Chapter 1
INTRODUCTION

1. **Definition of DBMS**: - What is database?

"Database Management System" consists of two phrases:

- Database
- Management System

"Database" is a collection of inter-related files.

The basic "*management* "operations are:

- Searching
- Addition
- Deletion
- Updation

A DBMS consist of collection of:

- related data &
- a set of programs to access & manipulate those data.

The primary goal of DBMS is to provide an environment that is:

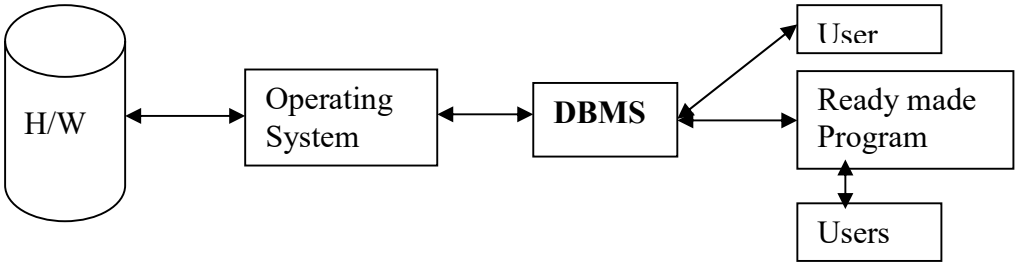
- convenient & simplicity - front end – GUI (Java, JSP, VB, .NET)
- efficient - back end (Access, Oracle, DB-2)

to use in retrieving & storing the database information.

The simplicity or convenience is provided for the user of database. Efficiency is provided from h/w point of view. The user interface should be graphical & simple to operate. The DBMS should use minimum set of computing resources to provide maximum throughput.

Following are the basic facilities provided by a DBMS :

1. Creation, addition, deletion, modification & retrieval.
2. protection of data & maintaining integrity.
3. Report generation.
4. Mathematical operation.



Note 1: - A DBMS may be characterized as simple internal processing but very large volume of input and output data. So, a DBMS application is basically **IO bound job**.

Note 2: Another Definition of DBMS:

row data/Fields → Record/row/info → table/File → Database

Fields are in the form of row data; fields organized in meaningful way forms records; collection of similar records forms a file; collection of inter-related files is a data-base.

2 Advantage of DBMS/Comparison of Traditional file system (T.F.S) & DBMS / Objectives of DBMS :-

Following are the advantages of DBMS approach:

1. Redundancy & Inconsistency can be controlled :

Since the files & application program are created by different programmers over a long period, so various files are likely to have different formats & information may be duplicate at many places. Duplicate information is data redundancy. It causes wastage of space. If modification is needed & it is not done at all the places, then it will cause inconsistency.

In the DBMS approach we design the data-base as a whole & apply normalization technique to reduce data redundancy. Relationship can be set to control inconsistency.

2. Simplicity in Data Access :

DBMS provides the environment & standard methods to access the data in various ways with simplicity.

The traditional file system does not provide such facility. Though we can create various facilities, but it needs considerable programming efforts.

3. Correlation of Data:

In T.F.S. data are scattered in various files which are not logically related. The files may be in different formats. It causes difficulty in accessing & organizing the data.

In the DBMS approach various data files are interlinked, standard are followed. It results in better organization of data.

4. Integrity Maintenance:

There are many types of integrity constraints like minimum & maximum permitted values of a data. To apply such constraints needs considerable programming efforts in TFS.

In DBMS these integrity constraints can be applied easily, because there are special provisions for it.

5. Atomicity:

Some operation requires either the complete set of operations or no operation. It is called as transaction. The atomicity property of transaction can easily be maintained in DBMS but not in T.F.S.

6. Concurrent Access:

In multi-user data-base, there can be situation in which multiple requests can arise simultaneously to access the database. Anomalies may arise here or a deadlock may occur.

T.F.S. dose not proved such facilities.

DBMS provide facilities for handling deadlocks & locking the database.

7. Data Security:

Every user should not be able to access the complete database. The data should be protected with the help of password. There should be protection and recovery of data form system failures.

The TFS does not provide such facilities.

The DBMS provides all such the facility by defining views so that a user can see only that data that is meant for him only. The data recovery facilities are also provided in case of system crash.

8. Enforcing Standards :

The development of TFS is not integrated, so enforcement of standards is not an easy task.

DBMS is being developed as a whole & there is central controller of the database so enforcement of standard is easy in DBMS approach.

9. Development time :

Since DBMS Provides simple environment & many standard facilities so the total development time is less in DBMS as compare to T.F.S.

10. Reduced application development time: incremental time to add each new application is reduced.

11. Flexibility to change data structures: database structure may evolve as new requirements are defined.

12. Availability of up-to-date information – very important for on-line transaction systems such as airline, hotel, car reservations.

13. Economies of scale: by consolidating data and applications across department's wasteful overlap of resources and personnel can be avoided.

Disadvantage of DBMS :-

1. **Lack of flexibility** :

DBMS normally provides the high level of facilities suitable only for commercial application. Operation like engineering analysis, system S/W development cannot be done in a

DBMS.

2. It requires heavy computing resources.
3. Problem associated with centralization in DBMS suffer with centralization problem.
4. High initial investment for H/W, S/W & training.
5. Increased complexity at system level.

3 Various Views of Data, Data Abstraction & Data Independence:

(A) Views of Data & Abstraction: -

Abstraction is hiding of implementation details to provide simplicity to the user. These are 3 levels of data abstraction in DBMS:-

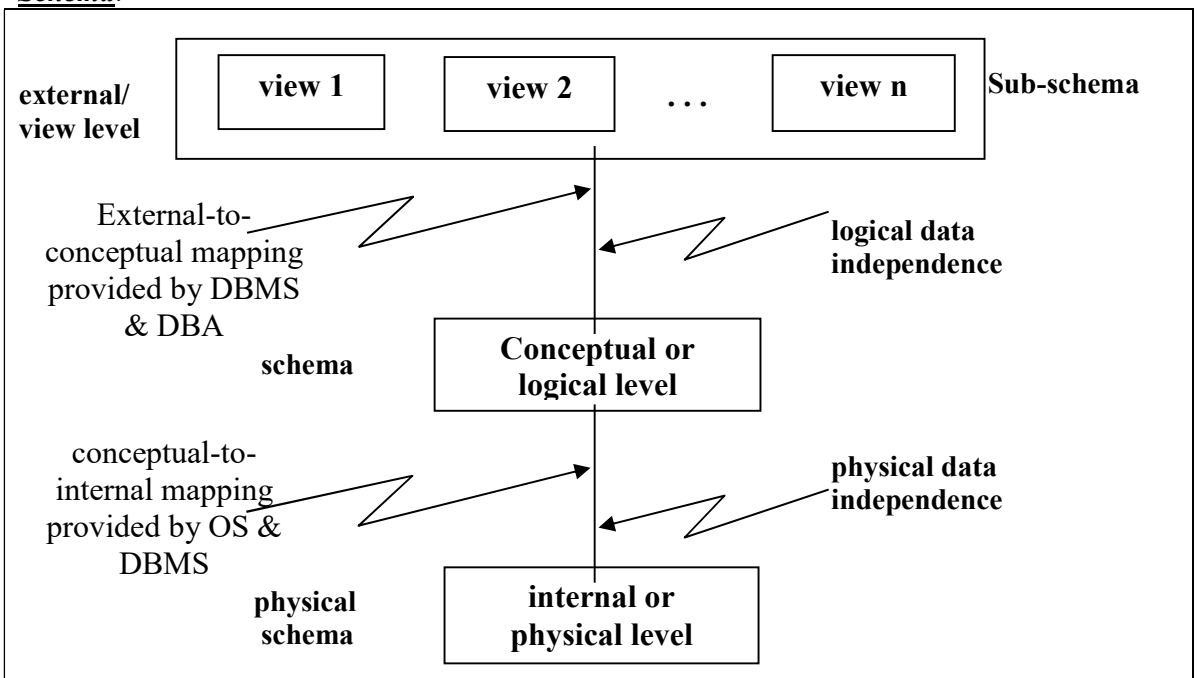
(i) Physical level or Internal level
(ii) Logical level or Conceptual level (**Schema**)
(iii) View level, user level or External level (**Sub-Schema**)

(i) Physical level or Internal level: - It explains how the data are actually stored on the secondary storage. It contains description of :

- Data structure used to store data on secondary storage
- Type of file organization
- Indexing method
- Access method
- Searching & sorting technique etc.

(ii) Logical level: - It describes what data are stored in the database & what relationship exists among them. It is also called the ***schema.*** It means that it describes complete view of database.

(iii) View level: - It is the highest level of abstraction & it describes only a part of entire database. Many users of the database may not be concerned with all information in database. They are provided the restricted and selected access of the complete database. It is also called ***Sub-Schema***.



Different Levels of Data Abstraction & Data Independence

NOTE: There is only one schema of a database; but there may be many sub-schema.

(B) Schema, Sub-Schema & Instances :-

(i) Schema:- It is the overall description of the database, *i.e.*, the files, their fields, their types, length, relationship between various files and fields etc.

Schema is the logical view of database for database administrator.

(ii) Sub-Schema:- As overall DBMS Structure may not be useful for all users, so a restricted portion of complete database is shown to the end users. It is called Sub-schema.

There is only one scheme & multiple Sub-schemas.

(iii) Instances: - The collection of information stored at a particular moment is called an instance. Instance changes after the operations, like – addition, deletion and updation. In the search operation the instance does not change.

(c) Data Independence :-

The ability to modify a level without affecting the other level is called data independence. It is highly desirable.

There are 2 type of data independence :

1. **Physical data independence:** It is the ability to modify the physical details without affecting the logical schema. It is provided by the mapping of DBMS & operating system.
2. **Logical data independence: -** It is the ability to modify the logical schema without causing applications programs to be rewritten. Also, the changes in view level should not cause changes in schema.

⇨ ***Note:*** *Logical data independence is more difficult to achieve because application program are heavily dependant upon the logical structure of the database.*

Example of University Database:

Conceptual schema:

- *Students(sid: string, name: string, login: string, age: integer, gpa:real)*
- *Courses(cid: string, cname:string, credits:integer)*
- *Enrolled(sid:string, cid:string, grade:string)*

Physical schema:

- Relations stored as unordered files.
- Index on first column of Students.

External Schema (View):

- *Course_info(cid:string,enrollment:integer)*

4. Database Languages:- There are 2 types of database languages:

1. Data Definition Language (DDL)
2. Data Manipulation Language (DML)

(1) DDL: The database schema is specified by a set of definition expressed by a special language called DDL. The commands may include:

- For creating tables
- Specifying primary key
- Establishing the relationship between files
- Storage structure of a file
- Access method of a file
- Integrity constraints like lower & upper bounds.
- Authorization information etc.

⇨ *NOTE: The results of compilation of DDL statement are stored in a special data structure called* ***data dictionary****. Data dictionary contains* ***data about data (i.e. metadata)****. It is referred to each time when the database is accessed.*

(2) DML :- (Data Manipulation Language**)** This language provides following facilities :

- Retrieval/access of information
- Insertion of new information
- Deletion of old information
- Modification of existing information

There are 2 type of DML :-

Procedural DML :- It requires the specification of **what** data are needed & **how** to get the data. The *"how"* needs algorithm. For example, Java, VB. C#, Pro-C, etc.

Non Procedural or Declarative DML: - It requires only the specification of **what** data is needed. For example SQL, QBE.

Comparison of Procedural & Declarative DML:

1) **Simplicity: -** Declarative DML is simpler because there is no need of specification of the procedure to get the data.
2) **Flexibility: -** The set of operation provided by declarative DML is very limited and

predefined. If complicated and uncommon operations are required, then procedural DML has to be used.

3) **Efficiency & Optimization: -** In case of declarative DML the steps of data manipulation are generated by the system it self. Sometimes the generated code is quiet inefficient & non-optimized. Whereas, expert procedural language programmers can write better code.

⇨ ***Note:*** *Now a day's a single language contains DDL & DML as separate ports.*

5a Types of DBMS users:

There are many database users. The classification is based on the responsibilities & level of expertise.

A). Database Designer :-

He is responsible for :

- Identifying which data is to be stored
- Selecting appropriate structure to represent & and to store the data.
- Normalization of database.

He works primarily during analysis & design phase of software development. He interacts with end user frequently to set up the appropriate structure and interface according to requirement. So, he/she is basically responsible for designing the schema.

B). Database Administrator(DBA) :-

Once database is developed, the person who has central control over the database is called DBA. He is responsible for :

- Authorizing access to the database.
- Coordinating & monitoring its use.
- Solutions to problems like violation of security.
- Backup activities
- Concurrency control management
- Change in the schema

C). End Users:-

These are the people who actually use the system frequently. ***DBMS primarily exists for these persons***. There are several categories of end users:

I. **Casual End User: -** He occasionally accesses the database, but may need variety of operations at different time. They use database query languages to specify their requirements. They may be managers of the organizations.

II. **Naive End User:-** Their main job is querying & updating the database using standard types of queries. They use ready made programs & well tested transactions, (also called ***canned transactions***). For example, Bank-tellers, Reservation clerks etc.

III. **Sophisticated End User: -**

It includes engineers, scientist, analysts etc. who thoroughly knows the facilities provided by DBMS. They can also write small programs or transactions to meet their requirement.

D). Application Programmers :-

These users can write sophisticated DML statements and complicated application programs as required by other users of the database.

E). Stand Alone Users :-

They maintain personal database by using ready made program packages that provide easy to use GUI. For example, a tax consultant is using Tally

5b Database Administrator (DBA):

DBA is the person who has central control over the database (data as well as programs). The main activities and responsibilities of DBA are as follows :

(i) Schema Definition: - DBA creates the original schema by writing DDL Statement. DDL Statements are compiled by DDL compiler; the results are stored in data-dictionary.

(ii) Storage structure and Access Method Definition: - DBA is responsible for selecting the appropriate organization of the database to store data on the secondary storage. He also fixes the access & search method that are most suitable according to the requirement.

(iii) Modification in Physical or Logical Schema:- In the dynamic environment hardware & S/W requirement keeps on changing. It is the responsibility of DBA to implement such changes with the help of application programmers & sophisticated users, according to end

user

(iv) Granting Authorization for Data Access:- It is the responsibility of DBA to limit & fix the authority or control of a user over the database. DBA can create new users , delete old users, modify the access permission of existing users.

(v) Integrity Specifications: - The data stored in the database must satisfy certain constraint. It is the responsibility of DBA to specify these constraints. For example - The marks of a subject must be from 0 to 100.

6 Data Dictionary, Data Directory or System Catalogue:

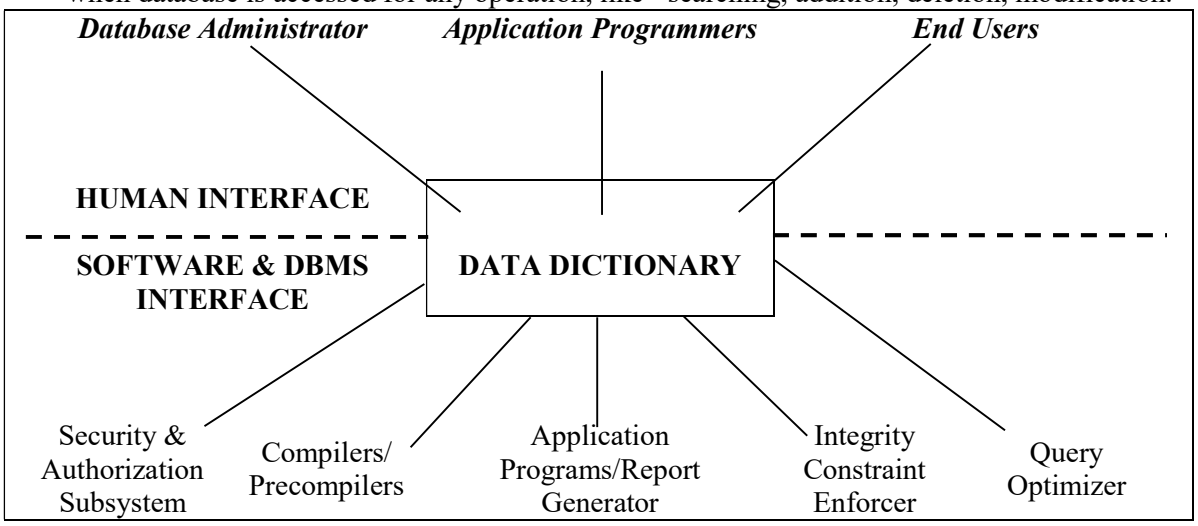
It contains ***meta-data***; it means ***data about data*** (also called ***metadata***), following be the information stored in the data dictionary:

- Names of the relation or table
- Names of attributed in each table
- Domain and length of attributes
- Integrity constraints about the attributes
- Accounting information
- No. of tuples or record in the relation
- Details of indexing structure etc.

Data dictionary contains description of schema. It is created when DDL statements are compiled.
Data dictionary may be classified as :

A). Passive data dictionary: It is used only by designers, users & administrators not by DBMS.

B). Active data dictionary or Data directory: It is used by the DBMS. It is referred to each time when database is accessed for any operation, like - searching, addition, deletion, modification.



Note:

1. ***Data dictionary is referred to each time when database is accessed for any operation, like - searching, addition, deletion, modification.***
2. ***Data dictionary contains description of schema. It is created when DDL statements are compiled.***
3. ***Since data dictionary is used heavily, it should have the organization and access method that provide fast access.***

7. Elements of DBMS/overall structure of DBMS :

The total system contains three basic components/levels:

(A) User level
(B) DBMS level
(C) Physical level

The total structure of DBMS can be viewed in two levels:

- query processor; the part that interacts with the user. Its main requirement is to provide simple & interactive environment.
- storage manager; the part that interacts with the hardware and auxiliary storage. Its main characteristic is to use resources optimally.

(A) User level : (Describes various types of user in brief)

(B) DBMS Level :It contains following components:

(1) Query Processor: Its components are:

(i) DDL Interpreter/Compiler: It interprets DDL statements & generates data dictionary.

(ii) DML Compiler: It translates DML statements into low level instruction that can be understood by the query evaluation engine. Moreover, it also optimizes the query.

(iii) Query Evaluation Engine: It executes the low level instructions generated by the DML precompiler.

(iv) Embedded DML pre-compiler: It converts DML statements embedded in an application program to normal procedure calls in the host language. The pre-compiler must interact with the DML compiler to generate the appropriate code.

(2) Storage Manager: Its components are:

(i) Authorization & Integrity Manager: It tests for the satisfaction of integrity constraints and checks the authority of users to access data.

(ii) Transaction Manager: It ensures that the database remains in consistent state despite system failures, and that concurrent transaction executions proceed without conflicting.

(iii) File Manager: It manages the allocation of space on disk storage and the data structures used to represent the information on the disk.

(iv) Buffer Manager: It is responsible for fetching the data to and form the disk storage and main memory, and decides what data to cache in memory.

(C) Physical Level

Following are the components:

(i) Data files: These store the actual data.

(ii) Data dictionary: It stores metadata.

(iii) Indices: These provide fast access to the data item.

8. Database Manager:

It is a software module that interacts with file manager and gets all works done on behalf of DBMS through file manager of operating system. It does following tasks:

(i) Interaction with file manager : - Dm is the lowest level of the DBMS it hider implementation details of o.s from the rest of DBMS

(ii) Integrity Enforcement: - The integrity constraints specified by the database administrator are actually enforced by database manager.

(iii) Security Enforcement: - The security provision specified by the DBA are actually enforced by DBM .

Note: - To enforce security & integrity constraints the design manages heavily user the data dictionary because all constraints are specified in that data structure.

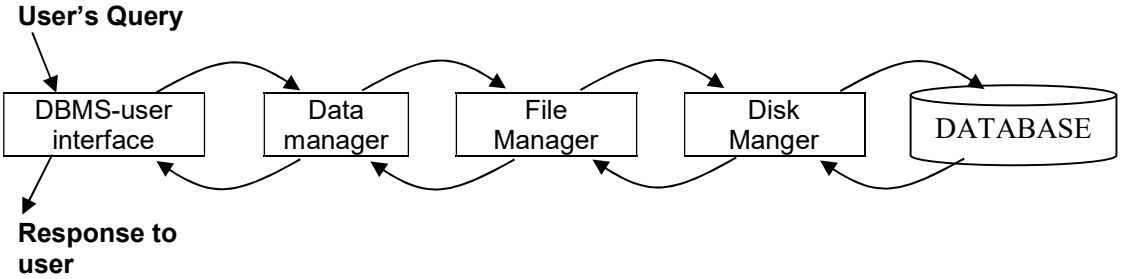
(iv) Back up & recovery: - The failure of the system may cause loss of data or inconsistent data state. it is the responsibility of DBM to recover it by using lock based transaction processing.

(v) Concurrency Control: - Multi-user suffers from concurrency problem like deadlocks. It is the responsibility of DBM to cope up with these problems.

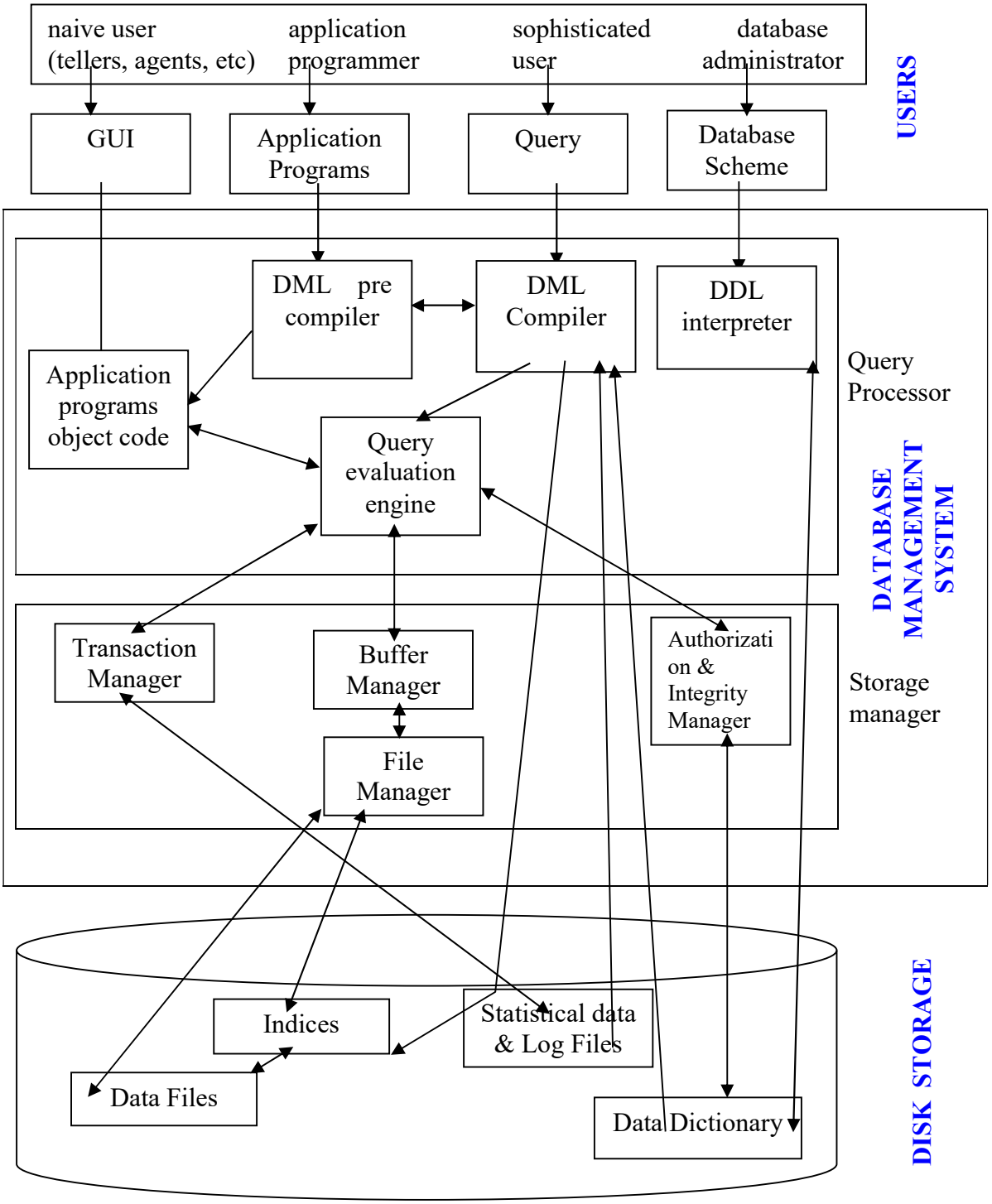
⇨ Note: Tightly coupled & loosely system: It is convenient for architectural purpose of to distinguish between data sub-language & its host language. But from users point of view these two may seen indistinguishable. If the two are heavily dependent on each other & hence are tough to distinguish, these are called tightly coupled. If these are clearly & easily separable then these are called loosely coupled.

9. Database Access steps:

Any access to the stored data is done by the data manager. The steps involved in database access are given in the figure:



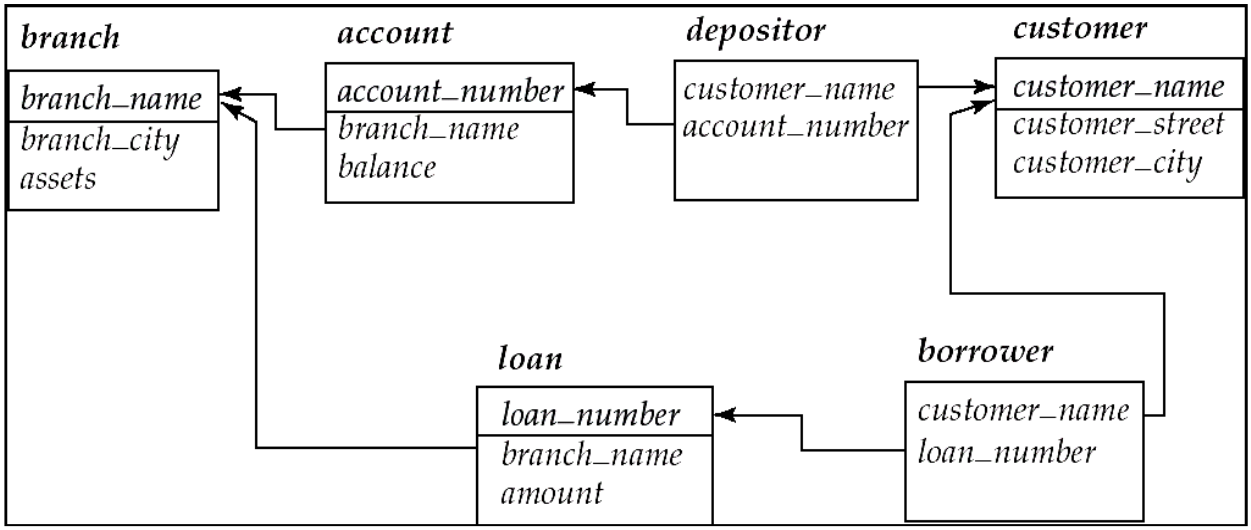
Components or Elements or over all structure of DBMS :-



Chapter 2
ER Model

The Bank Structure:

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)



1. Entity, Attribute & Relationship

(i) A **database** can be modeled as:
- a collection of entities,
- relationships among entities.

(ii) An **entity** is an object that exists and is distinguishable from other objects.
Example: specific person, company, event, plant

(iii) Entities have **attributes**, that describe properties/characteristics of entities.
Example: people have *names* and *addresses*

An entity is represented by a set of attributes that is descriptive properties possessed by all members of an entity set.

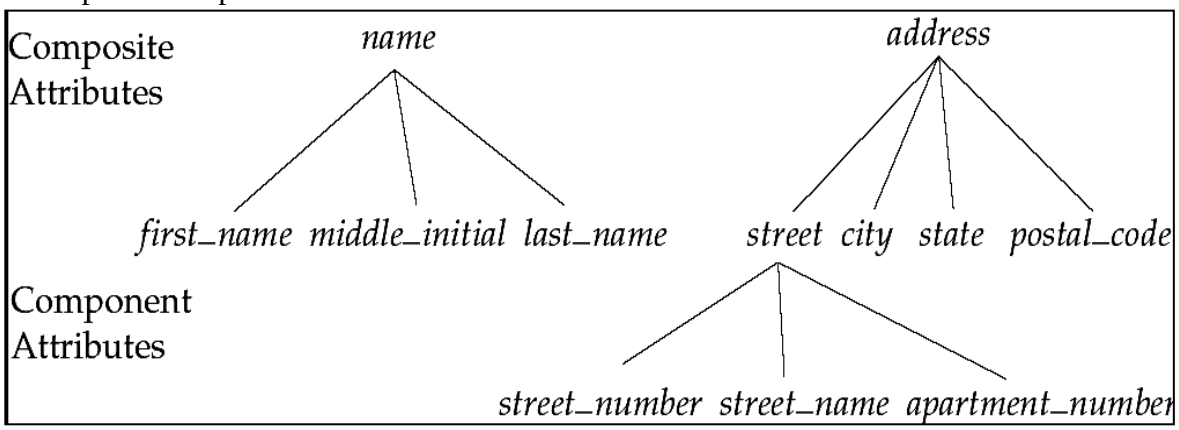
Example:

customer (customer_id, customer_name, customer_street, customer_city)
loan (loan_number, amount)

Domain – the set of permitted values for each attribute

Attribute types:

(a) Simple and *composite* attributes: For an account entity "balance" is a simple attribute. Example of composite attribute is as below:



(b) Single-valued and *multi-valued* attributes: For *account* entity "balance" is single valued, because it will have a single value for an account. For *person* entity *phone_numbers* is multi-valued attribute.
(c) Stored and Derived attributes: For *person* entity *date-of-birth* is stored attribute and *age* is the derived attribute. derived attributes can be computed from other attributes.

Example: age can be computed from date_of_birth

(iv) An **entity set** is a set of entities of the same type that share the same properties.

Example: set of all persons, companies, trees, holidays

(v) A **relationship** is an association among several entities

Example:

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

(vi) A **relationship set** is a mathematical relation among $n \geq 2$ entities, each taken from entity sets

$\{(e1, e2, \ldots en) \mid e1 \in E1, e2 \in E2, \ldots, en \in En\}$

where $(e1, e2, \ldots, en)$ is a relationship

Example:

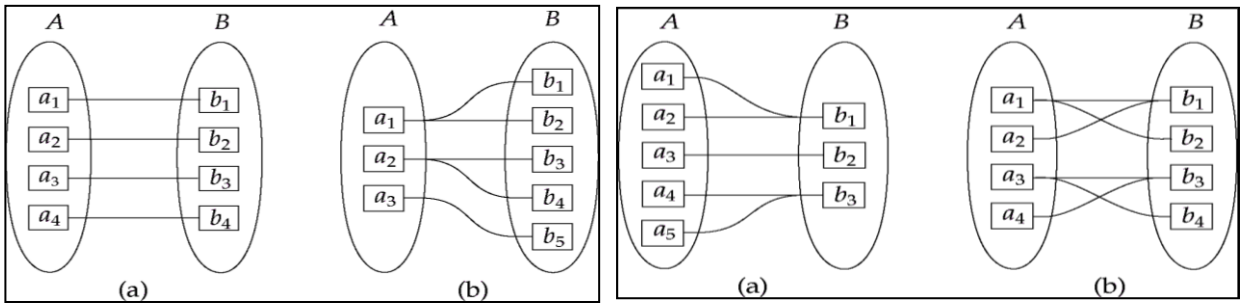
(Hayes, A-102) $\in$ *depositor*

2. Mapping Cardinality:

It is used to express the number of entities to which another entity can be associated via a relationship set. Normally, it is useful in describing binary relationship sets.

For a binary relationship set the mapping cardinality must be one of the following types:

One to one; One to many; Many to one; Many to many



Note: Some elements in *A* and *B* may not be mapped to any elements in the other set

3. Brief Concept of Keys:

A **super key** of an entity set is a set of one or more attributes whose values uniquely determine each entity.

A **candidate key** of an entity set is a minimal super key. So a candidate key has two essential properties:

- Unique Identification: Unique identifications
- Minimal or Non-redundancy: No proper sub set of the key can be the candidate key.

  Customer_id is candidate key of *customer*

  account_number is candidate key of *account*

Although several candidate keys may exist, one of the candidate keys is selected to be the **primary key.**

Primary Key of the Relationship Set:

The combination of primary keys of the participating entity sets forms a super key of a relationship set.

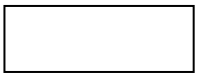
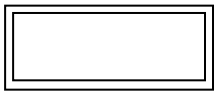
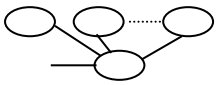
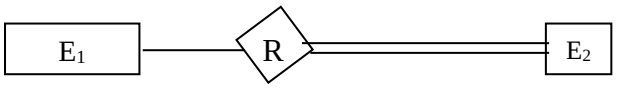
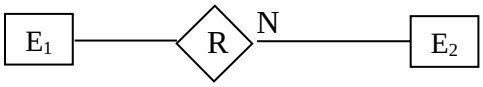
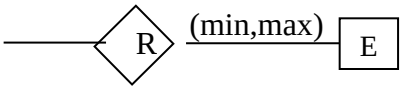
(*customer_id*, *account_number*) is the super key of *depositor*

NOTE: this means a pair of entity sets can have at most one relationship in a particular relationship set.

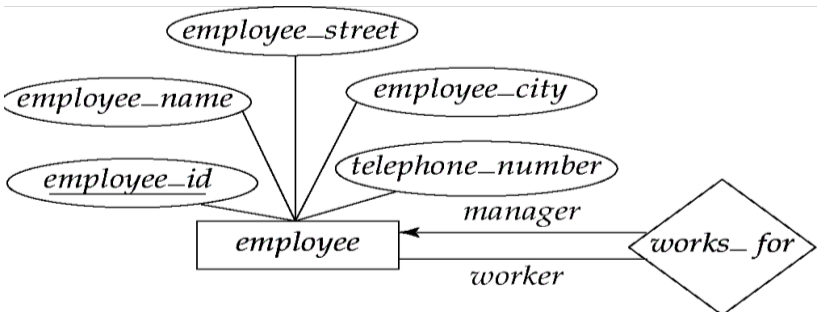
Example: if we wish to track all access_dates to each account by each customer, we cannot assume a relationship for each access. We can use a multivalued attribute though.

We must consider the *mapping cardinality* of the relationship set when deciding the what are the candidate keys.
We need to consider semantics of relationship set in selecting the *primary key* in case of more than one candidate key.

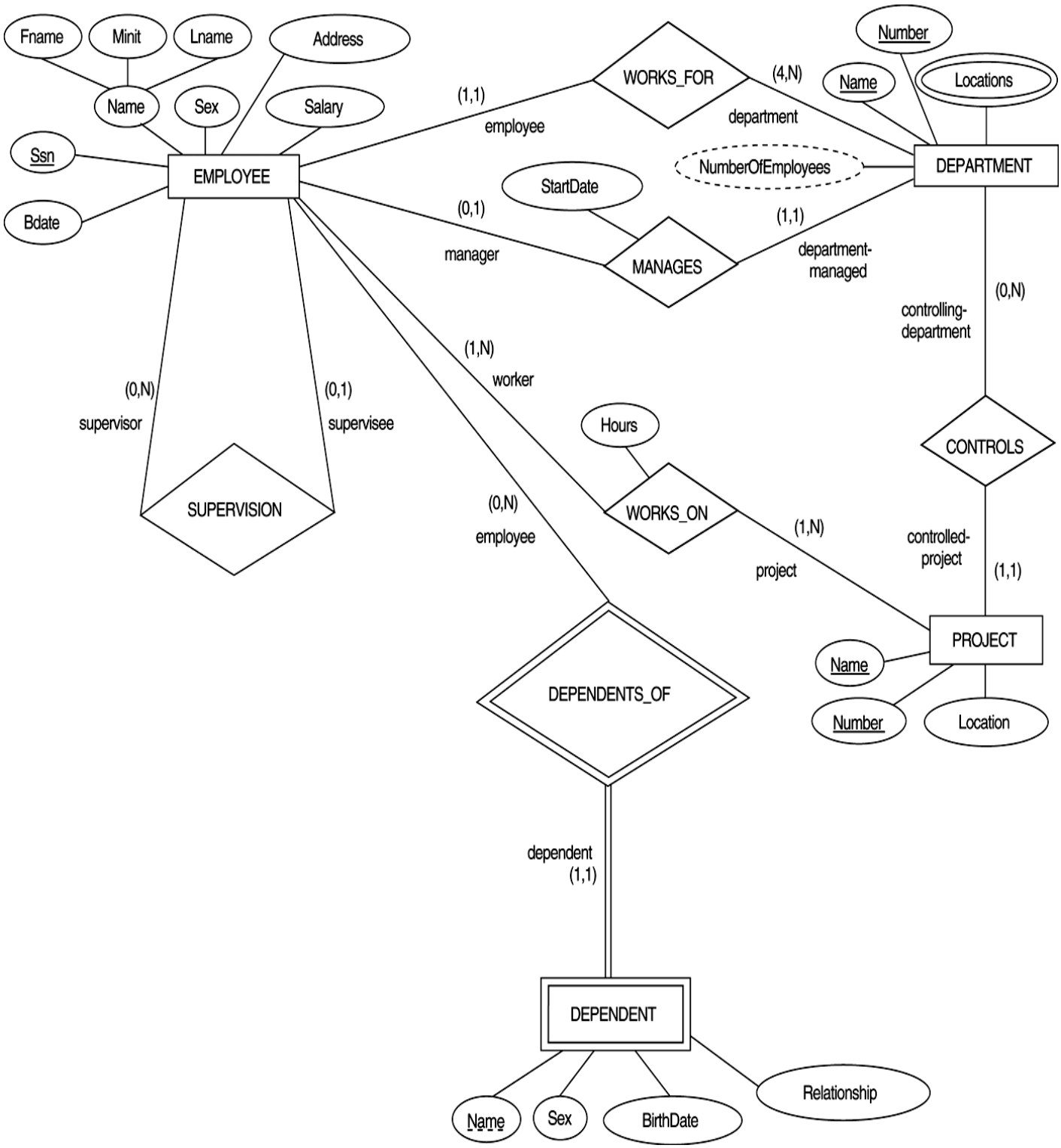
4. Diagrammatic Notations:

Symbol	Meaning
	ENTITY TYPE
	WEAK ENTITY TYPE
	RELATIONSHIP TYPE
	IDENTIFYING RELATIONSHIP TYPE
	ATTRIBUTE
	KEY ATTRIBUTE
	MULTIVALUED ATTRIBUTE
	COMPOSITE ATTRIBUTE
	DERIVED ATTRIBUTE
	TOTAL PARTICIPATION OF E2 IN R
	CARDINALITY RATIO 1:N FOR E1:E2 IN R
	STRUCTURAL CONSTRAINT (min, max) ON PARTICIPATION OF E IN R

Role:
Entity sets of a relationship need not be distinct. 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.



Sample ER Diagram for company database:



5. Constraint on Relationship

Constraints on Relationship Types(Also known as ratio constraints)

(i) **Maximum Cardinality** or Mapping Cardinality

One-to-one (1:1)

One-to-many (1:N) or Many-to-one (N:1)

Many-to-many

(ii) **Minimum Cardinality** (also called ***participation constraint*** or ***existence dependency*** constraints)

zero (optional/partial participation, not existence-dependent)

one or more (total participation , mandatory, existence-dependent)

We express cardinality constraints by drawing either a directed line (→), signifying "one," or an undirected line (—), signifying "many," between the relationship set and the entity set.

(i) Maximum Cardinality or Mapping Cardinality

Customer

id	name	street	city
100	ram	bada	gwalior
101	ajay	new market	bhopal
104	vijay	charbag	lucknow
110	ram	regal	indore
115	mohan	taj colony	agra

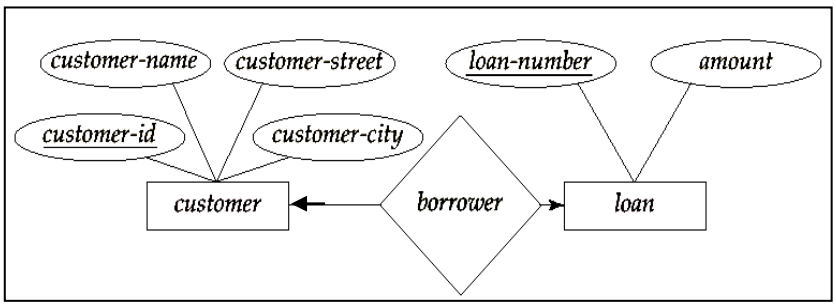
Loan

loan#	amount
l-500	10000
l-600	8000
l-200	8000

(fig-a)

One-to-one relationship:

A customer is associated with at most one loan via the relationship *borrower;* A loan is associated with at most one customer via *borrower*.

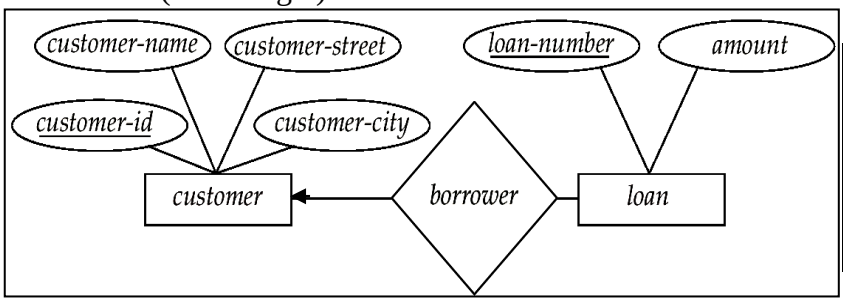


one-to-one : **(fig-b)**

customer id	loan#
100	l-500
104	l-600
115	l-200

One-to-many relationship:

a loan is associated with at most one customer via *borrower*, a customer is associated with several (including 0) loans via *borrower*.

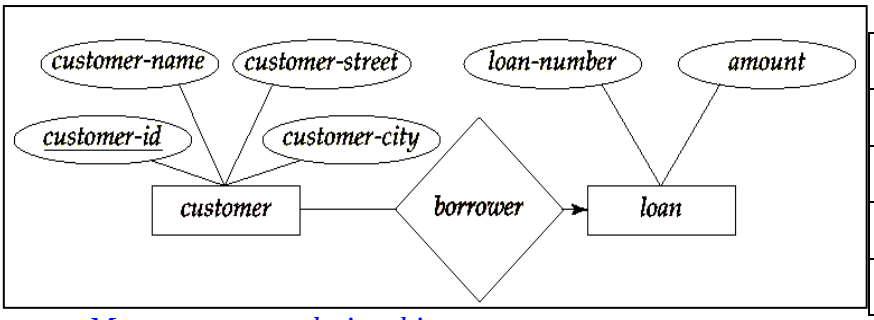


one-to-many : **(fig-c)**

customer id	loan#
100	l-500
104	l-600
104	l-200

Many-to-one relationship:

a loan is associated with several (including 0) customers via *borrower*, a customer is associated with at most one loan via *borrower.*

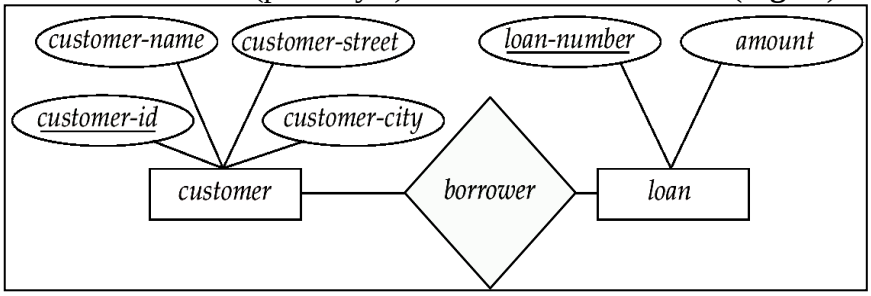


many-to-one : **(fig-d)**

customer id	loan#
100	l-500
104	l-600
115	l-200
110	l-200

Many-to-many relationship:

A customer is associated with several (possibly 0) loans via borrower. A loan is associated with several (possibly 0) customers via borrower. (**fig - e**)



customer id	loan#
100	l-500
104	l-600
115	l-200
110	l-200
100	l-600

(ii) Minimum Cardinality or Participation of an Entity Set in a Relationship Set:

Total participation (indicated by double line): Every entity in the entity set participates in at least one relationship in the relationship set.

E.g. 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. e.g.: Participation of customer in borrower is partial; because it is not necessary that that every customer will take a lone.

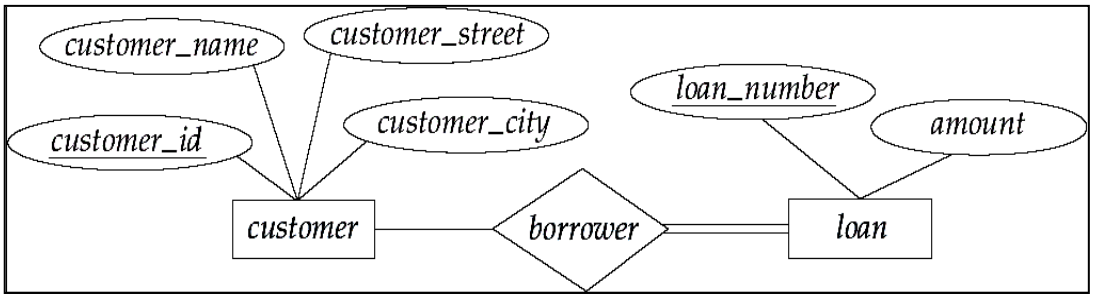


fig-a

Alternative Representation of Constraint:

As an alternative approach, participation constraint is specified on *each participation* of an entity type E in a relationship type R. This representation specifies that each entity e in E participates in *at least* min and *at most* max relationship instances in R.

Default (no constraint): min=0, max=n

Must have min≤max, min≥0, max ≥1

Examples:

A department has *exactly one* manager and an employee can manage *at most one* department.

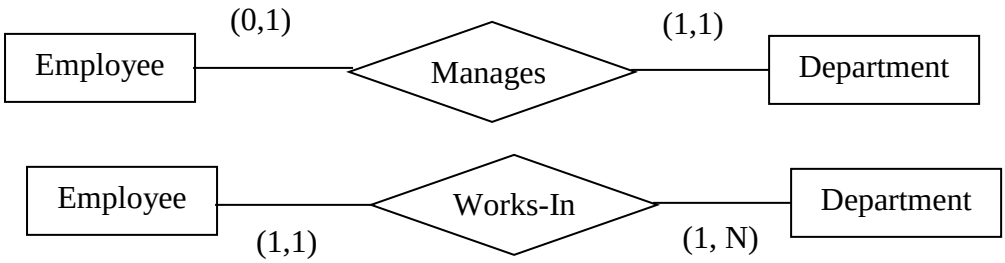
Specify (0,1) for participation of EMPLOYEE in MANAGES

Specify (1,1) for participation of DEPARTMENT in MANAGES

An employee can work for *exactly one* department but a department can have *any number of employees*.

Specify (1,1) for participation of EMPLOYEE in WORKS_FOR

Specify (0,n) for participation of DEPARTMENT in WORKS_FOR



7. Design Issues

Use of entity sets vs. attributes

Choice mainly depends on the structure of the enterprise being modeled, and on the semantics associated with the attribute in question.

Use of entity sets vs. relationship sets

Possible guideline is to designate a relationship set to describe an action that occurs between entities

Binary versus n-ary relationship sets

Although it is possible to replace any nonbinary (*n*-ary, for $n > 2$) relationship set by a number of distinct binary relationship sets, a *n*-ary relationship set shows more clearly that several entities participate in a single relationship.

Placement of relationship attributes

It may be sometimes confusing whether to place an attribute with a relationship or with an entity.

9. Weak Entity:

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 an *identifying entity set.*

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 depicted using a *double diamond*.

The Discriminator: 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.

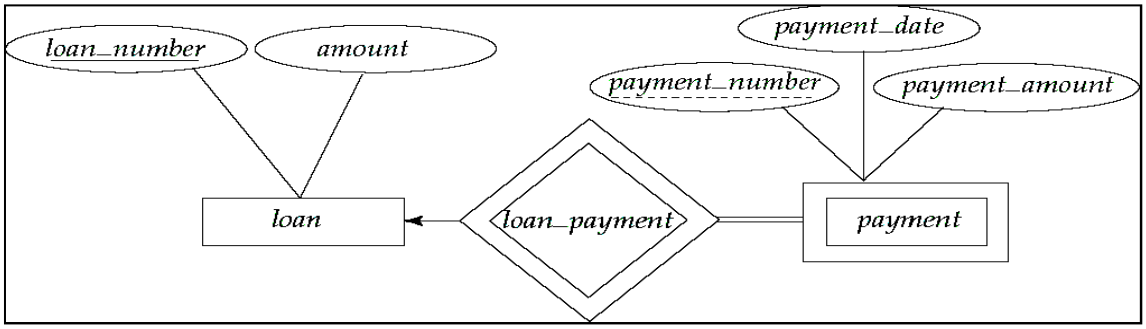
The Primary Key: The primary key of a weak entity set is formed by the primary key of the strong entity set on which the weak entity set is existence dependent, plus the weak entity set's discriminator.

We depict a weak entity set by double rectangles. We underline the discriminator of a weak entity set with a dashed line. For example:

payment_number – discriminator of the *payment* entity set

Primary key for *payment* – (*loan_number, payment_number*)

fig-a



Note: the primary key of the strong entity set is not explicitly stored with the weak entity set, since it is implicit in the identifying relationship.

entity set of payment:

these are related with loan no L-200

related with L-600

payment no.	date	amount
1	***1/1/2000***	***1000***
2	10/2/2000	1000
1	***1/1/2000***	***1000***
3	10/4/2000	1500
2	4/4/2000	1800

If *loan_number* were explicitly stored, *payment* could be made a strong entity, but then the relationship between *payment* and *loan* would be duplicated by an implicit relationship defined by the attribute *loan_number* common to *payment* and *loan*

10. Generalization

Top-down design process; we designate sub-groupings within an entity set that are distinctive from other entities in the set. These sub-groupings become lower-level entity sets that have attributes or participate in relationships that do not apply to the higher-level entity set.

It is depicted by a *triangle* component labeled ISA (E.g. *customer* "is a" *person*).

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.

Specialization

It is the process of defining a set of subclasses of a superclass. The set of subclasses is based upon some distinguishing characteristics of the entities in the superclass.

Example: {SECRETARY, ENGINEER, TECHNICIAN} is a specialization of EMPLOYEE based upon *job type*. An entity may have several specializations of the same superclass

Example: Another specialization of EMPLOYEE based in *method of pay* is {SALARIED_EMPLOYEE, HOURLY_EMPLOYEE}.

- Superclass/subclass relationships and specialization can be diagrammatically represented in EER diagrams

- Attributes of a subclass are called specific attributes. For example, TypingSpeed of SECRETARY
- The subclass can participate in specific relationship types. For example, BELONGS_TO of HOURLY_EMPLOYEE

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. Specializations can be at multiple level.

E.g. *permanent_employee* vs. *temporary_employee*, in addition to *officer* vs. *secretary* vs. *teller* Each particular employee would be

a member of one of *permanent_employee* or *temporary_employee*,
and also a member of one of *officer*, *secretary*, or *teller*

The ISA relationship also referred to as **superclass - subclass** relationship

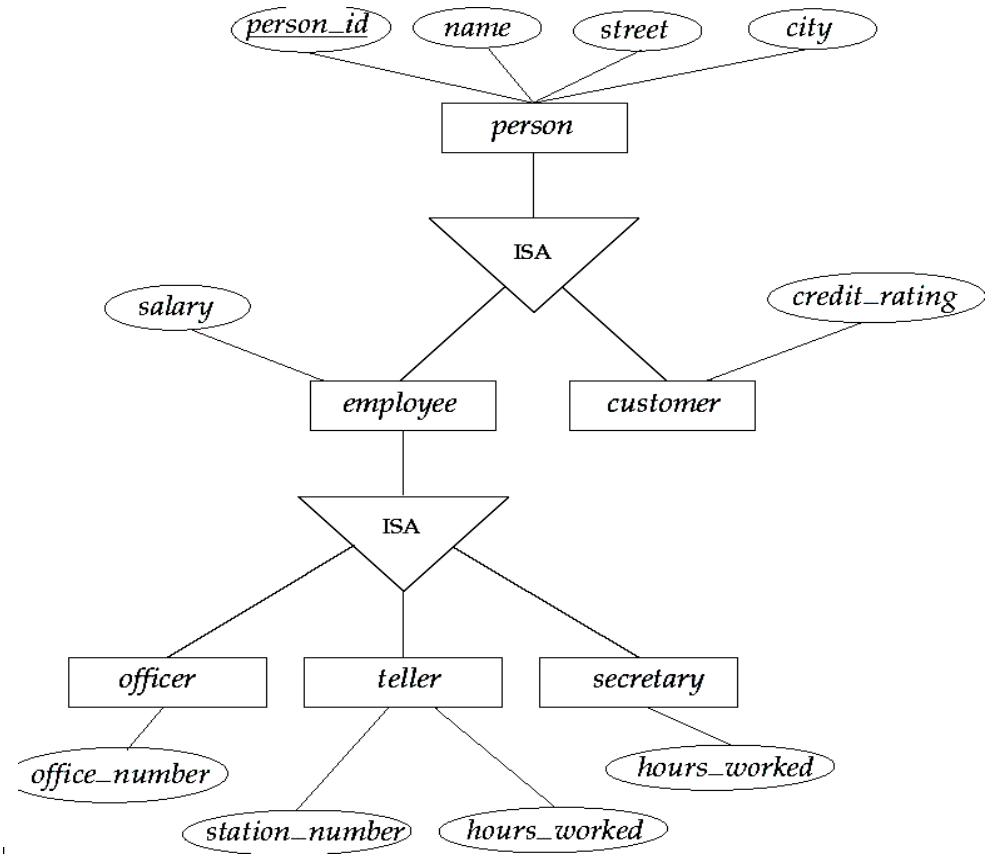


fig (a)

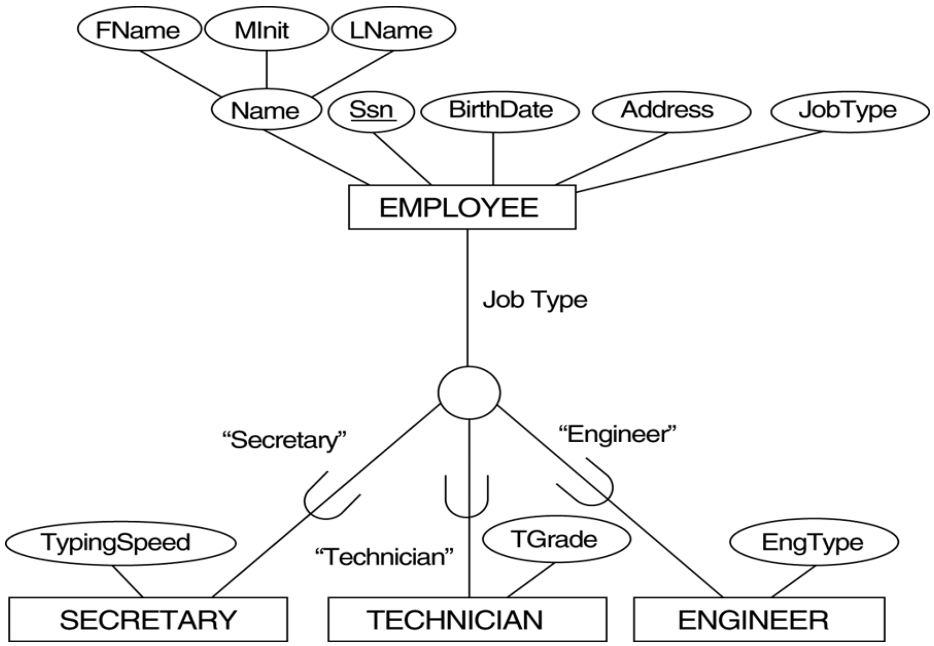


fig (b)

Diagrammatic notation sometimes used to distinguish between generalization and specialization. Arrow pointing to the generalized superclass represents a generalization. Arrows pointing to the specialized subclasses represent a specialization. We do not use this notation because it is often subjective as to which process is more appropriate for a particular situation. Data Modeling with Specialization and Generalization

- A superclass or subclass represents a set of entities
- Shown in rectangles in EER diagrams (as are entity types)
- Sometimes, all entity sets are simply called classes, whether they are entity types, superclasses, or subclasses

11. Constraint on Generalization & Specialization:

It is to determine which higher level entity will participate in low level entity set.

(I) predicate-defined, attribute defined or user-defined

- If we can determine exactly those entities that will become members of each subclass by a condition, the subclasses are called *predicate-defined* (or condition-defined) subclasses (**fig-a on page 10)**
  - Condition is a constraint that determines subclass members
  - Display a predicate-defined subclass by writing the predicate condition next to the line attaching the subclass to its superclass

- If all subclasses in a specialization have membership condition on same attribute of the superclass, specialization is called an *attribute defined*-specialization(the example on same page) (**Example fig-b on page 8**)
  - Attribute is called the defining attribute of the specialization
  - Example: JobType is the defining attribute of the specialization {SECRETARY, TECHNICIAN, ENGINEER} of EMPLOYEE

- If no condition determines membership, the subclass is called *user-defined*
  - Membership in a subclass is determined by the database users by applying an operation to add an entity to the subclass
  - Membership in the subclass is specified individually for each entity in the superclass by the user

(II) Disjoint or Overlapping Constraint:

- Specifies that the subclasses of the specialization must be *disjointed* (an entity can be a member of at most one of the subclasses of the specialization). It is specified by *d* in EER diagram (**fig-b** on page 10)
- If not disjointed, *overlap*; that is the same entity may be a member of more than one subclass of the specialization. It is specified by *o* in EER diagram. (**fig-a on page 10**)

(III) Completeness(Total) or Partial Constraint:

- *Total* specifies that every entity in the superclass must be a member of some subclass in the specialization/ generalization. It is shown in EER diagrams by a double line. (**fig a and b on page 10**)
- *Partial* allows an entity not to belong to any of the subclasses. It is shown in EER diagrams by a single line.

NOTE:

Hence, we have four types of specialization/generalization:

- *Disjoint, total*
- *Disjoint, partial*
- *Overlapping, total*
- *Overlapping, partial*

Note: Generalization usually is total because the superclass is derived from the subclasses.

The previous fig(page 14) is the example of disjoint and partial specialization.

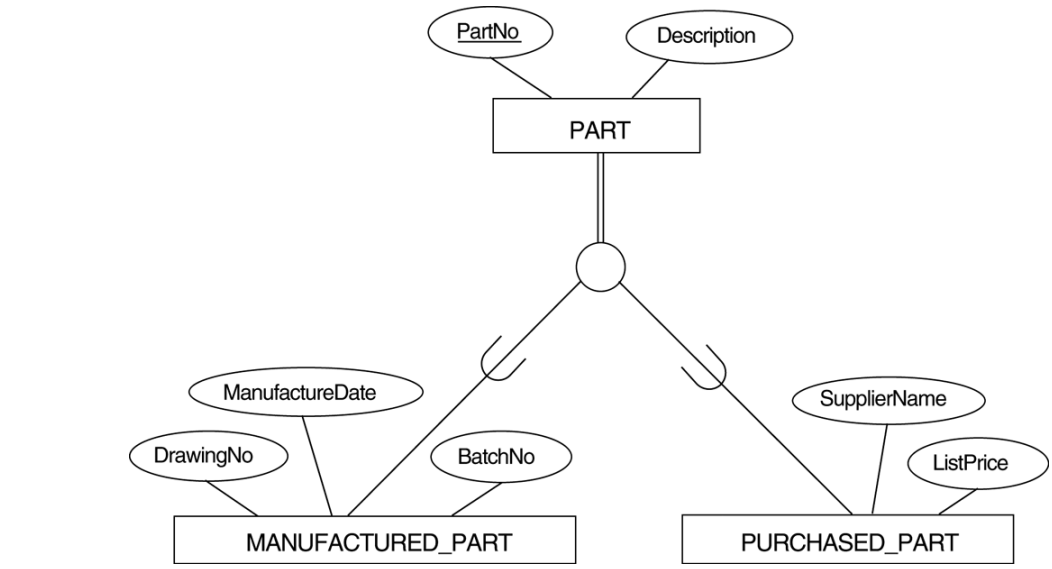


fig (a)

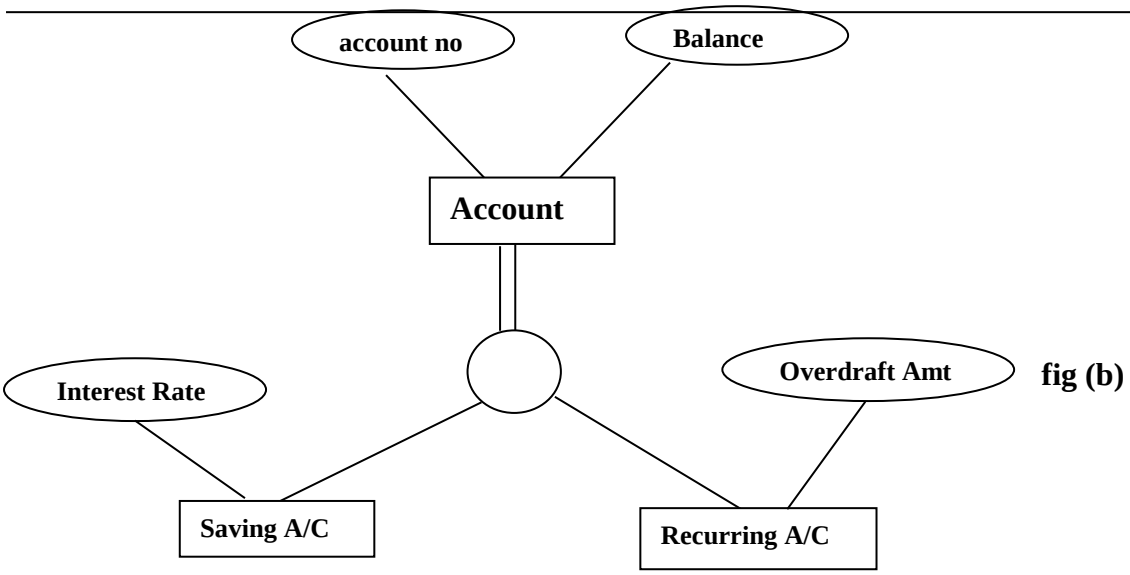


fig (b)

12. Hierarchy & Lattices

A subclass may itself have further subclasses specified on it. It forms:

- a hierarchy or
- a lattice

Hierarchy has a constraint that every subclass has only one superclass (called *single inheritance*).

In a *lattice*, a subclass can be subclass of more than one superclass (called *multiple inheritance*).

In a lattice or hierarchy, a subclass inherits attributes not only of its direct superclass, but also of all its predecessor superclasses

A subclass with more than one superclass is called a *shared subclass.*

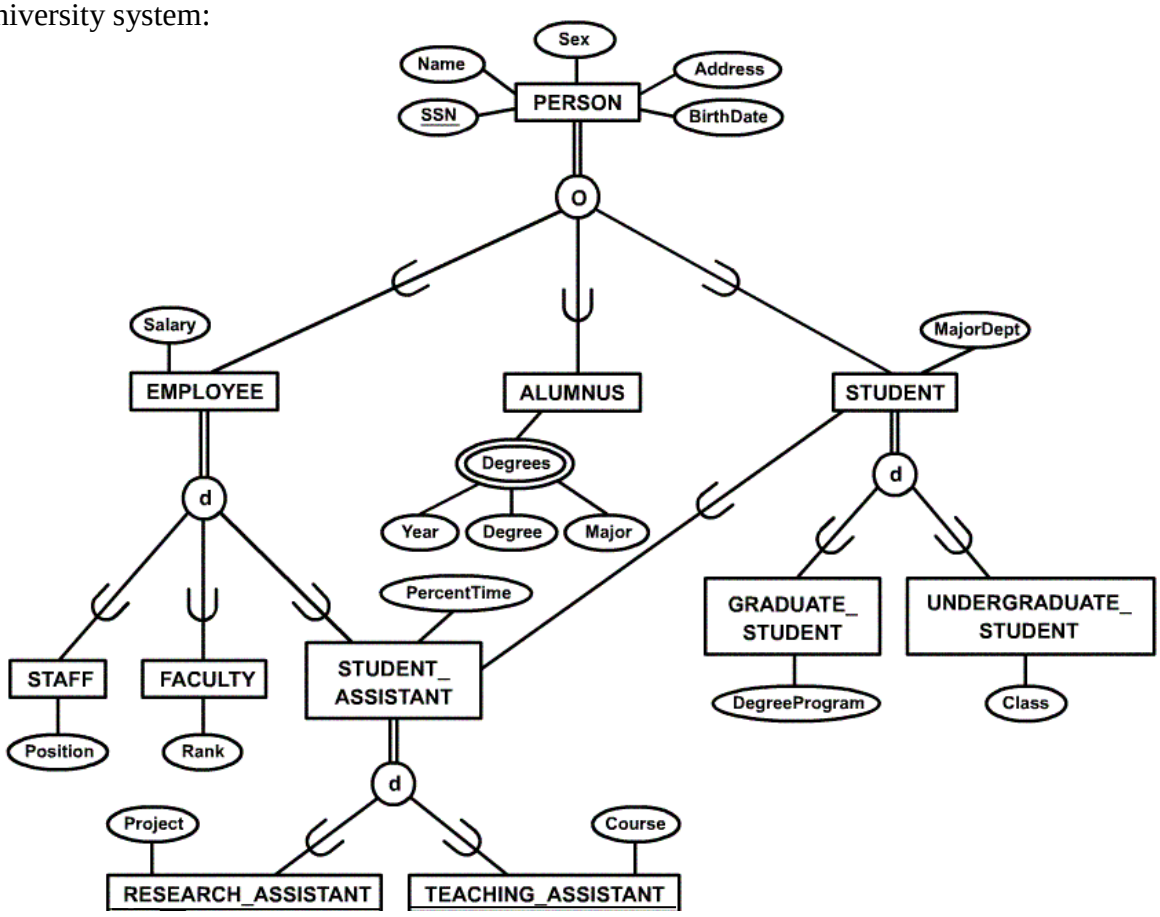
Specialization can have specialization hierarchies or lattices, or generalization hierarchies or lattices.

In specialization, start with an entity type and then define subclasses of the entity type by successive specialization (top down conceptual refinement process).

In generalization, start with many entity types and generalize those that have common properties (bottom up conceptual synthesis process).

In practice, the combination of two processes is employed.

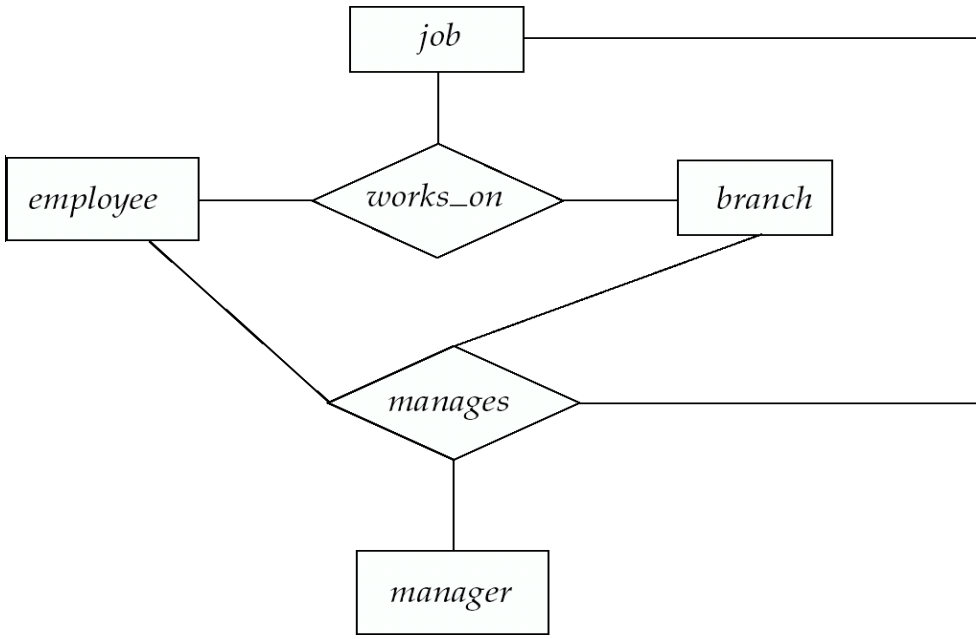
An example of a lattice:
University system:



13. Abstractions: hiding of details for simplicity.

Classification	A is a member of class B
Aggregation	B,C,D are aggregated into A A is made of/composed of B,C,D
Generalization	B,C,D can be generalized into A, **B is-an A, C is- an A, D is-an A**
Specialization	A can be specialized into B,C,D B,C,D are special cases of A.

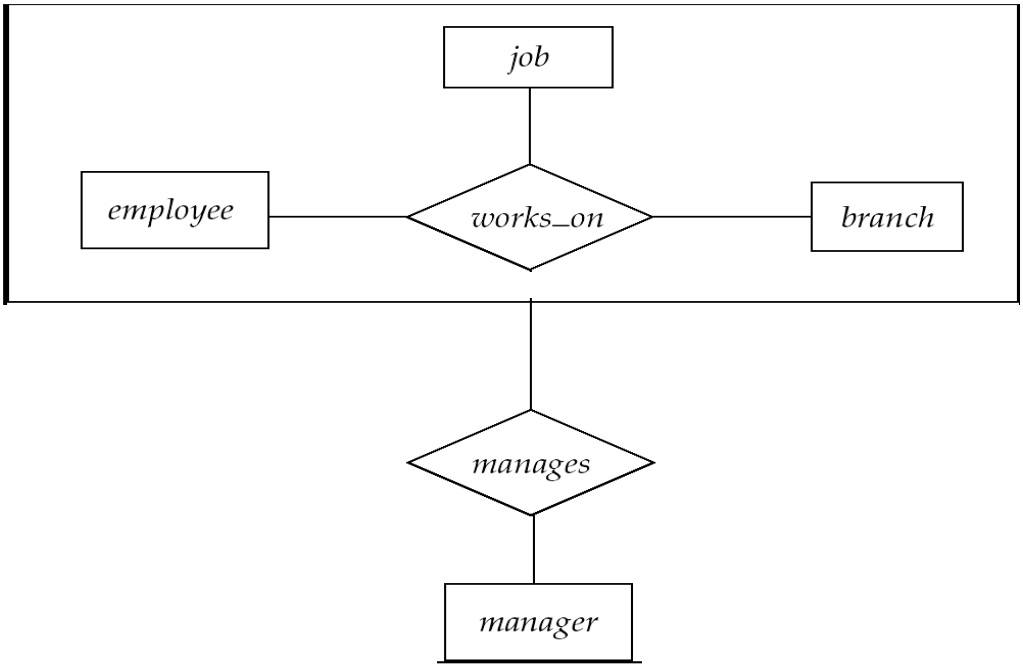
14. Aggregation:
Consider the ternary relationship *works_on*.



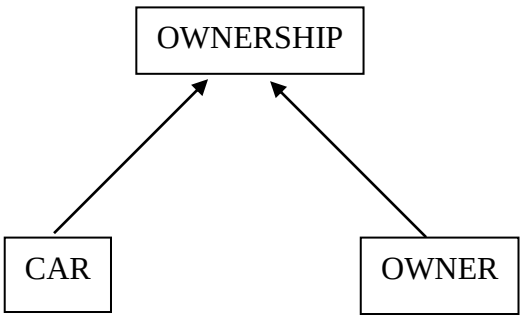
Suppose we want to record managers for tasks performed by an employee at a branch.
Relationship sets *works_on* and *manages* represent overlapping information. Every *manages* relationship corresponds to a *works_on* relationship. However, some *works_on* relationships may not correspond to any *manages* relationships. So we can't discard the *works_on* relationship.
We can eliminate this redundancy via *aggregation:*

- Treat relationship as an abstract entity
- Allows relationships between relationships
- Abstraction of relationship into new entity

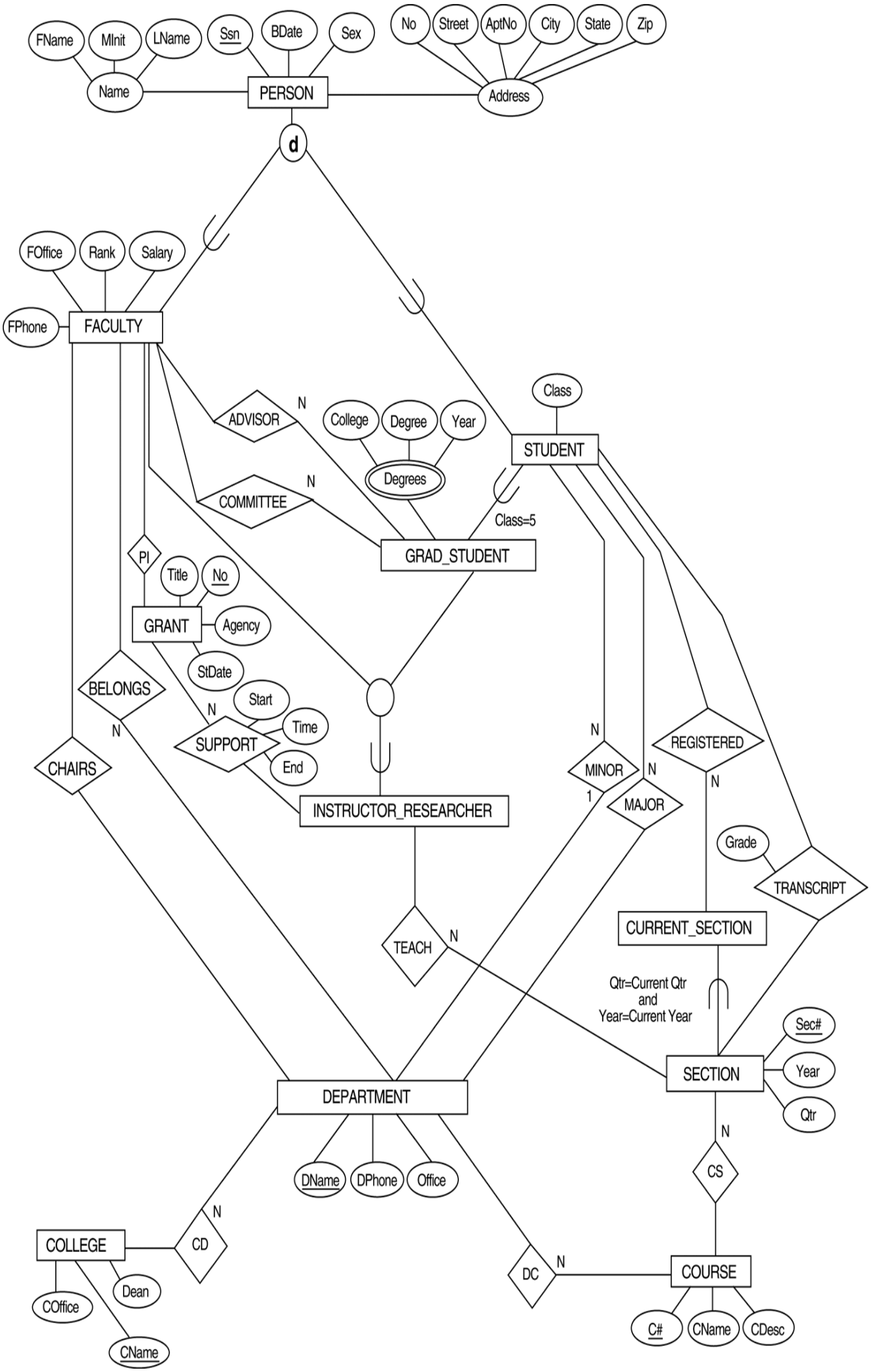
Without introducing redundancy, the following diagram represents:
An employee works on a particular job at a particular branch
An employee, branch, job combination may have an associated manager



Two Contexts for Aggregation
We aggregate two or more classes into a higher level concept. It may be considered a relationship or association between them.
Context1: CAR is an aggregate (composition) of Chassis, Drive-train, Other Systems, Wheels.
Context 2: OWNERSHIP is an aggregate (relationship) of CAR and OWNER



Example:
ER diagram of university system:


FName
MInit
LName
Name
Ssn
BDate
Sex
PERSON
No
Street
AptNo
City
State
Zip
Address
d
FOffice
Rank
Salary
FPhone
FACULTY
ADVISOR
N
College
Degree
Year
Degrees
COMMITTEE
N
Class
STUDENT
Class=5
GRAD_STUDENT
PI
Title
No
GRANT
Agency
StDate
BELONGS
N
Start
Time
End
SUPPORT
N
CHAIRS
INSTRUCTOR_RESEARCHER
N
MINOR
1
N
MAJOR
REGISTERED
N
Grade
TRANSCRIPT
CURRENT_SECTION
TEACH
N
Qtr=Current Qtr
and
Year=Current Year
Sec#
Year
Qtr
SECTION
DEPARTMENT
N
CS
DName
DPhone
Office
N
CD
COLLEGE
Dean
COffice
CName
DC
N
COURSE
C#
CName
CDesc

15. Reduction of ER-Diagrams into Tables / Relational Schema

Primary keys allow entity sets and relationship sets to be expressed uniformly as *relation schemas* that represent the contents of the database. A database which conforms to an E-R diagram can be represented by a collection of schemas. For each entity set and relationship set there is a unique schema that is assigned the name of the corresponding entity set or relationship set. Each schema has a number of columns (generally corresponding to attributes), which have unique names.

(I) Representing Entity Sets as Schemas (refer fig-a on page 7)

A strong entity set reduces to a schema with the same attributes.

loan(**loan-number**, amount)

A weak entity set becomes a table that includes a column for the primary key of the identifying strong entity set.

payment (***loan_number, payment_number***, *payment_date, payment_amount*)

(II) Representing Relationship Sets as Schemas (refer figures on page 5)

A relationship set is represented as a schema with attributes for the primary keys of the two participating entity sets, and any descriptive attributes of the relationship set.

Example: schema for relationship set borrower

borrower (*customer_id, loan_number*)

The primary key of the relationship set depends upon the mapping cardinality between the participating entity sets.

mapping cardinality from customer-to-loan	primary key of ***borrower***	refer example
1-to-1	p.k. of either customer or loan	fig-b on page 5
1-to-many	p.k. of loan	fig-c on page 5
many-to-1	p.k. of customer	fig-d on page 5
many-to-many	p.k. of customer + p.k. of loan	fig-e on page 5

(III) Redundancy of Schemas

(a) Relation with weak entity set: ***(refer fig-a on page 7)***

The schema/table corresponding to a relationship set linking a weak entity set to its identifying strong entity set is **redundant**.

Example: The *payment* schema already contains the attributes that would appear in the *loan_payment* schema (i.e., *loan_number* and *payment_number*).

(b) Relation with total participation constraint: ***(refer fig-a on page 6)***

If an entity set has *total participation* in a relationship set; then the table of the entity set can be merged into the table of the relationship set. Example:

There are three table for customer-loan schema.

customer(**id**, name, street, city)
loan(**loan#**, amount)
borrower(**id, loan#)**

Since, **loan has total participation in borrower**, hence only two tables are sufficient:

customer(**id**, name, street, city)
borrower(**id, loan#**, amount)

(c) Relation with partial participation constraint:

If participation is *partial participation* on the "many" side, replacing a schema by an extra attribute in the schema corresponding to the "many" side could result in null values.

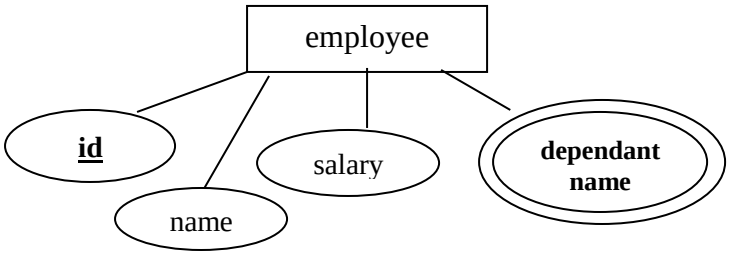
(IV) Composite and Multi-valued Attributes

(A) Composite attributes are flattened out by creating a separate attribute for each component attribute.

Example: given entity set *custome*r with composite attribute *name* with component attributes *first_name* and *last_name* the schema corresponding to the entity set has two attributes:

name.first_name and *name.last_name*

(B) A multivalued attribute M of an entity *E* is represented by a separate schema *EM*. Schema *EM* has attributes corresponding to the primary key of *E* and an attribute corresponding to multivalued attribute *M*.



The two table will be:

employee1(**id,** name, salary)

employee2(**id,** dependant-name)

(V) Aggregation (Refer figure on page 12)

To represent aggregation, create a table containing:

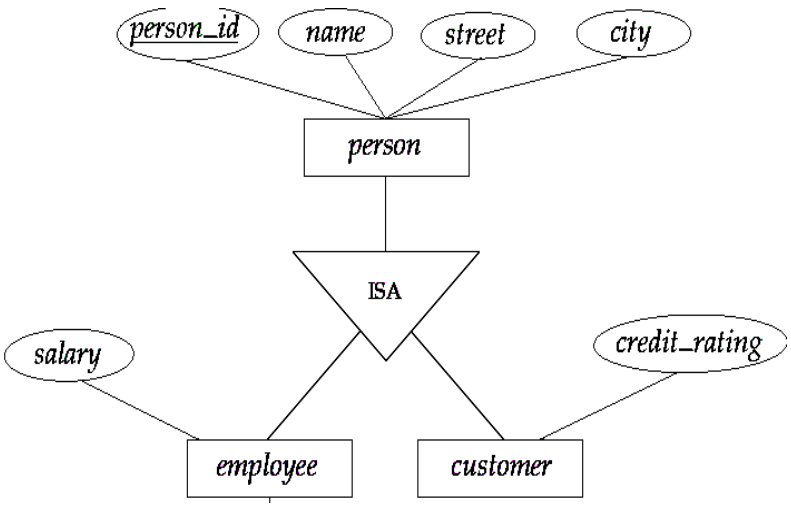
- primary key of the aggregated relationship,
- the primary key of the associated entity set
- any descriptive attributes

For example, to represent aggregation manages between relationship *works_on* and entity set *manager*, create a schema

manages (*employee_id, branch_name, title, manager_name*)

Schema *works_on* is redundant provided we are willing to store null values for attribute *manager_name* in relation on schema *manages.*

(V) Specialization & Generalization:



Method 1:
Form a schema for the higher-level entity. Form a schema for each lower-level entity set, include primary key of higher-level entity set and local attributes.

schema	attributes
person	*person-id, name, street, city*
customer	*person-id, credit_rating*
employee	*person-id, salary*

Drawback: getting information about, an *employee* requires accessing two relations, the one corresponding to the low-level schema and the one corresponding to the high-level schema.

Method 2:
Form a schema for each entity set with all local and inherited attributes

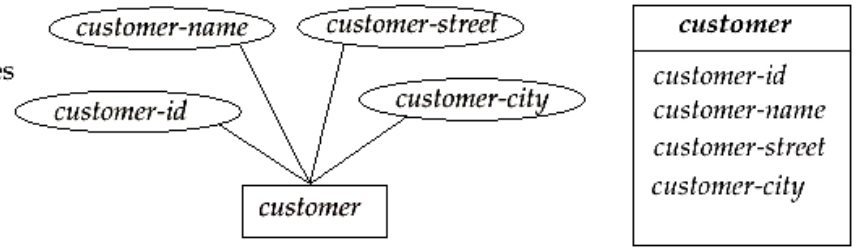
schema	attributes
customer	*person-id, name, street, city, credit_rating*
employee	*person-id , name, street, city, salary*

If specialization is *total*, the schema for the generalized entity set (*person*) not required to store information. It can be defined as a "view" relation containing union of specialization relations. But explicit schema may still be needed for foreign key constraints.
Drawback: *street* and *city* may be stored redundantly for people who are both customers and employees

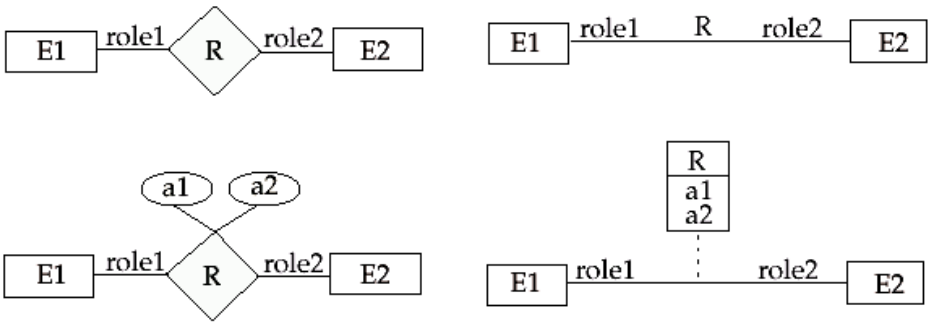
16. UML

It is *Unified Modeling Language*.
UML has many components to graphically model different aspects of an entire software system.UML Class Diagrams correspond to E-R Diagram, but there are several differences too.

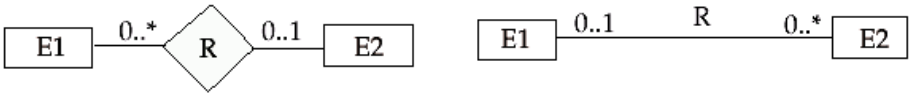
1. Entity sets and attributes



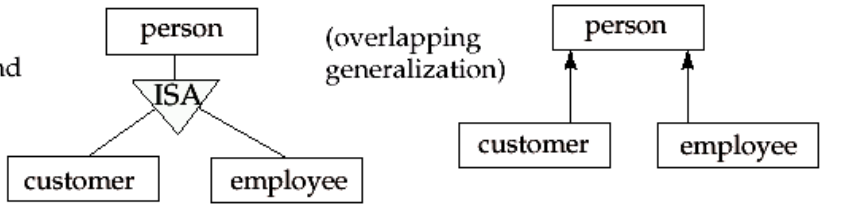
2. Relationships

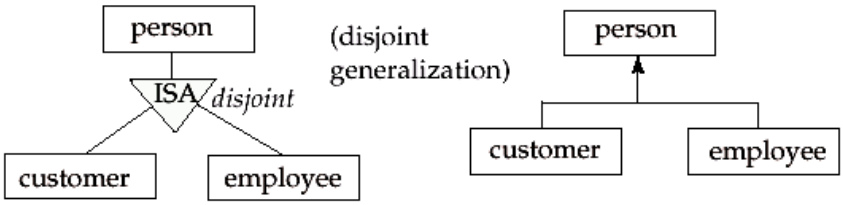


3. Cardinality constraints

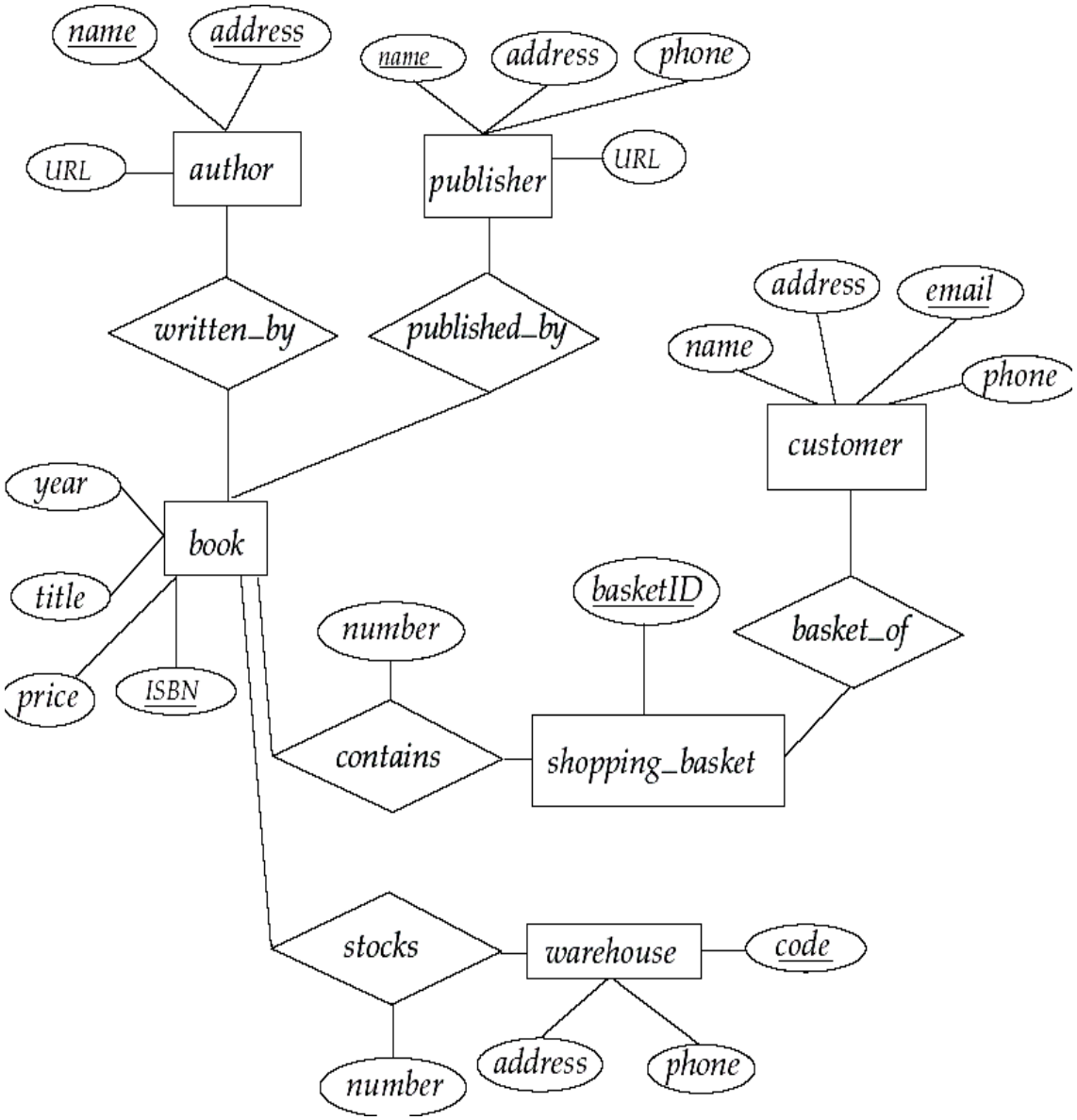


4. Generalization and Specialization

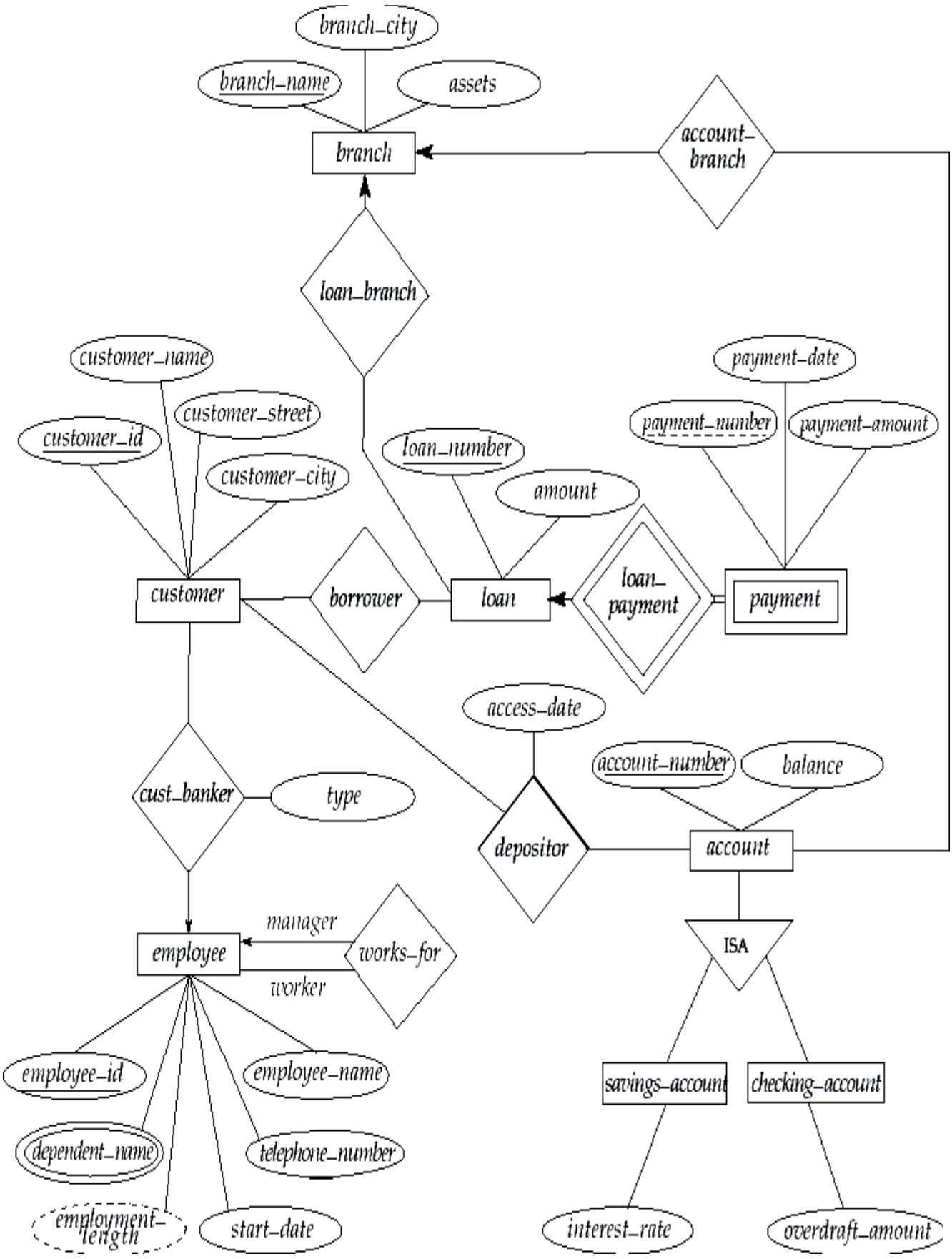




ER Diagram of a book store



ER Diagram of a bank:



Chapter 3
Relational Model

1. Some Definitions :-

1. Table & Relations:-

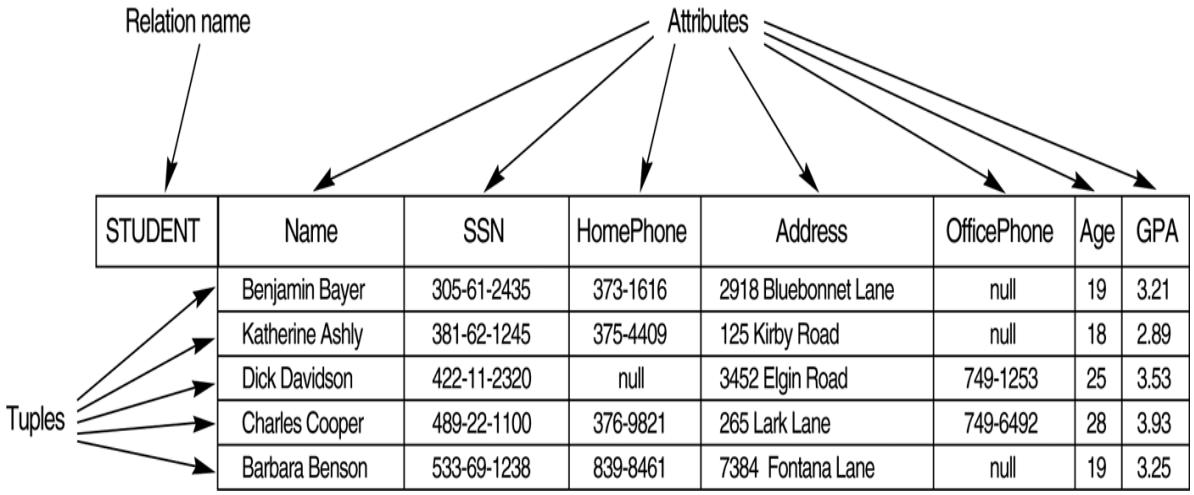
A relational database consists of collection of tables each of which is assigned a unique name.

A row in a table represents a relationship among a set of values. A row is also called a tuple.

A table is called a relation. The main advantage of treating tables as relation is that we can apply operations of relational algebra on tables. This mathematical treatment simplifies the operation.

RELATION: A table of values

- A relation may be thought of as a **set of rows or tuples**. ~~**(record)**~~
- A relation may alternately be though of as a **set of columns**. ~~**(field)**~~
- Each row represents a fact that corresponds to a real-world **entity** or **relationship**.
- Each row has a value of an item or set of items that uniquely identifies that row in the table.
- Sometimes row-ids or sequential numbers are assigned to identify the rows in the table.
- Each column typically is called by its column name or column header or attribute name.



STUDENT	Name	SSN	HomePhone	Address	OfficePhone	Age	GPA
	Benjamin Bayer	305-61-2435	373-1616	2918 Bluebonnet Lane	null	19	3.21
	Katherine Ashly	381-62-1245	375-4409	125 Kirby Road	null	18	2.89
	Dick Davidson	422-11-2320	null	3452 Elgin Road	749-1253	25	3.53
	Charles Cooper	489-22-1100	376-9821	265 Lark Lane	749-6492	28	3.93
	Barbara Benson	533-69-1238	839-8461	7384 Fontana Lane	null	19	3.25

Characteristics of a relation:

- **Ordering of tuples in a relation r(R)**: The tuples are *not* considered to be ordered, even though they appear to be in the tabular form.
- **Ordering of attributes in a relation schema R** (and of values within each tuple): We will consider the attributes in R(A1, A2, ..., An) and the values in t=<v1, v2, ..., vn> to be *ordered* .
- (However, a more general *alternative definition* of relation does not require this ordering).
- **Values in a tuple**: All values are considered *atomic* (indivisible). A special **null** value is used to represent values that are unknown or inapplicable to certain tuples.

2. **Domain :-** A domain is a pool of values from which specific attributes of specific relation draw these actual values. In a relation the values contained are a subset of the corresponding domain.
3. **Cartesian product: -** It is represented by x (cross). Let these are 2 domains:

   Name ={"Ram", "Shyam", "Mohan"}

   Account = {100,101}

   The Name x Account is

Name	Account
Ram	100
Ram	101
Shyam	100
Shyam	101
Mohan	100
Mohan	101

Name	Account
Ram	100
Shyam	100
Mohan	101

Any actual relation r on the 2 domains will be subset of this cartesian product A possible actual relation may be:

4. **Formal Definition of a Relation :-**

A relation R on a collection on domains D1, D2, …, D_n consist of two part :-

- A Header
- A Body

The header consists of a fix set of attributes. The body consists of a set of tuples or rows. Each tuple is an element of cartesian product of the domains; i.e.:

D1 x D2 x …….D_n .

So, there exist a tuple t, where

{ t1 t ∈ (D1 x D2 X …..D_n) }

5. Degree or Arity of relation :-

The arity or degree of a relation is simply the number of attributes of relation.

2. Concept of Keys:

1. Super Key :-

An attribute or set of attributes which is able **uniquely identity** a tuple is called a super key. The super key may be redundant or reducible it means that a proper subset of it may also be able to provide uniqueness. **eg.** Consider a relation

Student (en#, name, dob, father's name, address, branch)

Super key may be

(en# + name)

The attribute ***"name"*** is redundant because only ***en#*** can also provide uniqueness. In other words the super key is ***(en# + name)***; its proper subset en# can also provide uniqueness.

2. Candidate Key :-

A candidate key for a relation R is an attribute or set of attributes that satisfy following properties :

- **Uniqueness: -** No two distinct tuples of relation ***r*** have same values for these attributes in total.
- **Non Redundancy or Irreducibility:-** No proper subset of these attributes can provide uniqueness.

For eg. : In the above relation following are the candidate keys :

a. en #
b. name + father's name
c. name+address

Note:

1) ***These may be more than one candidate keys in a relation.***
2) ***Every candidate key is a super key but reverse is not true.***
3) ***A candidate key may be atomic or composite. A composite key contains more than one attribute.***

3. Primary Key:- There may be more then one candidate keys. Any one can be specified for unique identification during data definition. It is called primary key.

eg.: Out of above three candidate key, any one say ***en#*** may be taken as primary key.

4. Alternate Key:- If there are more than one candidate keys, any one is selected as primary key remaining are called alternate keys. **eg.** In the above relation

(Name + father's name) is the alternate key

5. Foreign Key: - Let there are two relations R_1 & R_2. Let A is the primary key of relation R_1. If A occurs in R_2 also, then it in a foreign key of R_2.

In the relation instance r2 only those values of A are allowed, that are in the relation instance r1.

R1(**A**, B, C)

R2(**X**, A) *A is the foreign key /reference key in R2 with respect to R1*

A	B	C
a1	b2	c1
a2	b1	c4
a3	b6	c8
a5	b9	c4

X	A
x1	a3
x5	a1
x6	a3
x8	a5

Note:- *Foreign keys are used to establish relationship between two tables*

eg. *Library data base*

book (book # , titter, author, publisher name, edition, price) — *Foreign Key with respect to publisher*

Publisher (publisher name, address, branch)

Only those publisher-names are permitted in the *publisher* relation that are available in the *book* relation

3. Integrity Constraint Rules:

A relational mode consists of same integrity rules to maintain the database consistency.

Constraints are *conditions* that must hold on *all* valid relation instances. There are three main types of constraints:

- Key constraints – Every relation must have a key for unique identification. It has been discussed.
- Entity integrity constraints.
- Referential integrity constraints

1. Entity Integrity :-

It is related with primary key. If attribute A of relation R is the prime attribute (a part of primary key) of R then A cannot have null values. It means that if a relation R has primary key (P1, P2, …, P_n), then any subset of this cannot be null.

Example :- account (cust-ID, acc# , balance)

Account sharing is possible & one person can have more than one accounts. In this case the primary key of the relation is (cust-ID, acc#)

cust-ID	acc#	Balance	
@	1728	1000	X
@	@	2000	X
S-510	@	10,000	X
t-1000	1580	25,000	√

Note:- *A relational database forces user to follow entity integrity rule automatically.*

2. Referential Integrity :-

Given two relations ***R*** & ***S***. Let ***R*** refers to the relation ***S*** through a set of attributes forms a foreign key in ***R***. The value of foreign key in a tuple in ***r*** must either be equal to primary key of a tuple of ***s*** or be entirely null. (**example – same as foreign key example on page number 3**)

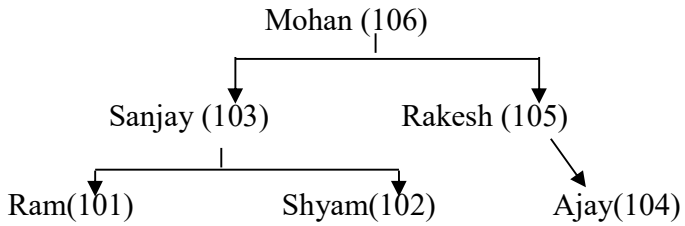
Note: - *The relation R & S are not necessarily distinct.*

Another example: We take a relation manager

Manager (emp#, name, manager-id)

An employee is headed by his manager. A manager is also employee of the company.

emp#	Name	Manager
101	Ram	103
102	Shyam	103
103	Sanjay	106
104	Ajay	105
105	Rakesh	106
106	Mohan	@



There is a record of Ram (101). Its manager field is 103, the name is Sanjay.

There is a record of Mohan (108). Its manager field is @ (NULL).

Deletion while maintaining Referential Integrity :-

If we want to delete a tuple that is a target of foreign key reference then to maintain database integrity, following options are available :-

a) Cascaded or Domino Deletion :- All tuples that contain references to the deleted tuple should also be deleted. This may cause the deletion of other tuples in turn.
eg.:- The tuple of emp# 105 is deleted, it will cause the tuple of 104 to be deleted necessarily.

b) Only the tuples that are not referenced be other tuple can be deleted **eg:-** Only the tuple of employees 101,102 and104 can be deleted.

c) The tuple can be deleted & the corresponding foreign key attribute are set to null.
eg. :- If tuple of employee 105 is deleted, the manager value of employee 104 will be set to null .

Note :- *The choice depends upon the* ***logic of application***

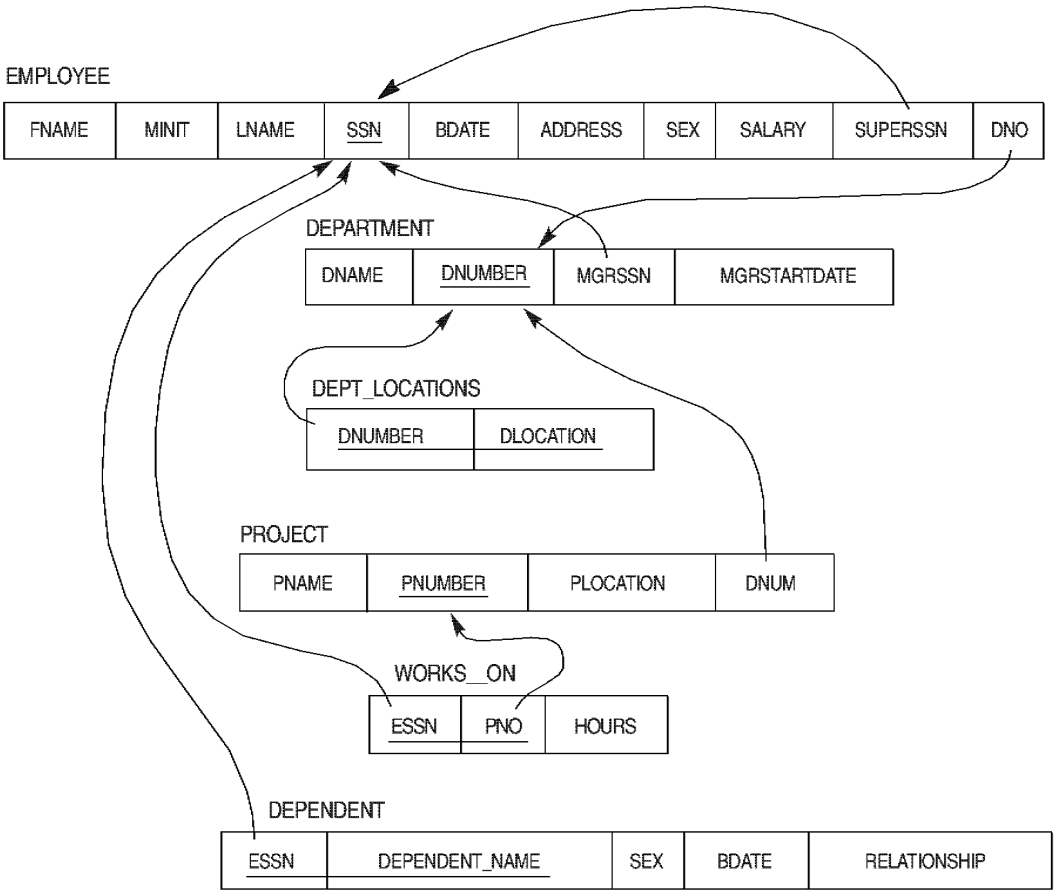
3. Attribute Integrity :-

Every attribute is required to satisfy the constraint that its values are taken from the relevant domain only

eg:- A marks of the student can be integer only & from 0 to 100 .

NOTE: Integrity rules are restrictions. These rules restrict user from invalid modifications in the database. If the database satisfies all the integrity (constraint) rules, it is said to be in the **consistent state**.

Examples:



Exercises:

1. Consider the following relations for a database that keeps track of student enrollment in courses and the books adopted for each course:

STUDENT(SSN, Name, Major, Bdate)
COURSE(Course#, Cname, Dept)
ENROLL(SSN, Course#, Quarter, Grade)
BOOK_ADOPTION(Course#, Quarter, Book_ISBN)
TEXT(Book_ISBN, Book_Title, Publisher, Author)

2. Consider the following six relations for an order-processing database application in a company:

CUSTOMER (Cust#, Cname, City)
ORDER (Order#, Odate, Cust#, Ord_Amt)
ORDER_ITEM (Order#, Item#, Qty)
ITEM (Item#, Unit_price)
SHIPMENT (Order#, Warehouse#, Ship_date)
WAREHOUSE (Warehouse#, City)

3. Consider the following relations for a database that keeps track of business trips of salespersons in a sales office:

SALESPERSON (SSN, Name, Start_Year, Dept_No)
TRIP (SSN, From_City, To_City, Departure_Date, Return_Date, Trip_ID)
EXPENSE (Trip_ID, Account#, Amount)

Specify the foreign keys for this schema, stating any assumptions you make.

***4*. Let R be a relation of arity n. What is the maximum no. of candidate keys the relation r might possess?**

Chapter 4
Relational Algebra

1. Basic Structure

Formally, given sets $D1, D2, \ldots. Dn$ a **relation** r is a subset of

$D1 \times D2 \times \ldots \times Dn$

Thus, a relation is a set of n-tuples $(a1, a2, \ldots, an)$ 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*

2. Definitions:

(A) Relation Schema:

$A1, A2, \ldots, An$ are *attributes*

$R = (A1, A2, \ldots, An)$ is a *relation schema*

Example:

Customer_schema = (*customer_name, customer_street, customer_city*)

$r(R)$ is a *relation* on the *relation schema R.* Example:

customer (Customer_schema)

(B) Relation Instance:

The current values (*relation instance*) of a relation are specified by a table. An element t of r is a *tuple*, represented by a *row* in a table. Order of tuples is irrelevant (tuples may be stored in an arbitrary order).

(C) Database:

A database consists of multiple relations. Information about an enterprise is broken up into 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 (e.g., two customers own an account)
- the need for null values (e.g., represent a customer without an account)

Hence, there is a need of normalization.

3. Query Languages:

It is a language in which user requests information from the database. Categories of languages are:

- Procedural
- Non-procedural, or declarative

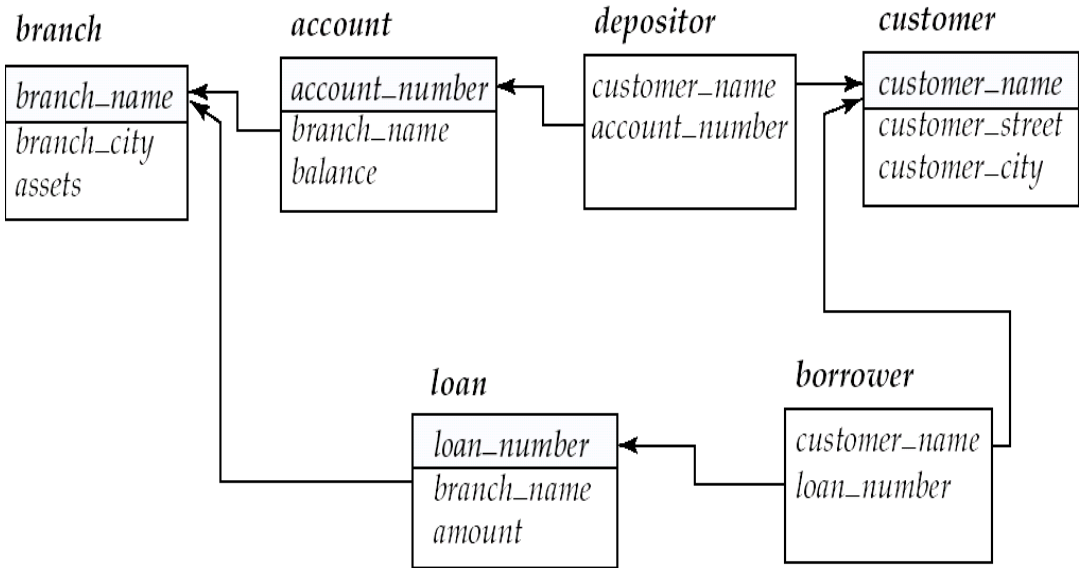
There is one more category called **"Pure" languages**. Pure languages form underlying basis of query languages that people use. Theses are:

- Relational algebra
- Tuple relational calculus
- Domain relational calculus

4. Example Relation:

Consider the bank schema:

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)



5. Relational Algebra:

It is a procedural pure language. It has six basic operators:

select: σ
project: Π
union: ∪ intersection: ∩
set difference: –
Cartesian product: x
rename: ρ

The operators take one or two relations as inputs and produce a new relation as a result.

6. Select Operation:

Notation: $\sigma\, p(r)$; p is called the **selection predicate or condition**

Defined as:

$\sigma\, p(r) = \{t \mid t \in r$ **and** $p(t)\}$

Where p is a formula in prepositional calculus consisting of **terms** connected by :

∧ (**and**), ∨ (**or**), ¬ (**not**)

Each **term** is one of:

<attribute> *op* <attribute> or <constant>

where *op* is one of: =, ≠, >, ≥. <. ≤

A	B	C	D
a	a	1	7
a	b	5	7
b	b	12	3
b	b	23	10

$\sigma_{A=B \wedge D > 5}(r)$

A	B	C	D
a	a	1	7
b	b	23	10

: Find all accounts of Morar branch.

$\sigma_{branch\_name="Morar"}(account)$

7. Projection Operation:

Notation:

$$\prod_{A_1, A_2, \ldots, A_k}(r)$$

where *A1*, *A2* are attribute names and *r* is a relation name.
The result is defined as the relation of *k* columns obtained by erasing the columns that are not listedDuplicate rows removed from result, since relations are sets.

A	B	C	D
a	a	1	7
a	b	5	7
b	b	12	3
b	b	23	10

$\prod_{A, C}(r)$

A	C
a	1
a	5
b	12
b	23

Example: To eliminate the *branch_name* attribute of *account*

$\prod_{account\_number, balance}(account)$

8. Union, Intersection and Set Difference Operation:

To perform above operations, following conditions must be true:

- *r, s* must have the *same* **arity** (same number of attributes)
- The attribute domains must be **compatible**

(these are the necessary conditions also for intersection and subtraction)

A	B
a	1
a	5
b	12
b	23

r

A	B
a	9
a	5
b	12

s

r U s

$r \cup s = \{t \mid t \in r \text{ or } t \in s\}$

A	B
a	1
a	5
b	12
b	23
a	9

r U s

A	B
a	1
a	5
b	12
b	23

r

A	B
a	9
a	5
b	12

s

r ∩ s

Defined as:
$r \cap s = \{t \mid t \in r \text{ and } t \in s\}$

A	B
a	5
b	12

r ∩ s

A	B
a	1
a	5
b	12
b	23

r

A	B
a	9
a	5
b	12

s

r - s

Defined as:
$r - s = \{t \mid t \in r \textbf{ and } t \notin s\}$

A	B
a	1
b	23

r - s

Note: $r \cap s = r - (r - s)$

9. Some queries on bank schema:

i) Find all loans of over Rs.1200

$\sigma_{amount > 1200}(loan)$

ii)Find the loan number for each loan of an amount greater than Rs.1200

$\prod_{loan\_number}(\sigma_{amount > 1200}(loan))$

iii)Find the names of all customers who have a loan, an account, or both, from the bank.

$\prod customer\_name\ (borrower) \cup \prod customer\_name\ (depositor)$

iv) Find the names of all customers who have a loan and an account at bank.

$\prod customer\_name\ (borrower) \cap \prod customer\_name\ (depositor)$

v) Find the information of all customers who live in Gwalior.

vi) Find only the name of those customers who live in Gwalior.

vii) Find the information of those loans that have amount greater than 10000.

viii) Find the loan-number and branch name of those loans that have amount greater than 10000.

ix) Find the account number of those accounts that have balance greater than 10000 and are opened in "morar" branch.

x) Find the name of all depositors.

xi) Find the name of those customers who have only borrowed but not deposited in the bank.

10. Cartesian Product Operation:

Notation r x s; Defined as:

$$r \times s = \{t\ q \mid t \in r \textbf{ and } q \in s\}$$

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

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

r

A	B
a	1
b	2

s

C	D	E
a	10	α
b	10	β
b	20	β
c	10	γ

r x s : Cartesian product/*cross join*

A	B	C	D	E
a	1	a	10	α
a	1	b	10	β
a	1	b	20	β
a	1	c	10	γ
b	2	a	10	α
b	2	b	10	β
b	2	b	20	β
b	2	c	10	γ

$\sigma_{A=C}(r \times s)$

A	B	C	D	E
a	1	a	10	α
b	2	b	10	β
b	2	b	20	β

theta join

11. Rename Operations:

It allows us to name, and therefore to refer to, the results of relational-algebra expressions.

Its also 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

$$\rho_{x(A_1, A_2, \ldots, A_n)} (E)$$

returns the result of expression E under the name X, and with the attributes renamed to *A1 , A2 , …., An* .

12. Natural Join Operation:

Let r and s be relations on schemas R and S respectively.

Then, r⋈ s is a relation on schema $R \cup S$ obtained as follows:

- Consider each pair of tuples tr from r and ts from s.
- If tr and ts have the same value on each of the attributes in $R \cap S$, add a tuple t to the result, where
  - t has the same value as tr on r
  - t has the same value as ts on s

Example:

$R = (A, B, C, D)$

$S = (E, B, D)$

Result schema = (A, B, C, D, E)

$r \bowtie s$ is defined as:

$\prod r.A, r.B, r.C, r.D, s.E\ (\sigma r.B = s.B \wedge r.D = s.D\ (r \times s))$

Example 1:

r

A	B
a	1
b	2

s

A	D	E
a	10	α
b	10	β
b	20	β
c	10	γ

r x s : Cartesian product

r.A	B	s.A	D	E
a	1	a	10	α
a	1	b	10	β
a	1	b	20	β
a	1	c	10	γ
b	2	a	10	α
b	2	b	10	β
b	2	b	20	β
b	2	c	10	γ

Natural Join : $r \bowtie s = \Pi_{r.A, r.B, s.D, s.E} (\sigma_{r.A=s.A}(r \times s))$

A	B	D	E
a	1	10	α
b	2	10	β
b	2	20	β

Example 2:

depositor

Name	Acc#
ram	100
shyam	120
ajay	100

Acccount

Acc#	Balance
100	7000
120	9000

depositor X acccount

Name	dep.Acc#	acc.Acc#	Balance	
ram	100	100	7000	
ram	**100**	**120**	**9000**	X
shyam	**120**	**100**	**7000**	X
shyam	120	120	9000	
ajay	100	100	7000	
ajay	**100**	**120**	**9000**	X

The crossed tuples are meaningless

$(\sigma_{depositor.acc\# = account.acc\#}$(depositor X account)

Name	dep.Acc#	acc.Acc#	Balance
ram	100	100	7000
shyam	120	120	9000
ajay	100	100	7000

depositor $\bowtie$ account =

$\Pi_{name, dep.acc\#, balance}$ $(\sigma_{depositor.acc\# = account.acc\#}$(depositor X account))

Name	dep.Acc#	Balance
ram	100	7000
shyam	120	9000
ajay	100	7000

Some Example:

(i) Find customer name, city and account number.
(ii)Find customer name, loan number and amount.
(iii) Find customer name who have deposited more than one lac.
(iv) Find customer name and city of customers who have deposited more than one lac.

(v) Find the names of all customers who have a loan at the Perryridge branch.

$\Pi_{customer\_name}(\sigma_{branch\_name="Perryridge" \wedge borrower.loan\_number = loan.loan\_number}(borrower \times loan))$

(vi) Find the names of all customers who have a loan at the Perryridge branch but do not have an account at any branch of the bank.

$\Pi_{customer\_name}(\sigma_{borrower.loan\# = loan.loan\# \wedge branch\_name = "Perryridge"}(borrower \times loan)) - \Pi_{customer\_name}(depositor)$

(vii) Find the names of all customers who have a loan at the Perryridge branch.

$\Pi_{customer\_name}(\sigma_{branch\_name = "Perryridge" \wedge borrower.loan\# = loan.loan\#}(borrower \times loan))$

(viii) Find the largest account balance

- Find those balances that are *not* the largest
- Rename *account* relation as *d* so that we can compare each account balance with all others
- Use set difference to find those account balances that were *not* found in the earlier step.

The query is:

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

15. Division Operation:

Notation: $r \div s$

Suited to queries that include the phrase "**for all**".

Let *r* and *s* be relations on schemas *R* and *S* respectively where

$R = (A_1, \ldots, A_m, B_1, \ldots, B_n)$

$S = (B_1, \ldots, B_n)$

that is $S \subseteq R$

The result of r ÷ s is a relation on schema $R - S = (A_1, \ldots, A_m)$

$$r \div s = \{ t \mid t \in \prod_{R-S}(r) \wedge \forall u \in s (t_u \in r) \}$$

Where t_u means the concatenation of tuples *t* and *u* to produce a single tuple.

Example:

r

A	B	C	D	E
α	a	α	a	1
α	a	γ	a	1
α	a	γ	b	1
β	a	γ	a	1
β	a	γ	b	3
γ	a	γ	a	1
γ	a	γ	b	1
γ	a	β	b	1

s

D	E
a	1
b	1

r ÷ s

A	B	C
α	a	γ
γ	a	γ

Some important points about division operations:

- Property
  - Let $q = r \div s$
  - Then *q* is the largest relation satisfying $q \times s \subseteq r$
- Definition in terms of the basic algebra operations

  Let *r(R)* and *s(S)* be relations, and let $S \subseteq R$
  - $r \div s = \prod_{R-S}(r) - \prod_{R-S}((\prod_{R-S}(r) \times s) - \prod_{R-S,S}(r))$

- To see why
  - $\Pi_{R\text{-}S,S}(r)$ simply reorders attributes of r
  - $\Pi_{R\text{-}S}(\Pi_{R\text{-}S}(r) \times s) - \Pi_{R\text{-}S,S}(r))$ gives those tuples t in $\Pi_{R\text{-}S}(r)$ such that for some tuple $u \in s$, $t_u \notin r$.

16. Exercise:

The given relational scheme is:

employee(ename, street, city)
works (ename, cname, salary)
company (cname, city)
manager (ename, mname)

Answer the following queries in relational algebra:

1. Find the names of all employees who works for xyz company.
2. Find names and cities of residence of all employees who works for XYZ company.
3. Find the name of all employees who works for companies located in Gwalior.
4. Find the name and city of all employees who works for companies located in Gwalior.
5. Find names & city of residence of all employees who works for XYZ company & earn more than Rs. 10,000 per month ?
6. Find the name of all employees who live in the same city as the company for which they work.
7. Find the name and city of residence of all managers.
8. Find the name of all managers who work for XYZ company.
9. Find the name and city of all managers who work for XYZ company.
10. Find the name of all employees who are not mangers.
11. Find the name of all employees who works for the managers living in Gwalior.
12. Find the name of all employees who live in the same city as do their manager.
13. Find the name of employee with maximum salary.
14. Find the name of manager with maximum salary.

Note: There are some extended relational algebra operations:

- Generalized Projection
- Aggregate Functions
- Outer Join

17. Generalized Projection:

It extends the projection operation by allowing **arithmetic functions** to be used in the projection list.

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

E is any relational-algebra expression. Each of $F1, F2, \ldots, Fn$ are arithmetic expressions 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:

$$\Pi_{customer\_name,\ limit - creditbalance}(credit\_info)$$

18. Aggregate Functions and Operations

Aggregation function takes a collection of values and returns a single value as a result.

avg: average value
min: minimum value
max: maximum value
sum: sum of values
count: number of values

Aggregate operation in relational algebra

$$_{G_1, G_2, \ldots, G_n}\vartheta_{F_1(A_1), F_2(A_2, \ldots, F_n(A_n))}(E)$$

E is any relational-algebra expression

$G_1, G_2 \ldots, Gn$ is a list of attributes on which to group (can be empty)

Each F_i is an aggregate function
Each A_i is an attribute name

A	B	C
α	α	7
α	β	7
β	β	3
β	β	10

$\mathcal{G}_{\mathbf{sum(c)}}(r)$

sum(c)
27

Example:
Relation *account* grouped by *branch-name*:

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

$branch\_name\ \mathcal{G}\ branch\_name,\ \mathbf{sum}(balance)\ (account)$

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

Result of aggregation does not have a name
- Can use rename operation to give it a name
- For convenience, we permit renaming as part of aggregate operation

$Branch\_name\ \mathcal{G}\ \mathbf{sum}(balance)\ (account)\ \mathbf{as}\ sum\_balance\ (balance)$

Exercises:
1. Find the average balance.
2. Find branch-wise average balance.
3. Find the count of those accounts that have balance greater than Rs.5000
4. Find account number and balance of those accounts that have balance greater than average balance.
5. Find the count of such accounts
6. Find the account number with maximum balance.

19. 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 involving *null* is *null*.
- Aggregate functions simply ignore null values (as in SQL)
- For duplicate elimination and grouping, null is treated like any other value, and two nulls are assumed to be the same (as in SQL)
- *null* signifies that the value is unknown or does not exist
- All comparisons involving *null* are (roughly speaking) **false** by definition.

Comparisons with null values return the special truth value: *unknown*
If *false* was used instead of *unknown*, then *not (A < 5)*
would not be equivalent to *A >= 5*

Three-valued logic using the 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*
In SQL "*P* **is unknown**" evaluates to true if predicate *P* evaluates to *unknown*
Result of select predicate is treated as *false* if it evaluates to *unknown*.

20. Outer Join Operation:

It is an extension of the join operation that avoids loss of information. It 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. It uses *null* values.

Example:

Relation *loan*:

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

Relation *borrower*:

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

(i) Inner Join

loan ⋈ borrower

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

(ii) 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***

(iii) 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***	Hays

(iv) Full / Symmetric 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***	Hays

21. Modification in the Database:

The content of the database may be modified using the following operations:

- Deletion
- Insertion
- Updating

All these operations are expressed using the assignment operator.

22. 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 only whole tuples; it cannot delete values on only particular attributes.
A deletion is expressed in relational algebra by:

$$r \leftarrow r - E$$

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

Examples:

(i)Delete all account records in the Perryridge branch.

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

<complete the query>

(ii)Delete all loan records with amount in the range of 0 to 50

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

<complete the query>

(iii)Delete all accounts at branches located in Gwalior.

$r1 \leftarrow \sigma_{branch\_city = \text{“Gwalior”}} (account \bowtie branch)$

$r2 \leftarrow \prod_{branch\_name,\ account\_number,\ balance} (r1)$

$r3 \leftarrow \prod_{customer\_name,\ account\_number} (r2 \bowtie depositor)$

$depositor \leftarrow depositor - r3$

$account \leftarrow account - r2$

Ex: Delete all customers of Gwalior and their related information.

23. Insertion:

To insert data into a relation, we either:

- specify a tuple to be inserted
- 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.

Examples:

(i)Insert information in the database specifying that Smith has $1200 in account A-973 at the Perryridge branch.

$account \leftarrow account \cup \{(\text{A-973, ”Perryridge”, 1200})\}$

$depositor \leftarrow depositor \cup \{(\text{“Smith”, A-973})\}$

(ii)Provide as a gift for all loan customers in the Perryridge branch, a $200 savings account. Let the loan number serve as the account number for the new savings account.

$r1 \leftarrow (\sigma_{branch\_name = \text{“Perryridge”}} (borrower \bowtie loan))$

$account \leftarrow account \cup \prod_{loan\_number,\ branch\_name,\ 200} (r1)$

$depositor \leftarrow depositor \cup \prod_{customer\_name,\ loan\_number} (r1)$

24. Updation:

It is a mechanism to change a value in a tuple without changing *all* values in the tuple. It uses the generalized projection operator to do this task

$$r \leftarrow \prod_{F_1, F_2, \ldots, F_l} (r)$$

Each *Fi* 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*, which gives the new value for the attribute

Examples:

(i)Make interest payments by increasing all balances by 5 percent.

$$account \leftarrow \prod_{account\_number,\ branch\_name,\ balance\ *\ 1.05} (account)$$

(ii)Pay all accounts with balances over $10,000 a 6 percent interest and pay all others 5 percent

$$account \leftarrow \prod_{account\_number,\ branch\_name,\ balance\ *\ 1.06} (\sigma_{BAL > 10000} (account)) \cup \prod_{account\_number,\ branch\_name,\ balance\ *\ 1.05} (\sigma_{BAL \le 10000} (account))$$

Q The given relation scheme is:

S(s#, sname, status, scity)
P(p#, pname, color)
J (j#, jname, jcity)
SPJ(s # , p# , j# , qty)

Answer the following queries in relational algebra:
(i)Get s# for supplier who supply project for j1
(ii)Find supplier name and city for project J1.
(iii)Get part no and part name for parts supplied by suppliers in Gwalior.
(iv)Get part no for project for items by a supplier in Gwalior to a project in Gwalior
(v)Get p# for parts supplies to project by a supplier in the same city as that project.
(vi)Get part name also in previous query.
(vii)Get project names for projects supplied by supplier, no. s1
(viii)Get colors of part supplied by supplier s1
(ix)Get supplier no. who is supplying maximum qty of part p1
(x)Get J # for projects supplied entirely by supplier S1

Q List two major reasons why we may choose to define a view?

Q List two major problems with processing update operation expressed in terms of views?

Q Let A be a relation of degree n ? How many different projections of A are there?

hint: ${}^nC_1 + {}^nC_2 + {}^nC_3 + \ldots\ldots + {}^nC_n = (2^n - 1)$

Q In ordinary arithmetic there is a special no 1 with has identity property for multiplication. Is there any analogous relation that place the same role in relational algebra?

Q A and B are two relations. Discuss about the candidate key for the following relations :

a) a restriction of relation A (selection)
b) projection of A
c) A times B, A x B
d) A Union B, AU B
e) A intersect B, A Ω B
f) A – B
g) A Join B
h) A divided by B , A ÷ B

Q **person(name, city, fname)**

a) Find the name persons who live in Gwalior.
b) Find the name persons whose father live in Gwalior.
c) Find the name of persons and name their fathers whose fathers live in Gwalior.
d) Find the name of those persons whose grand father live in Gwalior.

Q Consider the schema on page number 7:

a) Find the count of employees in ABC Company.
b) Find the count of employees working under the manger RAM.
c) Find the count of employees for each manager.
d) Find the company with the most employees.
e) Find the company with the smallest payroll.

***

θ-join and equijoin

Consider tables *Car* and *Boat* which list models of cars and boats and their respective prices. Suppose a customer wants to buy a car and a boat, but she doesn't want to spend more money for the boat than for the car. The θ-join on the relation *CarPrice* ≥ *BoatPrice* produces a table with all the possible options.

Car

CarModel	CarPrice
CarA	20'000
CarB	30'000
CarC	50'000

Boat

BoatModel	BoatPrice
Boat1	10'000
Boat2	40'000
Boat3	60'000

CarModel	CarPrice	BoatModel	BoatPric
CarA	20'000	Boat1	10'000
CarB	30'000	Boat1	10'000
CarC	50'000	Boat1	10'000
CarC	50'000	Boat2	40'000

If we want to combine tuples from two relations where the combination condition is not simply the equality of shared attributes then it is convenient to have a more general form of join operator, which is the θ-join (or theta-join). The θ-join is a binary operator that is written as or where *a* and *b* are attribute names, θ is a binary relation in the set {<, ≤, =, >, ≥}, *v* is a value constant, and *R* and *S* are relations. The result of this operation consists of all combinations of tuples in *R* and *S* that satisfy the relation θ. The result of the θ-join is defined only if the headers of *S* and *R* are disjoint, that is, do not contain a common attribute.
The simulation of this operation in the fundamental operations is therefore as follows:

$$R \ {}_{\varphi} \ S = \sigma_{\varphi}(R \times S)$$

In case the operator θ is the equality operator (=) then this join is also called an **equijoin**.
Note, however, that a computer language that supports the natural join and rename operators does not need θ-join as well, as this can be achieved by selection from the result of a natural join (which degenerates to Cartesian product when there are no shared attributes).

Semijoin

The semijoin is joining similar to the natural join and written as $R \ltimes S$ where *R* and *S* are relations. The result of the semijoin is only the set of all tuples in *R* for which there is a tuple in *S* that is equal on their common attribute names. For an example consider the tables *Employee* and *Dept* and their semi join:

Employee

Name	EmpId	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Production

Dept

DeptName	Manager
Sales	Harriet
Production	Charles

Employee Dept

Name	EmpId	DeptName
Sally	2241	Sales
Harriet	2202	Production

More formally the semantics of the semijoin is defined as follows:

$$R \ S = \{ \ t : t \ R, \ s \ S, \ \text{fun} \ (t \ s) \ \}$$

where fun(*r*) is as in the definition of natural join.
The semijoin can be simulated using the natural join as follows. If $a_1, \ldots, a_n$ are the attribute names of *R*, then

$$R \ S = \prod_{a1,\ldots,an}(R \ S).$$

Since we can simulate the natural join with the basic operators it follows that this also holds for the semijoin.

Antijoin

The antijoin, written as $R \triangleright S$ where R and S are relations, is similar to the natural join, but the result of an antijoin is only those tuples in R for which there is NOT a tuple in S that is equal on their common attribute names.
For an example consider the tables *Employee* and *Dept* and their antijoin:

Employee

Name	EmpId	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Production

Dept

DeptName	Manager
Sales	Harriet
Production	Charles

Employee Dept

Name	EmpId	DeptName
Harry	3415	Finance
George	3401	Finance

The antijoin is formally defined as follows:

$$R \triangleright S = \{ t : t \in R \mid \nexists s \in S : \text{fun}(t, s) \}$$

or

$R \triangleright S = \{ t : t \in R$, there is no tuple s of S that satisfies fun $(t, s) \}$

where fun(r) is as in the definition of natural join.
The antijoin can also be defined as the complement of the semijoin, as follows:

$$R \triangleright S = R - R \ltimes S$$

Given this, the antijoin is sometimes called the anti-semijoin, and the antijoin operator is sometimes written as semijoin symbol with a bar above it, instead of $\triangleright$.

person(<u>name</u>, city, fname)

a) Find the name persons who live in Gwalior.
b) Find the name persons whose father live in Gwalior.
c) Find the name of persons and name their fathers whose fathers live in Gwalior.
d) Find the name of those persons whose grand father live in Gwalior.

person

name	**city**	**fname**
ram	gwalior	ajay
shyam	bhopal	vijay
atul	agra	shaildev
ajay	gwalior	sumit
vijay	gwalior	amit
amit	gwalior	aman
sumit	indore	sandeep

father

name	city	fname
ram	gwalior	ajay
shyam	bhopal	vijay
atul	agra	shaildev
ajay	gwalior	sumit
vijay	gwalior	amit
amit	gwalior	aman
sumit	indore	sandeep

Chapter 5
FUNCTIONAL DEPENDENCE

1 Basics:

Functional Dependence :- Let R be a relation and x and y be two arbitrary subset of the set of attributes of R , then we say that y is functionally dependent on x, represented as:

$$X \longrightarrow Y$$

or X functionally determines Y, if **for each value of X in R there is precisely one value of Y associated with it.** Operationally, when two tuples of R agree on their X value, they will also agree on their Y value.

Notes:-

- *Here X is called the determinant and Y is called the dependent*
- *if X → Y, then cardinality from X to Y will be many to one.*
- *if cardinality between X & Y is 1 : 1, then we can say*

$$X \rightarrow Y$$
$$Y \rightarrow X$$

Eg:- Consider a relation

SCP (s#, p#, city, quantity)

where, the supplier with supplier number s who lives in "city", supplies the specified "quantity" of the part with p#.

Instance:-

S#	City	P#	Qty
S1	Gwalior	P1	100
S1	Gwalior	P2	100
S2	Indore	P1	200
S2	Indore	P2	200
S3	Indore	P2	300
S4	Gwalior	P2	400
S4	Gwalior	P4	400
S4	Gwalior	P5	400

Consider following functional dependence

s# → city

It means that, is s# is same in two tuples, then the city will also be the same.

Following are some more f.d.

(s # , p #) → qty.
(s # , p #) → city
(s # , p #) → {city, Qty}
(s # , p #) → s #
(s # , p #) → p #

Notes :-

*(i) **Functional dependence is a property of semantic or meaning associated with the attributes, which depends upon the logic of the application.***

(ii) In fact F.D. specifications are integrity constraints specifications. Because these limit the tuples in a relationship. In other words, a relation cannot have tuples which does not satisfy the functional dependence specified.

(iii) F.D's may be of two types :-

a) Time dependent
b) Time Independent

Time dependent F.D.'s are property of an instance of database; whereas time independent F.D.'s are the property of schema of database.

Hence obviously time independent F.D.'s are of our major interest.

(iv) Functional Dependence & Super Key:-

The notion a concept of functional dependence is the generalization of super key concept. Let X & Y are two sets of attributes such that:

$X \subseteq R \quad \& Y \subseteq R$

The functional dependence $X \rightarrow Y$ *holds on R if for two tuples* t_1 *&* t_2 *in R satisfies the following constraint*

If $t_1[X] = t_2[X]$

then $t_1[Y] = t_2[Y]$

If K in the super key of relation R then we say that

$K \rightarrow R$

Must hold true because for two tuples t_1 *&* t_2

If $t_1[K] = t_2[K]$

then $t_1[R] = t_2[R]$

It means that t_1 *&* t_2 *are duplicate tuples*

Ex: Let a relation be R(A, B, C, D)

A given instance of the relation is:

A	**B**	**C**	**D**
a1	b1	c2	d1
a2	b4	c3	d5
a3	b2	c1	d3

Let the given functional dependence are:

A → B

B → D

Can the following tuples be inserted in the relation:

(a1, b2, c1, d3)

(a2, b4, c4, d7)

(a3, b2, c2, d3)

Ex :- Given the relation R (A,B,C)

Instance:-

A	**B**	**C**
a_1	b_1	c_1
a_1	b_1	c_2
a_2	b_1	c_1
a_2	b_1	c_3

Determine all types of functional dependence in relation

Solution:-

- A → B
- C → B
- {C, A} → B
- {C, A} → R

Hence {C, A} is super key

Note:- *The functional dependence derived with this method is* ***time dependent*** *functional dependence. To find out time independent F.D. many instances of the relation has to be considered.*

2 Trivial & Non- Trivial Functional Dependence

A dependence is trivial if its dependent (R.H.S.) is a sub-set of determinent (L.H.S.). It means that for two sets of attributes A & B of the relation R, if

If $B \subseteq A$

Then A → B is trivial

For eg:- Given A relational R (A,B,C)

The F.D. BC → B is trivial

Normally we are interested non – trivial dependencies naturally

3 Armstrong's Inference Rule:-

. These rules help us to derive more FD's with the help of a given set of FD's. Let there is a relation R(x1, x2, x3, x4) and A and B are the two sets of attributes of R.

Rule 1: Reflexivity :-

If B ⊆ A, Then

A → B

It is trivial FD's & it is called Reflexivity

Rule 2 : Augmentation:-

if A → B

then AC → BC

Note:- AC means {A,C}

Rule 3: Determination:-

A → A

It is again trivial

Rule4: Decomposition :-

if A→ BC

Then A→ B & A→C

Note:- That the decomposition may only be in the dependent not in determinant.

Rule 5:Union:-

If A→ B and A → C

Then A→ BC

Rule 6: Transitivity:

If A → B and B → C

Then A → C

Rule 7: Composition :-

If A→ B & C →D

Then AC→ BD

Rule 8: Pseudo-transitivity rule:

If A→B and BC→ D, then AC→D.

It can be derived also:

if A→B,	then AC→BC (augmentation)
if AC→BC and BC→ D	then AC→D (Transitivity)

4 Closure set of functional dependencies:-

We can derive or deduce some functional dependencies with the help of a given set of FD's. A set of all possible FD's that are implied by a given set S of FD's called the closure of S, & it is represented by S^+,

Definitely $S \subseteq S^+$

For e g

$$S \left\{ \begin{array}{l} A \rightarrow B \\ B \rightarrow C \\ \\ A \rightarrow C \end{array} \right\} S^+$$

Ex:

$R = (A, B, C, G, H, I)$
$F = \{ A \rightarrow B$
$A \rightarrow C$
$CG \rightarrow H$
$CG \rightarrow I$
$B \rightarrow H \}$

Find some members of $F+$

$A \rightarrow H$	by transitivity from $A \rightarrow B$ *and* $B \rightarrow H$
$AG \rightarrow I$	by augmenting $A \rightarrow C$ with G, to get $AG \rightarrow CG$ and then transitivity with $CG \rightarrow I$
$CG \rightarrow HI$	by augmenting $CG \rightarrow I$ to infer $CG \rightarrow CGI$, and augmenting of $CG \rightarrow H$ to infer $CGI \rightarrow HI$, and then transitivity

5 Procedure for Computing F+

$F + = F$
repeat
    for each functional dependency f in $F+$
        apply reflexivity and augmentation rules on f
        add the resulting functional dependencies to $F +$
    for each pair of functional dependencies $f1$and $f2$ in $F +$
        if $f1$ and $f2$ can be combined using transitivity
            then add the resulting functional dependency to $F +$
until $F +$ does not change any further

Ex1 :
Let R = (A, B, C, D) and F = { A → B, A → C, BC → D }. Find whether the dependence A → D is in F^+.

Solution:

Since A→B and A→C	hence, A→BC	union or additivity
Since, A→BC and BC→ D	hence A → D	transitivity

So we can say that the dependence A → D is in F^+.

7. Closure of an Attribute or Attribute Set:

Let there be a relation R with some attributes. The X is an attribute or a set of attribute of R. The closure of X under a set of functional dependencies F, written as X^+, is the set of attributes {A1, A2, A3… Am} such that the FD $X \rightarrow A_i$ for $A_i \in X^+$ follows from F by the inference axioms for functional dependencies.

Algorithm to compute X^+

Let there be a relation R with some attributes. Let X is an attribute or a set of attributes. F is the set of functional dependence given on R. Then to find the closure of X, the following algorithm is applied.

```
X+ = X
repeat
        oldX+ := X+;
        for each functional dependence Y → Z in F do
                if  X+  ⊇ Y then X+ := X+  ∪   Z;
until ( X+ = oldX+)
```

Ex 1:
Let X = BCD and F = { A→BC, CD→E, E→C, D→AEH, ABH→BD, DH→BC}. Compute X+ of X under F.
Sol:
Initialize X+ to X so X+ := BCD.
Since the lhs of FD CD→E is a subset of X+, so X+ := BCDE.
Since thle lhs of FD D→AEH is a subset of X+, so X+ := ABCDEH.
X+ can not be augmented further, so the algorithm ends.
Hence, X+ of X under F is ABCDEH.

Ex 2:
There is a relation R(A, B, C, G, H, I). Let X = AG and
F = { A→B, A→C, CG→HI, B→H}. Compute X+ of X under F.
Sol:
Initialize X+ to X so X+ := AG.
Since the lhs of FD A→B is a subset of X+, so X+ := ABG.
Since the lhs of FD A→C is a subset of X+, so X+ := ABCG.
Since the lhs of FD CG→H is a subset of X+, so X+ := ABCGH.
Since the lhs of FD CG→I is a subset of X+, so X+ := ABCGHI.

X+ cannot be augmented further, so the algorithm ends.
Hence X+ of X under F is ABCGHI.

Ex3:
Given R(ABCDE) and the set of FDs on R given by F = { AB → CD,ABC → E,C → A}. What is X+ if X = ABC? What is the candidate key of R? In what normal form is R?

8. Uses of Attribute Closure

There are several uses of the attribute closure algorithm:
(i) Testing for superkey:

To test if α is a super-key, we compute α+, and check if α+ contains all attributes of *R*, then α is the super key.

(ii) Testing functional dependencies

To check if a functional dependency α → β holds (or, in other words, is in *F*+), just check if β ⊆ α+.

That is, we compute α+ by using attribute closure, and then check if it contains β.

Is a simple test, and very useful

(iii) Computing closure of F

For each γ ⊆ *R*, we find the closure γ+, and for each *S* ⊆ γ+, we output a functional dependency γ → *S*.

9. Testing whether or not an FD is in a closure

To find whether F implies X → Y or not, we simply compute X+ under the FDs set F. If Y ⊆ X+, then we can say that, the dependence X → Y is in F otherwise not.
Following is the algorithm; it uses the algorithm to find the closure of a set of attributes.

```
Compute X+ using the previous algorithm.
If Y ⊆ X+,     then X → Y ∈ F+ := true
               else X → Y ∈ F+ := false
```

Ex 1:

Let F = { A → BC, CD → E, E → C, D → AEH, ABH → BD, DH → C}. Find whether BCD → H is implied by F or not. In other words whether BCD → H is in F+ or not.

Sol:

Initialize X+ to X so X+ := BCD.

Since the lhs of FD CD → Eis a subset of X+, so X+ := BCDE

Since the lhs of FD D → AEH is a subset of X+, so X+ := ABCDEH.

Since, the RHS of FD, BCD → H is H and H ⊆ X+, so we can say that this FD is implied by F or it is in F+.

10. Equivalent Set of FD or Cover:

Given a set of FDs F, F+ is the closure of F and contains all FDs that can be derived from F. F+ can be very large; hence we'll look for a smaller set of FDd that are representative of the closure of F.

Let there be another set of FDs G. We say that F and G are equivalent if the closure of F is identically equal to the closure of G, i.e., F+ = G+.

We can say that F covers G and G covers F.

Exercise 2 on page 11

11. Testing whether an FD is redundant or not

Let there is a relation R, F is the set of FD on it. To find whether a FD X → Y is redundant or not:

- remove the FD X → Y from F; let the remaining set be F' { = F – (X → Y) }
- compute X^+ under the FDs set F'.
- If Y ⊆ X+, then we can say that, the FD X → Y is redundant in F otherwise not.

If an FD is redundant, it should be removed to minimize the set.

Ex: Let F = { BCD → H, A → BC, CD → E, E →C, D→AEH, ABH→BD, DH→C}. Find whether BCD → H is redundant or not.

Sol:

F' = F – (BCD → H) = {A → BC, CD → E, E →C, D→AEH, ABH→BD, DH→C}

Now we find BCD+

BCD+	=	BCD
	=	BCDE
	=	ABCDEH

BCD+ contains H; hence the FD BCD → H can be deduced by the remaining set of FD's; hence it is redundant.

12. Extraneous Attributes

Consider a set *F* of functional dependencies and the functional dependency α → β in *F*.

Attribute A is extraneous in α if $A \in \alpha$ and *F* logically implies

$(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.

Attribute *A* is extraneous in β if $A \in \beta$ and the set of functional dependencies

$(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ logically implies *F*.

Example: Given $F = \{A \rightarrow C, AB \rightarrow C\}$

B is extraneous in $AB \rightarrow C$ because $\{A \rightarrow C, AB \rightarrow C\}$ logically implies $A \rightarrow C$

(I.e. the result of dropping *B* from $AB \rightarrow C$).

Example: Given $F = \{A \rightarrow C, AB \rightarrow CD\}$

C is extraneous in $AB \rightarrow CD$ since $AB \rightarrow C$ can be inferred even after deleting *C*

Testing if an Attribute is Extraneous

Consider a set *F* of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in *F*.

(i) To test if attribute $A \in \alpha$ is extraneous in α

compute $(\{\alpha\} - A)^+$ using the dependencies in *F*

check that $(\{\alpha\} - A)^+$ contains A; if it does, *A* is extraneous

(ii) To test if attribute $A \in \beta$ is extraneous in β

compute α^+ using only the dependencies in

$F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$,

check that α^+ contains *A;* if it does, *A* is extraneous

12. Non-redundant Cover, minimal cover or canonical cover of FD:

There is a relation R with a given set F of functional dependencies. G is called the canonical cover of F if:

- all FD of F are implied by G
- there is no redundant FD in G
- there is no extraneous attribute on the right hand side of all the FD of G
- there is no extraneous attribute on the left hand side of all the FD of G

```
Input: A set of FDs F
Output: A non-redundant cover F

G := F;                    /*initialize G to F */
for each FD X→Y in G do
        if X→Y ∈ { F – ( X→Y) }+          /*{ F – (X → Y) } implies X→Y*/
                then F := { F – ( X→Y) };
[end for]

G := F;                                   /*G is non-redundant cover of F */
remove all extraneous attribute from the determinant;
remove all extraneous attribute from the dependant;

end;
```

The working method is:

- Break all RHS (dependents) to atomic attributes.
- Remove all redundant/extraneous FD from the set:
  - Take an FD X→Y from the set F. Consider F' = { F – (X→Y) }
  - Find X+ based on F'.
  - If $Y \subseteq X+$, then remove X→Y from the set
- Remove all extraneous attributes from the determinants of all FD's.

Ex 1:

If F = { A → BC, CD → E, E → C, D → AEH, ABH → BD, DH → BC}

Find the non-redundant cover of F.

Sol:

Consider the FD, CD→E; we find CD+ from set { F – (CD → E) }. It comes out to be ABCDEH. Since, the RHS E is subset of ABCDEH; hence the FD can be removed.

Now G = { A → BC, E → C, D → AEH, ABH → BD, DH → BC}
Now, consider the FD, DH→BC; we find DH+ from the FD set {G – (DH→BC) }. It comes out to be ABCDEH. Since, the RHS BC is subset of ABCDEH; hence this FD can also be removed.
No other FD can be removed, so the non-redundant cover of F is:

G = { A → BC, E → C, D → AEH, ABH → BD}

Ex2:
Find the canonical cover of
F = { A → BC, CD → E, E → C, D → AEH, ABH → BD, DH → BC}
Sol:

- First, we remove all redundant FD from F.
  Consider the FD, CD→E; we find CD+ from set { F – (CD → E) }. It comes out to be ABCDEH. Since, the RHS E is subset of ABCDEH; hence the FD can be removed.
  Now G = { A → BC, E → C, D → AEH, ABH → BD, DH → BC}
  Now, consider the FD, DH→BC; we find DH+ from the FD set {G – (DH→BC) }. It comes out to be ABCDEH. Since, the RHS BC is subset of ABCDEH; hence this FD can also be removed.
  No other FD can be removed, so the non-redundant cover of F is:

  G = { A → BC, E → C, D → AEH, ABH → BD}
- The FD A → BC can be decomposed into:

  A → B
  A → C
- The FD ABH → BD can be decomposed into

  ABH→B
  ABH→D

  Since, A → B, so both can be reduced to

  AH→B
  AH→D
- But, AH → B is redundant since, A → B is already there.

So the canonical cover is

Fc = { A → B, A → C, E → C, D → A, D → E, D → H, AH → D}

Notes:
1. Functional dependencies (FDs) are used to specify formal measures of the "goodness" of relational designs.
*2. FDs are **constraints** that are derived from the meaning and interrelationships of the data attributes*

13 Non-loss Decomposition:

The process of normalization causes decomposition of the relation. The decomposition must be non-loss type. It means that every information in the original relations must be preserved in the decomposed relations.
The information is in three forms:

- In attributes
- In tuples
- In functional dependence

All the three must be preserved during decomposition.

Let R be a relation. It is decomposed into n different relation, R1, R2, R3, …, Rn (need not be all disjoint). For the decomposition to be non-loss type following conditions must hold true:

(i) Attribute preservation:

To preserve the attributes of the original relation, following condition must hold true:

$$R = R1 \cup R2 \cup R3 \cup \ldots \cup Rn$$

It means that each Ri is a subset of R and every attribute in R appears at least once in Ri.

(ii) Tuple preservation:

Let r be a relation instance on schema R; the decomposed relation instances are r1, r2, …, r_n on the schemes R1, R2, …, Rn respectively, i.e.

$$r_i = \pi_{Ri}(r) \qquad \text{for } i = 1, 2, \ldots, n$$

then the following condition must hold true for preserving all the tuples and to remove spurious tuples:

$$r = r_1 \bowtie r_2 \bowtie r_3 \bowtie \ldots r_n$$

(iii) FD Preservation:

Let Fi is the set of FDs in the decomposed relation scheme Ri. Then to preserve each functional dependence, the following condition must hold true:

$$F_1^+ \cup F_2^+ \cup \ldots \cup F_n^+ = F^+$$

It means that the union of closure sets of FDs of decomposed relations must be equal to the closure set of FD of the original relation.

For an FD to be preserved, the dependant and its determinant must be in the same decomposed relation. (we are talking of canonical cover of the original FD set)

14. Lossless-join Decomposition

This condition is important to avoid generation of spurious tuples on joining the two relations. For the case of $R = (R1, R2)$, we require that for all possible relations r on schema R

$$r = \prod_{R1}(r) \bowtie \prod_{R2}(r)$$

A decomposition of R into $R1$ and $R2$ is lossless join if and only if at least one of the following dependencies is in F^+:

$$R1 \cap R2 \rightarrow R1$$
$$R1 \cap R2 \rightarrow R2$$

Ex:

$R = (A, B, C)$

$F = \{A \rightarrow B, B \rightarrow C\}$

Can be decomposed in two different ways:

(i) $R1 = (A, B)$, $R2 = (B, C)$

Lossless-join decomposition:

$R1 \cap R2 = \{B\}$ and $B \rightarrow BC$

Dependency preserving

(ii) $R1 = (B, C)$, $R2 = (A, C)$

Lossless-join decomposition:

$R1 \cap R2 = \{C\}$ and $C \rightarrow C$

C is neither super key of R1 nor the super key of R2.

Hence, the decomposition is not lossless join.

14. Miscellaneous Points:

1: Informally, each tuple in a relation should represent one entity or relationship instance. (Applies to individual relations and their attributes).

2: Design a schema that does not suffer from the insertion, deletion and update anomalies. If there are any present, then note them so that applications can be made to take them into account.

3: Relations should be designed such that their tuples will have as few NULL values as possible

- Attributes that are NULL frequently could be placed in separate relations (with the primary key)
- Reasons for nulls:
  - attribute not applicable or invalid
  - attribute value unknown (may exist)
  - value known to exist, but unavailable

4. Bad designs for a relational database may result in erroneous results for certain JOIN operations. The "lossless join" property is used to guarantee meaningful results for join operations.
The relations should be designed to satisfy the *lossless join condition*. No spurious tuples should be generated by doing a natural-join of any relations.

5. There are two important properties of decompositions:

(a) non-additive or losslessness of the corresponding join
(b) preservation of the functional dependencies.

Note that property (a) is extremely important and *cannot* be sacrificed. Property (b) is less stringent and may be sacrificed.

Exercises:

1. Suppose we have the following requirements for a university database that is used to keep track of students transcripts:
(a) The university keeps track of each student's name (SNAME), student number (SNUM), social security number (SSSN), current address (SCADDR) and phone (SCPHONE), permanent address (SPADDR) and phone (SPPHONE), birthdate (BDATE), sex (SEX), class (CLASS) (freshman, sophomore, ..., graduate), major department (MAJORDEPTCODE), minor department (MINORDEPTCODE) (if any), and degree program (PROG) (B.A., B.S., ..., Ph.D.). Both ssn and student number have unique values for each student.
(b) Each department is described by a name (DEPTNAME), department code (DEPTCODE), office number (DEPTOFFICE), office phone (DEPTPHONE), and college (DEPTCOLLEGE). Both name and code have unique values for each department.
(c) Each course has a course name (CNAME), description (CDESC), code number (CNUM), number of semester hours (CREDIT), level (LEVEL), and offering department (CDEPT). The value of code number is unique for each course.
(d) Each section has an instructor (INSTUCTORNAME), semester (SEMESTER), year (YEAR), course (SECCOURSE), and section number (SECNUM). Section numbers distinguish different sections of the same course that are taught during the same semester/year; its values are 1, 2, 3, ...; up to the number of sections taught during each semester.
(e) A transcript refers to a student (SSSN), refers to a particular section, and grade (GRADE).

Design an relational database schema for this database application. First show all the functional dependencies that should hold among the attributes. Then, design relation schemas for the database that are each in 3NF or BCNF. Specify the key attributes of each relation. Note any unspecified requirements, and make appropriate assumptions to make the specification complete.

2.

Consider the following two sets of functional dependencies
F= {A ->C, AC ->D, E->AD, E ->H} and
G = {A ->CD, E ->AH}.
Check whether or not they are equivalent.

3.
There is a relation R(A,B,C,D,E). The given FD's are
F = {A→BC, CD→E, B→D, E→A}.

- Find B+.
- List various candidate keys of R.
- Compute the closure of the following set of FD.
- Compute the canonical cover of F.

4.

Consider the universal relation R = {A, B, C, D, E, F, G, H, I, J} and the set of functional dependencies F = {AB→C, A →DE, B→F, F→GH, D→I J} .
What are the keys for R?

5.

Repeat exercise 3 for the following different set of functional dependencies
G = AB→C, BD→ EF, AD→ GH, A→I, H → J.

6.

Given relation R(A,B,C,D,E) with dependencies
AC → C
CD → □E
DE→ B
is AB a candidate key?
is ABD a candidate key?
(Find all candidate keys)

Steps for finding candidate keys:

1. for any set Y to be candidate key, Y+ must contain all the attributes; but the closure of no proper subset of Y can contain all attributes.
2. find all attributes of the given relation.
3. see all dependants, find missing attribute in dependants; let the group is X. Only X or any superset of X may be candidate key.
4. find the closure of X or closure of superset of X.
5. if a set Y is a candidate key, no proper superset can be candidate key (it will be a super key)

Chapter 6

Normalization

1 Definition:

Normalization is to breaking of a relation according to functional dependence.

Need of Normalization: - Some relations may have redundancy in storing the information. The redundancy may cause:

1. Space wastage
2. Possibility of inconsistency

Moreover, we may not be able to maintain some piece of information in un –normalized relations. It means that an un-normalized relation may suffer from various anomalies:

- Insertion anomaly
- Updation anomaly
- Deletion anomaly

Goals of Normalizations:

Let *R* be a relation scheme with a set *F* of functional dependencies. Decide whether a relation scheme *R* is in "good" form. In the case that a relation scheme *R* is not in "good" form, decompose it into a set of relation scheme {*R*1, *R*2, ..., *Rn*} such that

- each relation scheme is in good form
- the decomposition is a lossless-join decomposition
- the decomposition should be dependency preserving.

2. 1st Normal Form

A relation is in 1st NF

a) if the domains of all attributes are scalar or atomic (i.e. non composite) only
b) There are no repeating groups in the tuples.

(a) Example:

Consider a relation:

student(roll-number, name(fname, lname), dob(dd, mm, yy), marks)

The relation is not in 1st NF because the attribute name and date are composite. So it is to be decomposed:

student(roll-number, fname, lname, dd, mm,y y, marks) OR

- ***student(roll-number, marks)***
- ***name(roll-number, fname, lname)***
- ***dob(roll-number, dd, mm, yy)***

(b) Example:

UCTX(course, teacher, text)

A teacher teaches the course and hence all the texts of that course.

Teacher	Course	Text
Prof Ram	Physics	Mechanics
Prof Shyam		Optics
Prof Ram	Mathematics	Algebra
		Calculus

The above relation is not in 1st NF because it is containing repeating groups:

So the normalized relation will look like:

Teacher	Course	Text
Prof Ram	Physics	Mechanics
Prof Ram	Physics	Optics
Prof Shyam	Physics	Mechanics
Prof Shyam	Physics	Optics
Prof Ram	Maths	Algebra
Prof Ram	Maths	Calculus

Note: It cause MVD in the relation.

3. Full Functional Dependence:

It is also called **left irreducible** functional dependence.
Let there is relation scheme R(A, B, C, D, E) and let X and Y are two sub-sets of attributes of R. Let there be a functional dependence:

$X \rightarrow Y$

Let there is a set Z of attributes of R, such that

$Z \subset X$

That is Z is a proper subset of X.

(i) if the dependence

$Z \rightarrow Y$

Holds true for any subset Z of X, then, $X \rightarrow Y$ is called ***partial*** *or left reducible dependence.*

(ii) if the dependence

$Z \rightarrow Y$

Does not holds true for all subsets Z of X, then, $X \rightarrow Y$ is called ***full*** *or left irreducible dependence.*

Example:

Let there is a relation R(A, B, C, D, E).
(i) There is an FD:

$ACD \rightarrow E$

If any of the following FD holds true:

$A \rightarrow E$
$C \rightarrow E$
$D \rightarrow E$
$AC \rightarrow E$
$AD \rightarrow E$
$CD \rightarrow E$

Then the dependence $ACD \rightarrow E$ is *partial or left reducible dependence.*

(ii) There is an FD:

$BD \rightarrow C$

But, none of the following FD's are true:

$B \rightarrow C$
$D \rightarrow C$

Then the dependence $BD \rightarrow C$ is *full or left irreducible dependence.*

Note: If the left hand side i.e. determinant of and FD is atomic, that is a single attribute, the FD will always be full or irreducible type.

4. Transitive Dependence:

Let there is a relation scheme R(A, B, C, D, E). Let X, Y and Z are three arbitrary sub-sets of R (not necessarily be disjoint or distinct).
If the two FD holds true:

$X \rightarrow Y$
$Y \rightarrow Z$

Then transitively:

$X \rightarrow Z$, will also hold true.

5. Prime and Non-prime Attributes:

The attributes that are components of any candidate key of a relation, are called prime attributes. The remaining attributes are called non-prime attributes.
Let there is a relation scheme R(A, B, C, D, E). It has the candidate keys:

AB and BD

Then A, B and D are the prime attributes and C and E are called non-prime attributes.

6. Second Normal Form (2nd NF):-

A relation is in 2nd N.F. iff :

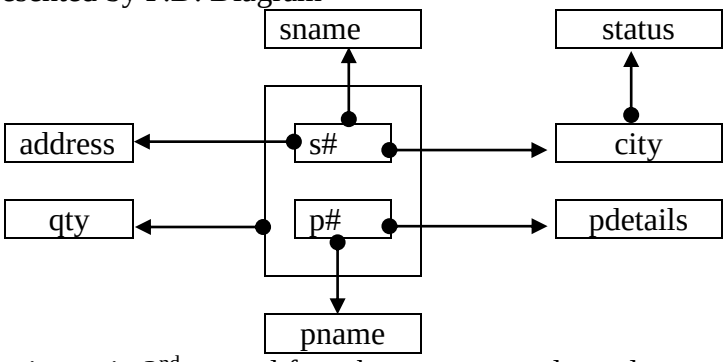
- All attributes are atomic, i.e. it is in 1st NF
- All **non-prime** attribute are fully functionally dependent (or left -irreducibly dependent) on the primary key.

It means that all domains must be atomic; and each functional dependence has the primary key as determinant and it is left irreducible also.

e.g:- SP(s# ,sname, address, p#, pname, pdetails, city, status, qty)

The primary key is **{s#, p#,}**

The F.D. are represented by F.D. Diagram



The above relation is **not** in 2nd normal form because every dependence except

{s#, p#} → qty

is implied by proper subsets of primary key.

Anomalies (Problems) with above relation:- There are two types of problem in the database:

- redundancy
- insertion, deletion and updation anomalies.

s#	Sname	address	p#	pname	pdetails	city	status	qty
s1	Xyz	12, bada	p1	mouse	Logitech	Gwalior	40	15
s1	Xyz	12, bada	p2	CPU	P-III	Gwalior	40	18
s1	xyz	12, bada	p3	keyboard	Chery	Gwalior	40	28
s2	abc	10, Hamidia	p2	CPU	P-III	Bhopal	50	15
s3	pqr	15, chandni	p2	CPU	P-III	Delhi	100	10

1. **Updation Anomaly: -** If we want to modify the address of supplies s1 to "*15, Mora*r", updation has to be done at 3 places. If any one place remains unchanged, it will cause inconsistent database.
2. **Insertion Anomaly: -** We want to insert a tuple of a new supplier say "*s4"* with address *"28, New market, Bhopal",* from which we have not started purchasing yet. We cannot do so because of entity integrity violation. As "*s4*" is not supplying any part, the p# to be null which is not allowed.

3. **Deletion Anomaly:-** If a supplier say s2 stops supplying, that is it is not supplying any part; then its all information has to removed from the database because a part of primary key i.e p# cannot be null due to entity integrity constraint**.**

So the relation has to be broken according to function dependence:

SP21(**s#**, sname, address, city , status)
SP22 (**p#,** pname , pdetails)
SP23 (**s#, p#,** qty)

Note:- Explain how the redundancy & anomaly has been removed.

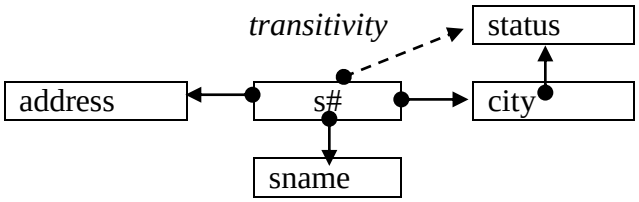
7. Third Normal Form (3rd NF):-

A relation is in 3NF iff:

a) All attributes are atomic, i.e. it is in 1st NF
b) All non-prime attribute are fully functionally dependent (or left -irreducibly dependent) on the primary key, it is in 2nd NF.
c) All non-prime attribute are non-transitively dependent upon the primary key.

e.g.:- supplier(s#, sname, address, city , status)
primary key (s#)

The functional dependencies are represented with the help of figure:



Anomalies (Problems) with above relation:- There are two types of problem in the database:

- redundancy
- insertion, deletion and updation anomalies.

An instance:-

s#	sname	Address	City	status
s1	XYZ	12, Bada	Gwalior	40
s2	ABC	11, Morar	Gwalior	40
s3	PQR	10, Hazira	Gwalior	40
s4	STU	12,DD Nagar	Gwalior	40
s5	AB	12,chandni chowk	Delhi	100

The above relation suffers from various anomalies:-

1. **Updation Anomaly:** Let the status of Gwalior city improves from 40 to 50. The modifications have to be done at 3 places. It is time consuming. Moreover, if any one place is left it causes inconsistency.
2. **Insertion Anomaly:** If we want to store information about a new city in which currently there is no supplier of our interest, we cannot do so because such insertion will cause violation of entity integrity. The s# field will be missing.
3. **Deletion Anomaly:** If we remove supplier s5 from our list the information about city Delhi has to be removed also.

Moreover, the above relation suffers from redundancy, which causes space wastage.
So the above relation has to be decomposed.

S31(**s#,** sname, address, city)
S32(**city** , status)

The about relation do not suffer from various anomaly. Explain, how?

Note: In X → Y and Y → Z, with X as the primary key, we consider this a problem only if Y is not a candidate key. When Y is a candidate key, there is no problem with the transitive dependency .

E.g., Consider EMP (SSN, Emp#, Salary).

Here, SSN → Emp# → Salary and Emp# is a candidate key.

General Normal Form Definitions (For Multiple Keys):

The above definitions consider the primary key only. The following more general definitions take into account relations with multiple candidate keys.

2nd NF: A relation schema R is in **second normal form** (**2NF**) if every non-prime attribute A in R is fully functionally dependent on *every key* of R

3rd NF: A relation schema R is in **third normal form** (**3NF**) if whenever a FD X → A holds in R, then either:

(a) X is a superkey of R, or

(b) ***A is a prime attribute of R (relaxation)***

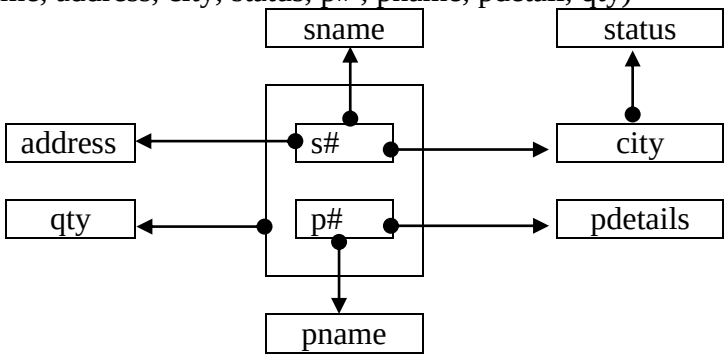
NOTE: Boyce-Codd normal form disallows condition (b) above. So BCNF is stronger as compared to 3rd NF.

8 Boyce-Codd Normal Form (BCNF):-

A relation is in BCNF if and only if every non-trivial left irreducible FD has a candidate key as its determinant.

Note: - A determinant is set of attribute which functionally determines some other attributes, i.e, it is on the left hand side of functional dependence.

e.g.:- SP(s#, sname, address, city, status, p# , pname, pdetail, qty)



Note: *Discuss anomalies with the original relation, i.e, SP and how these are removed in the normalized or decomposed relation.*

In the relation following are the determinants :

s#, p#, (s#, p#), city

So decomposed relation are

BCNF1(**s#,** sname, address, city)
BCNF2(**p#**, pname, pdetails)
BCNF3(**s#, p#,** qty)
BCNF4(**city**, status)

Another definition of BCNF:

A relation schema R is in **Boyce-Codd Normal Form** (**BCNF**) if whenever an FD

X → A

holds in R, then X is a superkey of R

Each normal form is strictly stronger than the previous one

- Every 2NF relation is in 1NF
- Every 3NF relation is in 2NF
- Every BCNF relation is in 3NF

9. BCNF v/s 3rd NF:

There exist relations that are in 3NF but not in BCNF

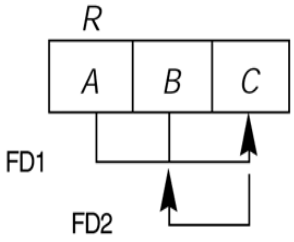
Ex1:

There is a relation

$R = (A, B, C)$

$F = \{AB \rightarrow C, C \rightarrow B\}$

The candidate keys are: *AB, AC*



R is in 3NF, because

AB → *C* *AB* is a superkey

C → *B;* *B* is contained in a candidate key

R is not in BCNF, because

C → *B;* C is not the super key

R is not in BCNF (though in 3rd NF), hence it contains some redundancy

A	B	C
*a*1	*b*1	*c*1
*a*2	*b*1	*c*1
*a*3	*b*1	*c*1
a4	*b*3	*c*2

As we see from the instance, there is:

- repetition of information (e.g., the relationship *b*1, *c*1)

But, any decomposition of R will not be dependency preserving.

There are some situations where:

- BCNF is not dependency preserving, and
- efficient checking for FD violation on updates is important

Solution:

We prefer the weaker normal form, Third Normal Form (3NF), that:

- Allows some redundancy
- But functional dependencies can be checked on individual relations without computing a join.
- There is always a lossless-join, dependency-preserving decomposition into 3NF.

Ex 2: There is a relation TEACHES:

STUDENT	COURSE	INSTRUCTOR
Narayan	Database	Mark
Smith	Database	Navathe
Smith	Operating Systems	Ammar
Smith	Theory	Schulman
Wallace	Database	Mark
Wallace	Operating Systems	Ahamad
Wong	Database	Omiecinski
Zelaya	Database	Navathe

{student, course} is a candidate key for this relation. Two FDs exist in the relation:

fd1: { student, course} → instructor

fd2: instructor → course

The relation is in 3NF but not BCNF.

We try to decompose the above relation in BCNF. Three possible decompositions for relation TEACH:

1. {student, instructor} and {student, course}
2. {course, instructor } and {course, student}
3. {instructor, course } and {instructor, student}

All three decompositions will lose fd1. Moreover, any of the decomposition is not non-loss join decomposition.

Out of the above three, only the 3rd decomposition will not generate spurious tuples after join.(and hence has the non-additivity property).

10. Test

There is a relation R that has the FD set F.

(A)Testing for a relation in BCNF

For all dependencies $\alpha \rightarrow \beta$

- compute α^+ (the attribute closure of α), and
- verify that it includes all attributes of *R*, that is, it is a superkey of *R*.

If the determinant of any of the FD is not the super-key; the relation is not in BCNF.

(B)Testing for a relation in 3rd NF

- Find all candidate keys of R
- For all dependencies $\alpha \rightarrow \beta$, either α should be the super key
  - compute α^+ (the attribute closure of α), and
  - verify that it includes all attributes of *R*, that is, it is a superkey of *R*.
- or β should be a prime attribute.

Note: Hence, to test whether a relation is in 3rd NF is tougher as compared to testing for it to be in BCNF. It is because for 3rd NF, there is a need to find all candidate keys of the relation.

Testing for 3NF has been shown to be NP-hard. Interestingly, decomposition into third normal form (described shortly) can be done in polynomial time.

11. BCNF Decomposition Algorithm

```
result := {R };
done := false;
compute F +;
while (not done) do
        if (there is a schema Ri in result that is not in BCNF)
                then begin
                        let α → β be a nontrivial functional dependency that holds on Ri
                                        such that α → Ri is not in F +, and α ∩ β = ∅;
                          result := (result – Ri ) ∪ (Ri – β) ∪ (α, β );
                end
                else done := true;
end of while;
```

Example:

$R = (A, B, C)$
$F = \{A \rightarrow B, B \rightarrow C\}$
Key = $\{A\}$
R is not in BCNF ($B \rightarrow C$ but B is not superkey)
Decomposition
$R1 = (B, C)$
$R2 = (A,B)$

12. 3NF Decomposition Algorithm

Let *Fc* be a canonical cover for *F;*

```
i := 0;
for each functional dependency α → β in Fc do
        if none of the schemas Rj, 1 ≤ j ≤ i contains α β
                then begin
                                i := i + 1;
                                Ri := α β
                        end
if none of the schemas Rj, 1 ≤ j ≤ i contains a candidate key for R
        then begin
                        i := i + 1;
                        Ri := any candidate key for R;
                end
return (R1, R2, ..., Ri)
```

13. Design Goals:

Goal for a relational database design is:

- BCNF.
- Lossless join.
- Dependency preservation.

But, if BCNF is causing any dependency to be destroyed, then 3rd NF is tolerated.

NOTE: Interestingly, SQL does not provide a direct way of specifying functional dependencies other than superkeys. We can specify FDs using assertions, but they are expensive to test. Even if we had a dependency preserving decomposition, using SQL we would not be able to efficiently test a functional dependency whose left hand side is not a key.

Question: It is said that level of normalization depends upon the semantic associated with the attributes. Justify the statement with the help of an example.

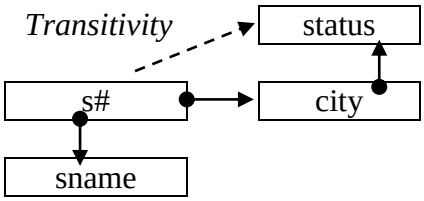
Ans: We take a relation

supplier(s#, sname, city, status)

We associate two different meanings with the attribute status:

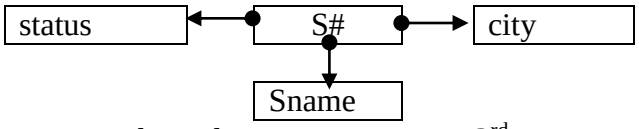
1. status represents the commercial level of the city
2. status represents the credibility status of the supplier.

(1) We associate the semantic with the attribute status that it is functionally determined by city. The FD diagram is:



The relation suffers from transitive dependence of status on primary key s#, hence it is not in 3rd NF.

(2) Now we say that the status is functionally determine by the s# which gives the credibility rating of the supplier. The FD diagram is:



This relation has no transitive dependence. Hence it is in 3rd NF.

Note:

(i) A binary relation is always in 4th NF.

(ii) A dependence that has the determinant which is not a candidate key, will certainly create redundancy and will have insertion, deletion and updation anomalies.

Exercises

1. Consider the universal relation R = {A, B, C, D, E, F, G, H, I} and the set of functional dependencies F = { AB -> C, A -> DE, B -> F, F -> GH, D -> IJ }. What is the key for R? Decompose R into 2NF, then 3NF relations.

2. Repeat exercise 3 for the following different set of functional dependencies
G = { AB -> C, BD -> EF, AD -> GH, A -> I, H -> J }

3. Consider the relation R, which has attributes that hold schedules of courses and sections at a university;

R = {CourseNo, SecNo, OfferingDept, CreditHours, CourseLevel, InstructorSSN, Semester, Year, Days_Hours, RoomNo, NoOfStudents}.

Suppose that the following functional dependencies hold on R:

{CourseNo} -> {OfferingDept, CreditHours, CourseLevel}

{CourseNo, SecNo, Semester, Year} ->{Days_Hours, RoomNo, NoOfStudents, InstructorSSN}

{RoomNo, Days_Hours, Semester, Year} -> {InstructorSSN, CourseNo, SecNo}

Try to determine which sets of attributes form keys of R. How would you normalize this relation?

6.

Consider the following relations for an order-processing application database at ABC, Inc.

ORDER (O#, Odate, Cust#, Total_amount)

ORDER-ITEM (O#, I#, Qty_ordered, Total_price, Discount%)

Assume that each item has a different discount. The Total_price refers to one item, Odate is he date on which the order was placed, and the Total_amount is the amount of the order. If we apply a natural join on the relations Order-Item and Order in this database, what does the resulting relation schema look like? What will be its key? Show the FDs in this resulting relation. Is it in 2NF? Is it in 3NF? Why or why not? (State any assumptions you make.)

7.

Consider the following relation:

CAR_SALE(Car#, Date_sold, Salesman#, Commision%, Discount_amt

Assume that a car may be sold by multiple salesmen and hence {CAR#, SALESMAN#} is the primary key. Additional dependencies are:

Date_sold ->Discount_amt

Salesman# ->commission%

Based on the given primary key, is this relation in 1NF, 2NF, or 3NF? Why or why not? How would you successively normalize it completely?

8.

Consider the following relation for published books:

BOOK (Book_title, Authorname, Book_type, Listprice, Author_affil, Publisher)

Author_affil referes to the affiliation of the author. Suppose the following dependencies exist:

Book_title -> Publisher, Book_type

Book_type -> Listprice

Author_name -> Author-affil

(a) What normal form is the relation in? Explain your answer.

(b) Apply normalization until you cannot decompose the relations further. State the reasons behind each decomposition.

14 Multi-valued Dependence:

Consider a relation:

CTX (course, teacher, text)

Teacher teaches a course; course contains text. A course contains many texts. The teacher teaches all the texts in the course.

An instance of the relation:

Course	Teacher	Text
Physics	Prof. Ram Prof. Shyam	Mechanics Electricity Modern Physics
Maths	Prof. Shyam	Mechanics Algebra Calculus

A teacher who teacher a course will essentially teach all the text of that course. Moreover a teacher can teach more then one course.

The above relation is not in 1st NF because of repeating groups.

The following instance structure will be in 1NF

Course	Teacher	Text
Physics	Prof. Ram	Mechanics
Physics	Prof. Ram	Electricity
Physics	Prof. Ram	Modern Physics
Physics	Prof. Shyam	Mechanics
Physics	Prof. Shyam	Electricity
Physics	Prof. Shyam	Modern Physics
Maths	Prof. Shyam	Mechanics
Maths	Prof. Shyam	Algebra
Maths	Prof. Shyam	Calculus

The primary key of above relation is: **(Course + Teacher + Text)**
The above relation contains **:**

1. No reducible functional dependence.
2. No transitive dependence implied by primary key.

It is simply because it is all key relation; hence the relation is in 3rd NF.
But still it contains redundancy & anomalies.

1. **Redundancy:-**The texts of a course are stored at multiple places.
2. **Updation:** if mechanics is to be modified as basic mechanics, this modification has to be done at more than one place, otherwise it will be an inconsistency. moth
3. **Deletion:** If Prof Shyam leaves the institute, the information about the text of course maths has to be deleted from the database.
4. **Insertion:** If Prof. Mohan Joins to teach maths. The corresponding entry has to be done at there place i.e for all text of maths.

Definition of Multi-Valued Dependence:
Let R be a relation and A, B & C be arbitrary subsets of attributes of R then we say that B is multi-dependent on A (or A multi-determines B),

$$A \twoheadrightarrow B$$

if there is a *set of* B values corresponding to a given A-value plus C-value pair in R depends only on A and is independent of C value.
eg:- In the relation CTX (course, teacher, text)

A is course
B is teacher
C is Text

A course does not have a single corresponding teacher, i.e., the functional dependence COURSE → TEACHER does not hold true. But each course has a well defined set of corresponding teachers. By "*well-defined*" we mean that for a given course *c* and a given text *x*, the set of teachers *t* matching the pair (c, x) in CTX depends upon the value c alone – it makes no difference what particular value of *x* we choose.
Here, we can say that

course $\twoheadrightarrow$ teacher

Note1:- *if for (A, C) pair*

$A \twoheadrightarrow B$

Then for (A, B) pair

$A \twoheadrightarrow C$

will also hold true. So some times the composite muli-valued dependence is represented as

$A \twoheadrightarrow B/C$

eg:- For course, teacher pair, text only depends upon the course not on the teacher. Hence

in the same relation

course $\twoheadrightarrow$ *text*

Is also true. In brief we can say that course multi-determines teacher and text both.

Course $\twoheadrightarrow$ *teacher/text*

***Note 2*:- *Multi-valued dependency is a consequence of 1st NF which does not permit an attribute in a tuple to have a set of values*.**

If we have two or more multi-valued independent attributes in the some relation, we get into a problem of having to repeat every value of one of the attributes with every value of the other attribute to keep the relation state consistent & to maintain the independence among the attributes involved .

Note 3 :- When two independent one to many relationship are mixed in the some relation, a multi-valued dependence arises.

For eg: A : B & A : C has one to many cordiality. Both pairs are mixed in one relation which generates the multi-valued dependence

A $\twoheadrightarrow$ *B/C*

In our example there are two independent cardinalities course : teacher & course : text which are to many & generates multi-value dependence.

Course $\twoheadrightarrow$ *teacher/text*

***Note 4:- For multi-value dependence to hold in a relation, the relation must consist of at least 3 disjoint sets of attributes*.**

Another formal definition of Multi-valued Dependence:

There is a relation scheme R(A, B, C). An MVD, A $\twoheadrightarrow$ B specified on R, imposes the following constraints on any relation state r of R:

if two tuples t_1 and t_2 exit in r such that $t_1[A] = t_2[A]$,

then two tuples t3 and t4 should also exist in r with following properties:

t1[A] = t2[A] = t3[A] = t4[A]

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

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

example: Let a horizontal part(selection) of a relation instance r of R is:

	A	B	C
t_1	a1	b1	c1
t_2	a1	b2	c2

then, the relation will also contain addition tuples t_3 and t_4, there value will be:

		A	B	C
	t_1	a1	b1	c1
	t_2	a1	b2	c2
tuples to be inserted essentially	t_3	a1	b1	c2
	t_4	a1	b2	c1

Whenever, A $\twoheadrightarrow$ B holds, we say that A multi-determines B. Because of the symmetry of the definition, whenever A $\twoheadrightarrow$ B holds in R, so does

A $\rightarrow\!\!\!\!\rightarrow$C. Hence, sometimes it is written as:

A $\rightarrow\!\!\!\!\rightarrow$ B | C

The formal definition specifies that, given a particular value of A, the set of values of B determined by this value of A is completely determined by A alone and does not depend upon the values of the remaining attributes C of R. Hence, whenever two tuples exists with distinct B value but same value of A, these values of B must be repeated in separate tuples with every distinct value of C that occurs with that same value of A.

In multi-valued dependence, the requirement is that if there is a certain tuple in a relation, then for consistency the relation must have additional tuples with similar values. Updates to the database affect these sets of tuples or causes the insertions of more than one tuples. Failure to perform these multiple updates leads to inconsistencies in the database. To avoid these multiple updates, it is preferable to replace undesirable MVD with a number of more desirable relation schemes.

15 Fourth Normal Form (4th NF):

A relation R is in 4th Normal from iff when there exist subset A & B of the attributes of R such that multi-valued dependence

A $\rightarrow\!\!\!\!\rightarrow$B

js satisfied then all attributes of R are also functionally dependent on A.

In other words, **a relation R is in 4NF iff every multi-valued dependence on R is implied by the candidate keys only**.

A relation R is in 4th NF with respect to a set of dependencies F(that includes functional dependencies and multi-valued dependencies) if, for every non-trivial multi-valued dependency A $\rightarrow\!\!\!\!\rightarrow$B in F^+, A is a super key of R.

e.g.:- The relation CTX (course, teacher, text) is not in 4NF because the multi-value dependence

Course $\rightarrow\!\!\!\!\rightarrow$ teacher/text

is there, which is not implied by the candidate key.

The candidate key is (course + teacher + text)

So the relation has to be decomposed.

CT (**course**, **teacher**)

CX (**course**, **text**)

The corresponding instances of database will be:

Course	Teacher
Physics	Prof. Ram
Physics	Prof.Shyam
Maths	Prof.shyam

Course	Text
Physics	Mechanics
Physics	Electricity
Physics	m. physics
Maths	Mechanics
Maths	Algebra
Maths	Calculus

There relation does not suffer from above discussed anomalies.

Alternative Defiition of 4th NF:

A relation schema *R* is in 4NF with respect to a set *D* of functional and multivalued dependencies if for all multivalued dependencies in *D*+ of the form $\alpha \rightarrow\!\!\!\!\rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following hold:

- $\alpha \rightarrow\!\!\!\!\rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$ or $\alpha \cup \beta = R$)
- α is a superkey for schema *R*

If a relation is in 4NF it is in BCNF.

4NF Decomposition Algorithm

```
result: = {R};
done := false;
compute D+;
Let Di denote the restriction of D+ to Ri
  while (not done)
  if (there is a schema Ri in result that is not in 4NF) then
    begin
                let α →→ β be a nontrivial multivalued dependency that holds
      on Ri such that α → Ri  is not in Di, and ϕ=β∩α;
      result :=  (result - Ri) ∪ (Ri - β)  ∪ (α, β);
    end
  else done:= true;
```

Note: each *Ri* is in 4NF, and decomposition is lossless-join

Example:

R =(*A, B, C, G, H, I*)

F = { $A \rightarrow\rightarrow B$
$B \rightarrow\rightarrow HI$
$CG \rightarrow\rightarrow H$ }

R is not in 4NF since $A \rightarrow\rightarrow B$ and *A* is not a superkey for *R*

Decomposition

a) *R1* = (*A, B*) (*R1* is in 4NF)
b) *R*2 = (*A, C, G, H, I*) (*R2* is not in 4NF)
c) *R*3 = (*C, G, H*) (*R*3 is in 4NF)
d) *R4* = (*A, C, G, I*) (*R4* is not in 4NF)
Since $A \rightarrow\rightarrow B$ and $B \rightarrow\rightarrow HI$, $A \rightarrow\rightarrow HI$, $A \rightarrow\rightarrow I$
e) *R5* = (*A, I*) (*R5* is in 4NF)
f)*R6* = (A, C, G) (R6 is in 4NF)

16. Theory of Multivalued Dependencies

Let *D* denote a set of functional and multivalued dependencies. The closure *D*+ of *D* is the set of all functional and multivalued dependencies logically implied by *D*.
Sound and complete inference rules for functional and multivalued dependencies:

1.Reflexivity rule. If α is a set of attributes and $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ holds.
2.Augmentation rule. If $\alpha \rightarrow \beta$ holds and γ is a set of attributes, then $\alpha\gamma \rightarrow \beta\gamma$ holds.
3.Transitivity rule. If $\alpha \rightarrow \beta$ holds and $\beta \rightarrow \gamma$ holds, then $\alpha \rightarrow \gamma$ holds.
4.**Complementation rule.** If $\alpha \twoheadrightarrow \beta$ holds, then $\alpha \twoheadrightarrow R - \beta - \alpha$ holds.
5.**Multivalued augmentation rule.** If $\alpha \twoheadrightarrow \beta$ holds and $\gamma \subseteq R$ and $\delta \subseteq \gamma$, then $\gamma\alpha \twoheadrightarrow \delta\beta$ holds.
6. **Multivalued transitivity rule**. If $\alpha \twoheadrightarrow \beta$ holds and $\beta \twoheadrightarrow \gamma$ holds, then $\alpha \twoheadrightarrow \gamma - \beta$ holds.
7. **Replication rule.** If $\alpha \twoheadrightarrow \beta$ holds, then $\alpha \twoheadrightarrow \beta$.
8. **Coalescence rule.** If $\alpha \twoheadrightarrow \beta$ holds and $\gamma \subseteq \beta$ and there is a δ such that $\delta \subseteq R$ and $\delta \cap \beta = \varnothing$ and $\delta \twoheadrightarrow \gamma$, then $\alpha \twoheadrightarrow \gamma$ holds.

Derived Rules:

We can simplify the computation of the closure of *D* by using the following rules (proved using rules 1-8).

Multivalued union rule. If $\alpha \twoheadrightarrow \beta$ holds and $\alpha \twoheadrightarrow \gamma$ holds, then $\alpha \twoheadrightarrow \gamma\beta$ holds.

Intersection rule. If $\alpha \twoheadrightarrow \beta$ holds and $\alpha \twoheadrightarrow \gamma$ holds, then $\alpha \twoheadrightarrow \beta \cap \gamma$ holds.

Difference rule. If $\alpha \twoheadrightarrow \beta$ holds and $\alpha \twoheadrightarrow \gamma$ holds, then $\alpha \twoheadrightarrow \beta - \gamma$ holds and $\alpha \twoheadrightarrow \gamma - \beta$ holds.

Example:

R = (*A, B, C, G, H, I)*

D = {*A* $\twoheadrightarrow$ *B*
B $\twoheadrightarrow$ *HI*
CG $\twoheadrightarrow$ *H*}

Some members of *D*+:

A $\twoheadrightarrow$ *CGHI.*

Since *A* $\twoheadrightarrow$ *B,* the complementation rule (4) implies that *A* $\twoheadrightarrow$ *R – B – A.*

Since *R – B – A = CGHI,* so *A* $\twoheadrightarrow$ *CGHI.*

A $\twoheadrightarrow$ *HI.*

Since *A* $\twoheadrightarrow$ *B* and *B* $\twoheadrightarrow$ *HI,* the multivalued transitivity rule (6) implies that *B* $\twoheadrightarrow$ *HI – B.*

Since *HI – B = HI, A* $\twoheadrightarrow$ *HI.*

B $\twoheadrightarrow$ *H.*

Apply the coalescence rule (8); *B* $\twoheadrightarrow$ *HI* holds.

Since *H* $\subseteq$ *HI* and *CG* $\twoheadrightarrow$ *H* and *CG* $\cap$ *HI* = Ø, the coalescence rule is satisfied with α being *B,* β being *HI,* δ being *CG,* and γ being *H*. We conclude that *B* $\twoheadrightarrow$ *H.*

A $\twoheadrightarrow$ CG.

A $\twoheadrightarrow$ CG*HI* and *A* $\twoheadrightarrow$ *HI.*

By the difference rule, *A* $\twoheadrightarrow$ CG*HI* – *HI.*

Since CG*HI* – *HI = CG, A* $\twoheadrightarrow$ CG.

17 JOIN DEPENDENCE

Consider a relation:

SPJ (S# , P#, J#)

It is a supplier, part & project relation. A supplier can supply any number of parts for any number of projects. Same part can be supplied by many suppliers & a part can be used in many projects. Supplier of a project may be more then one.

An instance of the data base

S#	P#	J#
S1	P1	J2
S1	P2	J1
S2	P1	J1
S1	P1	J1

The p.k. of relation (S#, P#, J#)

We take 3 different projection on this relation:

1. SP = $\prod_{S\#,P\#}$ (SPJ)

S#	P#
S1	P1
S1	P2
S2	P1

2. PJ = $\prod_{P\#,J\#}$ (SPJ)

P#	J#
P1	J2
P2	J1
P1	J1

3. JS= $\prod_{J\#,S\#}$ (SPJ)

J#	S#
J2	S1
J1	S1
J1	S2

Join two relations SP and PJ

SPJ1 = SP ⋈ PJ (on p#)

S#	P#	J#
S1	P1	J2
S1	P1	J1
S1	P2	J1
S2	P1	J2
S2	P1	J1

Extra or spurious tuple. it is not in original relation

So we do not get the same relation, i.e, SPJ1 ≠ SPJ.

we further perform a join operation:

SPJ2 = SPJI ⋈ PJ { on (S#, J#) }

S#	P#	J#
S1	P1	J2
S1	P1	J1
S1	P2	J1
S2	P1	J1

now, SPJ2 = SPJ

so, this relation suffers from join dependence.

Definition of Join Dependence:-

Join dependencies constraint the set of legal relations over a schema *R* to those relations for which a given decomposition is a lossless-join decomposition.

Let *R* be a relation schema and *R1* , *R2* ,..., *Rn* be a decomposition of *R*.

If *R* = *R1* ∪ *R2* ∪ .... ∪ *Rn*, we say that a relation *r*(*R*) satisfies the *join dependency* *(*R1* , *R2* ,..., *Rn*) if:

$$r = \prod_{R1}(r) \bowtie \prod_{R2}(r) \bowtie \ldots\ldots \bowtie \prod_{Rn}(r)$$

A join dependency is *trivial* if one of the *Ri* is *R* itself.

A join dependency *(*R1*, *R2*) is equivalent to the multivalued dependency *R1* ∩ *R2* ↠ *R2*.

Conversely, α ↠ β is equivalent to *(α ∪(*R* - β) , α ∪ β)

However, there are join dependencies that are not equivalent to any multivalued dependency.

18 Definition of 5th NF

It is called *projection join normal form* (PJNF). A relation R is in PJNF iff every join-dependence in R is implied by the candidate key of R only.

A relation schema R is in PJNF with respect to a set D of functional, multivalued, and join dependencies if for all join dependencies in D^+ of the form

$*(R1, R2, ..., Rn)$ where each $Ri \subseteq R$

and $R = R1 \cup R2 \cup ... \cup Rn$

at least one of the following holds:

- $*(R1, R2, ..., Rn)$ is a trivial join dependency.
- Every Ri is a superkey for R.

Since every multivalued dependency is also a join dependency, every PJNF schema is also in 4NF.

Fifth normal form deals with cases where information can be reconstructed from smaller pieces of information that can be maintained with less redundancy.

We may say that a record type is in fifth normal form when its information content **cannot** be reconstructed from several smaller record types, i.e., from record types each having fewer fields than the original record. The case where all the smaller records have the same key is excluded. If a record type can only be decomposed into smaller records which all have the same key, then the record type is considered to be in fifth normal form without decomposition.

Example:

Consider *Loan-info-schema* = (*branch-name, customer-name, loan-number, amount*). Each loan has one or more customers, is in one or more branches and has a loan amount; these relationships are independent, hence we have the join dependency

*(=(*loan-number, branch-name*), (*loan-number, customer-name*), (*loan-number, amount*))

Loan-info-schema is not in PJNF with respect to the set of dependencies containing the above join dependency. To put *Loan-info-schema* into PJNF, we must decompose it into the three schemas specified by the join dependency:

(*loan-number, branch-name*)
(*loan-number, customer-name*)
(*loan-number, amount*)

19. Domain-Key Normal Form (DKNF) (6th NF)

Domain declaration. Let A be an attribute, and let **dom** be a set of values. The domain declaration $A \subseteq$ **dom** requires that the A value of all tuples be values in **dom**.

Key declaration. Let R be a relation schema with $K \subseteq R$. The key declaration **key** (K) requires that K be a superkey for schema R ($K \rightarrow R$). All key declarations are functional dependencies but not all functional dependencies are key declarations.

General constraint. A general constraint is a predicate on the set of all relations on a given schema.

Let **D** be a set of domain constraints and let **K** be a set of key constraints for a relation schema R. Let **G** denote the general constraints for R. Schema R is in DKNF if **D** $\cup$ **K** logically imply **G**.

Example:

Accounts whose *account-number* begins with the digit 9 are special high-interest accounts with a minimum balance of 2500.

General constraint: ``If the first digit of t [*account-number*] is 9, then t [*balance*] $\geq$ 2500."

DKNF design:

Regular-acct-schema = (*branch-name, account-number, balance*)

Special-acct-schema = (*branch-name, account-number, balance*)

Domain constraints for {*Special-acct-schema*} require that for each account:

The account number begins with 9.

The balance is greater than 2500.

20. ER Model and Normalization

When an E-R diagram is carefully designed, identifying all entities correctly, the tables generated from the E-R diagram should not need further normalization. However, in a real (imperfect) design, there can be functional dependencies from non-key attributes of an entity to other attributes of the entity

Example: an *employee* entity with attributes *department_number* and *department_address*, and a functional dependency

$$department\_number \rightarrow department\_address$$

Good design would have made department an entity

Functional dependencies from non-key attributes of a relationship set possible, but rare --- most relationships are binary

21. Denormalization for Performance

We ay want to use non-normalized schema for performance. For example, displaying *customer_name* along with *account_number* and *balance* requires join of *account* with *depositor*

Alternative 1: Use denormalized relation containing attributes of *account* as well as *depositor* with all above attributes:

- faster lookup
- extra space and extra execution time for updates
- extra coding work for programmer and possibility of error in extra code

Alternative 2:use a materialized view defined as

$$account \bowtie depositor$$

Benefits and drawbacks same as above, except no extra coding work for programmer and avoids possible errors

: **Exercise** :

1. There is a relation R(A,B,C,D,E) with FD set F = {A→BC, CD→E, B→D, E→A}.
   - The relation is decomposed into R1(A, B, C) and R2(A, D, E). Show that the decomposition is lossless join type.
   - The relation is decomposed into R1(A, B, C) and R2(C, D, E). Show that the decomposition is lossless join type.

2. There is a relation R(A,B,C,D,E) with FD set F = {A→B, BC→E, ED→A}
   - List all keys of R.
   - Is R in 3NF.
   - Is R in BCNF.

***

Chapter 9
Transactions

1. Transaction Concept

A ***transaction*** is a ***unit*** of program-execution that accesses and possibly updates various data items. A transaction must see a consistent database. During transaction execution, the database may be inconsistent. When the transaction is committed, the database must be consistent. Two main issues to deal with:

- ★ Failures of various kinds, such as hardware failures and system crashes
- ★ Concurrent execution of multiple transactions

2. 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.

Isolation. 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.

Example of Fund Transfer

Transaction to transfer Rs.50 from account *A* to account *B*:

1. **read**(*A*)
2. *A* := *A* – 50
3. **write**(*A*)
4. **read**(*B*)
5. *B* := *B* + 50
6. **write**(*B*)

Consistency requirement – the sum of *A* and *B* is unchanged by the execution of the transaction.

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

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.

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).

3. Transaction State

Active, the initial state; the transaction stays in this state while it is executing

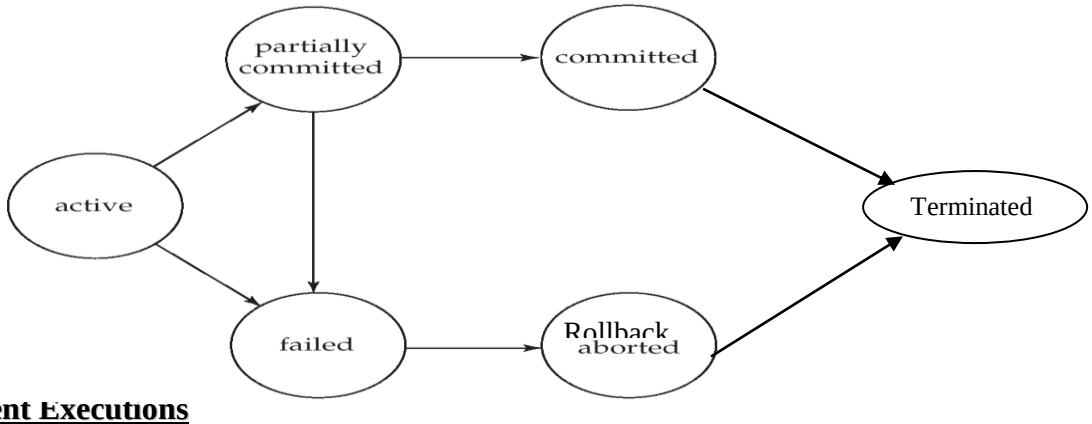
Partially committed, after the final statement has been executed.

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:

- ★ restart the transaction – only if no internal logical error
- ★ kill the transaction

Committed, after *successful completion.*



4. Concurrent Executions

Multiple transactions are allowed to run concurrently in the system. Advantages are:

- ★ **increased processor and disk utilization**, leading to better transaction ***throughput****:* one transaction can be using the CPU while another is reading from or writing to the disk.
- ★ **reduced average response time** for transactions: short transactions need not wait behind long ones.

Concurrency control schemes – It is mechanisms to achieve isolation, i.e., to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

5. Schedules : It is a group of related transactions.

Schedules – sequences that indicate 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
- ★ must preserve the order in which the instructions appear in each individual transaction.

Example 1: Let T_1 transfers Rs.5000/- from *A* to *B*, and T_2 transfers Rs.4000/- from *A* to *C*,. The following is a serial schedule, in which T_1 is followed by T_2.

Initially
A = 20000
B= 10000
C= 5000

T1 (A→ B)	T2 (A → C)
read(*A*) *A* := *A* – 5000 **write(*A*)** **read**(*B*) *B* := *B* + 5000 **write**(*B*)	
	read(*A*) *A* := *A* – 4000 write(A) **read**(*C*) C := *C* + 4000 **write**(C)

Example 2: Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule.

A = 20000
B= 10000
C= 5000

T1 (A→ B)	T2 (A → C)
read(*A*) *A* := *A* – 5000 **write(*A*)**	
	read(*A*) *A* := *A* – 4000 write(A)
read(*B*) *B* := *B* + 5000 **write**(*B*)	
	read(*C*) C := *C* + 4000 **write**(C)

Example3:The following concurrent schedule does not preserve the consistency of the database.

A = 20000
B= 10000
C= 5000

T1 (A→ B)	T2 (A → C)
read(*A*) *A* := *A* – 5000	
	read(*A*) *A* := *A* – 4000 write(A)
write(*A*) **read**(*B*) *B* := *B* + 5000 **write**(*B*)	
	read(*C*) C := *C* + 4000 **write**(C)

6. Serializability

Basic Assumption – Each transaction preserves database consistency. Thus 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:

1. conflict serializability 2. view serializability

We ignore operations other than **read** and **write** instructions, and we assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes. Our simplified schedules consist of only **read** and **write** instructions.

7. Conflict Serializability

Instructions l_i and l_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both l_i and l_j, and at least one of these instructions wrote Q.

1. l_i = **read**(Q), l_j = **read**(Q). l_i and l_j don't conflict.
2. l_i = **read**(Q), l_j = **write**(Q). They conflict.
3. l_i = **write**(Q), l_j = **read**(Q). They conflict
4. l_i = **write**(Q), l_j = **write**(Q). They conflict

Intuitively, a conflict between l_i and l_j forces a (logical) temporal order between them. If l_i and l_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 of a schedule that is not conflict serializable:

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

We are unable to swap instructions in the above schedule to obtain either the serial schedule <T_3, T_4 >, or the serial schedule < T_4, T_3 >

Schedule 3 below can be transformed into Schedule 1, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable.

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

⇒

T1	**T2**
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

8. 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 that value was produced by transaction T_j (if any), then transaction T_i must in schedule S' also read the value of Q that was produced by transaction T_j .
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'.

As can be seen, view equivalence is also based purely on **reads** and **writes** alone. A schedule S is **view serializable** it is view equivalent to a serial schedule. Every conflict serializable schedule is also view serializable.

The schedule given below is view-serializable but *not* conflict serializable. Every view serializable schedule that is not conflict serializable has **blind writes(?).**

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

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

9. Recoverability (see the jpg file transactionRecover.jpg)

Recoverable schedule — if a transaction T_j reads a data items previously written by a transaction T_i , the commit operation of T_i must appear before the commit operation of T_j.

The following schedule is not recoverable if T_9 commits immediately after the read

T8	T9
read(A);	
A = A-500;	
write(A)	
	read(A)
	B = A*2
	write(B)
	<commit>
read(C)	
<commit>	

The schedule is conflict serializable. But, suppose, T8 have some consistency error during read(C); it will have to rollback, making the changes in A undone. But, since T9 has committed earlier, so it cannot be rolled back. It means that, the value of B is calculated on that value of A, which does not exist.

Hence, the commit of T9 must be after commit of T8.

The schedule is conflict serializable but, not recoverable. Hence database must ensure that schedules are recoverable.

Cascading rollback – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

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

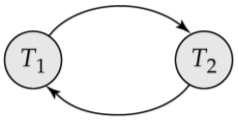
If T_{10} fails, T_{11} and T_{12} must also be rolled back. It can lead to the undoing of a significant amount of work.

Cascadeless schedules — 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. It is desirable to restrict the schedules to those that are cascadeless

Implementation of Isolation
Schedules must be conflict or view serializable, and recoverable, for the sake of database consistency, and preferably cascadeless. A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency. Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur. Some schemes allow only conflict-serializable schedules to be generated, while others allow view-serializable schedules that are not conflict-serializable.

10. Testing for Serializability

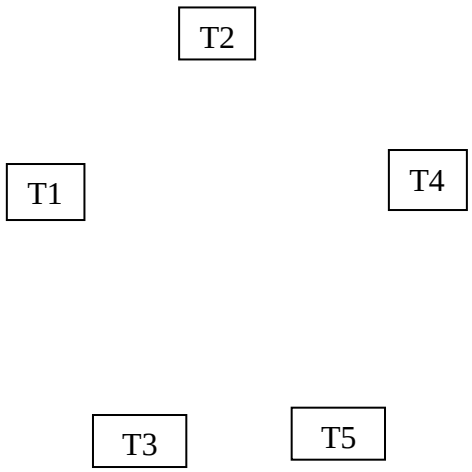
Consider some schedule of a set of transactions T_1, T_2, ..., T_n
Precedence graph — a direct graph where the vertices are the transactions (names). We draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier. We may label the arc by the item that was accessed.



Example

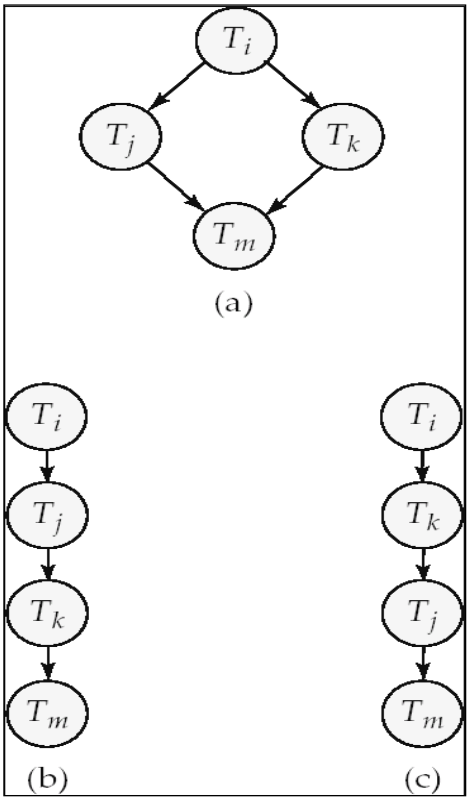
T1	T2	T3	T4	T5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				

The precedence graph:



11. Test for Conflict Serializability

A schedule is conflict serializable if and only if its precedence graph is acyclic. Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph. (Better algorithms take order $n + e$ where e is the number of edges.) If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph. This is a linear order consistent with the partial order of the graph. For example, a serializability order for Schedule A would be: $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$.



12. Test for View Serializability

The precedence graph test for conflict serializability must be modified to apply to a 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 unlikely. However practical algorithms that just check some *sufficient conditions* for view serializability can still be used.

***

Chapter 12
Recovery System

1. Failure Classification

Following may be the possible failures:

(A)Transaction failure :

- **Logical errors**: Transaction cannot complete due to some internal error condition
- **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock)

(B) System crash:

A power failure or other hardware or software failure causes the system to crash.

- **Fail-stop assumption**: non-volatile storage contents are assumed to not be corrupted by system crash. Database systems have numerous integrity checks to prevent corruption of disk data.

(C) Disk failure:

A head crash or similar disk failure destroys all or part of disk storage. Destruction is assumed to be detectable, disk drives use checksums to detect failures.

2. Recovery Algorithms

Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures. Recovery algorithms have two parts:

- Actions taken during normal transaction processing to ensure enough information exists to recover from failures – log files
- Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

3. Storage Structure

There are three types of storages:

(A) Volatile storage: It does not survive system crashes; Examples: main memory, cache memory

(B) Nonvolatile storage: It survives system crashes; examples: disk, tape, flash memory, non-volatile (battery backed up) RAM

(C) Stable storage: It is a mythical form of storage that survives all failures; it is **approximated** by maintaining **multiple copies on distinct nonvolatile media**

4. Stable-Storage Implementation

A stable storage maintains multiple copies of each block on separate disks. The copies can be at remote sites to protect against disasters such as fire or flooding.

Failure during data transfer can still result in inconsistent copies. Block transfer can result in:

- Successful completion
- Partial failure: destination block has incorrect information
- Total failure: destination block was never updated

5. Data Access

Physical blocks are those blocks residing on the disk. Buffer blocks are the blocks residing temporarily in main memory. Block movements between disk and main memory are initiated through the following two operations:

- **input**(*B*) transfers the physical block *B* to main memory.
- **output**(*B*) transfers the buffer block *B* to the disk, and replaces the appropriate physical block there.

Each transaction *Ti* has its private work-area in which local copies of all data items accessed and updated by it are kept. *Ti*'s local copy of a data item *X* is called *xi*. We assume, for simplicity, that each data item fits in, and is stored inside, a single block. Transaction transfers

data items between system buffer blocks and its private work-area using the following operations :

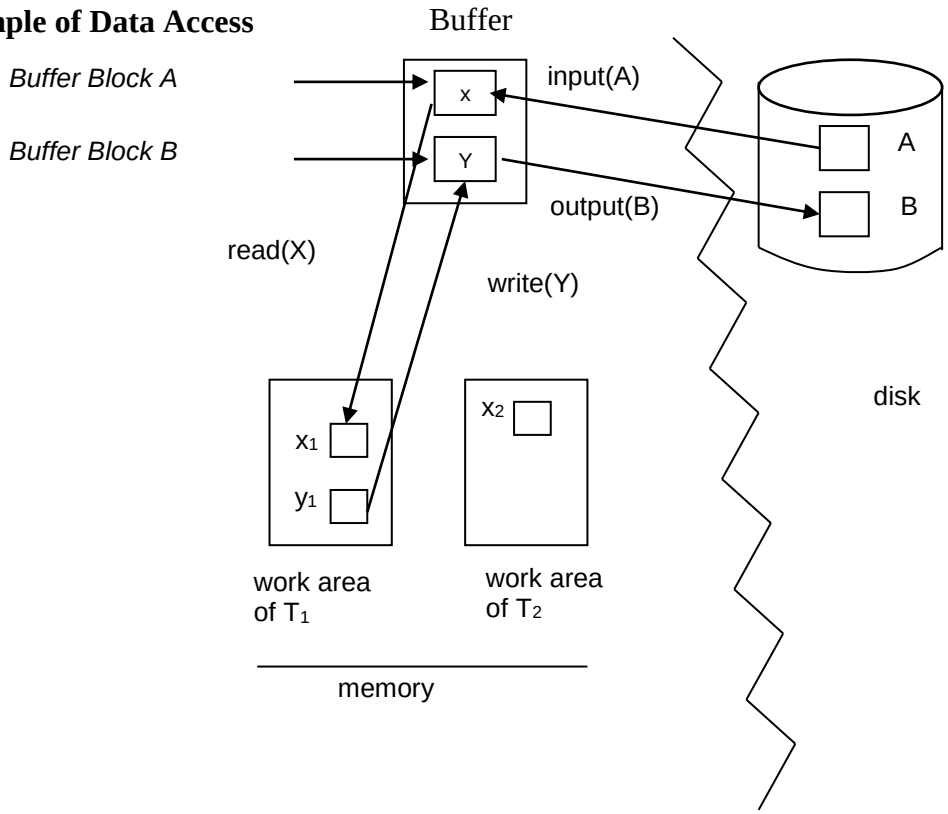
- **read**(X) assigns the value of data item X to the local variable xi.
- **write**(X) assigns the value of local variable xi to data item {X} in the buffer block.
- both these commands may necessitate the issue of an **input**(B_X) instruction before the assignment, if the block BX in which X resides is not already in memory.

In case of transactions processing:

- Perform **read**(X) while accessing X for the first time;
- All subsequent accesses are to the local copy.
- After last access, transaction executes **write**(X).

So, **output**(B_X) need not immediately follow **write**(X). System can perform the **output** operation when it deems fit.

Example of Data Access



6. Recovery and Atomicity

Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state.

Consider transaction Ti that transfers \$50 from account A to account B; goal is either to perform all database modifications made by Ti or none at all. Several output operations may be required for Ti (to output A and B). A failure may occur after one of these modifications have been made but before all of them are made. To ensure atomicity despite failures, we first output information describing the modifications to **stable storage** without modifying the database itself.

We study two approaches:

- **log-based recovery**, and
- **shadow-paging**

We assume (initially) that transactions run serially, that is, one after the other.

7. 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 Ti starts, it registers itself by writing a <Ti **start**>log record. ***Before*** Ti executes **write**(X), a log record <Ti, X, $V1$, $V2$> is written, where $V1$ is the value of X before the write, and $V2$ is the value to be written to X. Log record

notes that *Ti* has performed a write on data item *Xj* *Xj* had value *V1* before the write, and will have value *V2* after the write. When *Ti* finishes it last statement, the log record <*Ti* **commi**t> is written. **We assume for now that log records are written directly to stable storage (that is, they are not buffered).**

Two approaches using logs

- **(I).** Deferred database modification
- **(II).** Immediate database modification

(I) Deferred Database Modification

The **deferred database modification** scheme records all modifications to the log, but defers (***delay***) all the **write**s to **database** ~~**after**~~ **(during) partial commit**. Assume that transactions execute serially. Transaction starts by writing <*Ti* ***start***> record to log. A **write**(*X*) operation results in a log record <*Ti*, *X*, *V*> being written, where *V* is the new value for *X*.

Note: old value is not needed for this scheme

The write is not performed on *X* at this time, but is deferred. When *Ti* partially commits, <*Ti* **commit**> is written to the log. Finally, the log records are read and used to actually execute the previously deferred writes.

During recovery after a crash, a transaction needs to be redone if and only if both <*Ti* **start**> and<*Ti* **commit**> are there in the log. Redoing a transaction *Ti* (**redo***Ti*) sets the value of all data items updated by the transaction to the new values.

Crashes can occur while

- the transaction is executing the original updates, or
- while recovery action is being taken

Example: transactions *T0* and *T1* (*T0* executes before *T1*):

T0:	**T1**
read (A)	**read** (C)
A = A - 50	C = C- 100
Write (A)	**write** (C)
read (B)	
B = B + 50	
write (B)	

Below we show the log as it appears at three instances of time.

<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 950>	<T_0, A, 950>	<T_0, A, 950>
<T_0, B, 2050>	<T_0, B, 2050>	<T_0, B, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 600>	<T_1, C, 600>
		<T_1 commit>
(a)	(b)	(c)

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 **commi**t> is present

(c) **redo**(*T*0) must be performed followed by redo(*T*1) since
<*T*0 **commit**> and <*Ti* commit> are present

(II) 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. This scheme can be extended to postpone log record output, so long as prior to execution of an **output**(*B*) operation for a data block B, all log records corresponding to items *B* must be flushed to stable storage.
Output of updated blocks can take place at any time before or after transaction commit. Order in which blocks are output can be different from the order in which they are written.

Immediate Database Modification Example

Log	Write	Output
<*T*0 **start**>		
<*T0*, A, 1000, 950>		
*T*o, B, 2000, 2050		
	A = 950	
	B = 2050	
<*T*0 **commit**>		
<*T*1 **start**>		
<*T*1, C, 700, 600>		
	C = 600	
		B_B, B_C
<*T*1 **commit**>		
		B_A

Note: B_X denotes block containing X.

Recovery procedure has two operations instead of one:

- **undo**(*T*i) restores the value of all data items updated by *Ti* to their old values, going backwards from the last log record for *Ti*.
- **redo**(*T*i) sets the value of all data items updated by *Ti* to the new values, going forward from the first log record for *Ti*.

Both operations must be ***idempotent*, i.e.** even if the operation is executed multiple times the effect is the same as if it is executed once. It is needed since operations may get re-executed during recovery
When recovering after failure:

- Transaction *Ti* needs to be undone if the log contains the record <*Ti* **start**>, but does not contain the record <*Ti* **commit**>.
- Transaction *Ti* needs to be redone if the log contains both the record <*Ti* **start**> and the record <*Ti* **commit**>.

Undo operations are performed first, then redo operations.

Immediate DB Modification Recovery Example

Below we show the log as it appears at three instances of time.

<T_0 start>	<T_0 start>	<T_0 start>
<T_0, *A*, 1000, 950>	<T_0, *A*, 1000, 950>	<T_0, *A*, 1000, 950>
<T_0, *B*, 2000, 2050>	<T_0, *B*, 2000, 2050>	<T_0, *B*, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, *C*, 700, 600>	<T_1, *C*, 700, 600>
		<T_1 commit>
(a)	(b)	(c)

Recovery actions in each case above are:
(a) undo (*T*0): B is restored to 2000 and A to 1000.
(b) undo (*T*1) and redo (*T*0): C is restored to 700, and then *A* and *B* are set to 950 and 2050 respectively.
(c) redo (*T*0) and redo (*T*1): A and B are set to 950 and 2050 respectively. Then *C* is set to 600

8. Checkpoints

Problems in recovery procedure as discussed earlier:

- searching the entire log is time-consuming
- we might unnecessarily redo transactions which have already output their updates to the database.

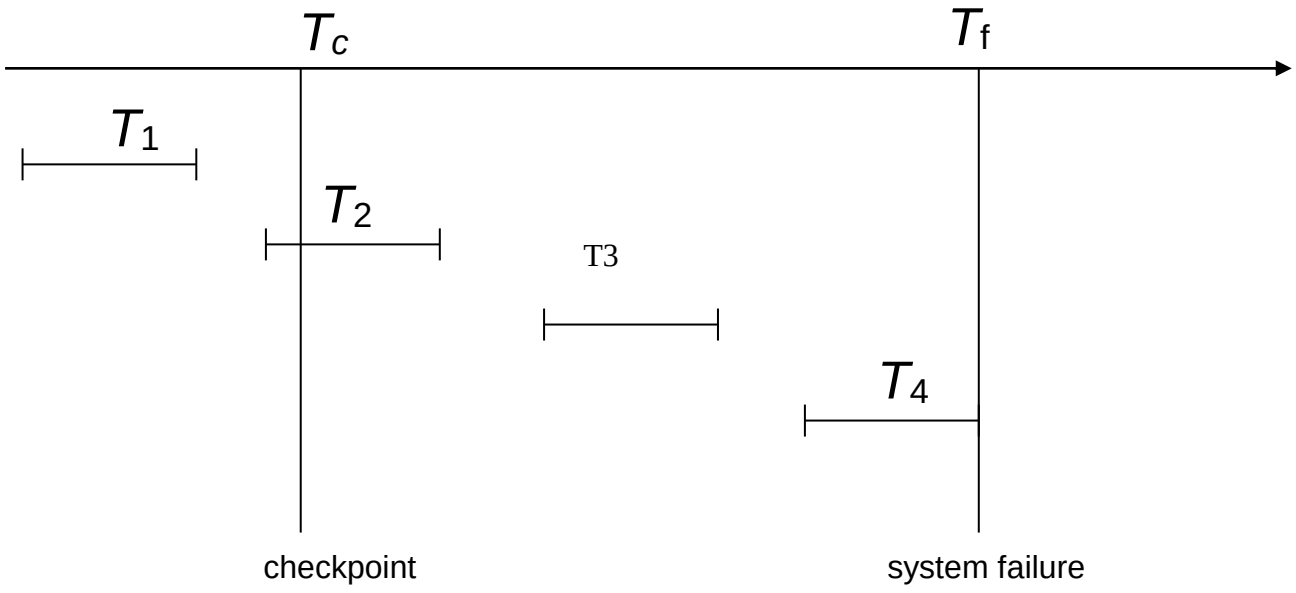
Streamline recovery procedure by periodically performing **checkpointing**

- Output all log records currently residing in main memory onto stable storage.
- Output all modified buffer blocks to the disk.
- Write a log record < **checkpoint**> onto stable storage.

During recovery we need to consider only the most recent transaction Ti that started before the checkpoint, and transactions that started after *Ti*.

- Scan backwards from end of log to find the most recent <**checkpoint**> record
- Continue scanning backwards till a record <*Ti* **start**> is found.
- Need only consider the part of log following above **star**t record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.
- For all transactions (starting from *Ti* or later) with no <*Ti* **commit**>, execute **undo*(Ti)***. (Done only in case of immediate modification.)
- Scanning forward in the log, for all transactions starting from *Ti* or later with a <*Ti* **commit**>, execute **redo*(Ti)***.

Example of Checkpoints



- *T*1 can be ignored (updates already output to disk due to checkpoint)
- *T*2 and *T*3 redone.
- *T*4 undone

9. Shadow Paging:

Shadow paging is an alternative to log-based recovery; this scheme is useful if transactions execute serially. The idea is - maintain *two* page tables during the lifetime of a transaction:

- the **current page table**, and
- the **shadow page table**

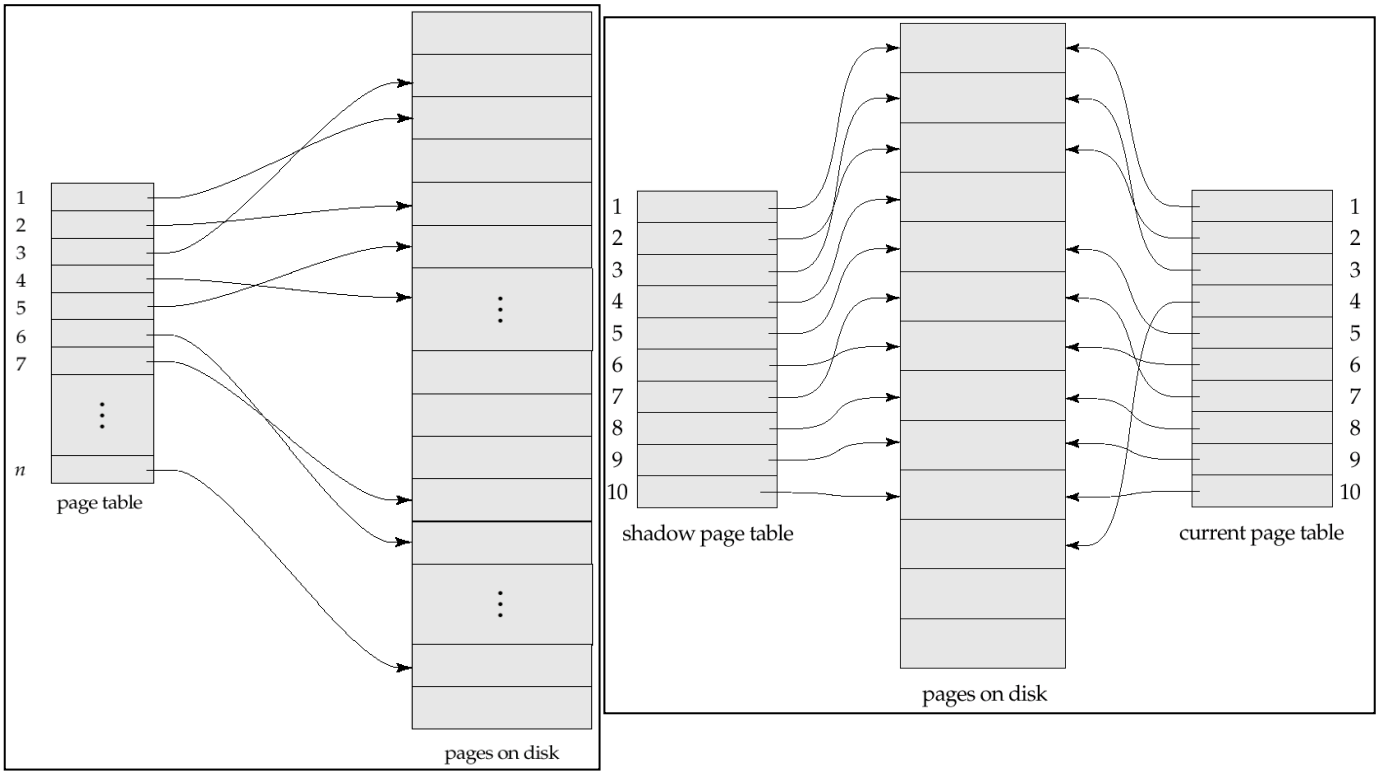
Store the shadow page table in nonvolatile storage, such that state of the database prior to transaction execution may be recovered. Shadow page table is never modified during execution.

To start with, both the page tables are identical. Only current page table is used for data item accesses during execution of the transaction.

Whenever any page is about to be written for the first time:

- A copy of this page is made onto an unused page.
- The current page table is then made to point to the copy
- The update is performed on the copy

Sample Page Table



Example of Shadow Paging

To commit a transaction :

- Flush all modified pages in main memory to disk.
- Output current page table to disk.
- Make the current page table the new shadow page table, as follows:
  - ⋆ keep a pointer to the shadow page table at a fixed (known) location on disk.
  - ⋆ to make the current page table the new shadow page table, simply update the pointer to point to current page table on disk

Once pointer to shadow page table has been written, transaction is committed. No recovery is needed after a crash — new transactions can start right away, using the shadow page table. Pages not pointed to from current/shadow page table should be freed (garbage collected).

Advantages of shadow-paging over log-based schemes:

- no overhead of writing log records
- recovery is trivial

Disadvantages :

- Copying the entire page table is very expensive. (Can be reduced by using a page table structured like a B+-tree No need to copy entire tree, only need to copy paths in the tree that lead to updated leaf nodes)
- Commit overhead is high even with above extension
- Need to flush every updated page, and page table
- Data gets fragmented (related pages get separated on disk)
- After every transaction completion, the database pages containing old versions of modified data need to be garbage collected
- Hard to extend algorithm to allow transactions to run concurrently (Easier to extend log based schemes)

10. Log Record Buffering

Log records are buffered in main memory, instead of being output directly to stable storage. Log records are output to stable storage when a block of log records in the buffer is full, or a **log force** operation is executed. Log force is performed to commit a transaction by forcing all its log records (including the commit record) to stable storage. Several log records can thus be output using a single output operation, reducing the I/O cost.

The rules below must be followed if log records are buffered:

- Log records are output to stable storage in the order in which they are created.
- Transaction *Ti* enters the commit state only when the log record <*Ti* **commit**> has been output to stable storage.
- Before a block of data in main memory is output to the database, all log records pertaining to data in that block must have been output to stable storage. This rule is called the **write-ahead logging or WAL** rule. Strictly speaking WAL only requires undo information to be output.

11. Database Buffering

Database maintains an in-memory buffer of data blocks. When a new block is needed, if buffer is full an existing block needs to be removed from buffer. If the block chosen for removal has been updated, it must be output to disk. As a result of the write-ahead logging rule, if a block with uncommitted updates is output to disk, log records with undo information for the updates are output to the log on stable storage first. No updates should be in progress on a block when it is output to disk. It can be ensured as follows:

- Before writing a data item, transaction acquires exclusive lock on block containing the data item
- Lock can be released once the write is completed. Such locks held for short duration are called ***latches***.
- Before a block is output to disk, the system acquires an exclusive latch on the block. Ensures no update can be in progress on the block

Database buffer can be implemented either in an area of real main-memory reserved for the database, or in virtual memory. Implementing buffer in reserved main-memory has drawbacks:

- Memory is partitioned before-hand between database buffer and applications, limiting flexibility.
- Needs may change, and although operating system knows best how memory should be divided up at any time, it cannot change the partitioning of memory.

Database buffers are generally implemented in virtual memory in spite of some drawbacks:

- When operating system needs to evict a page that has been modified, to make space for another page, the page is written to swap space on disk.
- When database decides to write buffer page to disk, buffer page may be in swap space, and may have to be read from swap space on disk and output to the database on disk, resulting in extra I/O. It is known as ***dual paging** problem*.

- Ideally when swapping out a database buffer page, operating system should pass control to database, which in turn outputs page to database instead of to swap space (making sure to output log records first). Dual paging can thus be avoided, but common operating systems do not support such functionality.

12. Failure with Loss of Nonvolatile Storage

So far we assumed no loss of non-volatile storage. Technique similar to checkpointing used to deal with loss of non-volatile storage.

- Periodically **dump** the entire content of the database to stable storage.
- No transaction may be active during the dump procedure; a procedure similar to checkpointing must take place:
  - Output all log records currently residing in main memory onto stable storage.
  - Output all buffer blocks onto the disk.
  - Copy the contents of the database to stable storage.
  - Output a record <**dump**> to log on stable storage.
- To recover from disk failure:
  - Restore database from most recent dump.
  - Consult the log and redo all transactions that committed after the dump

It can be extended to allow transactions to be active during dump; known as **fuzzy dump** or **online dump.**

***

Chapter 16
Relational Calculus

1. Tuple Relational Calculus

It is a nonprocedural query language, where each query is of the form

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

It is the set of all tuples t such that predicate P is true for t

t is a *tuple variable*, $t[A]$ denotes the value of tuple t on attribute A

$t \in r$ denotes that tuple t is in relation r

P is a *formula* similar to that of the predicate calculus

Predicate Calculus Formula It has:

- Set of attributes and constants
- Set of comparison operators: (e.g., $<, \leq, =, \neq, >, \geq$)
- Set of connectives: and ($\wedge$), or (v), not ($\neg$)
- Implication ($\Rightarrow$): x $\Rightarrow$ y, if x is true, then y is true

  $x \Rightarrow y \equiv \neg x \text{ v } y$
- Set of quantifiers:
  - $\exists t \in r\ (Q(t)) \equiv$ "there exists" a tuple in t in relation r such that predicate $Q(t)$ is true
  - $\forall t \in r\ (Q(t)) \equiv Q$ is true "for all" tuples t in relation r

2. Examples

Consider the Bank-schema:

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*)

Queries based on the schema:

(i) Find the *loan_number, branch_name*, and *amount* for loans of over Rs.1200

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

(ii) Find the loan number for each loan of an amount greater than Rs.1200

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

Notice that a relation on schema [*loan_number*] is implicitly defined by the query.

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

$$\{t \mid \exists s \in borrower\ (t[customer\_name] = s[customer\_name])$$
$$\vee \exists u \in depositor\ (t[customer\_name] = u[customer\_name])$$

(iv) Find the names of all customers who have a loan and an account at the bank.

$$\{t \mid \exists s \in borrower\ (t[customer\_name] = s[customer\_name])$$
$$\wedge \exists u \in depositor\ (t[customer\_name] = u[customer\_name])$$

(v) Find the names of all customers having a loan at the Perryridge branch

$$\{t \mid \exists s \in borrower\ (t[customer\_name] = s[customer\_name]$$
$$\wedge \exists u \in loan\ (u[branch\_name] = \text{"Perryridge"}$$
$$\wedge\ u[loan\_number] = s[loan\_number]))\}$$

(vi) Find the names of all customers who have a loan at the Perryridge branch, but no account at any branch of the bank.

$\{t \mid \exists s \in borrower\ (t[customer\_name] = s[\text{customer\_name}]$
$\wedge \exists u \in loan\ (u[branch\_name] = \text{“Perryridge”}$
$\wedge\ u[loan\_number] = s[\text{loan\_}number]))$
$\wedge$ **not** $\exists v \in depositor\ (t[customer\_name] =$
$v[customer\_name])\}$

(vii) Find the names of all customers having a loan from the Perryridge branch, and the cities in which they live.

$\{t \mid \exists s \in loan\ (s[branch\_name] = \text{“Perryridge”}$
$\wedge \exists u \in borrower\ (u[loan\_number] = s[loan\_number]$
$\wedge\ t[customer\_name] = u[customer\_name])$
$\wedge \exists\ v \in customer\ (u[customer\_name] = v[customer\_name]$
$\wedge\ t[customer\_city] = v[customer\_city]))\}$

(viii) Find the names of all customers who have an account at all branches located in Brooklyn:

$\{t \mid \exists\ r \in \mathbf{depositor}\ (t[customer\_name] = r[customer\_name]) \wedge$
$(\forall\ u \in branch\ (u[branch\_city] = \text{“Brooklyn”} \Rightarrow$
$\exists\ s \in depositor\ (r[customer\_name] = s[customer\_name]$
$\wedge \exists\ w \in account\ (\ w[account\_number] = s[account\_number]$
$\wedge\ (\ w[branch\_name] = u[branch\_name])))\}$

$\{\mathbf{t \mid \exists\ r \in depositor\ (t[customer\_name] = r[customer\_name]) \wedge}$
$\mathbf{(\forall\ u \in branch\ (u[branch\_city] = \text{“Brooklyn”}}$
$\mathbf{\wedge \exists\ w \in account\ (\ w[account\_number] = r[account\_number]}$
$\mathbf{\wedge\ (\ w[branch\_name] = u[branch\_name])\ ))\}}$

3. Safety of Expressions

It is possible to write tuple calculus expressions that generate infinite relations. For example,

$\{\ t \mid \neg\ t \in r\ \}$

results in an infinite relation if the domain of any attribute of relation r is infinite. To guard against the problem, we restrict the set of allowable expressions to safe expressions. An expression

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

in the tuple relational calculus is *safe* if every component of t appears in one of the relations, tuples, or constants that appear in P.

NOTE: this is more than just a syntax condition. For example:

$\{\ t \mid t[A] = 5 \vee$ **true** $\}$

is not safe --- it defines an infinite set with attribute values that do not appear in any relation or tuples or constants in P.

4. Domain Relational Calculus

It is a nonprocedural query language equivalent in power to the tuple relational calculus. Each query is an expression of the form:

$\{ < x1, x2, \ldots, xn > \mid P\ (x1, x2, \ldots, xn)\}$

where

$x1, x2, \ldots, xn$ represent domain variables.

P represents a formula similar to that of the predicate calculus.

5. Example Queries

Consider the same schema of Banking Enterprise.

(i) Find the *loan_number*, *branch_name*, and *amount* for loans of over Rs.1200

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

(ii) Find the names of all customers who have a loan of over $1200

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

(iii) Find the names of all customers and the loan amount who have a loan from the Perryridge branch:

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

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

(iv) Find the names of all customers having a loan, an account, or both at the Perryridge branch:

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

(v) Find the names of all customers who have an account at all branches located in Brooklyn:

$$\{< c > \mid \exists\, s,n\, (< c, s, n > \in \text{customer}) \wedge \forall\, x,y,z\, (< x, y, z > \in branch \wedge y = \text{“Brooklyn”})$$
$$\Rightarrow \exists\, a,b\, (<a,b,l> \in account \wedge < c,a > \in depositor)\}$$

6. Safety of Expressions

The expression:

$$\{ < x1, x2, \ldots, xn > \mid P\, (x1, x2, \ldots, xn\,)\}$$

is safe if all of the following hold:

1. All values that appear in tuples of the expression are values from *dom* (*P*) (that is, the values appear either in *P* or in a tuple of a relation mentioned in *P*).
2. For every “there exists” subformula of the form $\exists\, x\, (P1(x\,))$, the subformula is true if and only if there is a value of *x* in *dom* (*P*1) such that *P*1(*x*) is true.
3. For every “for all” subformula of the form $\forall$x (*P*1 (*x*)), the subformula is true if and only if *P*1(*x*) is true for all values *x* from *dom* (*P*1).