**Chapter 03: Relational Model and Algebra**

**Record Based Logical Model:** These are used in describing data at: **i.** Logical Data **ii.** View Level

In this model, data is structured in fixed format record of several types. Each type defines fixed a number of fields or attributes and each field is usually of fixed length. Following are the types of Record based Logical Model

1. **Relational Model**: It uses a collection of tables to represent both data and relationship among those data. Each table has multiple columns and each column has a unique name.
2. **Network Model**: In this, the data is represented as collection of records and relationship among data are represented by links, which can be viewed as pointers. The records in database are organized as collection of **arbitrary graphs**.
3. **Hierarchical Model**: It is similar to network model. In this is model the records in the database are organized as collections of **trees** rather than arbitrary graphs.

In the Relational model it uses a collection of tables to represent both data and relationship among those data.

Each table has multiple column, each column represents an attribute and each row represents the relationship among the set of values (data).

**Concepts:**

**A relational database is a collection of relation.**

**A relation resembles a “Table” or a “Flatfile” of records.**

**Each row in such a table represents a collection of such related data values**

**e.g Student**

| **RollNo** | **Name** | **Date\_of\_Registration** |
| --- | --- | --- |
| 01 | ABC | 10-02-2013 |
| 03 | XYZ | 10-02-2014 |
| 02 | SSSS | 10-02-2015 |

**Each row is called as *tuple* and each column is called as *attribute* of a relation.**

**Attribute :** Attribute is the name of a column in a table. The attributes are rollno,name etc. The no. of attributes of a relation is called as **arity (or degree or order of relation)**.

**No two attributes have a same name.**

**Domain** : Domain (D) is the set of values of the same data type. Domain of the attribute is defined as set of allowable values for the attribute.

Domain is a set having “homogeneous” members and it is conceptually similar to the data type concept in programming language.

According to the set of values, domain is classified into four categories

**1. Simple 2. Application Independent**

**3. Application dependent 4. Composite.**

A domain (D) is said to be simple if all its elements are non-decomposable.

Application Independent or Atomic domain is the general set of integers, real numbers or character strings.

Application dependent domain is having the values permitted in the database.

Composite can be specified as a set consisting of non-atomic values.

**Relation:** Relations is a collection of homogeneous tuples.

A relation with n attributes is a subset of Cartesian product of domains of those attributes.

e.g. Domain of RollNo(01,02,03)

Domain of Name(ABC,XYZ,SS)

Relation Student is subset of Cartesian product of domains of all these attributes.

Properties of Relation:

1. The relation has a name that is distinct from all other relation names
2. Each cell of the relation contain exactly one atomic(not divided) single value.
3. Each attribute has a distinct name
4. The values of an attribute are no duplicate tuples.
5. The order of attributes has no significance
6. The order of tuples has no significance

**Intension and Extension:**

The structure of relation together with a specification of the domains and any other restrictions on possible values, is called its **intension**, which is usually fixed unless the meaning of the relation is changed to include additional attributes.

The tuples are called the extension of the relation which changes over time.

**Degree of relation:** This is defined as no. of attributes it contains.

**Cardinality:** The cardinality of relation is the number of tuples it contains which is also extensions of the relation , which is determine at a particular instance.

**Relation Schema:** It defines the set of attributes of that relation. It is Denoted by R(A1,A2…An) is made up of relation name R and a list of attributes A1,A2,…An. Each A1 is the name played in some domain D in schema R. The domain A1 is denoted by dom(A1)

**Relational Database:** A collection of normalized relations with distinct relation names is called as relational database.

**Integrity Rules:** Integrity rules are the constraints or restrictions that apply to all instances of the database. There are 3 integrity rules

1. **Entity Integrity Rule** : In the base relation, no attribute of a primary key can be NULL
2. **Referential Integrity Rule** : If a foreign key exists in a relation, either the foreign key value must match a candidate key value of some tuple in its home relation or the foreign key value must be wholly NULL.

**Employee**



| EMP\_ID | Name | Works\_In | Reports\_to |
| --- | --- | --- | --- |





| Dept\_id | Location\_ID | Manage\_By | Members |
| --- | --- | --- | --- |

1. **Semantic Integrity Constraints**: Constraints on the values of attributes of attribute. e.g. The salary of employee will not more than his/her manager.

**Mapping ER-EER Model to Relational Model**

1. **Simple Attribute**

****

**Relation : *Department(Dept\_ID,Dept\_name,Manager)***

Strong entity type with simple attribute can be mapped into relations in a straightforward manner. Attributes acting as keys in ER model are retained as keys in the relational model.

1. **Composite Attribute**

****

**Relation : *Department(Loc\_Id,RGN\_No,Dept\_name,Manager)***

***Mapping of Relationship:***

****

**Relations : Insurance\_Record(PAN,………)**

The employee is associated with 1 insurance and vice versa. The Insurance\_Record is weak entity set. If the relation is with weak entity set then the primary key is of weak entity is primary key of the strong entity set. These key attributes are also form a foreign key into the owner entity type. The CASCADE option is used whenever the owner entity type is updated or deleted.

**Mapping 1:1 Relationship**

****

**Relation** : Manager(emp\_no,Dept\_id,Secretary)

In 1:1 binary relationship between two entity type S and T, chose one of them (Say S) as base relation

If either S or T has a total participation, chose that one as the base.

Include the primary key of the other entity type as a foreign key of the base.

Include any relationship attributes as attributes of the base.

**Mapping 1:N Relationship**

****

**Relation : Employee(EMP\_NO,Dept\_ID,……..) the dept\_id will become foreign key**

**Mapping 1:N Relationship**

For each binary 1:N relationship, identify the relation S that represents the entity type on the N side.

Use this as the base and create a relation including the key of the other entity type as foreign key

**Mapping M:N relationship :**

**In M:N relationship the identifying the record is difficult. E.g.**

****

**In the above example the Many employees are deputed to many other departments.**

**Employee(emp\_no,…..)**

**Department(dept\_id,…)**

**Deputed\_to(emp\_no,dept\_id,record\_no) foreign key (emp\_no,dept\_id)**

**In M:N Relationship:** In M:N relationship, it is not possible to collapse the relationship into any one of the entity type, since the relationship does not identify either entity type uniquely.

**A separate relation is required to complete the mapping.**

The CASCADE option on referential integrity should be used for the relation created during Update and Delete. This is because, each instance of the relationship has an existential dependency on the created relation.

**Mapping Multivalued Attributes:**

****

**Bird(Species)**

**Birdcolor(Species,color) FK Species from Bird table**

For each multi valued attributes of a given entity type S, create a separate relation that has the primary key of S, paired with all values of multi-valued attributes.

The CASCADE option should be used for referential integrity on the newly created relation.

**Query Languages:** Query language is a language in which user requests information from the database.

These languages are of higher level than standard programming language. Query language is classified as

| Query Language | |
| --- | --- |
| Procedural Language | Non-procedural Language |
|  |  |
| Relational Algebra | Tuple Relational Calculus |
|  | Domain Relational Calculus |

**Procedural Language:** In these language the user instructs the system to perform a sequence of operations on the database to compute the desired result. i.e. how to retrieve the data. Relational algebra is procedural language.

**Non-Procedural Language:** In these language the user describes the information desired without giving a specific procedure for obtaining that information. Tuple and Domain relational calculus are the non-procedural languages.

**Relational Algebra:** It is procedural language. It specifies the operations to be performed on existing relations to derive the result relation.

**Operations which take only one relation as input are unary operations.**

**Operations which take only two relation as input are Binary operations.**

| **Relational Algebra** | | |
| --- | --- | --- |
| **Basic Operations** | | **Additional Operations** |
|  |  |  |
| **Unary** | **Binary** |  |
|  |  |  |
| **Select** | **Cartesian Product** | **Intersection** |
| **Project** | **Union** | **Join** |
| **Rename** | **Set Difference** | **Division** |
|  |  | **Assignment** |

**Consider Player Relation**

| **Player\_id** | **Team\_id** | **Country** | **Name** | **Age** | **Runs** | **Wickets** |
| --- | --- | --- | --- | --- | --- | --- |
| 1001 | 101 | India | ABC | 25 | 300 | 40 |
| 1002 | 102 | Pak | Riyaz | 30 | 500 | 10 |
| 1003 | 101 | India | ABZ | 23 | 500 | 1 |
| 1004 | 103 | Aus | Garry | 34 | 800 | 0 |

**Select (**σ) : It will Works on a single relation R and defines a relation that contains only those tuples (rows) of R that satisfy the specified condition (predicate). It selects all the columns from the relation. It includes the duplicate values also. We can combine the conditions by ^ (and) ˬ(or). We can use the conditions such as **>,<,=,!=,>=,<= <>** etc.

**Syntax: σ column\_name condition if any (relation name)**

**e.g.**

List all staff with a salary greater than £10,000.

**σ salary > 10000 (Staff)**

| **Stno** | **Stname** | **Job** | **Salary** |
| --- | --- | --- | --- |
| 1 | ABC | Manager | 20000 |
| 2 | XYZ | Analyst | 30000 |
| 3 | PQR | administrator | 40000 |

E.g. From Player Relation Find all tuples for which country is India

**σ Country = “India” (Player)**

**o/p**

| **Player\_id** | **Team\_id** | **Country** | **Name** | **Age** | **Runs** | **Wickets** |
| --- | --- | --- | --- | --- | --- | --- |
| 1001 | 101 | India | ABC | 25 | 300 | 40 |
| 1003 | 101 | India | ABZ | 23 | 500 | 1 |

E.g. From Player Relation Find all tuples for which country is India and runs score >=500

**σ Country = “India” ^ runs>=500 (Player)**

**O/p**

| **Player\_id** | **Team\_id** | **Country** | **Name** | **Age** | **Runs** | **Wickets** |
| --- | --- | --- | --- | --- | --- | --- |
| 1003 | 101 | India | ABZ | 23 | 500 | 1 |

**Project /Projection** (π):It is used to project the distinct values from the relations. It can project the mentioned attributes. It will project all the attributes from the relation. It will removes the duplicate values. It reduces the arity of the relation.

e.g. List all the countries in player relation

π county(Player)

**o/p**

| **Country** |
| --- |
| India |
| Pak |
| Aus |

e.g. List all the teamid,countries in player relation

π team\_id,county(Player)

**o/p**

| **Team\_id** | **Country** |
| --- | --- |
| 101 | India |
| 102 | Pak |
| 103 | Aus |

**Compatible Relation:** Two relations R & S are said to be compatible relations if they satisfy following two conditions:

1. The relations R & S are of same arity. i.e. the no. of attributes are same.
2. The domain of ith attribute of R and ith attribute of S must be same for all i.

**Union Operation(U) :**If R & S are compatible relations, then union of R & S is the set theoretic union of R & S. The resultant relation P=R U S has tuples drawn from R & S such that, a tuple in P is either in R or S or in both of them. Union eliminates the all the duplicates. Union operation does not change the arity of resultant relation but change the cardinality

e.g. Consider R & S

| **R** | |  | **S** | |
| --- | --- | --- | --- | --- |
| **Id** | **Name** |  | **ID** | **Name** |
| 1 | Raj |  | 1 | Raj |
| 2 | Rahul |  | 4 | Anil |
| 3 | Sachin |  | 6 | Kapil |
| 4 | Anil |  | 7 | Sumit |
| 5 | Prasad |  |  |  |

**P=(R U S)**

| **P** | |
| --- | --- |
| **ID** | **Name** |
| 1 | Raj |
| 2 | Rahul |
| 3 | Sachin |
| 4 | Anil |
| 5 | Prasad |
| 6 | Kapil |
| 7 | Sumit |

**e.g**

| **Depositor** | |  | **Borrower** | |
| --- | --- | --- | --- | --- |
| **Acc\_No** | **Name** |  | **Loan\_No** | **Name** |
| A-231 | Rahul |  | P-3261 | Sachin |
| A-432 | Omkar |  | Q-6934 | Raj |
| R-321 | Sachin |  | S-4321 | Ramesh |
| S-231 | Raj |  | T-6281 | Anil |
| T-239 | Sumit |  |  |  |

**Find the name of customer having an account/loan**

**π name(Depositor) U π Name(Borrower)**

o/p

| **Name** |
| --- |
| Rahul |
| Omkar |
| Raj |
| Sumit |
| Sachin |
| Raj |
| Ramesh |
| Anil |

**Set Difference Operation (-)** : The difference operations removes common tuples from the first relation. It is denoted by (-). The expression R-S results in a relation containing those tuples in R but not is S. For set difference operation relations must be compatible. Set difference operation does not change the arity of resultant relation but change the cardinality.

**e**.g.

| **R** | |  | **S** | |
| --- | --- | --- | --- | --- |
| **Id** | **Name** |  | **ID** | **Name** |
| 1 | Raj |  | 1 | Raj |
| 2 | Rahul |  | 4 | Anil |
| 3 | Sachin |  | 6 | Kapil |
| 4 | Anil |  | 7 | Sumit |
| 5 | Prasad |  |  |  |

**P=R-S**

| **P** | |
| --- | --- |
| **Id** | **Name** |
| 2 | Rahul |
| 3 | Sachin |
| 5 | Prasad |

**From Depositor and Borrower relation find the name of customers having an account but not loan**

**π name(Depositor) - π Name(Borrower)**

| **Name** |
| --- |
| Rahul |
| Omkar |
| Sumit |

**From Depositor and Borrower relation find the name of customers having an loan but not account**

**π name(Borrower) - π Name(Depositor)**

| **Name** |
| --- |
| Ramesh |
| Anil |

**Cartesian Product Operations:** Cartesian product of two relations is the concatenation of tuples belonging to the two relations. It is denoted by **X.** If R & S are two relations then P=R X S, which contains all possible combinations of tuples in R & S. For Cartesian product does not required compatible relations.

e.g.

| **Employee** | |  | **Project** |
| --- | --- | --- | --- |
| **Emp\_id** | **Name** |  | **Project\_Id** |
| 101 | Sachin |  | DBMS1 |
| 103 | Rahul |  | DBMS2 |
| 104 | Omkar |

**R= Employee x Project**

| **Emp\_ID** | **Name** | **Project\_ID** |
| --- | --- | --- |
| 101 | Sachin | DBMS1 |
| 101 | Sachin | DBMS2 |
| 103 | Rahul | DBMS1 |
| 103 | Rahul | DBMS2 |
| 104 | Omkar | DBMS1 |
| 104 | Omkar | DBMS2 |

**Rename Operation :** used to rename. It is denoted as **ρ (rho)**.

Syntax : **ρ(New Relationname) (Existing Relation Name)**

**Set Intersection Operation (∩) :** It selects common tuples from two relations. The relations must be compatible. It does not change the arity of the resultant relations but it may change the cardinality. The result can be obtained as R **∩** S =R-(R-S)

E.g. P=R**∩** S

| **ID** | **Name** |
| --- | --- |
| 101 | Raj |
| 104 | Anil |

E.g. Find the name of customer having loan and account from borrower and depositor relation.

**π name(Borrower) ∩ π Name(Depositor)**

| **Name** |
| --- |
| Sachin |
| Raj |

**Division Operation(÷) :** Let R and S be relations and let S ⊆ R. i.e. any attribute of S is also in R. The relation R **÷** S is a relation on R-S i.e. on the schema containing all the attributes of schema R that are not in schema S.

e.g. 1.

| **P** | |  | **Q** | |
| --- | --- | --- | --- | --- |
| **AA** | **AA** |  | | **BB** |
| A1 | B1 |  | | B1 |
| A1 | B1 |  | | B2 |
| A2 | B1 |  | |  |
| A3 | B1 |  | |  |
| A4 | B2 |  | |  |
| A5 | B1 |  | |  |
| A5 | B2 |  | |  |

R=P **÷** Q

| **AA** |
| --- |
| A1 |
| A5 |

Consider :

| **Q** |
| --- |
| **BB** |
| B1 |

R=P **÷** Q

| **AA** |
| --- |
| A1 |
| A2 |
| A3 |
| A5 |

Consider :

| **Q** |
| --- |
| **BB** |
| B1 |
| B2 |
| B3 |

R=P **÷** Q

| **AA** |
| --- |
|  |

**Assignment Operation(🡨) :** It works similar to assignment in a programming language.

e.g. R1🡨 **π name(Borrower)**

R2🡨 **π name(Depositor)**

R🡨 R1-R2

**The natural join operation(): -** it is a binary operation and a combination of certain selections and a Cartesian product into one operation. It is denoted as ****. It is associative. It forms a Cartesian product of its two arguments. Then performs a selection forcing equality on those attributes those appear in both the relations. And finally removes duplicates attributes.

r(R): r is a relation with attributes R.  
s(S): s is a relation with attributes S.

R****S= **π R\_Schema U S\_Schema (**σR.A1=S.A1 ^ R.A2=S.A2 ^R.A3=S.A3….R.An=S.An)(RxS)  
If R **∩** S = Ф i.e. they have no attributes in common then **r  s = r X s**Example:-

| Employee | |  | Salary | |
| --- | --- | --- | --- | --- |
| ID | Name |  | ID | Salary |
| 101 | Sachin |  | 101 | 65000 |
| 103 | Rahul |  | 103 | 35000 |
| 104 | Kapil |  | 104 | 22000 |
| 107 | Ajay |  | 107 | 21910 |

Employee **** Salary

| ID | Name | Salary |
| --- | --- | --- |
| 101 | Sachin | 65000 |
| 103 | Rahul | 35000 |
| 104 | Kapil | 22000 |
| 107 | Ajay | 21910 |

**Group By Functions in Relational Algebra :**

**Sum:** It will take the sum of the attribute.

e.g. sum salary(employee)

**Average:** It will take the avg of the attribute.

e.g. Avg salary(employee)

**Count** : Counts the no. of records

e.g. Count branch\_name( employees)

Min and Max : min salary(employee), max salary (employee)

**Examples:**

| **Employees** | |  | **Assigned\_to** | |  | **Project** | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Emp\_ID** | **Name** |  | **Emp\_Id** | **Project\_ID** |  | **Project\_ID** | **Proj\_Name** | **Chief\_Arch** |
| 101 | Rahul |  | 101 | C-453 |  | C-231 | Oracle | 107 |
| 103 | Sachin |  | 103 | C-354 |  | C-278 | Java | 110 |
| 104 | Ashish |  | 104 | C-343 |  | C-353 | OS | 107 |
| 106 | Sumit |  | 104 | C-354 |  | C-354 | C | 104 |
| 107 | Anil |  | 106 | C-231 |  | C-453 | VB | 106 |
| 110 | Kapil |  | 106 | C-278 |  |
| 112 | Omkar |  | 106 | C-353 |  |
|  |  |  | 106 | C-354 |  |
|  |  |  | 106 | C-453 |  |
|  |  |  | 107 | C-231 |  |
|  |  |  | 107 | C-353 |  |
|  |  |  | 110 | C-278 |  |
|  |  |  | 112 | C-353 |  |
|  |  |  | 112 | C-354 |  |

**Q. 1 Get Emp\_Id of employees working on project C-353**

🡪 1. Find all the tuples of relation assigned\_to such that the value of project\_id=”C-353”

σ project\_id=C-353(Assigned\_to)

Project Emp\_Id from this relation

**π emp\_id**(σ project\_id=C-353(Assigned\_to))

| **Emp\_ID** |
| --- |
| 106 |
| 107 |
| 112 |

**Q. 2 Get details of employee number and name working on project C-353**

1. Find all the tuples of relation assigned\_to such that the value of project\_id=”C-353”

σ project\_id=C-353(Assigned\_to)

Project Emp\_Id from this relation

**π emp\_id**(σ project\_id=C-353(Assigned\_to))

To find names corresponding to emp\_id in result relation, use natural join with employee relation.

Employee **** **π emp\_id**(σ project\_id=C-353(Assigned\_to))

| **Emp\_ID** | **Name** |
| --- | --- |
| 106 | Sumit |
| 107 | Anil |
| 112 | Omkar |
|  |  |

1. **Obtain details of employees working on OS project.**

* Find Project\_Id from Project

σ proj\_name=’OS’(Project)

Project Project\_id from above result

**π Project\_id** (σ proj\_name=’OS’(Project))

Find emp\_id of employees working on that project from assigned\_to

Assigned\_to **** (**π Project\_id** (σ proj\_name=’OS’(Project)))

Project emp\_id from above result

**π Emp\_id**((Assigned\_to **** (**π Project\_id (**σ proj\_name=’OS’(Project)))

Now Find the Name of employee with natural join with employee relation.

**Employee  (π Emp\_id((Assigned\_to  (π Project\_id (σproj\_name=’OS’(Project))))**

1. **Gather details of employees working on both C-353 and C-354.**

🡪 Find Project\_id of C-353 and c-354

Q1🡨 **π Project\_id** (σ Project\_id=’C-353’ ˬ Project\_id=’C-354’(Assigned\_to))

Find emp\_id of employees working on both projects from Assigned\_to using **÷** operation

Q2🡨 Assigned\_to **÷** Q1

Find the details of employees working on both projects using employee relation and natural join

**Employee  (Assigned\_to ÷ (π Project\_id** (σ Project\_id=’C-353’ ^ Project\_id=’C-354’(Assigned\_to))))

**Q.5 Find the employees numbers of employees who work on at least all of the projects that employee 107 works on.**

**π Project\_id** (σ Emp\_id=107(Assigned\_to))

Find the employees working on all the projects that 107 works on using Assigned\_to relation and **÷** operation.

Assigned\_to **÷** (**π Project\_id** (σ Emp\_id=107(Assigned\_to))

The above results 107 also to remove this we have to set difference from it.

**(Assigned\_to ÷ (π Project\_id (σ Emp\_id=107(Assigned\_to)))-107**

**Q.6 Find the employees numbers of employees who do not work on C-453.**

**π emp\_id** (σ project\_id=’C-453’(Assigned\_to))

Subtract this result from assigned\_to relation

**π emp\_id (Assigned\_to)- π emp\_id (σ project\_id=’C-453’(Assigned\_to))**

**Q. 7 Get the emp\_id who work on all the projects.**

**π Project\_id (Project)**

Find all the employees working on all the project using ÷ operation

Assigned\_to ÷ **π Project\_id (Project)**

**Q. 8 List the employees who works on at least one project that employee 107 works on.**

**(π Project\_id (σ Emp\_id=107(Assigned\_to))**

**π emp\_id (Assigned\_to π Project\_id (σ Emp\_id=107(Assigned\_to)))-107**

**Consider Following Relation and Solve the queries by relational algebra.**

**Student(SSN,Name,Subject,DOB)**

**Course(Course\_Id,Name,Dept)**

**Enroll(SSN,Course\_ID,Sem,Grade)**

**Books\_Issued(Course\_Id,Sem,ISBN)**

**Text(ISBN,Title,Pbulisher,Author)**

1. **Find All course in the institute.**

**π Course\_id,Name,Dept (Course)**

1. **Find all student details registered for course\_id=10**

**σ Course\_id=10(Course)**

**π Course\_id** (**σ Course\_id=10(Course))**

**Enroll  (π Course\_id** (**σ Course\_id=10(Course))**

**π SSN (Enroll  (π Course\_id** (**σ Course\_id=10(Course)))**

**Student (π SSN (Enroll  (π Course\_id** (**σ Course\_id=10(Course))))**

**π SSN,Name,Subject,DOB**(**Student** **(π SSN (Enroll**  **(π Course\_id** (**σ Course\_id=10(Course))))**

1. **Find various books titles and authors for sem higher than 3**

**π Title,Author,Publisher**(**Text**  (**π ISBN(σ sem>3(Book\_Issued))))**

1. **Find all the students belongs to Comp Department**

**(Student**  **(π SSN(** Enroll (**π Course\_id( σ Dept=’Comp’(Course))))))**

1. Find the no. of students from Comp Dept.

**G count(SSN)(** Enroll (**π Course\_id( σ Dept=’Comp’(Course))))**

G is used for grouping.

**Relational Calculus:** It specifies what is to be retrieved rather than how to retrieve. It will not describe of how to evaluate the query. In first-order logic (or predicate calculus), predicate is a truth-valued function with arguments.

When we substitute values for arguments, functions yields an expression, called **proposition**, which can be either **true or false**.

If a predicate contains a variable (e.g. *x* is member of staff), there must be a range for *x*. This variable ranges over each tuple in a relation. When we substitute some values of this range for *x,* propagation may be true; for other values, it may be false.

When we applied Relational Calculus to database it is divided into **Tuple Relational Calculus and Domain Relational Calculus.**

**Tuple Relational Calculus:** It is a non-procedural language. In this we are interested in funding tuples for which predicate is true based on use of tuple variables. Tuple variable is a variable that ranges over a named relation: i.e. variable whose only permitted values are tuples of the relation.

The query is expressed as

**{t | P(t) }** i.e. it is a set of all tuples t such that predicate P is true for *t.* We use t[A] to denote the value of tuple t on attribute A , and we use *t ϵ r* to denote that tuple *t* is in relation *r.*

***To specify range of a tuple variable S as the staff relation using the notation: Staff(S)***

***To find set of all tuples S such that P(S) is true {S | P(S) }***

The tuple relational calculus expressions is of the form **{t | P(t) }** where p is the formula. Several tuple variables may be appear in the formula. A tuple relational calculus formula is built-up out of atoms. An atom has one of the following forms:

1. S ϵ R, where S is tuple variable and R is a relation.
2. s[x] Ѳ u[y], where s and u are tuple variables, x is an attribute on which s is defined, y is an attribute on which u is defined, and Ѳ is a comparison operator(<,><=,>= ,≠); we require that attributes x and y have domains whose members can be compared by Ѳ.
3. S[x] Ѳ c, where s is tuple variable, x is an attribute on which s is defined, Ѳ is comparison operator and c is a constant in the domain of attribute x.
4. Implication (⇒): x ⇒ y, if x if true, then y is true

*x* ⇒ *y* ≡ ¬*x* v *y*

1. Set of quantifiers:

∃ *t* ∈ *r* (*Q* (*t* ))≡”there exists” a tuple in *t* in relation *r*  
 such that predicate *Q* (*t* ) is true

∀*t* ∈ *r* (*Q* (*t* )) ≡ *Q* is true “for all” tuples *t* in relation *r*

**Consider the relation staff (staff\_id,name,salary)**

To find the details of all staff earning more than 10000

{ S | Staff(S) ^ S.salary>10000}

To find the particular attribute such as salary

{ S.Salary| Staff(S) ^ S.Salary>10000}

*Tuple relational calculus can use two quantifiers to tell how many instances the predicate applies to Existential quantifier* (∃) *“There exists” Universal quantifier* (∀) *“For All”*

*Tuple variables quantified by* ∃ or ∀ *are called bound variables otherwise called as Free variables. Only variables on the left hand side of | are free variables.*

***Consider Relation staff(staff\_no,Fname,Lname,Salary,BranchNo)***

***Branch(BranchNo,City)***

***List all names in staff such that the staff member works in London branch.***

*{S.Fname,S.Lname | Staff (S) ^* (∃B)(Branch(B) ^ (B.BranchNo=S.BranchNo) ^ B.City=”London”}

The universal quantifier is used in statements about every instance such as :

*(*∀B)(Branch(B) ^ B.City≠ “Paris”) Means “For all Branch Tuple, the address is not in Paris”

**List the staff who manage properties for rent in Glasgow**

{S | Staff(S) ^(∃P) (PropertyForRent(P)^ (P.StaffNo=S.StaffNo) ^(P.City=”Glasgow”)

With equivalent sql statement is

***Select \* from staff S where exists( select \* from PropertyForRent P where p.staffNo=S.StaffNo and P.city=’Glasgow’)***

***List the names of staff who currently do not manage any properties***

*{S.Fname,S.Lname| Staff(S)^ ((~*∃P)(PropertyForRent(P)^(S.StaffNo=P.StaffNo)))}

Or

*{S.Fname,S.Lname| Staff(S)^ ((*∀P)(PropertyForRent(P)^(S.StaffNo!=P.StaffNo)))}

*List the name of clients who have viewed a property for rent in Glasgow*

*{C.Fname,C.Lname | Clinet (C ) ^ ((*∃V)( ∃P) (Viewing (V) ^ PropertyForRent(P) ^(C.ClientNo=V.ClientNo) ^(V.PropertyNo=P.PropertyNo)^ P.City=’Glasgow’))}

***Please consider :***

S ϵ R, where S is tuple variable and R is a relation.

*ˬ or ^ and not*

***E.g.*** *Consider*

| ***SID*** | ***Sname*** | ***Age*** |
| --- | --- | --- |
| *1* | *Suhas* | *24* |
| *2* | *Jayendra* | *24* |
| *3* | *Sachin* | *35* |
| *4* | *Mahesh* | *23* |

1. ***Select all students having age below 30 years.***

*{ t | t* ϵ student ^ t[age]<25}

1. **Query to select only some attributes of table**

∃{t | Q(t)} where Q is condition.

{t | ∃ S ϵ Student[sid]=s[sid]^s[age]>25}

1. ***Union Operation:***

| ***IT\_Faculty*** | | |  | ***CS\_Faculty*** | | |
| --- | --- | --- | --- | --- | --- | --- |
| *FID* | *FNAME* | *AGE* |  | *FID* | *FNAME* | *AGE* |
| *F1* | *Amruta* | *24* |  | *F11* | *Amruta* | *24* |
| *F2* | *Reshma* | *24* |  | *F2* | *Reshma* | *24* |
| *F3* | *Bhavana* | *35* |  | *F31* | *Geeta* | *35* |

***Find the faculties who are teaching to IT or CS department.***

*{t|* ∃ S ϵ IT\_Faculty (t[Fid]=s[Fid] ^ ⌐ ∃ u ϵ CS\_Faculty(t[Fid]=u[Fid])}

1. ***Cross Product : Same as Relational Algebra.***

{t|∃ S ϵ Student ^ ∃ u ϵ Department}

1. ***Join Operation(Natural Join)=****{t**Q|t* ϵ r and q ∃ S ϵ S}

***e.g. Student(Sid,Sname,Did)***

***Department(Did,Dname) Find All students from IT Department.***

***{t|***∃ S ϵ student(t[sid]=s[sid] ^ ∃ u ϵ Department (u[Did]=s[Did] ^u[Dname]=’IT’

***Consider Player Relation***

***Player(Player\_id,Team\_Id,Name,Age,No\_Of\_Matches)***

1. ***Find all the tuples in player relation***

* *{t|t* ϵ player}

1. ***Find only those tuples for which age is less than 25***

* *{t| t* ϵ player ^[age]<25}

1. ***Find player\_ids of all players***

* *{t |* ∃ S ϵ player(t[player\_id]=s[player\_id]}

1. ***Find player\_ids and name of all players***

* ***{t|***∃ S ϵ player(t[player\_id]=s[player\_id] ^ t[name]=s[name])}

***Consider Following schema and solved by Tuple Relational Calculus***

*Project(proj\_id,proj\_name,chief\_arch)*

*Employee(Emp\_id,Emp\_name)*

*Assigned\_to(Proj\_id,emp\_id)*

1. ***Obtain the emp\_id working on C 353***

***{t|*** ∃ u ϵ assigned\_to(u[proj\_id]=C 353 ^ t[emp\_id]=u[emp\_id])}

1. ***Obtain the details of employees working on C 353***

***{t | t*** ϵ employee ^∃ u ϵ assigned\_to ^ u[emp\_id]=t[emp\_id] ^ u[proj\_id]=C 353))}

1. ***Get the complete details of employees working on operating system project.***

***{s | s*** ϵ employee ^ ∀ t ϵ project(t[proj\_name]=”Operating System”

* ∃ u ϵ Assigned\_to (u[proj\_id]=t[proj\_id^s[emp\_id]=u[emp\_id]))}

1. ***List the complete details of employees working on both C 353 and C 354***

*{s | s*  ϵ employee u1 ϵ Assigned\_to u2 ϵ Assigned\_to ^ u1[emp\_id]=u2[emp\_id]^s[emp\_id]=u1[emp\_id]^u1(proj\_id=C 353 ^ u2[proj\_id]=C 353)}

1. ***List the complete details of employees working on C 353 or C 354***

{s|s ϵ employee ^(∃ u1 ϵ Assigned\_to ^s[emp\_id]=u1[emp\_id]^u1[proj\_id]=C353 )ˬ (∃ u2 ϵ Assigned\_to ^ s[emp\_id]=u2[emp\_id]^ u2[proj\_id]= C354)}

**Tuple Relational Calculus:**It uses domain variables that take on values from an attribute’s domain, rather than values for an entire tuple. An expression in the domain relational calculus is of the form

{<X1,X2,…..Xn> | P (X1,X2,…..Xn)} where X1,X2… represents domain variables. P represents a formula composed of atoms. An atom has one of the following forms:

1. <X1,X2….Xn> ϵ R, where R is a relation on n attributes and X1,X2…Xn are domain variables or domain constants.
2. x Ѳ y, where x and y are domain variables, and Ѳ is a comparison operator(<,><=,>= ,≠); we require that attributes x and y have domains that can compared Ѳ.
3. x Ѳ c, where x is domain variable, Ѳ is comparison operator and c is a constant in the domain of attribute for which x is a domain variable.

**Example:**

1. Find the branch\_name,loan\_no and amount for loans of over 1200.

{<b,l,a> |<b,l,a> ϵ loan ^ a>1200}

1. Find the loan\_no for loans with amount greater than 1200.

{<l> | ∃ b,a(<b,l,a> ϵ loan a^>1200}

1. Find the names of all customers who have a loan from the SBI Khalapur Branch and Find the amount.

{<c,a> | ∃ l(<c,l, ϵ borrower ^ ∃ b(<b,l,a> ϵ loan ^ b=”SBI Khalapur”))}

1. Find the names of all customers who have a loan, an account or both at SBI Khalapur branch

{<c> | ∃ l(<c,l, ϵ borrower ^ ∃ b,a(<b,l,a> ϵ loan ^ b=”SBI Khalapur”)) ˬ ∃ a(<c,a, ϵ depositor ^ ∃ b,n(<b,a,n> ϵ account ^ b=”SBI Khalapur”))}

**Extended Relational Algebra Operations:**

1. **Generalised Projection**: It uses arithmetic functions in the projection list. The generalized projection operation has form

π f1,f2….fn (E ) where E is a any relational algebra expression. F1,f2,…fn are arithmetic expression involving constants and attributes in the schema of E.

e.g.

π player\_id,player\_name (40-age )

π cust\_name, limit-credit\_balance as **credit\_Available** (credit\_info)

1. **Aggregate Functions:** It takes a collection of values and return a single value as a result.

**e.g. Gsum(salary)(works)**

**Gcount-distinct(branch\_name)(works)**

**Other functions are Min, Max,Avg etc.**

1. **Outer Join:** It is an extension of the join operation to deal with missing information. There are 3 forms of outer join.
2. **Left Outer Join**(): It takes all tuples in the left relation that did not match with any tuples in the right relation, pads these tuples with null values for all other attributes from right relation and adds them to the result of natural join.
3. **Right Outer Join(** **) :** It takes all tuples in the right relation that did not match with any tuples in the left relation, pads these tuples with null values for all other attributes from left relation and adds them to the result of natural join.
4. **Full outer Join ( )** It takes all the tuples from left and right and padded them with Null values in the result.

**Consider the example**

| **Employee** | | |  | **Emp\_Branch** | | |
| --- | --- | --- | --- | --- | --- | --- |
| **Emp\_name** | **Street** | **City** |  | **Emp\_name** | **Branch** | **Salary** |
| Rohit | Laxmi | Mumbai |  | Sachin | ICICI | 20000 |
| Rahul | Shivaji | Pune |  | Rahul | SBI | 30000 |
| Sachin | Gandhi | Khalapur |  | Ashish | CBI | 40000 |
| Anil | LT | Pune |  | Rohit | HDFC | 50000 |

**Employee  Emp\_Branch**

| **Emp\_name** | **Street** | **City** | **Branch** | **Salary** |
| --- | --- | --- | --- | --- |
| Rohit | Laxmi | Mumbai | HDFC | 50000 |
| Rahul | Shivaji | Pune | SBI | 30000 |
| Sachin | Gandhi | Khalapur | ICICI | 20000 |

**Employee  Emp\_Branch**

| **Emp\_name** | **Street** | **City** | **Branch** | **Salary** |
| --- | --- | --- | --- | --- |
| Rohit | Laxmi | Mumbai | HDFC | 50000 |
| Rahul | Shivaji | Pune | SBI | 30000 |
| Sachin | Gandhi | Khalapur | ICICI | 20000 |
| Anil | LT | Pune | NULL | NULL |

**Employee  Emp\_Branch**

| **Emp\_name** | **Street** | **City** | **Branch** | **Salary** |
| --- | --- | --- | --- | --- |
| Sachin | Gandhi | Khalapur | ICICI | 20000 |
| Rahul | Shivaji | Pune | SBI | 30000 |
| Ashish | NULL | NULL | CBI | 40000 |
| Rohit | Laxmi | Mumbai | HDFC | 50000 |

**Employee  Emp\_Branch**

| **Emp\_name** | **Street** | **City** | **Branch** | **Salary** |
| --- | --- | --- | --- | --- |
| Rohit | Laxmi | Mumbai | HDFC | 50000 |
| Rahul | Shivaji | Pune | SBI | 30000 |
| Sachin | Gandhi | Khalapur | ICICI | 20000 |
| Ashish | NULL | NULL | CNI | 40000 |
| Anil | LT | Pune | Null | Null |

**Modification of Database:**

Add🡪 Insertion

Remove🡪 Deletion

Change🡪 Updating

**Insertion** : it is represented as

R🡨 r U E where r=relation and E is relational algebra expression

e.g. employee🡨 employee U{(“Raj”, “LT Road”,”Mumbai”)}

**Deletion** : it is represented as

R🡨 r - E where r=relation and E is relational algebra expression

e.g. employee🡨 employee - **σ** emp\_name=”Sachin”(employee)

**Updating** : it is represented as

R🡨 **π F1,F2….Fn**(R ) where F1 is the attribute

**Some Additional Queries find using Relational Algebra, Tuple Relational Calculus, and Domain Relational Calculus**

Consider the following relations

***Employee(emp\_name,street,city)***

***Works(emp\_name,comp\_name,salary)***

***Company(comp\_name,city)***

***Manages(emp\_name,manager\_name)***

1. **Find the names of employees who work for “First Bank Corp”**

**Relational Algebra:**

**π emp\_name(σ comp\_name=”First Bank Corp”(Works))**

**Tuple Relational calculus:**

{ t **|** ∃ S ϵ works(t[emp\_name]=s[emp\_name] s[comp\_name]=”First Bank Corp”)}

**Domain Relational calculus:**

{<e> **|** ∃ c,s(<e,c,s> ϵ works ^ c=”First Bank Corp”)}

1. **Find the names and cities of residence of all employees who work for “First Bank Corp”**

**Relational Algebra:**

**π emp\_name,city(σ comp\_name=”First Bank Corp”(employee works))**

**Tuple Relational calculus:**

{ t **|** ∃ s ϵ works(t[emp\_name]=s[emp\_name] ^s[comp\_name]=^ ∃ u ϵ employee (t[city]=u[city]^u[emp\_name”=t[emp\_name] ^ u[comp\_name]=”First Bank Corp”)}

**Domain Relational calculus:**

{<e,i> **|** ∃ s(<e,c,s> ϵ works ^ c=”First Bank Corp” ^ ∃ <e,t,i> ϵ (employee))}

1. **Find the names ,street and cities of residence of all employees who work for “FBC” and earn more than 10000.**

**Relational Algebra**

**π emp\_name,city,street(σ comp\_name=”First Bank Corp” ^ salary>10000(employee works))**

**Tuple**

{ t **|** ∃ s ϵ works(t[emp\_name]=s[emp\_name] ^s[comp\_name]=”FBC” ^ [salary]>10000) ^ ∃ u ϵ employee (t[city]=u[city]^t[street]=u[street] ^[emp\_name”=t[emp\_name] )}

**Domain Relational calculus:**

{<e,i,t> **|** ∃ s(<e,c,s> ϵ works ^ c=”First Bank Corp” ^ salary>10000 ^ ∃ <e,t,i> ϵ (employee))}

1. **Find the names of employees in the database who live in the same city as the company they work for.**

**Relational Algebra**

**π emp\_name,  (employee workscompany))**

**Tuple**

{ t **|** ∃ s ϵ works,u ϵ employee, p ϵ company (t[emp\_name]=s[emp\_name] =^s[comp\_name]=”FBC” ^ [salary]>10000) ^ ∃ u ϵ employee (t[city]=u[city]^t[street]=u[street] ^[emp\_name”=t[emp\_name] )}

**Domain Relational calculus:**

{<e> **|** ∃ <c,s>(<e,c,s> ϵ works ∃<t,i>(<e,t,i> ϵ employee ^ ∃<c,i> ϵ company))}

**Example : Tuple Relational Calculus**

*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 *loan\_number, branch\_name,* and *amount* for loans of over $1200**

{*t* | *t* ∈ *loan* ∧ *t* [*amount* ] > 1200}

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

{*t* |∃ *s ∈* loan (*t* [*loan\_number* ] = *s* [*loan\_number* ] ∧ *s* [*amount* ] > 1200)}

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

{*t* |∃*s* ∈ *borrower ( t* [*customer\_name* ] = *s* [*customer\_name* ])  
 ∨ ∃*u* ∈ *depositor* ( *t* [*customer\_name* ] = *u* [*customer\_name* ])

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

{*t* |∃*s* ∈ *borrower ( t* [*customer\_name* ] = *s* [*customer\_name* ])

∧ ∃*u* ∈ *depositor* ( *t* [*customer\_name* ] = *u* [*customer\_name*] )

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

{*t* |∃*s* ∈ *borrower* (*t* [*customer\_name* ] = *s* [*customer\_name* ]   
 ∧ ∃*u* ∈ *loan* (*u* [*branch\_name* ] = “Perryridge”  
 ∧ *u* [*loan\_number* ] = *s* [*loan\_number* ]))}

**Find the names of all customers who have a loan at the Perryridge branch, but no account at any branch of the bank**

{*t* |∃*s* ∈ *borrower* (*t* [*customer\_name* ] = *s* [customer\_name ]  
 ∧ ∃*u* ∈ *loan* (*u* [*branch\_name* ] = “Perryridge”  
 ∧ *u* [*loan\_number* ] = *s* [loan\_*number* ]))  
 ∧ **not** ∃*v* ∈ *depositor* (*v* [*customer\_name* ] =   
 *t* [*customer\_name* ])}

**Find the names of all customers having a loan from the Perryridge branch, and the cities in which they live**

{*t* |∃*s* ∈ *loan* (*s* [*branch\_name* ] = “Perryridge”  
 ∧ ∃*u* ∈ *borrower* (*u* [*loan\_number* ] = *s* [*loan\_number* ]  
 ∧ *t* [*customer\_name* ] = *u* [*customer\_name* ])  
 ∧ ∃ *v* ∈ *customer* (*u* [*customer\_name* ] = *v* [*customer\_name* ]  
 ∧ *t* [*customer\_city* ] = *v* [*customer\_city* ])))}

**Data Manipulation:** The DM part of relational model makes set processing facilities available to the user. Since relational operators are able to manipulate tables, the application programs do not need to use loops. The primary aim of database in a enterprise is to provide the information to the users within the enterprise. The process of querying a relational database is in essence a way of manipulating the tables that are the database. The relational algebra and relational calculus are two formal languages used for data manipulation.

**Data Integrity:** The aim of data integrity is to specify rules that implicitly or explicitly define a consistent database state or change of state.

The Integrity constraints are of two types:

1. **Static Integrity constraints**: These defines valid states of the data. These constraints include designations of primary key.
2. **Dynamic Integrity Constraints:** These are constraints that defines side-effects of various kinds of transactions carried out in the database. E.g. Trigger

**Static :**

**Entity Integrity Constraint(Primary Key)**: The primary key is important since it is the sole identifier for the rows in the table. Following are the constraints : i. No component of a primary key value can be null. ii. No two rows have a same primary key value. iii. Attempts to change the value of primary key must be carefully controlled. Either it is deleted the complete record or value must be check.

**Foreign Key** : The foreign key in table R is a set of columns whose values are required to match those of the primary key of some table S. R and S are not necessary distinct.

**Referential Integrity:** If a foreign key F in table R refers to and matches the primary key P of table S then every value of F must either be equal to a value of P or be wholly Null.

**Domains:** The domains must be specified and valid.

**Nulls:** The issue of Null values in database is important. A value in the database is null for several reasons. The most important the value is not present at a particular time. E.g. No. of dependent if employee is unmarried. Operations on the Null values is difficult.

**Advantages of the Relational Database :**

**Data Independence:** To provide a sharp and clear boundary between the logical and physical aspects of database management.

**Simplicity:** To provide a simpler structure than was being used at that time. A simple structure is easy to communicate to users and programmers and a wide variety of users in an enterprise can interact with simple model.

**Set Processing:** To provide facilities for manipulating a set of records at a time so that programmers are not operating on the database record by record.

**Sound theoretical Background:** To provide a theoretical background for the database management field.

**Relational Algebra Vs Relational Calculus:**

1. **Ease of Augmentation:** Since the algebra and calculus provided only basic selective power, it is necessary to augment that power by providing a facility to invoke a number of library functions. This augmentation is more easily done in relational calculus than in algebra.
2. **Scope for search Optimization:** Since the calculus only specifies what is to be retrieved and not how it is to be retrieved, it is easier for a database system to optimize database search to answer queries that are posed in calculus. Queries posed in the relational algebra can also be optimized but such optimization is likely to be only local since it would be difficult for a system to derive the properties of the data that needs to be retrieved given the operators that the user has specified.
3. **Authorization Capability**: It is likely to be easier to specify authorization by using relational calculus since authorization is likely to be based on the properties of the data that a user may be authorized to access or update.
4. Closeness to Natural Language: It is much natural of users in the relational calculus is specify what data they wish to retrieve than how the data is to be retrieved. Thus a language based on relational calculus is likely to be more user friendly and easier to use for users with limited computing background.

**Questions:**

1. Solve the above all queries using by Relational Algebra, Tuple & Domain Relational Calculus.
2. Explain the Different Relational Algebra operations with example
3. Natural Join
4. Rename
5. Set Difference
6. Generalised Projection
7. Group by functions
8. Division
9. Join
10. Division
11. Set Interaction
12. Union
13. Explain any three relational algebra operations with example.
14. Explain How weak entity is converted from ER Diagram into Relational Database schema.
15. Explain how the 1:1,1:M,M:M ,M:N relationships are converted into Relational Schema from ER Diagram