UNIT-II:

The E/R Models, The Relational Model, Relational Calculus, Introduction to Database Design, Database Design and ER Diagrams-Entities Attributes, and Entity Sets-Relationship and Relationship Sets-Conceptual Design With the ER Models, **The Relational Model** Integrity Constraints Over Relations- Key Constraints –Foreign Key Constraints-General Constraints, **Relational Algebra and Calculus**, Relational Algebra- Selection and Projection- Set Operation, Renaming – Joins- Division- More Examples of Queries, Relational Calculus, Tuple Relational Calculus- Domain Relational Calculus.

DATABASE DESIGN:

The database design process can be divided into six steps. The ER Model is most relevant to the first three steps. Next three steps are beyond the ER Model.

1. Requirements Analysis:

The very first step in designing a database application is to understand what data is to be stored in the database, what applications must be built on top of it, and what operations are most frequent and subject to performance requirements. The database designers collect information of the organization and analyzer, the information to identify the user's requirements. The database designers must find out what the users want from the database.

2. Conceptual Database Design:

Once the information is gathered in the requirements analysis step a conceptual database design is developed and is used to develop a high level description of the data to be stored in the database, along with the constraints that are known to hold over this data. This step is often carried out using the ER model, or a similar high-level data model.

3. Logical Database Design:

In this step convert the conceptual database design into a database schema (Logical Database Design) in the data model of the chosen DBMS. We will only consider **relational DBMSs**, and therefore, the task in the logical design step is to convert an ER schema into a relational database schema. The result is a conceptual schema, sometimes called the **logical schema**, in the relational data model.

Beyond the ER Design:

The first three steps are more relevant to the ER Model. Once the logical scheme is defined designer consider the physical level implementation and finally provide certain security measures. The remaining three steps of database design are briefly described below:

4. Schema Refinement:

The fourth step in database design is to analyze the collection of relations in our relational database schema to identify potential problems, and to refine it. In contrast to the requirements analysis and conceptual design steps, which are essentially subjective, schema refinement can be guided by some elegant and powerful theory.

5. **Physical Database Design:**

In this step we must consider typical expected workloads that our database must support and further refine the database design to ensure that it meets desired performance criteria. This step may simply involve building indexes on some tables and clustering some tables, or it may involve a substantial redesign of parts of the database schema obtained from the earlier design steps.

6. Security Design:

The last step of database design is to include security features. This is required to avoid unauthorized access to database practice after all the six steps. We required Tuning step in which all the steps are interleaved and repeated until the design is satisfactory.

What is Schema?

A database schema is the skeleton structure that represents the logical view of the entire database. (Or) The logical structure of the database is called as Database Schema. (or)

The overall design of the database is the database schema.

1. It defines how the data is organized and how the relations among them are associated.
2. It formulates all the constraints that are to be applied on the data.

E.g: STUDENT SID SNAME PHONENUMBER

What is Instance?

The actual content of the database at a particular point in time. (Or) The data stored in the database at any given time is an instance of the database

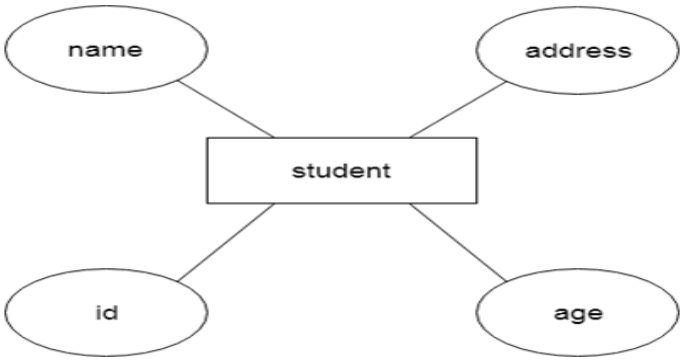
Sid	**Name**	**Phnumber**
1201	Venkat	9014901442
1202	Teja	9014774422
1203	Ram	9848022332

Entity Relationship model (E-R model):-

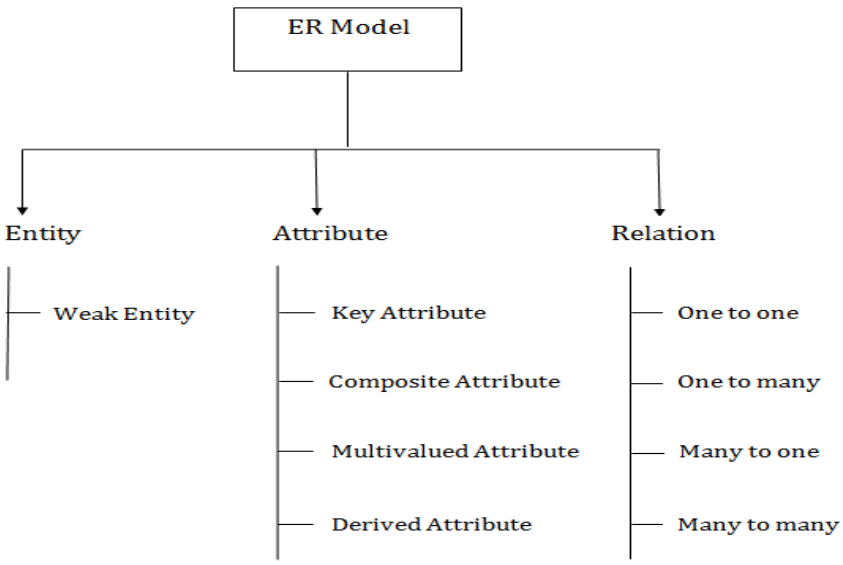
ER Model

- ER model stands for an Entity-Relationship model. It is a high-level data model. This model is used to define the data elements and relationship for a specified system.
- It develops a conceptual design for the database. It also develops a very simple and easy to design view of data.
- In ER modeling, the database structure is portrayed as a diagram called an entity-relationship diagram.

For example, suppose we design a school database. In this database, the student will be an entity with attributes like address, name, id, age, etc. The address can be another entity with attributes like city, street name, pin code, etc and there will be a relationship between them.



Components of ER Diagram



1. Entity:

An entity may be any object, class, person or place. In the ER diagram, an entity can be represented as rectangles.

Consider an organization as an example- manager, product, employee, department etc. can be taken as an entity.

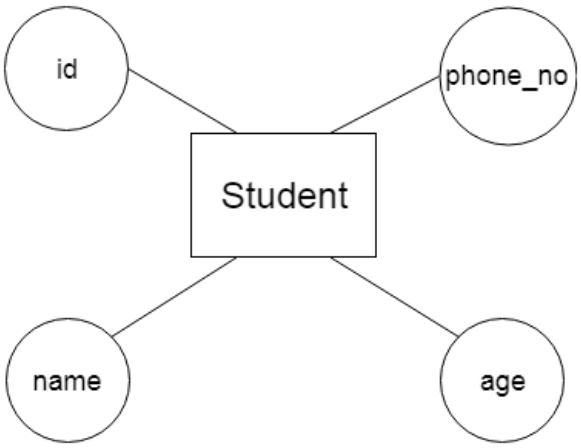
a. Weak Entity

An entity that depends on another entity called a weak entity. The weak entity doesn't contain any key attribute of its own. The weak entity is represented by a double rectangle.

2. Attribute

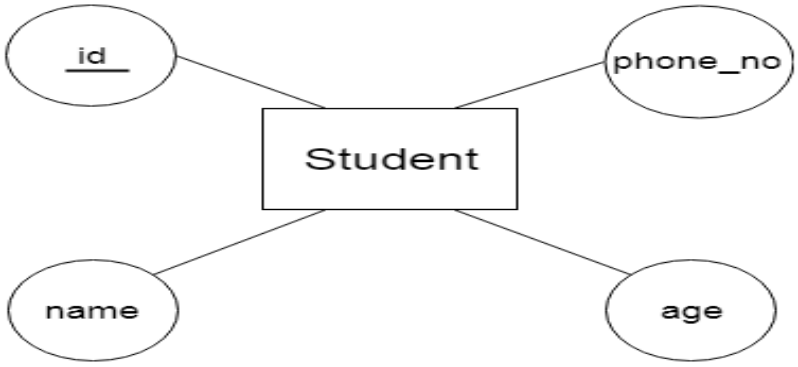
The attribute is used to describe the property of an entity. Eclipse is used to represent an attribute.

For example, id, age, contact number, name, etc. can be attributes of a student.



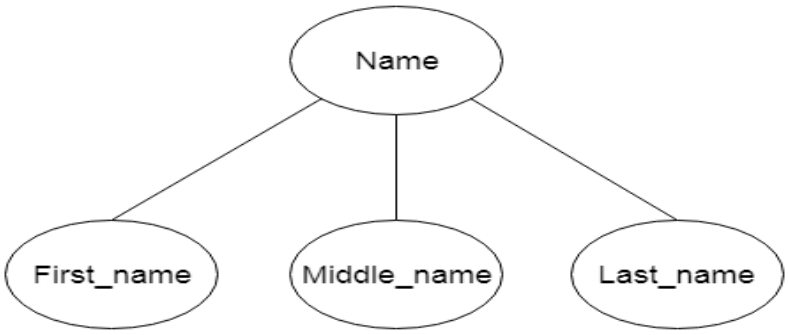
a. Key Attribute

The key attribute is used to represent the main characteristics of an entity. It represents a primary key. The key attribute is represented by an ellipse with the text underlined.



b. Composite Attribute

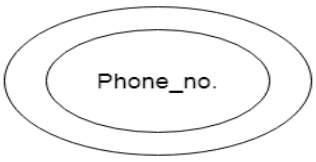
An attribute that composed of many other attributes is known as a composite attribute. The composite attribute is represented by an ellipse, and those ellipses are connected with an ellipse.



c. Multivalued Attribute

An attribute can have more than one value. These attributes are known as a multivalued attribute. The double oval is used to represent multivalued attribute.

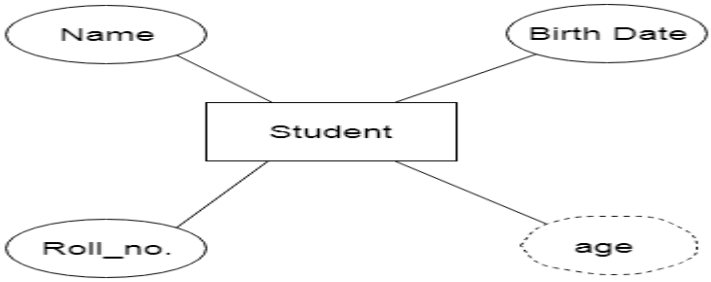
For example, a student can have more than one phone number.



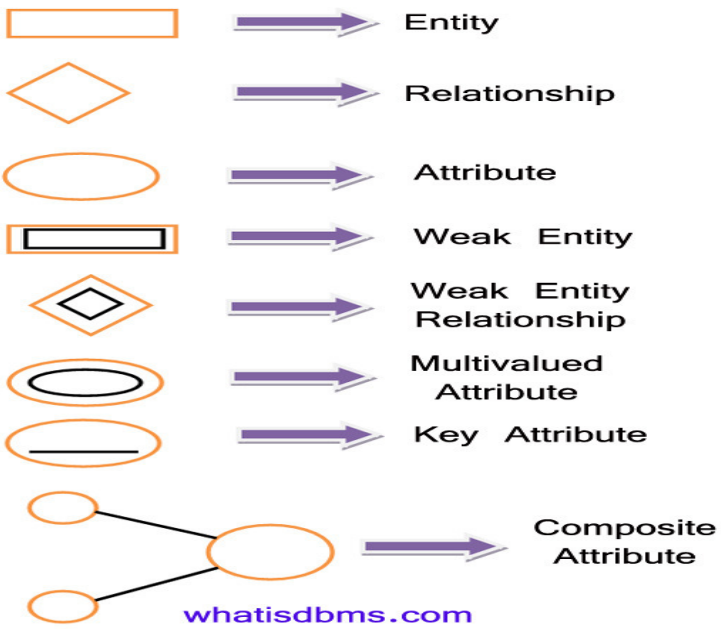
d. Derived Attribute

An attribute that can be derived from other attribute is known as a derived attribute. It can be represented by a dashed ellipse.

For example, A person's age changes over time and can be derived from another attribute like Date of birth.



E-R diagram notations:-



Relationships:

Relationship as "an association among entities"

For Ex: there is a relationship called DEPT– EMP between departments and employees, representing the fact that certain departments employ certain employees.

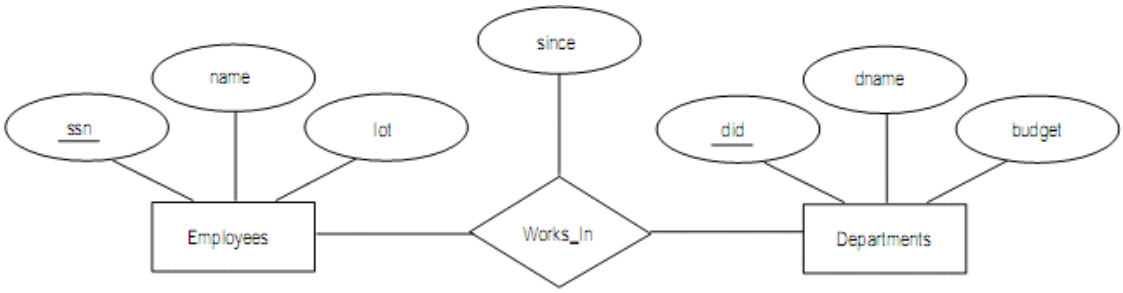


Figure 2.2 The Works_In Relationship Set

As with entities, it is necessary in principle to distinguish between **relationship types and relationship instances.**

The entries involved in a given relationship are said to be participants in that relationship. The number of participants in a given relationship is called the **degree** of that relationship.

Let R be a relationship type that involves entity type E as a participant. If every instance of E participates in at least one instance of R, then the participation of E in R is said to be **total**, otherwise it is said to be **partial.**

For Ex: if every employee must belong to a department, then the participation of employees in DEPT-EMP is total; if it is possible for a given department to have no employees at all, then the participation of departments in DEPT – EMP is partial.

The database structure, employing the E-R model is usually shown pictorially using entity – relationship.

Types of relationship are as follows:

a. One-to-One Relationship

When only one instance of an entity is associated with the relationship, then it is known as one to one relationship.

For example, A female can marry to one male, and a male can marry to one female.

b. One-to-many relationship

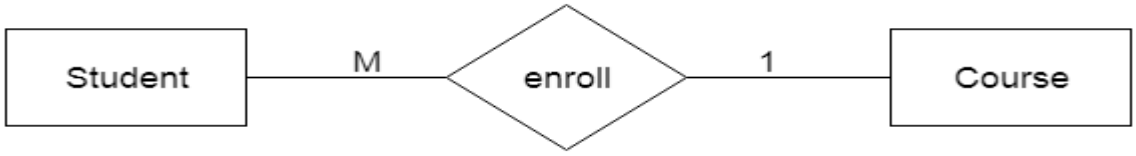
When only one instance of the entity on the left and more than one instance of an entity on the right associates with the relationship then this is known as a one-to-many relationship.

For example, Scientist can invent many inventions, but the invention is done by the only specific scientist.

c. Many-to-one relationship

When more than one instance of the entity on the left and only one instance of an entity on the right associates with the relationship then it is known as a many-to-one relationship.

For example, Student enrolls for only one course, but a course can have many students.



d. Many-to-many relationship

When more than one instance of the entity on the left and more than one instance of an entity on the right associates with the relationship then it is known as a many-to-many relationship.

For example, Employee can assign by many projects and project can have many employees.

Conceptual Database Design With The ER Model

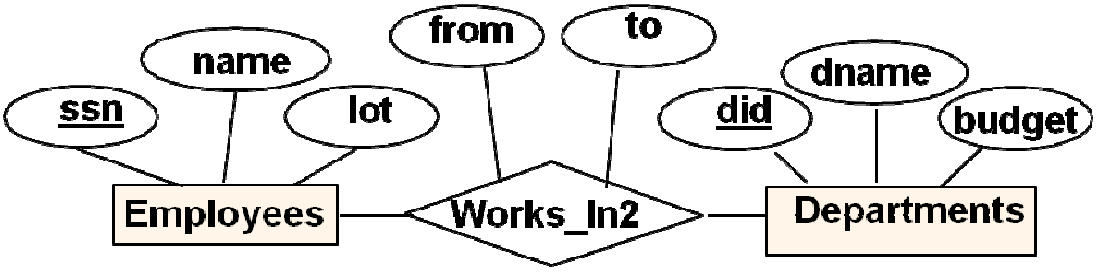
Developing an ER diagram presents several choices, including the following:

- Should a concept be modelled as an entity or an attribute?
- Should a concept be modelled as an entity or a relationship?
- What are the relationship sets and their participating entity sets?
- Should we use binary or ternary relationships?
- Should we use aggregation?

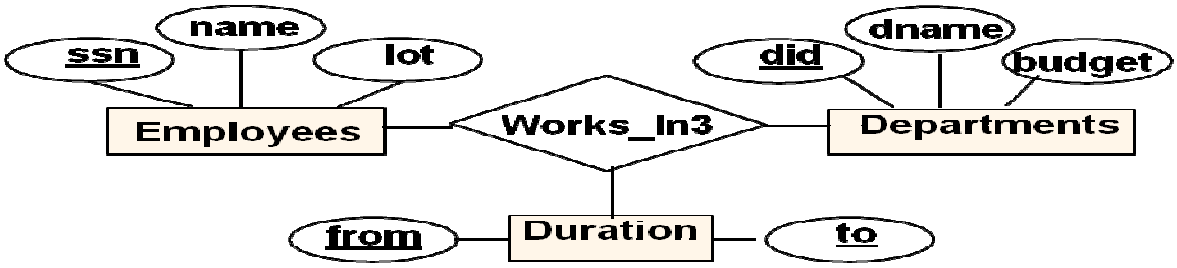
Entity versus Attribute

While identifying the attributes of an entity set, it is sometimes not clear whether a property should be modelled as an attribute or as an entity set. For example, consider adding address information to the Employees entity set. One option is to use an attribute address. This option is appropriate if we need to record only one address per employee, and it suffices to think of an address as a string. An alternative is to create an entity set called Addresses and to record associations between employees and addresses using a relationship

For another example of when to model a concept as an entity set rather than an attribute, consider the relationship set shown in Figure

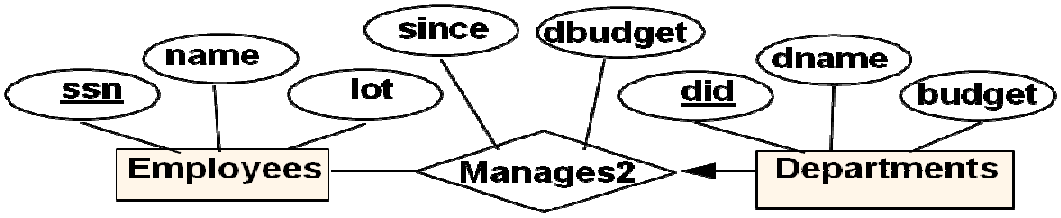


The problem is that we want to record several values for the descriptive attributes for each instance of the works-ln2 relationship. (This situation is analogous to wanting to record several addresses for each employee.) We can address this problem by introducing an entity set called, say, Duration, with attributes from and to, as shown in Figure



Entity versus Relationship

Consider the relationship set called Manages in Figure. Suppose that each department manager is given a discretionary budget (dbudget) , as shown in Figure, in which we have also renamed the relationship set to Manages2.



Given a department, we know the manager, as well the manager's starting date and budget for that department. This approach is natural if we assume that a manager receives a separate discretionary budget for each department that he or she manages.

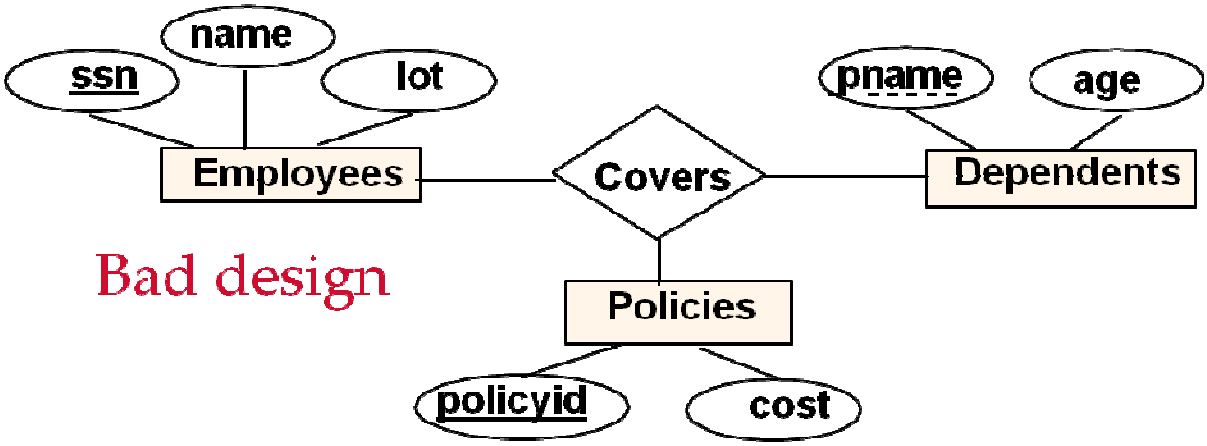
Binary versus Ternary Relationships

Consider the ER diagram shown in Figure. It models a situation in which an employee can own several policies, each policy can be owned by several employees, and each dependent can be covered by several policies.

Suppose that we have the following additional requirements:

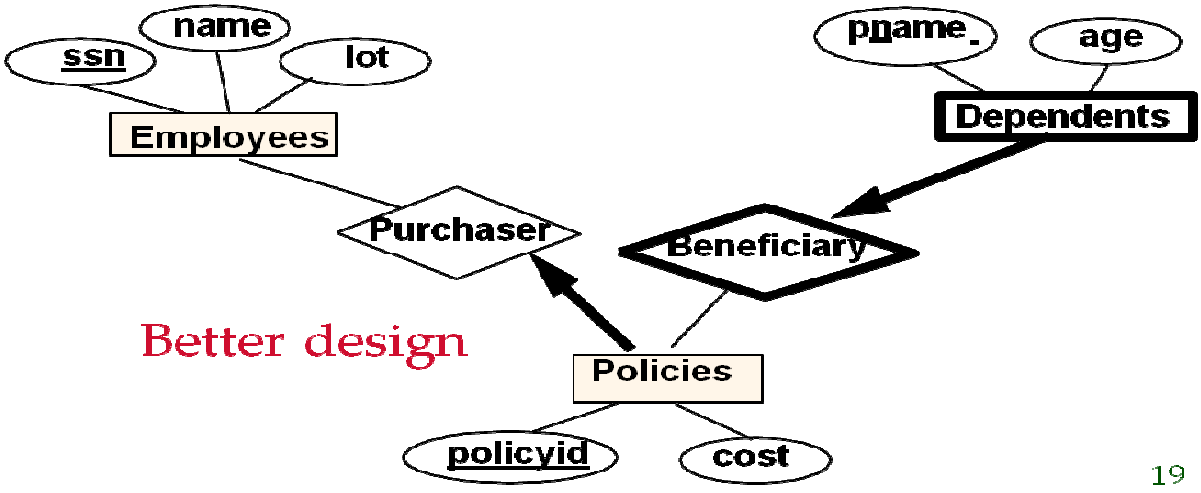
- A policy cannot be owned jointly by two or more employees.
- Every policy must be owned by some employee.

Dependents are a weak entity set, and each dependent entity is uniquely identified by taking pname in conjunction with the policyid of a policy entity (which, intuitively, covers the given dependent). The first requirement suggests that we impose a key constraint on Policies with respect to Covers, but this constraint has the unintended side effect that a



Policy can cover only one dependent. The second requirement suggests that we impose a total participation constraint on Policies. This solution is acceptable if each policy covers at least one dependent. The third requirement forces us to introduce an identifying relationship that is binary (in our version of ER diagrams, although there are versions in which this is not the case).

Even ignoring the third requirement, the best way to model this situation is to use two binary relationships, as shown in Figure



19

Class Hierarchies:

The entity set Employees may also be classified using a different criterion. For example, we might identify a subset of employees as SeniorEmps. We can Modify Figure 2.12 to reflect this change by adding a second ISA node &'3 a child of Employees and making SeniorEmps a child of this node. Each of these entity sets might be classified further, creating a multilevel IS A hierarchy.

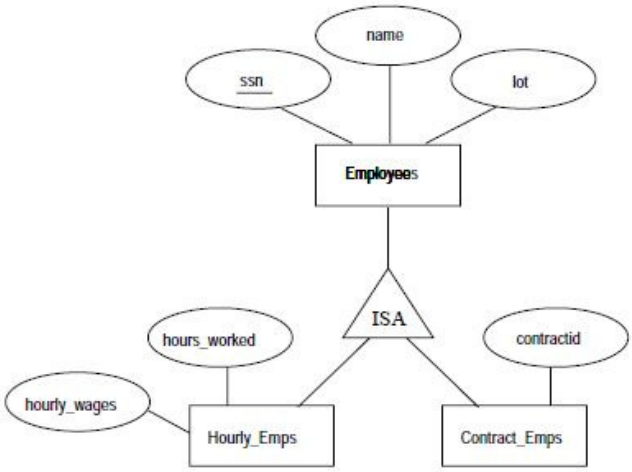


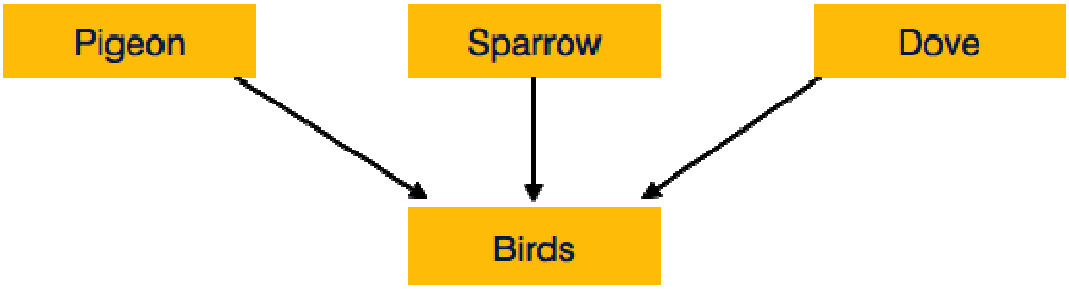
Figure 2.12 Class Hierarchy

Class hierarchy can be viewed one of two ways

1. Specialization (Top Down Approach)
2. Generalization (Bottom Up Approach)

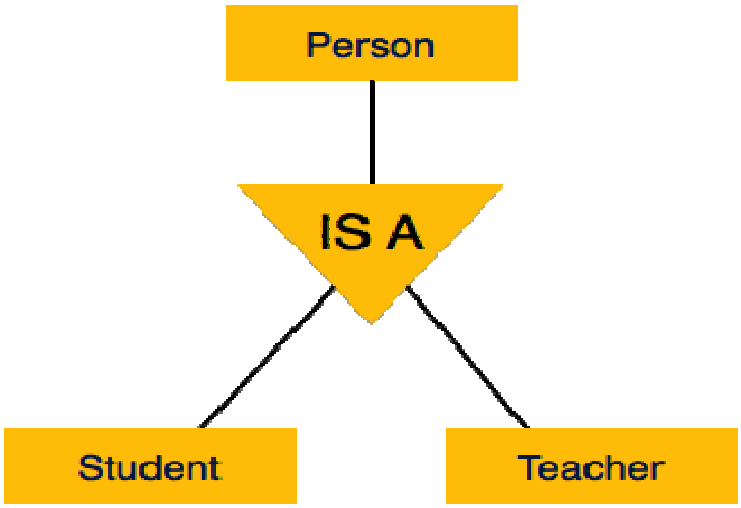
Generalization

As mentioned above, the process of generalizing entities, where the generalized entities contain the properties of all the generalized entities is called generalization. In generalization, a number of entities are brought together into one generalized entity based on their similar characteristics. For example, pigeon, house sparrow, crow and dove can all be generalized as Birds.



Specialization

Specialization is the opposite of generalization. In specialization, a group of entities is divided into sub-groups based on their characteristics. Take a group 'Person' for example. A person has name, date of birth, gender, etc. These properties are common in all persons, human beings. But in a company, persons can be identified as employee, employer, customer, or vendor, based on what role they play in the company.

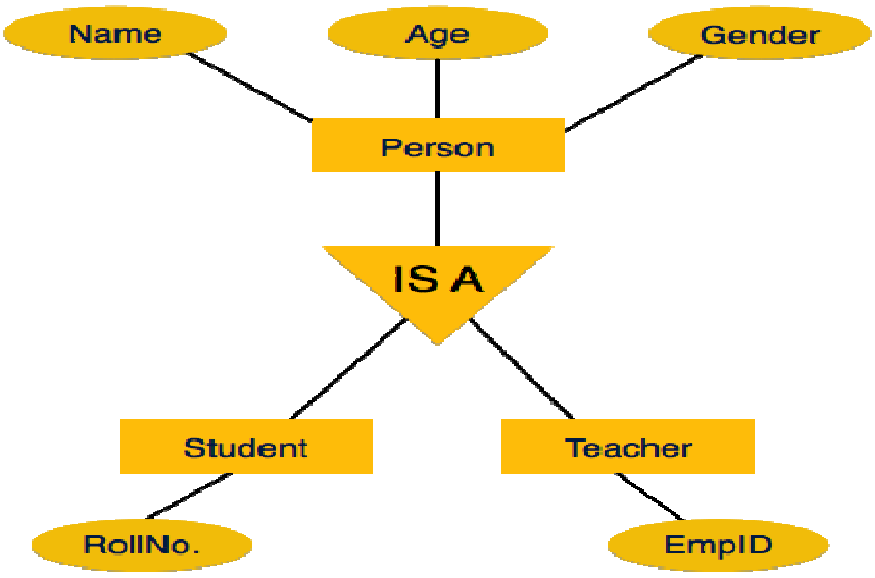


Similarly, in a school database, persons can be specialized as teacher, student, or a staff, based on what role they play in school as entities.

Inheritance

We use all the above features of ER-Model in order to create classes of objects in object-oriented programming. The details of entities are generally hidden from the user; this process known as **abstraction**.

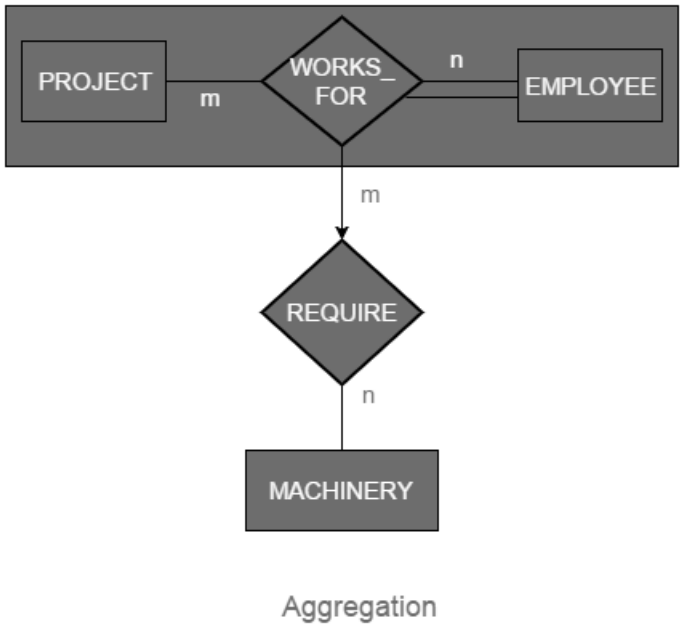
Inheritance is an important feature of Generalization and Specialization. It allows lower-level entities to inherit the attributes of higher-level entities.



For example, the attributes of a Person class such as name, age, and gender can be inherited by lower-level entities such as Student or Teacher.

Aggregation

An ER diagram is not capable of representing relationship between an entity and a relationship which may be required in some scenarios. In those cases, a relationship with its corresponding entities is aggregated into a higher level entity. For Example, Employee working for a project may require some machinery. So, REQUIRE relationship is needed between relationship WORKS_FOR and entity MACHINERY. Using aggregation, WORKS_FOR relationship with its entities EMPLOYEE and PROJECT is aggregated into single entity and relationship REQUIRE is created between aggregated entity and MACHINERY.



Aggregation

Relational Model

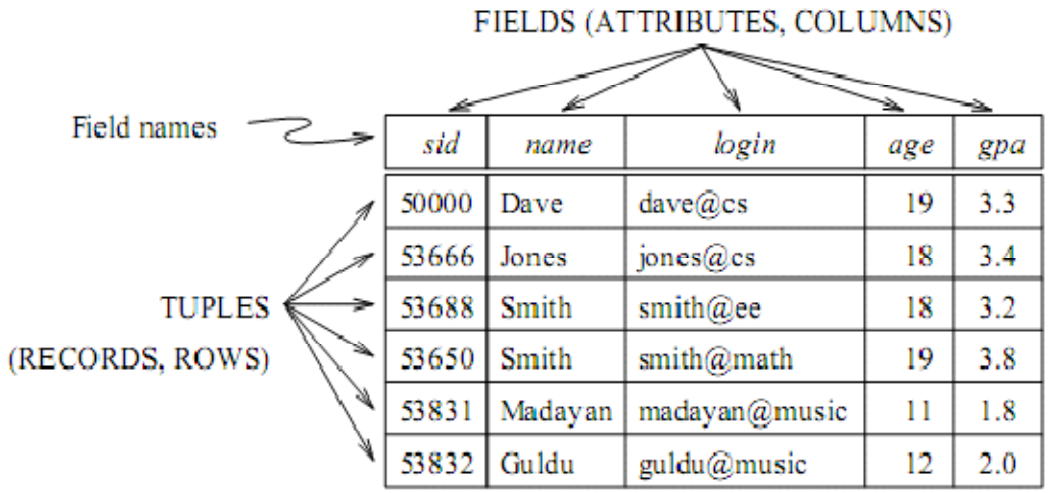
Introduction to the Relational Model

The main construct for representing data in the relational model is a relation. A relation consists of a relation schema and a relation instance. The relation instance is a table, and the relation schema describes the column heads for the table. We first describe the relation schema and then the relation instance. The schema specifies the relation's name, the name of each field (or column, or attribute), and the domain of each field. A domain is referred to in a relation schema by the domain name and has a set of associated values.

Eg: Students (Sid: string, name: string, login: string, age: integer, gpa: real)

This says, for instance, that the field named sid has a domain named string. The set of values associated with domain string is the set of all character strings.
An instance of a relation is a set of tuples, also called records, in which each tuple has the same number of fields as the relation schema. A relation instance can be thought of as a table in which each tuple is a row, and all rows have the same number of fields.

An instance of the Students relation appears in Figure.



sid	*name*	*login*	*age*	*gpa*
50000	Dave	dave@cs	19	3.3
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0

Figure 3.1 An Instance $S1$ of the Students Relation

A relation schema specifies the domain of each field or column in the relation instance. These domain constraints in the schema specify an important condition that we want each instance of the relation to satisfy: The values that appear in a column must be drawn from the domain associated with that column. Thus, the domain of a field is essentially the type of that field, in programming language terms, and restricts the values that can appear in the field.

Domain constraints are so fundamental in the relational model that we will henceforth consider only relation instances that satisfy them; therefore, relation instance means relation instance that satisfies the domain constraints in the relation schema.

The degree, also called arity, of a relation is the number of fields. The cardinality of a relation instance is the number of tuples in it. In Figure, the degree of the relation (the number of columns) is five, and the cardinality of this instance is six.

A relational database is a collection of relations with distinct relation names. The relational database schema is the collection of schemas for the relations in the database.

Creating and Modifying Relations

The SQL-92 language standard uses the word table to denote relation, and we will often follow this convention when discussing SQL. The subset of SQL that supports the creation, deletion, and modification of tables is called the **Data Definition Language (DDL)**.

To create the Students relation, we can use the following statement:

The CREATE TABLE statement is used to define a new table.

CREATE TABLE Students (Sid CHAR(20), name CHAR(30), login CHAR(20), age INTEGER, gpa REAL)

Tuples are inserted using the INSERT command. We can insert a single tuple into the Students table as follows:

INSERT INTO Students (sid, name, login, age, gpa) VALUES (53688, ‘Smith’, ‘smith@ee’, 18, 3.2)

We can delete tuples using the DELETE command. We can delete all Students tuples with name equal to Smith using the command:

DELETE FROM Students S WHERE S.name = ‘Smith’

We can modify the column values in an existing row using the UPDATE command. For example, we can increment the age and decrement the gpa of the student with sid 53688:

UPDATE Students S SET S.age = S.age + 1, S.gpa = S.gpa - 1 WHERE S.sid = 53688

Integrity Constraints Over Relations

An integrity constraint (IC) is a condition that is specified on a database schema, and restricts the data that can bstored in an instance of the database. If a database instance satisfies all the integrity constraints specified on the database schema, it is a legal instance. A DBMS enforces integrity constraints, in that it permits only legal instances to be stored in the database.

Key Constraints

Consider the Students relation and the constraint that no two students have the same student id. This IC is an example of a key constraint. A key constraint is a statement that a certain minimal subset of the fields of a relation is a unique identifier for a tuple. A set of fields that uniquely identifies a tuple according to a key constraint is called a **candidate key** for the relation; we often abbreviate this to just key. In the case of the Students relation, the (set of fields containing just the) sid field is a candidate key.

There are two parts to the definition of (candidate) key:

1. Two distinct tuples in a legal instance (an instance that satisfies all ICs, including the key constraint) cannot have identical values in all the fields of a key.

2. No subset of the set of fields in a key is a unique identifier for a tuple.

The first part of the definition means that in any legal instance, the values in the key fields uniquely identify a tuple in the instance

The second part of the definition means, for example, that the set of fields {sid, name} is not a key for Students, because this set properly contains the key {sid}.The set {sid, name} is an example of a **superkey**, which is a set of fields that contains a key.

Out of all the available candidate keys, a database designer can identify a **primary** key. Intuitively, a tuple can be referred to from elsewhere in the database by storing the values of its primary key fields. For example, we can refer to a Students tuple by storing its sid value.

Specifying Key Constraints in SQL

CREATE TABLE Students (sid CHAR(20), name CHAR(30), login CHAR(20), age INTEGER, gpa REAL, UNIQUE (name, age), CONSTRAINT Students Key PRIMARY KEY (sid))

This definition says that sid is the primary key and that the combination of name and age is also a key. The definition of the primary key also illustrates how we can name a constraint by preceding it with CONSTRAINT constraint-name. If the constraint is violated, the constraint name is returned and can be used to identify the error.

Foreign Key Constraints

Sometimes the information stored in a relation is linked to the information stored in another relation. If one of the relations is modified, the other must be checked, and perhaps modified, to keep the data consistent. An IC involving both relations must be specified if a DBMS is to make such checks. The most common IC involving two relations is a foreign key constraint.

Suppose that in addition to Students, we have a second relation:

Enrolled (sid: string, cid: string, grade: string)

To ensure that only bonafide students can enroll in courses, any value that appears in the sid field of an instance of the Enrolled relation should also appear in the sid field of some tuple in the Students relation. The sid field of Enrolled is called a **foreign key** and **refers** to Students. The foreign key in the referencing relation (Enrolled, in our example) must match the primary key of the referenced relation (Students), i.e., it must have the same number of columns and compatible data types, although the column names can be different.

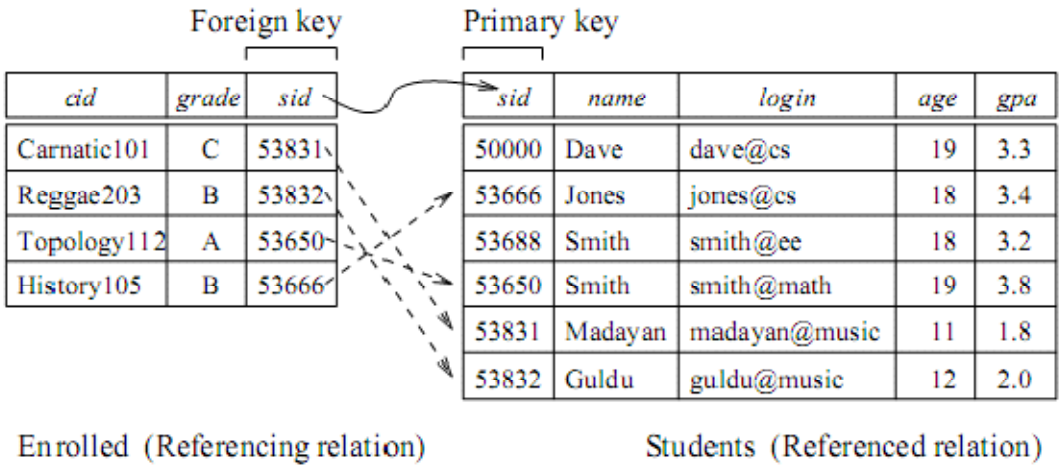


Figure 3.4 Referential Integrity

Specifying Foreign Key Constraints in SQL

CREATE TABLE Enrolled (sid CHAR(20), cid CHAR(20), grade CHAR(10), PRIMARY KEY (sid, cid), FOREIGN KEY (sid) REFERENCES Students)

Relational Algebra

Preliminaries:

A query language is a language in which user requests to retrieve some information from the database. The query languages are considered as higher level languages than programming languages. Query languages are of two types,

Procedural Language

Non-Procedural Language

1. In procedural language, the user has to describe the specific procedure to retrieve the information from the database.

Example: The Relational Algebra is a procedural language.

2. In non-procedural language, the user retrieves the information from the database without describing the specific procedure to retrieve it.

Example: The Tuple Relational Calculus and the Domain Relational Calculus are non-procedural languages.

Relational Algebra

The relational algebra is a procedural query language. It consists of a set of operations that take one or two relations (tables) as input and produce a new relation, on the request of the user to retrieve the specific information, as the output.

The relational algebra contains the following operations,

1) Selection 2) Projection 3) Union 4) Rename 5) Set-Difference

6) Cartesian product 7) Intersection 8) Join 9) Divide 10) Assignment

The Selection, Projection and Rename operations are called unary operations because they operate only on one relation. The other operations operate on pairs of relations and are therefore called binary operations.

1) The Selection (σ) operation:

The Selection is a relational algebra operation that uses a condition to select rows from a relation. A new relation (output) is created from another existing relation by selecting only rows requested by the user that satisfy a specified condition. The lower greek letter 'sigma σ' is used to denote selection operation.

General Syntax: **Selection condition (relation name)**

Example: Find the customer details that are living in Hyderabad city from customer relation.

σ city = 'Hyderabad' (customer)

The selection operation uses the column names in specifying the selection condition. Selection conditions are same as the conditions used in the 'if' statement of any programming languages, selection condition uses the relational operators < > <= >= != . It is possible to combine several conditions into a large condition using the logical connectives 'and' represented by '∧ 'and 'or' represented by '∨'.

Example:

Find the customer details who are living in Hyderabad city and whose customer_id is greater than 1000 in Customer relation.

σ city = 'Hyderabad' ^ customer_id > 1000 (customer)

2) The Projection (π) operation:

The projection is a relational algebra operation that creates a new relation by deleting columns from an existing relation i.e., a new relation (output) is created from another existing relation by selecting only those columns requested by the user from projection and is denoted by letter pi (π).

The Selection operation eliminates unwanted rows whereas the projection operation eliminates unwanted columns. The projection operation extracts specified columns from a table.

Example: Find the customer names (not all customer details) who are living in Hyderabad city from customer relation.

π customer_name (σ city = 'Hyderabad' (customer))

In the above example, the selection operation is performed first. Next, the projection of the resulting relation on the customer_name column is carried out. Thus, instead of all customer details of customers living in Hyderabad city, we can display only the customer names of customers living in Hyderabad city.

The above example is also known as relational algebra expression because we are combining two or more relational algebra operations (ie. selection and projection) into one at the same time.

Example: Find the customer names (not all customer details) from customer relation.

π customer_name (customer)

The above stated query lists all customer names in the customer relation and this is not called as relational algebra expression because it is performing only one relational algebra operation.

3) The Set Operations: (**Union**, Intersection, Set-Difference, Cartesian product)

i) Union 'U' Operation:

The union denoted by U It is a relational algebra operation that creates a union or combination of two relations. The result of this operation, denoted by d U b is a relation that includes all tuples that all either in d or in b or in both d and b, where duplicate tuples are eliminated.

Example: Find the customer_id of all customers in the bank who have either an account or a loan or both.

π customer_id (depositor) U **π customer_id (borrower)**

To solve the above query, first find the customers with an account in the bank. That is **π customer_id (depositor)**. Then, we have to find all customers with a loan in the bank, **π customer_id (borrower)**. Now, to answer the above query, we need the union of these two sets, that is, all customer names that appear in either or both of the two relations by

π customer_id (depositor) U π customer_id (borrower)

If some customers A, B and C are both depositors as well as borrowers, then in the resulting relation, their customer ids will occur only once because duplicate values are eliminated.

Therefore, for a union operation d U b to be valid, we require that two conditions to be satisfied,

i) The relations depositor and borrower must have same number of attributes / columns.

ii) The domains of i^{th} attribute of depositor relation and the ith attribute of borrower relation must be the same, for all i.

The Intersection '∩' Operation:

The intersection operation denoted by '∩' It is a relational algebra operation that finds tuples that are in both relations. The result of this operation, denoted by d ∩ b, is a relation that includes all tuples common in both depositor and borrower relations.

Example: Find the customer_id of all customers in the bank who have both an account and a loan.

π customer_id (depositor) ∩ π customer_id (borrower)

The resulting relation of this query lists all common customer ids of customers who have both an account and a loan. Therefore, for an intersection operation d ∩ b to be valid, it requires that two conditions to be satisfied as was the case of union operation stated above.

iii) The Set-Difference '-' Operation:

The set-difference operation denoted by' -' It is a relational algebra operation that finds tuples that are in one relation but are not in another.

Find the customer id of all customers in the bank who have an account but not a loan

Example:

π customer_id (depositor) - π customer_id (borrower)

The resulting relation for this query lists the customer ids of all customers who have an account but not a loan. Therefore a difference operation d - b to be valid, it requires that two conditions to be satisfied as was case of union operation stated above.

iv) The Cross-product (or) Cartesian product 'X' Operation:

The Cartesian-product operation denoted by a cross 'X' It is a relational algebra operation which allows to combine information from who relations into one relation.

Assume that there are n1 tuple in borrower relation and n2 tuples in loan relation. Then, the result of this operation, denoted by r = borrower X loan, is a relation 'r' that includes all the tuples formed by each possible pair of tuples one from the borrower relation and one from the loan relation. Thus, 'r' is a large relation containing n1 * n2 tuples.

The drawback of the Cartesian-product is that same attribute name will repeat.

Example: Find the customer_id of all customers in the bank who have loan > 10,000.

π customer_id (σ borrower.loan_no= loan.loan_no ((σ borrower.loan_no= (borrower X loan)))

That is, get customer_id from borrower relation and loan_amount from loan relation. First, find Cartesian product of borrower X loan, so that the new relation contains both customer_id, loan_amoount with each combination. Now, select the amount, by

σ **bloan_ampunt >** 10000.

So, if any customer have taken the loan, then borrower.loan_no = loan.loan_no should be selected as their entries of loan_no matches in both relation.

The Joins "⋈" Operation:

The join operation, denoted by joins '⋈'. It is a relational algebra operation, which is used to combine

(join) two relations like Cartesian-product but finally removes duplicate attributes and makes the operations (selection, projection, ..) very simple. In simple words, we can say that join connects relations on columns containing comparable information.

There are three types of joins,

i) Natural Join

ii) Outer Join

iii) Theta Join (or) Conditional Join

i) Natural Join:

The natural join is a binary operation that allows us to combine two different relations into one relation and makes the same column in two different relations into only one-column in the resulting relation. Suppose we have relations with following schemas, which contain data on full-time employees.

employee (emp_name, street, city) and.

employee_works(emp_name, branch_name, salary)

The relations are,

employee relation

emp_name	street	city
Coyote	Town	Hollywood
Rabbit	Tunnel	Carrotville
Smith	Revolver	Vally
Williams	Seaview	Seattle

emp_name	branch_name	salary
Coyote	Mesa	15000
Rabbit	Mesa	12000
Gates	Redmond	25000
Williams	Redmond	23000

employee_works relation

If we want to generate a single relation with all the information (emp_name, street, city, branch_name and salary) about full-time employees. Then, a possible approach would be to use the natural-join operation as follows,

Employee ⋈ employee_works

The result of this expression is the relation,

emp_name	street	city	branch_name	salary
Coyote	Town	Hollywood	Mesa	15000
Rabbit	Tunnel	Carrotville	Mesa	12000
Williams	Seaview	Seattle	Redmond	23000

result of Natural join

We have lost street and city information about Smith, since tuples describing smith is absent in employee_works. Similarly, we have lost branch_name and salary information about Gates, since the tuple describing Gates is absent from the employee relation. Now, we can easily perform select or reject query on new join relation.

iii) Theta Join (or) Condition join:

The theta join operation, denoted by symbol "⋈". It is an extension to the natural join operation that combines two relations into one relation with a selection condition (⋈).

The theta join operation is expressed as employee ⋈ salary < 19000 employee_works and the resulting is as follows,

Employee ⋈ salary > 20000 employee_works

There are two tuples selected because their salary greater than 20000 (salary > 20000).

The result of theta join as follows, The relations are,

emp_name	street	city
Coyote	Town	Hollywood
Rabbit	Tunnel	Carrotville
Smith	Revolver	Valley
Williams	Seaview	Seattle

emp_name	branch_name	salary
Coyote	Mesa	15000
Rabbit	Mesa	12000
Gates	Redmond	25000
Williams	Redmond	23000

The result of this expression is the relation,

emp_name	street	City	branch_name	salary
Gates	null	Null	Redmond	25000
Williams	Seaview	Seattle	Redmond	23000

result of Theta Join (or) Condition Join

The Division "÷" Operation:

The division operation, denoted by **"÷"**, is a relational algebra operation that creates a new relation by selecting the rows in one relation that does not match rows in another relation.

Let, Relation A is (x1, x2… xn, y1, y2… ym) and

Relation B is (y1, y2… ym),

Where, y1, y2… ym tuples are common to the both relations A and B with same domain

compulsory.

Then, A ÷ B = new relation with x1, x2… xn tuples. Relation A and B represents the dividend and devisor respectively. A tuple't' is in a ÷ b, if and only if two conditions are to be satisfied,

t is in π A-B (r)

for every tuple tb in B, there is a tuple ta in A satisfying the following two things,

1. ta[B] = tb[B]

2. ta[A-B] = t

Relational Calculus

Relational calculus is an alternative to relational algebra. In contrast to the algebra, which is procedural, the relational calculus is non-procedural or declarative.

It allows user to describe the set of answers without showing procedure about how they should be computed. Relational calculus has a big influence on the design of commercial query languages such as SQL and QBE (Query-by Example).

Relational calculus are of two types,

1. Tuple Relational Calculus (TRC)
2. Domain Relational Calculus (DRC)

Variables in TRC take tuples (rows) as values and TRC
Variables in DRC takes fields (attributes) as values and DRC had strong influence on QBE.

i) Tuple Relational Calculus (TRC):

The tuple relational calculus, is a non-procedural query language because it gives the desired information without showing procedure about how they should be computed.

A query in Tuple Relational Calculus (TRC) is expressed as **{T | p(T) }**

Where, T - tuple variable,
P (T) - 'p' is a condition or formula that is true for't'.

In addition to that we use,

T[A] - to denote the value of tuple t on attribute A and
T ϵ r - to denote that tuple t is in relation r.

Examples:

1) Find all loan details in loan relation.

{t | t ϵ loan }

This query gives all loan details such as loan_no, loan_date, loan_amt for all loan table in a bank.

2) Find all loan details for loan amount over 100000 in loan relation.

{ t | t ϵ loan ∧ t [loan_amt] > 100000 }

This query gives all loan details such as loan_no, loan_date, loan_amt for all loan over 100000 in a loan table in a bank.

ii) Domain Relational Calculus (DRC):

A Duple Relational Calculus (DRC) is a variable that comes in the range of the values of domain (data types) of some columns (attributes).
A Domain Relational Calculus query has the form,

{< x1, x2… xn > | p (< x1, x2… xn >) }

Where, each xi is either a domain variable or a constant and p(< x1, x2, …., xn >) denotes a DRC formula.
A DRC formula is defined in a manner that is very similar to the definition of a TRC formula. The main difference is that the variables are domain variables.

Examples:

1) Find all loan details in loan relation.

{< N, D, a > | <N, D, A > ϵ loan}

This query gives all loan details such as loan_no, loan_date, loan_amt for all loan table in a bank. Each column is represented with an initials such as N- loan_no, D – loan_date, A – loan_amt. The condition < N, D, A > ϵ loan ensures that the domain variables N, D, A are restricted to the column domain.