is a non atomic domain.

⇒ We require that, for all relations r, domains of all attributes must be atomic.

⇒ NULL value can be member of any domain. NULL means unknown or does not exist. ~~Atta~~ Null values cause a number of difficulties when we access or update database, so it should be eliminated if possible.

* Database schema:

→ Database schema is the logical design of database. It contains details of all relations of database.

→ Database instance is a snapshot of data in the database at a given instant of time

→ Relation schema consists list of attributes and their domains as well as integrity constraints.

→ Relation schema corresponds to programming language notion of type definition. The concept of a relation corresponds to notion of variable.

* Convention for relation & relational schema:

We use lowercase names for relations, and first letter uppercase and others in lowercase for relational schema.

example→ relation 'account' is given with attributes account-number, branch-name and balance. Schema for this relation is given as

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

account is a relation on Account-schema, it is depicted as: account (Account_schema)

→ Relation instance is a snapshot of data of relation at any moment of time. Concept of relation instance is similar to value of a variable in programming language. Value of variable may change with time; similarly contents of Relation instance may change with time as the relation is updated.

* Foreign Key :- One relation includes primary key of another relation as an attribute. This attribute is called as foreign key.

→ If r1 relation uses primary key of r2, it is called as foreign key of r1 relation. Relation r1 is called as referencing relation and r2 is called as referenced relation.

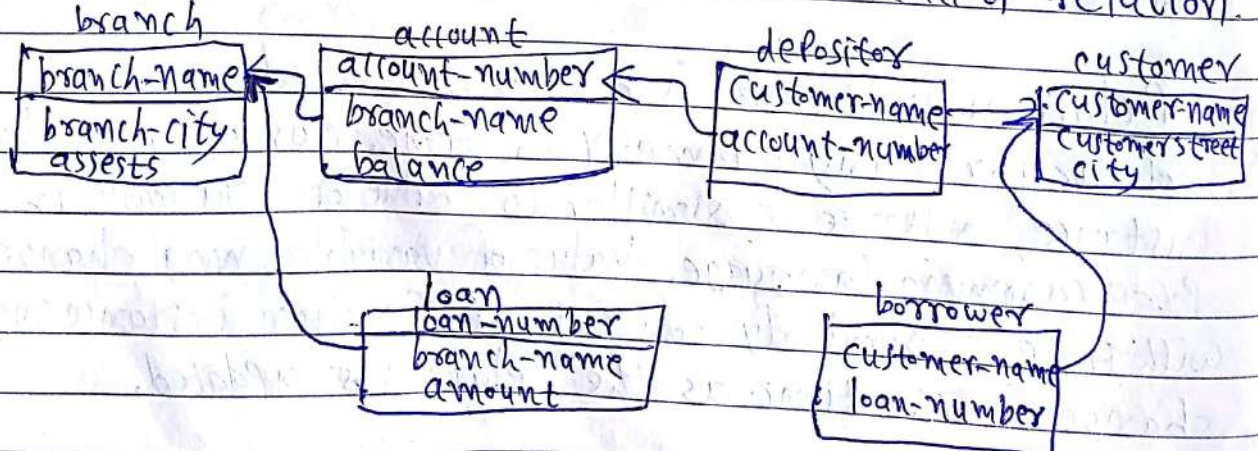
example→ branch-name attribute is a foreign key for account relation and primary key for branch relation.

*Schema diagram:-

A database schema along with primary key and foreign key dependencies, can be depicted pictorially by schema diagram.

→ In schema diagram each relation appears as a box, with attributes listed inside it and relation name above box. Primary key attributes are listed above one horizontal line inside box.

→ Foreign key dependencies appear as arrows from referencing relation to referenced relation.



Schema diagram for banking enterprise

SQL is non Procedural (4th Generation language)
PL/SQL is Procedural version of SQL.
↳ 3rd gen. language

→ Do not confuse schema diagram with E-R diagram. ER diagrams do not show foreign key explicitly, where as schema diagram do that.

Relational Algebra:

Relational algebra is a procedural query language. There are some Fundamental operations in relational algebra as well as some additional operations.

>> Fundamental Operations:-

1) Select operation:

→ Select operation selects the tuples that satisfy a given predicate. Lower-case letter sigma (σ) is used to denote selection. The predicate (condition) appears as subscript to σ. The argument relation is in parenthesis after σ.

→ In 'loan' relation, to select tuples where branch is 'Bombay', we write

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

Comparisons are done using $=, \neq, <, \leq, >, \geq$ in the predicate. We can also create larger predicate by combining several predicates by using connectives and ($\wedge$), or ($\vee$) and not ($\neg$).

example → To find tuples in which loan is more than ₹1200 and branch is bombay, we write:

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

2) Project operation:

Project operation is a unary operation which returns specific attributes of relation or Expression. (relational algebra expression).

→ relation is a set, so any duplicate rows are eliminated.

→ Projection is denoted by uppercase letter Pi (Π)

→ We denote those attributes that we want to display in the result as a subscript to Π. The relation is written as a argument in parenthesis

example→ If we want to list all loan-numbers and amount of loan relation then, it is written as:

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

3) Composition of Relational operations:-

Result of relational operation itself is a relation. More than one relational algebra operations can be applied one after another on relation to get desired result. Applying multiple operations on relation creates an relational algebra expression.

→ Composing relational algebra operations into relational algebra expression is like composing arithmetic operations (like +, −, *, ÷) into arithmetic expressions.

example→ "Find those customers who live in Surat," we can write query for this as:

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

4) Union operation :-

Union operation is used to add output of multiple relational algebra expressions.

example ⇒ Find the names of all bank customers who have either an account or a loan or both. To answer this query we need data from 'depositor' relation and 'borrower' relation.

1) Name of customers who have account in bank

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

2) Name of customers who have a loan in the bank

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

To answer the question, we need to perform union of these 2 expressions. The expression is given as:

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

Here, duplicate values are eliminated. This query does not contain details of customers who neither have account nor loan.

Note ⇒ Union is an binary operation. Before we perform union, we need to ensure that relations which are participating in union are compatible.

For a union operation $R \cup S$ to be valid, two conditions must hold:

1) R and S must have same number of attributes.
2) Domain of i^{th} attribute of R and i^{th} attribute of S must be same for all i.

5) Set Difference operation:

set difference operation, is denoted by −, allows us to find tuples that are in one relation but not in another. The expression R−S produces a relation containing those tuples in R but not in S.

Example→ Find all the customers who have an account but not a loan, this query is given as:

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

→ Like union operation, set difference is applied on relation which are compitible.
→ For a set difference operation R-S to be valid, R and S must have same no of attributes (same arity) and domains of ith attribute of R and ith attribute of S must be same.

6) Cartesian-Product operation:-

Cartesian-Product operation, denoted by (cross (X), allows us to combine information of any two relation (or relational algebra expression).

→ The Cartesian Product of relations r_1 and r_2 is written as: $r_1 \times r_2$.
→ It is possible that same attribute name may appear in both r_1 and r_2. We can distinguish such attributes by attaching its relation name with an attribute. For example, borrower and loan relation has one common attribute - loan number. Cartesian Product of this relations can be written as:

r = borrower X loan
= borrower.customer-name, borrower.loan-number, loan.loan-number, loan.branch-name, loan.amount

→ The attribute which are appearing only once, can be written without attaching table name to them.
So, r = borrower X loan
= customer-name, borrower.loan-number, loan.loan-number, branch-name, amount.

→ If we have n_1 - number of tuples in r_1 and n_2 - number of tuples in r_2, then cartesian product of r_1 and r_2 i.e. $r_1 \times r_2$ contains $n_1 * n_2$ - number of tuples.

→ Cartesian product associates every tuple from r_1 with every tuple of borrower. Thus, it shows all possibilities among two tables.

example ⇒ r = borrower x loan. is performed here:

customername	loan-number
Adams	L-16
Curry	L-93
Hayes	L-15

borrow table

loan-number	branch-name	amount
L-11	Karelibaug	900
L-14	O.P. road	1500
L-15	Gotri	1300

Loan table

r (borrower x loan)

customer-name	borrower. loan-number	loan. loannumber	branch-name	amount
Adams	L-16	L-11	Karelibaug	900
Adams	L-16	L-14	O.P. road	1500
Adams	L-16	L-15	Gotri	1300
Curry	L-93	L-11	Karelibaug	900
Curry	L-93	L-14	O.P. road	1500
Curry	L-93	L-15	Gotri	1300
Hayes	L-15	L-11	Karelibaug	900
Hayes	L-15	L-14	O.P. road	1500
Hayes	L-15	L-15	Gotri	1300

Now, Consider the question: Find names of all customers who have a loan at 'Gotri' branch.

→ To find out name of customers, we first need to find out that which customers are having loan actually. This answer is given by:

$\sigma_{\text{borrower.loan-number} = \text{loan.loan-number}}$ (borrower x loan)

→ Now, we will find the name of customers who have loan

at 'Gotri' branch.

$(\sigma_{branch\text{-}name = "Gotri"}(\sigma_{borrower.loan\text{-}no = loan.loan\text{-}no}(borrower \times loan)))$

But we want only customer name, Hence, final query is:

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

7) Rename Operation:

→ Unlike relation, relational algebra expressions do not have a name. This can be achieved by rename operator. It is denoted by Greek letter rho(ρ).

→ $\rho_x(E)$ - It returns the result of E with the name-x.

→ A relation r is itself a (trivial) relational algebra expression.

→ Another form of ~~rem~~ rename operation is:
$\rho_{x(A_1, A_2, \ldots A_n)}(E)$ returns the result of Expression E with name-x and with the attributes renamed to $A_1, A_2, \ldots A_n$.

→ To understand this operation, we consider the question:
Find largest account balance in the bank.

→ To solve this question, our strategy is
1) Compute first a temporary relation consisting of those balances that are not largest and 2) Take set difference between relation $\Pi_{balance}(account)$ and temporary relation computed in step-1, to get result.

Step-1: we have to compare each account balance with all other account balance values. It can be done by cross-product of (account X account) and comparing value of any two balances appearing in one resultant tuple. For that, we need to distinguish between 'balance' attribute of

both tables. We can rename the table and create its another reference, thus we can refer the relation twice without ambiguity.

query for step-1 is given as:

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

Here, we have created 2 instances of table account. one instance is account itself and another is d.

→ The above expression gives those balances in account relation for which a larger balance appears somewhere in account relation (renamed as d). The result contains all balances except largest one.

step-2: The query to find largest balance in the bank can be written as:

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

* Concept of Relational Algebra:

A basic expression in relational algebra consists of either one of following:

- A relation in database.
- A constant relation.

→ A Constant relation is, listing of tuples within {}.
example → {(A-101, Vadodara, 500), (A-215, Surat, 700)}.

→ A General expression is constructed from smaller subexpressions. If E_1 & E_2 are relational algebra expressions then $E_1 \cup E_2$; $E_1 - E_2$; $E_1 \times E_2$; $\sigma_p(E_1)$ where p is predicate or condition; $\Pi_s(E_1)$ where s is a list of attributes in E_1, and $\rho_x(E_1)$ are also relational algebra expressions.

Additional Relational Algebra operations:

If we restrict ourselves to just fundamental operations, certain common queries are lengthy to express. Additional operations do not add any power to algebra, but simplify common queries.

1) Set-Intersection operation:

Set-Intersection operation can be re-written in terms of fundamental set-difference operations. Intersection of R and s can be shown as:

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

example ⇒ Find all customers who have both a loan and an account. This query can be written as:

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

2) Natural Join operation:-

⇒ Natural join is a binary operation that combines certain selection operation and cartesian product into one operation. It is denoted using symbol '⋈'.

→ Find name of customers who have a loan at the bank. To answer this query we select relations borrower and loan & make cartesian product of them. Then we select tuples which have same loan-number in both relations, followed by projection of customer-name.

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

→ This query can be rewritten using natural join:

$$\Pi_{customer\_name}(borrower \bowtie join)$$

Join operation finds common ~~tuple~~ attribute in both

tables. Then it checks for same value on common attribute. It consider each such pair which follows condition and combine these pair of tuples into single tuple. Common attribute is considered only once in resultant relation. Natural join removes the duplicates from result.

→ If two relations have no attribute in common means $R \cap S = \phi$ then $R \bowtie S = R \times S$

→ We can also perform join of 3 tables.

Example ⇒ Find name of all branches with customers who have an account in bank and who live in surat.

$$\Pi_{branch\text{-}name}(\sigma_{customer\text{-}city = "surat"}(customer \bowtie account \bowtie depositor))$$

→ Directly joining of 3 tables is not possible. It is executed step by step. There are two possibilities

$(customer \bowtie account) \bowtie depositor$ **OR**

$customer \bowtie (account \bowtie depositor)$

⇒ Result of both ways are same. That is, natural join is associative.

→ Find all customers who have both loan and account at the bank. This query can be written as:

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

[Note: The question given above was solved using set intersection operation and that query was:

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

theta join: It is an extension to natural join. Theta join is given as:

$r \bowtie_\theta s = \sigma_\theta (r \times s)$ where θ is some condition or Predicate.

→ In theta join, Predicate θ may contain any relational operator like $=, !=, >, <, >=, <=$ to compare attributes of two relations.

3) Division Operation:

⇒ This operation is suited for queries that include the phrase "for all". Division operation is denoted by ÷.

→ Suppose we want to find all customers who have an account at all branches located in Vadodara.

→ This question will be divided into 2 parts:

Part-1 ⇒ All the branches located in Vadodara city

Query for this is given as:

$r_2 = \Pi_{branch\text{-}name}(\sigma_{branch\text{-}city = "vadodara"}(branch))$

Part-2 ⇒ All the customers who have account at a branch. Query for this part can be given as:

$r_1 = \Pi_{customername,\ branch\text{-}name}(depositor \bowtie account)$

Now, we want to find customers who appear in r_1 with every branch name in r_2. The operation that provides exactly those customers is the divide operation.

The Final query can be written as:

$r_1 \div r_2 = \Pi_{customer\text{-}name,\ branch\text{-}name}(depositor \bowtie account) \div \Pi_{branch\text{-}name}(\sigma_{branch\text{-}city = "Vadodara"}(branch))$

we take some tables as example and understand the working of division operation.

If, r_1 =

Customername	branchname
Simran	Satellite
Alka	O.P.road
Alka	Gotri
Mahesh	Bandra
Dhaval	Gotri
Kushal	Santacruz

and r_2 =

branch-name
Gotri
O.P.road

Now, result of division operation : $r_1 \div r_2$ is

customer-name
Alka

The output is customer-name: Alka because Alka has account at "all" the branches of city "Vadodara". No other person (customer) satisfy that condition.

4) Assignment Operation:

Sometimes it is convenient to assign parts of relational algebra expression to temporary relational variables. The assignment operation denoted by '←' works like assignment in a programming language.

→ Previous example is re-considered here to show use of assignment operation: Find all customers who have an account at all branches located in Vadodara.

temp1 ← $\Pi_{customer\text{-}name,\ branch\text{-}name}$ (depositor ⋈ account)

temp2 ← $\Pi_{branch\text{-}name}(\sigma_{branch\text{-}city = "Vadodara"}(branch))$

result = temp1 ÷ temp2

→ Assignment operation does not provide any output. It simply assigns result of a relational algebra expression to a relational variable. This variable can be used in further expressions.

→ By using assignment operation, a query can be written as a sequential program consisting of a series of assignments followed by an expression whose value is displayed as the result of the query.

→ Assignment must always be made to temporary relational variable. assignments to permanent relations leads to database modification.

→ This operation does not provide any additional power to algebra. It's just a convenient way to express complex queries.

Extended Relational Algebra Operations:

1) The Generalized Projection:

This operation extends the projection operation by allowing arithmetic functions to be used in the projection list. The generalized projection has form:

$$\Pi_{F_1, F_2, \ldots F_n}(E)$$

where E is a relational algebra expression and $F_1, F_2, \ldots F_n$ is an arithmetic expression involving constants and attributes of E.

→ Sometimes, arithmetic expression is simply an attribute or a constant.

Example: We want to show 15% increase in salary of employee. Query for this question is given as:

$$\Pi_{\text{customer-name, salary} + 0.15 * \text{salary}}(\text{employee})$$

The attribute resulting from 'salary + 0.15 * salary' does not have any name. We can apply rename operation to the result of generalized projection in order to give it a name.

$\Pi_{customername, (Salary + 0.15 * Salary)\ as\ Salary\_hike}$ (employee)

The 2nd attribute of generalized projection has been given a new name 'salary-hike'.

2) Aggregate functions:

→ Aggregate functions take collection of values and return a single value as a result. For example, aggregate function sum takes collection of values and returns sum of values. Other aggregate functions are sum, avg, count, min, max. Count returns no. of elements in the collection (no. of values in table)

→ for aggregate functions, symbol 'G' is used, which is called as 'caligraphic G'.

Example:- Find out total salary of all employees. We can write expression for this using G:

$\mathcal{G}_{sum(salary)}$ (employee)

The result of above query is a relation (table) with a single attribute, containing a single row with numerical value which is total of salary of all employees.

If the result contains multiple occurrences of a value or duplicate values then it can be eliminated by word 'distinct'. Consider a query to understand use of word 'distinct' which removes duplicate tuples:

Find number of branches appear in branch table.

Simple query can be written as:

$\mathcal{G}_{count(branch\text{-}name)}$ (branch)

It is possible that some branch names are repeated in the table. To retain only unique tuples, query is written as:

$\mathcal{G}_{count\text{-}distinct(branch\text{-}name)}$ (branch)

Now, suppose we want to find the total salary of all employees at each branch of bank separately, instead of sum of salary for entire Bank. To do so, we need to divide table employees into groups based on branch, and to apply aggregate function on each group. Following expression proides that query:

$$branch\text{-}name\,\mathcal{G}\,sum(salary)\,(employee)$$

The parameter on lefthand side of $\mathcal{G}$ shows that input relation is grouped based on value of branch-name. The expression sum(salary) on right-hand subscript of $\mathcal{G}$ indicates that for each group of tuples (means for each branch), the aggregation function sum must be applied on collection of values of salary attribute. The output relation consists of tuples with branch name, and sum of salaries for each branch.

3) Outer Join:

It is an extension of join operation to deal with missing information. To understand it, consider an example.

Consider employee and work-info relations: employee(ename, street, city) and work_info(ename branch-name, salary). Find out all the details of each employee. To get the result, one approach is to use natural join operation as follows: employee ⋈ work-info

employee

ename	street	city
Suman	Manek	ahmedabad
Ganesh	Fun street	Vadodara
Ramesh	Dandi	Surat

work-info

ename	branch-name	Salary
Ramesh	Dumas	12000
Suman	Vastrapur	14000
Shivam	Race course	10000

employee ⋈ work-info

ename	street	city	branch-name	salary
Suman	Manek	Ahmd	Vastrapur	14000
Ramesh	Dandi	Surat	Dumas	12000

The tuple describing Ganesh is not available in the result because that tuple is absent in work-info table. Similarly, tuple describing shivam of work-info is not available in the result because that tuple is absent in employee relation.

To overcome this issue we can use outer join operation. There are 3 types of outer join 1) Left outer join 2) Right outer join 3) full outer join.

i) Left outer join (⟕): It considers all tuples that matched condition as well as all other tuples of left relation that did not match in right relation and pads the tuples with null values for all attributes from right relation. It is denoted by ⟕

Example: We will apply left outer join on employee and work-info: employee ⟕ work-info

ename	street	city	branch-name	salary
suman	manek	Ahmedabad	vastrapur	14000
Ramesh	Dandi	surat	Dumas	12000
Ganesh	fun street	Vadodara	NULL	NULL

Here, all information of left table is available in the result of left outer join.

ii) Right outer join: It considers all tuples that matched condition as well as all other tuples of right relation that did not match left relation and pads tuples with NULL values for all attributes from left table. Denoted by: ⟖

Example:- We will apply Right outer join on employee and work-info : employee ⟖ work-info.

ename	street	city	branch-name	salary
Suman	Manek	Ahmedabad	Vastrapur	14000
Ramesh	Dandi	Surat	Dumas	12000
Shivam	NULL	NULL	race course	10000

Thus, all information of right relation is present in the result of right outer join.

(iii) Full outer join: It performs union of left and right outer join, Padding tuples from left relation that did not match any from right relation, as well as tuples from right relation that did not match any from the left relation. It is denoted by ⟗.

Example: We will apply full outer join on employee and work-info.

employee ⟗ work-info

ename	street	city	branch-name	salary
Suman	Manek	Ahmedabad	Vastrapur	14000
Ramesh	Dandi	surat	Dumas	12000
Ganesh	funstreet	Vadodara	NULL	NULL
Shivam	NULL	NULL	race course	10000

Here, all information of both tables is available in the result of full outer join.

* NULL Values:

Whenever we deal with null values, sometimes results are arbitrary. Hence, operations and comparisons on null values should be avoided, whenever possible.

→ null means unknown or nonexistent.
→ Any arithmetic operations (such as +, −, *, /) involving null values must return a null result.
→ Any Relational operations (such as <, <=, >, >=, =, !=) involving null value, returns the result unknown, because we can not say that result of comparison is True or false, thus we say result is unknown i.e. null.

→ For Boolean operations: (and, or, not)

1) And:- true and unknown = unknown
False and unknown = false, unknown and unknown = Unknown

2) or: True or unknown = True
false or unknown = unknown
unknown or unknown = unknown

3) Not: (not unknown) = unknown

→ For different relational operations:

I) select (σ): In select operation given as $\sigma_p(E)$ p is predicate or condition on relational algebra expression E for each tuple t. If p is true then tuple t is added to result, if p is unknown (null) or false, t is not added to result.

2) join: In a natural join of R and S means R⋈S, if two tuples $t_r \in R$ and $t_s \in S$ have null values in their common attribute then tuple do not match.

3) Project (Π): Project operation treats nulls just like other values when eliminating duplicates. Thus, if two tuples in the project operation result are exactly same and both have null values in the same fields, they are treated as duplicates.

4) Union, Intersection, difference:
These operations treat nulls just as the project operation does, they treat tuples that have same values on all fields as duplicates even if some of the fields have null values in both tuples.

5) Generalized projection: Arithmetic operations on constants and attributes involving null values return null value. Duplicate tuples containing null values are handled same as the project operation.

6) Aggregate: when nulls occur in grouping attributes, the aggregate operation treats them just as in Projection: If two tuples are same on all grouping attributes, the operation places them in the same group, even if some of their attribute values are null.

When nulls occur in aggregated attributes (means attributes on which aggregation is applied), the operation deletes null values, before applying aggregation.

7) Outer join: Outer join considers tuples that follow the condition as well as unmatched tuples of left hand table or Right hand table or both tables depending on the type of outer join. Thus, unmatched tuples, are added to the result, and for non existent values NULLs are Padded.

★ Modification of Database:-

Insert, delete and update operations are used to add, remove or change information in database.

1) Deletion:-

→ We can delete only entire tuple or tuples; we can not delete values of only particular attributes of any tuple.

→ Delete operation is expressed by:

$$r \leftarrow r - E$$

where r is a relation and E is a relational algebra query. Let us consider some example.

→ Delete all details of Smith's account.

$$\text{depositor} \leftarrow \text{depositor} - \sigma_{\text{customer-name} = \text{"Smith"}}(\text{depositor})$$

→ Delete all loans where amount is in range 0 to 50.

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

→ Delete all accounts at branches located in Surat city.

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

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

$account \leftarrow account - r_2$

→ Here, we are using temporary relations r_1 & r_2 to get the result.

2) Insertion:-

→ To insert data into a relation, we either specify a tuple to be inserted or a relational algebra expression whose result is a set of tuples to be inserted.

→ Attribute values for inserted tuples must be members of attribute's domain.

→ Tuples inserted must be of correct arity (number of attributes values must be same as number of attributes in the relation).

→ Insertion is given by: $r \leftarrow r \cup E$

→ We can insert a single tuple by taking E as a constant relation containing one tuple. For example, Suman has 1200 Rs. in account A-973 at karelibaug branch, written as: $account \leftarrow account \cup \{(A\text{-}973, \text{"karelibaug"}, 1200)\}$

$depositor \leftarrow depositor \cup \{(\text{"Suman"}, A\text{-}973)\}$

3) UPdation:-

→ Sometimes, we wish to change a value in the tuple without changing all values of tuple. To update we can use generalized Projection operator:

$$r \leftarrow \Pi_{F_1, F_2, \ldots F_n}(r)$$

where, $F_i = \begin{cases} i^{th} \text{ attribute of } r, \text{ if } i^{th} \text{ attribute is not to be updated} \\ \text{an expression involving only constants and attributes of } r, \text{ that gives new value to attribute} \end{cases}$

→ For example, interest of 5% is given to all account holders. i.e. balance of all accounts is to be increased by 5%.

Query will be written as:

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

⇒ Now, suppose accounts with balances above 10,000 Rs receive 6% interest, whereas all other receive 5% interest. Query is given by:

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

* Integrity Constraints:-

1) Domain Constraints:-

→ A domain of possible values must be associated with every value. For example, integer, character, date, number etc.

→ Declaring a domain for an attribute acts as a constraint on the value that it can take.

→ They are tested easily whenever a new data item is entered into the database.

2) Referential Integrity:-

→ There are cases where we wish to ensure that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.

→ Database modification can violet referential Integrity.

→ In case of violation, the normal procedure is to reject the action that caused the violation.

3) Assertions:-

→ Assertion is a condition that database must always satisfy.

→ Domain constraints and referential integrity are special forms of assertions.

→ While performing any modification in database, first assertions are checked. If they are fulfilled then only the modification to the database will be allowed.

Example: Every loan has atleast one borrower who maintains an account with a minimum balance of Rs. 1000.

4) Authorization:

→ Authorization will differentiate users and types of access they are permitted on.

example ⇒ - Read authorization, Insert authorization, update authorization, Delete authorization.

→ DDL is just like other programming languages, it gets input instructions and generate some output

→ Output of DDL is placed in data dictionary which contains Metadata (data about data).

Data Dictionary Contains:

Table name:

Column name	Type	Size