DATABASE MANAGEMENT SYSTEM

NOTES

SERIES -3

LinkedIn ATUL KUMAR
7777888822

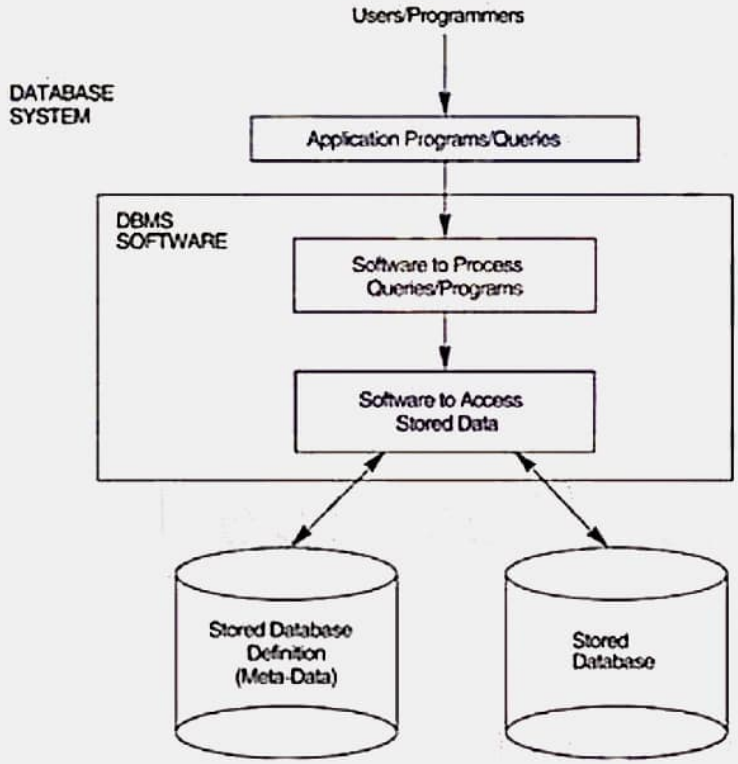
Books Referred: (1) Silberschatz, Korth, Sudarshan –Database System Concepts, 4th Edition
(2) Bipin C. Desai – An Introduction to Database Systems.
(3) Elmasri & Navathe – Fundamentals of Database Systems, 4th Edition

Database:

- ✓ It is a collection of interrelated data of an enterprise in a particular subject.
- ✓ It is a collection of data in an organized manner in a persistence media so that storing and retrieving data will be easier.
- ✓ Size of the database is not fixed & it can vary.
- ✓ A database can be generated and maintained manually or by computer.
- ✓ *Example:* Dictionary, Telephone Directory, Library Card Catalog.

DBMS:

- ✓ It is a collection of programs (software), which is used to create manipulate (insert, retrieve, delete, update) data in a database.
- ✓ It also facilitates the process of defining, constructing, manipulating and sharing databases among various users and applications.
- ✓ A DBMS is a 4th generation language.
- ✓ Example: Oracle, SQL Server, MS-Access



Application of Database System:

- ✓ **Banking:** For customer information, accounts, loans, transaction etc.
- ✓ **Airlines:** For reservation and schedule information.
- ✓ **Universities:** For student information, examination information, department information etc.
- ✓ **Credit Card Transactions:** For purchase through credit card and generation of monthly statement.
- ✓ **Telecommunication:** Keeping records of calls made, generating monthly bills, storing information about monthly bills, storing information about communication network etc.
- ✓ **Finance:** Storing information about sales & purchase of shares, stocks, bonds etc.
- ✓ **Human resource:** Storing information about employees, salaries, benefits etc.
- ✓ **Internet:** Storing user-id, password etc in a mailing system.

Database Systems versus File Systems / Disadvantages of File Systems:

- ✓ **Data Redundancy:** Same information may be duplicated in several files. For example the information of a particular customer may appear in two different files one storing saving account records and another storing loan account records.

  In database oriented approach duplicate records are avoided because here relationship is used.
- ✓ **Data inconsistency:** Data redundancy may leads to data inconsistency. For example any changes to records of one file will not affect the records of other file.

  In database oriented approach no redundancy of data so no chance of data inconsistency.
- ✓ **Difficulty in accessing data:** Conventional file-processing environments do not allow needed data to be retrieved in a convenient and efficient manner. For example to find out the list of customers who live in a particular area we have to search in multiple files separately.

  Database system can handle any type of query & produces the required information conveniently.
- ✓ **Data isolation:** Data are scattered in various files & the formats of these data may be different, so retrieving the appropriate data is difficult.

  In database system data are stored in one place. So data will be accessed in a faster manner.
- ✓ **Data Integrity Problems:** Data are stored in different places in different formats. So while combining to get the required data integrity (uniformity) problems are created.

  In database system as relationship is used there is no chance of data integrity problem.
- ✓ **Atomicity Problems:** In case of system failure data should be stored to the consistent state that existed prior to the failure. This is called atomicity. It is difficult to ensure atomicity in file-oriented approach.

  In database system atomicity can be ensured easily by writing codes for that.
- ✓ **Concurrent-access anomalies:** When multiple users access the same data simultaneously then there is a chance of data inconsistency.

  In database system the system guard against this possibility by some form of supervision.
- ✓ **Security:** No security in file oriented approach. Any user can access all data.

  In database oriented approach access restricted. Data available for certain group only.
- ✓ **Standards:** No standard is maintained in file-oriented approach.

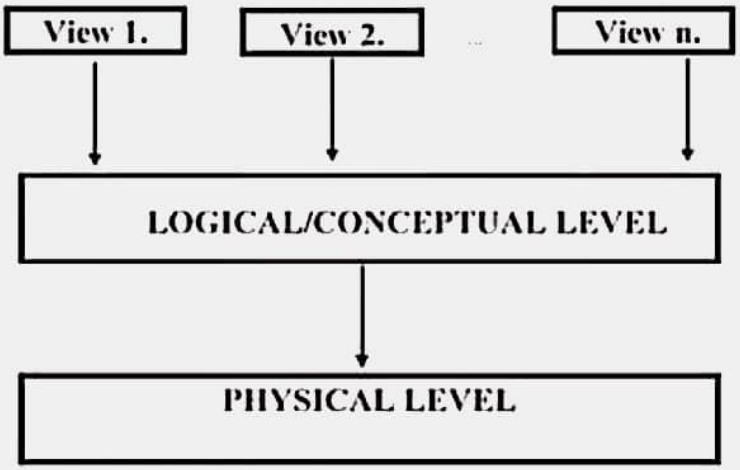
  In database-oriented approach valid data are stored only.

View of Data

Data Abstraction / Database System Architecture / ANSI Sparc Architecture:

To retrieve data efficiently complex data structures are used to represent the data in the database. As the users don't understand the details about the system, developer hides the complexity from users through the following levels of abstraction:

- ✓ **Physical Level / Internal Level:** It is the lowest level of abstraction, which describes *how* the data are actually stored (physical storage structure) and describes the data structures and access methods to be used by the database.
- ✓ **Logical Level / Conceptual Level:** This is the next level of abstraction, which specifies *what* data (database entities) are stored in the database and what relationships exist among those data. To decide what information to keep in the database DBA uses the logical level of abstraction.
- ✓ **View Level / External Level:** This is the highest level of abstraction, which describes part of the entire database of interest to a particular user group. The view level of abstraction exists to simplify their interaction with the system. The system may provide many views for the same database.



The three levels of data abstraction

Example: Employee database

- ✓ **Physical Level / Internal Level:** At this level an employee record can be described as a block of consecutive storage locations consisting of how much byte.
- ✓ **Logical Level / Conceptual Level:** At this level each record is described by a type definition and the interrelationship of these record types is also defined.
- ✓ **View Level / External Level:** At this level the users see a set of application programs, which hides the details of the data types. The user is allowed to access certain part of the database not the entire database.

Instances / Snapshot / Extension / Database State: The collection of information stored in the database at a particular moment is called an instance of the database.

Schema / Intension: The overall design of the database is called the database schema. Schemas are not changed frequently. The Physical schema describes the database design at physical level. Logical schema describes the database design at logical level. Subschema describes different views of the database at view level.

Data Independence: It is the ability to change the schema of one level of a database system without having to change the schema at the next higher level.

1. **Logical data independence:** It is the ability to change the conceptual schema without having to change external schema or application program.
   Example: Addition of records or data items, changing constraints, removing records or data items etc.
2. **Physical Data Independence:** It is the ability to change the internal schema without having to change conceptual schema and so external schema.
   Example: Creating additional access structures, providing an access path etc.

Advantages of DBMS:

- ✓ **Reduction of Redundancies:** Centralized control of data avoids duplication of data, which decreases the storage cost and eliminates data inconsistencies.
- ✓ **Shared Data:** DBMS allows data sharing between no of users and programs.
- ✓ **Data Integrity:** DBMS uses no of checks before storing any data in the database. By doing this DBMS ensures the data are correct and consistent.
- ✓ **Security:** DBMS ensures that proper access procedures are followed, including proper authentication schemes for access to the DBMS and additional checks before permitting access to sensitive data.
- ✓ **Data Independence:** DBMS allows dynamic changes at one level of database without affecting other levels.

Disadvantages of DBMS:

- ✓ **Cost:** The cost of purchasing or developing the software, cost of upgrading the hardware to store the DBMS and allow it to run and the cost of migration from a separate application environment to an integrate one is very high.
- ✓ **Complexity of Backup and recovery:** Backup & recovery operations are fairly complex in a DBMS environment.
- ✓ **Problems with centralization:** As the data is accessible from a single source failure of the central system disrupts the entire system. But that can be avoided by using distributed database.

Database Languages:

A database system provides a data definition language, which specifies the database schema, and a data manipulation language, which expresses database queries and updates.

1. **Data-Definition Language(DDL):**
   - ✓ It specifies the database schema.
   - ✓ For example to create a table "Student" we use the following SQL statement:
     create table Student (Roll **integer**, Name **char**(10))
   - ✓ After executing this statement it creates a table Student and updates a special table called **data dictionary** or **data directory**.
   - ✓ To specify the storage structure and access methods used by the database system we use a special type of DDL called **data storage and definition language**.
   - ✓ The data values stored in the database must satisfy certain **consistency constraints** specified by the DDL.
2. **Data-Manipulation Language (DML):**
   - ✓ Data manipulation includes retrieval of data stored in the database, insertion of new data into the database, deletion of data from the database and modification of data stored in the database.
   - ✓ Data manipulation language enables users to access or manipulate data as organized by different data model.
   - ✓ DMLs are of two types:
     - **Procedural DMLs:** It requires a user to specify what data are needed and how to get those data.
     - **Nonprocedural DMLs or Declarative DMLs:** It requires a user to specify what data are needed without specifying how to get those data.
   - ✓ A **query** is a statement requesting the retrieval of information. The portion of the DML that involves information retrieval is called **query language**.
   - ✓ Query example:
     select Student.Name **from** Student **where** Student.Roll=5
3. **Database Access from Application Programs:**
   - ✓ Application Programs are programs that are used to interact with database. Application programs are written in a host language like C, C++, Cobol, Java etc

- ✓ To access the database, DML statements need to be executed from the host language. It can be done in two ways:
    - By providing an API (set of procedures) that can be used to send the DDL and DML statements to the database and receive the result.
    - By extending the host language syntax to embed DML calls within the host language program.

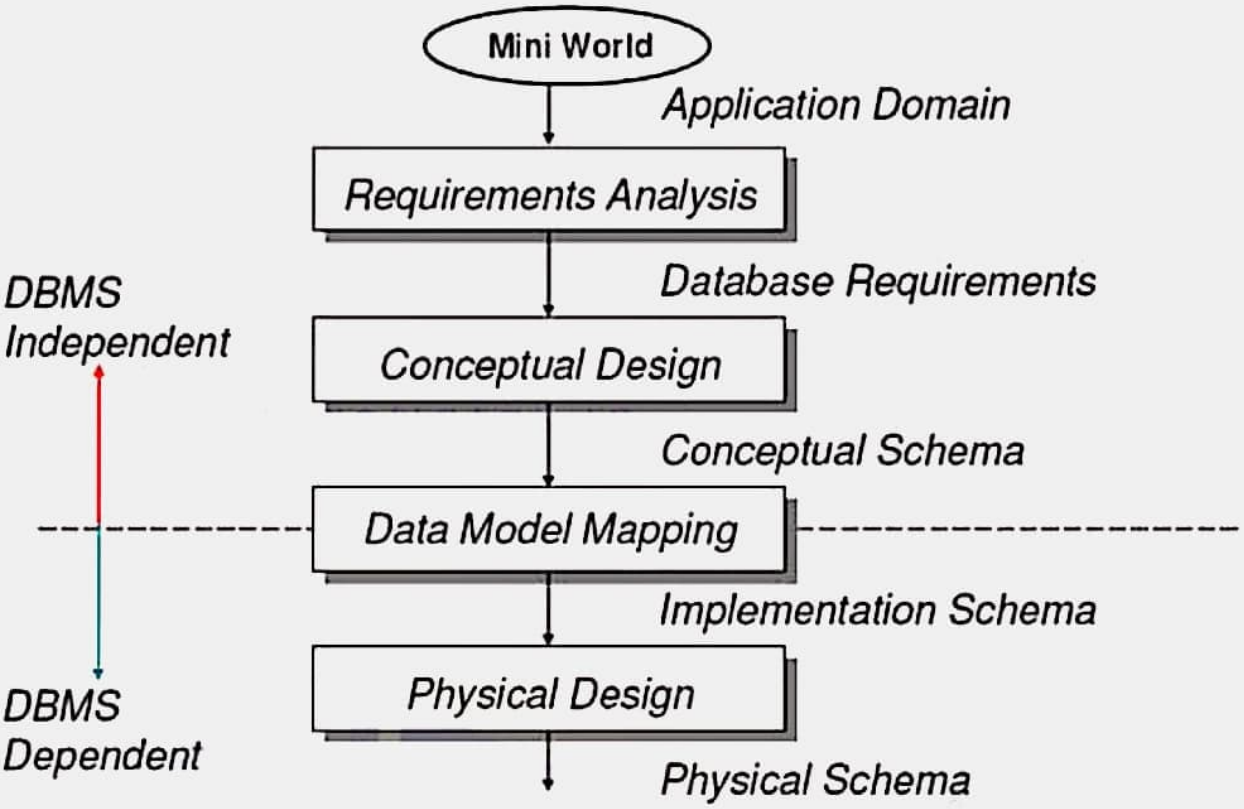
Database Users and Administrators:

1. **Database Users and User Interfaces:** Depending on the different types of uses there different types of users and for each type of user different types of interfaces have been designed.
    - ✓ **Naive Users / Parametric Users:** These are unsophisticated users who interact with the system by invoking one of the application programs that have been written previously. They also read reports from the database. Example: bank teller, a person checking his balance through Internet.
    - ✓ **Application Programmers:** These are computer professionals who write application programs to develop user interfaces. Using ***Rapid application development (RAD)*** tool application programmers construct forms and reports rapidly without writing programs.
    - ✓ **Sophisticated Users:** These users interact with the system without writing programs. They form their requests in a database query language and submit it to the query processor. Example: Engineers, Scientists, Business Analysts who need some summarized data. They use some of the tools like Online Analytical Processing (OLAP), Data Mining that views them the summarized data in different ways.
    - ✓ **Specialized Users:** These are sophisticated users who write specialized database applications that don't fit into the traditional data processing framework. Example: CAD systems, Knowledge-base expert systems, environment modeling system etc.
    - ✓ **Casual End Users:** These users occasionally access the database, but they may need different information each time. They use a sophisticated database query language to specify their requests and are typically middle or high-level managers or other occasional browsers.
2. **Database Administrators (DBA):** DBA has the central control over the system. The function / role of the DBA include:
    - ✓ **Schema definition:** The DBA creates the original database schema by a set of data definition statements in the DDL.
    - ✓ **Storage structure and access-method definition:** DBA decides what structure to be used to store the data and what method to be used to access that.
    - ✓ **Schema and physical-organization modification:** According to the need of the organization the DBA carries out changes to the schema and physical organization to improve the performance.
    - ✓ **Granting of authorization for data access:** The DBA administrator decides which user will access which part of the database.
    - ✓ **Routine maintenance:** DBA does some of the routine maintenance like periodically backing up the database, ensure that enough free disk space available for normal operations, monitors jobs running on the database etc.

Data models

- ✓ It is a collection of concepts that can be used to describe the structure of database and the technique to access the data according to the DBMS.
- ✓ It is a collection of conceptual tools for describing data, data relationships, data semantics (meaning) and consistency constraints.
- ✓ It is an integrated collection of concepts for describing data, relationships between data, and constraints on the data.
- ✓ It is a description of a container for data (data dictionary) a methodology for storing and retrieving data from that container (query language).

Phases of Database Design (Use of High-Level Data Model for Database Design):



The database is designed in the following phases:

- ✓ **Requirement Collection and Analysis:** In this step the database designer interacts with no of users to understand and document their data requirement. It is useful to specify functional requirements of the application.
- ✓ **Conceptual Design:** In this step a conceptual schema is created for the database using high-level data model. It includes the detailed description about entity types, relationships and constraints. The designer should ensure that the conceptual schema meets all the identified functional requirements. In this phase the goal is to arrive at an understanding of the principal data sources and data elements of interest to the business or organization, and the relationships between the data sources. The tool used here is the *Entity-Relationship Diagram* (ERD).
- ✓ **Logical Design / Data Model Mapping:** In this step the actual implementation of the database is done using a commercial DBMS. They use relational or object-relational data model. That means in this step the conceptual schema is transformed from high-level data model into the implementation data model. The tool used is relational schema.
- ✓ **Physical Design:** In this step the internal storage structures, indexes, access paths and file organizations for the database files are specified.

During the activities of the above steps application programs are designed and implemented as database transactions corresponding to the high-level transaction specification

Types of Data Model:

The following are some of the types of data models:

1. Object –Based Logical Models:
   a. Entity-Relationship Model
   b. Object-Oriented Model
   c. Semantic Data Model
   d. Functional Data Model
2. Record-Based logical models:
   a. Relational Model

b. Network Model
c. Hierarchical Model

3. Physical Data Models

The Entity-Relationship Model:

- ✓ ER Model is based on **objects**, called **entities** and of relationships among these objects.
- ✓ The E-R model is very useful in mapping the meaning and interactions of real-world enterprises onto a conceptual schema.

Entity Set:

- ✓ An entity is a "**thing**" or "**object**" in the real world that is distinguishable from other objects. Example: person, employee, account, loan etc.
- ✓ Same type of entities that share the same properties or attributes forms an **entity set**. Example: customer, loan etc.
- ✓ Individual entities that form an entity set are called **extension** of the entity set.
- ✓ Entity sets may not be disjoint.
- ✓ **Entity Type** describes the name of the entity & its attributes.
- ✓ Each entity is described by a set of **attributes**. Example: eno, salary, dob describes an employee, loan_no an amount describes a loan etc.
- ✓ Each entity has a value for each of its attributes.
- ✓ For each attribute there is a set of permitted values, called the **domain** or **value set** of that attribute.
- ✓ **Types of attributes**:
  - **Simple and Composite attributes**: Attributes, which can't be divided into subparts, are simple attributes and which can be divided into subparts are composite attributes. Example: Balance, Accno are simple attributes whereas Name, Address are composite attributes.
  - **Single-valued and multi-valued attributes**: The attribute for which we have a single value for each entity is called single-valued attribute and the attributes for which we have a set of values for a particular entity is called multi-valued attribute. Example: loan-number, age are single-valued attributes whereas phone, dependant-name, qualification are multi-valued attributes.
  - **Derived attributes**: The value of the attribute, which is derived from the value of other, related attribute is called derived attribute. Example: age derived from date-of-birth; Experience derived from date_of_joining etc. The value of the derived attribute is not stored but is computed when required.
- ✓ **Null Value:** An attribute takes a null value when an entity does not have a value for it (Example: middle-name, apartment-no) or if the value is not known (Example: first-name).

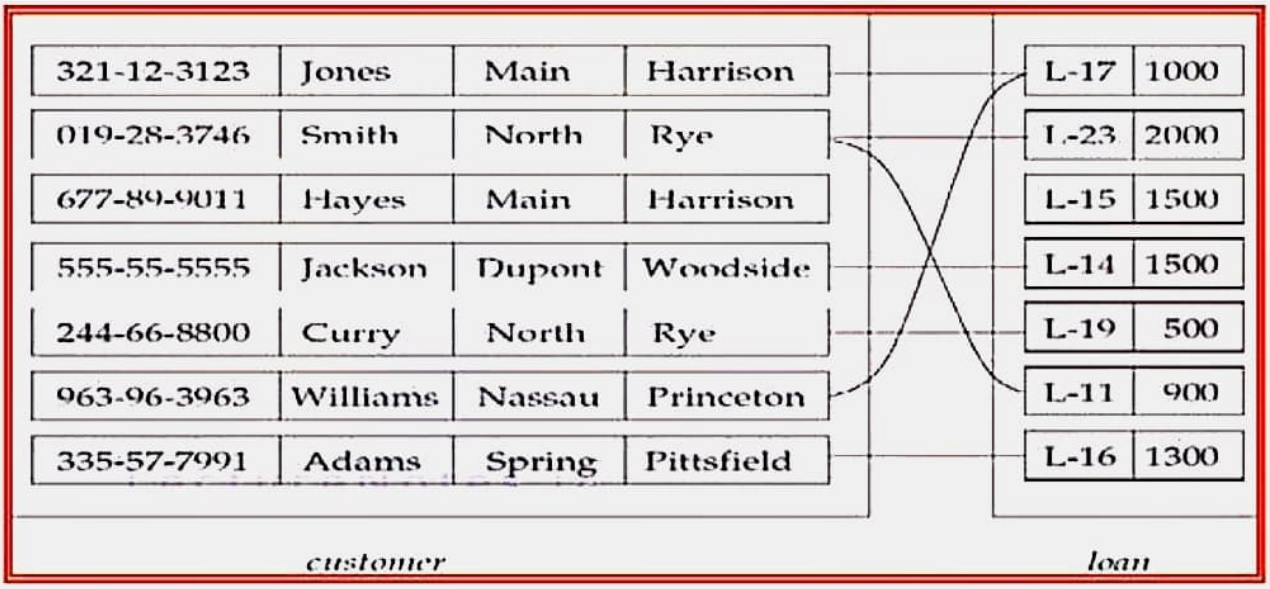
Relationship Set:

- ✓ A **relationship** is an association among several entities.
- ✓ Example: depositor associates customer and account entities.

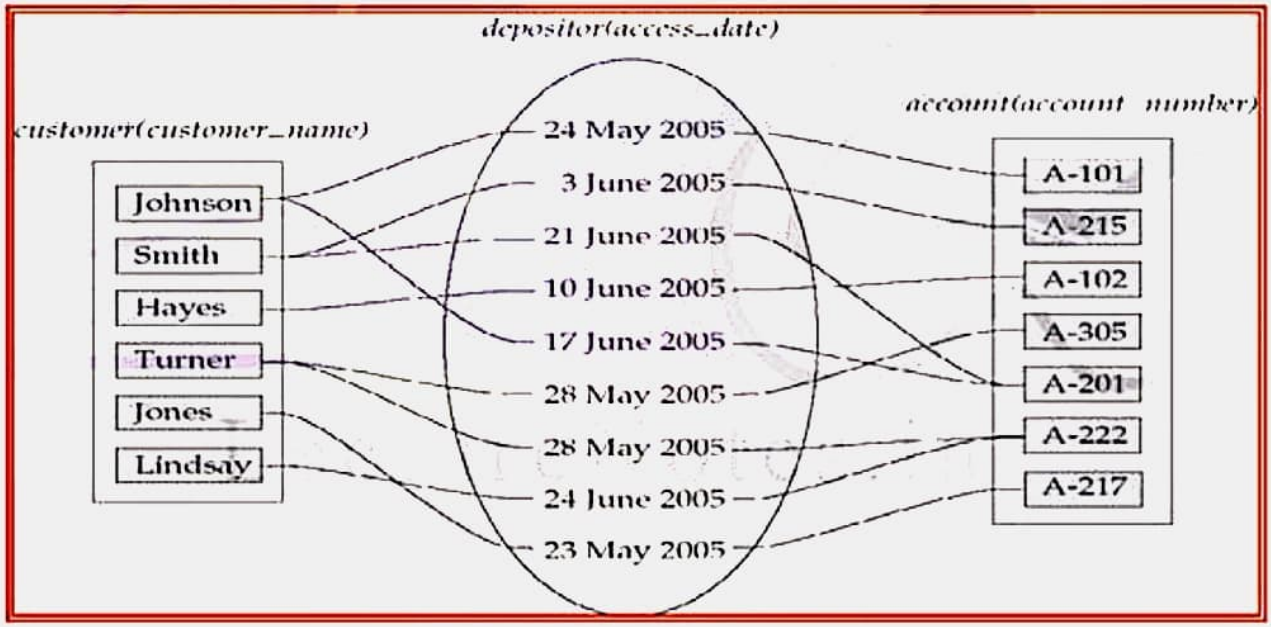
Hayes	*depositor*	A-102
customer entity	relationship set	*account* entity

(Hayes, A-102) ∈ *depositor*

- ✓ A **relationship set** is a set of relationships of the same type.
- ✓ It is a mathematical relationship on more than one entity sets. If $E_1, E_2, \ldots, E_n$ are entity sets then a relationship set r is a subset of $\{(e_1, e_2, \ldots, e_3) \mid e_1 \in E_1, e_2 \in E_2, \ldots, e_n \in E_n\}$ where $(e_1, e_2, \ldots, e_3)$ is a relationship.
- ✓ The entity sets $E_1, E_2, \ldots, E_n$ participate in relationship R.
- ✓ Example: Borrower is the relationship between two entity sets Customer & Loan.



- ✓ **Descriptive attributes:** A relationship may have some new attributes which are not present in the entity sets are called descriptive attributes. It describes about the relationship set.
  Example: In the Depositor relationship set between Customer and Account entity sets access-date can be a descriptive attribute. Similarly in the Registration relationship set between Student and Course entity sets Registration-date can be a descriptive attribute.



- ✓ **Ternary relationship:** The relationship between three entity sets is known as a ternary relationship. Example: The relationship works-on between the entity sets employee, branch and job (title, level) can be a ternary relationship.
- ✓ **Degree of a relationship:** Number of entity sets that participate in a relationship set is known as the degree of a relationship.

Constraints

Mapping Cardinalities / Cardinality ratio:

For a binary relationship set R between entity sets A and B, the mapping cardinality must be one of the followings:

- ✓ **One to one:** An entity set A is associated with at most one entity in B and an entity in B is associated with at most one entity in A.

- ✓ **One to many:** An entity set A is associated with any no of entities in B with a possibility of zero and an entity in B is associated with at most one entity in A.
- ✓ **Many to one:** An entity set A is associated with at most one entity in B and an entity in B can be associated with any number of entities in A with a possibility of zero.
- ✓ **Many to many:** An entity set A is associated with any no of entity in B with a possibility of zero and an entity in B is associated with any no of entities in A with a possibility of zero.

Total Participation / Partial Participation:

- ✓ The participation of an entity set E in a relationship set R is said to be **total** if every entity in E participates in at least one relationship in R.
  - **Example:** Participation of loan in the relationship borrower is total.
- ✓ The participation of an entity set E in a relationship set R is said to be **partial** if only some entities in E participates in relationship R.
  - **Example:** Participation of customer in the relationship borrower is partial.

Keys:

- ✓ **Entity Set Keys:**
  - A key allows us to identify a set of attributes that make sufficient to distinguish entities from each other.
  - **Superkey:** It is a set of one or more attributes that, taken collectively, allow us to uniquely identify an entity in the entity set.
  - **Candidate key:** Minimal superkeys are called candidate keys. Candidate key is a superkey for which no proper subset is a superkey.
  - **Primary key:** Out of all possible candidate keys the database designer chooses one as primary key.
  - **Example:** Entity set- Student (Regd-no, SIC-no, Name,Mark)

    Superkey – Regd-no; SIC-no; Regd-no, SIC-no, Name; SIC-No, Name, Mark

    Candidate Key – Regd-no, SIC-no

    Primary Key – Regd-no
- ✓ **Relationship keys:**
  - The primary key of a relationship set allows us to distinguish among various relationships of the set.
  - If the relationship set R has no attributes associate with it. then the set of attributes

    Primary_key(E_1) U primary-key(E_2) U. . . U Primary-key(E_n)

    describes an individual relationship in set R.
  - If the relationship set R has attributes $a_1, a_2, \ldots, a_m$ associated with it, then the set of attributes

    Primary_key(E_1) U primary-key(E_2) U. . . U Primary-key(E_n) U { $a_1, a_2, \ldots, a_m$ }

    describes an individual relationship in set R.
  - In both the cases, the set of attributes

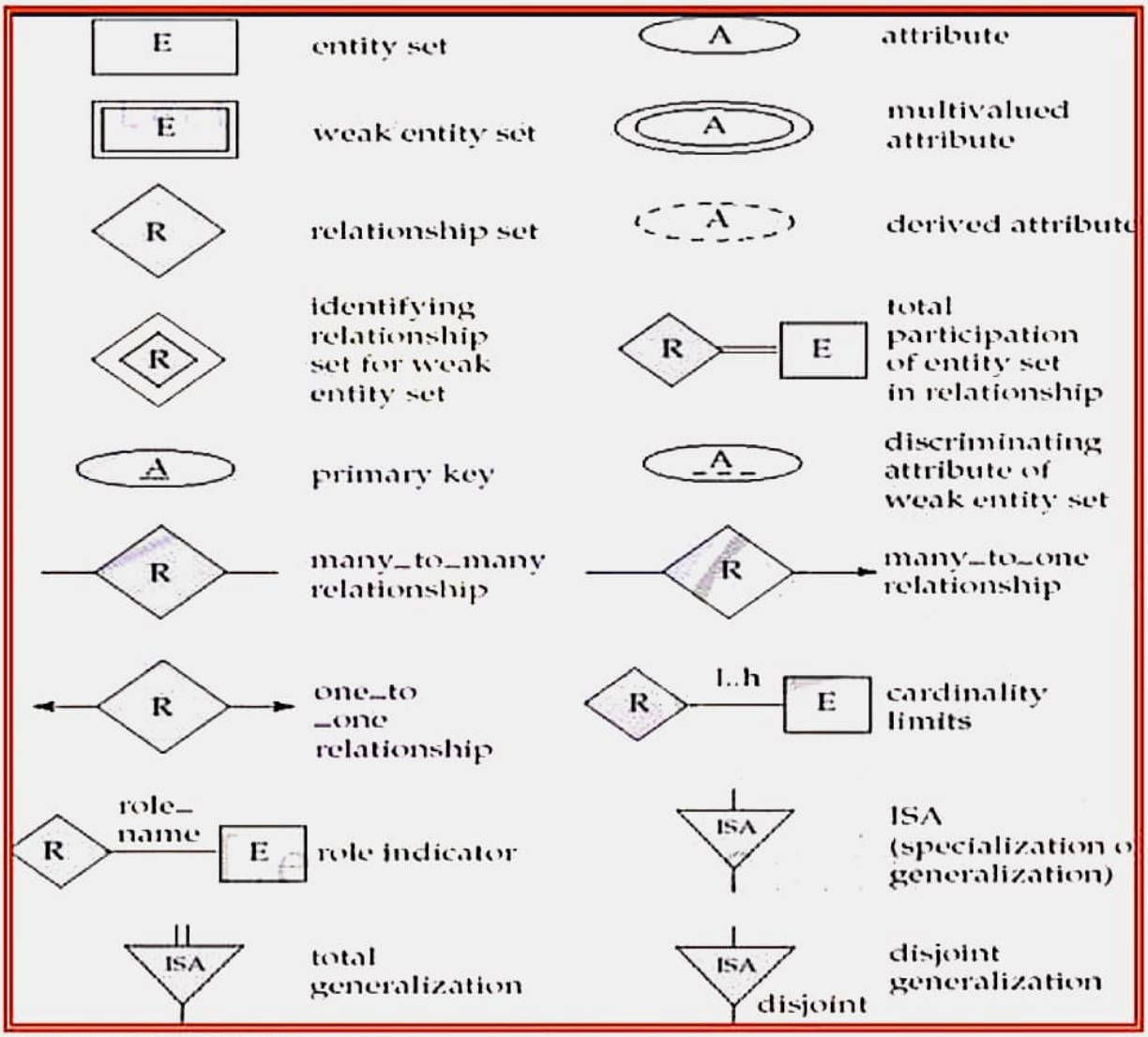
    Primary_key(E_1) U primary-key(E_2) U. . . U Primary-key(E_n)

    Forms a super key.
  - The structure of the primary key for the relationship set depends on the mapping cardinality of the relationship set.
  - **Example:** Consider a relationship set Depositor with attribute access-date between two entity sets Customer and Account. The primary key of Depositor will be:
    - **Case 1:** Primary Key (Customer) U Primary Key (Account) if the relationship is **many to many.**

- **Case 2:** Primary Key (Customer) if the relationship is **many to one** from Customer to Account.
- **Case 3:** Primary Key(Account) if the relationship is **many to one** from Account to Customer.
- **Case 4:** Primary Key (Customer) or Primary Key(Account) if the relationship is **one to one.**

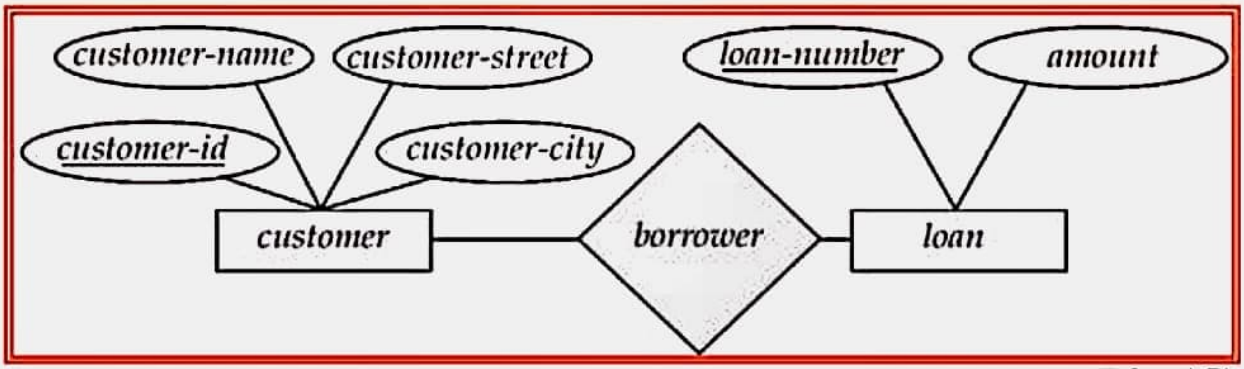
Entity-Relationship Diagram:

The major symbols used in the E-R Diagram are:

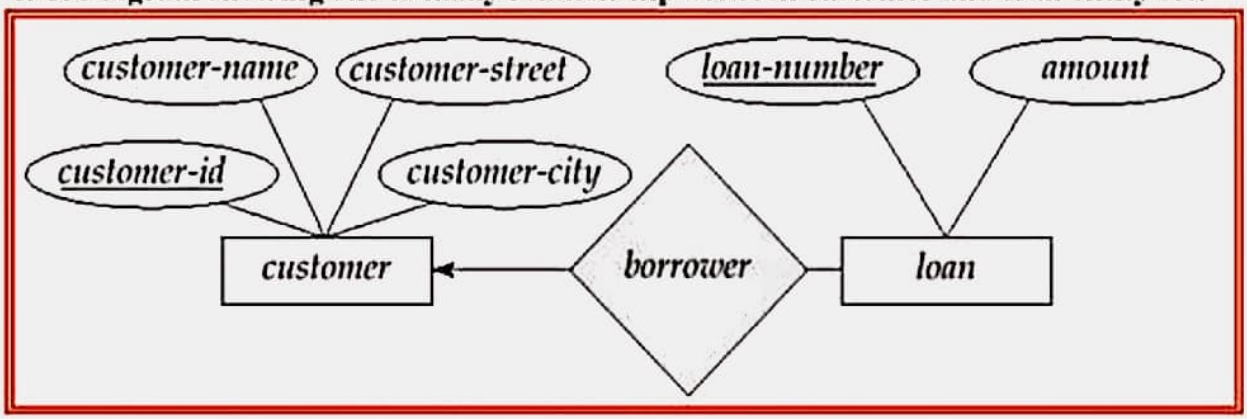


E-R Diagram Examples:

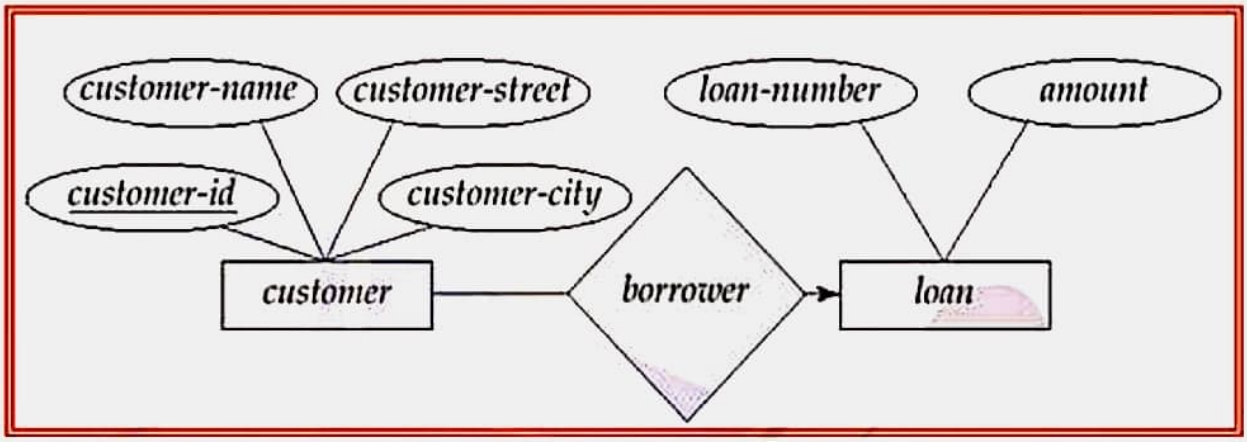
1. **E-R Diagram showing many to many relationship between customer and loan entity set:**



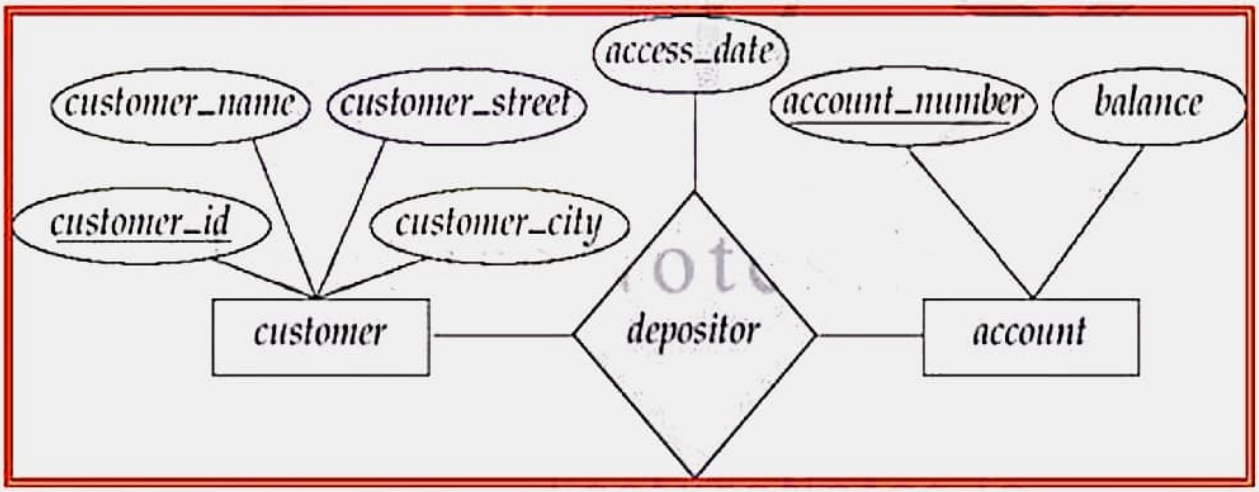
2. **E-R Diagram showing one to many relationship between customer and loan entity set:**



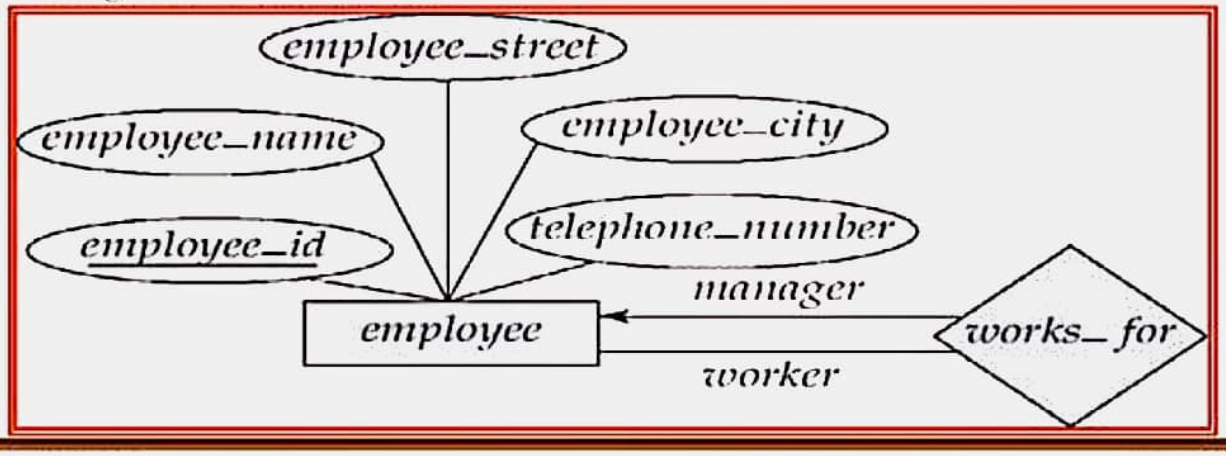
3. **E-R Diagram showing many to one relationship between customer and loan entity set:**



4. **E-R Diagram showing descriptive attributes associated with a relationship set:**

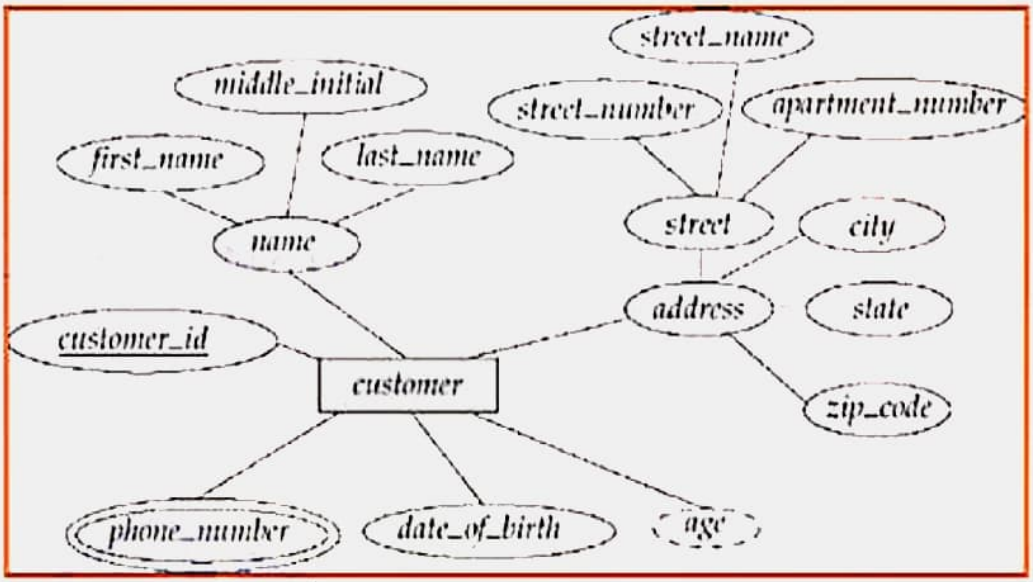


5. **E-R Diagram with role indicator:**

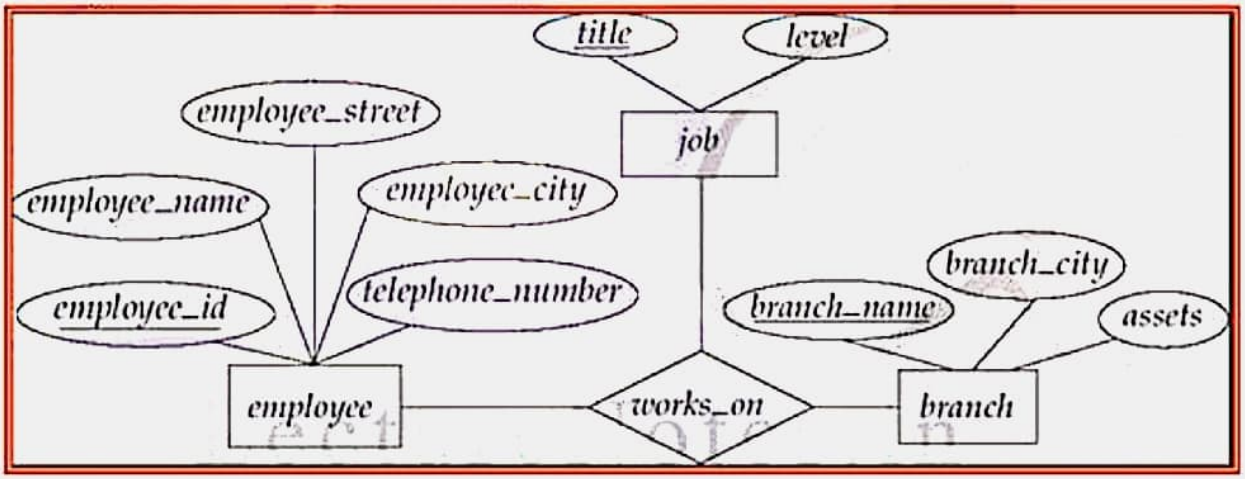


- ✓ The labels "manager" and "worker" are called **roles**; they specify how employee entities interact via the works_for relationship set.
- ✓ Roles are indicated in E-R diagrams by labeling the lines that connect diamonds to rectangles.
- ✓ Role labels are optional, and are used to clarify semantics of the relationship.

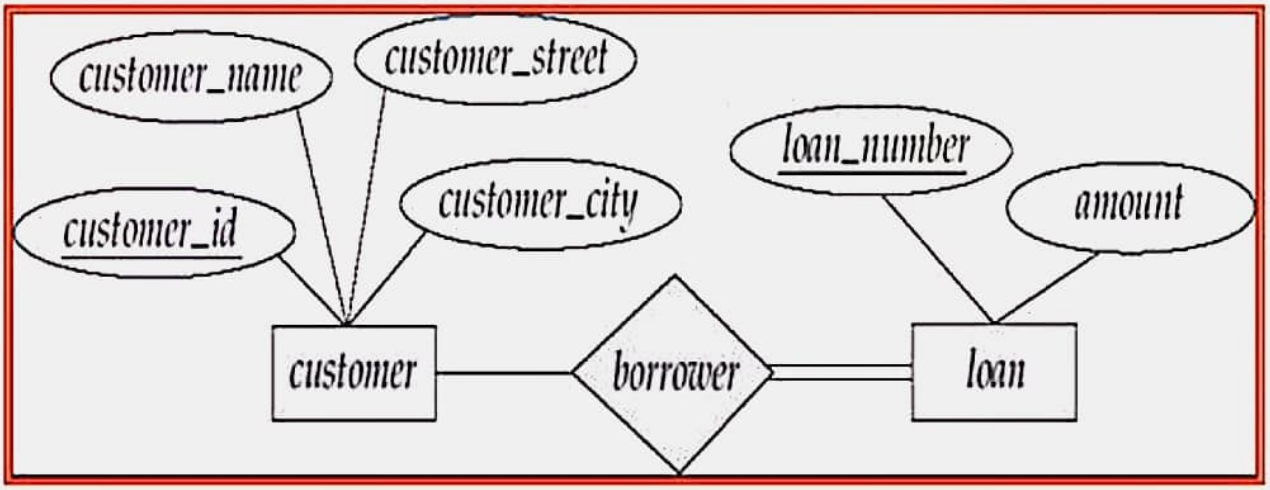
6. **E-R Diagram showing composite, multi-valued and derived attributes:**



7. **E-R Diagram with ternary relationship:**



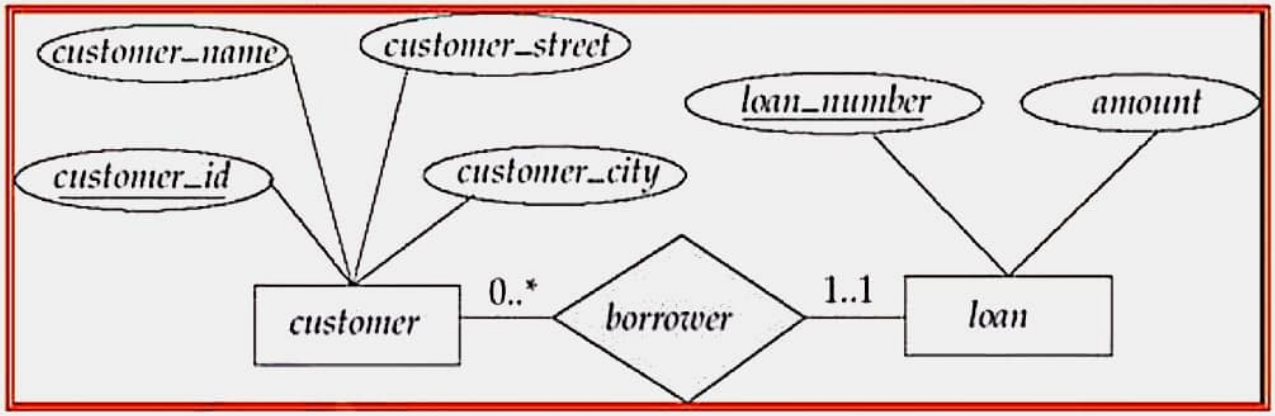
8. **E-R Diagram showing total participation:**



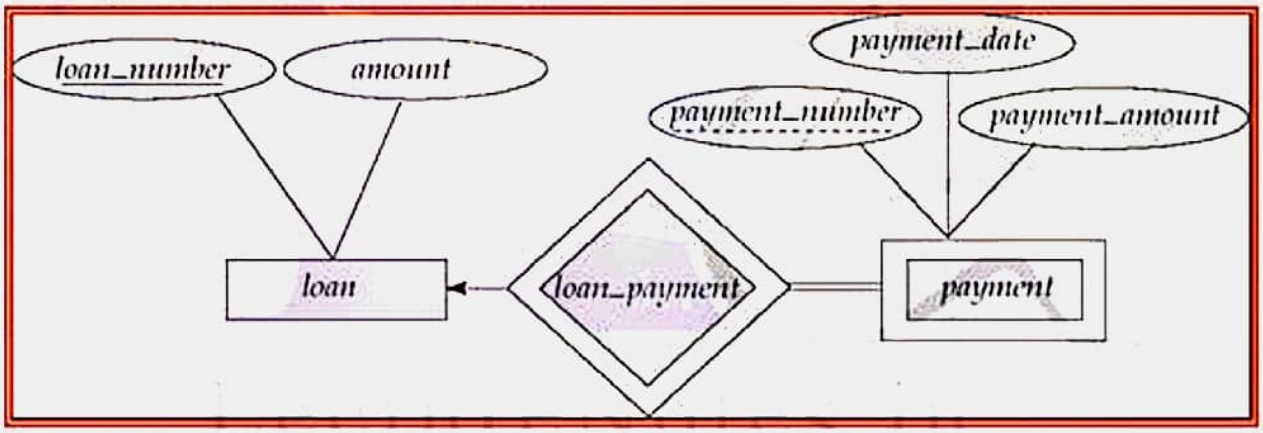
- ✓ **Total participation** (indicated by double line): every entity in the entity set participates in at least one relationship in the relationship set
  - Example: participation of loan in borrower is total
    - every loan must have a customer associated to it via borrower
- ✓ **Partial participation**: some entities may not participate in any relationship in the relationship set
  - Example: participation of customer in borrower is partial

9. E-R Diagram showing alternative notation for cardinality limits:

- ✓ An edge between an entity set and a binary relationship set can have an associated minimum and maximum cardinality in the form *l*..*h* where *l* is the minimum and *h* is the maximum cardinality.



10. E-R Diagram showing Weak Entity Sets

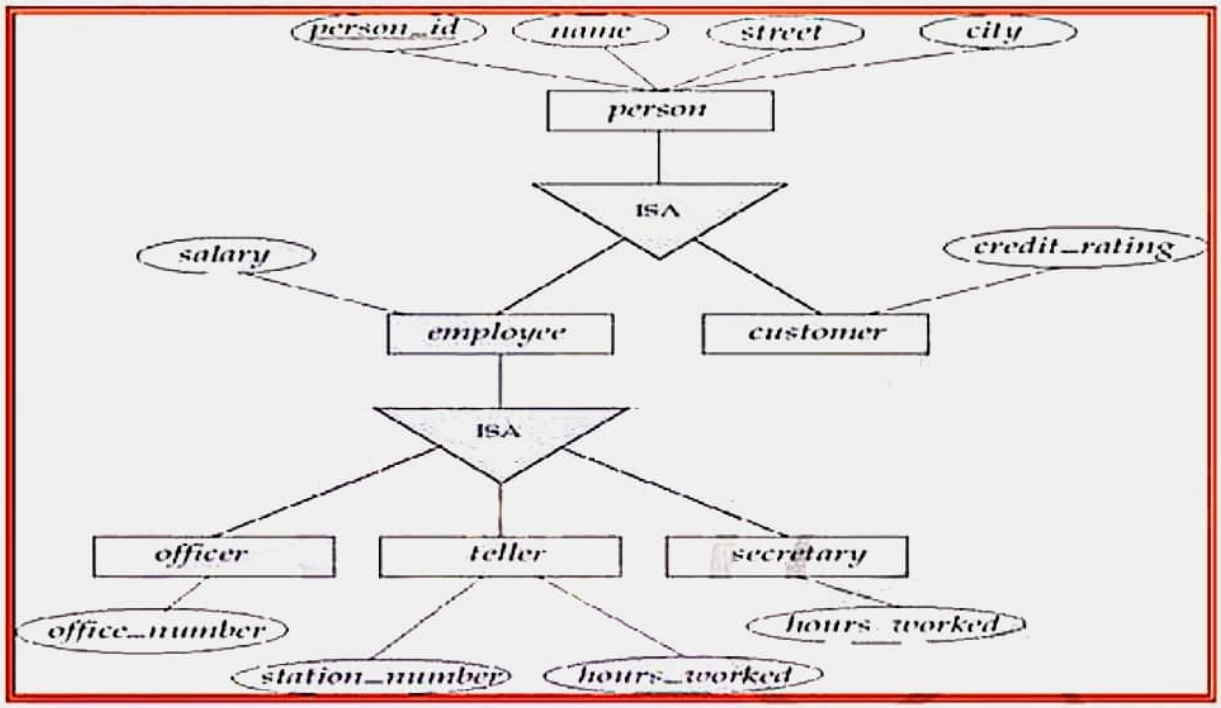


- ✓ An entity set that does not have a primary key is referred to as a **weak entity set.**
- ✓ The existence of a weak entity set depends on the existence of a **identifying or owner entity set.**
- ✓ The relationship associating the weak entity set with the identifying entity set is called **identifying relationship.**
- ✓ It must relate to the identifying entity set via a total, one-to-many relationship set from the identifying to the weak entity set
- ✓ Identifying relationship is shown using a double diamond.
- ✓ The **discriminator** (*or partial key*) of a weak entity set is the set of attributes that distinguishes among all the entities of a weak entity set that depends on one particular strong entity.
- ✓ The primary key of a weak entity set is formed by the primary key of the strong entity set on which the weak entity set is existence dependent, plus the weak entity set's discriminator.
- ✓ Weak entity set is shown by double rectangles.
- ✓ The discriminator of a weak entity set is shown with a dashed line.
- ✓ payment_number – discriminator of the *payment* entity set since for each loan a payment number uniquely identifies one single payment for that loan.
- ✓ Primary key for *payment* – (*loan_number, payment_number*)
- ✓ An identifying relationship should not have any descriptive attributes.

Extended E-R Features

Specialization:

- ✓ The process of designating sub groupings within an entity set is called **specialization**. It is a top-down design process
- ✓ **Example:** Specialization of person allows us to distinguish persons according to whether they are employees or customers. Specialization of account creates two entity sets saving-account and checking-account.
- ✓ The specialized entity sets will have all the attributes of original entity set with some additional attributes.
- ✓ In E-R diagram specialization is represented by a triangle component labeled **ISA.**
- ✓ The **ISA** relationship is referred as super class-subclass relationship.



Attribute inheritance: a lower-level entity set inherits all the attributes and relationship participation of the higher-level entity set to which it is linked.

Generalization:

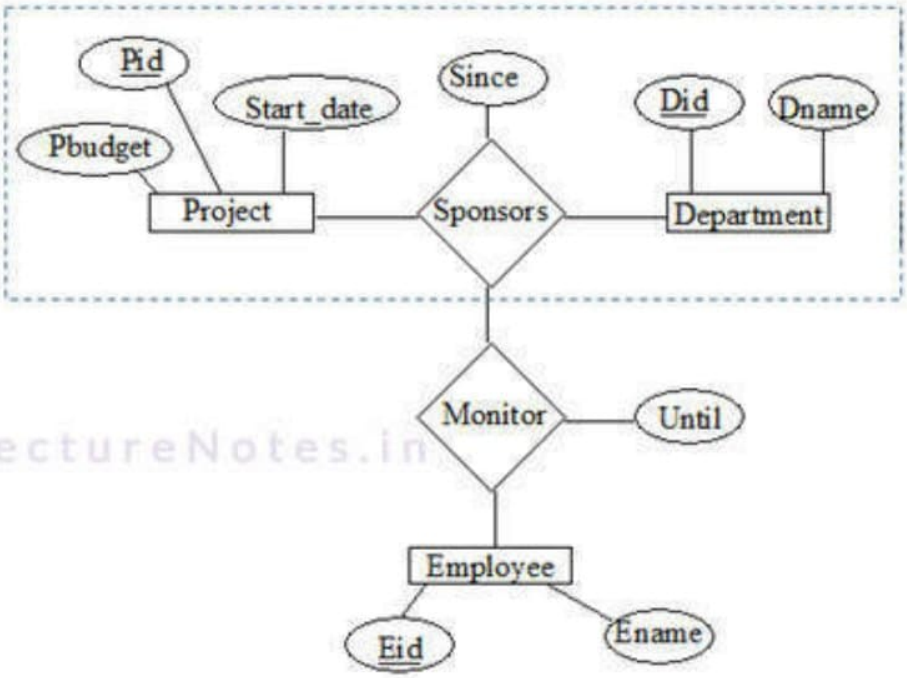
- ✓ It is a **bottom-up** design process that combines a number of entity sets that share the same features into a higher-level entity set.
- ✓ Specialization and generalization are simple inversions of each other; they are represented in an E-R diagram in the same way.
- ✓ The terms specialization and generalization are used interchangeably.

Constraints of Generalization / Specialization:

- ✓ Constraint on which entities can be members of a given lower-level entity set.
  - **condition-defined:** The lower entity sets membership is decided on the basis of whether or not an entity satisfies an explicit condition.
    - **Example:** All customers over 65 years are members of *senior-citizen* entity set; *senior-citizen* ISA *person*. All accounts having account-type="saving" are included in Saving Account entity set; Saving Account ISA an account.
  - **user-defined:** In this case the membership is not decided by a constraint rather the database user assigns entities of the higher level entity set to a lower level entity set.
    - **Example:** the database user decides the membership of a particular employee in a particular project group without any constraint.
- ✓ Constraint on whether or not entities may belong to more than one lower-level entity set within a single generalization.
  - **Disjoint:** A higher-level entity belongs to no more than one lower-level entity set.
    - **Example:** an entity of account entity set belong to only one lower-level entity set either saving or checking
    - In E-R diagram disjoint constraint is denoted by writing *disjoint* next to the ISA triangle
  - **Overlapping:** an entity can belong to more than one lower-level entity set within a single generalization.
    - **Example:** Some of the employees may be in more than one project groups.
- ✓ **Completeness constraint** -- specifies whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets within a generalization.
  - **total** : an entity must belong to one of the lower-level entity sets.
    - In E-R diagram it is denoted by using double line to connect the box representing the higher level entity to the triangle symbol.
    - **Example**: the account generalization is total.
  - **partial**: an entity need not belong to one of the lower-level entity sets.

Aggregation:

- ✓ It is an abstraction in which relationship sets (along with their associated entity sets) are treated as higher level entity sets and can participate in relationships.
- ✓ Aggregation allows us to indicate that a relationship set participates in another relationship set.
- ✓ **Example:**
  - Project and Department are two entity sets. They are related through a relationship set Sponsor.
  - Suppose the department that sponsors a project might assign employees to monitor the sponsorship. So there will be another relationship set Monitor.
  - We need to associate the monitor relationship set with employee entity set and Sponsor relationship set.
  - But we can associate only entity sets to a relationship set. One relationship set can't be associated with another relationship set. For this we need aggregation.
  - Here the relationship set Sponsor will be treated like a higher-level entity set and hence can be associated with another relationship set.



- ✓ So aggregation is needed when we need to express a relationship with another relationship set.
- ✓ In the above example if we don't need the "until" attribute of the monitor relationship set then we can use a ternary relationship among Project, Employee and Department entity sets.
- ✓ In case of a constraint like each sponsorship be monitored by at most one employee we can't express it by a ternary relationship. But it can be expressed by using an arrow from aggregated relationship to the monitor relationship set.

Relational Model:

- ✓ This model uses a collection of tables to represent both data and the relationship between them.
- ✓ All data is logically structured within relations.
- ✓ A relation is a **table** with columns and rows.

account_number	*branch_name*	*balance*
A-101	Downtown	500
A-102	Perryridge	400
A-201	Brighton	900
A-215	Mianus	700
A-217	Brighton	750
A-222	Redwood	700
A-305	Round Hill	350

- ✓ All data about a particular subject. (Person, place, thing, entity, or event) is stored in a table.
- ✓ A **field** (column, attribute) is used to store data in the database and represents a characteristic of the subject of the table in which it resides.
- ✓ A **record** (row, tuple) represents a unique instance of the subject of the table. It is composed of the entire set of fields in a table.
- ✓ For each attribute there is a set of permitted values called as **domain** of that attribute.
- ✓ Mathematically a relation is a subset of a Cartesian product of a list of domains.
- ✓ Formally, given sets $D_1, D_2, \ldots D_n$ where each D_i is a domain.

A ***relation*** r is a subset of

$$D_1 \times D_2 \times \ldots \times D_n$$

Thus, a relation is a set of n-tuples $(a_1, a_2, \ldots, a_n)$ where each $a_i \in D_i$

- ✓ **Example:** If

  customer_name = {Jones, Smith, Curry, Lindsay, …}
  customer_street = {Main, North, Park, …}
  customer_city = {Harrison, Rye, Pittsfield, …}

  Then r = { (Jones, Main, Harrison),
  (Smith, North, Rye),
  (Curry, North, Rye),
  (Lindsay, Park, Pittsfield)
  }

  is a relation over

  customer_name x customer_street x customer_city
- ✓ Attribute values are (normally) required to be **atomic**; that is, indivisible.
  Examle: the value of an attribute can be an account number, but cannot be a set of account numbers
- ✓ Domain of an attribute is said to be atomic if all its members are atomic.
- ✓ It is possible for several attributes to have the same domain.
- ✓ The special value ***null*** is a member of every domain. It causes complications in the definition of many operations.
- ✓ **Arity** of a relation is the no of domains in the corresponding relation.
- ✓ **Cardinality** of a relation is the no of tuples present in the relation.

Database Schema:

- ✓ **Database Schema** is the logical design of the database.
- ✓ **Database instance** is a snap-shot of the data in the database at a particular instance.
- ✓ **Relation Schema:** If $A_1, A_2, \ldots, A_n$ are *attributes* then $R = (A_1, A_2, \ldots, A_n)$ is a *relation schema.*
  Example:
  Customer_schema = (customer_name, customer_street, customer_city)
- ✓ r(R) denotes a *relation r* on the *relation schema.*
  Example: customer (Customer_schema).
- ✓ **Relation Instance:** The current values of a relation specified in a table at a particular instance.

DEFINITION SUMMARY

Informal Terms		Formal Terms
Table		Relation
Column		Attribute/Domain
Row		Tuple
Values in a column		Domain
Table Definition		Schema of a Relation
Populated Table		Extension

Database

- ✓ A database consists of multiple relations
- ✓ Information about an enterprise is broken up into no of parts, with each relation storing one part of the information

- *Account* : stores information about accounts
  depositor : stores information about which customer owns which account
  customer : stores information about customers

✓ Storing all information as a single relation such as
Bank(account_number, balance, customer_name, ..) results in:
- repetition of information
  - Example: if two customers own an account (What gets repeated?)
- the need for null values
  - Example: to represent a customer without an account

Characteristics of Relations:

1. **Ordering of tuples in a relation:** The order in which tuples appear in a relation is irrelevant. However in a file, records are stored on disk in an order and while displaying the same order is followed.
2. **Ordering of values within a tuple:** An n-tuple is an ordered list of n values, so ordering of values in a tuple is important in a relation schema. However at a logical level the order of attributes and their values is not that important as long as correspondence between attributes and values is maintained.
3. **Values and Nulls in the tuples:** Each value in a tuple is an atomic value. Hence composite and multi valued attributes are not allowed.
   A special value null is used to represent the values of the attributes that may be unknown or may not apply to a tuple.
4. **Interpretation (Meaning) of a Relation:** The relation schema can be interpreted as a declaration or a type of assertion. For example the schema of a student relation asserts that a student entity has roll, name and address.

Keys:

✓ Super Key, Candidate key and Primary key are same as E-R Model.

✓ **Foreign Key:** The attribute of a relation that corresponds to the primary key of another relation is called a foreign key.
- **Example:** *customer_name* and *account_number* attributes of *depositor* are foreign keys to *customer* and *account* respectively.

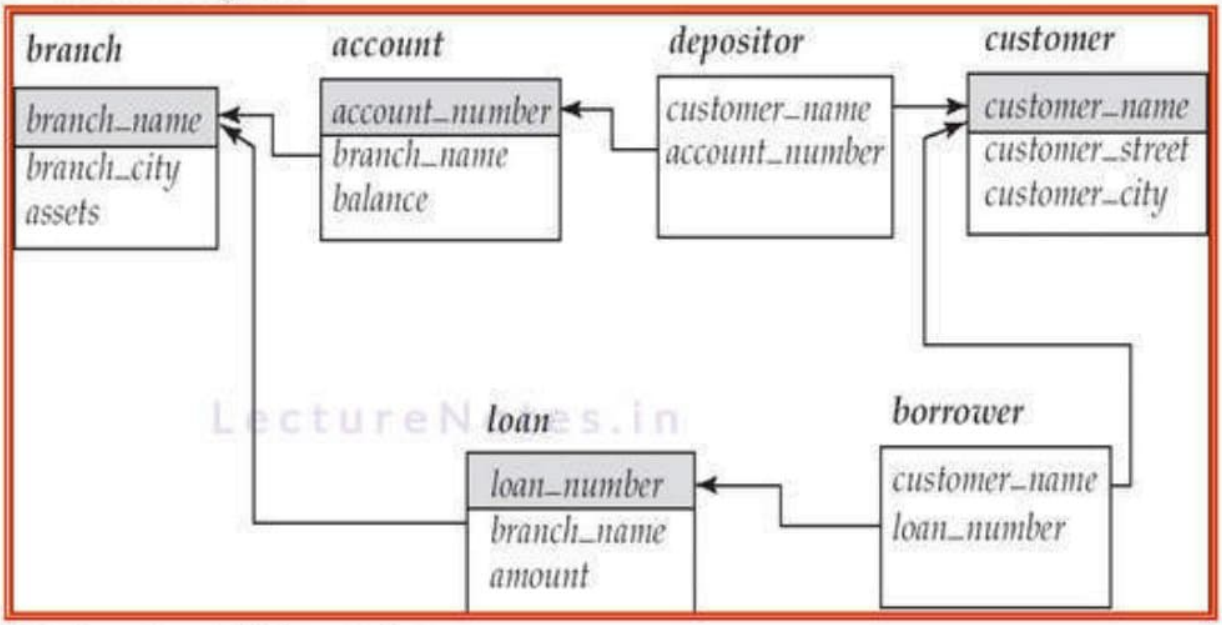
✓ Only values occurring in the primary key attribute of the **referenced relation** may occur in the foreign key attribute of the **referencing relation.**

✓ **Alternate Key** (or **secondary key**): It is any candidate key which is not selected to be the primary key (PK).
- **Example:** A relational database with a table "employee" could have attributes like "employee_id", "bank_acct_no", and so on. In this case, both "employee_id" and "bank_acct_no" serve as unique identifiers for a given employee, and could thus arguably be used for a primary key. Hence, both of them are called "candidate keys". If, for example, "bank_acct_no" was chosen as the primary key, "employee_id" would become the alternate key.

✓ **Partial key**: A key which identifies a subset of a set of information items (e.g. database "records"), and which could narrow the subset to one item if other partial key(s) were combined with it. It is similar to discriminator of E-R Model.
- **Example:** In the relation EMP_PROJ (having attributes ENO, ENAME, PNUMBER, HOURS) PNUMBER is the partial key because within each tuple PNUMBER **has** unique values.

✓ **Composite Key:** It is a primary key that consists of more than one column.

Schema diagram:



Relational model Constraints:

Constraints are the restriction on the actual values in a database state. Constraints on database are generally divided into the following categories:

1. **Model-based constraints:** Constraints that are inherent in the data model. The characteristics of relations are inherent constraints. Example: Constraint that a relation cannot have duplicate tuples.
2. **Schema-based constraints:** Constraints are directly expressed in the schemas of the data model by specifying them in the DDL. Example: Domain Constraint, Key Constraint, Constraint on Nulls, Entity Integrity Constraint, Referential Integrity Constraint.
3. **Application-based constraints:** Constraints that can't be directly expressed in the schemas of the data model and hence must be expressed and enforced by the application programs.

Integrity Constraint:

- ✓ For the uniformity of data while storing the data in the database some conditions are checked which is known as integrity constraint.
- ✓ Integrity constraint ensures that changes made to the database by authorized users don't result in a loss of data consistency.
- ✓ It protects the database from accidental damage.

1. **Domain Constraint:**
   - It is a set of permitted values in one column or attribute. Domain constraint allows us to test the values inserted into the database satisfy the domain of the attribute or not.
   - We provide name of a domain, meaning of a domain, type of domain, size of domain range of values etc.
2. **Entity Integrity Constraint:**
   - No primary key value can be null because primary key value is used to identify individual tuples in a relation.
3. **Referential Integrity:**
   - For every value of a foreign key there must be a primary key with that value.

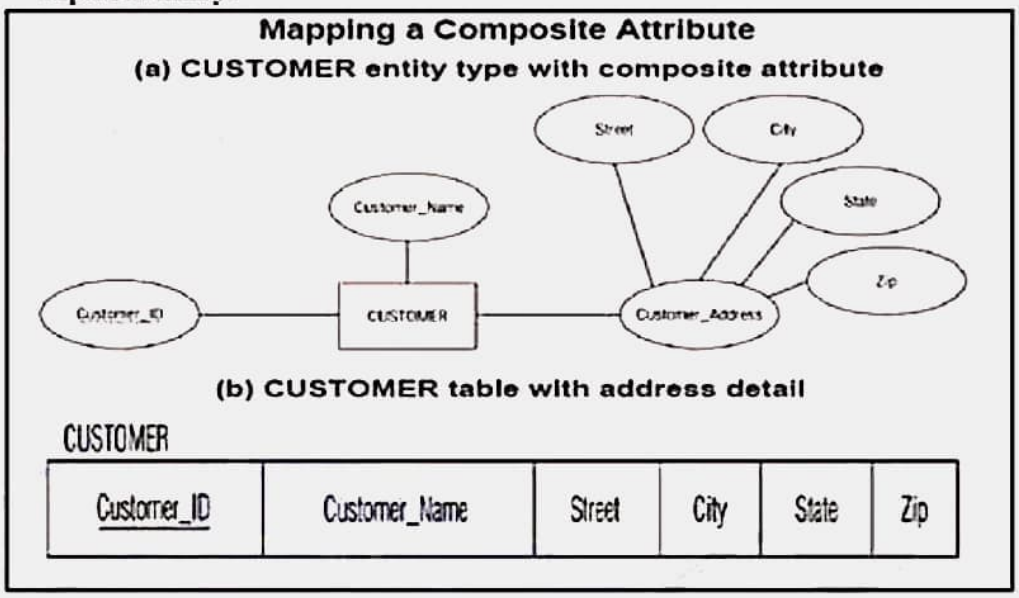
     Example: For every value of Customer_name in the depositor table there must be a matching value of customer_name in the customer table.
   - The primary key must exist before the foreign key can be defined.

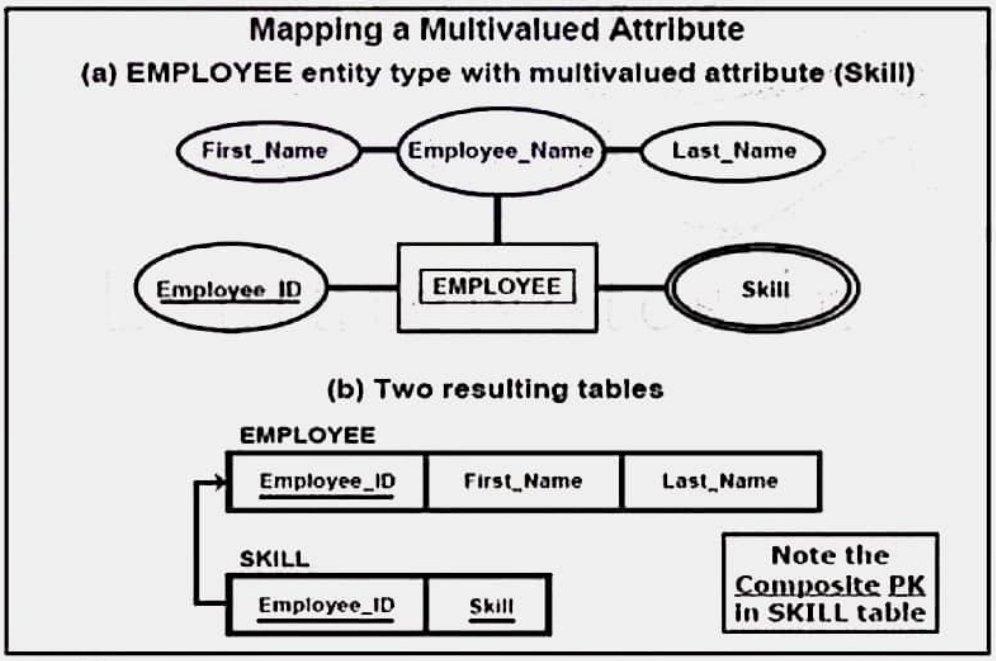
     Example: Create customer and account table before depositor table.

4. **Key Constraint:**
   - While inserting the data into a relation key constraint can be violated if the key value in the new tuple already exists.

Mapping E-R to Relational Model:

1. Map Regular Entities to Tables
   - Composite Attributes: Use only their simple, component attributes.
   - Multivalued Attribute: Becomes a separate table with a foreign key taken from the superior entity.

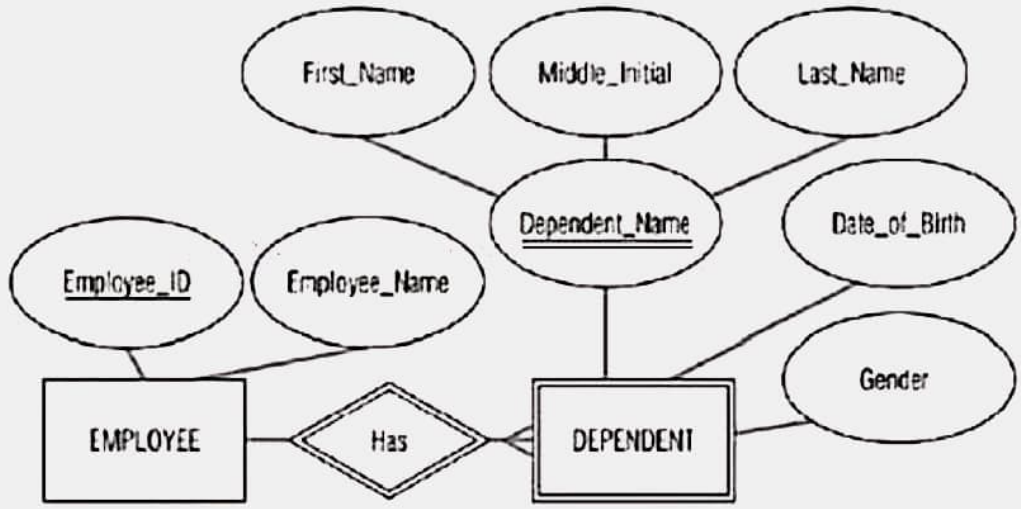




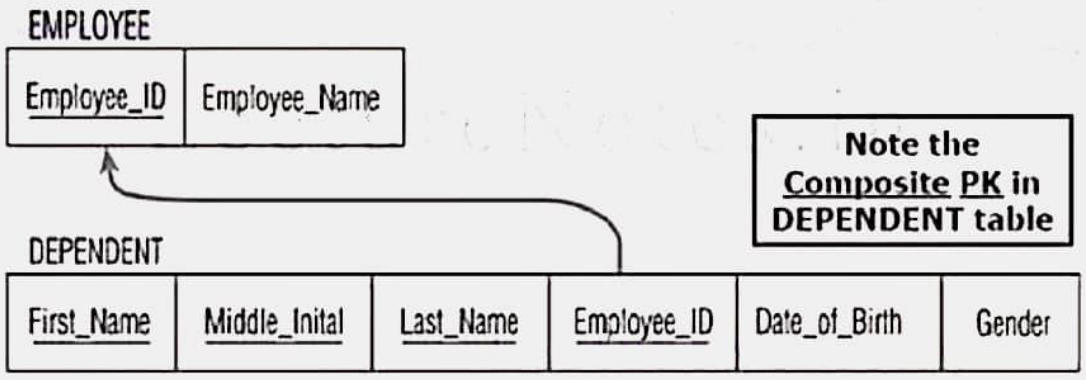
2. Map Weak Entities
   - Becomes a separate table with a foreign key taken from the strong entity.
   - Primary key is composed of the partial identifier (discriminator) of the weak entity plus the primary key from the strong entity (i.e., composite PK)

Example of mapping a weak entity

(a) Weak entity DEPENDENT

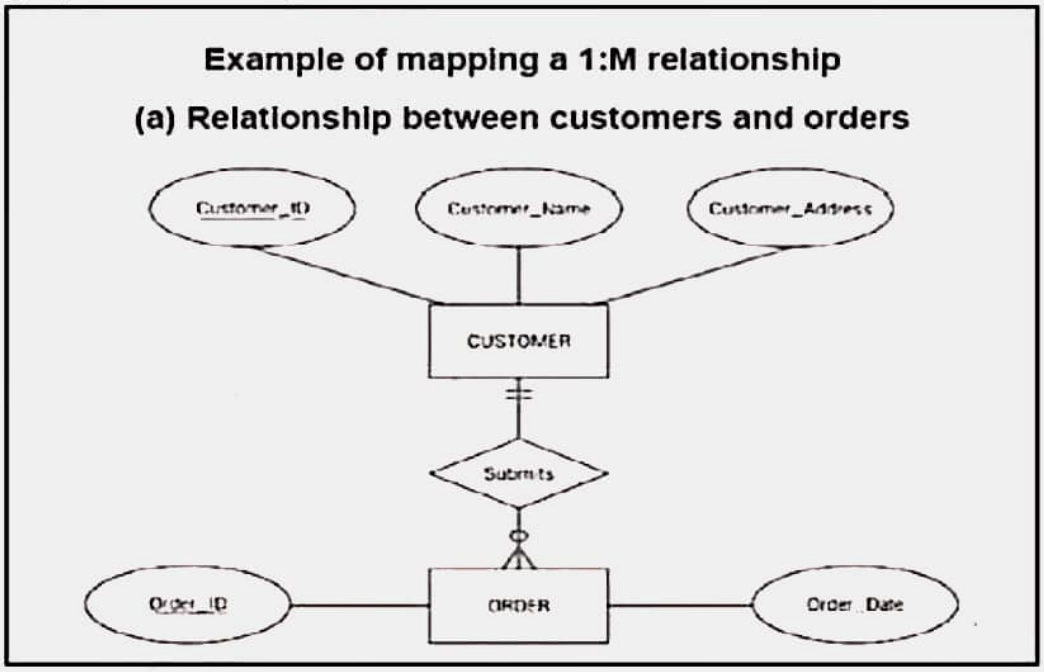


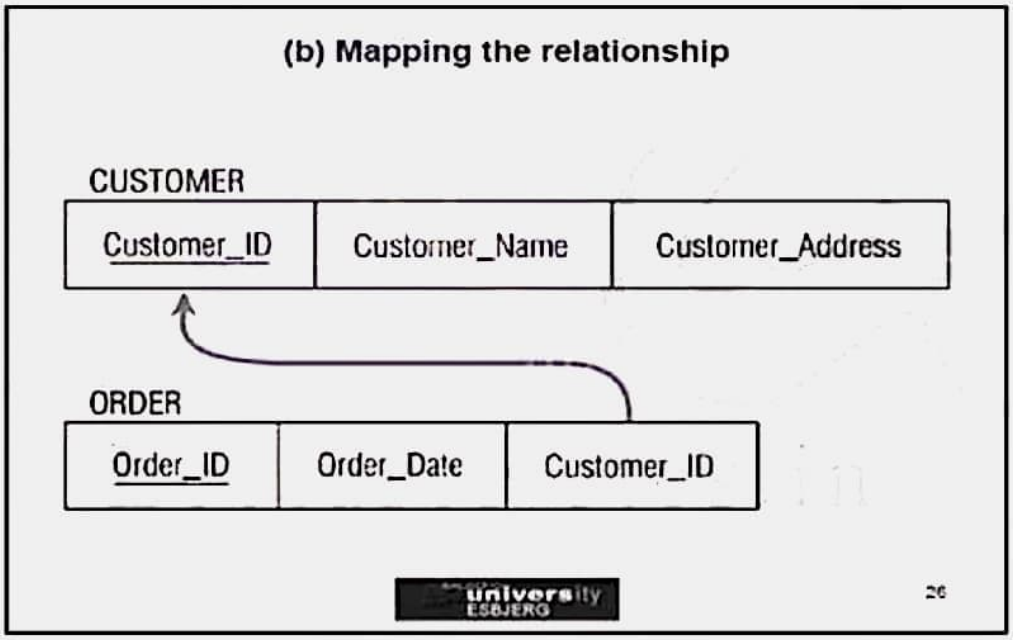
(b) Tables resulting from mapping weak entity



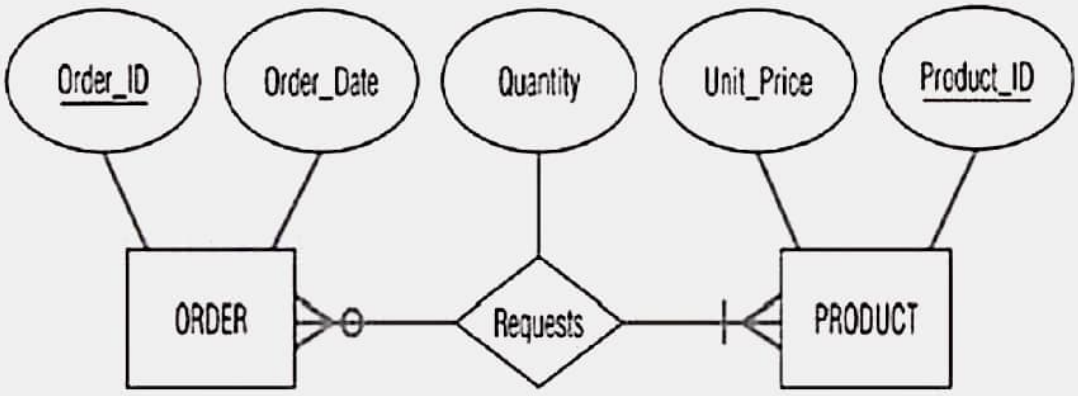
3. Map Binary Relationships
   - One-to-Many - Primary key on the *one* side becomes a foreign key on the *many* side.
   - Many-to-Many - Create a *new table* with the primary keys of the two entities as its Primary Key.

- One-to-One - Primary key on the *mandatory*side becomes a foreign key on the *optional* side (*if* optionalities are asymmetric).

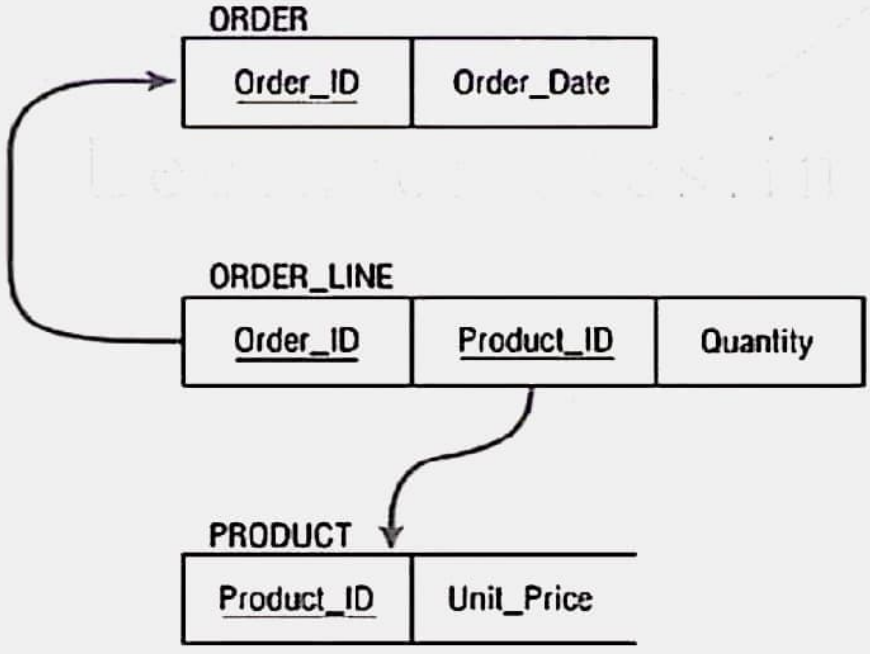


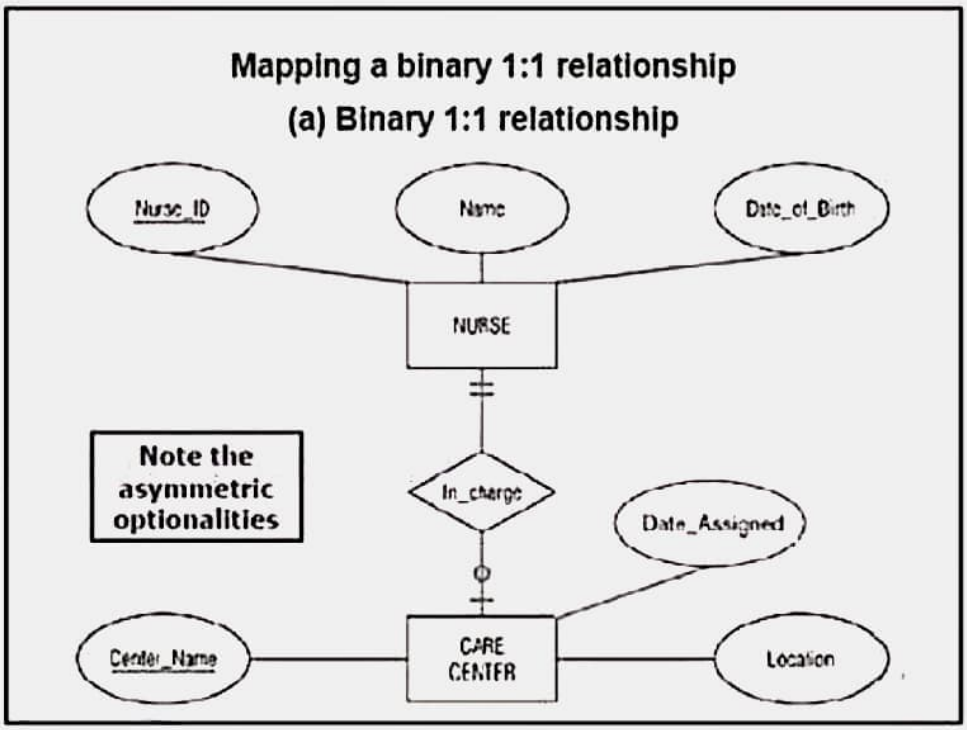


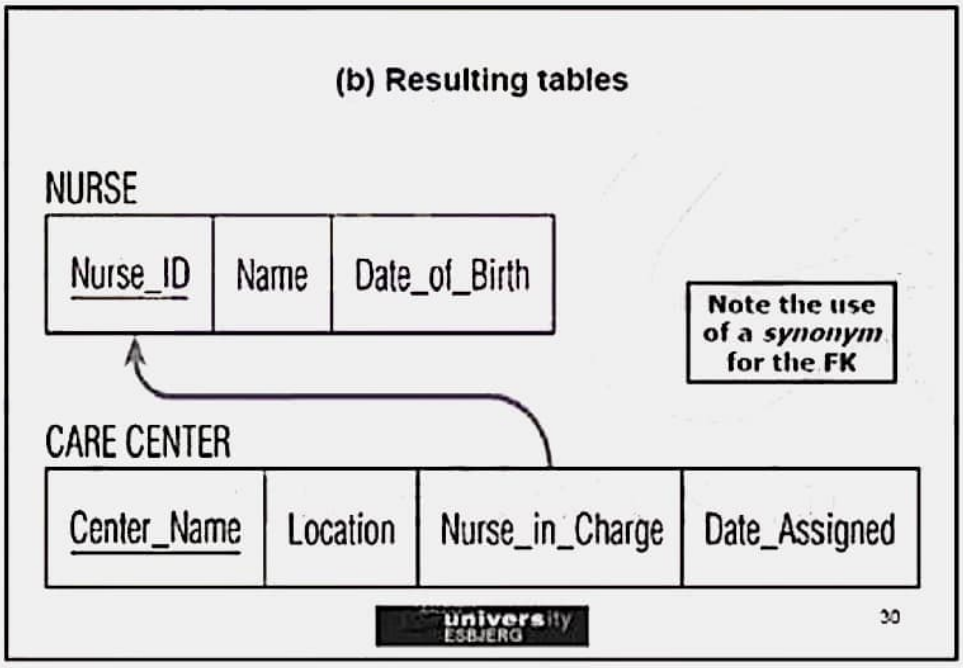
Example of mapping an M:N relationship
(a) Requests relationship (M:N)



(b) Three resulting tables



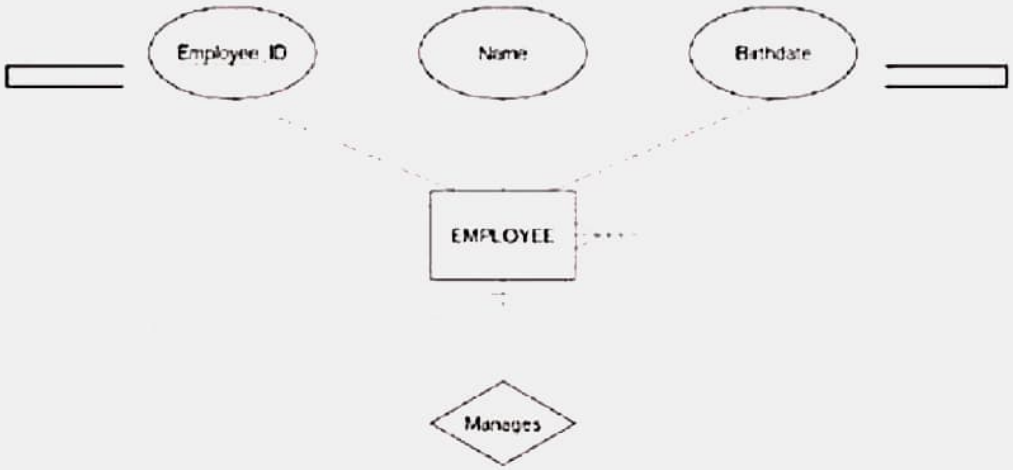




4. Map Unary (Recursive) Relationships
   - One-to-Many: Recursive foreign key in the *same* table.
   - Many-to-Many (e.g., bill-of-materials):
     Two tables result:
     - One for the entity type.
     - One for an associative relation in which the primary key has two fields, both taken from the identifier of the original entity.

Mapping a unary 1:M relationship

(a) EMPLOYEE entity type with Manages relationship



(b) EMPLOYEE table with *recursive foreign key*

Note mandatory use of *synonym*

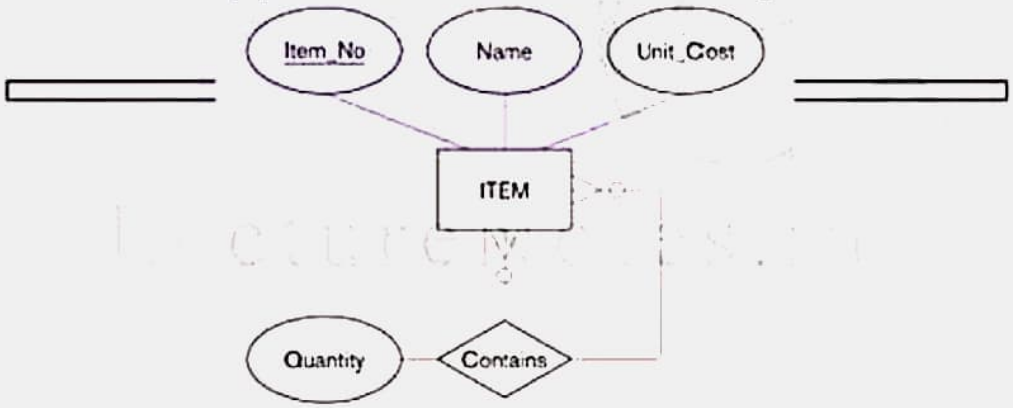
EMPLOYEE

Employee_ID	Name	Birthdate	Manager_ID

37

Mapping a unary M:N relationship

(a) "Bill-of-materials" relationship



(b) Two resulting tables

ITEM

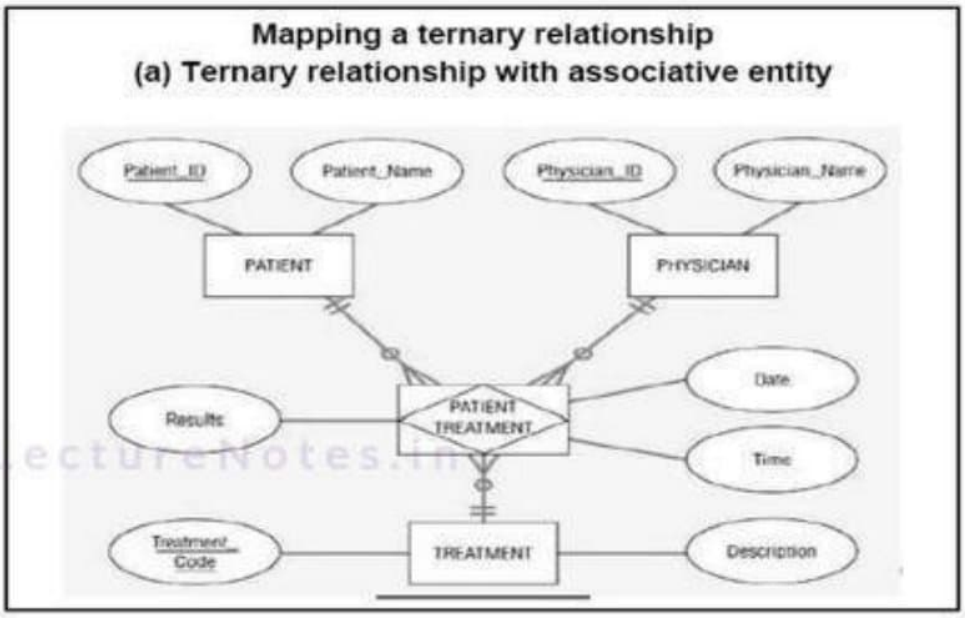
Item_No	Name	Unit_Cost

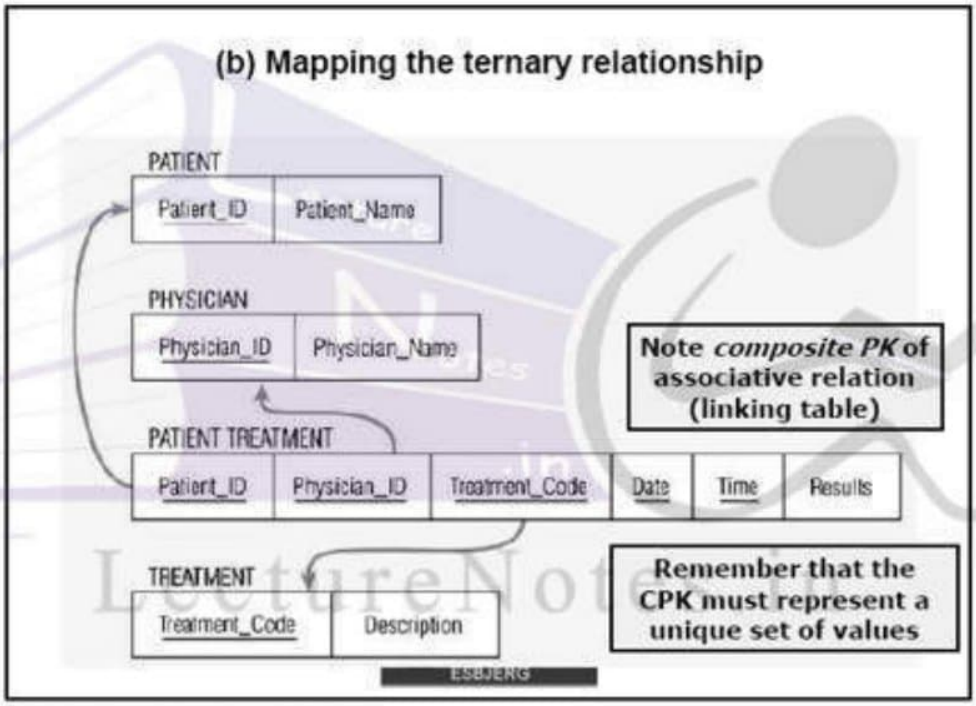
COMPONENT

Item_No	Component_No	Quantity

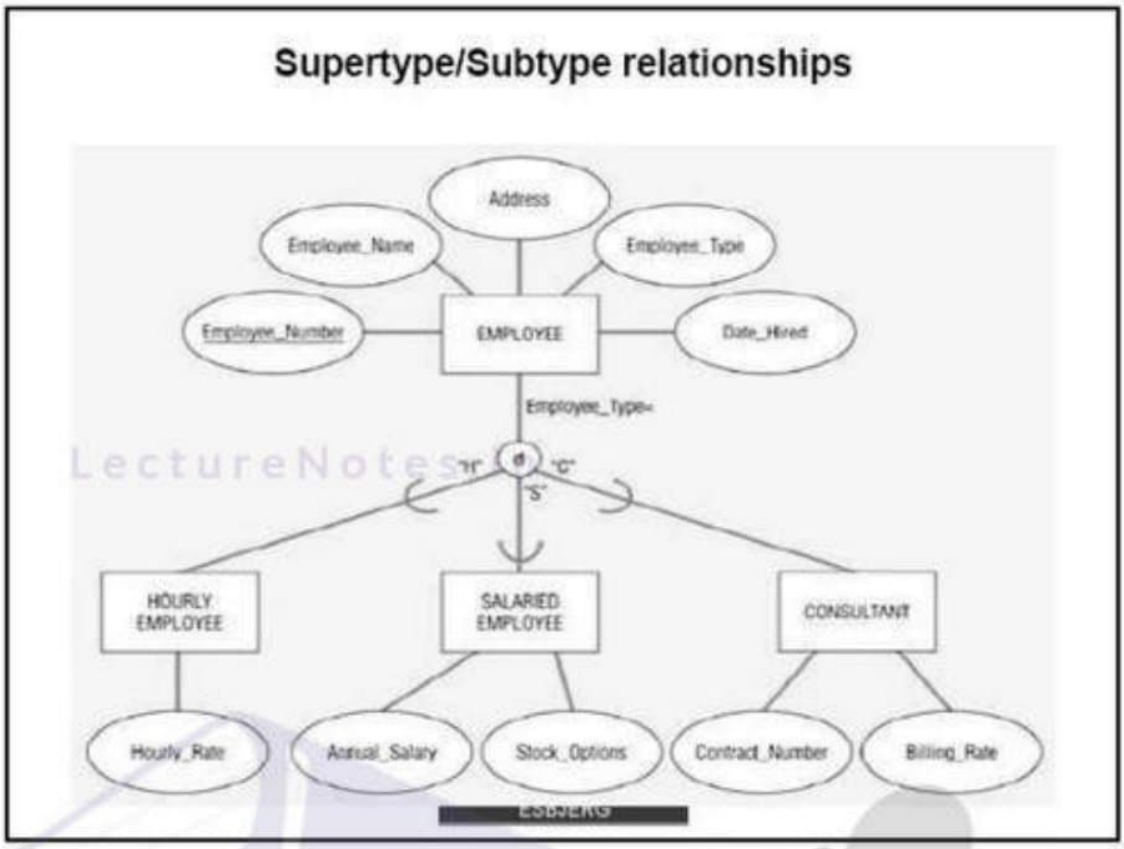
Note *composite PK*, two FKs referencing the same PK, and mandatory use of *synonym*

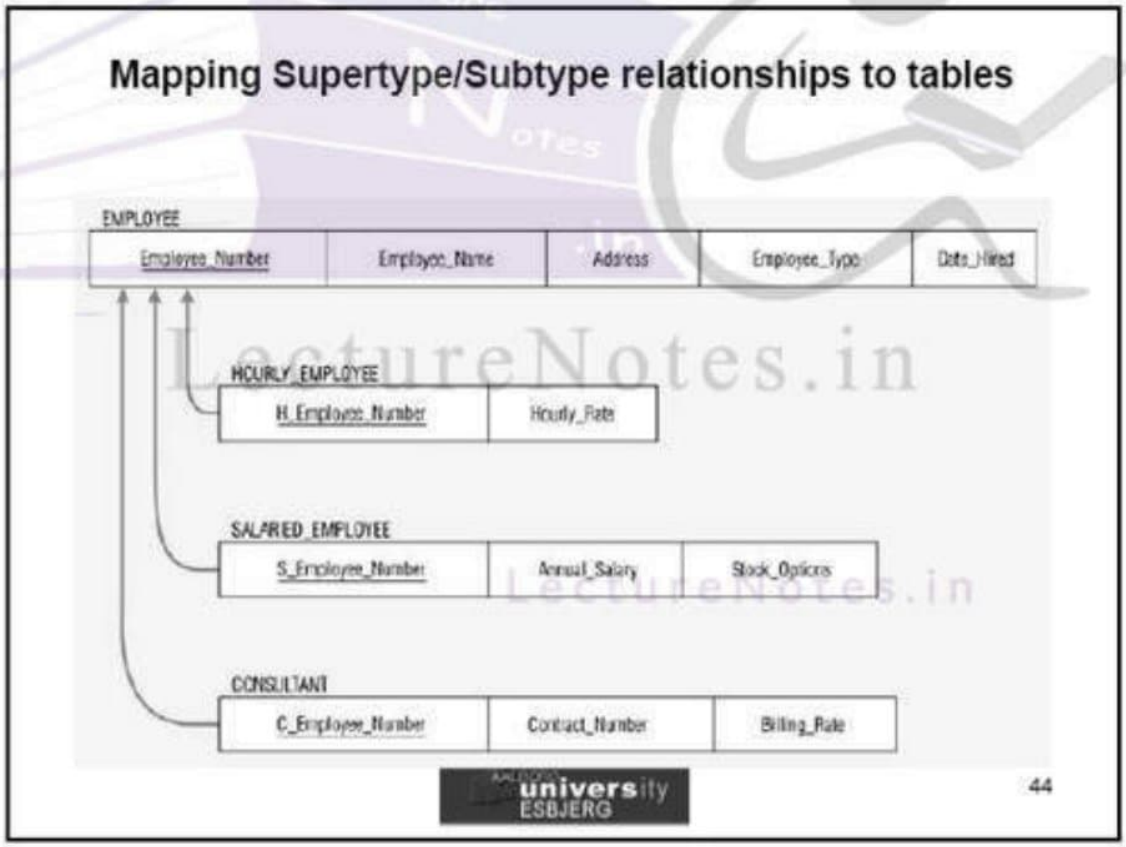
5. Map Ternary (and *n*-ary) Relationships:
   - One table for each original entity and one for the common relationship (associative entity) (e.g., a ternary relationship maps to a total of four tables).
   - Table representing the associative entity has foreign keys to each entity in the relationship.





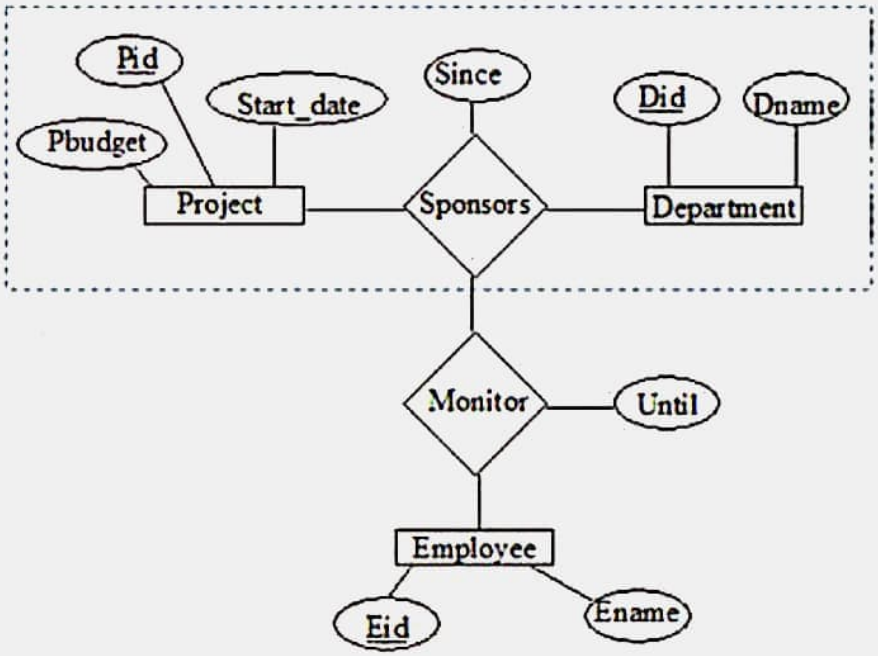
6. Map Supertype/Subtype Relationships:
   - Create a separate table for the supertype and each of the subtypes.
   - Assign common fields, including subtype discriminator, to the supertype table
   - Assign to the subtype tables those fields unique to each subtype
   - Assign to the subtype tables the primary key of the supertype table (which also functions as a FK referencing the supertype).

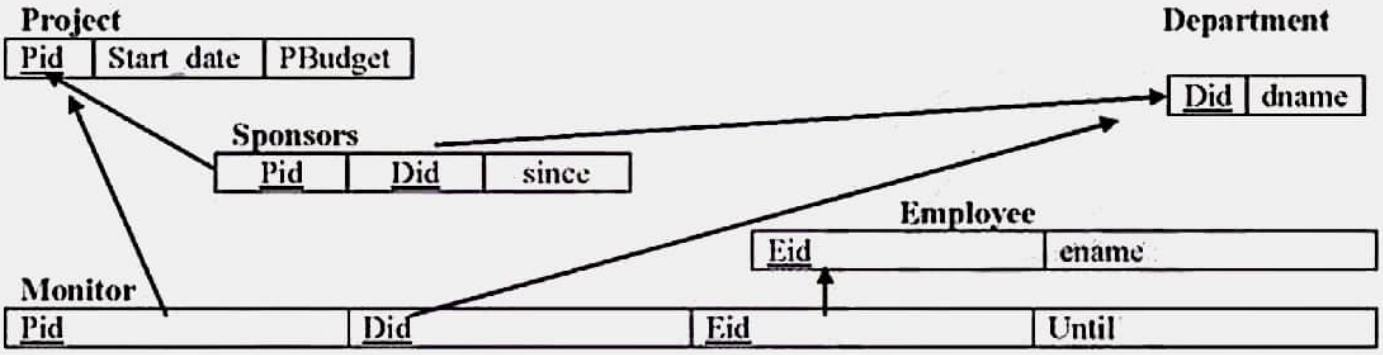




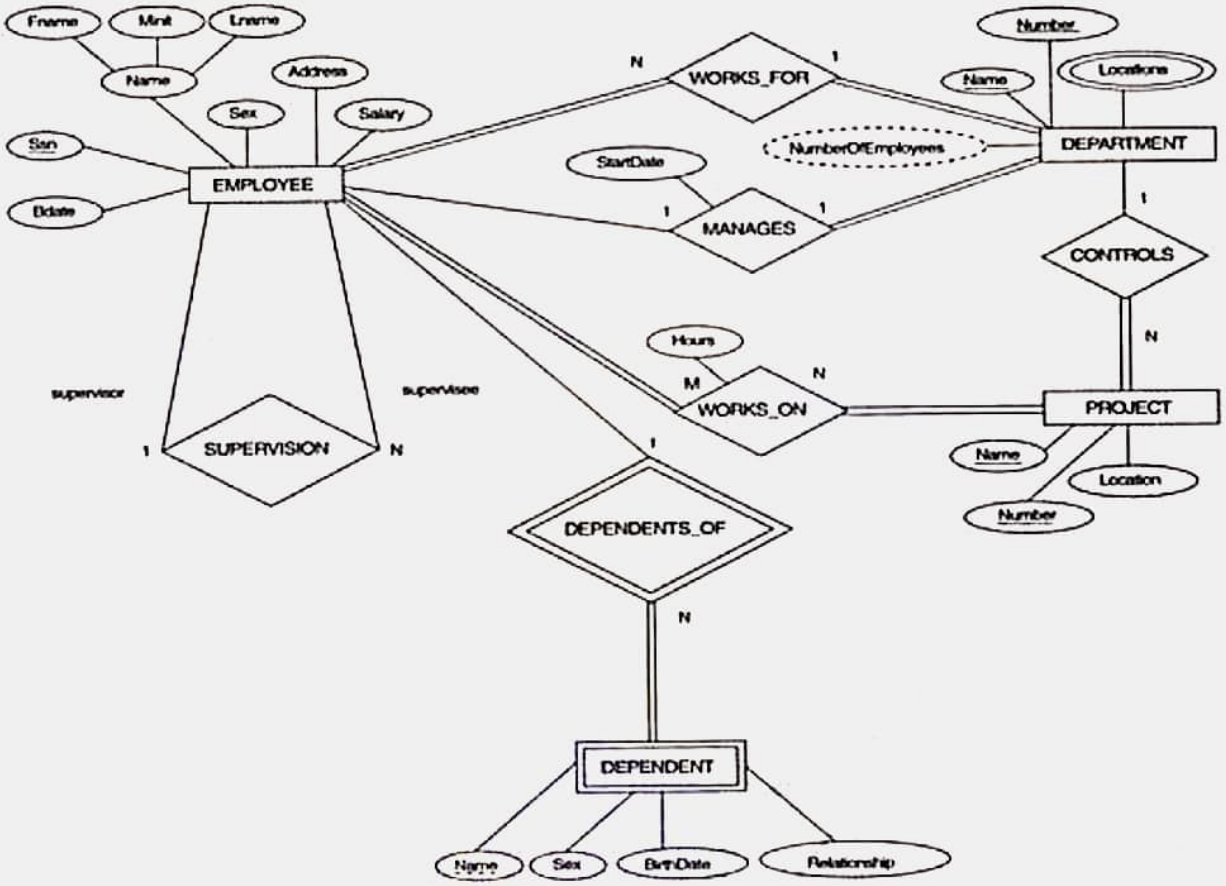
7. **Map E-R Diagram with aggregation**:
   - Create a separate table for each entity set (Project, Department & Employee).
   - Create the Sponsors relationship with attributes pid, did & since where pid & did are the composite primary key.

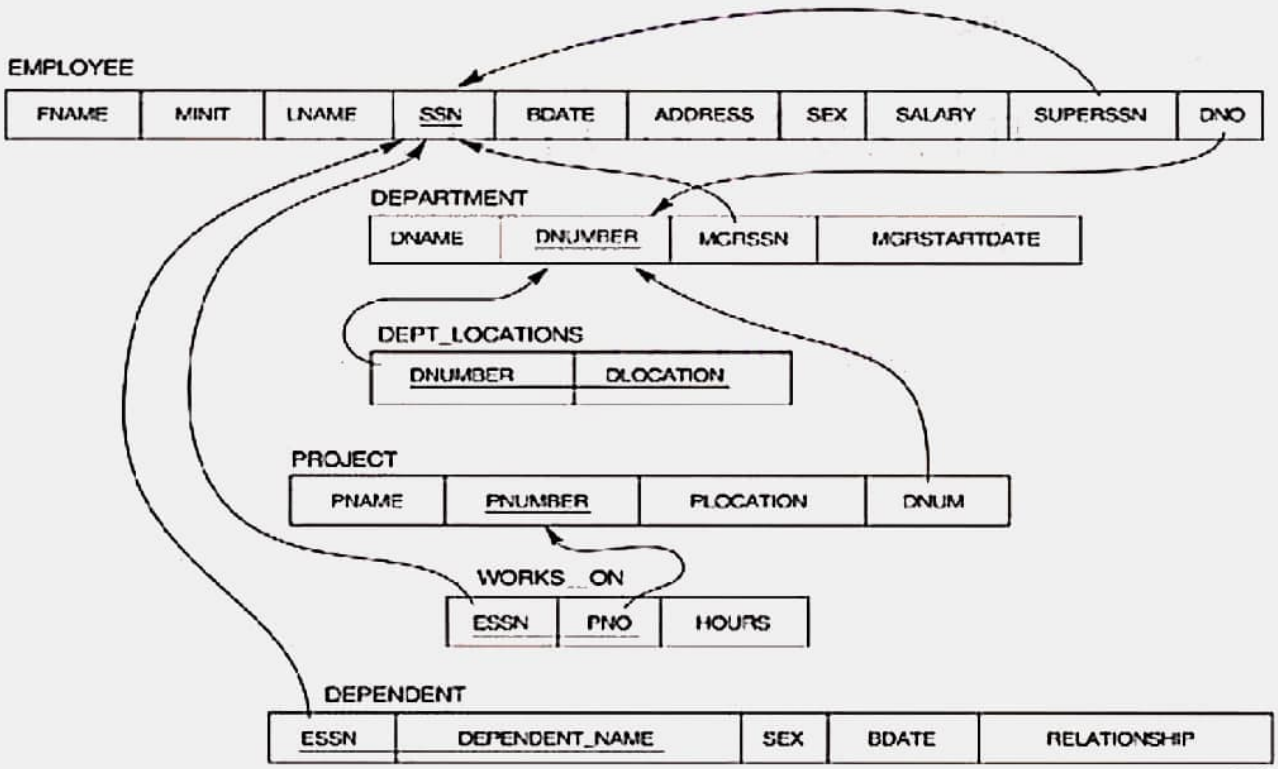
- Create the monitor relationship with key attributes of Employee (eid), key attribute of Sponsors (pid,did) and descriptive attribute until. Eid, pid & did forms the composite primary key.





Complete Example of Mapping E-R to Relational Model:





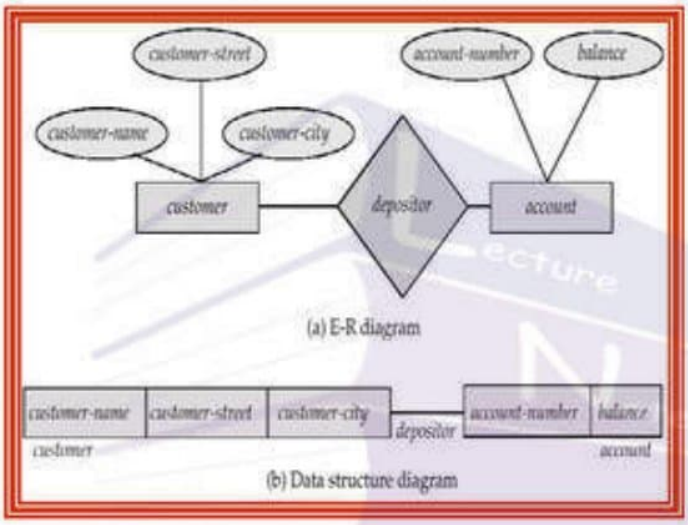
The Network Model:

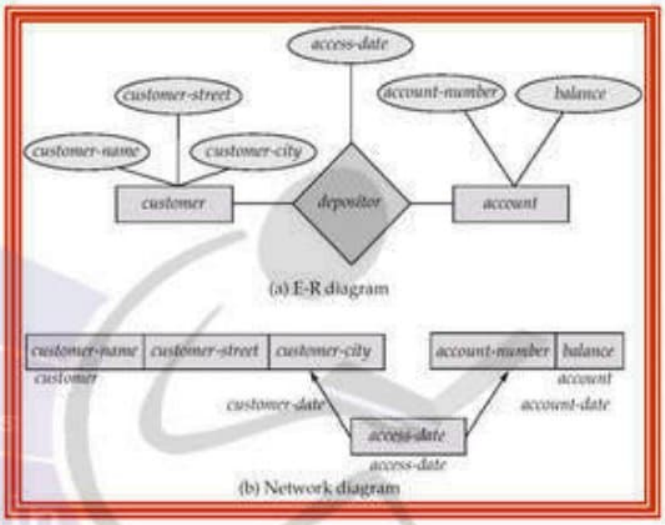
- ✓ Data are represented by collections of *records* similar to an entity in the E-R model.
- ✓ Records and their fields are represented as *record type*
- ✓ **Example:**

type *customer* = **record**
customer-name: string;
customer-street: string;
customer-city: string;
end

type *account* = **record**
account-number: integer;
balance: integer;
end

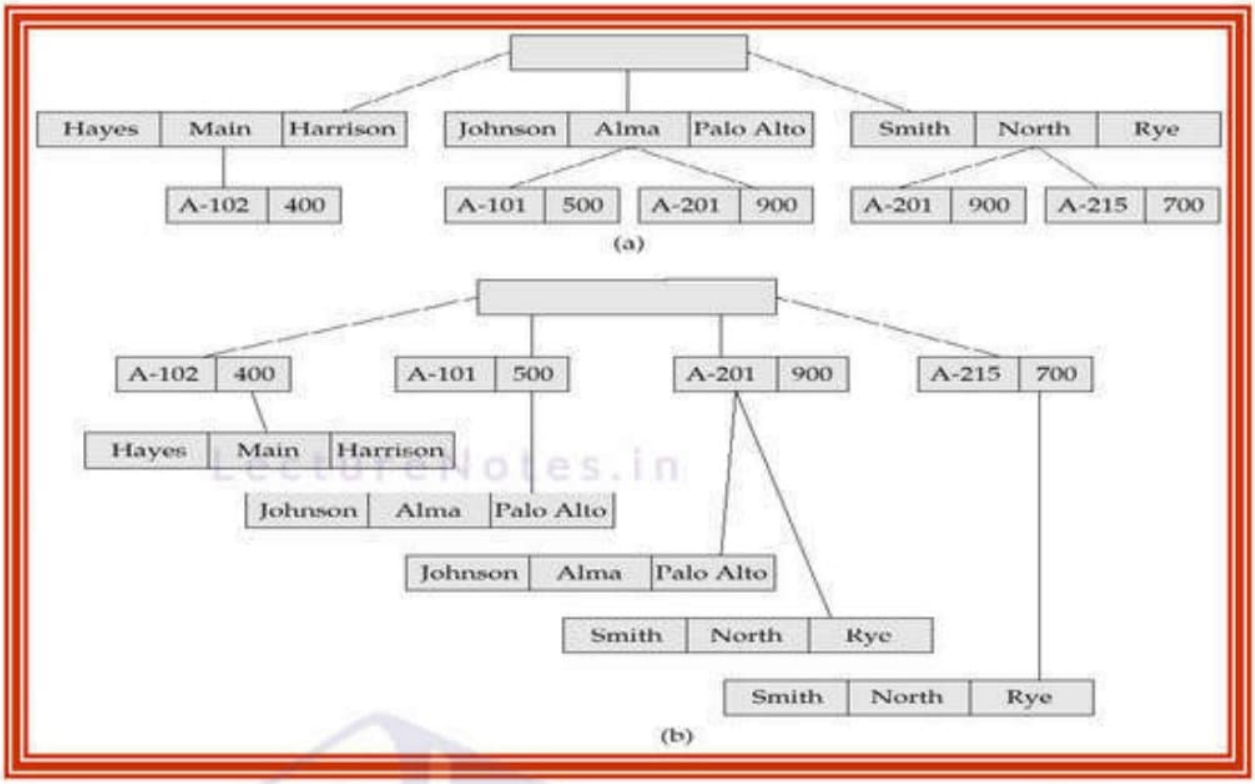
- ✓ The records in the database are organized as collection of arbitrary graphs.
- ✓ Relationships among data are represented by *links* similar to a restricted (binary) form of an E-R relationship
- ✓ Restrictions on links depend on whether the relationship is many-many, many-to-one, or one-to-one.
- ✓ A data-structure diagram consists of two basic components:
  - **Boxes**, which correspond to record types.
  - **Lines**, which correspond to links.





The Hierarchical Model:

- ✓ A hierarchical database consists of a collection of *records*, which are connected to one another through *links*.
- ✓ A record is a collection of fields, each of which contains only one data value.
- ✓ A link is an association between precisely two records.
- ✓ The hierarchical model differs from the network model in that the records are organized as collections of trees rather than as arbitrary graphs.
- ✓ The schema for a hierarchical database consists of
  - ***boxes***, which correspond to record types
  - ***lines***, which correspond to links
- ✓ Record types are organized in the form of a *rooted tree.*
  - No cycles in the underlying graph.
  - Relationships formed in the graph must be such that only one-to-many or one-to-one relationships exist between a parent and a child.
- ✓ Database schema is represented as a collection of tree-structure diagrams.
  - *single* instance of a database tree
  - The root of this tree is a dummy node
  - The children of that node are actual instances of the appropriate record type.



The Object Oriented Data Model:

A core object-oriented data model consists of the following basic object-oriented concepts:

- ✓ **Object and Object identifier**: Any real world entity is uniformly modeled as an object (associated with a unique id: used to pinpoint an object to retrieve).
- ✓ **Attributes and methods**: every object has a state (the set of values for the attributes of the object) and a behavior (the set of methods - program code - which operate on the state of the object). The state and behavior encapsulated in an object are accessed or invoked from outside the object only through explicit message passing.
- ✓ **Class**: a means of grouping all the objects which share the same set of attributes and methods. An object must belong to only one class as an instance of that class (instance-of relationship). A class is similar to an abstract data type. A class may also be primitive (no attributes), e.g., integer, string, Boolean.
- ✓ **Class hierarchy and inheritance**: derive a new class (subclass) from an existing class (superclass). The subclass inherits all the attributes and methods of the existing class and may have additional attributes and methods. single inheritance (class hierarchy) vs. multiple inheritance (class lattice).

Database System Structure:

A database system is partitioned into the following modules depending on their function:

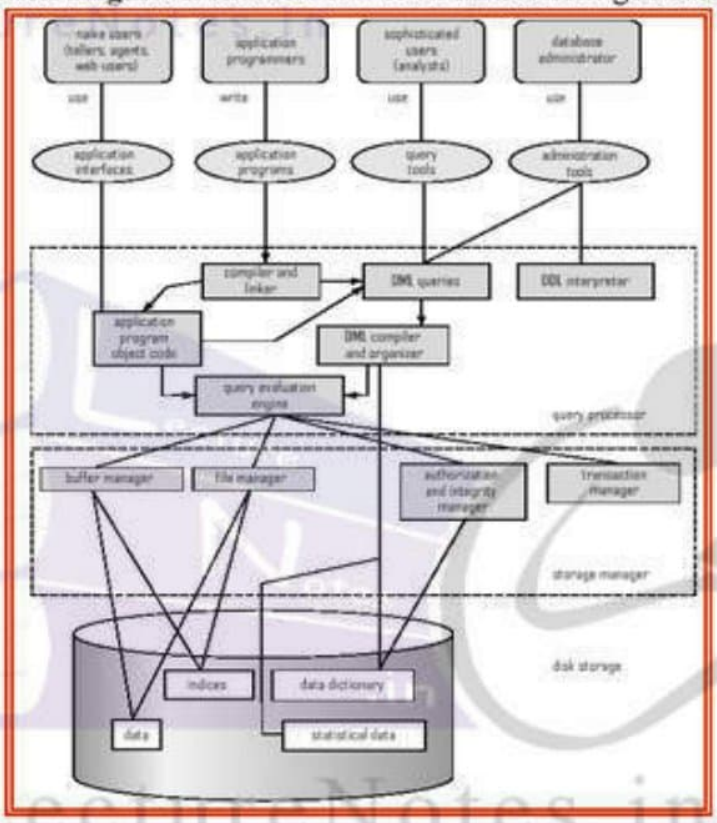
1. **Storage Manager:** It is a program module that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system. It is responsible for storing, retrieving and updating data in the database. The components of the storage manager includes:
   - ✓ **Authorization and Integrity Manager:** It checks the integrity constraints and the authority of the users to access data.
   - ✓ **Transaction Manager:** It ensures that database remains in consistent state in case of system failure or in concurrent transaction.
   - ✓ **File Manager:** It manages the allocation of disk space and data structures used to represent data stored on disk.

- ✓ **Buffer Manager:** It fetches data from disk to memory and decides what data to cache in main memory.

The storage manager implements several data structures as part of physical implementation:

- ✓ **Data Files:** It stores the database itself.
- ✓ **Data Dictionary:** It stores the metadata.
- ✓ **Indices:** It provides faster access to the data items.

2. **Query Processor:** It includes the following components:
   - ✓ **DDL interpreter:** It interprets DDL statement and records the definition in the data dictionary.
   - ✓ **DML compiler:** It translates DML statement in a query language into an evaluation plan consisting of low-level instructions for the query evaluation engine. It also performs query optimization.
   - ✓ **Query evaluation Engine:** It executes low-level instructions generated by DML compiler.



Query Language:

- ✓ It is a language in which a user requests information from the database.
- ✓ Types of Query Language:
  - **Procedural Language:** The user instructs the system to perform a sequence of operations on the database to compute the desired result.
    Example: relational algebra.
  - **Non-Procedural Language / Declarative Language:** The user describes the desired information without giving a specific procedure.
    Example: relational calculus.

Relational Algebra:

- ✓ It is a procedural language.
- ✓ It consists of a set of operations that take one or two relations as input and produce a new relation as the output.

Bank Database

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

Fundamental operations:

1. **The Select Operation (σ):**
   - ✓ The select operation selects tuples from a relation that satisfy a given predicate.
   - ✓ Notation: $\sigma_{\text{selection predicate}}$ (Relation)
   - ✓ Example: $\sigma_{\text{branch-name="Sahid Nagar"}}$ (Loan)
     $\sigma_{\text{salary} > 10000}$ (Employee)
   - ✓ The selection predicate can use =, ≠, <, ≤, >, ≥ operators for comarisions.
   - ✓ Two or more predicates can be combined by using the connectives ∧ (and), ∨ (or) and ¬ (not)
2. **The Project Operation (Π):**
   - ✓ The project operation produces a relation with the attributes mentioned in the attribute list.
   - ✓ Duplicate rows removed from result, since relations are sets
   - ✓ Notation: $\Pi_{\text{attribute-list}}$ (Relation)
   - ✓ Example: $\Pi_{\text{Loan-no, amount}}$ (Loan)
     $\Pi_{\text{empno, salary}}$ (Employee)
3. **The Union Operation (∪):**
   - ✓ The Union operation combines the tuples of two relations to form a new relation by removing duplicate rows and returns the relation.
   - ✓ Notation: $r \cup s$ *which is defined as* $r \cup s = \{t \mid t \in r \text{ or } t \in s\}$
   - ✓ For $r \cup s$ to be valid r and s must be compatible i.e.
     - r, *s* must have the *same* **arity** (same number of attributes).

- The domain of the i^{th} attribute of r and the i^{th} attribute of s must be same for all i.

✓ Example: to find all customers with either an account or a loan

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

4. The Set Difference Operation (–):

✓ The set difference operation finds the tuples that are in one relation but not in another relation.

✓ Notation: $r - s$ which is defined as $r - s = \{t \mid t \in r \text{ or } t \notin s\}$

✓ For $r - s$ to be valid r and s must be compatible

- r, s must have the same arity (same number of attributes).
- The domain of the i^{th} attribute of r and the i^{th} attribute of s must be same for all i.

✓ Example: to find all customers of the bank who have an account but not a loan

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

5. The Cartesian-Product Operation /Cross Product (X):

✓ The Cartesian-product operation allows us to combine information from any two relations.

✓ Notation: $r \times s$ which is defined as $r \times s = \{t\,q \mid t \in r \textbf{ and } q \in s\}$.

$$R(A_1, A_2, \ldots, A_n) \times S(B_1, B_2, \ldots, B_m) = Q(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m)$$

✓ Assume that attributes of r(R) and s(S) are disjoint. (That is, $R \cap S = \phi$).

✓ If attributes of r(R) and s(S) are not disjoint, then renaming must be used.

✓ **Example 1:** Find the name of the customers who have a loan at the Link road branch.

$$\Pi_{\text{customer\_name}}(\sigma_{\text{borrower.loan-no=loan.loan-no}}(\sigma_{\text{branch-name="Link Road"}}(\text{borrower} \times \text{loan})))$$

✓ **Example 2:** Find the names of all customers who have a loan at the Link Road branch but do not have an account at any branch of the bank.

$$\Pi_{\text{customer\_name}}(\sigma_{\text{borrower.loan-no=loan.loan-no}}(\sigma_{\text{branch-name="Link Road"}}(\text{borrower} \times \text{loan}))) - \Pi_{\text{customer\_name}}(\text{depositor})$$

✓ **Example 3:** Find the name of each female employee's dependant

$$\Pi_{\text{FNAME, LNAME, DEPENDENTNAME}}(\sigma_{\text{Employee.SSN=Dependent.ESSN} \wedge \text{SEX='F'}}(\text{Employee} \times \text{Dependent}))$$

6. The Rename Operation (ρ):

✓ The Rename operation allows us to name, and therefore to refer to, the results of relational-algebra expressions.

✓ It allows us to refer to a relation by more than one name.

✓ Example: $\rho_X(E)$ returns the expression E under the name X

✓ If a relational-algebra expression E has arity n, then

$$\rho_{X(A1,A2,\ldots,An)}(E)$$

returns the result of expression E under the name X, and with the attributes renamed to $A_1, A_2, \ldots, A_n$.

✓ **Example:** Find the largest account balance.

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

Formal Definition of Relational Algebra

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

- A relation in the database
- A constant relation

✓ Let $E1$ and $E2$ be relational-algebra expressions then the following are all relational-algebra expressions:

- $E_1 \cup E_2$
- $E_1 - E_2$
- $E_1 \times E_2$
- $\sigma_P(E_1)$ where P is the predicate on attributes in E_1
- $\Pi_S(E_1)$ where S is a list consisting of some of the attributes in E_1

- $\rho_X(E_1)$ where X is the new name for the result of E_1

Additional Operations:

- ✓ Although additional operations do not add any power to the relational algebra, but that simplify common queries. Some of the additional operations are:

7. The Set Intersection Operation ($\cap$):

- ✓ The Set Intersection operation results the common tuples of two relations to form a new relation.
- ✓ Notation: $r \cap s$ *which is defined as* $r \cap s = \{t \mid t \in r \text{ and } t \in s\}$
- ✓ For $r \cap s$ to be valid r and s must be compatible i.e.
  - r, *s* must have the *same* **arity** (same number of attributes).
  - The domain of the i^{th} attribute of r and the i^{th} attribute of s must be same for all i.
- ✓ Example: to find all customers who have a loan and an account.

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

- ✓ Note: $r \cap s = r - (r - s)$

8. The Natural-Join Operation ($\bowtie$):

- ✓ The Set Natural-Join is a binary operation that results to combine certain selections and a Cartesian product into one operation.
- ✓ Definition: Let r(R) and s(S) be two relations. The natural join of r and s, denoted by $r \quad s = \Pi_{R \cup S}(\sigma_{r.A1=s.A1 \wedge r.A2=s.A2 \wedge \ldots \wedge r.An=s.An} \; r \times s)$ where $R \cap S = \{A_1, A_2, \ldots, A_n\}$
- ✓ **Example1:** Find the name of all customers who have a loan at the bank and find the amount of the loan.

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

- ✓ **Example2:** Find the name of all branches with customers who have an account in the bank and who live in Bhubaneswar.

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

- ✓ **Example3:** Find all customers who have both a loan and an account at the bank.

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

- ✓ The **Theta join** operation is an extension to the natural-join operation that allows us to combine a selection and a Cartesian product into a single operation.

$$r \bowtie_{\theta} S = \sigma_{\theta}(r \times s)$$

- ✓ The type of join in which only comparison operator used is =, is called **EQUIJOIN**. In EQUIJOIN we always have one or more pair of attributes that have identical values in every tuple.

9. The Division Operation (÷):

- ✓ The division operator is used to queries that include the phrase "**for all**". **Definition:** If A and B are two relations and C= A÷B then for each tuple in C its product with the tuples of B must be in A.
- ✓ **Formal Definition:** Let r(R) and s(S) be two relations, and $S \subseteq R$; that is every attribute of schema S is also in schema R. The relation r ÷s is a relation on schema R-S. A tuple t is in r ÷s if and only if both of two conditions hold:
  1. t is in $\Pi_{R-S}(r)$
  2. For every tuple t_s in s, there is a tuple t_r in r satisfying both of the following:
     a. $t_r[S] = t_s[S]$
     b. $t_r[R-S] = t$

A	B
a 1	b 1
a 1	b 2
a 2	b 1
a 3	b 1
a 4	b 2
a 5	b 1
a 5	b 2

P

B
b 1
b 2

Q

A
a 1
a 5

P ÷ Q

B
b 1

R

A
a 1
a 2
a 3
a 5

P ÷ R

B
b 1
b 2
b 3

T

A

P ÷ T

✓ **Example:** Find all customers who have an account at all branches located in Cuttack.

Branch

Branch_ name	Branch-city	Assets
Link Road	Cuttack	200000
Buxi Bazar	Cuttack	500000
Chandini Chowk	Cuttack	600000
Baramunda	Bhubaneswar	700000
Ram Mandir	Bhubaneswar	800000
Rajmahal	Bhubaneswar	900000

Depositor

Customer _Name	Acc_ no
Raj	1120
Shyam	4521
Kapil	2930
Ajit	1259
Shyam	4444
Shyam	2938

Account

Acc_ no	Brach_n ame	Bal
1120	Link Road	11000
4521	Buxi Bazar	31219
2930	Baramunda	66676
1259	Ram mandir	77875
4444	Chandini Chowk	9012
2938	Link Road	1110

A

Customer_ Name	Branch_Name
Raj	Link Road
Shyam	Buxi Bazar
Kapil	Baramunda
Ajit	Ram mandir
Shyam	Chandini Chowk
Shyam	Link Road

B

Branch_ Name
Link Road
Buxi Bazar
Chandini Chowk

C

Customer _Name
Shyam

A: $\Pi_{\text{customer\_name, branch\_name}}$ (depositor $\bowtie$ Account)

B: $\Pi_{\text{branch\_name}}$ ($\sigma_{\text{branch\_city="Cuttack"}}$ Branch)

C: A÷B

10. The Assignment Operation (←):

- ✓ The assignment operation provides a convenient way to express complex queries.
- ✓ 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 results of the query.
- ✓ Example1: Find all customers who have an account at all branches located in Cuttack.

  A ← $\Pi_{\text{customer\_name, branch\_name}}$ (depositor $\bowtie$ Account)

  B ← $\Pi_{\text{branch\_name}}$ ($\sigma_{\text{branch\_city="Cuttack"}}$ Branch)

  Result ← A ÷ B
- ✓ Example2: List all the employees who work on all projects where "Smith" works.

  R1← Employee $\bowtie$ Works_On

  R2 ← $\Pi_{\text{FNAME, LNAME, PNO}}$ (R1)

  R3 ← Π_{PNO} ($\sigma_{\text{FNAME="Smith"}}$ (R1))

  Result ← R2 ÷ R3

Extended Relational-Algebra Operations:

1. **The Generalized Projection:**
   - ✓ The generalized projection operation extends the projection operation by allowing arithmetic functions to be used in the projection list.
   - ✓ **Notation:** $\Pi_{F1,F2,\ldots Fn}(E)$
     where E is any relational expression, and each of F1, F2, …,Fn is an arithmetic expression involving constants and attributes in the schema of E.
   - ✓ **Example:** Given relation *credit_info(customer_name, limit, credit_balance)*, find how much more each person can spend:
     $\Pi_{\text{customer\_name, limit-credit\_balance}}$ (credit_info)
2. **The Aggregate Functions:**
   - ✓ The aggregate function takes a collection of values and returns a single value as a result.
   - ✓ The aggregate function in relational algebra are:
     - **avg**: average value
     - **min**: minimum value
     - **max**: maximum value
     - **sum**: sum of values
     - **count**: number of values
   - ✓ **Notation:** $_{G1, G2, \ldots, Gn}\mathcal{G}_{F1(A1), F2(A2), \ldots, Fn(An)}(E)$
     where E is any relational expression, G1, G2, …, Gn is a list of attributes on which to group (may be empty), each F_i is an aggregate function and A_i is an attribute name.
   - ✓ **Example:**
     Account (group by branch-name)
   - ✓

branch_name	account_number	balance
Perryridge	A-102	400
Perryridge	A-201	900
Brighton	A-217	750
Brighton	A-215	750
Redwood	A-222	700

branch_name $\mathcal{G}$ **sum**(balance) (account)

branch_name	sum(*balance*)
Perryridge	1300
Brighton	1500
Redwood	700

   - ✓ For convenience result of aggregate function can have a name.
     Example: branch_name $\mathcal{G}$ **sum**(balance) as sum_balance (account)
3. **The Outer Join:**
   - ✓ An extension of the join operation that avoids loss of information.
   - ✓ Computes the join and then adds tuples form one relation that does not match tuples in the other relation to the result of the join.
   - ✓ The **left outer join** takes all tuples in the left relation that did not match with any tuple in the right relation, pads the tuples with null values for all other attributes from the right relation and adds them to the result of the natural join.

- ✓ The **right outer join** takes all tuples in the right relation that did not match with any tuple in the left relation, pads the tuples with null values for all other attributes from the left relation and adds them to the result of the natural join.
- ✓ The **full outer join** does both of those operations, padding the tuples from the left relation that did not match any from the right relation, as well as tuples from the right relation that did not match any from left relation, and adding them to the result of the natural join.
- ✓ **Example:**

Loan Relation:

loan_number	branch_name	amount
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

Borrower Relation:

customer_name	loan_number
Jones	L-170
Smith	L-230
Hayes	L-155

Join: Loan ⋈ Borrower

loan_number	branch_name	amount	customer_name
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

Left Outer Join: Loan ⟕ Borrower

loan_number	branch_name	amount	customer_name
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	Null

Right Outer Join: Loan ⟖ Borrower

loan_number	branch_name	amount	customer_name
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	null	null	Hayes

Full Outer Join: Loan ⟗ Borrower

loan-number	Branch_name	Amount	Customer_name
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	Null
L-155	Null	Null	Hayes

4. Null Values:

- ✓ It is possible for tuples to have a null value, denoted by *null*, for some of their attributes.
- ✓ *null* signifies an unknown value or that a value does not exist.
- ✓ The result of any arithmetic expression (such as +, -, *, / etc) involving *null* is *null.*
- ✓ Aggregate functions simply ignore null values.
- ✓ For duplicate elimination and grouping, null is treated like any other value, and two nulls are assumed to be the same.
- ✓ Comparisons (such as >, <, <=, >=, ≠) with null values return the special truth value: *unknown*
- ✓ Boolean operation with truth value *unknown*:

- **OR:** (*unknown* **or** *true*) = *true*,
  (*unknown* **or** *false*) = *unknown*
  (*unknown* **or** *unknown*) = *unknown*
- **AND:** (*true* **and** *unknown*) = *unknown*,
  (*false* **and** *unknown*) = *false*,
  (*unknown* **and** *unknown*)= *unknown*
- **NOT:** (**not** unknown) = unknown

Modification of the Database

✓ The content of the database may be modified using the operations: Deletion, Insertion, and Updating.

✓ All these operations are expressed using the assignment operator.

✓ **Deletion:**

- A delete request is expressed similarly to a query, except instead of displaying tuples to the user, the selected tuples are removed from the database.
- It can delete the entire tuple. It cannot delete values of some attributes.
- A deletion is expressed in relational algebra by:
  $r \leftarrow r - E$, where r is a relation and E is a relational algebra query.
- Example1: Delete all account holders in the "Baramunda" branch.
  $account \leftarrow account - \sigma_{branch\_name = "Baramunda"} (account)$
- Example2: Delete all records with amount in range of 0 to 50.
  $loan \leftarrow loan - \sigma_{amount>=0 \text{ and } amount <= 0} (loan)$

✓ **Insertion:**

- To insert data into a relation, we either specify a tuple to be inserted or we write a query whose result is a set of tuples to be inserted.
- In relational algebra, an insertion is expressed by:
  $r \leftarrow r \cup E$, where r is a relation and E is a relational algebra expression.
- The insertion of a single tuple is expressed by letting E be a constant relation containing one tuple.
- Example1: Insert information in the database specifying that Smith has $1200 in account A-973 at the Perryridge branch.
  $account \leftarrow account \cup \{("A-973", "Perryridge", 1200)\}$
  $depositor \leftarrow depositor \cup \{("Smith", "A-973")\}$

✓ **Updation:**

- A mechanism to change a value in a tuple without changing all values in the tuple.
- Use the generalized projection operator to do this task:
  $r \leftarrow \Pi_{F1, F2, ..., Fn} (r)$, where each F_i is either the i^{th} attribute of r, if the i^{th} attribute is not updated, or, if the attribute is to be updated, F_i is an expression, involving only constants and the attributes of r that gives the new value of attribute.
- Example1: Make interest payments by increasing all balances by 5 percent.
  $account \leftarrow \Pi_{account\_number, branch\_name, balance * 1.05} (account)$

Relational Calculus:

- ✓ It is a non-procedural language.
- ✓ A relational calculus expression specifies what is to be retrieved rather than how to retrieve it.

EMPLOYEE

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO

DEPARTMENT

DNAME	DNUMBER	MGRSSN	MGRSTARTDATE

DEPT_LOCATIONS

DNUMBER	DLOCATION

PROJECT

PNAME	PNUMBER	PLOCATION	DNUM

WORKS_ON

ESSN	PNO	HOURS

DEPENDENT

ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP

The Tuple Relational Calculus:

- ✓ The tuple relational calculus is based on specifying a number of tuple variables.
- ✓ A simple tuple relational calculus query is of the form:

  {t | COND (t) }

  where t is a tuple variable and COND(t) is a conditional expression involving t. The result of such a query is the set of all tuples t that satisfy COND(t).
- ✓ **Example1:** Find all employees whose salary is above 10000

  {t | EMPLOYEE (t) and t.SALARY>10000} **[Retrieves all attributes of t]**
- ✓ **Example2:** Find only the first name of employees whose salary is above 10000

{t.FNAME | Employee (t) and t.SALARY>10000}**[Retrieves FName attribute of t]**

- ✓ A general expression of the tuple relational calculus is of the form:

$$\{t_1.A_j, t_2.A_k, \ldots, t_n.A_m \mid COND(t_1, t_2, \ldots, t_n, t_{n+1}, t_{n+2}, \ldots, t_{n+m})\}$$

where $t_1, t_2, \ldots, t_n, t_{n+1}, t_{n+2}, \ldots, t_{n+m}$ are tuple variables, each A_i is an attribute of the relation on which t_i ranges and COND is a condition or formula.

- ✓ **Example3:** Retrieve the name and address of all employees wo work for "Research" Department.

{t.FNAME, t.LNAME, t.ADDRESS | EMPLOYEE(t) AND (∃d) (DEPARTMENT(d) AND d.DNAME='Research'AND d.DNUMBER=t.DNO)}

- ✓ **Example4:** For every project located in 'Chennai', list the project number, the controlling department number, and the department manager's last name, date of birth and address.

{p.PNUMBER, p.DNUM, m.LNAME, m.DOB, m.ADDRESS | PROJECT(p) AND EMPLOYEE(m) AND p.PLOCATION='Chennai' AND ((∃d)(DEPARTMENT(d) AND p.DNUM=d.DNUMBER AND d.MGRENO=m.ENO}

- ✓ **Example5:** For each employee, retrieve the employee's first and last name and the first and last name of his/her immediate supervisor.

{e.FNAME, e.LNAME, s.FNAME, s.LNAME | EMPLOYEE(e) AND EMPLOYEE(s) AND e.SUPERSENO=s.ENO}

- ✓ **Example6:** Find the name of each employee who works on some project controlled by the department number 5.

{e.LNAME, e.FNAME | EMPLOYEE(e) AND ((∃x)(∃w) (PROJECT(x) AND WORKS_ON(w) AND x.DNUM=5 AND w.EENO=e.ENO AND x.PNUMBER=w.PNO}

- ✓ **Example7:** Make a list of projects numbers for projects that involve an employee whose last name is 'Kumar', either as a worker or as manager of the controlling department for the project.

{p.PNUMBER | PROJECT(p) AND ((∃e)(∃w)(EMPLOYEE(e) AND WORKS_ON(w) AND w.PNO=p.PNUMBER AND e.LNAME='Kumar' AND e.ENO=w.EENO)) or ((∃m)(∃d)(EMPLOYEE(m) AND DEPARTMENT(d) AND p.DNUM=d.DNUMBER AND d.MGRENO=m.ENO AND m.LNAME='Kumar'))))}

- ✓ **Example6A:** Find the name of each employee who works on all project controlled by the department number 5.

{e.LNAME, e.FNAME | EMPLOYEE(e) AND ((∀x) (NOT(PROJECT(x)) OR NOT(x.DNUM=5) OR ((∃w) (WORKS_ON(w) AND w.EENO=e.ENO AND x.PNUMBER=w.PNO)))))}

- ✓ **Example6B:** Find the name of each employee who works on all project controlled by the department number 5.

{e.LNAME, e.FNAME | EMPLOYEE(e) AND (NOT (∃x) (PROJECT(x) AND (x.DNUM=5) AND (NOT (∃w) (WORKS_ON(w) AND w.EENO=e.ENO AND x.PNUMBER=w.PNO)))))}

- ✓ **Example8:** Find the name of each employee who have no dependants.

{e.LNAME, e.FNAME | EMPLOYEE(e) AND (NOT (∃d) (DEPENDANT(d) AND e.ENO=d.EENO))}

- ✓ **Example8A:** Find the name of each employee who have no dependants.

{e.LNAME, e.FNAME | EMPLOYEE(e) AND ((∀d) (NOT(DEPENDANT(d)) OR NOT(e.ENO=d.EENO)))}

- ✓ **Example9:** List the names of managers who have at least one dependant.

{e.LNAME, e.FNAME | EMPLOYEE(e) AND ((∃d)(∃p)(DEPARTMENT(d) AND DEPENDANT(p) AND e.ENO=d.MGRENO AND p.EENO=e.ENO))}

- ✓ **Safe Expression:** An expression is said to be safe if all its result are from the domain of the expression.

Example: {t | NOT(EMPLOYEE(t))} is an unsafe expression.

The Domain Relational Calculus:

- ✓ The domain relational calculus is based on the variables range over single values from domain of attributes. To form a relation of degree n for a query result, we must have n of these domain variables-one for each attribute.
- ✓ A domain relational calculus query is of the form:

$$\{ x_1, x_2, \ldots, x_n \mid \text{COND}(x_1, x_2, \ldots, x_n, x_{n+1}, x_{n+2}, \ldots, x_{n+m}) \}$$

where $x_1, x_2, \ldots, x_n, x_{n+1}, x_{n+2}, \ldots, x_{n+m}$ is are domain variables that range over domains and COND is a condition or formula of the domain relational calculus.

- ✓ **Example1:** Retrieve the date of birth and address of the employee whose name is 'Mahesh B. Kumar'

{uv | (∃q) (∃r) (∃s) (∃t) (∃w) (∃x) (∃y) (∃z) (Employee(qrstuvwxyz) AND q='Mahesh' AND r='B.' AND s='Kumar')}

- ✓ **Example2:** Retrieve the name and address of all employees who work for the 'Research' Department.

{qsv | (∃z) (∃l) (∃m) (Employee(qrstuvwxyz) AND Department(lmno) AND l='Research' AND m=z)}

- ✓ **Example3:** For every project located in 'Chennai', list the project number, the controlling department number and the department manager's last name, date of birth and address.

{iksuv | (∃j) (∃m) (∃n) (Project(hijk) AND Employee(qrstuvwxyz) AND Department(lmno) AND k=m AND n=t AND j='Chennai')}

- ✓ **Example4:** Find the name of employees who have no dependents.

{qs | (∃t) (Employee(qrstuvwxyz) AND (NOT(∃l) (Dependent(lmnop) AND t=l)))}

✓ **Example4A:** Find the name of employees who have no dependents.

{qs | (∃t) (Employee(qrstuvwxyz) AND ((∃l) (NOT(Dependent(lmnop)) OR NOT(t=l))))}

✓ **Example5:** List the name of managers who have at least one dependent.

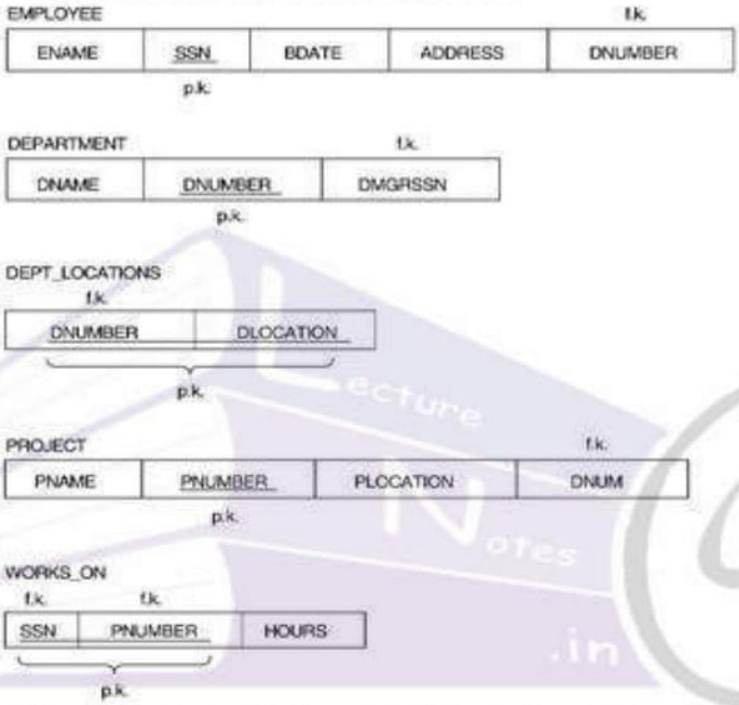
{sq | (∃t) (∃j) (∃l) (Employee(qrstuvwxyz) AND Department(hijk) AND Dependent(lmnop) AND t=j AND l=t)}

Relational Database Design

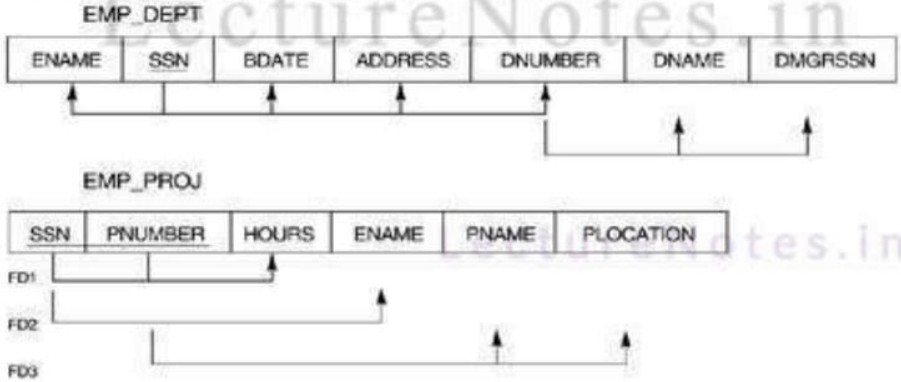
Informal Design Guidelines for Relational Schemas:

1. Semantics of the relation attributes

- **GUIDELINE 1:** Design a relation schema so that it is easy to explain its meaning. Do not combine attributes from multiple entity types and relationship types into a single relation.
- Attributes of different entities (EMPLOYEEs, DEPARTMENTs, PROJECTs) should not be mixed in the same relation.
- Only foreign keys should be used to refer to other entities.
- Entity and relationship attributes should be kept apart as much as possible.
- A simplified version of company relational database schema is given as:



- Some of the poor designs of relation schema are:



- EMP_DEPT mixes attributes of employees and departments (Fig a), and EMP_PROJ mixes attributes of employees and projects (Fig b).

2. Redundant Information in Tuples and Update Anomalies

- By mixing attributes of multiple entities we have to store the information redundantly which wastes storage space.
- Redundant information storage creates problems with the following three **update anomalies:**

- **Insertion anomalies:** To insert a new employee tuple into EMP_DEPT we must insert some values for the attributes like DNUMBER, DNAME, DMGRSSN etc or NULL values. If we are inserting some values in these attributes then these values should be consistent with the values in other tuples for the same department.

  It is difficult to insert a new department that has no employees. The only way to insert is store NULL values in the attributes like ENAME, SSN etc. But as SSN is primary key so we can't store NULL here.
- **Deletion anomalies:** If we delete the employee tuple of the last employee of a department from EMP_DEPT then department information of that department will be lost.
- **Modification anomalies:** In EMP_DEPT, if we change the value of one of the attributes (eg. DMGRSSN) of a particular department we must update the tuples of all employees who work in that department to make the database consistent.

redundancy

EMP_DEPT

ENAME	SSN	BDATE	ADDRESS	DNUMBER	DNAME	DMGRSSN
Smith,John B.	123456789	1965-01-09	731 Fondren,Houston,TX	5	Research	333445555
Wong,Frankin T.	333445555	1955-12-08	638 Voss,Houston,TX	5	Research	333445555
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle,Spring,TX	4	Administration	987654321
Wallace,Jennifer S.	987654321	1941-06-20	291 Berry,Bellaire,TX	4	Administration	987654321
Narayan,Ramesh K.	666884444	1962-09-15	975 FireOak,Humble,TX	5	Research	333445555
English,Joyce A.	453453453	1972-07-31	5631 Rice,Houston,TX	5	Research	333445555
Jabbar,Ahmad V.	987987987	1969-03-29	980 Dallas,Houston,TX	4	Administration	987654321
Borg,James E.	888665555	1937-11-10	450 Stone,Houston,TX	1	Headquarters	888665555

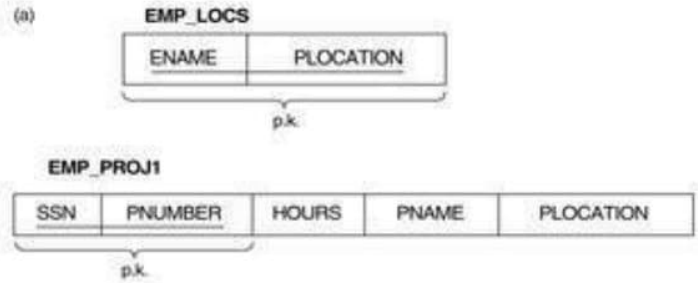
- **GUIDELINE 2:** Design the base relation schema so that no insertion, deletion and modification anomalies are present in the relations. If any anomalies are present, note them clearly and make sure that the programs that update the database will operate correctly.

3. Null Values in Tuples

- If many of the attributes do not apply to all tuples in a relation we find many NULL values in a relation.
- Many NULL values can waste space at the logical level.
- NULL values also does not account for the aggregate operations like COUNT, SUM etc.
- Null values can have multiple interpretations like:
  - Attribute not applicable or invalid.
  - Attribute value for this tuple is unknown.
  - Attribute value is known but absent.
- **GUIDELINE 3:** As far as possible, avoid placing attributes in a base relation whose values may frequently be NULL. If NULLS are unavoidable, make sure that they apply in exceptional cases only and do not apply to a majority of tuples in the relation.
- Attributes that are NULL frequently could be placed in separate relations (with the primary key).

4. Spurious Tuples

- Let us decompose the EMP_PROJ relation into two relations EMP_LOCS and EMP_PROJ1 as follows.
- But when we join back them using NATURAL join we do not get the correct original information.

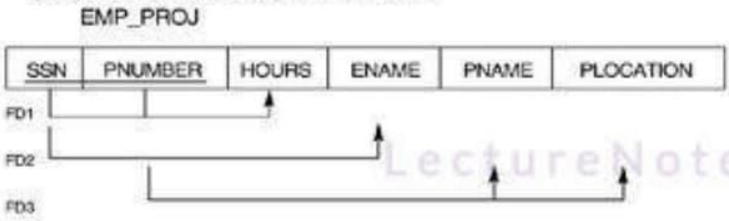


- This happens because PLOCATION is the attribute that relates EMP_LOCS and EMP_PROJ1 but PLOCATION is neither primary key nor foreign key in any of these two relations.
- **GUIDELINE 4:** Design relation schema so that they can be joined with equality conditions on attributes that are either primary keys or foreign keys in a way that guarantees that no spurious tuples are generated. Do not have relations that contain matching attributes other than foreign key-primary key combinations. If such relations are unavoidable, do not join them on such attributes, because they may produce spurious tuples.

Functional Dependencies

1. Functional Dependency

- FDs are constraints that are derived from the meaning and interrelationships of the data attributes.
- **Definition:** Let R is a relation schema with n attributes A1, A2,…,An. A functional dependency, denoted by X→Y, between two set of attributes X and Y that are subsets of R specifies a constraint on the possible tuples that can form a relation state r of R. The constraint is that, for any two tuples t_1 and t_2 in r that have $t_1[X] = t_2[X]$, they must have $t_1[Y] = t_2[Y]$.
- Y depends on X or Y is determined by X or Y is functionally dependent on X. Similarly X determines Y.
- If X is the candidate key of R then X→Y for any subset of attributes Y of R
- If X→Y in R, this does not say whether or not Y→X in R.
- Relation extensions r(R) that satisfy the functional dependency constraints are called legal relation states (or legal extensions) of R.
- **Example 1:** SSN→ENAME
  PNUMBER→{PNAME, PLOCATION}
  {SSN, PNUMBER}→HOURS

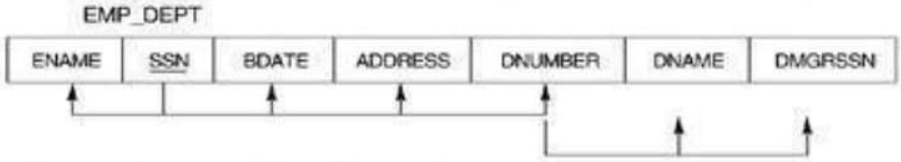


- **Example 2:** AB→C

A	B	C	D
a1	b1	c1	d1
a1	b1	c1	d2
a1	b2	c2	d1
a2	b1	c3	d1

2. Inference Rules for Functional Dependencies

- **Definition:** Let F is a set of all functional dependencies. The set of all dependencies that include F as well as all dependencies that can be inferred from F is called the **closure** of F denoted by F^+.
- **Example:** F= { SSN→{ENAME, BDATE, ADDRESS, DNUMBER},
  DNUMBER→{DNAME, DMGRSSN} }



F^+= { SSN→{ENAME, BDATE, ADDRESS, DNUMBER},
DNUMBER→{DNAME, DMGRSSN},
SSN→{DNAME, DMGRSSN},
ENO→ENO
DNUMBER→DNAME }

- To determine a systematic way to infer dependencies, we must discover a set of inference rules that can be used to infer new dependencies from a given set of dependencies.
- The notation F | = X→Y denote the functional dependency X→Y is inferred from the set of functional dependencies F.
- **Inference Rules:**
  - **IR1 (Reflexive Rule):** If X ⊇ Y, then X→Y.
  - **IR2 (Augmentation Rule):** {X→Y}| = XZ→YZ.
  - **IR3 (Transitive Rule):** {X→Y, Y→Z }| = X→Z.
  - **IR4 (Decomposition Rule):** {X→YZ}| = X→Y.
  - **IR5 (Union or Additive Rule):** {X→Y, X→Z }| = X→YZ.
  - **IR6 (Pseudotransitive Rule):** {X→Y, WY→Z }| = WX→Z.
- IR1, IR2 and IR3 is known as **Armstrong's Axiom** or **Armsrong's Inference Rule.**
- A functional dependency X→Y is **trivial** if X ⊇ Y; otherwise it is **nontrivial**.
- **Proof of Inference Rules**
  - **Proof of IR1**

    Suppose X ⊇ Y and there exists at least two tuples t_1 and t_2 in some instance r of R such that $t_1[X] = t_2[X]$. But as X ⊇ Y definitely $t_1[Y] = t_2[Y]$.
  - **Proof of IR2**

    Suppose X→Y holds and XZ→YZ does not hold in a relation instance r of R. Suppose there exists two tuples t_1 and t_2 such that (1) $t_1[X]=t_2[X]$ (2)$t_1[Y]=t_2[Y]$ (3) $t_1[XZ]=t_2[XZ]$ and (4) $t_1[YZ]\neq t_2[YZ]$. From (1) and (3) we get (5) $t_1[Z]=t_2[Z]$. From (2) and (5) we get (6) $t_1[YZ]=t_2[YZ]$ which contradicts (4)
  - **Proof of IR3**

    Let X→Y and Y→Z both hold in a relation r. For any two tuples t_1 and t_2 if $t_1[X] = t_2[X]$ we must have $t_1[Y] = t_2[Y]$. From this and Y→Z we have $t_1[Z] = t_2[Z]$. So X→Z
  - **Proof of IR4**

    It is given that X→YZ. As YZ ⊇ Y implies YZ→Y. From this two we get X→Y

- **Proof of IR5**

  It is given that (1) X→Y and (2) X→Z. Using IR2 on 1 by augmenting X we get (3) X→XY. Similarly using IR2 on 2 by augmenting Y we get (4)XY→YZ. Using IR3 on 3 and 4 we get X→YZ.

- **Proof of IR6**

  It is given that (1) X→Y and (2) WY→Z. Using IR2 on 1 by augmenting W we get (3) WX→WY. Using IR3 on 3 and 2 we get WX→Z.

➢ **Example (Applying IRs)**

R(A, B, C) with F={A→B, B→C}

Applying IR1: A→A, A→{}, B→B, B→{}, C→C, C→{}, AB→A, AB→B, AB→AB, AB→{}, AC→A, AC→C, AC→AC, AC→{}, BC→B, BC→C, BC→BC, BC→{}, ABC→A, ABC→B, ABC→C, ABC→AB, ABC→AC, ABC→BC, ABC→ABC, ABC→{}

Applying IR2: AC→BC, AB→AC, A→AB, B→BC

Applying IR3: A→C,

➢ The Inference Rules IR1, IR2 and IR3 are **sound**. That means given a set of functional dependencies F specified on a relation schema R, any dependency that we can infer from F by using IR1 through IR3 holds in every relation state r of R that satisfies dependencies in F.

➢ The Inference Rules IR1, IR2 and IR3 are **complete.** That means using IR1 through IR3 repeatedly to infer dependencies until no more dependencies can be inferred results in the complete set of all possible dependencies until no more dependencies can be inferred from F.

➢ **Attribute Closure** X^+ with respect to F, which is the set of attributes A such that X→A can be inferred using the Armstrong Axioms. The following algorithm finds the closure of X:

➢ **Algorithm 1: Determining X^+, the closure of X under F**

Closure=X;

Repeat until there is no change:

{

If there is an FD U→V in F such that U ⊆ Closure,

Then set Closure =Closure ∪ V

}

➢ **Example:** Consider a relation EMP_PROJ having the following functional dependencis:

F={SSN→ENAME, PNUMBER→ {PNAME, PLOCATION}, {SSN, PNUMBER}→HOURS}

$\{SSN\}^+$ = { SSN, ENAME}

$\{PNUMBER\}^+$ = {PNUMBER, PNAME, PLOCATION}

$\{SSN, PNUMBER\}^+$ = {SSN, PNUMBER, ENAME, PNAME, PLOCATION, HOURS}

➢ Closure of the attributes can be used to determine the keys of a relation.

3. Equivalence of Sets of Functional Dependencies

➢ **Definition:** A set of functional dependencies F is said to **cover** another set of functional dependencies E if every FD in E is also in F^+; that is, if every dependency in E can be inferred from F. Alternatively we can say E is **covered by** F.

➢ **Definition:** Two set of functional dependencies E and F are equivalent if $E^+=F^+$. That is E is equivalent to F if E covers F and F covers E.

To determine whether F covers E we calculate X^+ with respect to F for each FD X→Y in E and then check whether X^+ includes the attributes Y.

Example: R=(A, B, C, D, E, F)

F_1={A→BC, B→CDE, AE→F}, F_2={A→BCF, B→DE, E→AB}

Check whether F_1 and F_2 are equivalent or not.

Answer: To check F_1 covers F_2:

A^+={A, B, C, D, E, F} contains B, C, F

B^+={B, C, D, E, } contains D, E

E^+={E} doesn't contain A, B

So F_1 doesn't covers F_2. Hence F_1 and F_2 are not equivalent.

- **Example:** R=(A, C, D, E, H)

  F_1={A→C, AC→D, E→AD, E→H}, F_2={A→CD, E→AH }

  Check whether F_1 and F_2 are equivalent or not.

Answer: To check F_1 covers F_2:

A^+={A, C, D } contains C, D

E^+={A, D, E, H} doesn't contain A, H

So F_1 covers F_2.

To check F_2 covers F_1:

A^+={A, C, D} contains C

$\{A,C\}^+$={A, C, D} contains D

E^+={A, C, D, E, H} contains A, D, H

So F_2 covers F_1.

Hence F_1 and F_2 are equivalent.

4. Minimal Set of Functional Dependencies / Minimal Cover / Canonical Cover

- **Definition:** A minimal cover of a set of functional dependencies E is a minimal set of dependencies F that is equivalent to E
- **Formal Definition:** A set of functional dependencies F to be minimal if it satisfies the following conditions:
  1. Every dependency in F has a single attribute for its right hand side.
  2. We cannot replace any dependency X→A in F with a dependency Y→A, where Y is a proper subset of X, and still have a set of dependencies that is equivalent to F.
  3. We cannot remove any dependency from F and still have a set of dependencies that is equivalent to F.
- **Algorithm:** Finding Minimal Cover for a set of Functional Dependencies E
  1. Set F:=E.
  2. Replace each functional dependency X→{A_1, A_2, . . . , A_n} in F by the n functional dependencies X→A_1, X→A_2, …, X→A_n.
  3. For each functional dependency X→A in F for each attribute B that is an element of X if {{F-{X→A}} ∪ {(X-{B})→A}} is equivalent to F, then replace X→A with (X-{B})→A in F.
  4. For each remaining functional dependency X→A in F if {F-{X→A}} is equivalent to F, then remove X→A from F.
- **Problem 1:** Consider a relation R(A, B, C) with the functional dependencies set E={A→BC, B→C}. Find the minimal cover of E.

Answer:

Step1:

F={A→BC, B→C}

Step 2:

F={A→B, A→C, B→C}

Step 3:

Not applicable to the current problem.

Step 4:

Check for {A→B, A→C, B→C} is equivalent to { A→C, B→C}

Check for {A→B, A→C, B→C} is equivalent to { A→B, B→C}

Check for {A→B, A→C, B→C} is equivalent to { A→B, A→C}

We find only{A→B, A→C, B→C} and { A→B, B→C} are equivalent.

So F= {A→B, B→C}

- **Problem 2:** Consider a relation R(A, B, C, D) with the functional dependencies set E={A→BC, B→C, AC→D}. Find the minimal cover of E.

Answer:

Step1:

F={A→BC, B→C, AC→D}

Step 2:

F={A→B, A→C, B→C, AC→D}

Step 3:

Check for {A→B, A→C, B→C, AC→D} and {A→B, A→C, B→C, C→D} are equivalent or not.

Check for {A→B, A→C, B→C, AC→D} and {A→B, A→C, B→C, A→D} are equivalent or not.

We find {A→B, A→C, B→C, AC→D} and are equivalent.

So F={A→B, A→C, B→C, A→D}

Step 4:

Check for {A→B, A→C, B→C, A→D} is equivalent to {A→C, B→C, A→D}

Check for {A→B, A→C, B→C, A→D} is equivalent to {A→B, B→C, A→D}

Check for {A→B, A→C, B→C, A→D} is equivalent to {A→B, A→C, A→D}

Check for {A→B, A→C, B→C, A→D} is equivalent to {A→B, A→C, B→C}

We find only {A→B, A→C, B→C, A→D} is equivalent to {A→B, B→C, A→D}are equivalent.

So F= {A→B, B→C, A→D}

Repeat Step 4 still F is irreducible. In fact we find F is irreducible.

- **Problem 3:** Consider a relation R(A, B, C, D) with the functional dependencies set F={A→BC, B→C, A→B, AB→C, AC→D }. Find the minimal cover of F.

Answer:

Step1:

Writing the FDs in such a way that each has a singleton right-hand side: A→B, A→C, B→C, A→B, AB→C, AC→D.

We get A→B two times. Removing that we get A→B, A→C, B→C, AB→C, AC→D.

Step 2:

From AC→D we can eliminate C because as we know A→C and by augmentation IR A→AC. But we have AC→D. So by transitivity A→D. So the C attribute in the left hand side of AC→D is redundant. Now F ={ A→B, A→C, B→C, AB→C, A→D}

Step 3:

We can eliminate AB→C because we have A→C. Applying augmentation IR AB→BC. From this using decomposition IR we get AB→C. Now F ={ A→B, A→C, B→C, A→D}

Step 4:

From A→B, B→C we get A→C. So A→C is redundant. Removing A→C we get F ={ A→B, B→C, A→D} which is the minimal cover because this F is irreducible.

Normal Forms Based on Primary Keys

1. Normalization of Relations

- **Normalization:** The process of decomposing unsatisfactory "bad" relations by breaking up their attributes into smaller relations.
- **Normal Form:** Condition using keys and FDs of a relation to certify whether a relation schema is in a particular normal form.
- **Normalization of data** can be looked upon as a process of analyzing the given relation schema based on their FDs and primary keys to achieve the desirable properties of
  - Minimizing redundancy and
  - Minimizing the insertion, deletion and update anomalies.
- While decomposition normalization process should ensure the following two properties are satisfied:
  - **Lossless join or nonadditive property:** It guarantees that the spurious tuples are not generated with respect to the relation schemas created after decomposition.
  - **Dependency Preservation Property:** It ensures that each functional dependency is represented in some individual relation resulting after decomposition.
- **Denormalization:** It is the process of storing the join of higher normal form relations as a base relation—which is in a lower normal form.

2. Definitions of Keys and Attributes Participating in Keys.

- **Superkey:** A superkey of a relation schema $R = \{A_1, A_2, ..., A_n\}$ is a set of attributes S subset-of R with the property that no two tuples t_1 and t_2 in any legal relation state r of R will have $t_1[S] = t_2[S]$.
- **Key:** A Key K is a superkey with the additional property that removal of any attribute from K will cause K not to be a superkey any more.
- **Candidate Key & Secondary Key:** If a relation schema has more than one key, each is called a candidate key. One of the candidate keys is arbitrarily designated to be the **primary key,** and the others are called **secondary keys.**
- **Prime Attribute:** A prime attribute must be a member of some candidate key.
- **Nonprime Attribute:** It is not a member of any candidate key.

3. First Normal Form (1NF)

- **1NF states that the domain of an attribute must include only atomic (simple, indivisible) values.**
- 1NF disallows having a set of values (in case of multi valued attribute), a tuple of values (in case of composite attribute) or a combination of both as an attribute value for a single tuple.
- **Solution:** Form new relations for each non-atomic attribute or nested relations.
- **Example1**: (a) DEPARTMENT relation schema not in 1NF (b) Extension of DEPARTMENT relation (c) 1NF relation with redundancy (d) 1NF relation without redundancy.

(a)

DEPARTMENT

DNAME	DNUMBER	DMGRSSN	DLOCATIONS

(b)

DEPARTMENT

DNAME	DNUMBER	DMGRSSN	DLOCATIONS
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

(c)

DEPARTMENT

DNAME	DNUMBER	DMGRSSN	DLOCATION
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

(d)

DEPARTMENT

DNAME	DNO	DMGRSSN
Research	5	333445555
Administration	4	987654321
Headquarters	1	888665555

DEPT_LOC

DNO	DLOCATIONS
5	Bellaire
5	Sugarland
5	Houston
4	Stafford
1	Houston

➢ **Example2**: (a) EMP_PROJ (with a nested relation relation schema) not in 1NF (b) Extension of DEPARTMENT relation showing **nested relation** in each tuple. (c) Decomposing EMP_PROJ into EMP_PROJ1, EMP_PROJ2.

(a)

EMP_PROJ

SSN	ENAME	PROJS	
		PNUMBER	HOURS

(b)

EMP_PROJ

SSN	ENAME	PNUMBER	HOURS
123456789	Smith,John B.	1	32.5
		2	7.5
666884444	Narayan,Ramesh K.	3	40.0
453453453	English,Joyce A.	1	20.0
		2	20.0
333445555	Wong,Franklin T.	2	10.0
		3	10.0
		10	10.0
		20	10.0

(c)

EMP_PROJ1

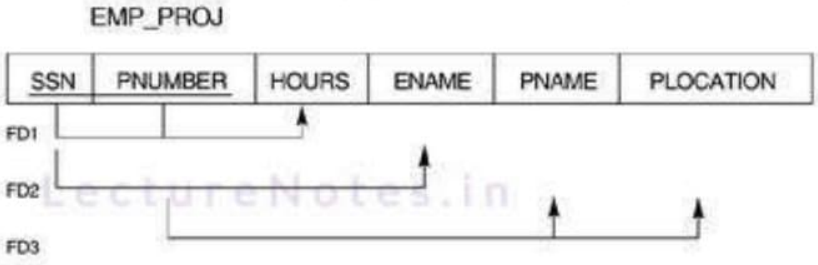
SSN	ENAME

EMP_PROJ2

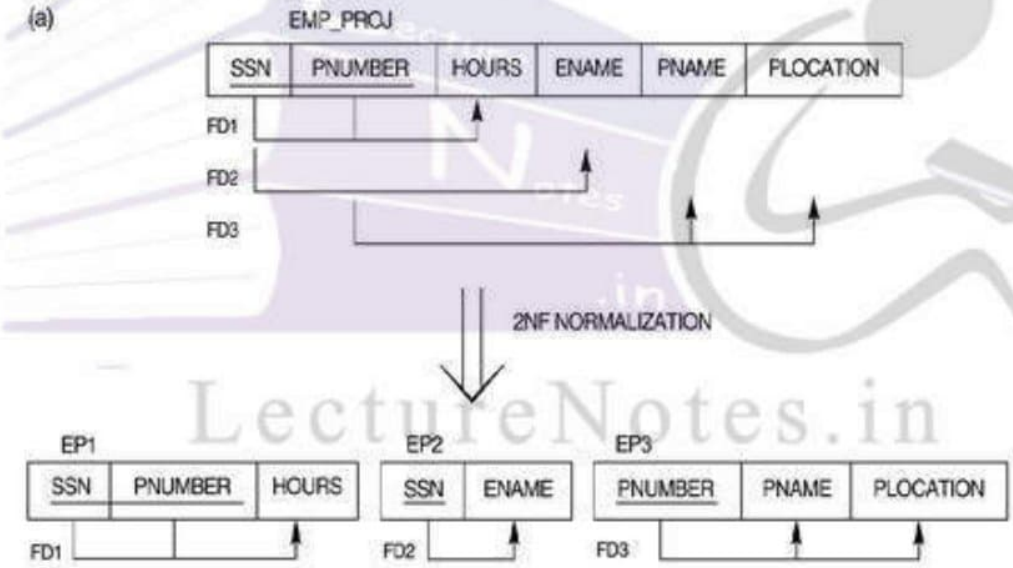
SSN	PNUMBER	HOURS

4. Second Normal Form (2NF)

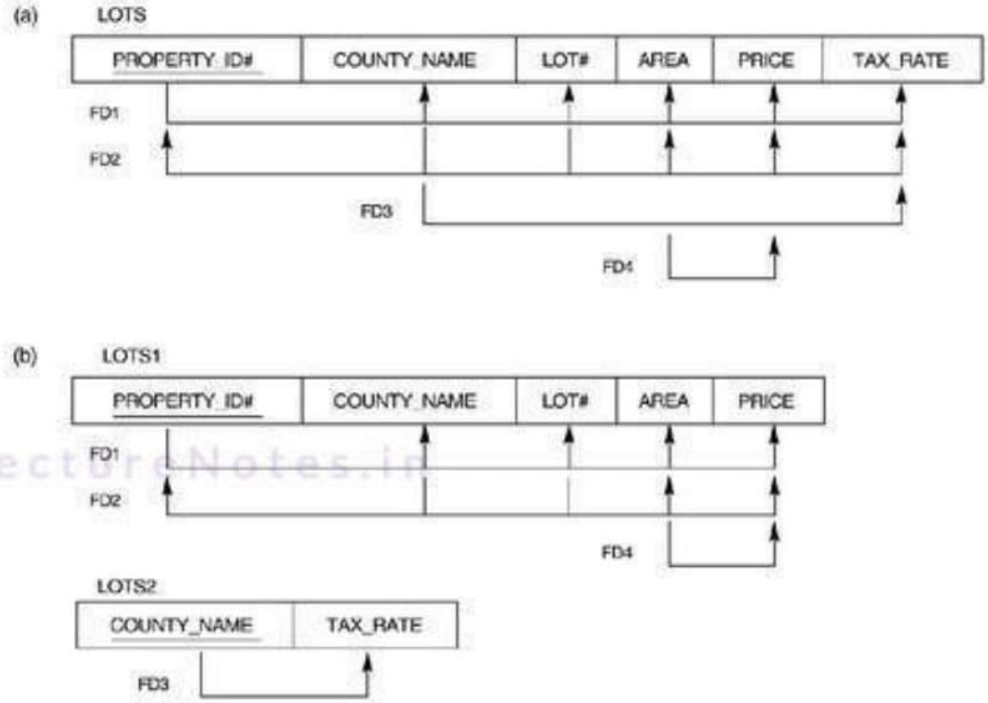
- **Full Functional Dependency:** A FD X→Y is said to be **Full functional dependency** when for any A∈X, (X-{A}) does not functionally determines Y. If (X-{A})→Y then we say X→Y is **partial dependency**.
- **Example:** {SSN, PNUMBER}→HOURS is a full functional dependency where as {SSN, PNUMBER}→ENAME is a partial functional dependency.



- **A relation is said to be in 2NF if it is in 1NF and if every nonprime attribute A in R is fully functionally dependent on any key (candidate) of R.**
- If the key contains a single attribute the test need not be applied at all.
- **Example1:** EMP_PROJ is in 1NF but it is not in 2NF because for the FD2 and FD3. In FD2 ENAME is dependent on SSN but it should be dependent on (SSN, PNUMBER). Similarly PNAME and PLOCATION are dependent on only PNUMBER but they should be dependent on (SSN, PNUMBER).

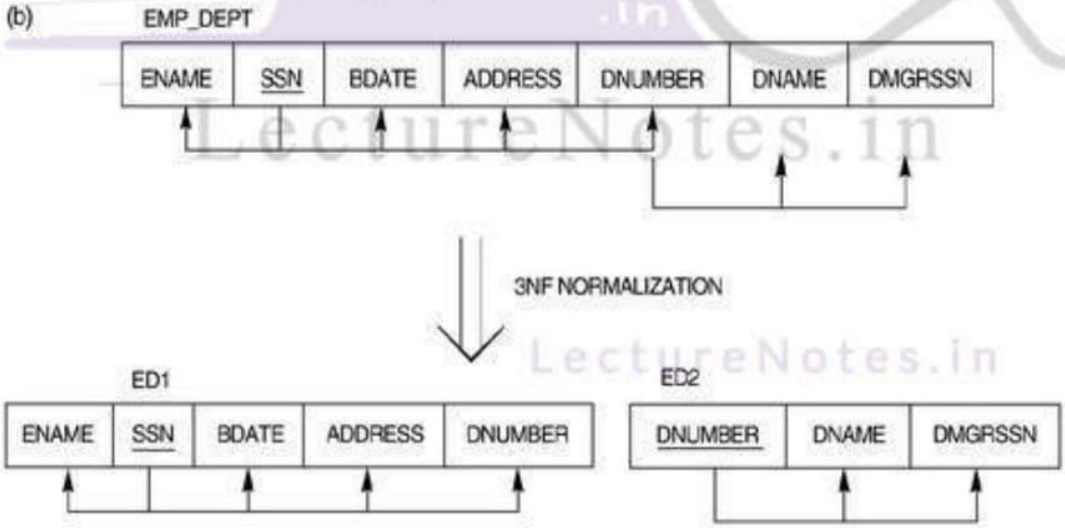


- **Solution:** Decompose and set up a new relation for each partial key with its dependent attributes. Make sure to keep a relation with the original primary key and any attribute that are fully functionally dependent on it.
- **Example2:** In the LOTS relation there are two candidate keys: PROPERTY_ID# and {COUNTY_NAME, LOT#}. In FD1 and FD2 no problem in full functionally dependent. FD3 is violating 2NF because the attribute TAX_RATE is dependent on only COUNTY_NAME but it should be dependent on {COUNTY_NAME and LOT#}. FD4 does not violate 2NF because the left hand side of FD4 that is AREA is not a key attribute.

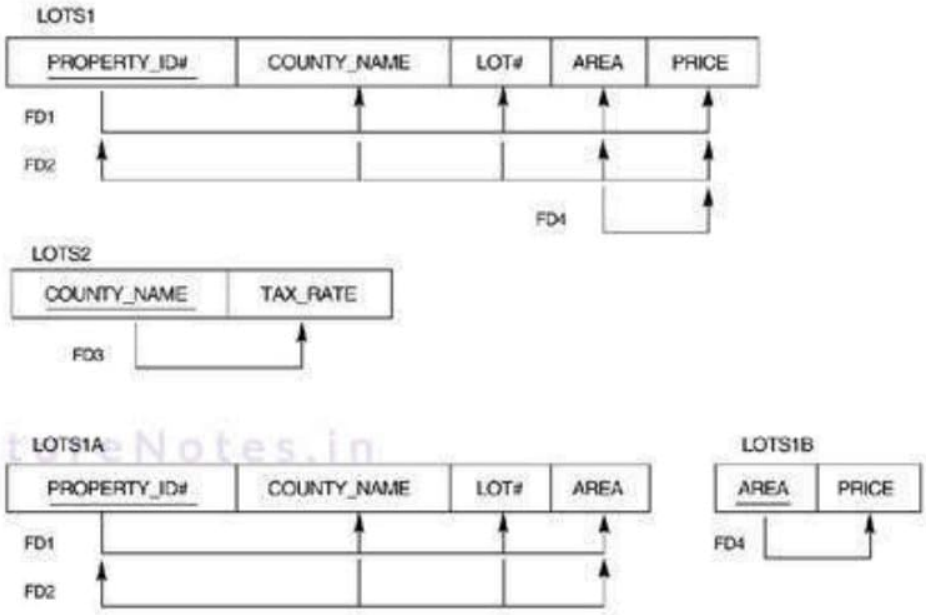


5. Third Normal Form (3NF)

- **Transitive Functional Dependency:** A FD X→Y in a relation schema is a **transitive dependency** if there is a set of attributes Z that is neither a candidate key nor a subset of any key of R and both X→Z and Z→Y holds
- **Example:** SSN→DMGRSSN is a transitive dependency through DNUMBER because we have SSN→DNUMBER and DNMUBER→DMGRSSN and DNUMBER is neither a key nor a part of key.
- **A relation is in 3NF if it is in 2NF and no nonprime attribute of R is transitively dependent on the primary key.**

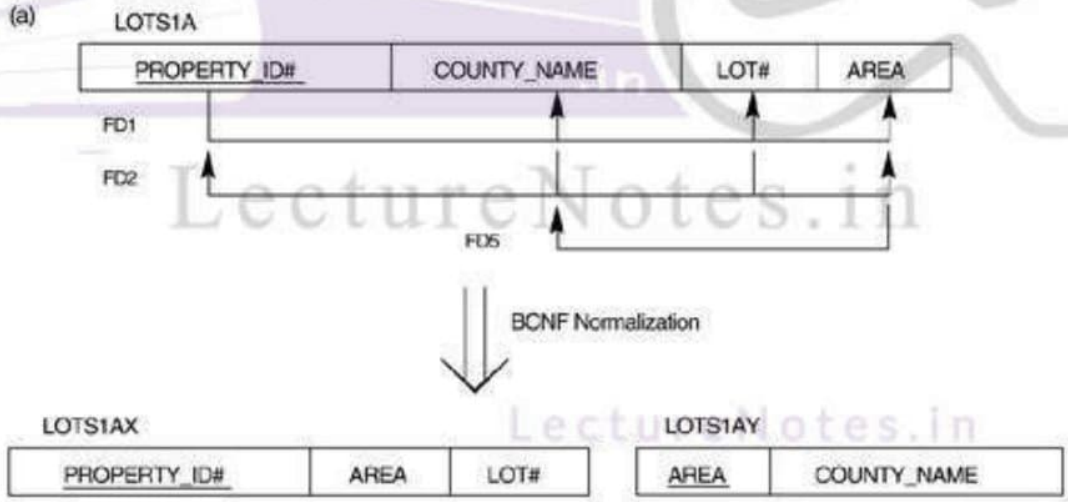


- **Solution:** Decompose and set up a relation that includes the non key attributes that functionally determine other non key attribute.
- **General Definition of 3NF:** A relation R is in 3NF if it is in 2NF and for every nontrivial functional dependency X→A holds in R either X is a super key or A is a prime attribute.
- **Example:** LOTS2 is in 3NF but LOTS1 is not in 3NF because for the FD4 as neither AREA is super key nor PRICE is prime attribute. To normalize LOTS1 into 3NF it is decomposed to LOTS1A and LOTS1B

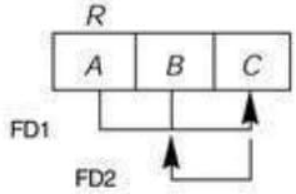


6. BOYCE-CODD Normal Form (BCNF)

- **A relation schema R is in BCNF if whenever a nontrivial functional dependency X→ A holds in R, then X is a super key of R.**
- BCNF is simpler and stricter than 3NF. Because every relation in BCNF is also in 3NF; however a relation in 3NF is not necessarily in BCNF.
- Any relation schema with only two attributes is automatically in BCNF.
- **Example:** In the following relation LOTS1A there are three FDs FD1, FD2 and FD5. But in FD5 AREA is not the super key. So we have to decompose it into two more relations LOTS1AX and LOTSA1Y to satisfy BCNF. This decomposition loses the functional dependency FD2.



- **Example: Relation in 3NF but not in BCNF.**



- **Example:** Consider the following relation TEACH having two given functional dependencies {Roll, Course}→Tutor and Tutor→Course. Here {Roll, Course} is the key. So it is in 3NF. But due to FD Tutor→Course it is not in BCNF. We can decompose the relation in three ways:

1. {Roll, Tutor} and {Roll, Course}
2. {Course, Tutor} and {Course, Roll}
3. {Tutor, Course} and {Tutor, Roll}

Out of these three alternatives third one is correct because the Tutor in this decomposition is the common attribute between the decomposed relations and it is primary key in one relation and foreign key in the other relation.

Roll	Course	Tutor
123	Physics	Einstein
123	Music	Muzrat
234	Biology	Darwin
675	Physics	Bohr
999	Physics	Einstein

Tutor	Course
Einstein	Physics
Muzrat	Music
Darwin	Biology
Bohr	Physics

Roll	Tutor
123	Einstein
123	Muzrat
234	Darwin
675	Bohr
999	Einstein

NOTE: All three decompositions loses the functional dependency {Roll, Course}→Tutor.

7. Properties of Relational Decompositions

- **Attribute Preservation condition:** Let R is a relation having attributes $A_1, A_2, \ldots, A_n$. Using the functional dependencies, the algorithms decompose the relation R into a set of relation schemas $D=\{R_1, R_2, \ldots, R_m\}$. We have to check that each attribute in R will appear in at least one relation schema R_i in the decomposition D so that no attributes are lost.

  Formally $\bigcup_{i=1}^{m} Ri = R$

- **Dependency Preservation:** Given a set of dependencies F on R, the projection of F on R_i, deonoted by $\pi_{Ri}(F)$ where R_i is a subset of R, is the set of dependencies $X \rightarrow Y$ in F^+ such that the attributes in $X \rightarrow Y$ are all contained in R_i. Hence the projection of F on each relation schema R_i in the decomposition D is the set of functional dependencies in F^+, the closure of F, such that all their left and right hand side attributes are in R_i.

  So a decomposition $D=\{R_1, R_2, \ldots, R_m\}$ of R is dependency preserving with respect to F if the union of the projections of F on each R_i in D is equivalent to F; that is $((\pi_{R1}(F)) \cup (\pi_{R2}(F) \cup \ldots \cup (\pi_{Rm}(F)))^+ = F^+$

- **Lossless (Nonadditive) Join Property:** A decomposition $D=\{R_1, R_2, \ldots, R_m\}$ of R has the lossless (nonadditive) join property with respect to the set of dependencies F on R if, for every relation state r of R that satisfies F, the following holds, where * is the NATURAL JOIN of all the relations in D:

  $*(\pi_{R1}(r), \ldots, \pi_{R2}(r)) = r$

Algorithm for Testing of Lossless Join Property

Input: A universal relation R, a decomposition $D= \{R_1, R_2, \ldots, R_m\}$ of R, and a set F of functional dependencies.

1. Create an initial matrix S with one row i for each relation R_i in D, and one column j for each attribute A_j in R.
2. Set $S(i,j) := b_{ij}$ for all matrix entries.
3. For each row i representing relation schema R_i

{

for each column j representing attribute A_j

{

if (relation R_i includes attribute A_j)

then set S(i,j) := a_j;

}

}

4. Repeat the following loop until a complete loop execution results in no changes to S

{

for each functional dependency X→F in F

{

for all rows in S that have the same symbols in the columns corresponding to attribute in X

{

Make the symbols in each column that corresponds to an attribute in Y be the same in all these rows as follows:

If any of the rows has an "a" symbol for the column, set the other rows to that same "a" symbol in the column.

If no "a" symbol exists for the attribute in any of the rows, choose one of the "b" symbol that appears in one of the rows for the attribute and set the other rows to that same "b" symbol in the column;

}

}

}

5. If a row is made up entirely of "a" symbols

then the decomposition has the lossless join property.

Else it does not.

8. Algorithm for Relational Database Schema Design

➢ **Dependency-Preserving Decomposition into 3NF Schemas:** The following algorithm creates a dependency-preserving decomposition D= {R_1, R_2, …, R_m}of a universal relation R based on a set of functional dependencies F, such that each R_i in D is in 3NF. It only guarantees only the dependency-preserving property. It doesn't guarantee the lossless join property.

Algorithm for Relation Synthesis into 3NF with Dependency Preservation

Input: A universal relation R and a set of functional dependencies F on the attributes of R.

1. Find a minimal cover G for F.
2. For each left-hand-side X of a functional dependency that appears in G, create a relation schema in D with attributes {X ∪{A_1} ∪{A_2}… ∪{A_k}}, where X→A_1, X→A_2, . . ., X→A_k are the only dependencies in G with X as the left hand side (X is the key of this relation);
3. Place any remaining attributes (that have not been placed in any relation) in a single relation schema to ensure the attribute preservation property.

➢ **Lossless (Nonadditive) Join Decomposition into BCNF Schemas:** The following algorithm decomposes a universal relation R={A_1,A_2,…A_n}into a decomposition D= {R_1, R_2, …, R_m} such that each R_i in D is in BCNF and the decomposition D has the lossless join property with respect to F.

Algorithm for Relational Decomposition into BCNF with Nonadditive Join Property:

Input: A universal relation R and a set of functional dependencies F on the attributes of R.

1. Set D:={R}.
2. While there is a relational schema Q in D that is not in BCNF do

{

Choose a relation schema Q in D that is not in BCNF;
Find a functional dependency X→Y in Q that violates BCNF;
Replace Q in D by two relation schemas (Q-Y) and (X∪Y)
}

- **Dependency-Preserving and Nonadditive (Lossless) Join Decomposition into 3NF Schemas:** The following algorithm creates a dependency-preserving decomposition D= {R_1, R_2, …, R_m}of a universal relation R based on a set of functional dependencies F, such that each R_i in D is in 3NF. It has the nonadditive (lossless) join property.

Algorithm for Relation Synthesis into 3NF with Dependency Preservation and Nonadditive (Lossless) Join Property

Input: A universal relation R and a set of functional dependencies F on the attributes of R.

1. Find a minimal cover G for F.
2. For each left-hand-side X of a functional dependency that appears in G, create a relation schema in D with attributes {X ∪{A_1} ∪{A_2}… ∪{A_k}}, where X→A_1, X→A_2, . . ., X→A_k are the only dependencies in G with X as the left hand side (X is the key of this relation);
3. If none of the relation schemas in D contains a key of R, then create one more relation schema in D that contains attributes that for a key of R.

9. Multivalued Dependencies and Fourth Normal Form

- **Multivalued Dependency:** A multivalued dependency X–>>Y specified on relation schema R, where X and Y are both subsets of R specifies the constraint on any relation state r of R: if two tuples t1 and t2 exist in r such that $t_1[X]=t_2[X]$, then two tuples t_3 and t_4 should exist in r with the following properties,

  $t_3[X]=t_4[X]= t_1[X]=t_2[X]$

  $t_3[Y]=t_1[Y]$ and $t_4[Y]=t_2[Y]$

  $t_3[R-(XUY)]=t_2[R-(XUY)]$ and $t_4[R-(XUY)]=t_1[R-(XUY)]$

- **Example:** The relation EMP with two MVDs: ENAME–>>PNAME and ENAME–>>DNAME.

EMP

ENAME	PNAME	DNAME
Smith	X	John
Smith	Y	Anna
Smith	X	Anna
Smith	Y	John

EMP_PROJECTS

ENAME	PNAME
Smith	X
Smith	Y

EMP_DEPENDENTS

ENAME	DNAME
Smith	John
Smith	Anna

- **Trivial MVD:** If R-(XUY)=ϕ then we say the multivalued dependency is trivial.
- **Fourth Normal Form:** A relation R is in 4NF with respect to a set of dependencies F (includes all FDs and MVDs) if, for every nontrivial multivalued dependency X–>>Y in F^+, X is a super key for R.

- **Example:** The EMP relation is not in 4NF because in both the non trivial dependencies ENAME–>>PNAME and ENAME–>>DNAME, ENAME is not a super key of EMP. After decomposing it into EMP_PROJECTS and EMP_DEPENDENTS we do not have any non trivial MVDs because both the existing MVDs are trivial now. So EMP_PROJECTS and EMP_DEPENDENTS are in 4NF.

10. Join Dependencies and Fifth Normal Form

- **Join Dependency:** Let R is a relation, and let A, B, . . . ,Z be arbitary subsets of the set of attributes of R. Then we say that R satisfies the JD *(A, B, . . . ,Z) if and only if R is equal to the join of its projections on A, B, ….,Z.
- **Example:** The relation SUPPLY satisfies the JD *(R1, R2, R3). Applying NATURAL JOIN to any two of these relations produces spurious tuples but applying NATURAL JOIN to all three relations does not.
- Suppose there is a constraint: whenever a supplier s supplies part p, and a project j uses part p, and the supplier s supplies at least one part to project j then supplier s will be supplying part p to project j. This constraint is same as JD(R1, R2, R3). If this constraint holds the tuple below the dotted line must holds in the legal state of SUPPLY relation.

SUPPLY

SNAME	PARTNAME	PROJNAME
Smith	Bolt	ProjX
Smith	Nut	ProjY
Adamsky	Bolt	ProjY
Walton	Nut	ProjZ
Adamsky	Nail	ProjX
Adamsky	Bolt	ProjX
Smith	Bolt	ProjY

R1

SNAME	PARTNAME
Smith	Bolt
Smith	Nut
Adamsky	Bolt
Walton	Nut
Adamsky	Nail

R2

SNAME	PROJNAME
Smith	ProjX
Smith	ProjY
Adamsky	ProjY
Walton	ProjZ
Adamsky	ProjX

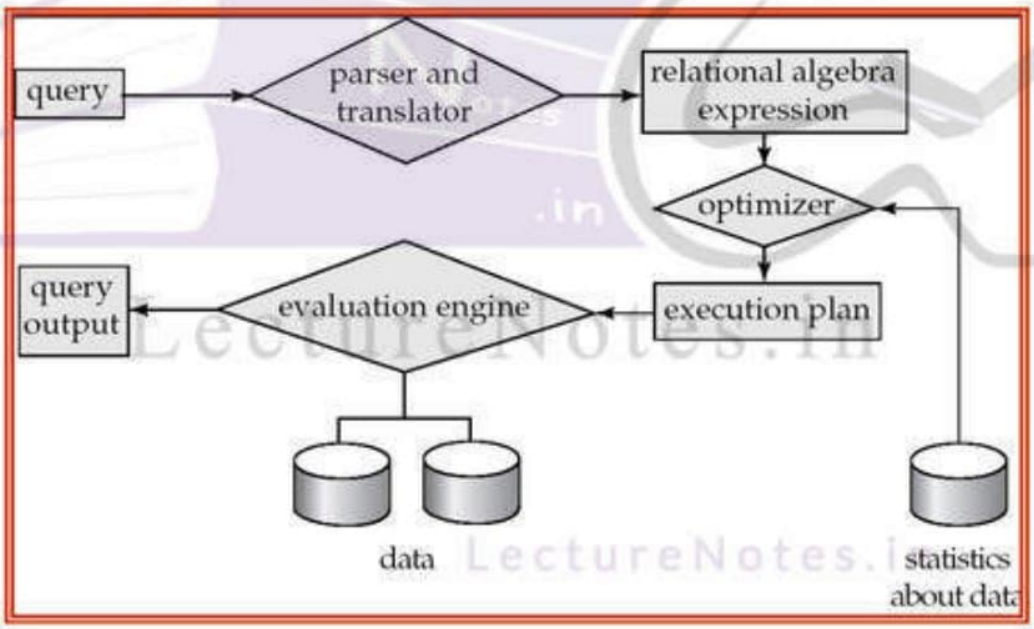
R3

PARTNAME	PROJNAME
Bolt	ProjX
Nut	ProjY
Bolt	ProjY
Nut	ProjZ
Nail	ProjX

- **Trivial Join Dependency:** A join dependency JD(R_1, R_2,…, R_n), specified on relation schema R is a trivial JD if one of the relation schema R_i in JD(R_1, R_2,…, R_n) is equal to R.
- **Fifth Normal Form / Project Join Normal Form (PJNF)-:** A relation R is in 5NF with respect to a set F of functional, multivalued, and join dependencies if for every non trivial join dependency JD(R_1, R_2,…, R_n) in F^+, every R_i is a super key of R.
- **Example:** The above relation SUPPLY is not in 5NF because it has a non trivial JD(R_1, R_2, R_3) but the attributes of any of the relations R_1, R_2, R_3 is not forming the super key of SUPPLY. Note that the candidate key of SUPPLY consists of all its attributes. But by decomposing SUPPLY into three relations R_1, R_2, R_3 all three satisfies 5NF because the do not involve any join dependencies at all.

Query Processing:

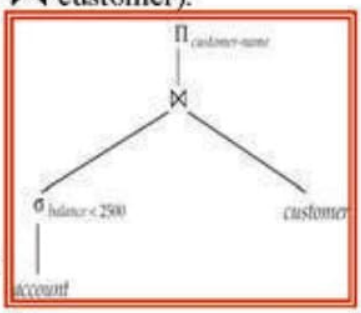
- ✓ Query Processing refers to a range of activities involved in extracting data from a database.
- ✓ Steps involved in processing a query are:
  - **Parsing and translation:** Translate the query into its internal form. This is then translated into relational algebra. The Parser checks syntax and verifies relations.
  - **Optimization:**
    - A relational algebra expression may have many equivalent expressions. For example the equivalent relational algebra expression for the SQL statement "select balance from account where balance<2500" can be $\sigma_{balance<2500}$ ($\Pi_{balance}$ (account)) or $\Pi_{balance}$ ($\sigma_{balance<2500}$ (account)).
    - Each relational algebra operation can be evaluated using one of several different algorithms.
    - Annotated expression specifying detailed evaluation strategy is called an query-evaluation plan or query-execution plan. For example it can use an index on *balance* to find accounts with balance < 2500 or can perform complete relation scan and discard accounts with balance ≥ 2500.
    - Amongst all equivalent evaluation plans choose the one with lowest cost.
    - Cost is estimated using statistical information (number of tuples in each relation, size of tuples etc.) from the database catalog.
  - **Evaluation:** The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.



Evaluation of Expressions:

- ✓ There are two alternatives to evaluate an expression containing multiple operations:
  - **Materialization:** The result of each evaluation is materialized in a temporary relation for subsequent use. But if the size of the temporary relation is large enough then we have to write it onto the disk.
    - **Materialized Evaluation:** Evaluate one operation at a time starting at the lowest level. Use intermediate results materialized into temporary relations to evaluate next level operations.

- **Example:** Consider the following relational algebra expression:
  $\Pi_{Customer\_name}(\sigma_{balance<2500}(account) \bowtie customer)$.
  The steps for execution of the query are:
  1. $R_1 \leftarrow \sigma_{balance<2500}(account)$
  2. $R_2 \leftarrow R_1 \bowtie customer$
  3. $Result \leftarrow \Pi_{Customer\_name}(R_2)$



- Materialized Evaluation is always applicable.
- Cost of writing results to disk and reading Them back can be quite high.
  Overall cost = Sum of individual operations + Cost of writing intermediate results to disk.
- **Double Buffering:** This technique uses two output buffers for each operation, when one is full write it to disk while the other is getting filled. Using this technique the execution of the algorithm will be faster.

- **Pipelinining:** The result of one operation is passed on to the next without storing in a temporary relation.
  - **Pipelined Evaluation:** Evaluate several operations simultaneously, passing the results of one operation to the next.
  - **Example:** In the last example without storing the result of $\sigma_{balance<2500}(account)$ in R_1 directly send the tuples for join operation. Similarly without storing the results of join operation in R_2 pass the tuples to the project operation.
  - Pipelining is much **cheaper** than materialization: no need to store a temporary relation on disk.
  - **Pipeline Implementation:** Pipelines can be executed in two ways: demand driven and producer driven.
    - **Demand driven** or **lazy evaluation:** System repeatedly requests next tuple from top level operation. Each operation requests next tuple from children operations as required, in order to output its next tuple.
    - **Producer-driven** or **eager pipelining:** Operators produce tuples eagerly and pass them up to their parents. Buffer is maintained between operators. Child puts tuples in buffer, parent removes tuples from buffer. If the buffer is full child waits till there is space in the buffer and then generates more tuples. System schedules operations that have space in output buffer and can process more input tuples.
  - Pipelining may not always possible. Example: sort, hash-join etc.

Transformation of Relational Expressions:

- ✓ Two relational algebra expressions are said to be **equivalent** if the two expressions generate the same set of tuples on every legal database instance. Order of tuples is irrelevant in this case.
- ✓ In SQL, input and outputs are multiset of tuples. Two expressions in the multiset version of the relational algebra are said to be equivalent if the two expressions generate the same multiset of tuples on every legal database instance.
- ✓ An equivalence rule says that expressions of two forms are equivalent. When two expressions are equivalent we can replace one expression by another.

Equivalence Rules:

1. Conjuctive selection operations can be deconstructed into a sequence of individual selections. This transformation is referred to as a cascade of σ.
   $\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$
2. Selection operations are commutative.
   $\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$
3. Only the final operations in a sequence of projection operations are needed, the others can be omitted.

$\Pi_{L1}(\Pi_{L2}(\ldots(\Pi_{Ln}(E))\ldots)) = \Pi_{L1}(E)$

4. Selections can be omitted with Cartesian product and theta joins.
   a. $\sigma_\theta(E_1 \times E_2) = E1 \bowtie_\theta E_2$
   b. $\sigma_{\theta 1}(E_1 \bowtie_{\theta 2} E_2) = E1 \bowtie_{\theta 1 \wedge \theta 2} E_2$
5. Theta-join operations are commutative
   $E_1 \bowtie_{\theta 2} E_2 = E_2 \bowtie_{\theta 2} E_1$
6. a. Natural join operations are associative.
   $(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$
   b. Theta join operations are associative in the following manner:
   $(E_1 \bowtie_{\theta 1} E_2) \bowtie_{\theta 2 \wedge \theta 3} E_3 = E_1 \bowtie_{\theta 1 \wedge \theta 3} (E_2 \bowtie_{\theta 2} E_3)$
   where θ_2 involves attributes from only E2 and E3
7. The selection operation distributes over the theta-join operation under the following two conditions:
   a. It distributes when all the attributes in selection condition θ_0 involves only the attributes of one of the expression (Let E_1) being joined.
   $\sigma_{\theta 0}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta 0}(E_1)) \bowtie_\theta E_2)$
   b. It distributes when all the attributes in selection condition θ_1 involves only the attributes of E_1 and θ_2 involves only the attributes of E_2
   $\sigma_{\theta 1 \wedge \theta 2}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta 1}(E_1)) \bowtie_\theta (\sigma_{\theta 1}(E_2))$
8. The projection operation distributes over the theta-join operation under the following two conditions:
   a. Let L_1 and L_2 be the attributes of E_1 and E_2 respectively. Suppose that the join condition θ involves only attributes in $L_1 \cup L_2$. Then
   $\Pi_{L1 \cup L2}(E_1 \bowtie_\theta E_2) = (\Pi_{L1}(E_1)) \bowtie_\theta (\Pi_{L2}(E_2))$
   b. Consider a join $E_1 \bowtie E_2$. Let L_1 and L_2 be the set of attributes from E_1 and E_2 respectively. Let L_3 be the attributes of E_1 that are involved in join condition θ, but are not in L1∪L2, and let L4 be the attributes of E2 that are involved in join condition θ, but are not in L1∪L2. Then
   $\Pi_{L1 \cup L2}(E_1 \bowtie_\theta E_2) = (\Pi_{L1 \cup L2}((\Pi_{L1 \cup L3}(E_1)) \bowtie_\theta (\Pi_{L2 \cup L4}(E_2)))$
9. The set operations union and intersection are commutative.
   $E_1 \cup E_2 = E_2 \cup E_1$
   $E_1 \cap E_2 = E_2 \cap E_1$
   Set difference is not commutative.
10. Set union and intersection are commutative.
   $(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$
   $(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$
11. The selection operation distributes over union, intersection and set difference operations.
   $\sigma_P(E_1 - E_2) = \sigma_P(E_1) - \sigma_P(E_2)$
   $\sigma_P(E_1 \cup E_2) = \sigma_P(E_1) \cup \sigma_P(E_2)$
   $\sigma_P(E_1 \cap E_2) = \sigma_P(E_1) \cap \sigma_P(E_2)$
   And
   $\sigma_P(E_1 - E_2) = \sigma_P(E_1) - E_2$
   $\sigma_P(E_1 \cap E_2) = \sigma_P(E_1) \cap E_2$
12. The projection operation distributes over the union operation.
   $\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$

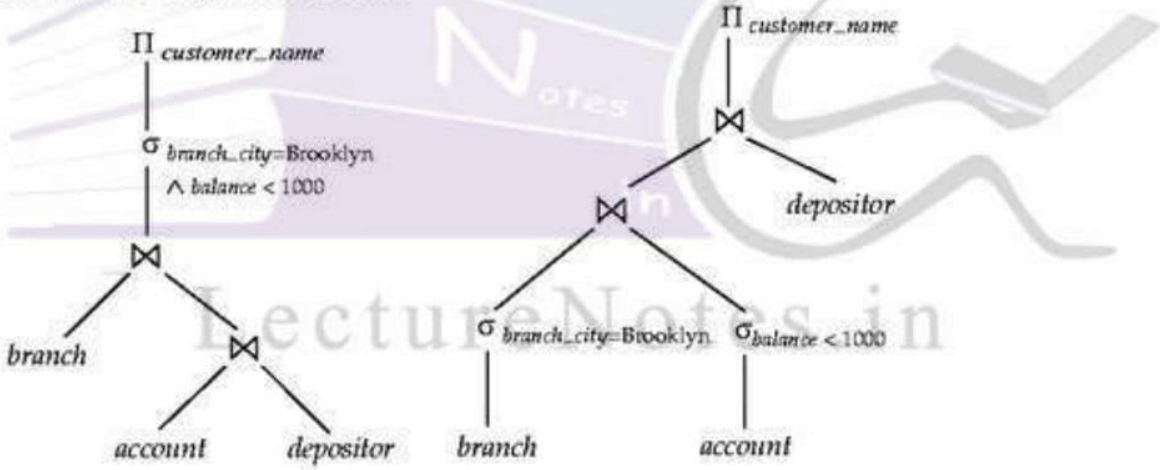
Examples of Transformations:

- ✓ Consider the bank example with following schema:
  Branch-schema = (branch-name, branch-city, assets)
  Account-schema = (account-number, branch-name, balance)

Depositior-schema = (customer-name, account-number)

Example 1

- ✓ Query: "Find the name of all customers who have an account at any branches located in Brooklyn."
- ✓ RAE1: $\Pi_{\text{customer-name}}(\sigma_{\text{branch-city}=\text{"Brooklyn"}}(\text{branch} \bowtie (\text{account} \bowtie \text{depositor})))$
- ✓ The RAE1 is equivalent to RAE2 but RAE2 generates smaller intermediate relations.
  RAE2: $\Pi_{\text{customer-name}}((\sigma_{\text{branch-city}=\text{"Brooklyn"}}(\text{branch})) \bowtie (\text{account} \bowtie \text{depositor}))$
  We can get the RAE2 from RAE1 by applying rule 7a
- ✓ Let us modify the last query to the following query:
  Query: "Find the name of all customers who have an account at any branches located in Brooklyn and have an account balance more than 1000."
  RAE3: $\Pi_{\text{customer-name}}(\sigma_{\text{branch-city}=\text{"Brooklyn"} \wedge \text{balance}>1000}(\text{branch} \bowtie (\text{account} \bowtie \text{depositor})))$
  Here in the RAE3 we cannot apply rule 7a since the selection condition contains the attributes from two relations. But we can apply first rule 6a and then 7a as follows:
  RAE4: $\Pi_{\text{customer-name}}(\sigma_{\text{branch-city}=\text{"Brooklyn"} \wedge \text{balance}>1000}((\text{branch} \bowtie \text{account}) \bowtie \text{depositor}))$
  RAE5: $\Pi_{\text{customer-name}}((\sigma_{\text{branch-city}=\text{"Brooklyn"} \wedge \text{balance}>1000}(\text{branch} \bowtie \text{account})) \bowtie \text{depositor})$
- ✓ Applying rule 1 to the selection sub expression we get:
  RAE6: $\Pi_{\text{customer-name}}((\sigma_{\text{branch-city}=\text{"Brooklyn"}}(\sigma_{\text{balance}>1000}(\text{branch} \bowtie \text{account}))) \bowtie \text{depositor})$
- ✓ Again performing selections early we get:
  RAE6: $\Pi_{\text{customer-name}}(((\sigma_{\text{branch-city}=\text{"Brooklyn"}}(\text{branch})) \bowtie (\sigma_{\text{balance}>1000}(\text{account}))) \bowtie \text{depositor})$
  By applying rule 7a and rule 1 to RAE4 we get RAE6. We can get RAE6 from RAE4 directly by applying rule 7b.
- ✓ A set of equivalence rules is said to be minimal if no rule can be derived from any combination of the others. So the equivalence rules from 1 to 12 are not minimal. If we use non-minimal sets of equivalence rules the number of ways of generating an expression increases. So query optimizer use minimal set of equivalence rules.



(a) Initial expression tree (b) Tree after multiple transformations

Example 2

- ✓ Let us consider the following query by example:
  $\Pi_{\text{customer-name}}((\sigma_{\text{branch-city}=\text{"Brooklyn"}}(\text{branch}) \bowtie \text{account}) \bowtie \text{depositor})$
- ✓ After computing $(\sigma_{\text{branch-city}=\text{"Brooklyn"}}(\text{branch}) \bowtie \text{account})$ we obtain a relation whose schema is:
  (branch-name, branch-city, assets, account-number, balance)
- ✓ But to join the result of $(\sigma_{\text{branch-city}=\text{"Brooklyn"}}(\text{branch} \bowtie \text{account})$ with depositor we need only account-number. So modifying the expression we get the resultant expression:
  $\Pi_{\text{customer-name}}(\Pi_{\text{account-number}}((\sigma_{\text{branch-city}=\text{"Brooklyn"}}(\text{branch}) \bowtie \text{account})) \bowtie \text{depositor}))$
  This expression reduces the size of the intermediate result.
- ✓ Performing the projection as early as possible reduces the size of the relation to be joined.

Example 3 (Join Ordering)

- ✓ We know from rule 6a the natural join operations are associative.
  $(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$

- ✓ If $E_1 \bowtie E_2$ is smaller than $E_2 \bowtie E_3$ then we should choose the first one so that we compute and store a smaller temporary relation.
- ✓ Consider the expression:

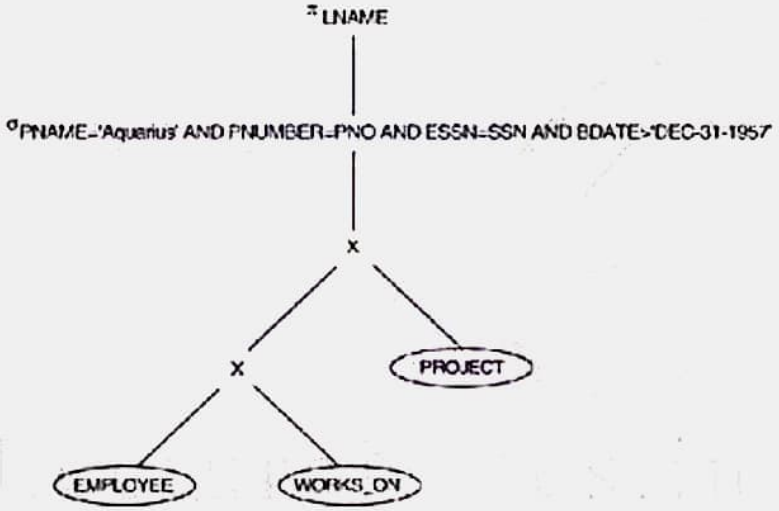
$$\Pi_{\text{customer-name}}((\sigma_{\text{branch-city}=\text{"Brooklyn"}}(\text{branch})) \bowtie \text{account} \bowtie \text{depositor})$$

- ✓ Here there are two alternatives either we join branch and account first and then join the result with depositor or we join account and depositor first and then join the result with branch.
- ✓ We prefer the first one since the first choice gives us the smaller intermediate relation because only a small fraction of bank customers are likely to have accounts in branches located in Brooklyn.

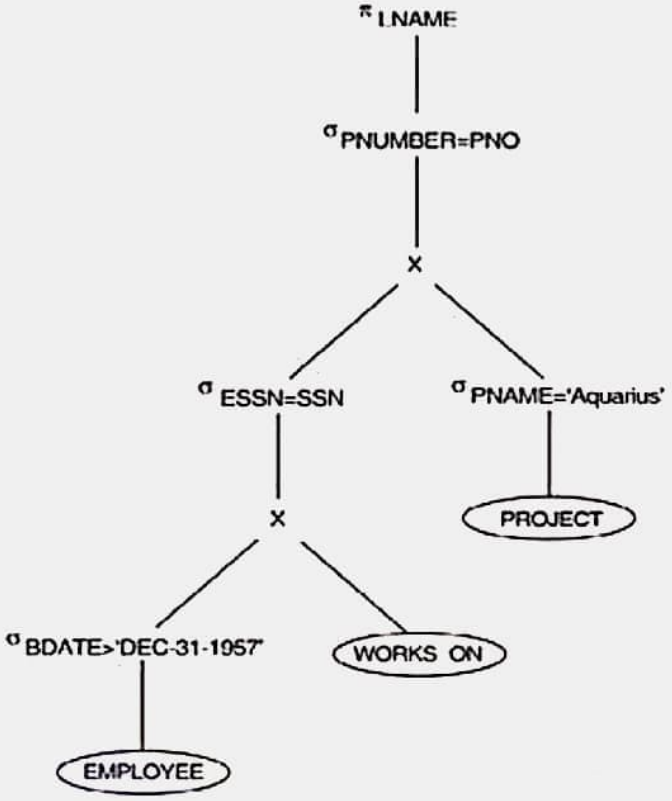
Example 4

- ✓ Consider the following schema:
  EMPLOYEE = (Fname, Lname, SSN, Bdate, Address, Sex, Salary, Superssn, DNo)
  WORKS_ON = (ESSN, PNO, Hours)
  PROJECT = (Pname, Pnumber, Plocation, DNo)
- ✓ **Question:** Find the last names of employees born after 1957 who work on a project named 'Aquarius'
- ✓ **SQL:** **SELECT** LNAME
  FROM EMPLOYEE, WORKS_ON, PROJECT
  WHERE PNAME='AQUARIUS' **AND** PNUMBER=PNO **AND** EENO=ENO **AND** DOB > '31-12-1957'
- ✓ Steps in converting a query tree during heuristic optimization:

Step 1 (Initial query tree for SQL query)

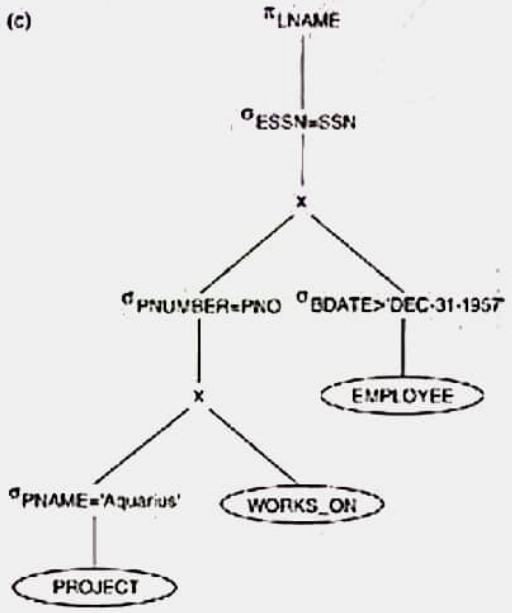


Step 2 (Moving SELECT operation down the tree)

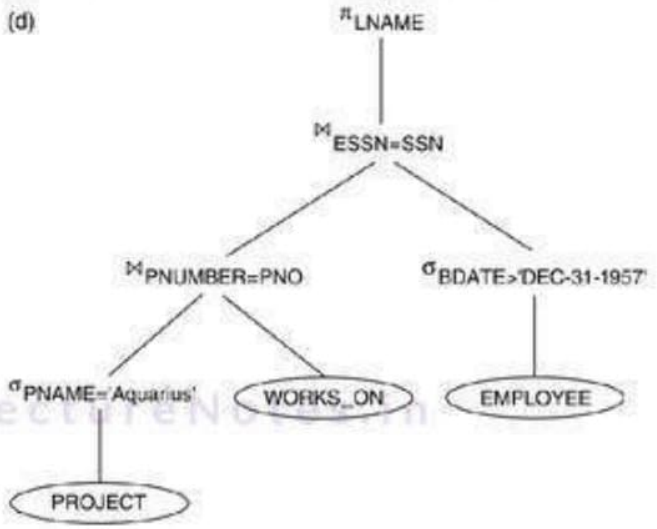


Step 3 (Applying more restrictive SELECT operation)

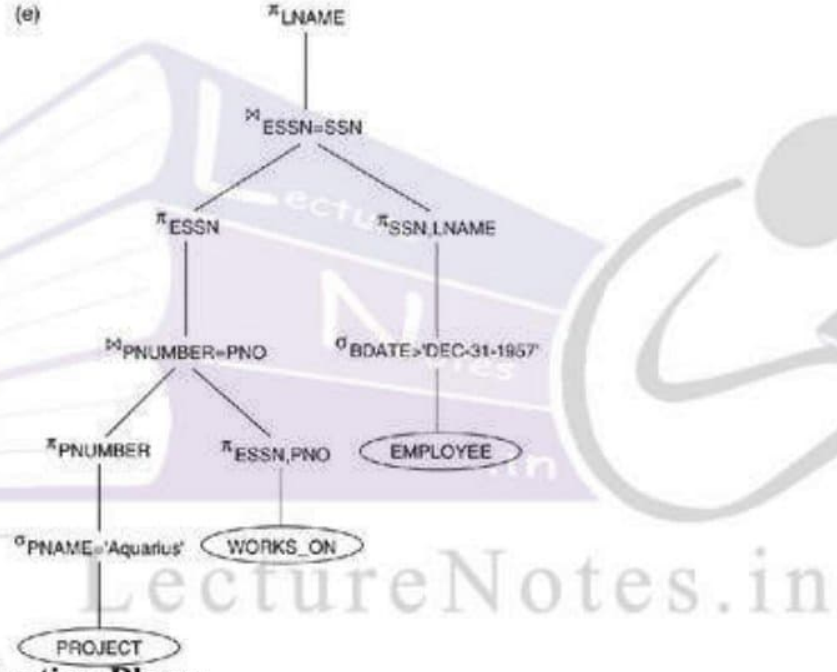
(c)



Step 4 (Replacing CARTESIAN PRODUCT and SELECT with JOIN)



Step 5 (Moving project operation down the tree)



Choice of evaluation Plans:

- ✓ Practical query optimizers incorporate elements of the following two broad approaches:
  - Search all the plans and choose the best plan in a cost-based fashion.
  - Use heuristics to choose a plan.

Cost based optimization:

- ✓ Suppose we want to find the best join order for $r_1 \bowtie r_2 \bowtie r_3 \bowtie \ldots \bowtie r_n$
- ✓ Here we have to find the cost for all possible join orders to find the best join order.
- ✓ There are $(2(n-1))!/(n-1)!$ different join orders for above expression. For n=7 the number is 665280. So it is very difficult to find the cost of all possible orders.
- ✓ Without generating the cost of all possible join ordrs using Dynamic Programming, the least-cost join order for any subset of $\{r1, r2, \ldots rn\}$ is computed only once and stored for future use.
- ✓ The time complexity of dynamic programming is $O(3^n)$ and space complexity is $O(2^n)$.
- ✓ Cost-based optimization is expensive, but worthwhile for queries on large datasets (typical queries have small n, generally < 10)

Heuristic optimization:

- ✓ Cost-based optimization is expensive, even with dynamic programming.
- ✓ Systems may use *heuristics* to reduce the number of choices that must be made in a cost-based fashion.
- ✓ Heuristic optimization transforms the query-tree by using a set of rules that typically (but not in all cases) improve execution performance:
  - Perform selection early (reduces the number of tuples)
  - Perform projection early (reduces the number of attributes)
  - Perform most restrictive selection and join operations (i.e. with smallest result size) before other similar operations.
- ✓ Some systems use only heuristics; others combine heuristics with partial cost-based optimization.

Transaction

- ✓ A transaction is a unit of program execution that accesses and possibly updates various data items.
- ✓ A transaction is initiated by a user program written in a high level data manipulation language or programming language like C, C++, Java etc.
- ✓ A transaction must see a consistent database.
- ✓ During transaction execution the database may be temporarily inconsistent.
- ✓ When the transaction completes successfully (is committed), the database must be consistent.
- ✓ After a transaction commits, the changes it has made to the database persist, even if there are system failures.
- ✓ Multiple transactions can execute in parallel.
- ✓ Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions
- ✓ To preserve the integrity of data the database, system must ensure the following set of properties known as **ACID properties**:
  - **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
  - **Consistency.** Execution of a transaction in isolation preserves the consistency of the database. Execution of transaction should not lead to inconsistent state.
  - **Isolation.** It ensures that although actions of several transactions might be interleaved the net effect is identical to executing all transactions serially.

    Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.

    That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.
  - **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.
- ✓ Transactions access data using two operations:
  - **read (X):** Transfers the data item X from the database to a local buffer.
  - **write (X):** Transfers the data item from the local buffer to the database.
- ✓ **Example:** T_i is a transaction that transfers $50 from Account A to account B.

```
T1:  read(A);
     A:=A-50
     write(A)
     read(B)
     B:=B+50
     Write(B)
```

Atomicity requirement: If the transaction fails after step 3 and before step 6, then the system should ensure that its updates are not reflected in the database. Else an inconsistency will result.

Consistency requirement: The sum of A and B is unchanged by the execution of the transaction.

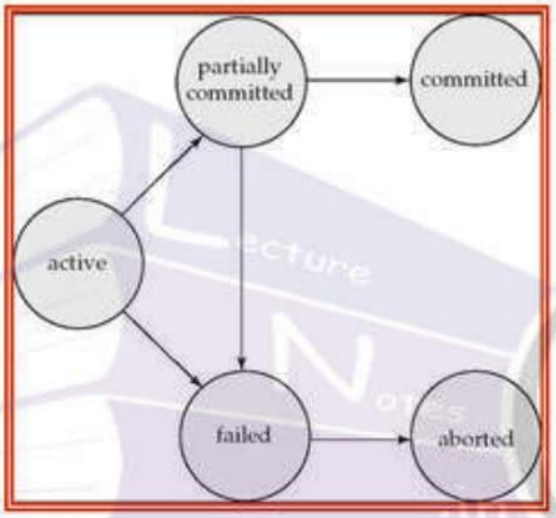
Isolation requirement: If between steps 3 and 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database (the sum A + B will be less than it should be).

- Isolation can be ensured trivially by running transactions **serially**, that is one after the other.
- However, executing multiple transactions concurrently has significant benefits.

Durability requirement: once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist despite failures.

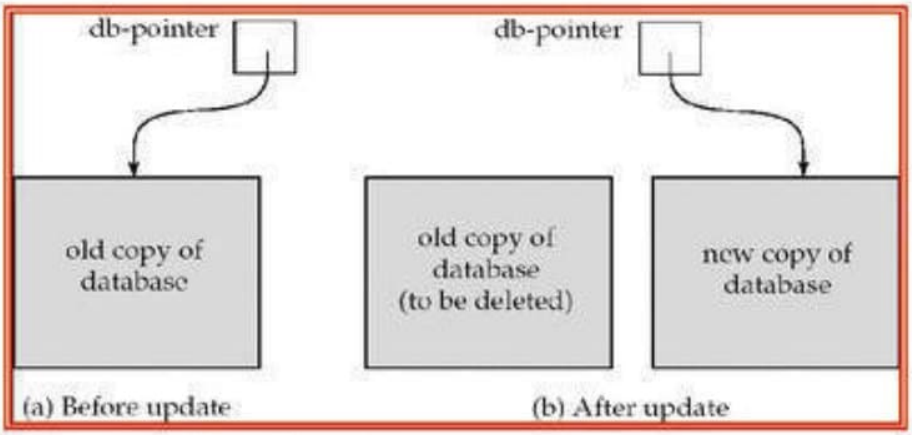
Transaction States

- ✓ If a transaction doesn't complete it's execution successfully then the transaction is **aborted**.
- ✓ When the changes caused by an aborted transaction have been undone, we say that the transaction has been **rolled back**.
- ✓ A transaction that completes its execution successfully is said to be **committed**.
- ✓ A transaction must be in one of the following states:
  - **Active:** The initial state; the transaction stays in this state while it is executing
  - **Partially committed:** After the final statement has been executed the transaction enters into this state.
  - **Failed:** After the discovery that normal execution can no longer proceed.
  - **Aborted:** After the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted: either restart the transaction (can be done only if no internal logical error) or kill the transaction.
  - **Committed:** After successful completion.
- ✓ A transaction is terminated if has either committed or aborted.
- ✓ State diagram of a transaction:



Implementation of Atomicity and Durability

- ✓ The **recovery-management** component of a database system implements the support for atomicity and durability.
- ✓ The **shadow-database** scheme:
  - Assume that only one transaction is active at a time.
  - A pointer called db_pointer always points to the current consistent copy of the database.
  - All updates are made on a **shadow copy** of the database, and **db_pointer** is made to point to the updated shadow copy only after the transaction reaches partial commit and all updated pages have been flushed to disk.
  - In case transaction fails, old consistent copy pointed to by **db_pointer** can be used, and the shadow copy can be deleted.



- ✓ The shadow-database scheme:
  - Assumes disks do not fail
  - Useful for text editors, but extremely inefficient for large databases because a single transaction requires copying the entire database.
  - Does not handle concurrent transactions

Concurrent Execution

- ✓ Multiple transactions are allowed to run concurrently in the system.
- ✓ The advantages of concurrent execution are:
  - **Improve throughput & resource utilization:** If the system executes multiple transactions at a time then while one transaction is doing some I/O operation another transaction can be processed by CPU. This leads to better transaction **throughput** (number of transactions executed in a given amount of time) one transaction can be using the CPU while another is reading from or writing to the disk
  - **Reduced waiting time:** Short transactions need not wait behind long ones. So it reduces average response time.
- ✓ **Concurrency control scheme:** It is a mechanism used by the database system to achieve isolation. That is, it is used to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database
- ✓ **Schedule:** A sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed.
  - A schedule for a set of transactions must consist of all instructions of those transactions.
  - Schedule must preserve the order in which the instructions appear in each individual transaction.
- ✓ **Example:** Let T_1 transfer $50 from A to B, and T_2 transfer 10% of the balance from A to B.

Schedule 1

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Schedule 2

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	

Schedule 3

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

Schedule 4

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

- ✓ **Serial Schedule:** Each serial schedule consists of a sequence of instructions from various transactions, where instructions belonging to one single transaction appear together in that schedule. **Example:** Schedule 1 & 2.
- ✓ **Concurrent Schedule:** In a concurrent schedule the instructions belonging to one transaction may not appear together in a schedule. Example: Schedule 3 & 4. Only Schedule 3 is equivalent to schedule 1.
- ✓ After executing a serial schedule the database remains in consistent state.
- ✓ During the execution of the concurrent schedule there is a chance of inconsistency in the database.
- ✓ The **concurrency-control component** ensures that any schedule that gets executed will leave the database in a consistent state.

Serializability

- ✓ Serial execution of a set of transactions preserves database consistency.
- ✓ A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
  - Conflict serializability
  - View serializability

Conflict Serializability

- ✓ Let us consider a Schedule S in which there are two consecutive instructions I_i and I_j of transactions T_i and T_j respectively.
- ✓ If I_i and I_j refer to different data items, then we can swap I_i and I_j without affecting the results of any instruction in the schedule.
- ✓ If I_i and I_j refer to the same data item Q then there are four cases to consider:
  - I_i = **read**(Q), I_j = **read**(Q). The order of I_i and I_j don't matter.
  - I_i = **read**(Q), I_j = **write**(Q). They order matters.
  - I_i = **write**(Q), I_j = **read**(Q). They order matters.
  - I_i = **write**(Q), I_j = **write**(Q). They order matters.
- ✓ So two instructions I_i and I_j conflict if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation..
- ✓ If I_i and I_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.
- ✓ If a schedule S can be transformed into a schedule S´ by a series of swaps of non-conflicting instructions, we say that S and S´ are **conflict equivalent**.
- ✓ We say that a schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule.
- ✓ **Example 1:** Schedule 3 is can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable.

Schedule 3

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Schedule 6

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Schedule 7

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

- ✓ **Example 2:** Schedule 7 is not conflict serializable since it is not equivalent to either the serial schedule <T_3, T_4> or the serial schedule <T_4, T_3>
- ✓ It is possible to have two schedules that produce the same outcome, but are not conflict equivalent.
- ✓ **Example:** Schedule 8 is not conflict equivalent to the serial schedule <T_1, T_5> or <T_5, T_1> however the final values of account A and B after the execution of either Schedule 8 or serial schedule <T_1, T_5> are the same.

View Serializability

- ✓ Let S and S´ be two schedules with the same set of transactions. S and S´ are **view equivalent** if the following three conditions are met:
  1. For each data item Q, if transaction T_i reads the initial value of Q in schedule S, then transaction T_i must, in schedule S´, also read the initial value of Q.
  2. For each data item Q, if transaction T_i executes **read**(Q) in schedule S, and if that value was produced by a write(Q) operation executed by transaction T_j, then the read(Q) operation of transaction T_i must, in schedule S´, also read the value of Q that was produced by the same write(Q) operation of transaction T_j .

Schedule 9

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

3. For each data item Q, the transaction (if any) that performs the final **write**(Q) operation in schedule S must perform the final **write**(Q) operation in schedule S´.

- ✓ **Example:** Schedule 1 is not view equivalent to schedule 2. But schedule 1 is view equivalent to schedule 3.
- ✓ **View serializable:** A schedule S is view serializable if it is view equivalent to a serial schedule.
- ✓ **Example:** Schedule 9 is view equivalent to the serial schedule <T_3, T_4, T_6>. So it is view serializable to <T_3, T_4, T_6>
- ✓ Every conflict serializable schedule is also view serializable, but there are view serializable schedules that are not conflict serializable. **Example:** Schedule 9.
- ✓ If a transaction performs write operation on data item Q without reading the same data item Q then that write is known as **blind write**.
- ✓ Every view serializable schedule that is not conflict serializable has **blind writes.**

Recoverability

- ✓ In a system that allows concurrent execution if a transaction T_i fails we have to undo the effect of this transition to ensure the atomicity property. It is also necessary to ensure that any transaction T_j that is dependent on T_i is also aborted.
- ✓ The following two types of schedules are acceptable from viewpoint of recovery from transaction failure.

Recoverable Schedules

- ✓ If a transaction T_j reads a data item previously written by a transaction T_i , then the commit operation of T_i appears before the commit operation of T_j.
- ✓ **Example:** The following Schedule 11 is not recoverable if T_9 commits immediately after the read(A), before T_8 commits.
- ✓ Database must ensure that schedules are recoverable.

Cascadeless Schedules

- ✓ **Cascading rollback: A** single transaction failure leads to a series of transaction rollbacks.

Schedule 11

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

Schedule 12

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

- ✓ **Example:** If T_{10} fails, T_{11} and T_{12} must also be rolled back.
- ✓ Cascading rollback lead to the undoing of a significant amount of work.
- ✓ **Cascadeless schedules:** The schedule in which cascading rollbacks cannot occur. For each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.
- ✓ Every cascadeless schedule is also recoverable.

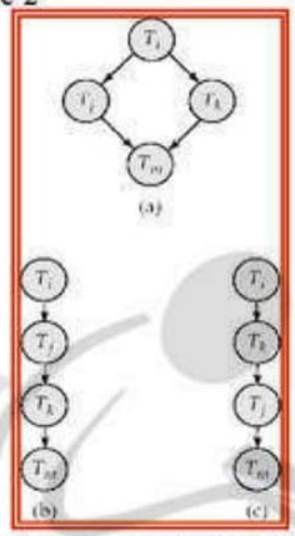
Testing for Serializability

The following method determines whether a schedule is conflict serializable or not:

- ✓ Consider some schedule of a set of transactions $T_1, T_2, ..., T_n$.
- ✓ **Precedence graph: A** direct graph constructed from a schedule S where the vertices are the transactions (names).
- ✓ The set of edges consists of all edges $T_i \rightarrow T_j$ for which one of three conditions holds:
  - T_i executes write(Q) before T_j executes read(Q).
  - T_i executes read(Q) before T_j executes write(Q).
  - T_i executes write(Q) before T_j executes write(Q).
- ✓ **Example:**

Precedence graph for Schedule 1 | Precedence graph for Schedule 2

T_1 T_2

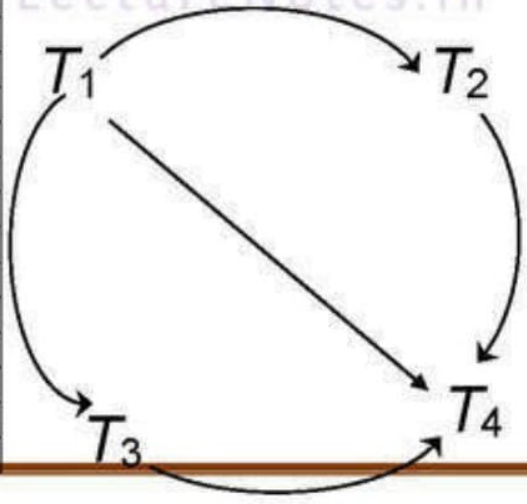


Precedence graph for Schedule 4

- ✓ A schedule is conflict serializable if and only if its precedence graph is acyclic.
- ✓ Cycle-detection algorithms exist which take $O(n^2)$time, where n is the number of vertices in the graph.
- ✓ If precedence graph is acyclic, the serializability order can be obtained by a topological sorting of the graph.
- ✓ **Example:** ⟶
- ✓ The precedence graph test for conflict serializability cannot be used directly to test for view serializability.
- ✓ The problem of checking if a schedule is view serializable falls in the class of NP-complete problems.
- ✓ Thus existence of an efficient algorithm is extremely unlikely.

Example: A schedule with its precedence graph.

Schedule A				
T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
Read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				



Concurrency Control

- ✓ Concurrency control mechanism ensures that the system control the interaction among the concurrent transactions so that the consistency of the database is preserved.

Lock-Based Protocols

Locks:

- ✓ To ensure serializability the system must access the data in a mutual exclusive manner.
- ✓ A lock is a mechanism to control concurrent access to a data item. If a transaction is holding a lock on a data item X then only it can access X.
- ✓ Data items can be locked in two modes:
  - **Exclusive (X) mode:** Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
  - **Shared (S) mode:** Data item can only be read. S-lock is requested using **lock-S** instruction.
- ✓ Lock requests are made to concurrency-control manager from the transaction depending on the type of operations the transaction want to perform on the data item. Transaction can proceed only after request is granted.
- ✓ **Compatibility:** Let A and B represent arbitrary lock modes. Suppose a transaction T_i requests a lock mode A on data item Q on which transaction T_j ($T_i \neq T_j$) currently holds a lock of mode B. If transaction T_i can be granted a lock on Q immediately in spite of presence of mode B lock, then we say that mode A is compatible with mode B.
- ✓ **Compatibility matrix:** It represents the compatibility between two modes. An element comp(A, B) of the matrix has the value true if and only if mode A is compatible with mode M.

	S	X
S	true	false
X	false	false

Lock-compatibility matrix comp

- ✓ A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions.
- ✓ Any number of transactions can hold shared locks on an item, but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.
- ✓ If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.
- ✓ **Example:** Consider the transactions T_1 (which transfers $50 from account B to account A) and T_2 (displays the total amount in account A and B).
- ✓ Considering A = $100 and B = $200 if the transactions T_1 and T_2 are executed serially (in any order) then T_2 will display $300.
- ✓ If the transactions T_1 and T_2 are executed concurrently as per the schedule 1 then T_2 displays $250, which is incorrect.

T_1: lock-X(B);	T_2: lock-S(A);
read(B);	read(A);
B := B - 50;	unlock(A);
write(B);	lock-S(B);
unlock(B);	read(B);
lock-X(A);	unlock(B);
read(A);	display(A+B)
A := A + 50;	
write(A);	
unlock(A)	

Schedule 1		
T_1	T_2	Concurrency Control manager
lock-X(B)		
		grant-X(B,T_1)
read(B)		
B: = B-50		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B,T_2)
	read(B)	
	unlock(B)	
	display(A+B)	
lock-X(A)		
		grant-X(A,T_1)
read(A);		
A := A + 50;		
write(A);		
unlock(A)		

Modified Schedule 1		
T_3	T_4	Concurrency Control manager
lock-X(B)		
		grant-X(B,T_3)
read(B)		
B: = B-50		
write(B)		
	lock-S(B)	
		grant-S(B, T_4)
	read(B)	
	lock-S(A)	
		grant-S(A,T_4)
	read(A)	
	display(A+B)	
	unlock(B)	
	unlock(A)	
lock-X(A)		
		grant-X(A,T_3)
read(A);		
A := A + 50;		
write(A);		
unlock(B)		
unlock(A)		

- ✓ The reason for inconsistency is T_1 unlocked the data item B too early.
- ✓ If we delay the unlocking to the end of the transaction in T_1 we get T_3 and if we delay the unlocking to the end of the transaction to the end of T_2 then we get T_4.
- ✓ Using T2 and T4 if we create a schedule (Modified Schedule 1) that will never make the database inconsistent.
- ✓ A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks.
- ✓ **Pitfalls of Lock-based Protocol:**

  Consider the partial schedule: ⟶

  Neither T_3 nor T_4 can make progress. Because by executing lock-S(B) causes T_4 to wait for T_3 to release its lock on B, and by executing lock-X(A) causes T_3 to wait for T_4 to release its lock on A. Such a situation is called a **deadlock**.
- ✓ To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.

T_3	T_4
lock-X(*B*)	
read(*B*)	
B := *B* − 50	
write(*B*)	
	lock-S(*A*)
	read(*A*)
	lock-S(*B*)
lock-X(*A*)	

Schedule 2

Granting of Locks:

- ✓ **Starvation** is also possible if concurrency control manager is badly designed.
  For example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
  - The same transaction is repeatedly rolled back due to deadlocks.
- ✓ Concurrency control manager can be designed to prevent starvation.
- ✓ We can avoid starvation of transactions by granting locks in the following manner:
  - When a transaction Ti requests a lock on data item Q in a particular mode M, the concurrency control manager grants the lock provided that:
    - There is no other transaction holding a lock on Q in a mode that conflict with M.
    - There is no other transaction that is waiting for a lock on Q, and that made its lock request before T_i.

The Two-Phase Locking Protocol:

- ✓ This is a protocol, which ensures conflict-serializable schedules.
- ✓ Each transaction issue lock and unlock requests in two phases:
  - **Growing Phase:** A transaction may obtain locks but may not release locks.
  - **Shrinking Phase:** A transaction may release locks, but may not obtain any new locks.
- ✓ When a transaction starts it is in growing phase. So the transaction acquires locks as needed. Once the transaction releases a lock it enters the shrinking phase and so can't request more lock requests.
  Example: Transaction T3 and T4 are two phase where as T1 and t2 are not.
- ✓ This protocol assures serializability. If we order the transactions according to their **lock points** (i.e. the point where a transaction obtained its final lock) then we get the serializabitlity ordering for the transactions.
- ✓ Two phase locking does not ensure freedom from deadlock. **Example:** Transaction T3 and T4 are two phase in Schedule 2 but they are deadlocked.
- ✓ **Variants of two Phase Locking:**
  - **Strict two-phase locking Protocol:** Cascading roll-back is possible under two-phase locking. To avoid this, we follow a modified protocol called **strict two-phase locking**. This protocol requires that not only that locking be two phase, but also that all exclusive locks taken by a transaction be held until that transaction commits/aborts.
    Example: In the partial schedule 3 the failure of T_5 after the read(A) step of T_7 leads to cascading rollback of T_6 and T_7. But by implementing strict two-phase locking we can avoid this situation.
  - **Rigorous two-phase locking:** is even stricter: here *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.
- ✓ **Lock Conversions:** Consider the following two transactions T_8 and T_9:

  - If we employ two phase locking protocol to execute the transactions T_8 and T_9 then we can't execute these transactions concurrently because as T_8 needs to write A so T_8 needs an exclusive lock and it will release it after the end of transaction T_8. So up to that T_9 can't get the shared lock on A. That is why T_8 and T_9 will execute serially.

Partial Schedule 3

T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

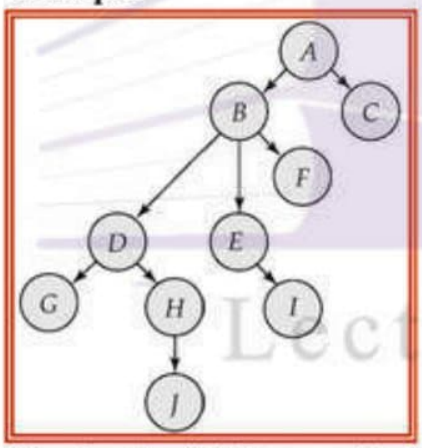
T_8	T_9
read(A)	read(A)
read(B)	read(B)
read(C)	display(A+B)
read(D)	
write(A)	

- This mechanism allows for **upgrading** a **shared lock** to **exclusive lock** and **downgrading** an **exclusive lock** to **shared lock**.
- But upgrading can take place only in growing phase and downgrading can take place only in shrinking phase.

Schedule with lock conversion	
T8	**T9**
lock-S(A)	
read(A)	
	lock-S(A)
	read(A)
lock-S(B)	
read(B)	
	lock-S(B)
	read(B)
lock-S(C)	
read(C)	
	display(A+B)
	unlock(A)
	unlock(B)
lock-S(D)	
read(D)	
upgrade(A)	
write(A)	
unlock(A)	
unlock(B)	
unlock(C)	
unlock(D)	

Graph-Based Protocols:

- ✓ Graph-based protocols are an alternative to two-phase locking.
- ✓ It uses additional information on how each transaction will access the database.
- ✓ Impose a partial ordering $\rightarrow$ on the set $\mathbf{D} = \{d_1, d_2, ..., d_h\}$ of all data items.
- ✓ If $d_i \rightarrow d_j$ then any transaction accessing both d_i and d_j must access d_i before accessing d_j.
- ✓ **D** may now be viewed as a directed acyclic graph, called a **database graph**.
- ✓ The **tree-protocol** is a simple kind of graph protocol in which:
  - Only exclusive locks are allowed.
  - The first lock by T_i may be on any data item.
  - Subsequently, a data Q can be locked by T_i only if the parent of Q is currently locked by T_i.
  - Data items may be unlocked at any time.
  - A data item that has been locked and unlocked by T_i cannot subsequently be relocked by T_i
- ✓ All schedules that are legal under the tree protocol are conflict serializable.
- ✓ **Example:**



Tree-structured database graph

Serializable Schedule under the tree protocol			
T10	**T11**	**T12**	**T13**
lock-X(B)			
	lock-X(D)		
	lock-X(H)		
	unlock(D)		
lock-X(E)			
lock-X(D)			
unlock(B)			
unlock(E)			
		lock-X(B)	
		lock-X(E)	
	unlock(H)		
lock-X(G)			
unlock(D)			
			lock-X(D)
			lock-X(H)
			unlock(D)
			unlock(H)
		unlock(E)	
		unlock(B)	
unlock(G)			

- ✓ The tree protocol ensures freedom from deadlock.
- ✓ The tree protocol does not ensure recoverability and cascadelessness.
- ✓ The following alternative improves concurrency but ensures only recoverability:
  - For each data item with an uncommitted write we record the transaction, which performed last write to the data item.
  - Whenever a transaction T_i performs a read of an uncommitted data item we record a commit dependency of T_i on the transaction that performed the last write to the data item.
  - Transaction T_i is then not permitted to commit until the commit of all transactions on which it has a commit dependency.
  - If any these transactions on which T_i has commit dependency aborts, T_i must also be aborted.

- ✓ **Advantage over two-phase locking:**
  - The tree locking protocol is deadlock-free, no rollbacks are required
  - Unlocking may occur earlier which leads to shorter waiting times, and also increases in concurrency.
- ✓ **Disadvantages:**
  - Transactions may have to lock data items that they do not access.
    Example: To access the data items A and J the transaction has to lock B, D, H unnecessarily.
  - When the transaction does not knows about what data it is going to lock next then the transaction has to lock the root of the tree which can reduce the concurrency.
  - Protocol does not guarantee cascade freedom

Timestamp-Based Protocols

Timestamps:

- ✓ Each transaction is issued a timestamp when it enters the system. If an old transaction T_i has time-stamp $TS(T_i)$, a new transaction T_j is assigned time-stamp $TS(T_j)$ such that $TS(T_i) < TS(T_j)$.
- ✓ **Implementation of Timestamp:**
  - When a transaction enters the system it's timestamp is equal to the value of the system clock.
  - When a transaction enters the system it's timestamp is equal to the value of the logical counter, which is incremented after a new timestamp has been assigned.
- ✓ If $TS(T_i) < TS(T_j)$, then the system must ensure that the produced schedule is equivalent to the serial schedule $<T_i, T_j>$
- ✓ The protocol manages concurrent execution such that the time-stamps determine the serializability order.
- ✓ To implement this scheme, the protocol maintains for each data Q two timestamp values:
  - **W-timestamp**(Q) is the largest time-stamp of any transaction that executed **write**(Q) successfully.
  - **R-timestamp**(Q) is the largest time-stamp of any transaction that executed **read**(Q) successfully.

The Timestamp-Ordering Protocol:

- ✓ The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.
- ✓ Suppose a transaction T_i issues a **read**(Q)
  - If $TS(T_i) <$ **W**-timestamp(Q), then T_i needs to read a value of Q that was already overwritten. Hence, the **read** operation is rejected, and T_i is rolled back.
  - If $TS(T_i) \geq$ **W**-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to **max**(R-timestamp(Q), $TS(T_i)$).
- ✓ Suppose that transaction T_i issues **write**(Q).
  - If $TS(T_i) <$ R-timestamp(Q), then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced. Hence, the **write** operation is rejected, and T_i is rolled back.
  - If $TS(T_i) <$ W-timestamp(Q), then T_i is attempting to write an obsolete value of Q. Hence, this **write** operation is rejected, and T_i is rolled back.
  - Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to $TS(T_i)$.
- ✓ If a transaction T_i is rolled back by the concurrency control scheme the system assigns a new timestamp and restarts it.
- ✓ Timestamp protocol ensures freedom from deadlock as no transaction ever waits.
- ✓ But the schedule may not be cascade-free, and may not even be recoverable.
- ✓ We can make the schedule recoverable, in one of the following ways:
  - **Performing all writes together at the end of the transaction** can ensure recoverability and cascadelessness. While the writes are in progress, no transaction is permitted to access any data items that have been written

- Recoverability and cascadelessness can also be guaranteed by **using limited locking** whereby reads of uncommitted items are postponed until the transaction that updated the item commits.

Thomas' Write Rule:

- ✓ This is a modified version of the timestamp-ordering protocol in which **obsolete write** operations may be ignored under certain circumstances.
- ✓ When T_i attempts to write data item Q, if $TS(T_i) < W\text{-timestamp}(Q)$, then T_i is attempting to write an obsolete value of {Q}.
- ✓ Rather than rolling back T_i as the timestamp ordering protocol would have done, this {**write**} operation can be ignored.
- ✓ Otherwise this protocol is the same as the timestamp ordering protocol.
- ✓ Thomas' Write Rule makes use of view serializability by deleting obsolete write operations from the transactions that issue them.
- ✓ **Example:** This schedule is not conflict serializable under any of the locking protocols, tree based protocols or time-stamp ordering protocol. But if we apply Thomas' Write Rule we have to ignore write(Q) of T_3. The result is a schedule that is view equivalent to the serial schedule $<T_3, T_4>$.

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

Validation-Based Protocols (Optimistic Concurrency Control Scheme):

- ✓ This scheme is used where a majority of transactions are read-only transactions and the rate of conflict among the transactions is very low.
- ✓ This scheme imposes fewer overheads on the system. But for this the system has to monitor the system.
- ✓ Execution of each transaction T_i is done in two or three of the following phases:
  1. **Read phase**: Transaction T_i reads the values of different data items and writes only to temporary local variables.
  2. **Validation phase**: Transaction T_i performs a "validation test" to determine if local variables can be written to the disk without violating serializability.
  3. **Write phase**: If T_i is validated, the updates are applied to the database; otherwise, T_i is rolled back.
- ✓ The three phases of concurrently executing transactions can be interleaved, but each transaction must go through the three phases in that order.
- ✓ Assume for simplicity that the validation and write phase occur together, atomically and serially i.e. only one transaction executes validation/write at a time.
- ✓ Also called as **optimistic concurrency control** since transaction executes fully in the hope that all will go well during validation.
- ✓ Each transaction T_i has 3 timestamps:
  1. **Start(T_i)** : The time when T_i started its execution.
  2. **Validation(T_i)**: The time when T_i entered its validation phase.
  3. **Finish(T_i)** : The time when T_i finished its write phase.
- ✓ Serializability order is determined by timestamp given at validation time, to increase concurrency.
- ✓ Thus $TS(T_i)$ is given the value of Validation(T_i).
- ✓ This protocol is useful and gives greater degree of concurrency if probability of conflicts is low.

- ✓ **Validation Test:** The validation test for transaction T_j requires that, for all T_i with TS (T_i) < TS (T_j) either one of the following condition holds:
  1. **Finish**(T_i) < **Start**(T_j)
  2. **Start**(T_j) < **Finish**(T_i) < **Validation**(T_j) and the set of data items written by T_i does not intersect with the set of data items read by T_j.
- ✓ If the validation succeeds then T_j can be committed. Otherwise T_j is aborted.
- ✓ The validation scheme guards against cascading rollback.
- ✓ There is a possibility of starvation

Schedule 5	
T_{14}	T_{15}
read(B)	
	read(B)
	B:=B-50
	read(A)
	A:=A+50
read(A)	
<validate>	
display<A+B>	
	<validate>
	write(B)
	write(A)

Multiversion Schemes:

- ✓ Multiversion schemes keep old versions of data item to increase concurrency.
- ✓ Each successful **write** results in the creation of a new version of the data item written.
- ✓ Use timestamps to label versions.
- ✓ When a **read**(Q) operation is issued, select an appropriate version of Q based on the timestamp of the transaction, and return the value of the selected version.
- ✓ **read**s never have to wait as an appropriate version is returned immediately.

Multiversion Timestamp Ordering:

- ✓ Each data item Q has a sequence of versions <Q_1, Q_2,....., Q_m>.
- ✓ Each version Q_k contains three data fields:
  - **Content** is the value of version Q_K.
  - **W-timestamp**(Q_k) is the timestamp of the transaction that created (wrote) version Q_k
  - **R-timestamp**(Q_K) is the largest timestamp of any transaction that successfully read version Q_K
- ✓ When a transaction T_i creates a new version Q_k of Q, Q_K's W-timestamp and R-timestamp are initialized to TS(T_i).
- ✓ R-timestamp of Q_k is updated whenever a transaction T_j reads Q_k, and TS(T_j) > R-timestamp(Q_k).
- ✓ Suppose that transaction T_i issues a **read**(Q) or **write**(Q) operation. Let Q_k denote the version of Q whose write timestamp is the largest write timestamp less than or equal to TS(T_i).
  - If transaction T_i issues a **read**(Q), then the value returned is the content of version Q_k.
  - If transaction T_i issues a **write**(Q):
    1. if TS(T_i) < R-timestamp(Q_k), then transaction T_i is rolled back.
    2. else if TS(T_i) = W-timestamp(Q_k), the contents of Q_k are overwritten.
    3. else a new version of Q is created.
- ✓ Reads always succeed and is never made to wait.
- ✓ Reading of a data item also requires the updating of the R-timestamp field.

Multiversion Two phase locking:

- ✓ Differentiates between read-only transactions and update transactions.
- ✓ **Update transactions** acquire read and write locks, and hold all locks up to the end of the transaction. That is, update transactions follow rigorous two-phase locking.
  - Each successful **write** results in the creation of a new version of the data item written.
  - Each version of a data item has a single timestamp whose value is obtained from a counter **ts-counter** that is incremented during commit processing.
- ✓ Read-only transactions are assigned a timestamp by reading the current value of **ts-counter** before they start execution; they follow the multiversion timestamp-ordering protocol for performing reads.
- ✓ When an update transaction wants to read a data item:
  - it obtains a shared lock on it, and reads the latest version.

- ✓ When it wants to write an item
  - it obtains X lock on; it then creates a new version of the item and sets this version's timestamp to ∞.
- ✓ When update transaction T_i completes, commit processing occurs:
  - T_i sets timestamp on the versions it has created to **ts-counter** + 1
  - T_i increments **ts-counter** by 1
- ✓ Read-only transactions that start after T_i increments **ts-counter** will see the values updated by T_i.
- ✓ Read-only transactions that start before T_i increments the **ts-counter** will see the value before the updates by T_i.
- ✓ Only serializable schedules are produced.

Deadlock Handling

Schedule	
T_1	T_2
lock-X(X)	
write(X)	
	lock-X(Y)
	write(Y)
lock-X(Y)	
write(Y)	
	lock-X(X)
	write(X)

- ✓ System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.

  If there exists a set of transactions $\{T_0, T_1, \ldots, T_n\}$such that T_0 is waiting for T_1, T_1 is waiting for T_2,…,T_{n-1} is waiting for T_n and T_n is waiting for T_0 then all the transactions in the set are in deadlock.
- ✓ **Example:** Consider the following two transactions:

  T_1: write (X) T_2: write(Y)
  write(Y) write(X)

  In this schedule T_1 is waiting for T_2, which has locked data item Y, and T_2 is waiting for T_1, which has locked data item X. So the system is in dead lock
- ✓ Methods for handling dead lock:
  - **Deadlock prevention:** This protocol ensures that the system will never enter into a deadlock state.
  - **Deadlock detection** and **deadlock recovery:** If we can't prevent the deadlock then we are allowing the system to enter in the deadlock. So we have to detect whether deadlock occurred or not if occurred then we should recover from the deadlock.

Deadlock Prevention:

There are two approaches to deadlock prevention:

- ✓ Approach 1
  - Each transaction locks all its data items before it begins its execution. Either all are locked in one step or none are locked.
  - Disadvantages:
    - It is difficult to predict what data items to be locked before the start of transaction.
    - Data item utilization is very low.
- ✓ Approach 2
  - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).
  - Once a transaction has locked a particular item, it cannot request locks on items that precede that item in the oredering.
- ✓ Approach 3
  - This scheme uses transaction timestamps for the sake of deadlock prevention alone.
  - Two different schemes are used using timestamp:
    - The **Wait-die Scheme:** It is non-preemptive technique. Older transaction may wait for younger one to release data item. Younger transactions never wait for older ones they are rolled back (dies) instead. Note that TS(Older)< TS(Younger). A transaction may die several times before acquiring needed data item.

      Example: Suppose T_{22}, T_{23}, T_{24} have the timestamps 5, 10,15 respectively. If T_{22} requests a data item held by T_{23}, then T_{22} will wait. If T_{24} requests a data item held by T_{23}, then T_{24} will be rolled back.
    - The **Wound-wait scheme:** It is a preemptive technique. Older transaction wounds (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.

      Example: If T_{22} requests a data item held by T_{23}, then the data item will be preempted from T_{23}, and T_{23} will be rolled back. If T_{24} requests a data item held by T_{23}, then T_{23} will wait.
  - Both in wait-die and in wound-wait schemes, a rolled back transaction is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided.

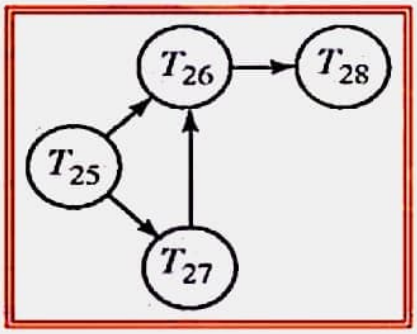
- Difference between wait-die and wound-die:
  - In the wait-die scheme the older transaction must wait for younger transaction to release its data item. But in wound-wait scheme an older transaction never waits for a younger transaction.
  - In the wait-die scheme the transaction may die and rolled back several times before getting a data item. But in wound-die scheme a transaction wounded and rolled back once and next it waits. That is why it may have fewer rollbacks than wait-die scheme.

Timeout-Based Schemes:

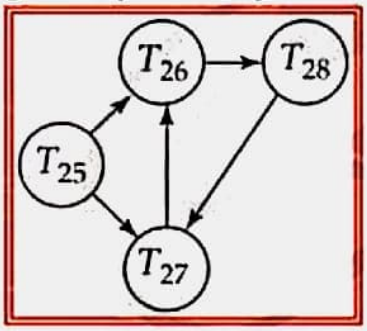
- ✓ A transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
- ✓ It is simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.

Deadlock Detection:

- ✓ Deadlocks can be described as a **wait-for graph**, which consists of a pair G = (V,E), where V is a set of vertices (all the transactions in the system) and E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.
- ✓ If $T_i \rightarrow T_j$ is in E, then there is a directed edge from T_i to T_j, implying that T_i is waiting for T_j to release a data item.
- ✓ When T_i requests a data item currently being held by T_j, then the edge $T_i \rightarrow T_j$ is inserted in the wait-for graph. This edge is removed only when T_j is no longer holding a data item needed by T_i.
- ✓ The system is in a deadlock state if and only if the wait-for graph has a cycle.
- ✓ The system must invoke a deadlock-detection algorithm periodically to look for cycles.
- ✓ **Example:**



Wait-for graph with no cycle



Wait-for graph with a cycle

Deadlock Recovery:

When deadlock is detected the system has to recover from the deadlock. To recover from deadlock the system has to rolled some of the transactions, which are in deadlock. For that the system has to take the following actions:

- **Selection of a victim:** The system has to select one or more transactions (make a victim) to roll back to break down the deadlock. The system selects that transaction as victim that will incur minimum cost depending on
  - How long the transaction has computed and how much left.
  - How many data items the transaction has used
  - How many more data items the transaction needs for it to complete.
  - How many transactions will be involved in the roll back.
- **Rollback:** Determine how far to roll back transaction.
  - **Total rollback:** Abort the transaction and then restart it.
  - **Partial rollback:** Roll back transaction only as far as necessary to break deadlock.
- **Starvation:** Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation.

Database Recovery System

Failure classification

There are various types of failures that may occur in a system. Some of them are:

- ✓ **Transaction Failure:**
  - **Logical Error:** Transaction fails due to some internal error condition like bad input, overflow resource limit exceeds etc.
  - **System Error:** The database system terminates an active transaction due to an error condition like deadlock.
- ✓ **System crash**
- ✓ **Disk failure**

Recovery and Atomicity:

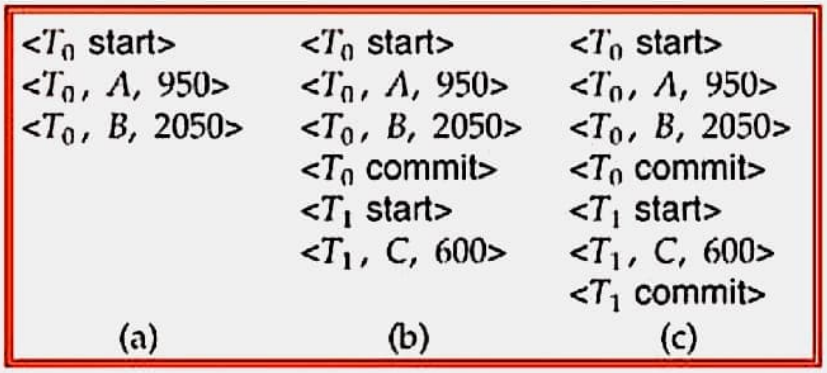
- ✓ Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state.
- ✓ To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
- ✓ There are two recovery approaches:
  - **Log-based recovery**
  - **Shadow-paging**

Log-Based Recovery

- ✓ A **log** is kept on stable storage.
- ✓ The log is a sequence of **log records**, and maintains a record of update activities on the database.
- ✓ When transaction T_i starts, it registers itself by writing a <T_i **start**> log record
- ✓ Before T_i executes **write**(X), a log record <T_i, X, V_1, V_2> is written, where V_1 is the value of X before the write, and V_2 is the value to be written to X.
- ✓ When T_i finishes it last statement, the log record <T_i **commit**> is written.
- ✓ There are two approaches using logs:
  - **Deferred database modification**
    - The **deferred database modification** scheme records all modifications to the log, but defers all the **writes** to after partial commit.
    - Transaction starts by writing <T_i **start**> record to log.
    - A write(X) operation results in a log record <T_i, X, V> being written, where V is the new value for X. Note that old value is not needed for this scheme
    - The write is not performed on X at this time, but is deferred.
    - When T_i partially commits, <T_i **commit**> is written to the log.
    - Finally, the log records are read and used to actually execute the previously deferred writes.
    - DDuring recovery after a crash, a transaction needs to be **redone** if and only if both <T_i **start**> and <T_i **commit**> are there in the log.
    - Redoing a transaction T_i (**redo** T_i) sets the value of all data items updated by the transaction to the new values.
    - **Example:** Transactions T_0 and T_1 (T_0 executes before T_1):

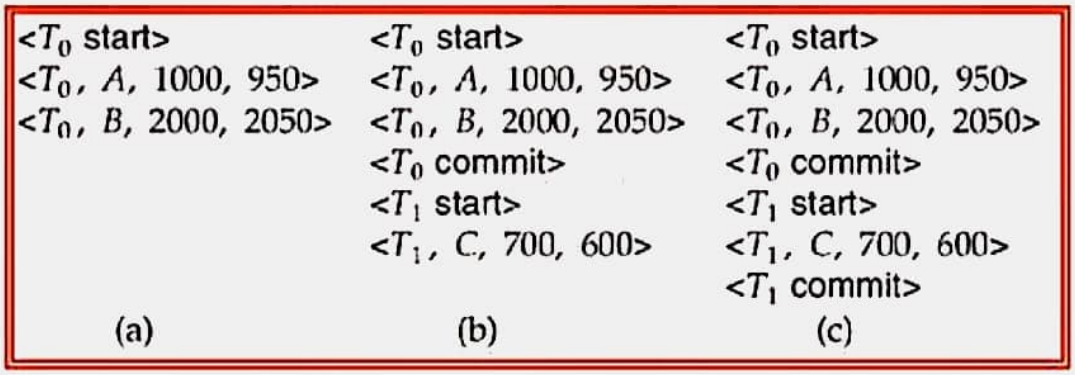
T_0:		T_1 :	
	read (A)		**read** (C)
	A: - A - 50		C := C- 100
	write (A)		**write** (C)
	read (B)		
	B := B + 50		
	write (B)		

    - The following figure shows the log as it appears at three instances of time:



- If log on stable storage at time of crash is as in case:
  (a) No redo actions need to be taken
  (b) redo(T_0) must be performed since <T_0 **commit**> is present
  (c) **redo**(T_0) must be performed followed by redo(T_1) since <T_0 **commit**> and <T_1 commit> are present

- **Immediate database modification:**
  - The **immediate database modification** scheme allows database updates of an uncommitted transaction to be made as the writes are issued
  - Since undoing may be needed, update logs must have both old value and new value.
  - Update log record must be written before database item is written.
  - We assume that the log record is output directly to stable storage
  - Recovery procedure has two operations instead of one:
    - **undo**(T_i) restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i
    - **redo**(T_i) sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i
  - When recovering after failure:
    - Transaction T_i needs to be undone if the log contains the record <T_i **start**>, but does not contain the record <T_i **commit**>.
    - Transaction T_i needs to be redone if the log contains both the record <T_i **start**> and the record <T_i **commit**>.
  - The following figure shows the log as it appears at three instances of time:



  - Recovery actions in each case above are:
    - Undo T_0: B is restored to 2000 ad A to 1000.
    - Undo T_1 and redo T_0: C is restored to 700, and then A and B are set to 950 and 2050 respectively.
    - Redo T_0 and redo T_1: A and B are set to 950 and 2050

Shadow-Paging

- ✓ It requires fewer disk accesses under certain circumstances.
- ✓ The database is divided into no of pages and these pages are stored in the disk randomly (not in a particular order).
- ✓ To find the location of the i^{th} page in the disk we use a page table. The page table has n entries one for each page in the disk. Each record in the page table stores the page number and a pointer to the disk where it is stored permanently.
- ✓ Shadow paging uses two page tables during the life of transaction: the **current page table** and **shadow page table.**
- ✓ During the execution of the transaction the shadow page is never changed.
- ✓ The current page table is changed when a transaction performs a write operation.
- ✓ Whe a transaction performs a write operation:
    1. If the i^{th} page (in which page X resides) is not in main memory then issue input(X).
    2. If this is the write first performed on the ith page by this transaction then modify the current page table as follows:
        a. Find an unused page on disk.
        b. Delete the page found in step 2a from the list of free page frames.
        c. Modify the current page table such that the i^{th} entry points to the page found in step 2a.
    3. Assign the value of x_j to X in the buffer page.
- ✓ When the transaction commits the current page table is written to a nonvolatile storage. After that the current page table becomes the new shadow page table and the next transaction begins.
- ✓ The shadow page table is stored in the non-volatile storage. The current page table may be stored in memory.
- ✓ If a transaction fails before committing, we copy the shadow page table into memory and use it for next transaction.
- ✓ **Advantages:** Overhead of log-record output is eliminated.
- ✓ **Drawbacks:**
    - Commit overhead
    - Data Fragmentation
    - Garbage collection