SIR C R REDDY COLLEGE OF ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Eluru-534007, Andhra Pradesh State, INDIA.
(Affiliated to JNTUK, Kakinada - Approved by AICTE - Accredited by NAAC)

STUDY MATERIAL

DATABASE MANAGEMENT SYSTEMS

II B.TECH II YEAR

[JNTU-K R19 REGULATION]

Prepared by
M. Ganesh Babu
Assistant Professor
Dept. CSE

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY: KAKINADA
KAKINADA – 533 003, Andhra Pradesh, India

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

II Year – II Semester		L	T	P	C
		3	1	0	4
DATABASE MANAGEMENT SYSTEMS					

Course Objectives:

- To introduce about database management systems
- To give a good formal foundation on the relational model of data and usage of Relational Algebra
- To introduce the concepts of basic SQL as a universal Database language
- To demonstrate the principles behind systematic database design approaches by covering conceptual design, logical design through normalization
- To provide an overview of physical design of a database system, by discussing Database indexing techniques and storage techniques

Course Outcomes:

By the end of the course, the student will be able to

- Describe a relational database and object-oriented database
- Create, maintain and manipulate a relational database using SQL
- Describe ER model and normalization for database design
- Examine issues in data storage and query processing and can formulate appropriate solutions
- Outline the role and issues in management of data such as efficiency, privacy, security, ethical responsibility, and strategic advantage

UNIT I

Introduction: Database system, Characteristics (Database Vs File System), Database Users(Actors on Scene, Workers behind the scene), Advantages of Database systems, Database applications. Brief introduction of different Data Models; Concepts of Schema, Instance and data independence; Three tier schema architecture for data independence; Database system structure, environment, Centralized and Client Server architecture for the database.

UNIT II

Relational Model: Introduction to relational model, concepts of domain, attribute, tuple, relation, importance of null values, constraints (Domain, Key constraints, integrity constraints) and their importance BASIC SQL: Simple Database schema, data types, table definitions (create, alter), different DML operations (insert, delete, update), basic SQL querying (select and project) using where clause, arithmetic & logical operations, SQL functions(Date and Time, Numeric, String conversion).

UNIT III

Entity Relationship Model: Introduction, Representation of entities, attributes, entity set, relationship, relationship set, constraints, sub classes, super class, inheritance, specialization, generalization using ER Diagrams. SQL: Creating tables with relationship, implementation of key and integrity constraints, nested queries, sub queries, grouping, aggregation, ordering, implementation of different types of joins, view(updatable and non-updatable), relational set operations.

UNIT IV

Schema Refinement (Normalization): Purpose of Normalization or schema refinement, concept of functional dependency, normal forms based on functional dependency(1NF, 2NF and 3 NF), concept of surrogate key, Boyce-codd normal form(BCNF), Lossless join and dependency preserving decomposition, Fourth normal form(4NF), Fifth Normal Form (5NF).

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY: KAKINADA
KAKINADA – 533 003, Andhra Pradesh, India

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

UNIT V

Transaction Concept: Transaction State, Implementation of Atomicity and Durability, Concurrent Executions, Serializability, Recoverability, Implementation of Isolation, Testing for Serializability, Failure Classification, Storage, Recovery and Atomicity, Recovery algorithm.

Indexing Techniques: B+ Trees: Search, Insert, Delete algorithms, File Organization and Indexing, Cluster Indexes, Primary and Secondary Indexes , Index data Structures, Hash Based Indexing: Tree base Indexing ,Comparison of File Organizations, Indexes and Performance Tuning

Text Books:

1) Database Management Systems, 3/e, Raghurama Krishnan, Johannes Gehrke, TMH
2) Database System Concepts,5/e, Silberschatz, Korth, TMH

Reference Books:

1) Introduction to Database Systems, 8/e C J Date, PEA.
2) Database Management System, 6/e Ramez Elmasri, Shamkant B. Navathe, PEA
3) Database Principles Fundamentals of Design Implementation and Management, Corlos Coronel, Steven Morris, Peter Robb, Cengage Learning.

e-Resources:

1) https://nptel.ac.in/courses/106/105/106105175/
2) https://www.geeksforgeeks.org/introduction-to-nosql/

UNIT - 1

INTRODUCTION TO DBMS

Database is a collection of related data and data is a collection of facts and figures that can be processed to produce information.

Mostly data represents recordable facts. Data aids in producing information, which is based on facts. For example, if we have data about marks obtained by all students, we can then conclude about toppers and average marks.

A ***database management system*** stores data in such a way that it becomes easier to retrieve, manipulate, and produce information.

CHARACTERISTICS OF DBMS

Traditionally, data was organized in file formats. DBMS was a new concept then, and all the research was done to make it overcome the deficiencies in traditional style of data management. A modern DBMS has the following characteristics −

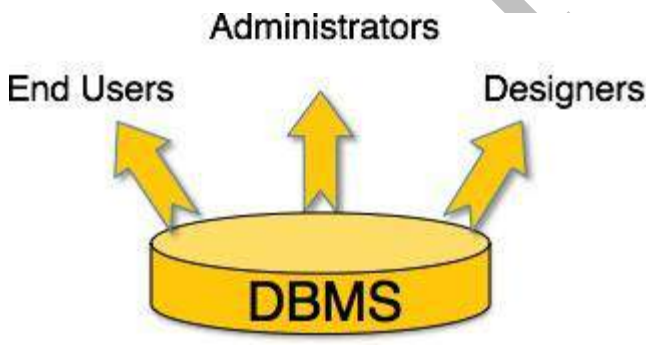
- **Real-world entity** − A modern DBMS is more realistic and uses real-world entities to design its architecture. It uses the behavior and attributes too. For example, a school database may use students as an entity and their age as an attribute.
- **Relation-based tables** − DBMS allows entities and relations among them to form tables. A user can understand the architecture of a database just by looking at the table names.
- **Isolation of data and application** − A database system is entirely different than its data. A database is an active entity, whereas data is said to be passive, on which the database works and organizes. DBMS also stores metadata, which is data about data, to ease its own process.
- **Less redundancy** − DBMS follows the rules of normalization, which splits a relation when any of its attributes is having redundancy in values. Normalization is a mathematically rich and scientific process that reduces data redundancy.
- **Consistency** − Consistency is a state where every relation in a database remains consistent. There exist methods and techniques, which can detect attempt of leaving database in inconsistent state. A DBMS can provide greater consistency as compared to earlier forms of data storing applications like file-processing systems.
- **Query Language** − DBMS is equipped with query language, which makes it more efficient to retrieve and manipulate data. A user can apply as many and as different filtering options as required to retrieve a set of data. Traditionally it was not possible where file-processing system was used.
- **ACID Properties** − DBMS follows the concepts of **A**tomicity, **C**onsistency, **I**solation, and **D**urability (normally shortened as ACID). These concepts are applied on transactions, which manipulate data in a database. ACID properties help the database stay healthy in multi-transactional environments and in case of failure.
- **Multiuser and Concurrent Access** − DBMS supports multi-user environment and allows them to access and manipulate data in parallel. Though there are restrictions on

transactions when users attempt to handle the same data item, but users are always unaware of them.

- **Multiple views** − DBMS offers multiple views for different users. A user who is in the Sales department will have a different view of database than a person working in the Production department. This feature enables the users to have a concentrate view of the database according to their requirements.
- **Security** − Features like multiple views offer security to some extent where users are unable to access data of other users and departments. DBMS offers methods to impose constraints while entering data into the database and retrieving the same at a later stage. DBMS offers many different levels of security features, which enables multiple users to have different views with different features. For example, a user in the Sales department cannot see the data that belongs to the Purchase department. Additionally, it can also be managed how much data of the Sales department should be displayed to the user. Since a DBMS is not saved on the disk as traditional file systems, it is very hard for miscreants to break the code.

DATA BASE USERS

A typical DBMS has users with different rights and permissions who use it for different purposes. Some users retrieve data and some back it up. The users of a DBMS can be broadly categorized as follows −



- **Administrators** − Administrators maintain the DBMS and are responsible for administrating the database. They are responsible to look after its usage and by whom it should be used. They create access profiles for users and apply limitations to maintain isolation and force security. Administrators also look after DBMS resources like system license, required tools, and other software and hardware related maintenance.
- **Designers** − Designers are the group of people who actually work on the designing part of the database. They keep a close watch on what data should be kept and in what format. They identify and design the whole set of entities, relations, constraints, and views.
- **End Users** − End users are those who actually reap the benefits of having a DBMS. End users can range from simple viewers who pay attention to the logs or market rates to sophisticated users such as business analysts.

DBMS ARCHITECTURE

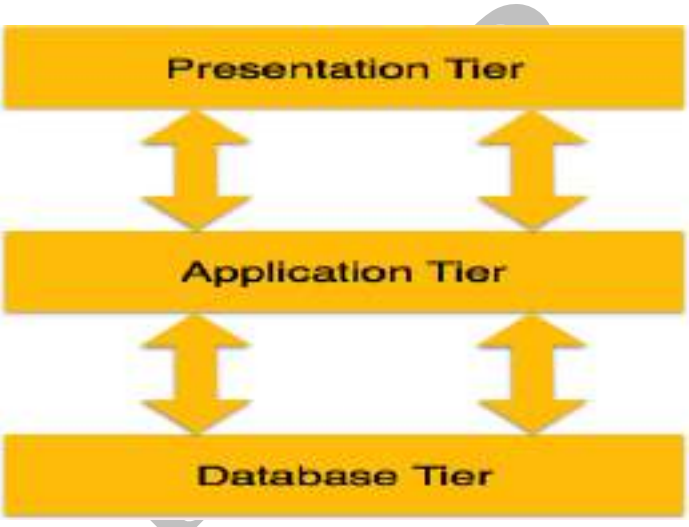
The design of a DBMS depends on its architecture. It can be centralized or decentralized or hierarchical. The architecture of a DBMS can be seen as either single tier or multi-tier. An n-tier architecture divides the whole system into related but independent **n** modules, which can be independently modified, altered, changed, or replaced.

In 1-tier architecture, the DBMS is the only entity where the user directly sits on the DBMS and uses it. Any changes done here will directly be done on the DBMS itself. It does not provide handy tools for end-users. Database designers and programmers normally prefer to use single-tier architecture.

If the architecture of DBMS is 2-tier, then it must have an application through which the DBMS can be accessed. Programmers use 2-tier architecture where they access the DBMS by means of an application. Here the application tier is entirely independent of the database in terms of operation, design, and programming.

3-tier Architecture

A 3-tier architecture separates its tiers from each other based on the complexity of the users and how they use the data present in the database. It is the most widely used architecture to design a DBMS.



- **Database (Data) Tier** − At this tier, the database resides along with its query processing languages. We also have the relations that define the data and their constraints at this level.
- **Application (Middle) Tier** − At this tier reside the application server and the programs that access the database. For a user, this application tier presents an abstracted view of the database. End-users are unaware of any existence of the database beyond the application. At the other end, the database tier is not aware of any other user beyond the application tier. Hence, the application layer sits in the middle and acts as a mediator between the end-user and the database.

- **User (Presentation) Tier** − End-users operate on this tier and they know nothing about any existence of the database beyond this layer. At this layer, multiple views of the database can be provided by the application. All views are generated by applications that reside in the application tier.

Multiple-tier database architecture is highly modifiable, as almost all its components are independent and can be changed independently.

DATA MODELS

Data models define how the logical structure of a database is modeled. Data Models are fundamental entities to introduce abstraction in a DBMS. Data models define how data is connected to each other and how they are processed and stored inside the system.

The very first data model could be flat data-models, where all the data used are to be kept in the same plane. Earlier data models were not so scientific, hence they were prone to introduce lots of duplication and update anomalies.

Entity-Relationship Model

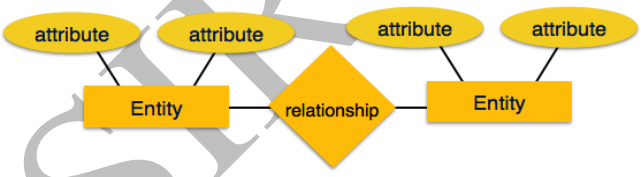
Entity-Relationship (ER) Model is based on the notion of real-world entities and relationships among them. While formulating real-world scenario into the database model, the ER Model creates entity set, relationship set, general attributes and constraints.

ER Model is best used for the conceptual design of a database.

ER Model is based on –

- **Entities** and their *attributes.*
- **Relationships** among entities.

These concepts are explained below.



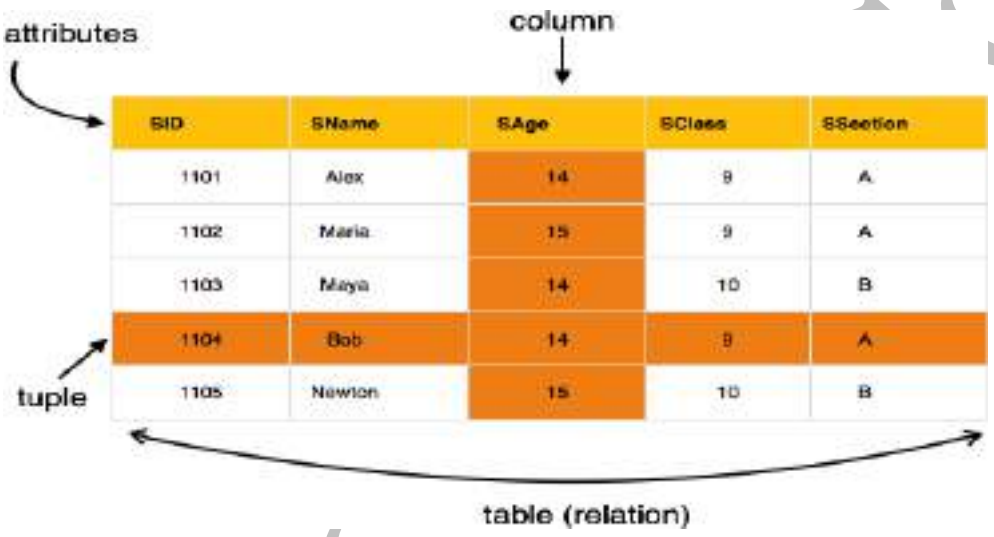
- **Entity** − An entity in an ER Model is a real-world entity having properties called **attributes**. Every **attribute** is defined by its set of values called **domain**. For example, in a school database, a student is considered as an entity. Student has various attributes like name, age, class, etc.
- **Relationship** − The logical association among entities is called ***relationship***. Relationships are mapped with entities in various ways. Mapping cardinalities define the number of association between two entities.

Mapping cardinalities −

- one to one
- one to many
- many to one
- many to many

Relational Model

The most popular data model in DBMS is the Relational Model. It is more scientific a model than others. This model is based on first-order predicate logic and defines a table as an **n-ary relation**.



The main highlights of this model are −

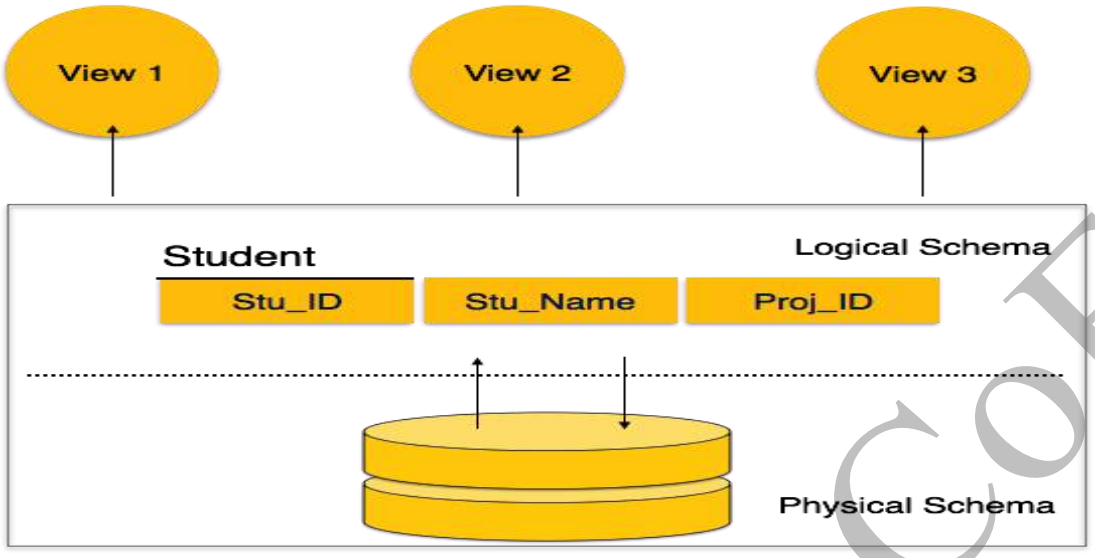
- Data is stored in tables called **relations**.
- Relations can be normalized.
- In normalized relations, values saved are atomic values.
- Each row in a relation contains a unique value.
- Each column in a relation contains values from a same domain.

Database Schema

A database schema is the skeleton structure that represents the logical view of the entire database. It defines how the data is organized and how the relations among them are associated. It formulates all the constraints that are to be applied on the data.

A database schema defines its entities and the relationship among them. It contains a descriptive detail of the database, which can be depicted by means of schema diagrams. It’s the database

designers who design the schema to help programmers understand the database and make it useful.



A database schema can be divided broadly into two categories −

- **Physical Database Schema** − This schema pertains to the actual storage of data and its form of storage like files, indices, etc. It defines how the data will be stored in a secondary storage.
- **Logical Database Schema** − This schema defines all the logical constraints that need to be applied on the data stored. It defines tables, views, and integrity constraints.

Database Instance

It is important that we distinguish these two terms individually. Database schema is the skeleton of database. It is designed when the database doesn't exist at all. Once the database is operational, it is very difficult to make any changes to it. A database schema does not contain any data or information.

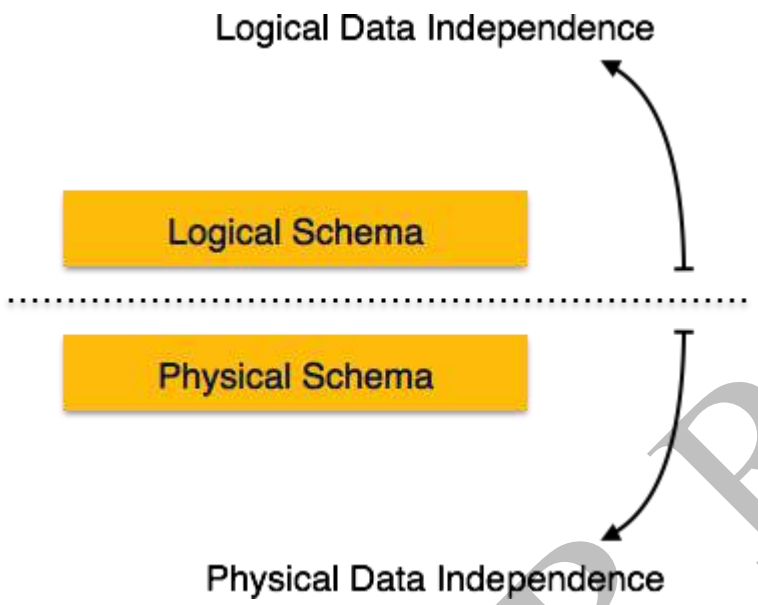
A database instance is a state of operational database with data at any given time. It contains a snapshot of the database. Database instances tend to change with time. A DBMS ensures that its every instance (state) is in a valid state, by diligently following all the validations, constraints, and conditions that the database designers have imposed.

DBMS - Data Independence

If a database system is not multi-layered, then it becomes difficult to make any changes in the database system. Database systems are designed in multi-layers as we learnt earlier.

Data Independence

A database system normally contains a lot of data in addition to users' data. For example, it stores data about data, known as metadata, to locate and retrieve data easily. It is rather difficult to modify or update a set of metadata once it is stored in the database. But as a DBMS expands, it needs to change over time to satisfy the requirements of the users. If the entire data is dependent, it would become a tedious and highly complex job.



Metadata itself follows a layered architecture, so that when we change data at one layer, it does not affect the data at another level. This data is independent but mapped to each other.

Logical Data Independence

Logical data is data about database, that is, it stores information about how data is managed inside. For example, a table (relation) stored in the database and all its constraints, applied on that relation.

Logical data independence is a kind of mechanism, which liberalizes itself from actual data stored on the disk. If we do some changes on table format, it should not change the data residing on the disk.

Physical Data Independence

All the schemas are logical, and the actual data is stored in bit format on the disk. Physical data independence is the power to change the physical data without impacting the schema or logical data.

For example, in case we want to change or upgrade the storage system itself − suppose we want to replace hard-disks with SSD − it should not have any impact on the logical data or schemas.

Advantages of DBMS

The database management system has a number of advantages as compared to traditional computer file-based processing approach. The DBA must keep in mind these benefits or capabilities during databases and monitoring the DBMS.
The Main advantages of DBMS are described below.

- **Controlling Data Redundancy**

In non-database systems each application program has its own private files. In this case, the duplicated copies of the same data is created in many places. In DBMS, all data of an organization is integrated into a single database file. The data is recorded in only one place in the database and it is not duplicated.

- **Sharing of Data**

In DBMS, data can be shared by authorized users of the organization. The database administrator manages the data and gives rights to users to access the data. Many users can be authorized to access the same piece of information simultaneously. The remote users can also share same data. Similarly, the data of same database can be shared between different application programs.

- **Data Consistency**

By controlling the data redundancy, the data consistency is obtained. If a data item appears only once, any update to its value has to be performed only once and the updated value is immediately available to all users. If the DBMS has controlled redundancy, the database system enforces consistency.

- **Integration of Data**

In Database management system, data in database is stored in tables. A single database contains multiple tables and relationships can be created between tables (or associated data entities). This makes easy to retrieve and update data.

- **Integration Constraints**

Integrity constraints or consistency rules can be applied to database so that the correct data can be entered into database. The constraints may be applied to data item within a single record or the may be applied to relationships between records.

- **Data Security**

Form is very important object of DBMS. You can create forms very easily and quickly in DBMS. Once a form is created, it can be used many times and it can be modified very easily. The created forms are also

saved along with database and behave like a software component. A form provides very easy way (user-friendly) to enter data into database, edit data and display data from database. The non-technical users can also perform various operations on database through forms without going into technical details of a fatabase.

- **Report Writers**

Most of the DBMSs provide the report writer tools used to create reports. The users can create very easily and quickly. Once a report is created, it can be used may times and it can be modified very easily. The created reports are also saved along with database and behave like a software component.

- **Control Over Concurrency**

In a computer file-based system, if two users are allowed to access data simultaneously, it is possible that they will interfere with each other. For example, if both users attempt to perform update operation on the same record, then one may overwrite the values recorded by the other. Most database management systems have sub-systems to control the concurrency so that transactions are always recorded with accuracy.

- **Backup and Recovery Procedures**

In a computer file-based system, the user creates the backup of data regularly to protect the valuable data from damage due to failures to the computer system or application program. It is very time consuming method, if amount of data is large. Most of the DBMSs provide the 'backup and recovery' sub-systems that automatically create the backup of data and restore data if required.

- **Data Independence**

The separation of data structure of database from the application program that uses the data is called data independence. In DBMS, you can easily change the structure of database without modifying the application program.

Advantages of DBMS

One of the main advantages of using a database system is that the organization can exert,via theDBA, centralized management and control over the data. The database Administrator is the focus of the centralized control.
Any application requiring a change in the structure
Of a data record requires an arrangement with the DBA, who makes the necessary modifications .
Such modifications do not affect other applications or Users of the record in question.

Reduction of Redundancies:

Centralized control of data by the DBA avoids unnecessary duplication of data and effectively reduces the total amount of data storage required.
It also eliminates the extra processing necessary to trace the required data in a large mass of data.

Elimination of Inconsistencies:

The main advantage of avoiding duplication is the elimination of inconsistencies that tend to be present in redundant data files.
Any redundancies that exist in the DBMS are controlled and the system ensures that these multiple copies are consistent.

SharedData:

A database allows the sharing of data under its control by any number of application programs or users.
For example, the applications for the public relations
And payroll departments can share the same data.

Integrity:

Centralized control can also ensure that adequate checks are incorporated in the DBMS to provide data integrity.
Data integrity means that the data contained in the database is both accurate and consistent.
Therefore, data values being entered for the storage could be checked to ensure that they fall within a specified range and are of the correct format.

Security:

Data is of vital importance to an organization and may be confidential . Such confidential data must not be accessed by unauthorized persons . The DBA who has the ultimate responsibility for the data in the DBMS can ensure that proper access procedures are followed, including proper authentication schemes for access to the DBMS and additional checks before permitting access to sensitive data .Different levels of security could be implemented for various types of data and operations.

Disadvantages of DBMS

Cost of software/hardware and migration:

A significant disadvantage of the DBMS system is cost .In addition to the cost of purchasing or developing the software, the hardware has to be upgraded to allow for the extensive programs and work spaces required for their execution and storage .
The processing overhead introduced by DBMS to implement security, integrity, and sharing of the data causes a degradation of the response and through – put times .
An additional cost is that of migration from a traditionally separate application environment to an integrated one.

Problem associated with centralization :

While centralization reduces duplication, the lack of duplication requires that the database be adequately backed up so that in the case of failure the data can be recovered. Centralization also means that the data is accessible from a single source .

This increases the potential severity of security breaches and disruption of the operation of the organization because of downtimes and failures .
The replacement of a monolithic centralized database by a federation of independent and cooperating distributed databases resolves some of the problems resulting from failures and downtimes.

Complexity of Backup and Recovery:

Backup and recovery operations are fairly complex in a DBMS environment, and this is exacerbated in a concurrent multi user database system .
Furthermore, a database system requires a certain amount of controlled redundancies and duplication to enable access to related data items .

APPLICATION OF DBMS:

Databases are used to support internal operations of organizations and to underpin online interactions with customers and suppliers (see Enterprise software).

Databases are used to hold administrative information and more specialized data, such as engineering data or economic models. Examples of database applications include computerized library systems, flight reservation systems and computerized parts inventory systems.

Application areas of DBMS

1. Banking: For customer information, accounts, and loans, and banking transactions.

2. Airlines: For reservations and schedule information. Airlines were among the first to use databases in a geographically distributed manner - terminals situated around the world accessed the central database system through phone lines and other data networks.

3. Universities: For student information, course registrations, and grades.

4. Credit card transactions: For purchases on credit cards and generation of monthly statements.

5. Telecommunication: For keeping records of calls made, generating monthly bills, maintaining balances on prepaid calling cards, and storing information about the communication networks.

6. Finance: For storing information about holdings, sales, and purchases of financial instruments such as stocks and bonds.

7. Sales: For customer, product, and purchase information.

8. Manufacturing: For management of supply chain and for tracking production of items in factories, inventories of items in warehouses / stores, and orders for items.

9. Human resources: For information about employees, salaries, payroll taxes and benefits, and for generation of paychecks

UNIT II

Relational Model: Introduction to relational model, concepts of domain, attribute, tuple, relation, importance of null values, constraints (Domain, Key constraints, integrity constraints) and their importance BASIC SQL: Simple Database schema, data types, table definitions (create, alter), different DML operations (insert, delete, update), basic SQL querying (select and project) using where clause, arithmetic & logical operations, SQL functions(Date and Time, Numeric, String conversion).

Relational Model concept

Relational model can represent as a table with columns and rows. Each row is known as a tuple. Each table of the column has a name or attribute.

Domain: It contains a set of atomic values that an attribute can take.

Attribute: It contains the name of a column in a particular table. Each attribute Ai must have a domain, dom(Ai)

Relational instance: In the relational database system, the relational instance is represented by a finite set of tuples. Relation instances do not have duplicate tuples.

Relational schema: A relational schema contains the name of the relation and name of all columns or attributes.

Relational key: In the relational key, each row has one or more attributes. It can identify the row in the relation uniquely.

Example: STUDENT Relation

NAME	ROLL_NO	PHONE_NO	ADDRESS	AGE
Ram	14795	7305758992	Noida	24
Shyam	12839	9026288936	Delhi	35
Laxman	33289	8583287182	Gurugram	20
Mahesh	27857	7086819134	Ghaziabad	27
Ganesh	17282	9028 9i3988	Delhi	40

- In the given table, NAME, ROLL_NO, PHONE_NO, ADDRESS, and AGE are the attributes.
- The instance of schema STUDENT has 5 tuples.
- t3 = <Laxman, 33289, 8583287182, Gurugram, 20>

Properties of Relations

- Name of the relation is distinct from all other relations.
- Each relation cell contains exactly one atomic (single) value
- Each attribute contains a distinct name
- Attribute domain has no significance
- tuple has no duplicate value
- Order of tuple can have a different sequence
- The SQL **NULL** is the term used to represent a missing value. A NULL value in a table is a value in a field that appears to be blank.
- A field with a NULL value is a field with no value. It is very important to understand that a NULL value is different than a zero value or a field that contains spaces.
- Syntax
- The basic syntax of **NULL** while creating a table.
- SQL> CREATE TABLE CUSTOMERS(
- ID INT NOT NULL,
- NAME VARCHAR (20) NOT NULL,
- AGE INT NOT NULL,
- ADDRESS CHAR (25) ,
- SALARY DECIMAL (18, 2),
- PRIMARY KEY (ID)
-);
- Here, **NOT NULL** signifies that column should always accept an explicit value of the given data type. There are two columns where we did not use NOT NULL, which means these columns could be NULL.
- A field with a NULL value is the one that has been left blank during the record creation.
- Example
- The NULL value can cause problems when selecting data. However, because when comparing an unknown value to any other value, the result is always unknown and not included in the results. You must use the **IS NULL** or **IS NOT NULL** operators to check for a NULL value.
- Consider the following CUSTOMERS table having the records as shown below.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|1|Ramesh|32|Ahmedabad|2000.00|
|2|Khilan|25|Delhi|1500.00|
|3|kaushik|23|Kota|2000.00|
|4|Chaitali|25|Mumbai|6500.00|
|5|Hardik|27|Bhopal|8500.00|
|6|Komal|22| MP        ||
|7|Muffy|24|Indore||
+----+----------+-----+-----------+----------+
```

- Now, following is the usage of the **IS NOT NULL**operator.

```
SQL> SELECT  ID, NAME, AGE, ADDRESS, SALARY
  FROM CUSTOMERS
  WHERE SALARY IS NOT NULL;
```

- This would produce the following result −

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
+----+----------+-----+-----------+----------+
```

- Now, following is the usage of the **IS NULL** operator.

```
SQL> SELECT  ID, NAME, AGE, ADDRESS, SALARY
  FROM CUSTOMERS
  WHERE SALARY IS NULL;
```

- This would produce the following result −

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  6 | Komal    |  22 | MP        |          |
|  7 | Muffy    |  24 | Indore    |          |


+----+----------+-----+-----------+----------+
```

In general, each NULL value is considered to be different from every other NULL in the database. When a NULL is involved in a comparison operation, the result is considered to be UNKNOWN. Hence, SQL uses a three-valued logic with values **True, False** and **Unknown.** It is, therefore, necessary to define the results of three-valued logical expressions when the logical
connectives AND, OR, and NOT are used.

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

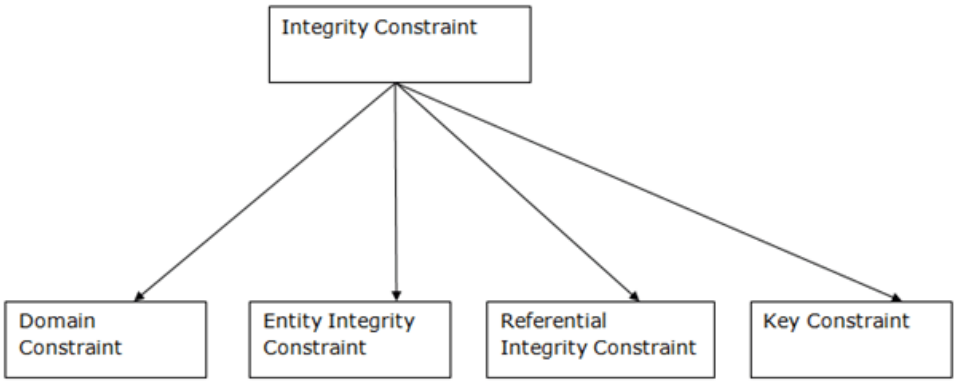
OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

NOT	TRUE
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

Integrity Constraints

- Integrity constraints are a set of rules. It is used to maintain the quality of information.
- Integrity constraints ensure that the data insertion, updating, and other processes have to be performed in such a way that data integrity is not affected.
- Thus, integrity constraint is used to guard against accidental damage to the database.

Types of Integrity Constraint



1. Domain constraints

- Domain constraints can be defined as the definition of a valid set of values for an attribute.
- The data type of domain includes string, character, integer, time, date, currency, etc. The value of the attribute must be available in the corresponding domain.

Example:

ID	NAME	SEMENSTER	AGE
1000	Tom	1st	17
1001	Johnson	2nd	24
1002	Leonardo	5th	21
1003	Kate	3rd	19
1004	Morgan	8th	A

Not allowed. Because AGE is an integer attribute

2. Entity integrity constraints

- The entity integrity constraint states that primary key value can't be null.
- This is because the primary key value is used to identify individual rows in relation and if the primary key has a null value, then we can't identify those rows.

- A table can contain a null value other than the primary key field.

Example:

EMPLOYEE

EMP_ID	EMP_NAME	SALARY
123	Jack	30000
142	Harry	60000
164	John	20000
	Jackson	27000

Not allowed as primary key can't contain a NULL value

3. Referential Integrity Constraints

- A referential integrity constraint is specified between two tables.
- In the Referential integrity constraints, if a foreign key in Table 1 refers to the Primary Key of Table 2, then every value of the Foreign Key in Table 1 must be null or be available in Table 2.

Example:

(Table 1)

EMP_NAME	NAME	AGE	D_No
1	Jack	20	11
2	Harry	40	24
3	John	27	18
4	Devil	38	13

Foreign key

Not allowed as D_No 18 is not defined as a Primary key of table 2 and in table 1, D_No is a foreign key defined

Relationships

(Table 2)

Primary Key

D_No	D_Location
11	Mumbai
24	Delhi
13	Noida

4. Key constraints

- Keys are the entity set that is used to identify an entity within its entity set uniquely.
- An entity set can have multiple keys, but out of which one key will be the primary key. A primary key can contain a unique and null value in the relational table.

Example:

ID	NAME	SEMENSTER	AGE
1000	Tom	1st	17
1001	Johnson	2nd	24
1002	Leonardo	5th	21
1003	Kate	3rd	19
1002	Morgan	8th	22

Not allowed. Because all row must be unique

Oracle Built-In-Functions:

1. **ASCII:** Returns the number code that represents the specific character.
   Query synatx: ASCII(single_character)

```
SQL> CREATE TABLE STUDENT(SNAME VARCHAR(20),SRNO INT);

Table created.

SQL> SELECT ASCII(SNAME) AS NumCodeOffFirstChar FROM STUDENT;

no rows selected

SQL> INSERT INTO STUDENT(SNAME,SRNO) VALUES('SRINVI','522');

1 row created.

SQL> INSERT INTO STUDENT(SNAME,SRNO) VALUES('SHRI','545');

1 row created.

SQL> SELECT ASCII(SNAME) AS NumCodeOffFirstChar FROM STUDENT;

NUMCODEOFFFIRSTCHAR
-------------------
                 83
                 83
```

2. **CONCAT:** Concatenates two or more strings together.
   Query syntax:CONCAT(string1,string2)

```
SQL> select concat(SNAME,SRNO) from student;

CONCAT(SNAME,SRNO)
------------------------------------------------------------
SRINVI522
SHRI545
```

3. **LENGTH:** Returns the length of the specified string
   Query synatx: LENGTH(string)

```
SQL> select length('hello') from dual;

LENGTH('HELLO')
---------------
              5
```

4. **REPLACE:** Replaces a sequence of characters in a string with another set of character.
   Query syntax: REPLACE(expression,pattern,replacement)

```
SQL> select replace('good morning','morning','day') from dual;

REPLACE(
--------
good day
```

String functions in SQL:

1. **LOWER:** All the letters in 'string_value' is converted to lowercase.
   Query syntax: LOWER(string_value)

```
SQL> select lower('Hello World') from dual;

LOWER('HELL
-----------
hello world
```

2. **UPPER:** All the letters in 'string_value' is converted to upper case.
   Query syntax: UPPER(string_value)

```
SQL> select upper('hello world') from dual;

UPPER('HELL
-----------
HELLO WORLD
```

3. **INITCAP:** All the letters in 'string_value' is converted to mixed case.
   Query syntax:INITCAP(string_value)

```
SQL> select initcap('hello world') from dual;

INITCAP('HE
-----------
Hello World
```

4. **LPAD:** Returns 'string_value' left-padded with 'pad_value' .length of the whole string will be of 'n' characters**.**
   Query syntax: LPAD(string_value, n, pad_value)

```
SQL> select lpad('hello',7,'#') from dual;

LPAD('H
-------
##hello
```

5. **RPAD:** Returns 'string_value' right-padded with 'pad_value', length of the whole string will be of 'n' characters.
   Query syntax: RPAD(string_value,n,pad_value)

```
SQL> select rpad('hello',7,'#') from dual;

RPAD('H
-------
hello##
```

Numeric functions in SQL:

1. **ABS:** Returns absolute value of the number 'x'
   Query syntax: ABS(x)

```
SQL> select abs(1) from dual;

   ABS(1)
----------
         1
```

2. **CEIL:** Returns integer value that is greater than or equal to the number 'x'
   Query syntax: CEIL(x)

```
SQL> select ceil(5.432) from dual;

CEIL(5.432)
-----------
          6
```

3. **FLOOR:** Returns integer value that is less than or equal to the number 'x'
   Query syntax: FLOOR(x)

```
SQL> select floor(5.432) from dual;

FLOOR(5.432)
------------
           5
```

4. **ROUND:** Returns rounded off value of the number 'x' upto 'y'.
   Query syntax: ROUND(x,y)

```
SQL> select round(5.432,1) from dual;

ROUND(5.432,1)
--------------
           5.4
```

Date functions in SQL:

1. ADD_MONTHS: Return a date value after adding ‘n’ months to the date ‘x’.

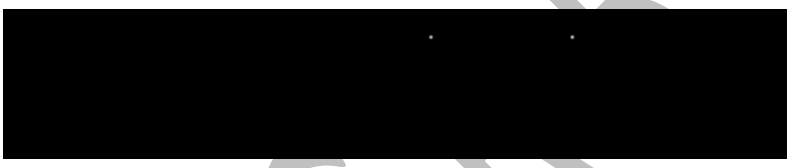
Query syntax: ADD_MONTHS(date,n)

```
SQL> SELECT ADD_MONTHS('16-Sep-81', 3) FROM DUAL;

ADD_MONTH
---------
16-DEC-81
```

2. MONTHS_BETWEEN: Returns the number of months between dates x1 and x2

Query syntax: MONTHS_BETWEEN(x1,x2)



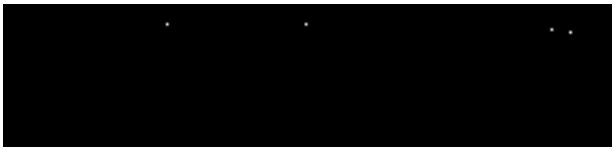
3. LAST_DAY: It is used to determine the number of days remaining in a month from the date 'x' specified.

Query syntax: LAST_DAY(x)

Conversion functions in SQL:

1. TO_CHAR: Converts Numeric and Date values to a character string value. It cannot be used for calculations since it is a string value.

Query syntax: TO_CHAR (x [,y])

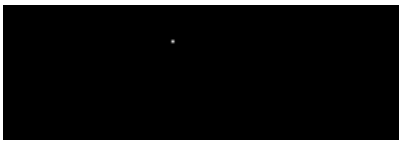


2. TO_DATE: Converts a valid Numeric and Character values to a Date value. Date is formatted to the format specified by 'date_format'.

Query syntax: TO_DATE (x [, date_format])

3. NVL: If 'x' is NULL, replace it with 'y'. 'x' and 'y' must be of the same datatype.

Query syntax: NVL(x,y)



Queries using operators in SQL.

An operator manipulates individual data items and returns a result. The data items are called operands or arguments. Operators are represented by special characters or by keywords. For example, the multiplication operator is represented by an asterisk (*) and the operator that tests for nulls is represented by the keywords IS NULL. There are two general classes of operators: unary and binary. Oracle Database Lite SQL also supports set operators.

Arithmetic operators in sql:

1.+ (unary operator): Makes operand positive.

2.-(unary operator): Negates operand.

```
SQL> select* from employee;

ENAME             EID        SAL
---------- ---------- ----------
karthik             1      20000
pqrs                2       3500

SQL> select sal-3500 from employee;

  SAL-3500
----------
     16500
         0
```

3./ (Division operator): Division (numbers and dates)

```
SQL> select sal/5 from employee;

     SAL/5
----------
      4000
       700
```

4.* (Multiplication operator): Multiplication

```
SQL> select sal*3 from employee;

     SAL*3
----------
     60000
     10500
```

Comparison operators:

a)Equal to Operator(=): Checks if the values of two operands are equal or not, if yes then; condition becomes true.

```
SQL> create table employee(ename varchar(10),eid int,esal int);

Table created.

SQL> insert into employee(ename,eid,esal) values('shri','1','30000');

1 row created.

SQL> insert into employee(ename,eid,esal) values('srinvi','2','25000');

1 row created.

SQL> insert into employee(ename,eid,esal) values('arjun','3','50000');

1 row created.

SQL> insert into employee(ename,eid,esal) values('jesweer','4','65000');

1 row created.
```

```
SQL> select ename from employee where esal=50000;

ENAME
----------
arjun
```

2.Not Equal to Operator(!=): Checks if the values of two operands are equal or not, if values are not equal then;condition becomes true.

```
SQL> select ename from employee where esal!=50000;

ENAME
----------
shri
srinvi
jesweer
```

3.Greater than Operator(>): Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.

```
SQL> select ename from employee where esal>30000;

ENAME
----------
arjun
jesweer
```

4.Less than Operator (<): Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.

```
SQL> select ename from employee where esal<30000;

ENAME
----------
srinvi
```

4.Greater than or Equal to Operator (>=): Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.

```
SQL> select ename from employee where esal>=30000;

ENAME
----------
shri
arjun
jesweer
```

Logical operators:

a)ALL Operator: The ALL operator is used to compare a value to all values in another value set.

b)AND Operator: The ALL operator is used to compare a value to all values in another value set.

```
SQL> select ename from employee where esal>=30000 and esal<65000;

ENAME
----------
shri
arjun
```

c) ANY Operator: The ANY operator is used to compare a value to any applicable value in the list as per the condition.

CREATE TABLE

reating a basic table involves naming the table and defining its columns and each column's data type.

The SQL **CREATE TABLE** statement is used to create a new table.

Syntax

The basic syntax of the CREATE TABLE statement is as follows −

CREATE TABLE table_name(
column1datatype,

```
column2datatype,
column3datatype,
   .....
columnNdatatype,
   PRIMARY KEY( one or more columns )
);
```

CREATE TABLE is the keyword telling the database system what you want to do. In this case, you want to create a new table. The unique name or identifier for the table follows the CREATE TABLE statement.

Then in brackets comes the list defining each column in the table and what sort of data type it is. The syntax becomes clearer with the following example.

A copy of an existing table can be created using a combination of the CREATE TABLE statement and the SELECT statement. You can check the complete details at Create Table Using another Table.

Example

The following code block is an example, which creates a CUSTOMERS table with an ID as a primary key and NOT NULL are the constraints showing that these fields cannot be NULL while creating records in this table −

```
SQL> CREATE TABLE CUSTOMERS(
   ID   INT              NOT NULL,
   NAME VARCHAR (20)     NOT NULL,
AGE  INT              NOT NULL,
ADDRESS  CHAR(25),
   SALARY   DECIMAL (18,2),
   PRIMARY KEY (ID)
);
```

You can verify if your table has been created successfully by looking at the message displayed by the SQL server, otherwise you can use the **DESC** command as follows −

```
SQL> DESC CUSTOMERS;
+---------+--------------+------+-----+---------+-------+
| Field   | Type         | Null | Key | Default | Extra |
+---------+--------------+------+-----+---------+-------+
| ID      | int(11)      | NO   | PRI |         |       |
| NAME    | varchar(20)  | NO   |     |         |       |
| AGE     | int(11)      | NO   |     |         |       |
| ADDRESS | char(25)     | YES  |     | NULL    |       |
| SALARY  | decimal(18,2) | YES  |     | NULL    |       |
+---------+--------------+------+-----+---------+-------+
5 rows in set (0.00 sec)
```

Now, you have CUSTOMERS table available in your database which you can use to store the required information related to customers.

The SQL **DROP TABLE** statement is used to remove a table definition and all the data, indexes, triggers, constraints and permission specifications for that table.

NOTE − You should be very careful while using this command because once a table is deleted then all the information available in that table will also be lost forever.

Syntax

The basic syntax of this DROP TABLE statement is as follows −

```
DROP TABLE table_name;
```

Example

Let us first verify the CUSTOMERS table and then we will delete it from the database as shown below −

```
SQL> DESC CUSTOMERS;
+---------+--------------+------+-----+---------+-------+
|Field|Type|Null|Key|Default|Extra|
+---------+--------------+------+-----+---------+-------+
| ID      |int(11)| NO   | PRI |||
| NAME    |varchar(20)| NO   ||||
| AGE     |int(11)| NO   ||||
| ADDRESS |char(25)| YES  || NULL    ||
|SALARY  |decimal(18,2)| YES  || NULL    ||
+---------+--------------+------+-----+---------+-------+
5 rows inset(0.00 sec)
```

This means that the CUSTOMERS table is available in the database, so let us now drop it as shown below.

```
SQL> DROP TABLE CUSTOMERS;
Query OK,0 rows affected (0.01 sec)
```

The SQL **INSERT INTO** Statement is used to add new rows of data to a table in the database.

Syntax

There are two basic syntaxes of the INSERT INTO statement which are shown below.

```
INSERT INTO TABLE_NAME (column1, column2, column3,...columnN)
VALUES (value1, value2, value3,...valueN);
```

Here, column1, column2, column3,...columnN are the names of the columns in the table into which you want to insert the data.

You may not need to specify the column(s) name in the SQL query if you are adding values for all the columns of the table. But make sure the order of the values is in the same order as the columns in the table.

The **SQL INSERT INTO** syntax will be as follows −

```
INSERT INTO TABLE_NAME VALUES (value1,value2,value3,...valueN);
```

Example

The following statements would create six records in the CUSTOMERS table.

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1,'Ramesh',32,'Ahmedabad',2000.00);

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (2,'Khilan',25,'Delhi',1500.00);

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (3,'kaushik',23,'Kota',2000.00);

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (4,'Chaitali',25,'Mumbai',6500.00);

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (5,'Hardik',27,'Bhopal',8500.00);

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (6,'Komal',22,'MP',4500.00);
```

You can create a record in the CUSTOMERS table by using the second syntax as shown below.

```
INSERT INTO CUSTOMERS
VALUES (7, 'Muffy', 24, 'Indore', 10000.00 );
```

All the above statements would produce the following records in the CUSTOMERS table as shown below.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|1|Ramesh|32|Ahmedabad|2000.00|
|2|Khilan|25|Delhi|1500.00|
|3|kaushik|23|Kota|2000.00|
|4|Chaitali|25|Mumbai|6500.00|
|5|Hardik|27|Bhopal|8500.00|
|6|Komal|22| MP       |4500.00|
|7|Muffy|24|Indore|10000.00|
+----+----------+-----+-----------+----------+
```

The SQL **SELECT** statement is used to fetch the data from a database table which returns this data in the form of a result table. These result tables are called result-sets.

Syntax

The basic syntax of the SELECT statement is as follows −

```
SELECT column1, column2, columnN FROM table_name;
```

Here, column1, column2... are the fields of a table whose values you want to fetch. If you want to fetch all the fields available in the field, then you can use the following syntax.

```
SELECT * FROM table_name;
```

Example

Consider the CUSTOMERS table having the following records −

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|1|Ramesh|32|Ahmedabad|2000.00|
|2|Khilan|25|Delhi|1500.00|
|3|kaushik|23|Kota|2000.00|
|4|Chaitali|25|Mumbai|6500.00|
|5|Hardik|27|Bhopal|8500.00|
|6|Komal|22| MP       |4500.00|
|7|Muffy|24|Indore|10000.00|
+----+----------+-----+-----------+----------+
```

The following code is an example, which would fetch the ID, Name and Salary fields of the customers available in CUSTOMERS table.

```
SQL> SELECT ID, NAME, SALARY FROM CUSTOMERS;
```

This would produce the following result −

```
+----+----------+----------+
| ID | NAME     | SALARY   |
+----+----------+----------+
|  1 | Ramesh   |  2000.00 |
|  2 | Khilan   |  1500.00 |
|  3 | kaushik  |  2000.00 |
|  4 | Chaitali |  6500.00 |
|  5 | Hardik   |  8500.00 |
|  6 | Komal    |  4500.00 |
|  7 | Muffy    | 10000.00 |
+----+----------+----------+
```

If you want to fetch all the fields of the CUSTOMERS table, then you should use the following query.

```
SQL> SELECT * FROM CUSTOMERS;
```

This would produce the result as shown below.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

The SQL **WHERE** clause is used to specify a condition while fetching the data from a single table or by joining with multiple tables. If the given condition is satisfied, then only it returns a specific value from the table. You should use the WHERE clause to filter the records and fetching only the necessary records.

The WHERE clause is not only used in the SELECT statement, but it is also used in the UPDATE, DELETE statement, etc., which we would examine in the subsequent chapters.

Syntax

The basic syntax of the SELECT statement with the WHERE clause is as shown below.

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition]
```

You can specify a condition using the comparison or logical operators like >, <, =, **LIKE, NOT**, etc. The following examples would make this concept clear.

Example

Consider the CUSTOMERS table having the following records −

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|1|Ramesh|32|Ahmedabad|2000.00|
|2|Khilan|25|Delhi|1500.00|
|3|kaushik|23|Kota|2000.00|
|4|Chaitali|25|Mumbai|6500.00|
|5|Hardik|27|Bhopal|8500.00|
|6|Komal|22| MP       |4500.00|
|7|Muffy|24|Indore|10000.00|
+----+----------+-----+-----------+----------+
```

The following code is an example which would fetch the ID, Name and Salary fields from the CUSTOMERS table, where the salary is greater than 2000 −

```
SQL> SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE SALARY >2000;
```

This would produce the following result −

```
+----+----------+----------+
| ID | NAME     | SALARY   |
+----+----------+----------+
|  4 | Chaitali |  6500.00 |
|  5 | Hardik   |  8500.00 |
|  6 | Komal    |  4500.00 |
|  7 | Muffy    | 10000.00 |
+----+----------+----------+
```

The following query is an example, which would fetch the ID, Name and Salary fields from the CUSTOMERS table for a customer with the name **Hardik**.

Here, it is important to note that all the strings should be given inside single quotes ('). Whereas, numeric values should be given without any quote as in the above example.

```
SQL> SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE NAME ='Hardik';
```

This would produce the following result −

```
+----+----------+----------+
| ID | NAME     | SALARY   |
+----+----------+----------+
|  5 | Hardik   |  8500.00 |
```

The SQL **AND** & **OR** operators are used to combine multiple conditions to narrow data in an SQL statement. These two operators are called as the conjunctive operators.

These operators provide a means to make multiple comparisons with different operators in the same SQL statement.

The AND Operator

The **AND** operator allows the existence of multiple conditions in an SQL statement's WHERE clause.

Syntax

The basic syntax of the AND operator with a WHERE clause is as follows −

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition1] AND [condition2]...AND [conditionN];
```

You can combine N number of conditions using the AND operator. For an action to be taken by the SQL statement, whether it be a transaction or a query, all conditions separated by the AND must be TRUE.

Example

Consider the CUSTOMERS table having the following records −

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
```

```
+----+----------+-----+-----------+----------+
```

Following is an example, which would fetch the ID, Name and Salary fields from the CUSTOMERS table, where the salary is greater than 2000 and the age is less than 25 years −

```
SQL> SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE SALARY >2000 AND age <25;
```

This would produce the following result −

```
+----+-------+----------+
| ID | NAME  | SALARY   |
+----+-------+----------+
|  6 | Komal |  4500.00 |
|  7 | Muffy | 10000.00 |
+----+-------+----------+
```

The OR Operator

The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause.

Syntax

The basic syntax of the OR operator with a WHERE clause is as follows −

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition1] OR [condition2]...OR [conditionN]
```

You can combine N number of conditions using the OR operator. For an action to be taken by the SQL statement, whether it be a transaction or query, the only any ONE of the conditions separated by the OR must be TRUE.

Example

Consider the CUSTOMERS table having the following records −

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|1|Ramesh|32|Ahmedabad|2000.00|
|2|Khilan|25|Delhi|1500.00|
|3|kaushik|23|Kota|2000.00|
|4|Chaitali|25|Mumbai|6500.00|
|5|Hardik|27|Bhopal|8500.00|
|6|Komal|22| MP       |4500.00|
|7|Muffy|24|Indore|10000.00|
+----+----------+-----+-----------+----------+
```

The following code block hasa query, which would fetch the ID, Name and Salary fields from the CUSTOMERS table, where the salary is greater than 2000 OR the age is less than 25 years.

```
SQL> SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE SALARY >2000 OR age <25;
```

This would produce the following result −

```
+----+----------+----------+
| ID | NAME     | SALARY   |
+----+----------+----------+
|  3 | kaushik  |  2000.00 |
|  4 | Chaitali |  6500.00 |
|  5 | Hardik   |  8500.00 |
|  6 | Komal    |  4500.00 |
|  7 | Muffy    | 10000.00 |
+----+----------+----------+
```

The SQL **UPDATE** Query is used to modify the existing records in a table. You can use the WHERE clause with the UPDATE query to update the selected rows, otherwise all the rows would be affected.

Syntax

The basic syntax of the UPDATE query with a WHERE clause is as follows −

```
UPDATE table_name
SET column1 = value1, column2 = value2...., columnN = valueN
WHERE [condition];
```

You can combine N number of conditions using the AND or the OR operators.

Example

Consider the CUSTOMERS table having the following records −

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|1|Ramesh|32|Ahmedabad|2000.00|
|2|Khilan|25|Delhi|1500.00|
|3|kaushik|23|Kota|2000.00|
|4|Chaitali|25|Mumbai|6500.00|
|5|Hardik|27|Bhopal|8500.00|
|6|Komal|22| MP       |4500.00|
|7|Muffy|24|Indore|10000.00|
+----+----------+-----+-----------+----------+
```

The following query will update the ADDRESS for a customer whose ID number is 6 in the table.

```
SQL> UPDATE CUSTOMERS
SET ADDRESS ='Pune'
WHERE ID =6;
```

Now, the CUSTOMERS table would have the following records −

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | Pune      |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

If you want to modify all the ADDRESS and the SALARY column values in the CUSTOMERS table, you do not need to use the WHERE clause as the UPDATE query would be enough as shown in the following code block.

```
SQL> UPDATE CUSTOMERS
SET ADDRESS ='Pune', SALARY =1000.00;
```

Now, CUSTOMERS table would have the following records −

```
+----+----------+-----+---------+---------+
| ID | NAME     | AGE | ADDRESS | SALARY  |
+----+----------+-----+---------+---------+
|  1 | Ramesh   |  32 | Pune    | 1000.00 |
|  2 | Khilan   |  25 | Pune    | 1000.00 |
|  3 | kaushik  |  23 | Pune    | 1000.00 |
|  4 | Chaitali |  25 | Pune    | 1000.00 |
|  5 | Hardik   |  27 | Pune    | 1000.00 |
|  6 | Komal    |  22 | Pune    | 1000.00 |
|  7 | Muffy    |  24 | Pune    | 1000.00 |
+----+----------+-----+---------+---------+
```

The SQL DELETE Query is used to delete the existing records from a table.

You can use the WHERE clause with a DELETE query to delete the selected rows, otherwise all the records would be deleted.

Syntax

The basic syntax of the DELETE query with the WHERE clause is as follows −

```
DELETE FROM table_name
WHERE [condition];
```

You can combine N number of conditions using AND or OR operators.

Example

Consider the CUSTOMERS table having the following records −

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
```

```
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

The following code has a query, which will DELETE a customer, whose ID is 6.

```
SQL> DELETE FROM CUSTOMERS
WHERE ID =6;
```

Now, the CUSTOMERS table would have the following records.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

UNIT-III

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

1. OVERVIEW OF DATABASE DESIGN: (**********)

→The **database design process** can be divided into six steps. The **ER model** is most relevant to the first three steps:

(1) **Requirements Analysis:** The very first step in designing a database application is to understand what data is to be stored in the database, what applications must be built on the database and what operations must be performed on the database. In other words, we must find out what the users want from the database. This process involves discussions with user groups, a study of the current operating environment, how it is expected to change an analysis of any available documentation on existing applications and so on.

(2) **Conceptual Database Design:** The information gathered in the requirements analysis step is used to develop a high-level description of the data to be stored in the database, along with the constraints that are known to hold over this data. The goal is to create a description of the data that matches to how both users and developers think of the data. This facilities discussion among all the people involved in the design process i.e., developers and as well as users who have no technical background. In simple words, the conceptual database design phase is used in drawing ER model.

(3) **Logical Database Design:** We must implement our database design and convert the conceptual database design into a database schema (a description of data) in the data model (a collection of high level data description constructs that hide many low level storage details) of the DBMS. We will consider only consider relational DBMSs, and therefore, the task in the logical design step is to convert the conceptual database design in the form of an ER schema (Entity-Relationship schema) into a relational database schema.

(4) **Schema Refinement:** The fourth step in database design is to analyze the collection of relations in our relational database schema to identify future problems, and to refine (clear) it.

(5) **Physical Database Design:**. This step may simply involve building indexes on some tables and clustering some tables, or it may involve redesign of parts of the database schema obtained from the earlier design steps.

(6) **Application andSecurity Design:** Any software project that involves a DBMS must consider applications that involve processes and identify the entities.

Example: Users, user groups, departments, etc.

→We must describe the role of each entity in every process. As a security design, for each role, we must identify the parts of the database that just not is accessible and we must take steps to ensure that these access rules are enforced.

Conceptual design:

- **What are the entities and relationships in the enterprise?**
- **What information about these entities and relationships should be stored in the database?**
- **What are the integrity constraints or business rules that hold?**
- **A database schema in the ER Model can be represented pictorially (ER diagrams)**
- **An ER diagram can be mapped into a relational schema**

E-R MODEL:

→An **entity–relationship model (ER model)** is a systematic way of describing and defining a business process. An ER model is typically implemented as a database.
→The main components of E-R model are: entity set and relationship set.

→Here are the geometric shapes and their meaning in an E-R Diagram –

Rectangle: Represents Entity sets.

Ellipses: Attributes

Diamonds: Relationship Set

Lines: They link attributes to Entity Sets and Entity sets to Relationship Set

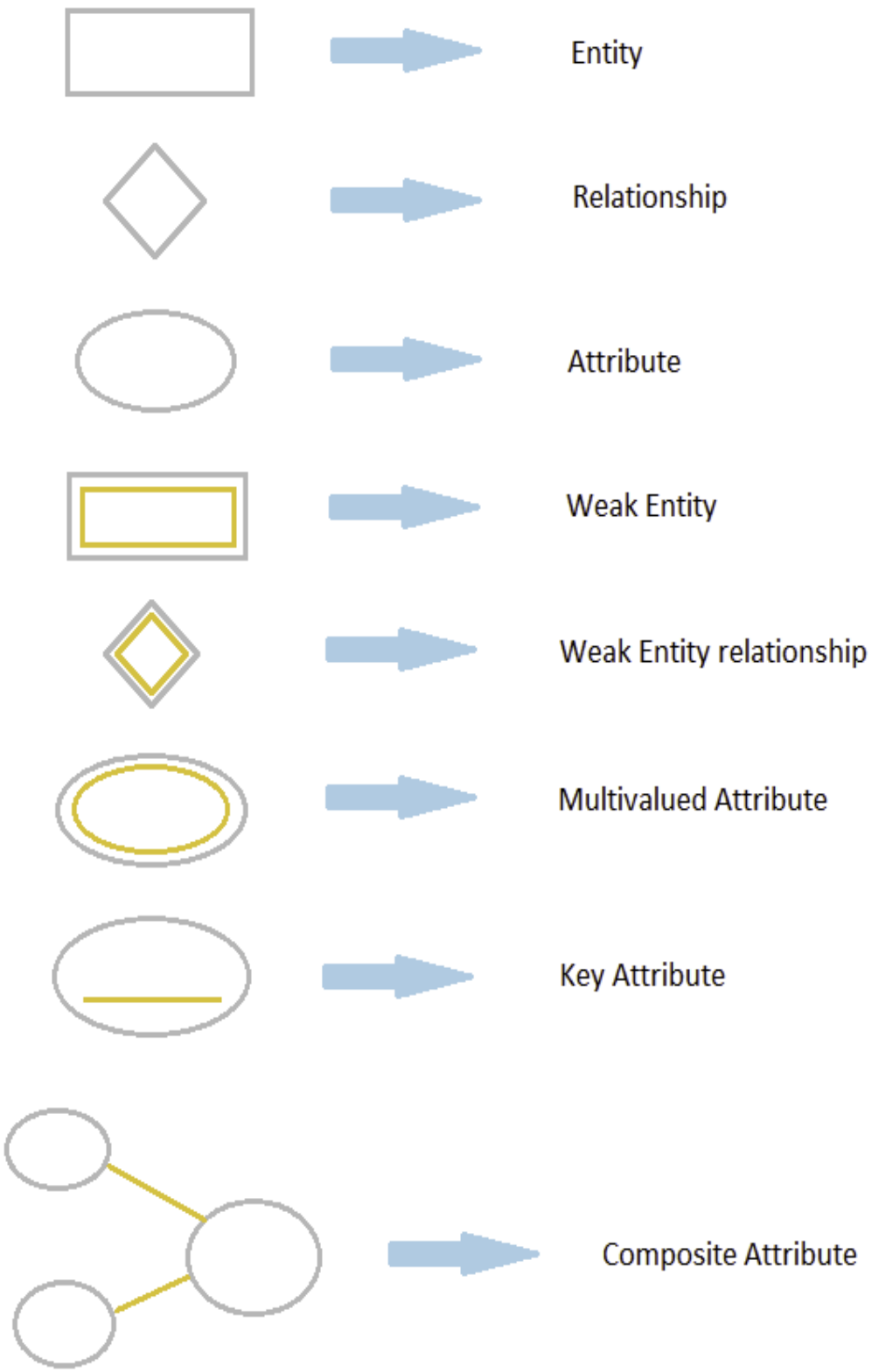
Double Ellipses: Multivalued Attributes

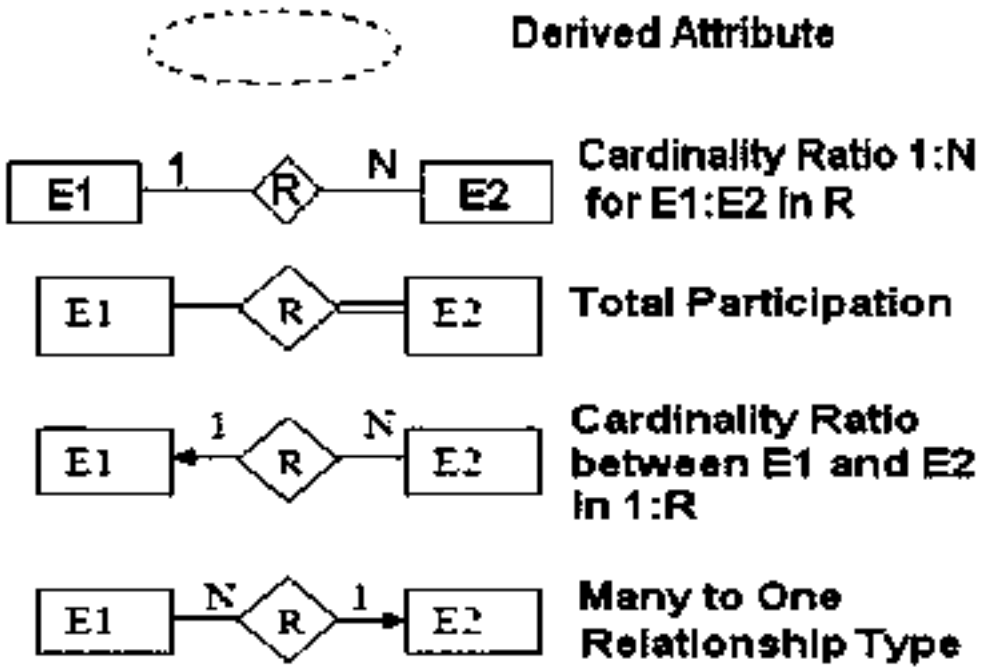
Dashed Ellipses: Derived Attributes

Double Rectangles: Weak Entity Sets

Double Lines: Total participation of an entity in a relationship set

Symbols and Notations





2. ENTITIES, ATTRIBUTES, AND ENTITY SETS:

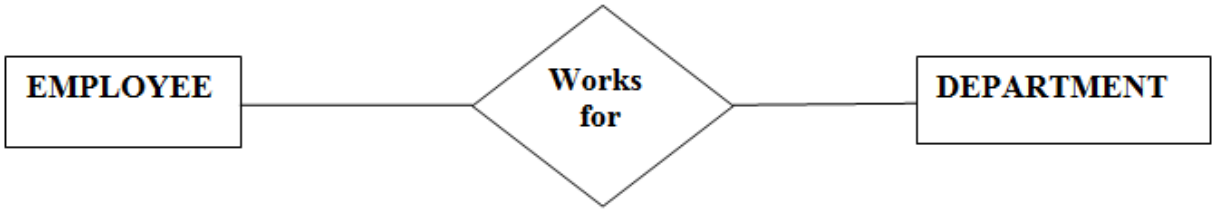
ENTITIES:

An **Entity**is an object that exists and is distinguishable from other objects.

Example: Specific person, Company, Event, Plant, Building, Room, Chair, Course, Employee etc.

In E-R Diagram, an **entity** is represented using rectangles. Name of the Entity is written inside the rectangle.
→**Examples**: STUDENT, EMPLOYEE, ACCOUNT etc.

A **Strong entity** is represented by simple rectangle as shown above.
→Consider an **example** of an Organization. Employee, Manager, Department, Product and many more can be taken as entities from an Organization.



A **Weak entity** is an entity that depends on another entity. Weak entity doesn't have key attribute of their own. Double rectangle represents weak entity.

Examples: CLASS_SECTION, DEPENDANT etc.

An **Entity set**is a set of entities of the same type that share the same properties.
→**An Entity set** is a collection of similar entities.
Examples: set of all persons, companies, Job positions, Courses, Academic staff, Managers, Employees etc.
–All entities in an entity set have the same set of attributes. (Until we consider ISA hierarchies, anyway!)
–Each entity set has a *key*.
–Each attribute has a *domain*.

→The **Employees entity set** with attributes ssn, name, and lot is shown in Figure 2.1. An entity set is represented by a rectangle, and an attribute is represented by an oval. Each attribute in the **primary key** is underlined.

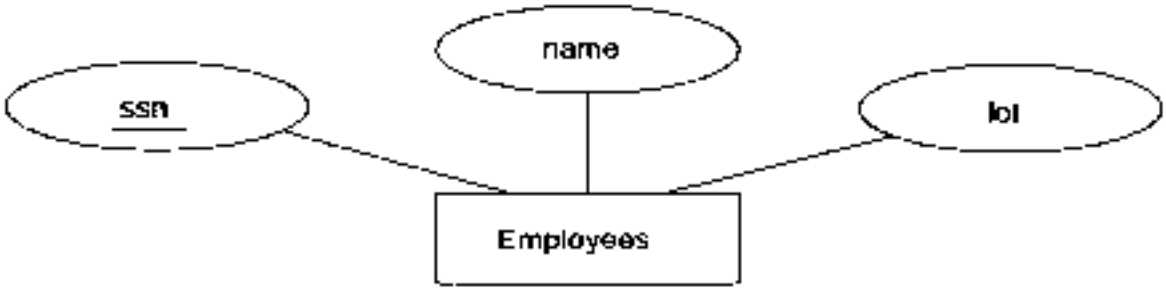
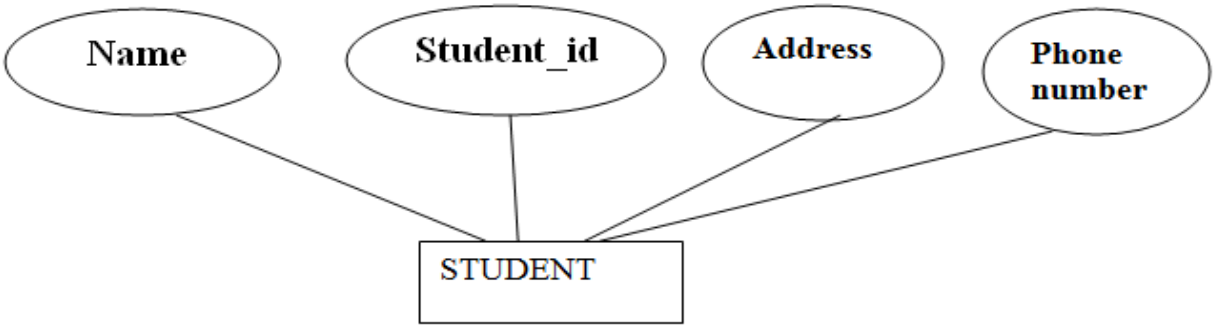


Figure 2.1 The Employees Entity Set

ATTRIBUTES:

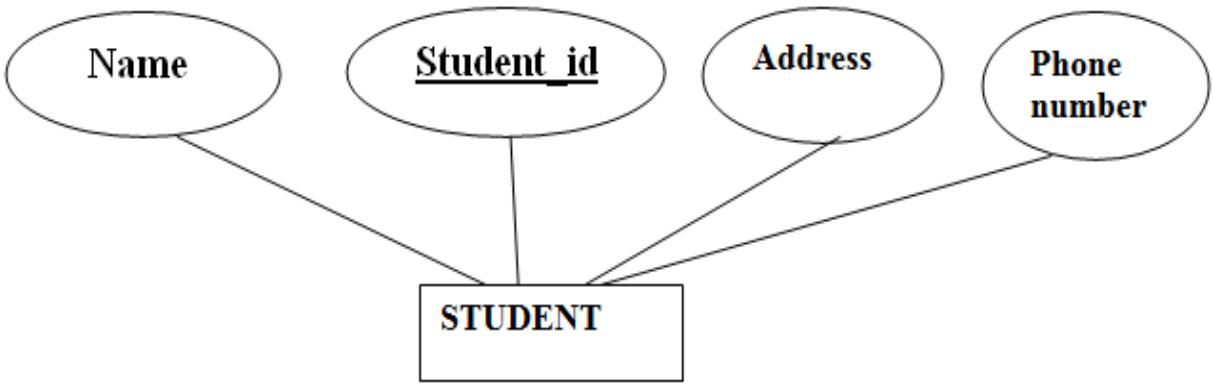
An entity is represented by a set of **attributes. Attributes are** descriptive properties possessed by each member of an entity set.

An **Attribute** describes a property or characteristics of an entity. For example, Name, Age, Address etc can be attributes of a Student. An attribute is represented using eclipse.

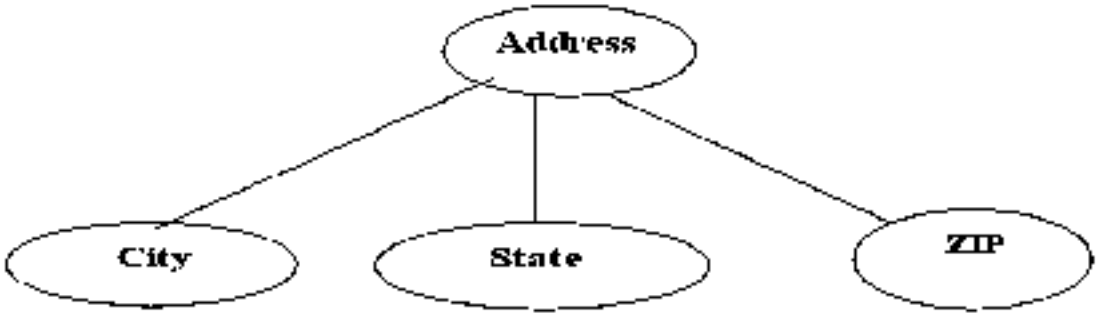


Example:
Customer=(customer_id, customer_name, customer_street, customer_city)
loan = (loan_number, amount)

Key Attribute:
Key attribute represents the main characteristics of an Entity. It is used to represent Primary key. Ellipse with underlying lines represents Key Attribute.



Composite Attribute:
An attribute can also have their own attributes. These attributes are known as **Composite** attribute.
→ Composite attributes can be divided into subparts. For example, an attribute name could be structured as a composite attribute consisting of first-name, middle-initial, and last-name.
Attribute Divided into sub parts. Eg. Name (First name, Middle Name, last name)

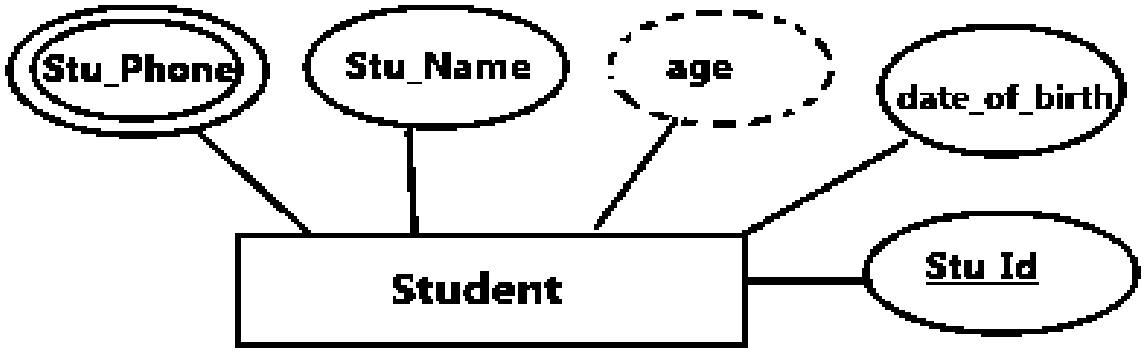


Multivalued Attributes:(**********)
An attribute that can hold multiple values is known as multivalued attribute. We represent it with double ellipses in an E-R Diagram. E.g. A person can have more than one phone numbers so the phone number attribute is multivalued.

There may be instances where an attribute has a set of values for a specific entity. Consider an employee entity set with the attribute phone-number. An employee may have zero, one, or several phone numbers, and different employees may have different numbers of phones. This type of attribute is said to be attribute having more than one values. Eg.Phone Number.

Derived Attribute: A derived attribute is one whose value is dynamic and derived from another attribute. It is represented by dashed ellipses in an E-R Diagram. E.g. Person age is a derived attribute as it changes over time and can be derived from another attribute (Date of birth).

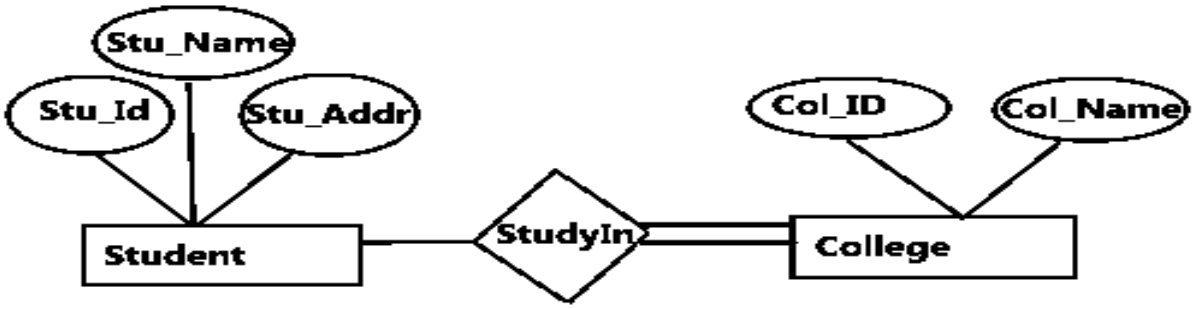
E-R diagram with multivalued and derived attributes:



Total Participation of an Entity set:

A total participation of an entity set represents that each entity in entity set must have at least one relationship in a relationship set. For example: In the below diagram each college must have at least one associated student.

.



E-R Digram with total participation of College entity set in StudyIn relationship Set – This indicates that each college must have atleast one associated Student.

3. RELATIONSHIPS AND RELATIONSHIP SETS: (****)

→A **relationship** is an association (connection) among (between) two or more entities.

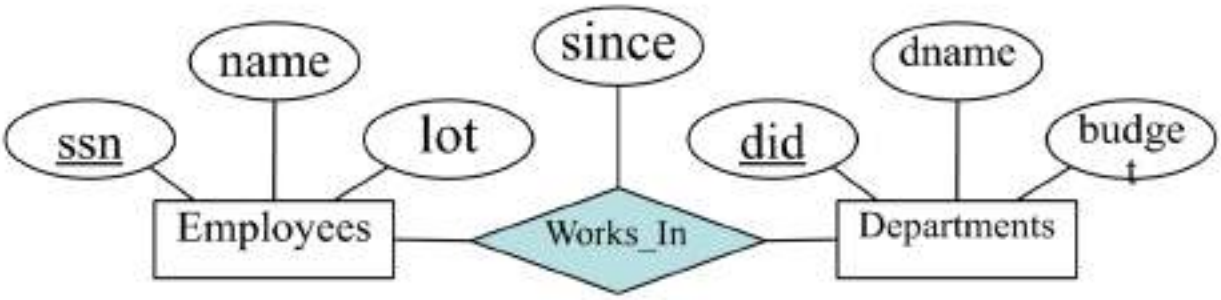


Figure 2.2 The Works_In Relationship Set

Example: We may have the relation Works_In among entities, Employees and Departments i.e., an Employee Works_In a Department.

→A **relationship set** is a collection of similar relationships or we collect a set of similar relationships into a relationship set.
A relationship set can be thought of as a set of n-tuples:

{(e1, ..., en) | e1 ∈ E1, ..., en ∈ En}

Each n-tuple denotes a relationship involving n entities e1 through en, where entity ei is in entity set Ei. Note that several relationship sets might involve the same entity sets. For example, we can also have a Manages relationship set involving Employees and Departments.

A **relationship** can also have **descriptive attributes**. **Descriptive attributes** are used to record information about the relationship, rather than about any one of the participating entities.

Example: In Works_In relationship 'since' attribute captures information about participating entities Employees and Departments.
But, for a given employee-department pair, we cannot have more than one associated 'since' attribute value.

→An instance of a relationship (or) relationship instance set is a set of relationships. An instance can be thought of as a 'snapshot' (a short description) of the relationship set at some instant in time.
An instance of the Works In relationship set is shown in **Figure 2.3.** Each Employees entity is denoted by its **ssn**, and each Departments entity is denoted by its **did**, for simplicity.

The 'since' value is shown beside each relationship as 'many-to-many' relationships and total participation i.e., the employee with ssn (123-22-3666) Works_In did (51) since 1/1/91, similarly the employee with ssn (231-31-5368) Works_In did (51) since 3/3/93 and so on.

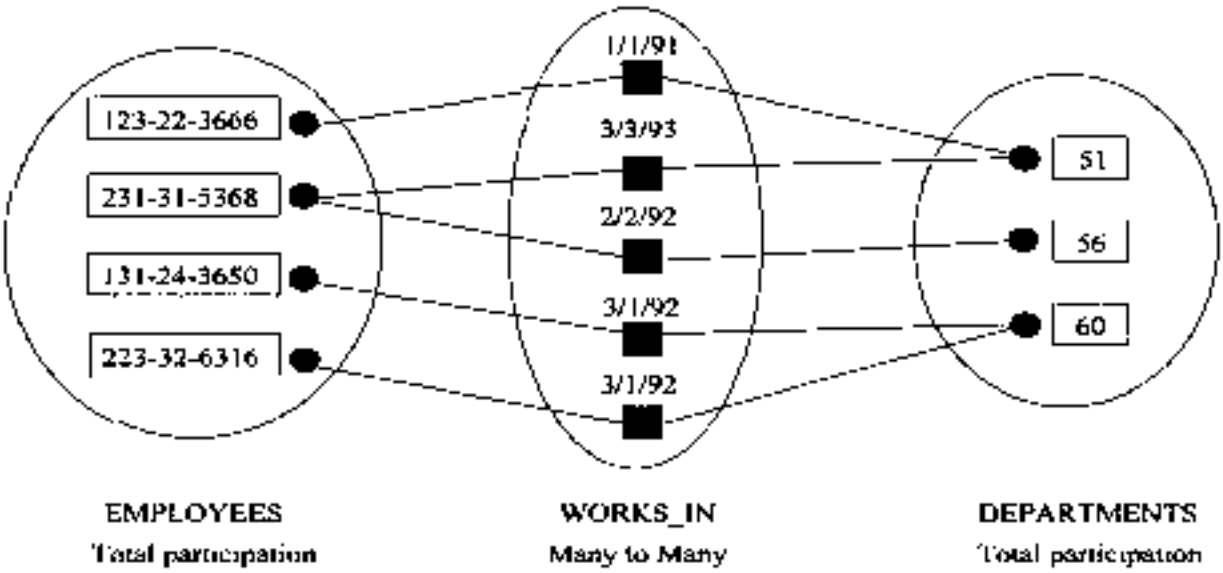


Figure 2.3 An Instance of the Works_In Relationship Set

→**Ternary relationship** is an association (connection) between three entities an employee, a department, and a location.

Example: Each department has offices in several location and we want to record the location at which each employee works.

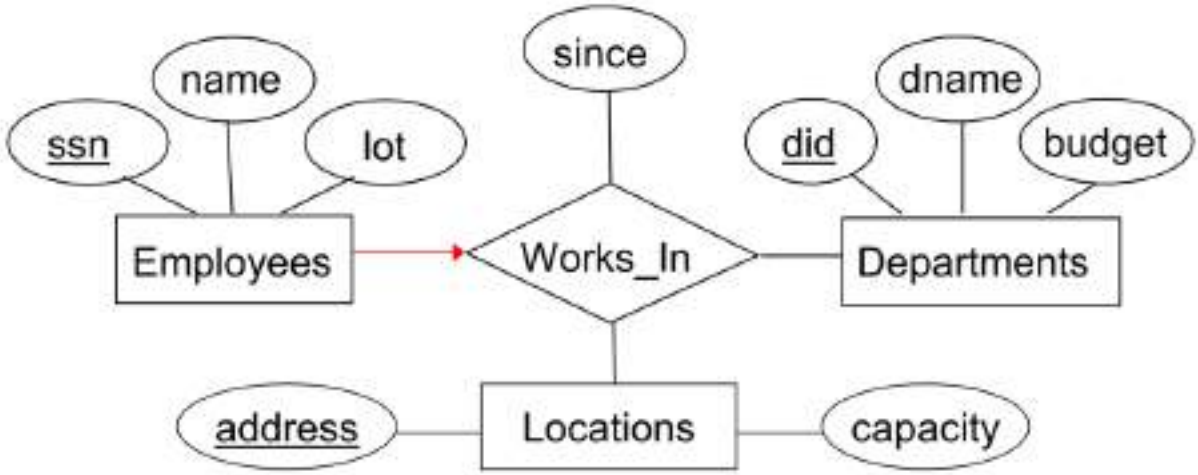


Figure 2.4 A Ternary Relationship set

→The entity sets that participate in a relationship set need not be only one. Sometimes a relationship might involve two entities in the same entity set.

For example, consider the Reports_To relationship set that is shown in **Figure 2.5**. Since employees report to other employees, every relationship in Reports_To is of the form (emp1, emp2), where both emp1 and emp2 are entities in Employees.

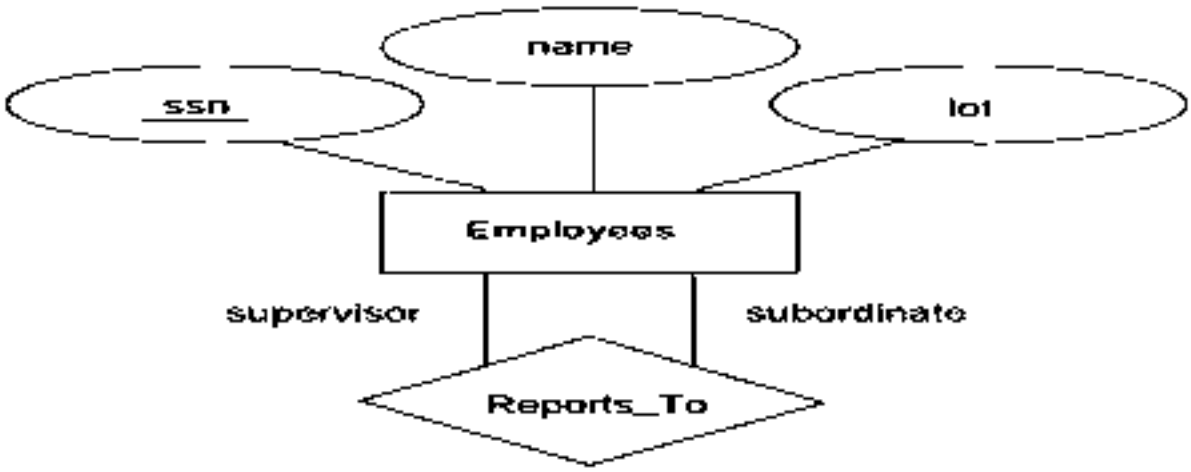


Figure 2.5 The Reports_To Relationship set

However, they play different roles: emp1 reports to the managing employee emp2, which is reflected in the role indicators supervisor and subordinate in **Figure 2.5.**

If an entity set plays more than one role, the role indicator concatenated with an attribute name from the entity set gives us a unique name for each attribute in the relationship set. For example, the Reports To relationship set has attributes corresponding to the ssn of the supervisor and the ssn of the subordinate, and the names of these attributes are supervisor ssn and subordinate ssn.

4. ADDITIONAL FEATURES OF ER MODEL:

→Following constructs are the features in the ER Model that allows us to describe some common properties of the data in expressing ER Model.

Key Constraints: (*********)

Consider the Works_In relationship shown in **Figure 2.2.**

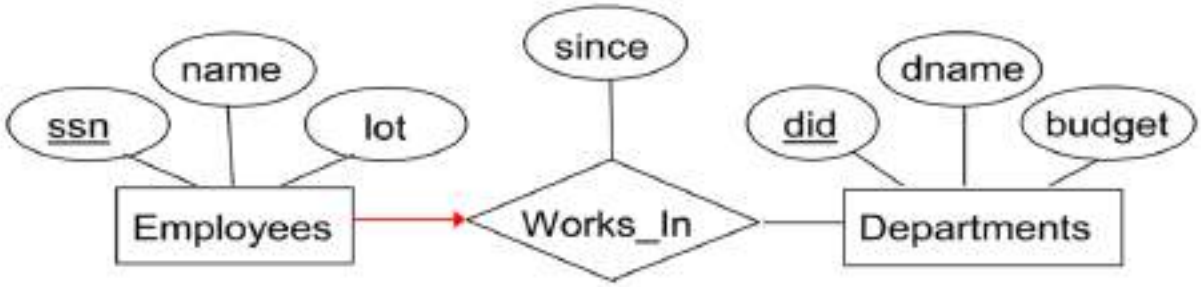


Figure 2.2 The Works_In Relationship Set

An employee can work in several departments, and a department can have several employees, as illustrated in the Works_In instance shown in **Figure 2.3.**

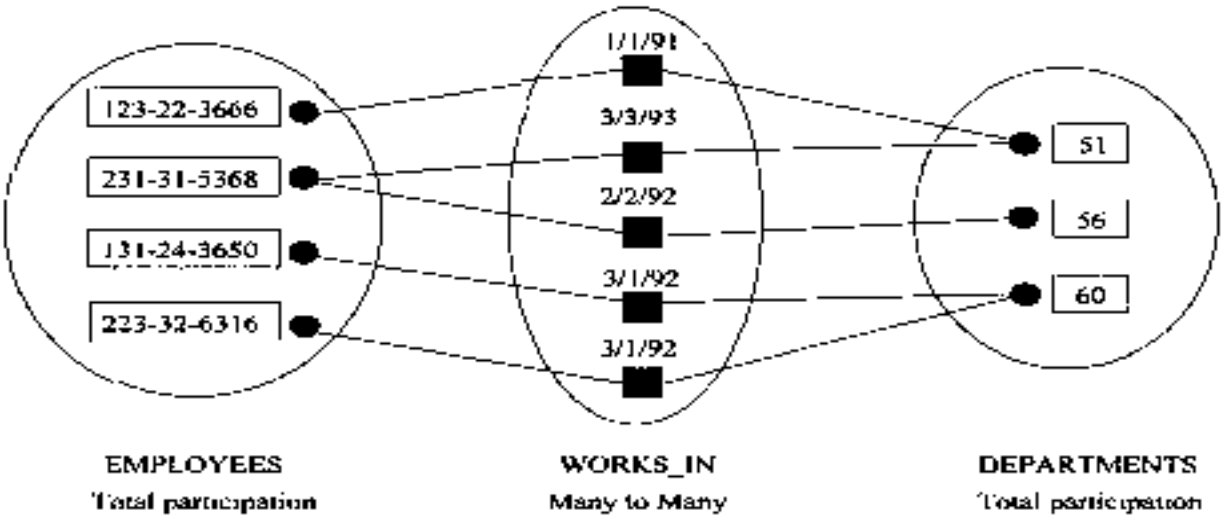


Figure 2.3 An Instance of the Works_In Relationship Set

Here, Employee 231-31-5368 has worked in Department 51 since 3/3/93 and in Department 56 since 2/2/92. Department 51 has two employees. Thus one department can have many employees.

But, if we want to have only one employee in department, then it is an example of **Kay constraint.**

Example: Consider another relationship set called Manages between the Employees and Departments entity sets as in the **Figure 2.6**.

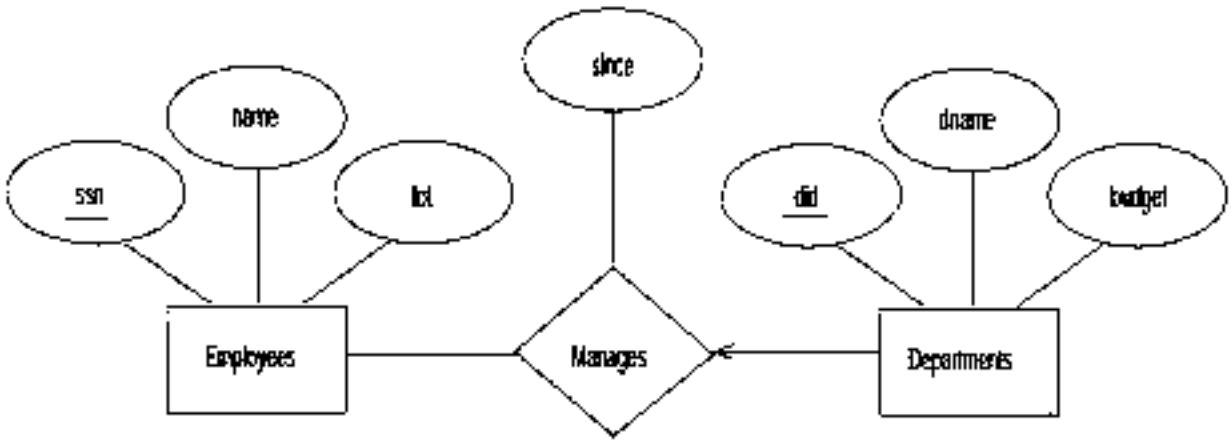


Figure 2.6 Key Constraint on Manages

Here, each department can have only one manager. The restriction that each department can have only one manager is an example of key constraints. This restriction is indicated in the above ER diagram by using an arrow from departments to manages, such that a department can have only one manager.

An instance of the Manages relationship set is shown in **Figure 2.7.** While this is also a potential instance for the Works In relationship set, the instance of Works In shown in Figure 2.3 violates the key constraint on Manages.

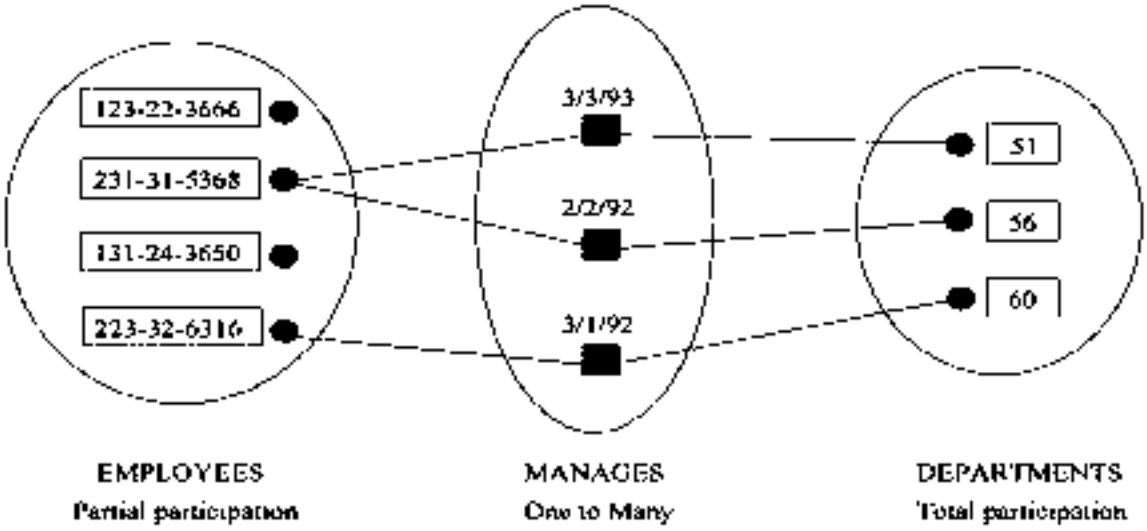


Figure 2.7 An Instance of the Manages Relationship Set

Key Constraints for Ternary Relationships:

In **Figure 2.8**, we show a ternary relationship with key constraints. Each employee works in at most one department, and at a single location.

An instance of the Works_In3 relationship set is shown in **Figure 2.9**. Notice that each department can be associated with several employees and locations, and each location can be associated with several departments and employees; however, each employee is associated with a single department and location.

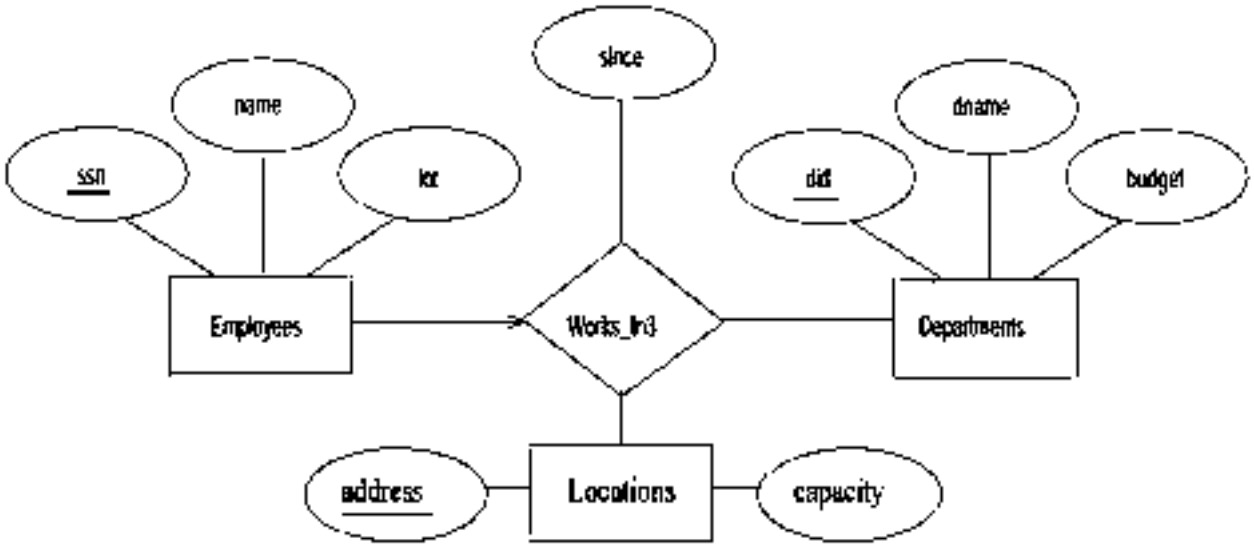


Figure 2.8 A Ternary Relationship Set with Key Constraints

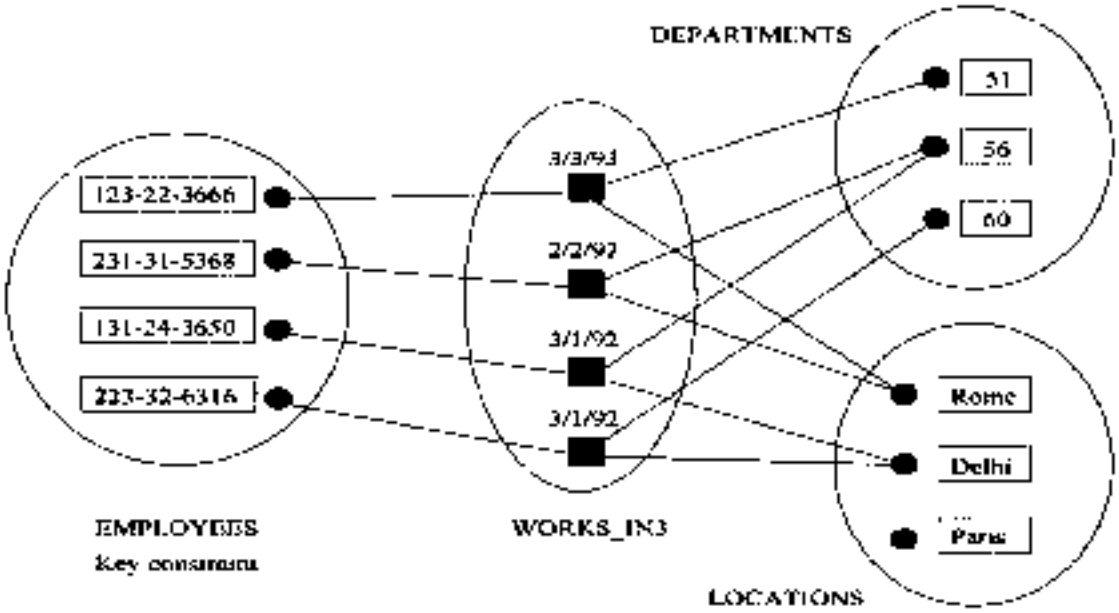


Figure 2.9 An Instance of Works_In3

Participation Constraints: (**********)

The ER diagram in **Figure 2.10** shows both the Manages and Works_In relationship sets and all the given constraints. If the participation of an entity set in a relationship set is total, the two are connected by a thick line; independently, the presence of an arrow indicates a key constraint. The instances of Works_In and Manages shown in Figures 2.3 and 2.7 satisfy all the constraints in **Figure 2.10.**

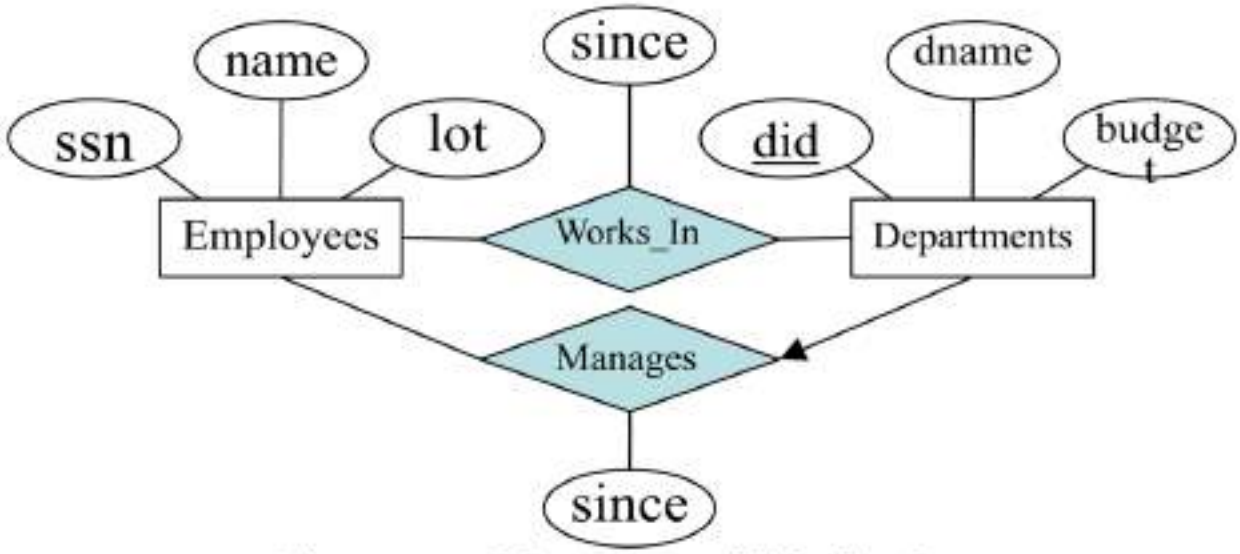


Figure 2.10 Manages and Works_In

Weak Entities:

An entity set attributes that does not have a primary key within them, is termed as a **weak entity set**. As an example, consider the entity set Dependents, which has the two attributes pname and age, illustrated with the ER diagram as shown in **Figure 2.11**

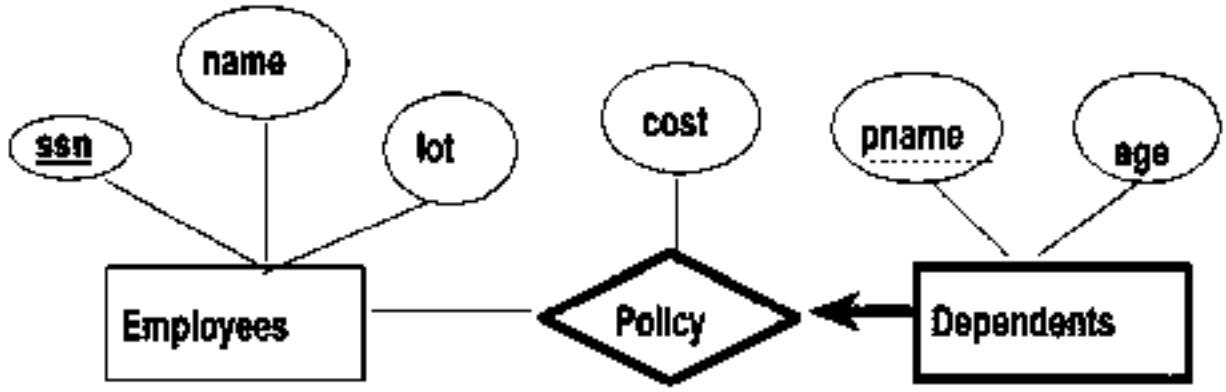


Figure 2.11 A Weak Entity Set

A dependent is an example of a weak entity set. A weak entity can be identified uniquely only by considering some of its attributes in conjunction with the primary key of another entity, which is called the identifying owner.

The following restrictions must hold:
→The owner entity set and the weak entity set must participate in a one-to-many relationship set (one owner entity is associated with one or more weak entities, but each weak entity has a single owner).
→This relationship set is called the identifying relationship set of the weak entity set. The weak entity set must have total participation in the identifying relationship set.

The Dependents weak entity set and its relationship to Employees is shown in **Figure 2.11.** The total participation of Dependents in Policy is indicated by linking them with a dark line. The arrow from Dependents to Policy indicates that each Dependents entity appears in at most one Policy relationship. To underscore the fact that Dependents is a weak entity and Policy is its identifying relationship, we draw both with dark lines. To indicate that pname is a partial key for Dependents, we underline it using a broken line. This means that there may well be two dependents with the same pname value.

Class Hierarchies:
To classify the entities in an entity set into subclass entity is known as **class hierarchies.**
Example: we might want to classify Employees entity set into subclass entities Hourly_Emps entity set and a Contract _Emps entity set to distinguish the basis on which they are paid. Then the class hierarchy is illustrated as shown in **Figure 2.12.**

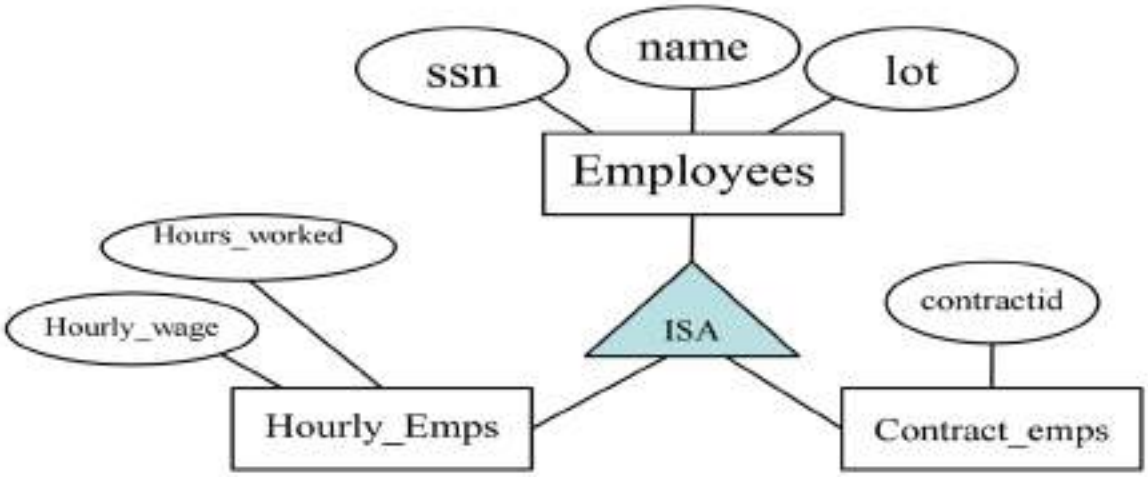


Figure 2.12 Class hierarchy

This **class hierarchy** illustrates the inheritance concept. Where, the subclass attributes ISA (read as: is a) superclass attributes, indicating the "is a" relationship (inheritance concept).
Therefore, the attributes defined for a Hourly_Emps entity set are the attributes of Hourly_Emps plus attributes of employees (because subclass can have superclass properties). Likewise the attributes defined for a Contract_Emps set are the attributes of Contract_Emps plus attributes of Employees.

→A **class hierarchy** can be viewed in one of two ways:

Specialization:

→An **employee is** specialized into subclasses. Specialization is the process of identifying subsets (subclasses) of an entity set (the superclass) that share some distinguishing characteristics. Here, the superclass (Employees) is defined first, the subclasses (Hourly_Emps, Contract_Emps etc.) are defined next and subclass-specific attributes and relationship sets are then added.

Generalization:

→Generalization is the process of identifying(defining) some generalized (common) characteristics of a collection of (two or more) entity sets and creating a new entity set that contains entities possessing these common characteristics. Here, the subclasses (Hourly_Emps, Contract_Emps, etc.,) are defined first the superclass (Employees) is defined next.
In shortly, Hourly_Emps and Contract_Emps are generalized by Employees.

→The **class hierarchy** can specify two kinds of constraints. They are

Overlapped Constraints:

Overlap constraints determine whether two subclasses are allowed to contain the same entity.
Example: can Akbar be both a Hourly_Emps entity and a Contract_Emps entity? The answer is no.
Other example, can Akbar be both a Contract_Emps entity and a Senior_Emps entity (among them) the answer is, Yes.
Thus, this is a specialization hierarchy property. We denote this by writing "Contract_Emps overlaps Senior_Emps".

Covering Constraints:

Covering constraints determine whether the entities in the subclasses collectively include all entities in the superclass.

Example: Should every Employees entity be a Hourly_Emps or Contract_Emps? The answer is, No. He can be a Daily_Emps.
→Other example, should every Motor_Vehicle (superclass) be a bike (subclass) or a car (subclass)? The answer is yes.
Thus generalization hierarchies' property is that every instance of a superclass is an instance of a subclass.
We denote this by writing "bikes and cars cover Motor_Vehicles"

AGGREGATION: (**********)

→Used when we have to model a relationship involving (entity sets and) a relationship set.
→**Aggregation** allows us to indicate that a relationship set (identified through a dashed box) participates in another relationship set.
→**Aggregation** allows a relationship set to be treated as an entity set for purposes of participation in (other) relationship sets.
This is illustrated in **Figure**, with a dashed box around **Sponsors** (and its participating entity sets) used to denote aggregation. This effectively allows us to treat Sponsors as an entity set for purposes of defining the **Monitors** relationship set.

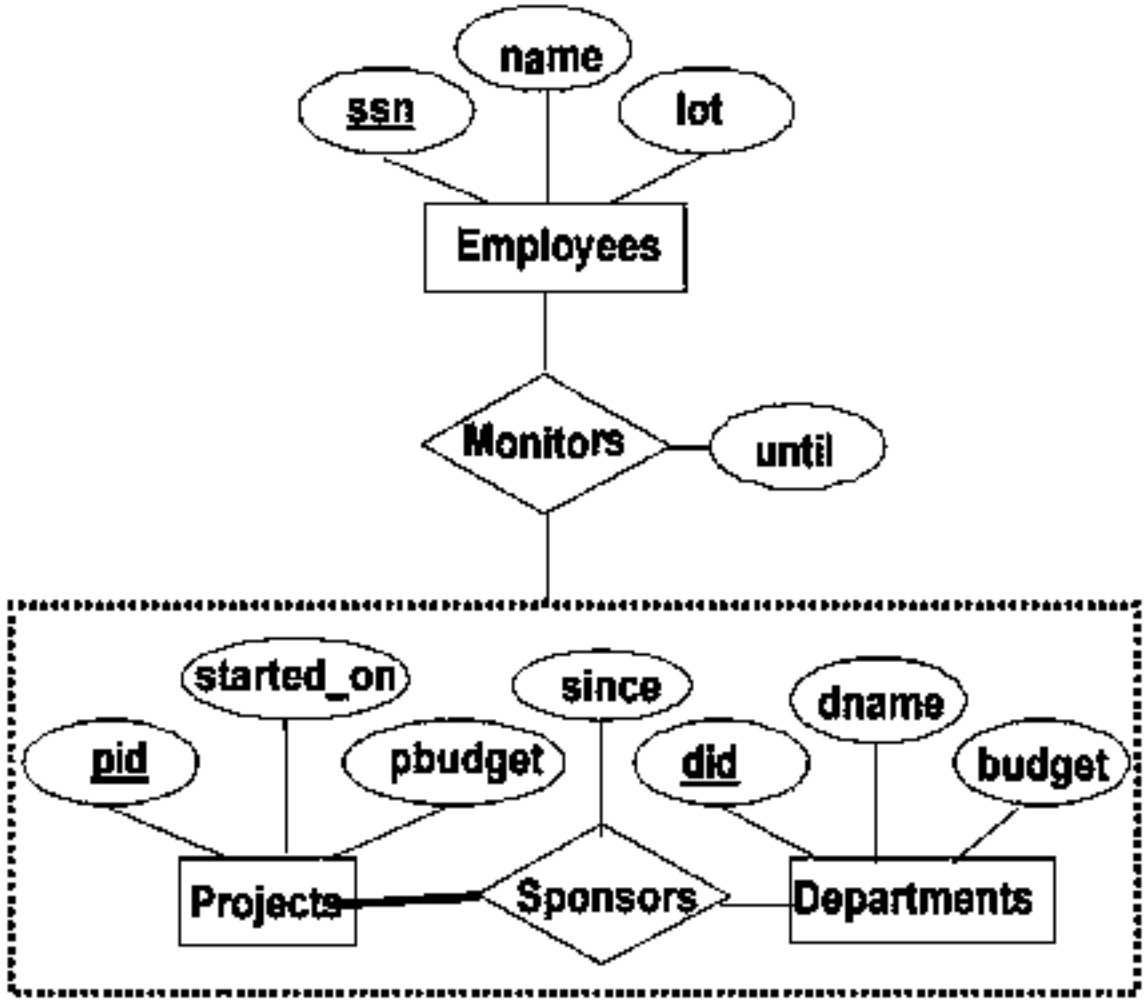
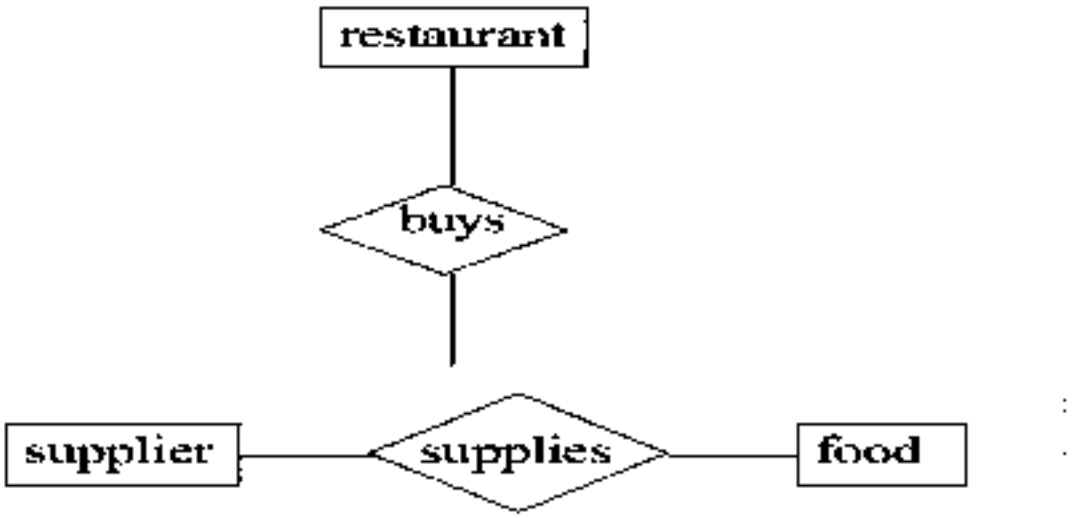


Figure. Aggregation

Restaurant Example:



Note:
→For Ternary relationship, it can only see which specific restaurant buys what kind food from which supplier.
→For Aggregation, you have more information about which supplier supplies a food item. Any restaurant needs that item can choose from that list.

Uses of Aggregation:
We use an aggregation, when we need to express a relationship among relationships. Thus, there are really two distinct relationships, sponsors and monitors, each with its own attributes.
Example: The Monitors relationship has an attribute until that records the ending date until when the employee is appointed as the sponsorship monitoring.

5. CONCEPTUAL DATABASEDESIGN WITH ER- DIAGRAMS:

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

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

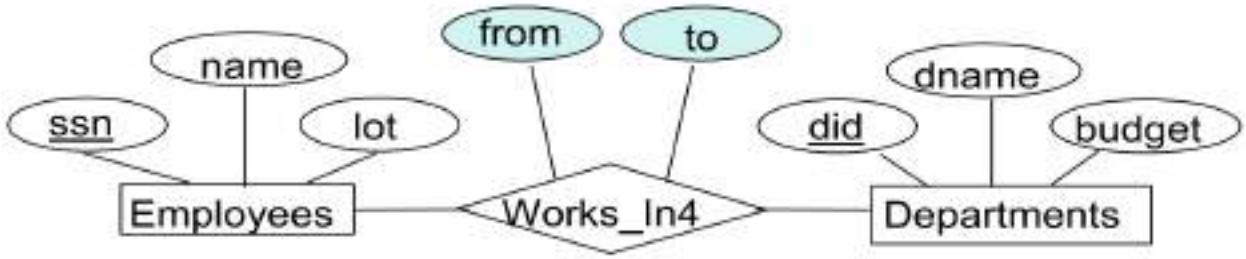
Entity Vs Attributes:
While identifying the attributes of an entity set, it is sometimes not clear, whether a property should be modeled as an attribute or as an entity set.

→Should **address** be an attribute of Employees or an entity (connected to Employees by a relationship)?
→Depends upon the use we want to make of address information, and the semantics of the data:

- If we have several addresses per employee, address must be an entity (since attributes cannot be set-valued).
- If the structure (city, street, etc.,) is important, e.g., we want to retrieve employees in a given city, address must be modeled as an entity (since attributes values are atomic).

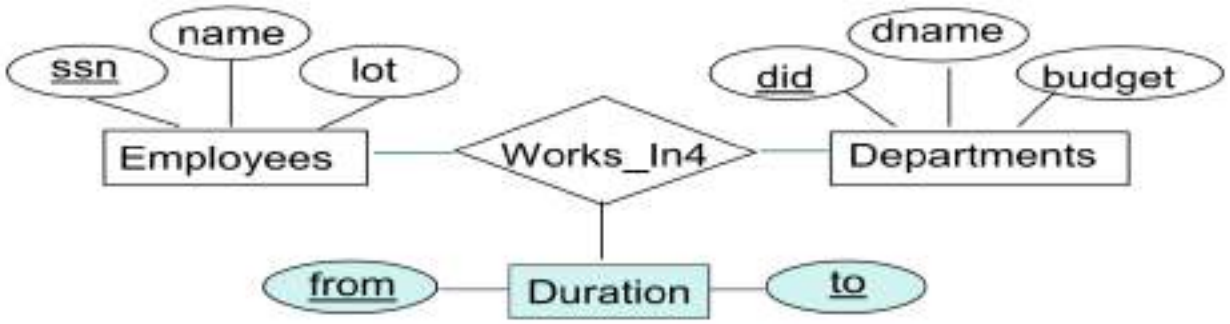
- **Works-In4** does not allow an employee to work in a department for two or more periods. A relationship is uniquely identified by the participating entities.

$$R = \{(e_1, \ldots, e_n) \mid e_1 \in E_1, \ldots, e_n \in E_n\}$$



- Similar to the problem of wanting to record several working periods for an employee in Work_In4. We want to record **several values of the descriptive attributes for each instance of this relationship.** Accomplished by introducing new entity set**, Duration.**

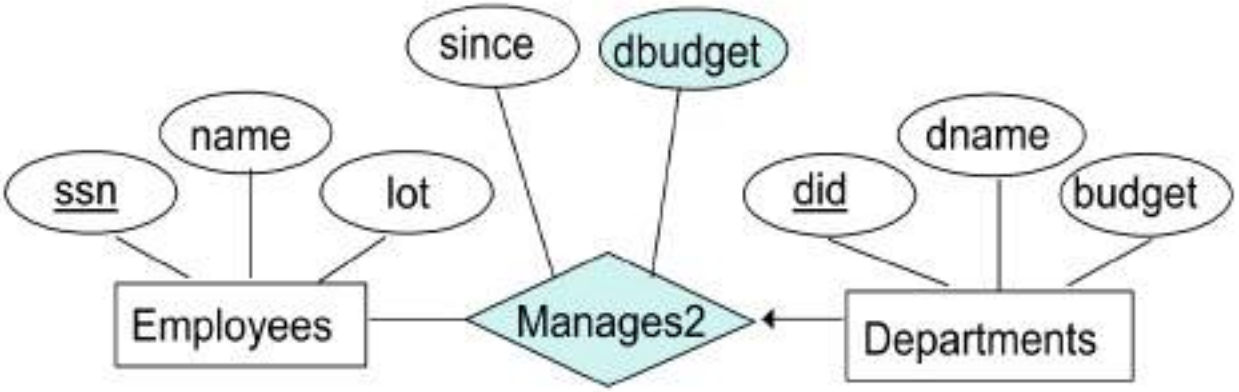
.



Entity Vs Relationship:

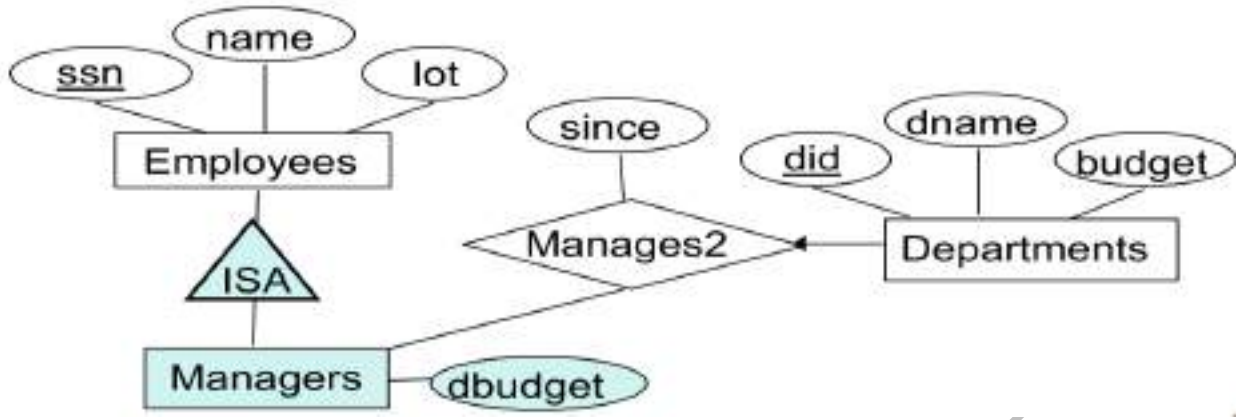
It is not always clear whether an object is best expressed by an entity set or a relationship set.

Example: If a manager gets a separate discretionary budget (dbudget) for each department he or she manages.



What if a manager gets a discretionary budget that covers all managed departments?

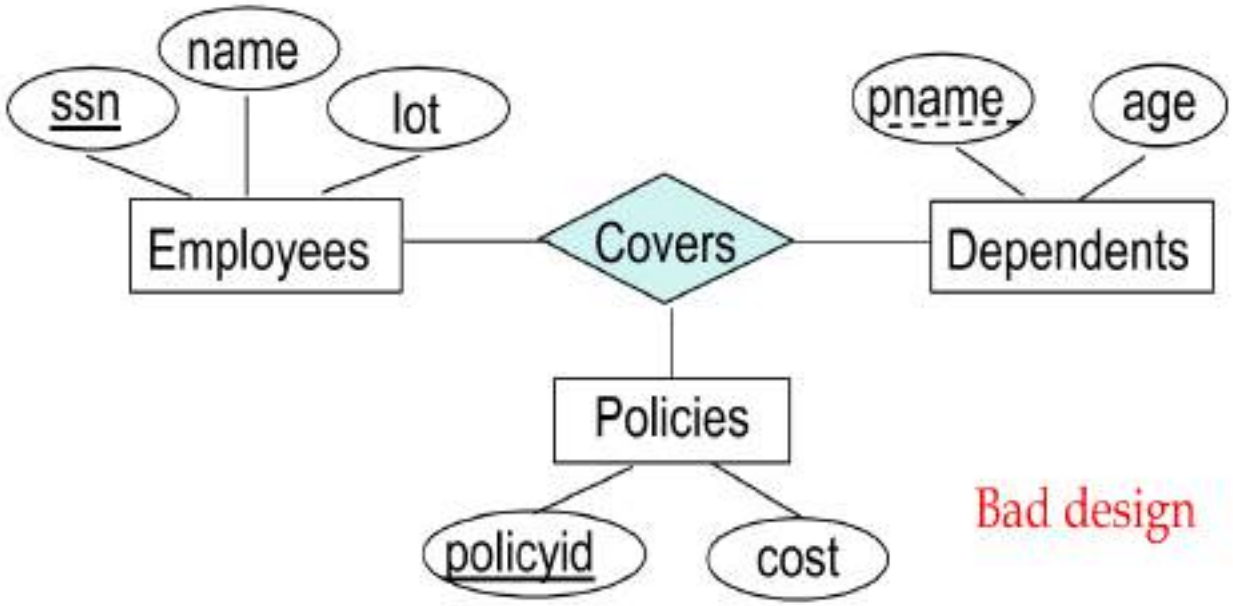
- **Redundancy:**dbudget stored for each department managed by the manager.
- **Misleading:** suggests **dbudge**t is associated with department – manager combination.



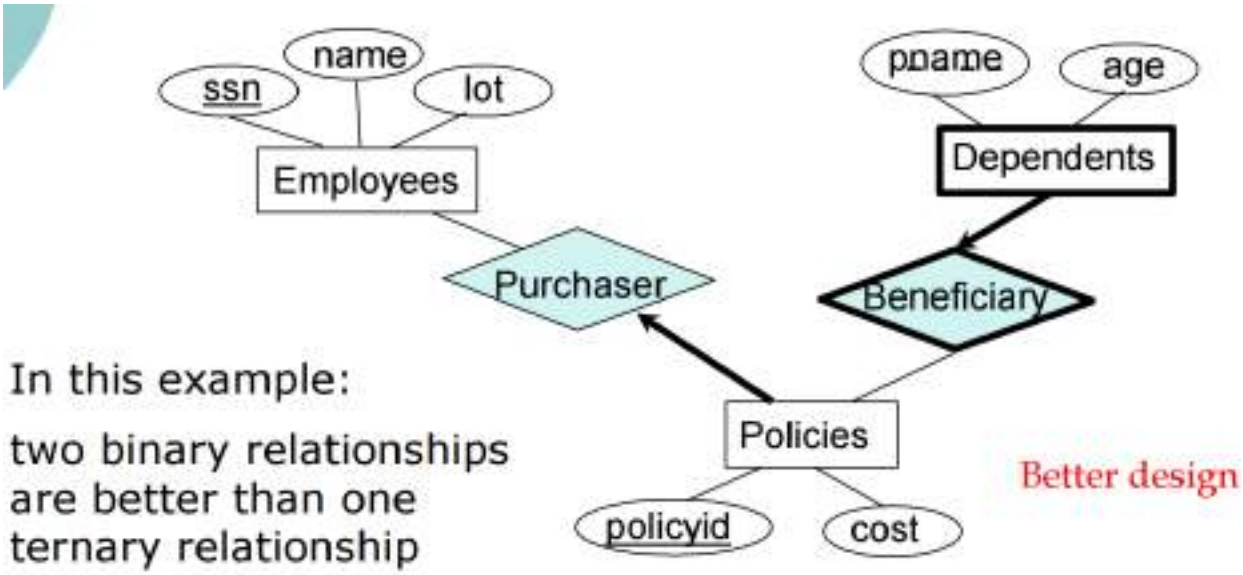
Binary versus Ternary Relationships:
It is always possible to replace a non-binary (n-array, for n>2) relationship set by a number of distinct binary relationship sets.

- A **Bad design** below if:
  Each policy is owned by just 1 employee, and, Dependents is a weak entity set, and each dependent is tied to the covering policy.

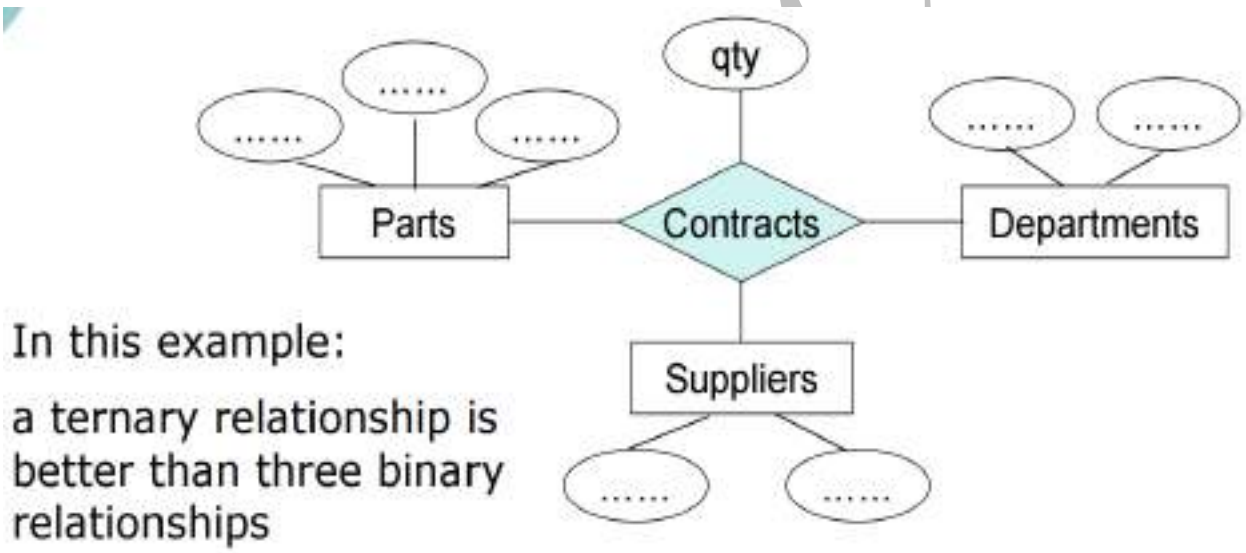
Bad design: Policies involves in two relationships.



What are the additional constraints in 2nd diagram?
Better design:



Another example: The contract specifiesthat a supplier will supplysome quantity of a part to a department.



Aggregation Vs Ternary Relationships:

→The choice between using aggregation or a ternary relationship is mainly determined by the existence of a relationship that relates a relationship set to an entity set (or second relationship set). The choice may also be guided by certain integrity constraints that we want to express.
→The**monitors** is a distinct relationship, with a descriptive attribute. (In Figure)

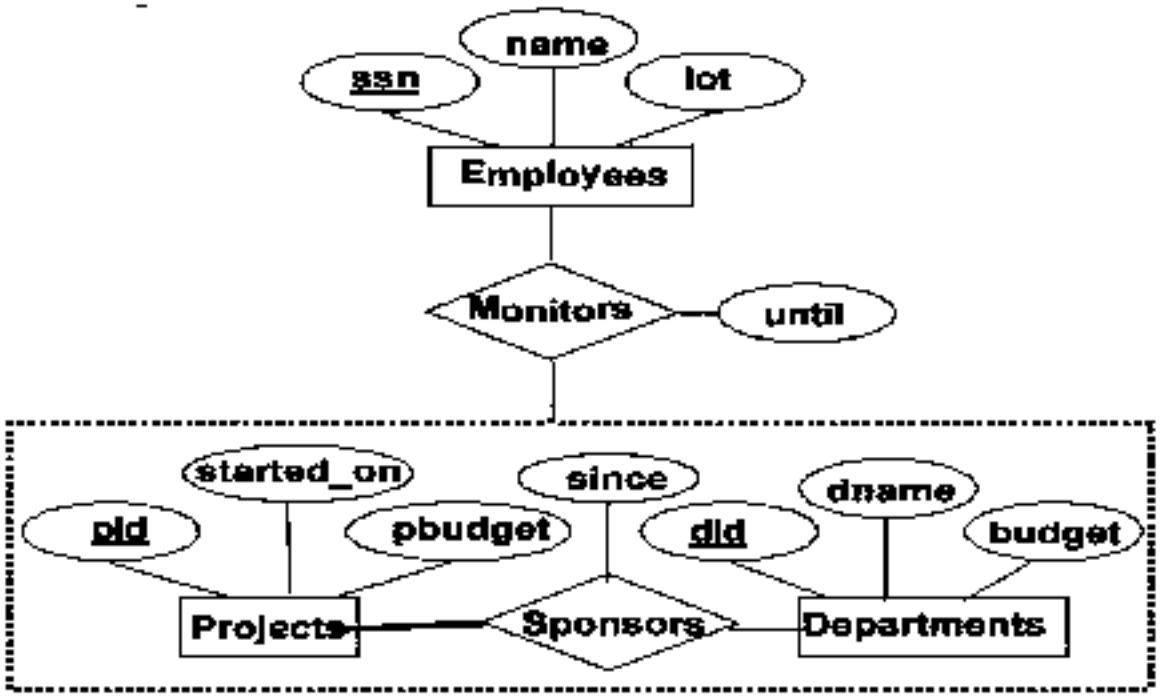


Figure. Aggregation

→If we don't need to record the ***until*** attribute of Monitors, then we might reasonably use a **ternary relationship. (In Figure)**

→Also, it can say that each sponsorship is monitored by at most one employee.

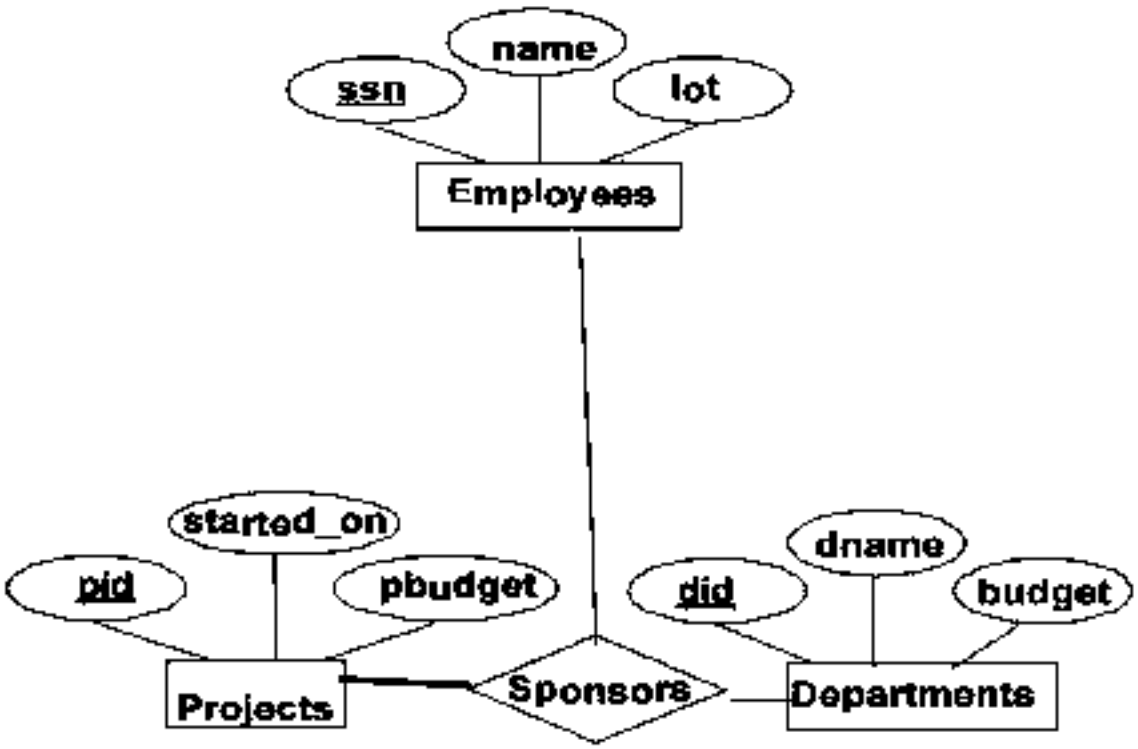


Figure. Using Ternary Relationship Instead of Aggregation.

ER DIAGRAMS EXAMPLES

1) Entity Relationship (ER) Modeling - Learn with a Complete Example

Here we are going to design an Entity Relationship (ER) model for a **college database.** Say we have the following statements.

1. **A college contains many departments**
2. **Each department can offer any number of courses**
3. **Many instructors can work in a department**
4. **An instructor can work only in one department**
5. **For each department there is a Head**
6. **An instructor can be head of only one department**
7. **Each instructor can take any number of courses**
8. **A course can be taken by only one instructor**
9. **A student can enroll for any number of courses**
10. **Each course can have any number of students**

Step 1: Identify the Entities

What are the entities here?

From the statements given, the entities are

1. Department
2. Course
3. Instructor
4. Student

Step 2: Identify the relationships

1. One department offers many courses. But one particular course can be offered by only one department. hence the cardinality between department and course is One to Many (1:N)
2. One department has multiple instructors. But instructor belongs to only one department. Hence the cardinality between department and instructor is One to Many (1:N)
3. One department has only one head and one head can be the head of only one department. Hence the cardinality is one to one. (1:1)
4. One course can be enrolled by many students and one student can enroll for many courses. Hence the cardinality between course and student is Many to Many (M:N)
5. One course is taught by only one instructor. But one instructor teaches many courses. Hence the cardinality between course and instructor is Many to One (N :1)

Step 3: Identify the key attributes

- "**Departmen_Name**" can identify a department uniquely. Hence Department_Name is the key attribute for the Entity "Department".
- **Course_ID** is the key attribute for "Course" Entity.
- **Student_ID** is the key attribute for "Student" Entity.
- **Instructor_ID** is the key attribute for "Instructor" Entity.

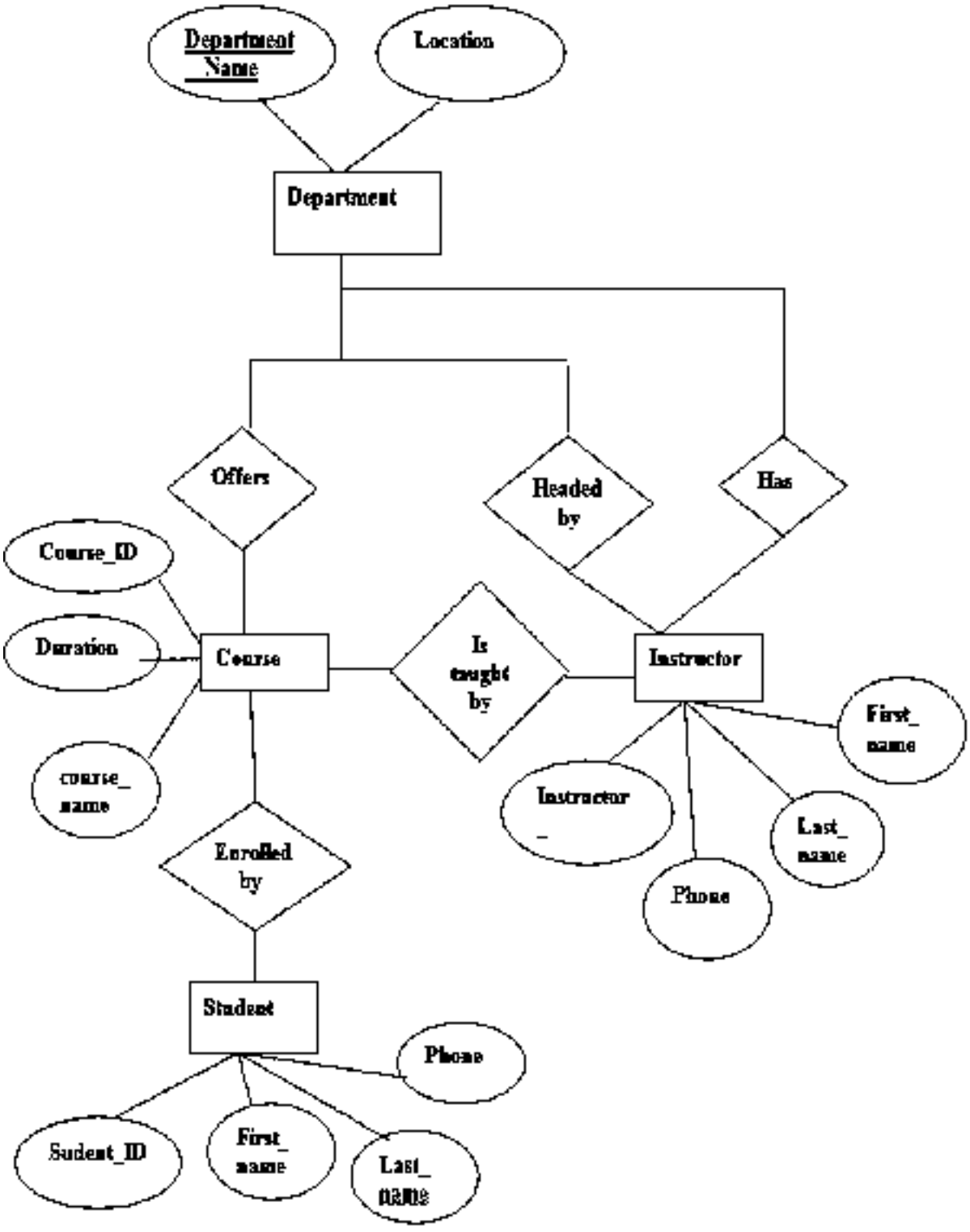
Step 4: Identify other relevant attributes

- For the department entity, other attributes are location

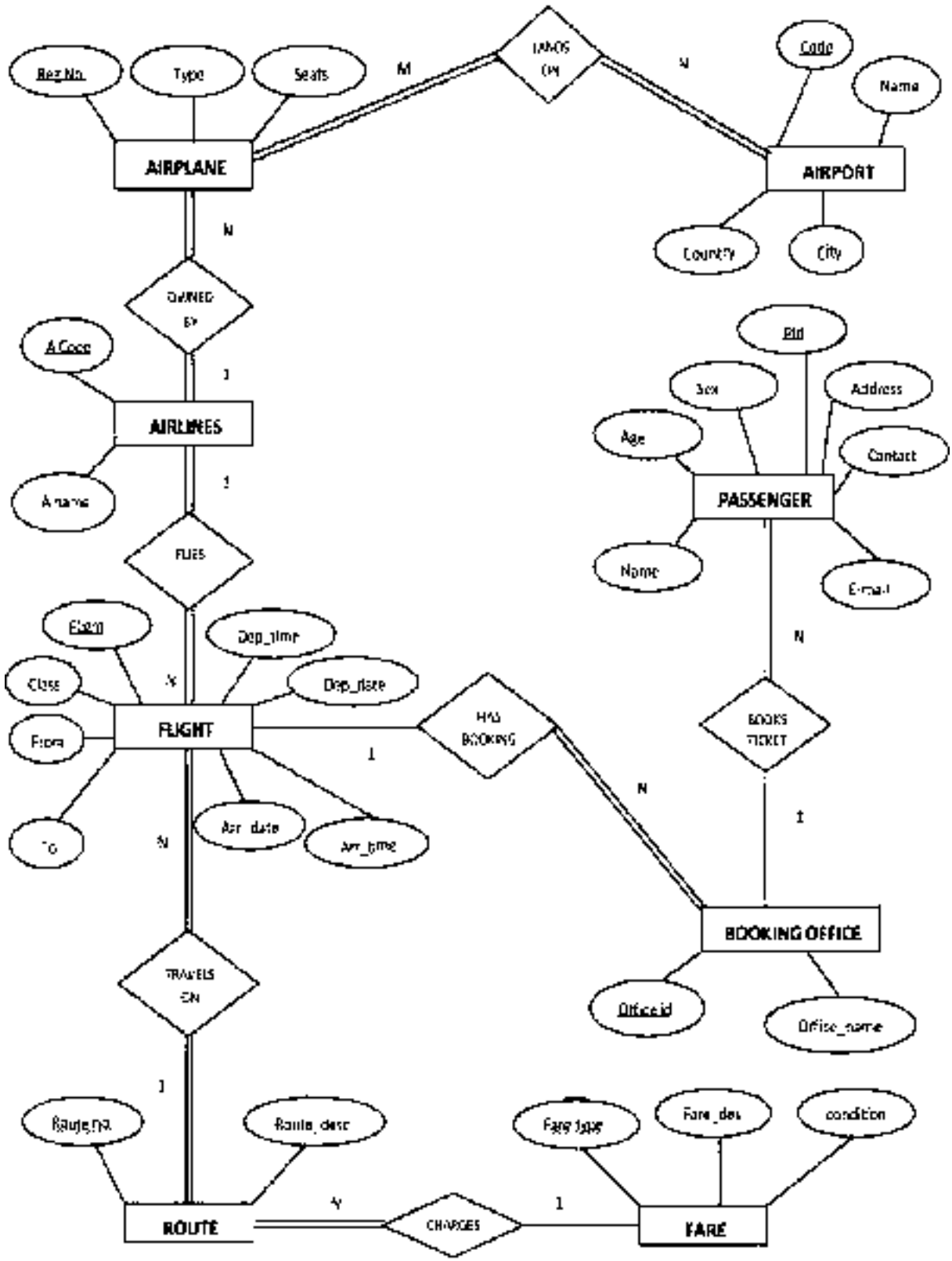
- For course entity, other attributes are course_name, duration
- For instructor entity, other attributes are first_name, last_name, phone
- For student entity, first_name, last_name, phone

Step 5: Draw complete ER diagram
By connecting all these details, we can now draw ER diagram as given below.

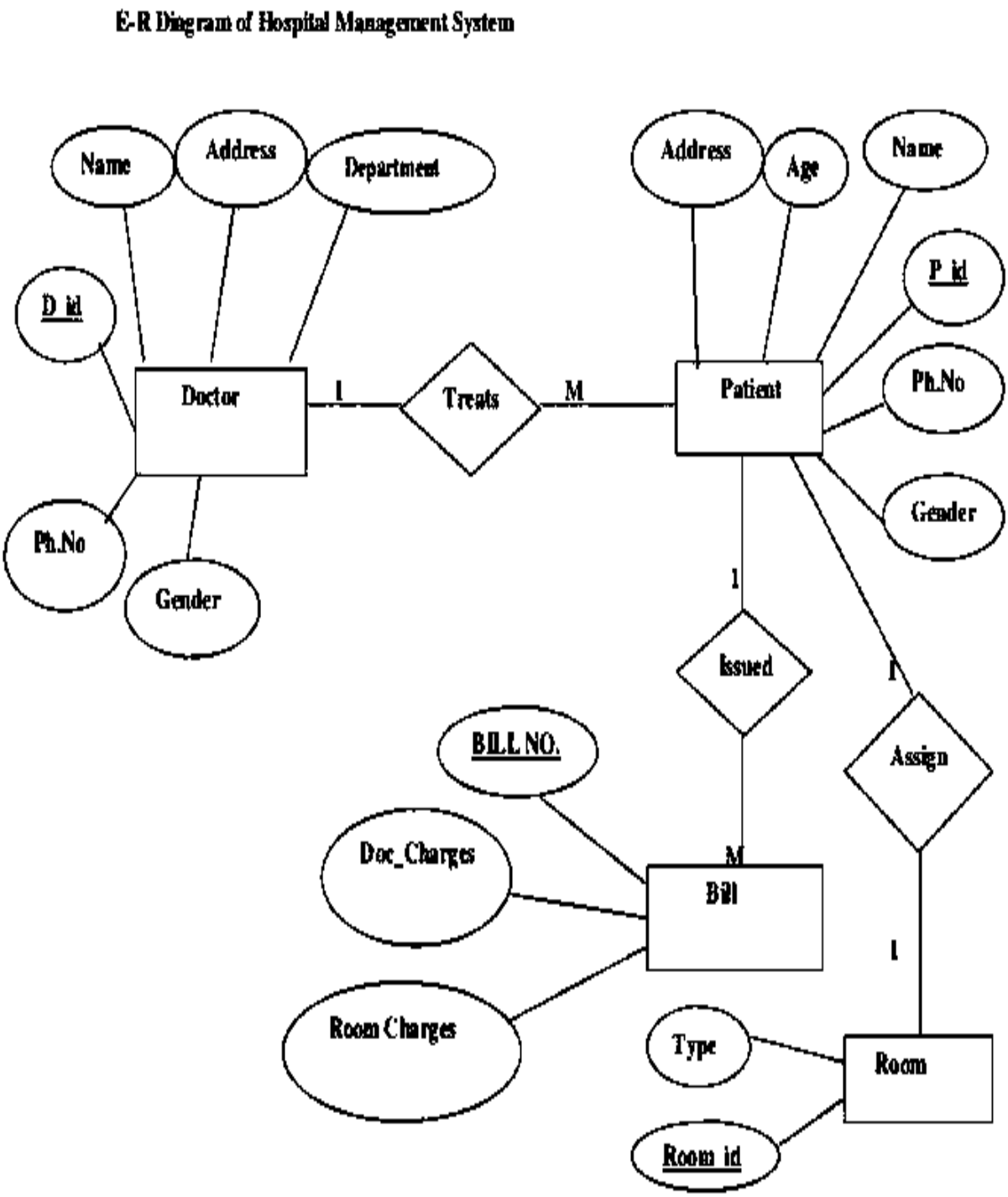
2) ER DIGRAM FOR COLLEGE DATABSAE



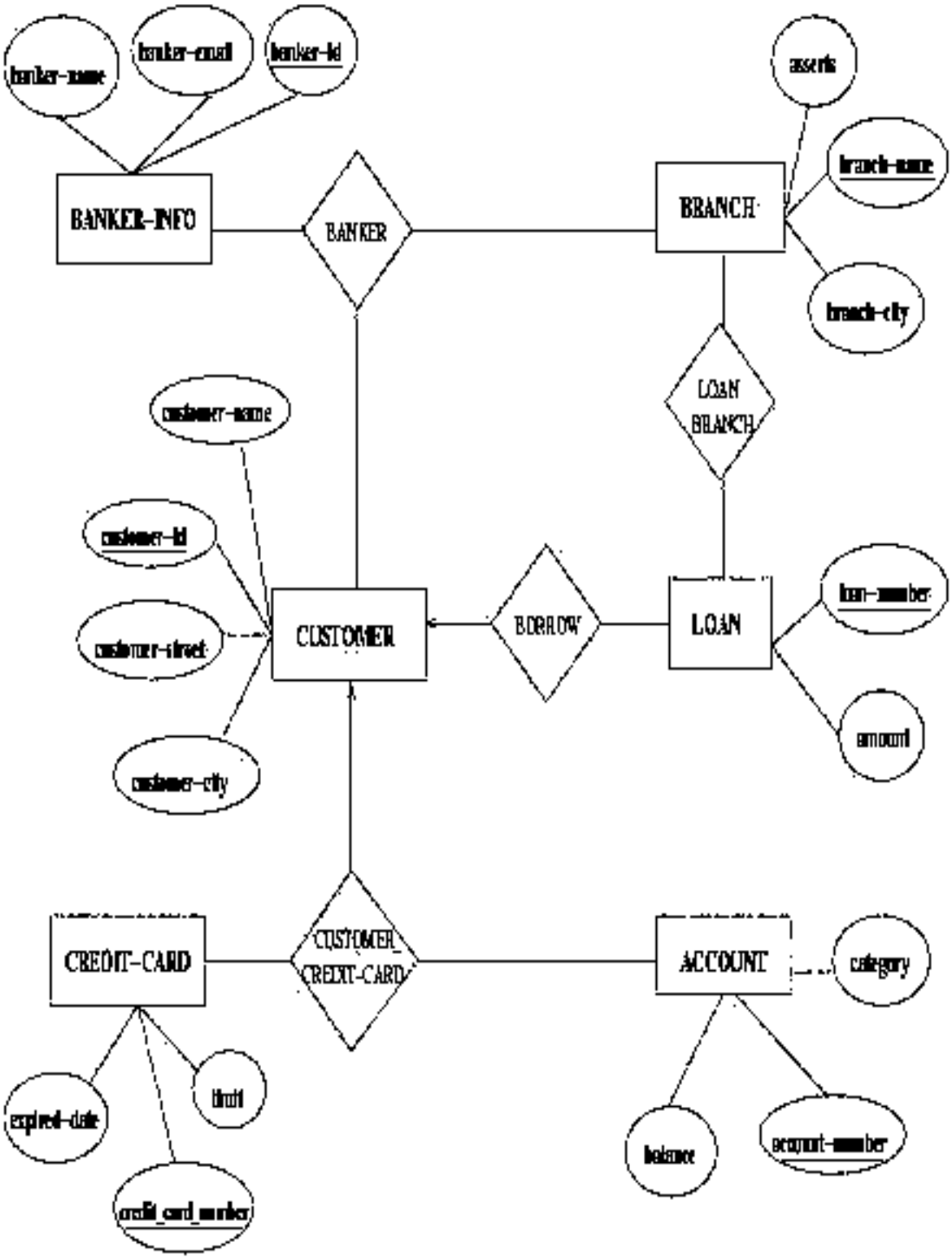
3) ER DIGRAM FOR AIRLINES - RESERVATION SYSTEM



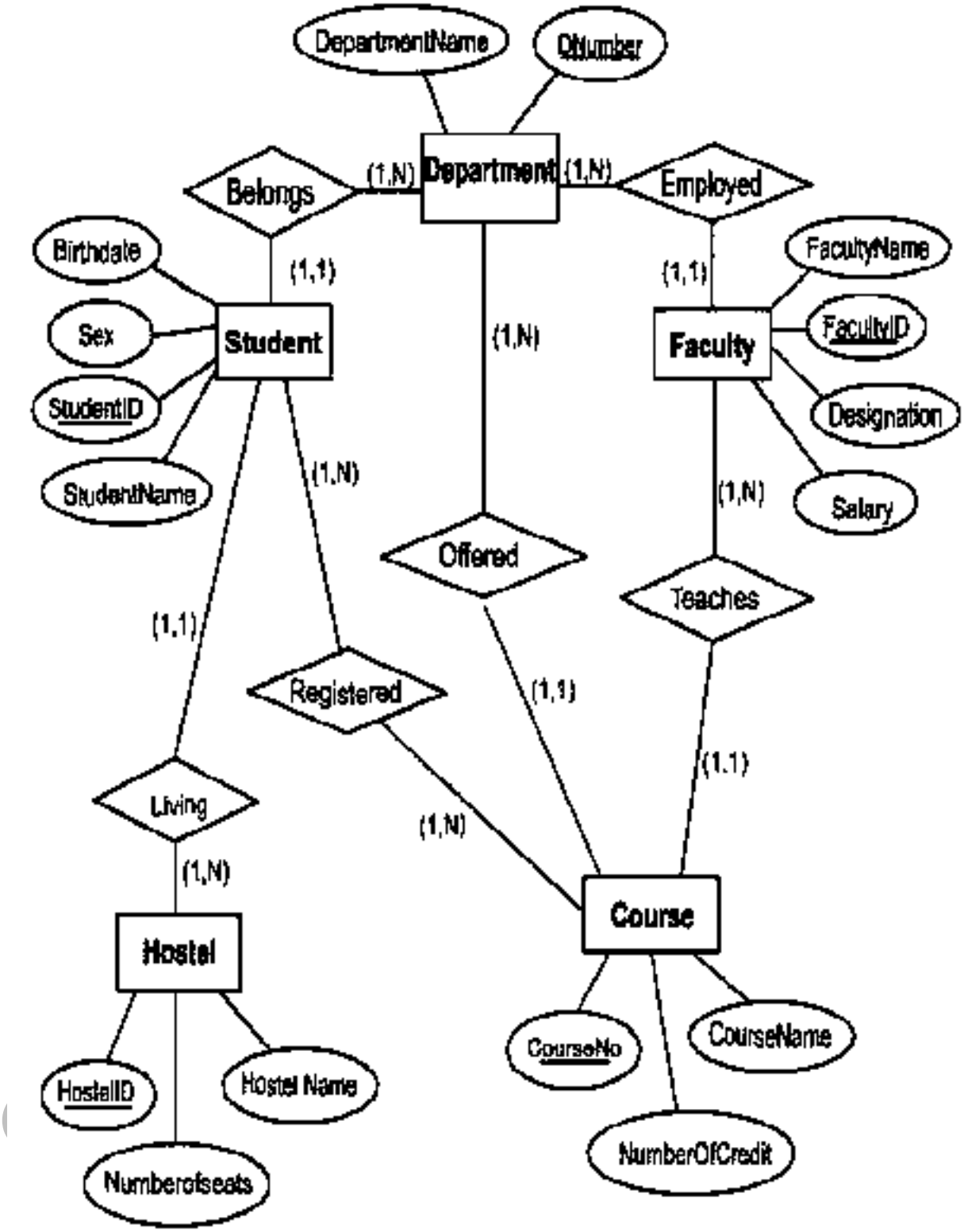
4) ER Diagram for Hospital Management System:



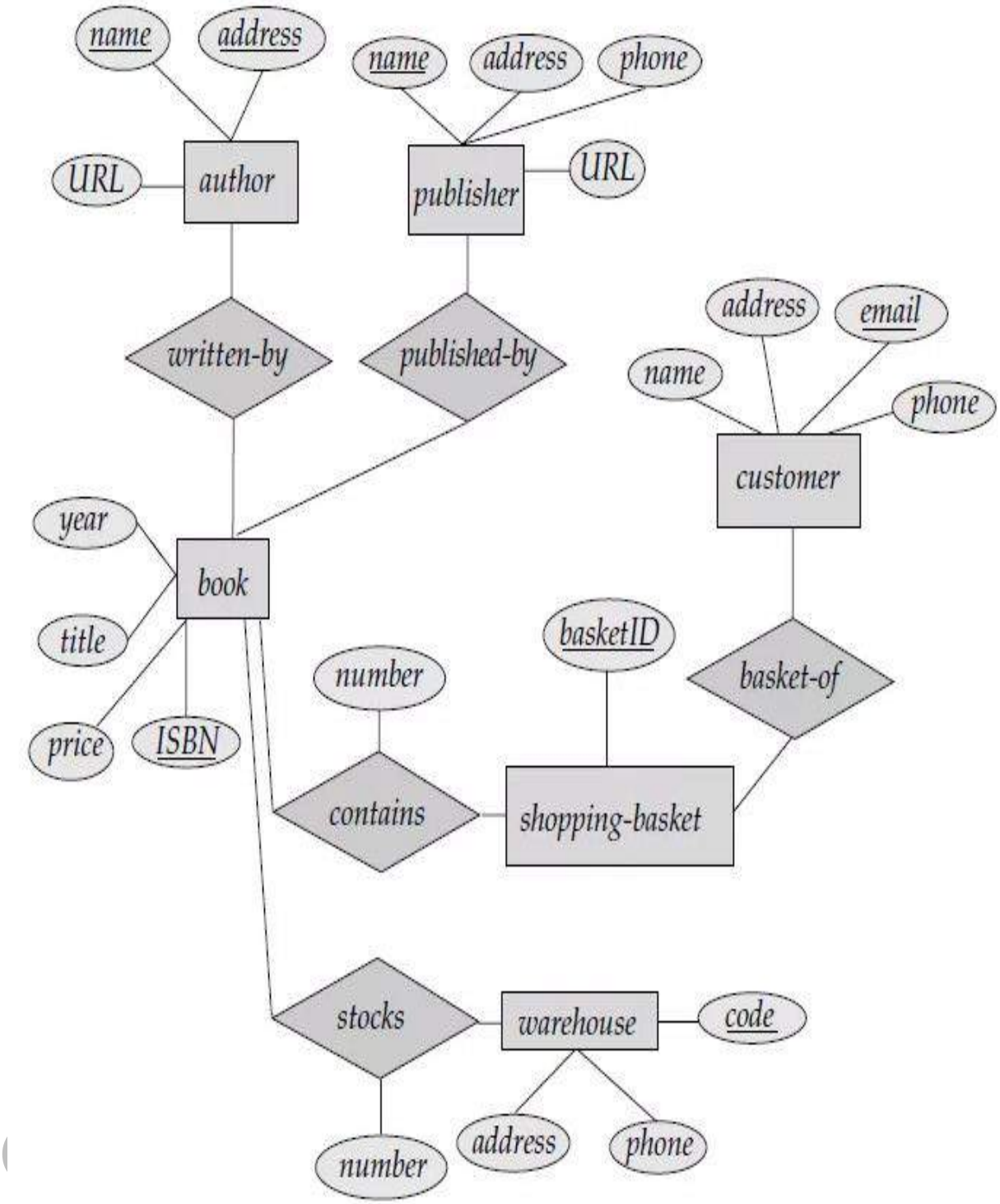
5) ER Diagram for Banking System



6) ER Diagram for College management system



7) ER Diagram for Online Book Store

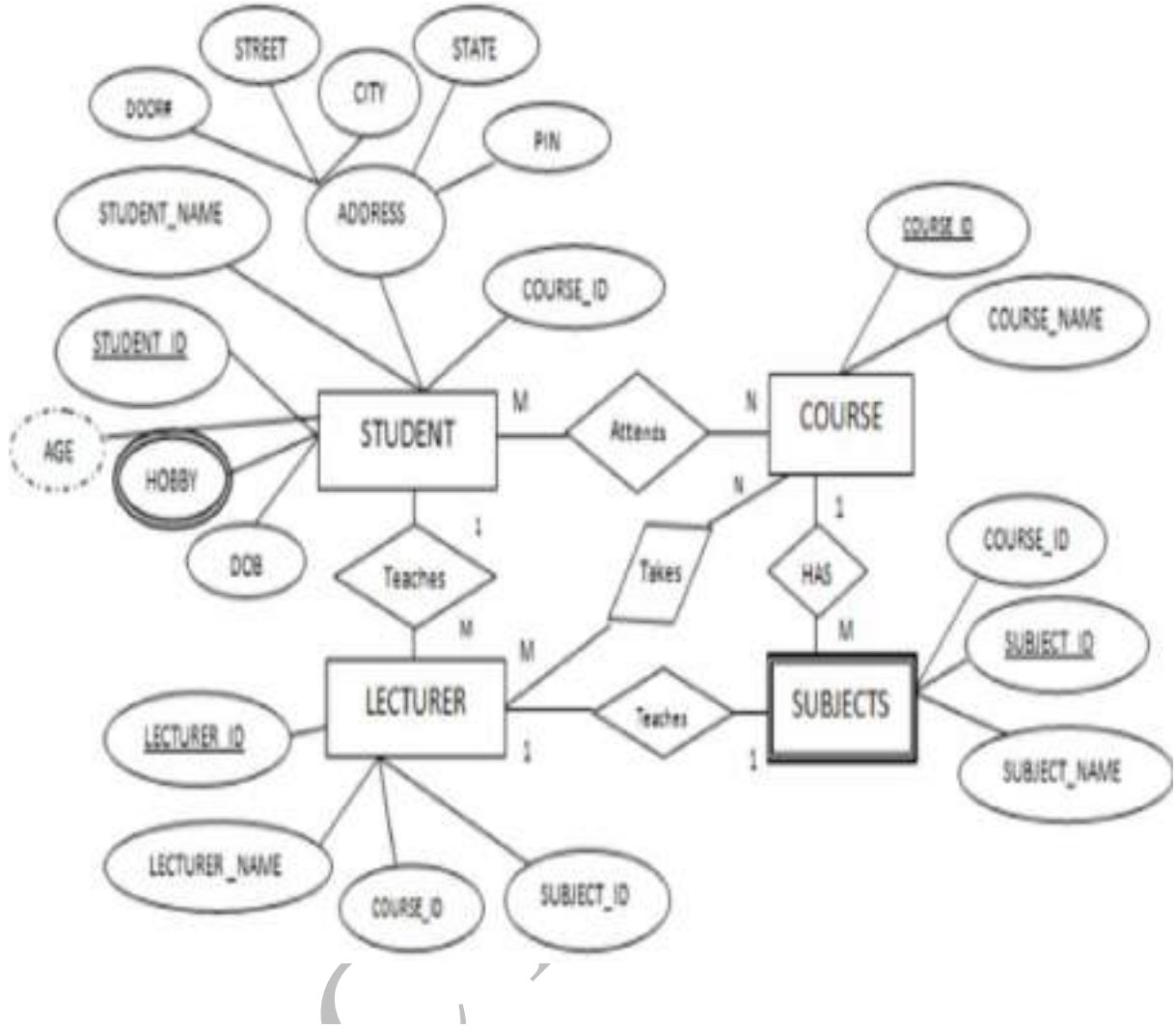


ER Diagram for Online BookStore

TRANSFORM ER DIAGRAM INTO TABLES

There are various steps involved in converting it into tables and columns. Each type of entity, attribute and relationship in the diagram takes their own depiction here. Consider the

ER diagram below and will see how it is converted into tables, columns and mappings.



The basic rule for converting the ER diagrams into tables is

- **Convert all the Entities in the diagram to tables.**

All the entities represented in the rectangular box in the ER diagram become independent tables in the database. In the below diagram, STUDENT, COURSE, LECTURER and SUBJECTS forms individual tables.

- **All single valued attributes of an entity is converted to a column of the table**

All the attributes, whose value at any instance of time is unique, are considered as columns of that table. In the STUDENT Entity, STUDENT_ID, STUDENT_NAME form the columns of STUDENT table. Similarly, LECTURER_ID, LECTURER_NAME form the columns of LECTURER table. And so on.

- **Key attribute in the ER diagram becomes the Primary key of the table.**

In diagram above, STUDENT_ID, LECTURER_ID, COURSE_ID and SUB_ID are the key attributes of the entities. Hence we consider them as the primary keys of respective table.

- **Declare the foreign key column, if applicable.**

In the diagram, attribute COURSE_ID in the STUDENT entity is from COURSE entity. Hence add COURSE_ID in the STUDENT table and assign it foreign key constraint. COURSE_ID and SUBJECT_ID in LECTURER table forms the foreign key column. Hence by declaring the foreign key constraints, mapping between the tables are established.

- **Any multi-valued attributes are converted into new table.**

A hobby in the Student table is a multi-valued attribute. Any student can have any number of hobbies. So we cannot represent multiple values in a single column of STUDENT table. We need to store it separately, so that we can store any number of hobbies, adding/ removing / deleting hobbies should not create any redundancy or anomalies in the system. Hence we create a separate table STUD_HOBBY with STUDENT_ID and HOBBY as its columns. We create a composite key using both the columns.

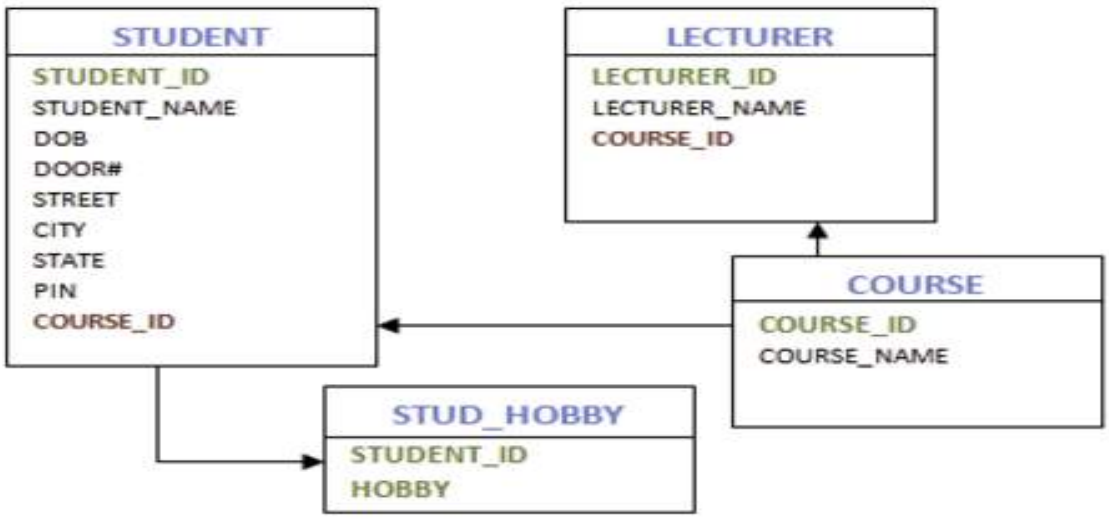
- **Any composite attributes are merged into same table as different columns.**

In the diagram above, Student Address is a composite attribute. It has Door#, Street, City, State and Pin. These attributes are merged into STUDENT table as individual columns.

- **One can ignore derived attribute, since it can be calculated at any time.**

In the STUDENT table, Age can be derived at any point of time by calculating the difference between DateOfBirth and current date. Hence we need not create a column for this attribute. It reduces the duplicity in the database.

→These are the very basic rules of converting ER diagram into tables and columns, and assigning the mapping between the tables. Table structure at this would be as below:



RELATIONAL MODEL:

→The **Relational Database** is a collection of one or more relations, where each relation is a table with rows and columns.

→The main construct for representing data in the relational model is a relation (table). A relation consists of a **relation schema and a relation instance.** The relation instance is a table, and the relation schema describes the column heads for the table.

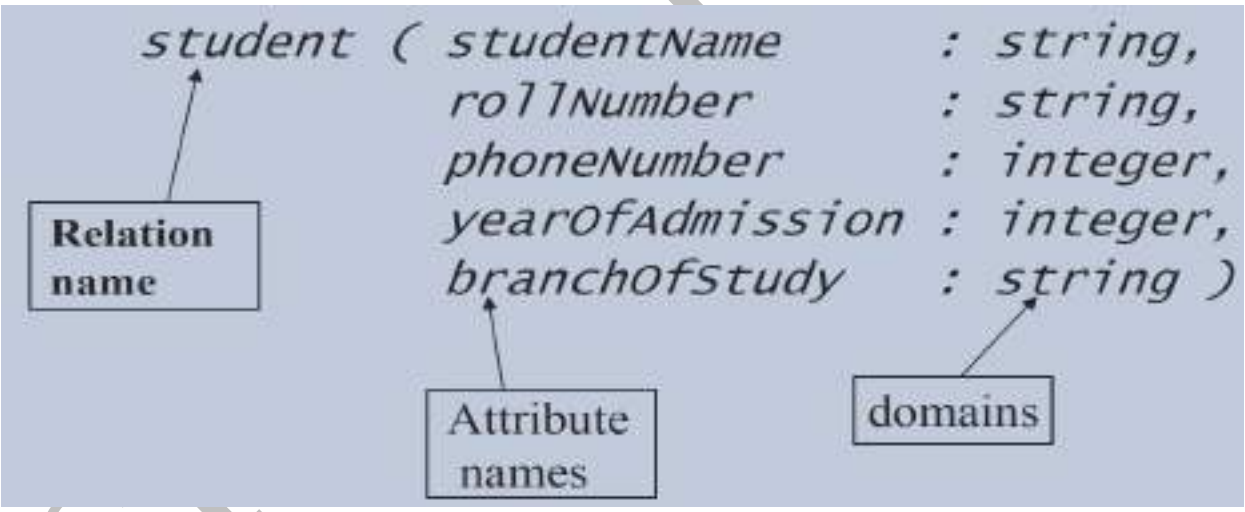
→The **schema** specifies the relation's name, the name of each field (or column, or attribute), and the domain of each field. A **domain** is referred to in a relation schema by the domain name and has a set of associated values.

Example of student information in a university database to illustrate the parts of a relation schema:

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

The field named sid has a domain named string. The set of values associated with domain string is the set of all character strings.

Example2:

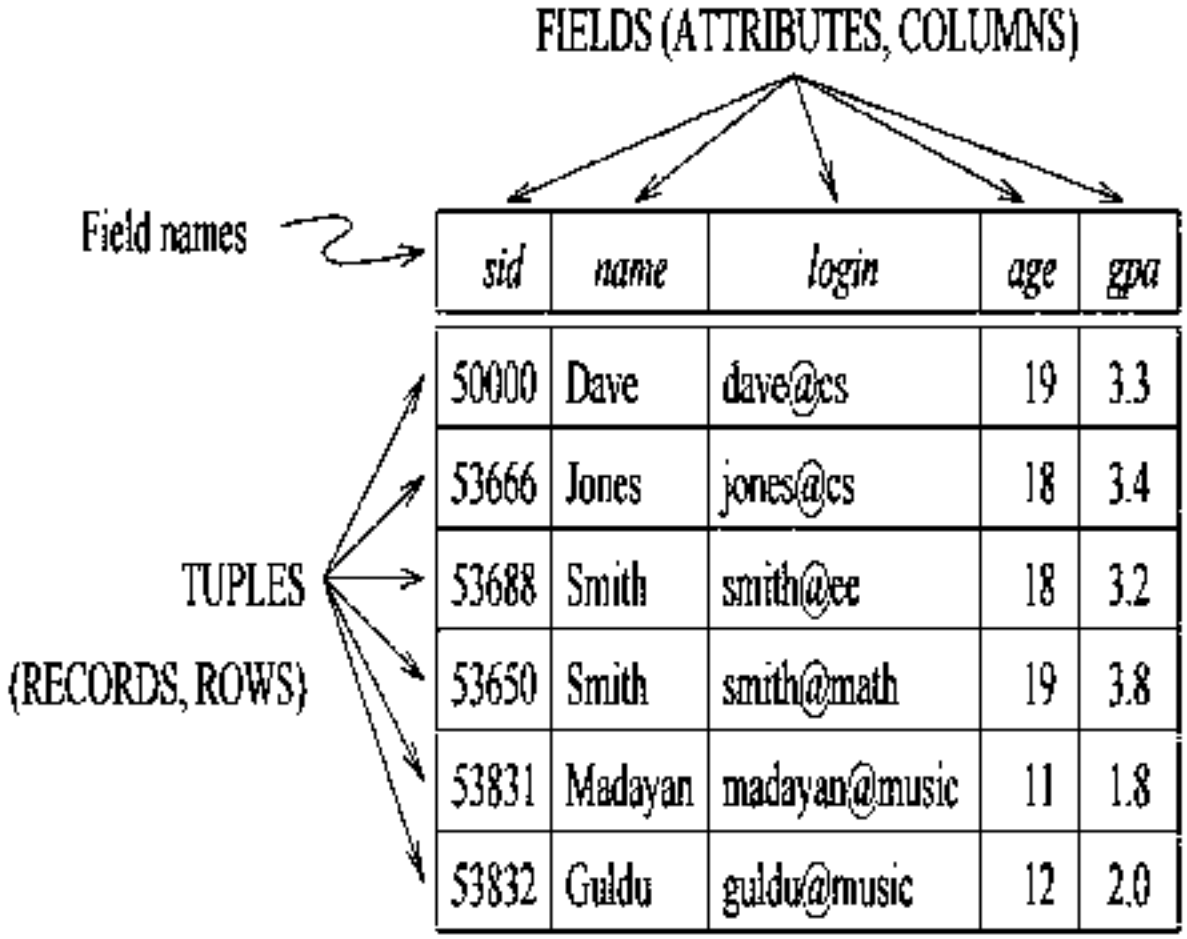


Domain–set of *atomic* (or *indivisible*) values –data type

→An **instance of a relation** is a set of **tuples**, also called **records**, in which each tuple has the same number of fields as the relation schema. A relation instance can be thought of as a table in which each tuple is a row, and all rows have the same number of fields

An instance of the Students relation appears in **Figure 3.1.**

The instance S1 contains six tuples and has, as we expect from the schema, five fields. Note that no two rows are identical. This is a requirement of the relational model—each relation is defined to be a set of unique tuples or rows.



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

Figure 3.1 An Instance S1 of the Students Relation

Cardinality = 3, degree = 5, all rows distinct.

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

→The **degree**, also called **arity**, of a relation is the number of fields. The **cardinality of a relation instance** is the number of tuples in it. In **Figure 3.1**, the **degree of the relation** (the number of columns) is five, and the **cardinality** of this instance is six.

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

More formally, let R(f_1:D1, ..., f_n:Dn) be a relation schema, and for each f_i, $1 \leq i \leq n$, let Dom_i be the set of values associated with the domain named Di. An instance of R that satisfies the domain constraints in the schema is a set of tuples with n fields:

$$\{ (f_1 : d_1, \ \ldots \ , f_n : d_n) \mid d_1 \in Dom_1, \ \ldots \ , d_n \in Dom_n \}$$

Another Example:

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

Figure 3.2 An Alternative Representation of Instance *S1* of Students

→A **relational database** is a collection of relations with distinct relation names. The **relational database schema** is the collection of schemas for the relations in the database. For example, University database with relations called Students, Faculty, Courses, Rooms, Enrolled, Teaches, and Meets In. An **instance of a relational database** is a collection of relation instances, one per relation schema in the database schema; of course, each relation instance must satisfy the domain constraints in its schema.

Creating and Modifying Relations Using SQL-92:

The SQL-92 language standard uses the word table to denote relation. The subset of SQL that supports the creation, deletion, and modification of tables is called the Data Definition Language (DDL).

Domain Types in SQL:

1. **char (n):**Fixed length character string, with user-specified length n.
2. **varchar (n) (or) character varying):**Variable length character strings, with user-specified maximum length n.
3. **int or integer:** An integer (a finite subset of the integers that is machine dependent).
4. **smallint:**a small integer (a machine-dependent subset of the integer domain type).
5. **numeric(p,d):**Fixed point number, with user-specified precision of pdigits, with n digits to the right of decimal point.
6. **Real (or) double precision:**Floating point and double-precision floating point numbers, with machine-dependent precision.
7. **float (n):**Floating point number, with user-specified precision of at least n digits.
8. **date:** a calendar date, containing four digit year, month, and day of the month.
9. **time**: the time of the day in hours, minutes, and seconds.

→The **CREATE TABLE** statement is used to define a new table. To create the Students relation, we can use the following statement:

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

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

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

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

```
DELETE
FROM    Students S
WHERE   S.name = 'Smith'
```

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

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

→The **WHERE** clause is applied first and determines which rows are to be modified. The **SET** clause then determines how these rows are to be modified.

consider the following variation of the previous query:

```
UPDATE  Students S
SET     S.gpa = S.gpa - 0.1
WHERE   S.gpa >= 3.3
```

→If this query is applied on the instance S1 of Students shown in **Figure 3.1**, we obtain the instance shown in **Figure 3.3.**

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

Figure 3.1 An Instance *S1* of the Students Relation

sid	*name*	*login*	*age*	*gpa*
50000	Dave	dave@cs	19	3.2
53666	Jones	jones@cs	18	3.3
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.7
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0

Figure 3.3 Students Instance *S1* after Update

2. INTEGRITY CONSTRAINTS OVER RELATIONS:(****)

An **integrity constraint (IC)** is a condition that ensures the correct insertion of the data and prevents unauthorized data access thereby preserving the consistency of the data.

For **example**, the roll number of a student cannot be a decimal value. The database enforces the constraint that the instance of roll number can have only integer values.

Integrity constraints are specified and enforced at different times:

1. When the DBA or end user defines a database schema, he or she specifies the ICs that must hold on any instance of this database.
2. When a database application is run, the DBMS checks for violations and disallows changes to the data that violate the specified ICs.

There are three types of integrity constraints in addition to domain constraint. They are:

1. **Key Constraints**
2. **Foreign Key Constraints.**
3. **General Constraints.**

1). KEY CONSTRAINTS:

→A **key constraint** is a statement that a certain minimal subset of the fields of a relation is a unique identifier for a tuple.

→Consider the Students relation and the constraint that no two students have the same student id. This IC is an example of a **key constraint**.

TYPES OF KEY CONSTRAINTS:

1. **Candidate key**
2. **Super key**
3. **Primary key**
4. **Foreign key**

1. CANDIDATE KEY:

→A candidate key is a collection of fields/columns/attributes that uniquely identifies a tuple.

→Let us take a closer look at the above definition of a (candidate) key.

→There are two parts to the definition:

1. Two distinct tuples in a legal instance (an instance that satisfies all ICs, including the key constraint) cannot have identical values in all the fields of a key.
2. No subset of the set of fields in a key is a unique identifier for a tuple.

Example: In "**customer**" relation the attribute "**cid**" is a key, it uniquely defines a tuple in a relation. No two rows in a relation "**customer**" can have the same "**cid"** value.

→The set of attributes that form a **candidate key** need not be all keys. The attributes may be treated as candidates to be taken as key.

Example: The set (**cid, cname**) is a **candidate key** which means either **cid** or **cname** can be taken as key but not both. Each of them independently and uniquely identifies a particular row. The alternate keys are candidate keys that are not taken as keys.

2. COMPOSITE KEY:

→Composite key consist of more than one attribute that uniquely identifies a tuple in a relation. All the attributes that form a set of keys and all of them taken together determines a unique row in a table.

Example: The set (**cid, accno**) is a **composite key** which maintains the uniqueness of each row. Both **cid, accno** are taken as keys.

3. SUPER KEY:

A **super key** is a combination of both **candidate key** and **composite key**. That is a set of attributes or a single attribute that uniquely identifies a tuple in a relation.

Example: Consider the **super key** {**cid, accno, cname**}
Here, all the three attributes taken together can identify a particular record or a combination of any two attributes can identify a particular record or any one of the attribute can identify a particular record.

4. PRIMARY KEY:

Only a single attribute can uniquely identify a particular record. More specifically, it can be defined as the candidate key, which has been selected as key to identify unique records.

Example: "cid" attribute in "customer" relation can be treated as PRIMARY KEY.

→Summary of Key (With respect to "customers" relation)

1) **Super key {cid, cname, accno}**
2) **Candidate key {cid, cname}**
3) **Composite key {cid, accno}**
4) **Primary key {cid}**

Specifying Key Constraints in SQL-92:

→In SQL, we can eliminate the chances of inserting duplicate data by using a unique constraint. This constraint helps the user to insert unique values for the columns which have been declared as unique, forming a candidate key any one of the columns among them can be declared as primary by using **primary key** constraints.

→**Example**, Consider the creation of "Students" table.

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

This example shows the creation of Students table with attributes sid, name, login, age, gpa, unique key is used on columns name and age which ensures that the values inserted in these columns are unique. The last line of declaration defines a primary key constraint.

→The syntax used for defining constraint is,

CONSTRAINT constraint-name PRIMARY KEY (key)

i.e., CONSTRAINT StudentsKey PRIMARY KEY (sid)

The line declares sid as **primary key** for **Students relation**. If the user inserts repeated values for**"sid"** then error occurs and constraint-name is return indicating violation of constraint.

---

2). Foreign Key Constraints(*******)

→A **foreign key (FK)** is a column or combination of columns that is used to establish and enforce a link between the data in two tables. You can create a foreign key by defining a **FOREIGN KEY constraint** when you create or modify a table.

→In a **foreign key reference**, a link is created between two tables when the column or columns that hold the primary key value for one table are referenced by the column or columns in another table. This column becomes a foreign key in the second table.

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

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

→To ensure that only bona fide students can enroll in courses, any value that appears in the sid field of an instance of the Enrolled relation should also appear in the sid field of some tuple in the Students relation. The sid field of Enrolled is called a foreign key and refers to Students. The **foreign key** in the **referencing relation** (Enrolled) must match the primary key

of the **referenced relation** (Students), i.e., it must have the same number of columns and compatible data types, although the column names can be different.

→This constraint is illustrated in **Figure 3.4.** As the figure shows, there may well be some students who are not referenced from Enrolled (e.g., the student with sid =50000).

→However, every **sid**value that appears in the instance of the Enrolled table appears in the primary key column of a row in the Students table.

→A **FOREIGN KEY constraint** does not have to be linked only to a PRIMARY KEY constraint in another table; it can also be defined to reference the columns of a **UNIQUE constraint** in another table. A FOREIGN KEY constraint can contain null values; however, if any column of a composite FOREIGN.

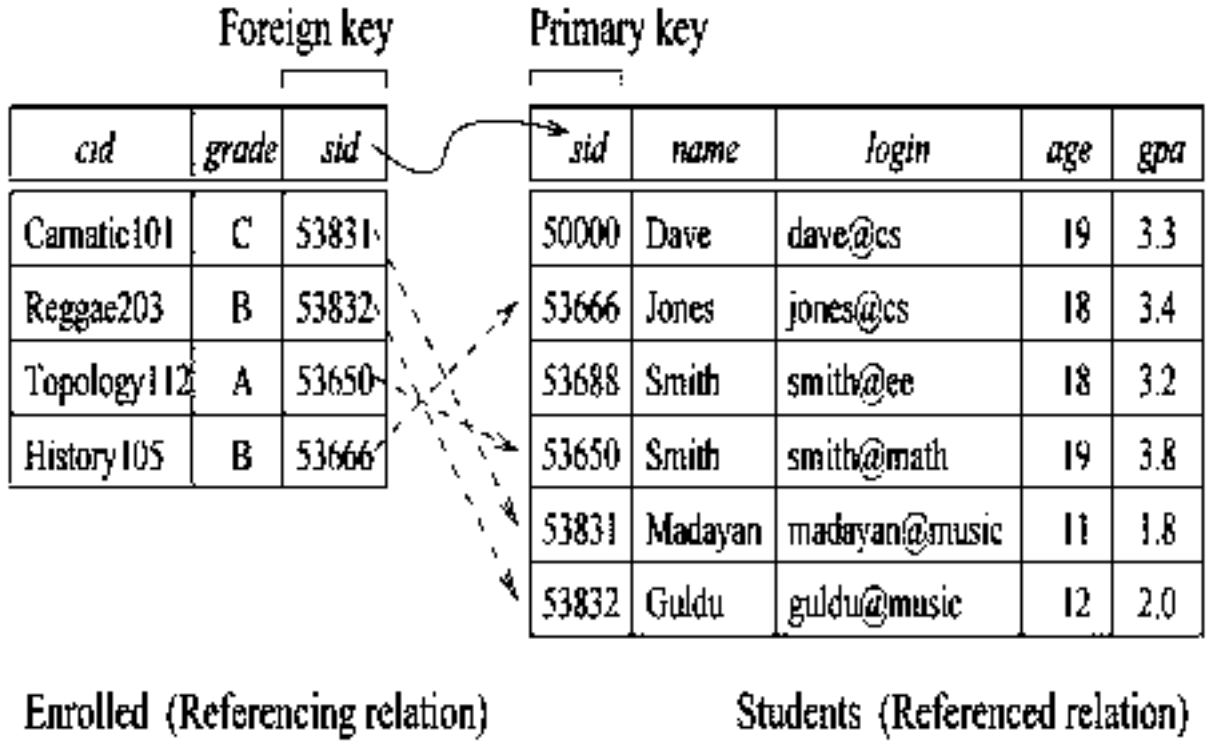


Figure 3.4 Referential Integrity

Specifying Foreign Key Constraints in SQL:

Let us define Enrolled(*sid:* `string`, *cid:* `string`, *grade:* `string`):

```
CREATE TABLE Enrolled ( sid     CHAR(20),
```

```
cid    CHAR(20),
grade CHAR(10),
PRIMARY KEY (sid, cid),
FOREIGN KEY (sid) REFERENCES Students )
```

The statement **FOREIGN KEY (sid) REFERENCES Students** means that the foreign key sid uses primary id sid of employee relation as a reference. Every tuple with sid must match a tuple in Students relation.

The **foreign key constraint** states that every **sid** value in Enrolled must also appear in Students, that is, sid in Enrolled is a **foreign key** referencing Students.

3).General Constraints:

Domain, primary key, and foreign key constraints are considered to be a fundamental part of the relational data model and are given special attention in most commercial systems. Sometimes, however, it is necessary to specify more general constraints.

E**xample**: we may require that student ages be within a certain range of values; given such an IC specification, the DBMS will reject inserts and updates that violate the constraint. This is very useful in preventing data entry errors. If we specify that all students must be at least 16 years old, then age are valid cases i.e., legal instance. Rest of all the others having lesser than 16 years are called as invalid cases i.e., illegal instance. Instance of Students shown in **Figure 3.1** is illegal because two students are underage. If we disallow the insertion of these two tuples, we have a legal instance, as shown in **Figure 3.5.**

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

Figure 3.1 An Instance $S1$ of the Students Relation

sid	*name*	*login*	*age*	*gpa*
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8

Figure 3.5 An Instance $S2$ of the Students Relation

→The IC that students must be older than 16, is known as an extended domain constraint, because we are restricting age values more stringently (strictly), than by simply using a standard domain such as integer.

→In general, constraints domain, primary and foreign key constraints can also specify the maximum limit.
Example: we require a student whose age is greater than 18 must have a gpa greater than 3.

→There are two types of general constraints. They are

1. Table Constraints: These are applied on a particular table and are checked every table whenever that specific table is updated.
2. Assertions: These assertions are applied on collection of tables and are checked every time whenever theses tables are applied.

---

3. ENFORCING INTEGRITY CONSTRAINTS:

→**Integrity Constraints(IC)** are the rules that when applied on relations restricts the insertion of incorrect data and also helps to prevent deletion and updating of consistent data that may lead to loss of data integrity. And, therefore one should be very careful when applying integrity constraints on relations.

The operations such as insertion, deletion and updating must be discarded if they are found to violate integrity constraints. This section provides a brief on different violations of ICs and also the solutions to handle these violations.

Consider the instance S1 of Students shown in **Figure 3.1**. The following insertion violates the primary key constraint because there is already a tuple with the **sid 53688**, and it will be rejected by the DBMS:

```
INSERT
INTO    Students   (sid, name, login, age, gpa)
VALUES  (53688, 'Mike', 'mike@ee', 17, 3.4)
```

→The following insertion violates the constraint that the primary key cannot contain **null:**

```
INSERT
INTO    Students   (sid, name, login, age, gpa)
VALUES  (null, 'Mike', 'mike@ee', 17, 3.4)
```

→Deletion does not cause a violation of domain, primary key or unique constraints.
→However, an update can cause violations, similar to an insertion:

```
UPDATE  Students S
SET     S.sid = 50000
WHERE   S.sid = 53688
```

This update violates the primary key constraint because there is already a tuple with **sid 50000.**

→In addition to the instance S1 of Students, consider the instance of Enrolled shown in Figure 3.4. Deletions of Enrolled tuples do not violate referential integrity, but insertions of Enrolled tuples could. The following insertion is illegal because there is no student with **sid 51111:**

```
INSERT
INTO    Enrolled   (cid, grade, sid)
VALUES  ('Hindi101', 'B', 51111)
```

EXAMPLE:

```
CREATE TABLE Enrolled ( sid    CHAR(20),
                        cid    CHAR(20),
                        grade  CHAR(10),
                        PRIMARY KEY (sid, cid),
                        FOREIGN KEY (sid) REFERENCES Students
                                ON DELETE CASCADE
                                ON UPDATE NO ACTION )
```

→This example explains the options when delete or update operation are performed. These options are included as a part of foreign key declaration. No action is the default option which means both update and delete operations are rejected.

1) On Delete Cascade: Means when a row is deleted from Students relation, then all the rows referred to this deleted row in Enrolled relation must also be deleted.

2) On Update Cascade: Means when updations are carried in Students relation for the primary key attribute then all these updations must also be carried out in Enrolled.

3) On Delete Set Default: Means when a row is deleted in Students, then that row in Enrolled relation can be set to same default value.

4) On Delete Set Null: Means on deleting the row in Students the same row can be assigned a NULL value in Enrolled relation.

NOTE: SQL even provides the facility to delay the applications of constraints on relation and also immediate application of constraints. This is possible with these two additional constraints,

1) **Deferred mode**
2) **Immediate mode**

The syntax for this constraint is,

```
SET CONSTRAINT Constraint-name DEFERRED

SET CONSTRAINT Constraint-name IMMEDIATE
```

→Usually, constraints are checked at the end of SQL statements and if the constraints are violated then the statements are rejected. But with differed constraint, constraint checks are postponed and are checked at the time of commit.

RELATIONAL ALGEBRA:

→**Relational algebra** is a procedural query language, which takes instances of relations as input and yields instances of relations as output. It uses operators to perform queries. An operator can be either **unary** or **binary**. They accept relations as their input and yield relations as their output. Relational algebra is performed recursively on a relation and intermediate results are also considered relations.

Example schemas:
Sailors (sid: integer, sname: string, rating: integer, age: real)
Boats (bid: integer, bname: string, color: string)
Reserves (sid: integer, bid: integer, day: date)
Example Instances:

R1

sid	bid	day
22	101	10/10/96
58	103	11/12/96

Figure 4.1 InstanceS1 of Sailors

S1

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

Figure 4.2 Instance S2 of Sailors

S2

sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

Figure 4.3 Instance R1 of Reserves

→The "Sailors" and "Reserves" relations are our examples. We'll use positional or named field notation, assume that names of fields in query results are `inherited' from names of fields in query input relations.

→**The fundamental operations of relational algebra are:**

1. **Basic operators:**
   a) **Selection**
   b) **Projection**

2. **Set Operations:**
   a) **Union**
   b) **Intersection**
   c) **Set-difference**
   d) **Cross-product**

3. **Renaming**
4. **Joins**
   a) **Condition joins**
   b) **Equijoin**
   c) **Natural join**
5. **Division**
6. **Assignment operation.**

---

1. Selection and Projection:

→Relational algebra includes operators to select rows from a relation (σ) and to project columns (π). These operations allow us to manipulate data in a single relation.

Selection - σ Selects a subset of rows from relation.

Projection- π Deletes unwanted columns from relation.

SELECTION (σ):

The **selection operation** is a unary operation. This is used to find horizontal subset of relation or tuples of relation.

It selects tuples that satisfy the given predicate from a relation. It is denoted by **sigma**(**σ**).

Notation − σ_p(r)

Where **σ** stands for selection predicate and **r** stands for relation. *p* is prepositional logic formula which may use connectors like **and, or,** and **not**. These terms may use relational operators like − =, ≠, ≥, < , >, ≤.

Example: If you want all the **Sailors having rating more than 8** from instance S2 of Sailors. The query is,

$$\sigma_{rating>8}(S2)$$

The result is shown in **Figure 4.4**

sid	*sname*	*rating*	*age*
28	yuppy	9	35.0
58	Rusty	10	35.0

Figure 4.4 $\sigma_{rating>8}(S2)$

PROJECTION (π):

The **projection operation** is a unary operation which applies only on a single relation at a time. This is used to select vertical subset of relation (i.e., columns of table)
It projects column(s) that satisfy a given predicate. It is denoted by **pi (π).**

Notation − $\prod_{A1, A2, An}$ **(r)**

Where A_1, A_2 , A_n are attribute names of relation **r**.

Duplicate rows are automatically eliminated, as relation is a set.

Example: If you can find out **all sailors names and ratings** from instance S2 of Sailors. The query is,

$$\pi_{sname,rating}(S2)$$

The result is shown in **Figure 4.5**

sname	*rating*
yuppy	9
Lubber	8
guppy	5
Rusty	10

Figure 4.5 $\pi_{sname,rating}(S2)$

→ Suppose that we wanted to find out **only the ages of sailors**. The expression

$$\pi_{age}(S2)$$

evaluates to the relation shown in **Figure 4.6.**

age
35.0
55.5

Figure 4.6 $\pi_{age}(S2)$

→ For example, we can compute the **names and ratings of highly rated sailors** by combining two of the preceding queries. The expression

$$\pi_{sname,rating}(\sigma_{rating>8}(S2))$$

produces the result shown in **Figure 4.7.** It is obtained by applying the selection to S2 (to get the relation shown in Figure 4.4) and then applying the projection.

sname	*rating*
yuppy	9
Rusty	10

Figure 4.7 $\pi_{sname,rating}(\sigma_{rating>8}(S2))$

2. SET OPERATIONS:

The relational algebraic operations can be divided into basic set oriented operations (Union, Intersection, Set difference and Cartesian product).

×	*Cross-product*	Allows us to combine two relations.
—	*Set-difference*	Tuples are in relation. 1, but not in relation. 2.
∪	*Union*	Tuples are in relation. 1 or in relation. 2.
∩	*Intersection*	Tuples are in relation. 1 and in relation. 2.

The UNION (∪) Operation:

→R∪S returns a relation instance containing all tuples that occur in either relation instance R or relation instance S (or both). R and S must be union-compatible, and the schema of the result is defined to be identical to the schema of R.

Notation – R U S

→Two relation instances are said to be **union-compatible** if the following conditions hold:

- They have the same number of the fields, and
- Corresponding fields, taken in order from left to right, have the same domains.
- Duplicate tuples are automatically eliminated.

Note that field names are not used in defining union-compatibility. For convenience, we will assume that the fields of R ∪ S inherit names from R, if the fields of R have names.

→The union of S1 and S2 is shown in **Figure 4.8**. Fields are listed in order; field names are also inherited from S1. S2 has the same field names, of course, since it is also an instance of Sailors. In general, fields of S2 may have different names; recall that we require only domains to match. Note that the result is a set of tuples. Tuples that appear in both S1 and S2 appear

only once in S1 ∪ S2. Also, S1 ∪ R1 is not a valid operation because the two relations are not union-compatible.

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0
44	guppy	5	35.0
28	yuppy	9	35.0

Figure 4.8 S1 ∪ S2

The INTERSECTION (∩) Operation:

→R∩S returns a relation instance containing all tuples that occur in both R and S. The relations R and S must be **union-compatible**, and the schema of the result is defined to be identical to the schema of R.

Notation − R ∩ S

If the relations contain nothing as common then the result will be an empty relation. Rules of set union operations are also applicable here.

→The intersection of S1 and S2 is shown in **Figure 4.9.**

sid	sname	rating	age
31	lubber	8	55.5
58	rusty	10	35.0

Figure 4.9S1∩ S2

The SET-DIFFERENCE (−) Operation:

→**R−S** returns a relation instance containing all tuples that occur in R but not in S. The relations R and S must be union-compatible, and the schema of the result is defined to be identical to the schema of R.

The result of set difference operation is tuples, which are present in one relation but are not in the second relation. It removes the common tuples of two relations and produces a new relation having rest of the tuples of first relation.

Notation − R − S

→ It finds all the tuples that are present in **R** but not in **S**.

→The set-difference S1 − S2 is shown in **Figure 4.10.**

sid	sname	rating	age
22	dustin	7	45.0

Figure 4.10S1- S2

The CROSS-PRODUCT (×) Operation:

→**R ×S** returns a relation instance whose schema contains all the fields of R (in the same order as they appear in R) followed by all the fields of S (in the same order as they appear in S). The result of R × S contains one tuple hr, si (the concatenation of tuples r and s) for each pair of tuples r ∈ R, s ∈ S. The cross-product operation is sometimes called **Cartesian product**.

→We will use the convention that the fields of R × S inherit names from the corresponding fields of R and S. It is possible for both R and S to contain one or more fields having the same name; this situation creates a naming conflict. The corresponding fields in R × S are unnamed and are referred to solely by position.

→ It combines information of two different relations into one.

Notation − R X S

→The result of the cross-product **S1 × R1** is shown in **Figure 4.11.** Because R1 and S1 both have a field named sid, by our convention on field names, the corresponding two fields in S1 × R1 are unnamed, and referred to solely by the position in which they appear in **Figure 4.11**. The fields in S1 × R1 have the same domains as the corresponding fields in R1 and S1. In **Figure 4.11sid** is listed in parentheses to emphasize that it is not an inherited field name; only the corresponding domain is inherited.

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	11/12/96
58	rusty	10	35.0	22	101	10/10/96
58	rusty	10	35.0	58	103	11/12/96

Figure 4.11S1 × R1

3. Renaming (ρ): (********)

The rename (*ρ)* operation is a unary operation which is used to give names to relational algebra expressions.

The results of relational algebra are also relations but without any name. The rename operation allows us to rename the output relation. 'Rename' operation is denoted with small Greek letter **rho** *ρ*.

Suppose, you want to find Cartesian product of a relation with itself then by using rename operator we give an alias name to that relation. Now, you can easily multiply that relation with its alias. It is helpful in removing ambiguity.

Notation − ρ_x (E)

→Where the result of expression **E** is saved with name of **x**.

For example, the expression **ρ(C (1 → sid1, 5 → sid2), S1 × R1)** returns a relation that contains the tuples shown in **Figure 4.11** and has the following schema:

C (sid1: integer, **sname**: string, **rating**: integer, **age**: real, **sid2**: integer, **bid**: integer, **day**: dates).

4. Joins: (********)

The join operation is one of the most useful operations in relational algebra and is the most commonly used way to combine information from two or more relations.

The join operation denoted by "join" or "⋈", is a relational algebra operation, which is used to combine (join) two relations like Cartesian-product but finally removes duplicate attributes (same column to only one column) and makes the operations (selection, projection etc.,) very simple. In simple words, we can say that join connects relations on columns containing comparable information.

There are three types of joins. Namely, they are

1. **Condition Joins**
2. **Equi Join** and
3. **Natural join**.

1. Condition Joins:

→The most general version of the join operation accepts a join condition c and a pair of relation instances as arguments, and returns a relation instance. The join condition is identical to a selection condition in form. The operation is defined as follows:

$$R \bowtie_c S = \sigma_c(R \times S)$$

Thus⋈ *is defined to be* a cross-product followed by a selection. Note that the condition c can (and typically does) refer to attributes of both R and S. The reference to an attribute of a relation, say R, can be by position (of the form R.i) or by name (of the form R.name).

→**Example:** the result of $S1 \bowtie_{S1.sid < R1.sid} R1$ is shown in **Figure 4.12**. Because **sid** appears in both S1 and R1, the corresponding fields in the result of the cross-product S1 × R1 (and therefore in the result

of $S1 \bowtie_{S1.sid < R1.sid} R1$ are unnamed. Domains are inherited from the corresponding fields of S1 and R1.

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	58	103	11/12/96

$S1 \bowtie_{S1.sid < R1.sid} R1$

Figure 4.12

- ***Result schema***same as that of cross-product.
- Fewer tuples than cross-product, might be able to compute more efficiently
- Sometimes called a ***theta-join*.**

2.Equijoin:

→ It is a special case of condition join where the condition *c* contains only ***equalities.***

→**Equijoin** is same as condition join, the only difference is that, equijoin uses equity '=' operator to join the two relations.

The schema of the result of an equijoin contains the fields of R (with the same names and domains as in R) followed by the fields of S that do not appear in the join conditions. If this set of fields in the result relation includes two fields that inherit the same name from R and S, they are unnamed in the result relation.

We illustrate $S1 \bowtie_{R.sid=S.sid} R1$ in **Figure 4.13**. Notice that only one field called sid appears in the result.

sid	*sname*	*rating*	*age*	*bid*	*day*
22	Dustin	7	45.0	101	10/10/96
58	Rusty	10	35.0	103	11/12/96

Figure 4.13 $S1 \bowtie_{R.sid=S.sid} R1$

→**Result schema**similar to cross-product, but only one copy of fields for which equality is specified.

3. Natural Join:

→**Natural join** does not use any comparison operator. It does not concatenate the way a Cartesian product does. We can perform a **Natural Join** only if there is at least one common attribute that exists between two relations. In addition, the attributes must have the same name and domain.

→**Natural join** acts on those matching attributes where the values of attributes in both the relations are same.

→Special case of the join operation R ⋈S is an equijoin in which equalities are specified on all fields having the same name in R and S. In this case, we can simply omit the join condition; the default is that the join condition is a collection of equalities on all common fields. This special case a **natural join**, and with this result is guaranteed not to have two fields with the same name.

The equijoin expression $S1 \bowtie_{R.sid=S.sid} R1$ is actually a natural join and can simply be denoted as S1⋈R1, since the only common field is sid. If the two relations have no attributes in common, S1 ⋈R1 is simply the cross-product.

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	58	103	11/12/96

Figure.S1⋈R1

5. Division:

→The **division operator** is useful for expressing certain kinds of queries that include the phrase "for all'. It is denoted by **(/)**. It is alike the inverse of Cartesian product.

Consider two relation instances A and B in which A has (exactly) two fields x and y and B has just one field y, with the same domain as in A. We define the division operation A/B as the set of all x values (in the form of unary tuples) such that for every y value in (a tuple of) B, there is a tuple <x, y> in A.

Another way to understand division is as follows. For each x value in (the first column of) A, consider the set of y values that appear in (the second field of) tuples of A with that x value. If this set contains (all y values in) B, the x value is in the result of A/B.

An analogy with integer division may also help to understand division. For integers A and B, A/B is the largest integer Q such that Q ∗ B ≤ A. For relation instances A and B, A/B is the largest relation instance Q such that Q × B ⊆ A.

Division is illustrated in **Figure 4.14.**

A

sno	*pno*
s1	p1
s1	p2
s1	p3
s1	p4
s2	p1
s2	p2
s3	p2
s4	p2
s4	p4

B1

pno
p2

B2

pno
p2
p4

B3

pno
p1
p2
p4

A/B1

sno
s1
s2
s3
s4

A/B2

sno
s1
s4

A/B3

sno
s1

Figure 4.14 Examples Illustrating Division

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0
29	Brutus	1	33.0
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35.0
64	Horatio	7	35.0
71	Zorba	10	16.0
74	Horatio	9	35.0
85	Art	3	25.5
95	Bob	3	63.5

Figure 4.15 An Instance $S3$ of Sailors

sid	*bid*	*day*
22	101	10/10/98
22	102	10/10/98
22	103	10/8/98
22	104	10/7/98
31	102	11/10/98
31	103	11/6/98
31	104	11/12/98
64	101	9/5/98
64	102	9/8/98
74	103	9/8/98

Figure 4.16 An Instance $R2$ of Reserves

bid	*bname*	*color*
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

Figure 4.17 An Instance $B1$ of Boats

1. Basic SQL,Query:

→**Structured Query Language (SQL)** is the most widely used commercial relational database language. It was originally developed at IBM in the SEQUEL-XRM and System-R projects (1974–1977).

The **SQL language** has several aspects to it:

1. The Data Definition Language (DDL): This subset of SQL supports the creation, deletion, and modification of definitions for tables and views. Integrity constraints can be defined on tables, either when the table is created or later. The DDL also provides commands for specifying access rights or privileges to tables and views. Although the standard does not discuss indexes, commercial implementations also provide commands for creating and deleting indexes.

2. The Data Manipulation Language (DML): This subset of SQL allows users to pose queries and to insert, delete, and modify rows.

3. Embedded and dynamic SQL: Embedded SQL features allow SQL code to be called from a host language such as C or COBOL. Dynamic SQL features allow a query to be constructed (and executed) at run-time.

4. Triggers: The new SQL:1999 standard includes support for triggers, which are actions executed by the DBMS whenever changes to the database meet conditions specified in the trigger.

5. Security: SQL provides mechanisms to control users' access to data objects such as tables and views.

6. Transaction management: Various commands allow a user to explicitly control aspects of how a transaction is to be executed.

7. Client-server execution and remote database access: These commands control how a client application program can connect to an SQL database server, or access data from a database over a network.

2. THE FORM OF A BASIC SQL QUERY:

→The basic form of an SQL query is as follows:

```
SELECT    [ DISTINCT ] select-list
FROM      from-list
WHERE     qualification
```

→**Such a query intuitively corresponds** to a relational algebra expression involving selections, projections, and cross-products.

→Every query must have a SELECT clause, which specifies columns to be retained in the result, and a FROM clause, which specifies a cross-product of tables. The optional WHERE clause specifies selection conditions on the tables mentioned in the FROM clause.

sid	sname	rating	age
22	Dustin	7	45.0
29	Brutus	1	33.0
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35.0
64	Horatio	7	35.0
71	Zorba	10	16.0
74	Horatio	9	35.0
85	Art	3	25.5
95	Bob	3	63.5

Figure 5.1 An Instance $S3$ of Sailors

sid	bid	day
22	101	10/10/98
22	102	10/10/98
22	103	10/8/98
22	104	10/7/98
31	102	11/10/98
31	103	11/6/98
31	104	11/12/98
64	101	9/5/98
64	102	9/8/98
74	103	9/8/98

Figure 5.2 An Instance $R2$ of Reserves

bid	bname	color
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

Figure 5.3 An Instance $B1$ of Boats

- The from-list in the **FROM clause** is a list of table names. A table name can be followed by a range variable; a range variable is particularly useful when the same table name appears more than once in the from-list.
- The **select-list** is a list of (expressions involving) column names of tables named in the from-list. Column names can be prefixed by a range variable.
- The qualification in the **WHERE clause** is a Boolean combination (i.e., an expression using the logical connectives AND, OR, and NOT) of conditions of the form expression op expression, where op is one of the comparison operators {<=, =, <>, >=, >}. 2 An expression is a column name, a constant, or an (arithmetic or string) expression.
- The **DISTINCT keyword** is optional. It indicates that the table computed as an answer to this query should not contain duplicates, that is, two copies of the same row. The default is that duplicates are not eliminated.

SELECT Clause:

→Let us consider a simple query:

(Q15) Find the names and ages of all sailors.

```
SELECT DISTINCT S.sname, S.age
FROM    Sailors S
```

→The answer is a set of rows, each of which is a pair **<sname, age>.** If two or more sailors have the same name and age, the answer still contains just one pair with that name and age. This query is equivalent to applying the projection operator of relational algebra.

→The answer to this query with and without the keyword DISTINCT on instance S3 of Sailors is shown in **Figures 5.4 and 5.5.** The only difference is that the tuple for Horatio appears twice if DISTINCT is omitted; this is because there are two sailors called Horatio and age 35

sname	*age*
Dustin	45.0
Brutus	33.0
Lubber	55.5
Andy	25.5
Rusty	35.0
Horatio	35.0
Zorba	16.0
Art	25.5
Bob	63.5

Figure 5.4 Answer to Q15

sname	*age*
Dustin	45.0
Brutus	33.0
Lubber	55.5
Andy	25.5
Rusty	35.0
Horatio	35.0
Zorba	16.0
Horatio	35.0
Art	25.5
Bob	63.5

Figure 5.5 Answer to Q15 without DISTINCT

(Q11) Find all sailors with a rating above 7.

```
SELECT S.sid, S.sname, S.rating, S.age
FROM   Sailors AS S
WHERE  S.rating > 7
```

→This query uses the optional keyword **AS** to introduce a range variable. Incidentally, when we want to retrieve all columns, as in this query, SQL provides convenient shorthand: We can simply write **SELECT ***. This notation is useful for interactive querying, but it is poor style for queries that are intended to be reused and maintained.

Conceptual evaluation strategy:

1. Compute the cross-product of the tables in the from-list.
2. Delete those rows in the cross-product that fail the qualification conditions.

3. Delete all columns that do not appear in the select-list.
4. If DISTINCT is specified, eliminate duplicate rows.

We illustrate the conceptual evaluation strategy using the following query:
(Q1) Find the names of sailors who have reserved boat number 103.

It can be expressed in SQL as follows.

```
SELECT  S.sname
FROM    Sailors S, Reserves R
WHERE   S.sid = R.sid AND R.bid=103
```

sid	*bid*	*day*
22	101	10/10/96
58	103	11/12/96

Figure 5.6 Instance *R3* of Reserves

sid	*sname*	*rating*	*age*
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

Figure 5.7 Instance *S4* of Sailors

→The first step is to construct the cross-product S4 × R3, which is shown in **Figure 5.8.**

sid	*sname*	*rating*	*age*	*sid*	*bid*	*day*
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	11/12/96
58	rusty	10	35.0	22	101	10/10/96
58	rusty	10	35.0	58	103	11/12/96

Figure 5.8 *S4 × R3*

→The second step is to apply the qualification S.sid = R.sid AND R.bid=103. This step eliminates all but the last row from the instance shown in **Figure 5.8.**

→The third step is to eliminate unwanted columns; only sname appears in the SELECT clause. This step leaves us with the result shown in **Figure 5.9**, which is a table with a single column and, as it happens, just one row.

sname
rusty

Figure 5.9 Answer to Query Q1 on R3 and S4

Examples of Basic SQL Queries:

(Q16) Find the sids of sailors who have reserved a red boat.

```
SELECT   R.sid
FROM     Boats B, Reserves R
WHERE    B.bid = R.bid AND B.color = 'red'
```

(Q2) Find the names of sailors who have reserved a red boat.

```
SELECT   S.sname
FROM     Sailors S, Reserves R, Boats B
WHERE    S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
```

(Q3) Find the colors of boats reserved by Lubber.

```
SELECT B.color
FROM   Sailors S, Reserves R, Boats B
WHERE  S.sid = R.sid AND R.bid = B.bid AND S.sname = 'Lubber'
```

(Q4) Find the names of sailors who have reserved at least one boat.

```
SELECT  S.sname
FROM    Sailors S, Reserves R
WHERE   S.sid = R.sid
```

Expressions and Strings in the SELECT Command:

(Q17) Compute increments for the ratings of persons who have sailed two different boats on the same day.

```
SELECT  S.sname, S.rating+1 AS rating
FROM    Sailors S, Reserves R1, Reserves R2
WHERE   S.sid = R1.sid AND S.sid = R2.sid
        AND R1.day = R2.day AND R1.bid <> R2.bid
```

→Also, each item in a qualification can be as general as expression1 = expression2.

```
SELECT  S1.sname AS name1, S2.sname AS name2
FROM    Sailors S1, Sailors S2
WHERE   2*S1.rating = S2.rating-1
```

(Q18) Find the ages of sailors whose name begins and ends with B and has at least three characters.

```
SELECT  S.age
FROM    Sailors S
WHERE   S.sname LIKE 'B_%B'
```

The only such sailor is Bob, and his age is 63.5.

3. UNION, INTERSECT, AND EXCEPT:

→The **UNION operation** combines two relations and automatically eliminates the duplicate tuples.
→The **INTERSECT operation** finds the common tuples of two relations and eliminates the duplicate tuples.
→The **EXCEPT operation** finds the tuples which are in one relation but not in the other relation and automatically eliminates duplicate tuples.

(Q5) Find the names of sailors who have reserved a red or a green boat.

```
SELECT  S.sname
FROM    Sailors S, Reserves R, Boats B
WHERE   S.sid = R.sid AND R.bid = B.bid
        AND (B.color = 'red' OR B.color = 'green')
```

→The OR query (Query Q5) can be rewritten as follows:

```
SELECT S.sname
FROM   Sailors S, Reserves R, Boats B
WHERE  S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
UNION
SELECT S2.sname
FROM   Sailors S2, Boats B2, Reserves R2
WHERE  S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'green'
```

(Q6) Find the names of sailors who have reserved both a red and a green boat.

```
SELECT S.sname
FROM   Sailors S, Reserves R1, Boats B1, Reserves R2, Boats B2
WHERE  S.sid = R1.sid AND R1.bid = B1.bid
       AND S.sid = R2.sid AND R2.bid = B2.bid
       AND B1.color='red' AND B2.color = 'green'
```

→AND query (Query Q6) can be rewritten as follows:

```
SELECT S.sname
FROM   Sailors S, Reserves R, Boats B
WHERE  S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
INTERSECT
SELECT S2.sname
FROM   Sailors S2, Boats B2, Reserves R2
WHERE  S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'green'
```

(Q19) Find the sids of all sailors who have reserved red boats but not green boats.

```
SELECT S.sid
FROM   Sailors S, Reserves R, Boats B
WHERE  S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
EXCEPT
SELECT S2.sid
FROM   Sailors S2, Reserves R2, Boats B2
WHERE  S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'green'
```

```
SELECT R.sid
FROM   Boats B, Reserves R
WHERE  R.bid = B.bid AND B.color = 'red'
EXCEPT
SELECT R2.sid
FROM   Boats B2, Reserves R2
WHERE  R2.bid = B2.bid AND B2.color = 'green'
```

(Q20) Find all sids of sailors who have a rating of 10 or have reserved boat 104.

```
SELECT S.sid
FROM   Sailors S
WHERE  S.rating = 10

UNION
SELECT R.sid
FROM   Reserves R
WHERE  R.bid = 104
```

4. NESTED QUERIES:

Introduction to Nested Queries:

(Q1) Find the names of sailors who have reserved boat 103.

```
SELECT S.sname
FROM   Sailors S
WHERE  S.sid IN ( SELECT R.sid
                  FROM   Reserves R
                  WHERE  R.bid = 103 )
```

(Q2) Find the names of sailors who have reserved a red boat.

```
SELECT   S.sname
FROM     Sailors S
WHERE    S.sid IN ( SELECT R.sid
                    FROM   Reserves R
                    WHERE  R.bid IN ( SELECT B.bid
                                      FROM   Boats B
                                      WHERE  B.color = 'red' )
```

(Q21) Find the names of sailors who have not reserved a red boat.

```
SELECT   S.sname
FROM     Sailors S
WHERE    S.sid NOT IN ( SELECT R.sid
                        FROM   Reserves R
                        WHERE  R.bid IN ( SELECT B.bid
                                          FROM   Boats B
                                          WHERE  B.color = 'red' )
```

Correlated Nested Queries:

(Q1) Find the names of sailors who have reserved boat number 103.

```
SELECT S.sname
FROM   Sailors S
WHERE  EXISTS ( SELECT *
                FROM   Reserves R
                WHERE  R.bid = 103
                AND  R.sid = S.sid )
```

Set-Comparison Operators:

(Q22) Find sailors whose rating is better than some sailor called Horatio.

```
SELECT S.sid
FROM   Sailors S
WHERE  S.rating > ANY ( SELECT S2.rating
                        FROM   Sailors S2
                        WHERE  S2.sname = 'Horatio' )
```

(Q23) Find sailors whose rating is better than every sailor called Horatio.

→We can obtain all such queries with a simple modification to Query Q22: just replace ANY with ALL in the WHERE clause of the outer query.

(Q24) Find the sailors with the highest rating.

```
SELECT S.sid
FROM   Sailors S
WHERE  S.rating >= ALL ( SELECT S2.rating
                         FROM   Sailors S2 )
```

More Examples of Nested Queries:

(Q6) Find the names of sailors who have reserved both a red and a green boat.

```
SELECT S.sname
FROM   Sailors S, Reserves R, Boats B
WHERE  S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
       AND S.sid IN ( SELECT S2.sid
                      FROM   Sailors S2, Boats B2, Reserves R2
                      WHERE  S2.sid = R2.sid AND R2.bid = B2.bid
                             AND B2.color = 'green' )
```

```
SELECT S3.sname
FROM   Sailors S3
WHERE  S3.sid IN (( SELECT R.sid
                    FROM   Boats B, Reserves R
                    WHERE  R.bid = B.bid AND B.color = 'red' )
                    INTERSECT
                    (SELECT R2.sid
                    FROM   Boats B2, Reserves R2
                    WHERE  R2.bid = B2.bid AND B2.color = 'green' ))
```

(Q9) Find the names of sailors who have reserved all boats.

```
SELECT S.sname
FROM   Sailors S
WHERE  NOT EXISTS (( SELECT B.bid
                     FROM   Boats B )
                     EXCEPT
                     (SELECT R.bid
                     FROM   Reserves R
                     WHERE  R.sid = S.sid ))
```

```
SELECT S.sname
FROM   Sailors S
WHERE  NOT EXISTS ( SELECT B.bid
                    FROM   Boats B
                    WHERE  NOT EXISTS ( SELECT R.bid
                                        FROM   Reserves R

                                        WHERE  R.bid = B.bid
                                        AND R.sid = S.sid ))
```

5. AGGREGATE OPERATORS:

→Aggregate functions operate on a multiset of values and return a single value. Typical aggregate functions are **min, max, sum, count,** and **avg.**

→These features represent a significant extension of relational algebra.
→SQL supports five aggregate operations, which can be applied on any column, say A, of a relation:

1. **COUNT ([DISTINCT] A):** The number of (unique) values in the A column.

2. **SUM ([DISTINCT] A):** The sum of all (unique) values in the A column.

3. **AVG ([DISTINCT] A):** The average of all (unique) values in the A column.

4. **MAX (A):** The maximum value in the A column.

5. **MIN (A):** The minimum value in the A column.

Examples:

(Q25) Find the average age of all sailors

```
SELECT  AVG  (S.age)
FROM    Sailors S
```

On instance S3, the average age is 37.4.

(Q26) Find the average age of sailors with a rating of 10.

```
SELECT  AVG  (S.age)
FROM    Sailors S
WHERE   S.rating = 10
```

There are two such sailors, and their average age is 25.5. MIN (or MAX) can be used instead of AVG in the above queries to find the age of the youngest (oldest) sailor.

(Q27) Find the name and age of the oldest sailor.

Consider the following attempt to answer this query:

```
SELECT  S.sname, MAX  (S.age)
FROM    Sailors S
```

```
SELECT  S.sname, S.age
FROM    Sailors S
WHERE   S.age = ( SELECT MAX (S2.age)
                  FROM  Sailors S2 )
```

```
SELECT  S.sname, S.age
FROM    Sailors S
WHERE   ( SELECT MAX (S2.age)
          FROM Sailors S2 ) = S.age
```

(Q28) Count the number of sailors.

```
SELECT COUNT (*)
FROM   Sailors S
```

(Q29) Count the number of different sailor names.

```
SELECT COUNT ( DISTINCT S.sname )
FROM   Sailors S
```

(Q30) Find the names of sailors who are older than the oldest sailor with a rating of 10.

```
SELECT S.sname
FROM   Sailors S
WHERE  S.age > ( SELECT MAX ( S2.age )
                 FROM   Sailors S2
                 WHERE  S2.rating = 10 )
```

```
SELECT S.sname
FROM   Sailors S
WHERE  S.age > ALL ( SELECT S2.age
                     FROM   Sailors S2
                     WHERE  S2.rating = 10 )
```

The GROUP BY and HAVING Clauses:

→**Group by clause** is used to group the results of a SELECT query based on one or more columns. It is also used with SQL functions to group the result from one or more tables.

→Syntax for using Group by clause is as follows,

```
SELECT    [ DISTINCT ] select-list
FROM      from-list
WHERE     qualification
GROUP BY  grouping-list
HAVING    group-qualification
```

- The select-list in the SELECT clause consists of (1) a list of column names and (2) a list of terms having the form aggop (column-name) AS new-name. The optional AS new-name term gives this column a name in the table that is the result of the query. Any of the aggregation operators can be used for aggop.
- Every column that appears in (1) must also appear in grouping-list. The reason is that each row in the result of the query corresponds to one group, which is a collection of rows that agree on the values of columns in grouping-list. If a column appears in list (1), but not in grouping-list, it is not clear what value should be assigned to it in an answer row.

- The expressions appearing in the group-qualification in the HAVING clause must have a single value per group. The intuition is that the HAVING clause determines whether an answer row is to be generated for a given group. Therefore, a column appearing in the group-qualification must appear as the argument to an aggregation operator, or it must also appear in grouping-list.

- If the GROUP BY clause is omitted, the entire table is regarded as a single group.

For example, consider the following query.

(Q31) Find the age of the youngest sailor for each rating level.

```
SELECT    S.rating, MIN (S.age)
FROM      Sailors S
GROUP BY  S.rating
```

(Q32) Find the age of the youngest sailor who is eligible to vote (i.e., is at least 18 years old) for each rating level with at least two such sailors.

```
SELECT    S.rating, MIN (S.age) AS minage
FROM      Sailors S
WHERE     S.age >= 18
GROUP BY  S.rating
HAVING    COUNT (*) > 1
```

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0
29	Brutus	1	33.0
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35.0
64	Horatio	7	35.0
71	Zorba	10	16.0
74	Horatio	9	35.0
85	Art	3	25.5
95	Bob	3	63.5

Figure 5.10 Instance *S3* of Sailors

rating	*age*
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
9	35.0
3	25.5
3	63.5

Figure 5.11 After Evaluation Step 3

rating	*age*
1	33.0
3	25.5
3	63.5
7	45.0
7	35.0
8	55.5
8	25.5
9	35.0
10	35.0

Figure 5.12 After Evaluation Step 4

rating	*minage*
3	25.5
7	35.0
8	25.5

Figure 5.13 Final Result in Sample Evaluation

More Examples of Aggregate Queries:

(Q33) For each red boat, find the number of reservations for this boat.

```
SELECT    B.bid, COUNT (*) AS sailorcount
FROM      Boats B, Reserves R
WHERE     R.bid = B.bid AND B.color = 'red'
GROUP BY  B.bid
```

```
SELECT    B.bid, COUNT (*) AS sailorcount
FROM      Boats B, Reserves R
WHERE     R.bid = B.bid
GROUP BY  B.bid
HAVING    B.color = 'red'
```

(Q34) Find the average age of sailors for each rating level that has at least two sailors.

```
SELECT    S.rating, AVG (S.age) AS avgage
FROM      Sailors S
GROUP BY  S.rating
HAVING    COUNT (*) > 1
```

```
SELECT    S.rating, AVG ( S.age ) AS avgage
FROM      Sailors S
GROUP BY  S.rating
HAVING    1 < ( SELECT COUNT (*)
                FROM   Sailors S2
                WHERE  S.rating = S2.rating )
```

→After identifying groups based on rating, we retain only groups with at least two sailors. The answer to this query on instance S3 is shown in **Figure 5.14.**

rating	avgage
3	44.5
7	40.0
8	40.5
10	25.5

Figure 5.14 Q34 Answer

rating	avgage
3	45.5
7	40.0
8	40.5
10	35.0

Figure 5.15 Q35 Answer

rating	avgage
3	45.5
7	40.0
8	40.5

Figure 5.16 Q36 Answer

(Q35) Find the average age of sailors who are of voting age (i.e., at least 18 years old) for each rating level that has at least two sailors.

```
SELECT    S.rating, AVG ( S.age ) AS avgage
FROM      Sailors S
WHERE     S. age >= 18
GROUP BY  S.rating
HAVING    1 < ( SELECT COUNT (*)
                FROM   Sailors S2
                WHERE  S.rating = S2.rating )
```

(Q36) Find the average age of sailors who are of voting age (i.e., at least 18 years old) for each rating level that has at least two such sailors.

```
SELECT    S.rating, AVG ( S.age ) AS avgage
FROM      Sailors S
WHERE     S. age > 18
GROUP BY  S.rating
HAVING    1 < ( SELECT COUNT (*)
                FROM   Sailors S2
                WHERE  S.rating = S2.rating AND S2.age >= 18 )
```

```
SELECT     S.rating, AVG ( S.age ) AS avgage
FROM       Sailors S
WHERE      S. age >= 18
GROUP BY   S.rating
HAVING     COUNT (*) >= 1
```

```
SELECT  Temp.rating, Temp.avgage
FROM    ( SELECT     S.rating, AVG ( S.age ) AS avgage,
                     COUNT (*) AS ratingcount
          FROM       Sailors S
          WHERE      S. age >= 18
          GROUP BY   S.rating ) AS Temp
WHERE  Temp.ratingcount >= 1
```

(Q37) Find those ratings for which the average age of sailors is the minimum overall ratings.

```
SELECT     S.rating
FROM       Sailors S
WHERE      AVG (S.age) = ( SELECT    MIN (AVG (S2.age))
                           FROM      Sailors S2
                           GROUP BY  S2.rating )
```

```
SELECT  Temp.rating, Temp.avgage
FROM    ( SELECT    S.rating, AVG (S.age) AS avgage,
          FROM      Sailors S
          GROUP BY S.rating) AS Temp
WHERE   Temp.avgage = ( SELECT MIN (Temp.avgage) FROM Temp )
```

The answer to this query on instance $S3$ is $\langle 10, 25.5 \rangle$.

As an exercise, the reader should consider whether the following query computes the same answer, and if not, why:

```
SELECT    Temp.rating, MIN ( Temp.avgage )
FROM      ( SELECT    S.rating, AVG (S.age) AS avgage,
            FROM      Sailors S
            GROUP BY  S.rating ) AS Temp
GROUP BY Temp.rating
```

5. NULL VALUES:

→The SQL **NULL** is the term used to represent a missing value. A NULL value in a table is a value in a field that appears to be blank.

→A field with a NULL value is a field with no value. It is very important to understand that a NULL value is different than a zero value or a field that contains spaces.
→The basic syntax of **NULL** while creating a table:

```
SQL> CTREATE TABLE CUSTOMERS (
        ID          INT             NOT NULL,
        NAME        VARCHAR2 (20)    NOT NULL,
        AGE         INT             NOT NULL,
       ADDRESS      CHAR (25),
       SALARY       DECIMAL (18, 2),
       PRIMARY KEY (ID)
);
```

→Here, **NOT NULL** signifies that column should always accept an explicit value of the given data type. There are two columns, where we did not use NOT NULL, which means these columns could be NULL.
→A field with a NULL value is one that has been left blank during record creation.

Example:

The NULL value can cause problems when selecting data, however, →because when comparing an unknown value to any other value, the result is always unknown and not included in the final results.
→You must use the **IS NULL** or **IS NOT NULL** operators in order to check for a NULL value.
→Consider the following table, CUSTOMERS having the following records:

ID	NAME	AGE	ADDRESS	SALARY
1	ANUPAMA	52	Ahmadabad	2000.00
2	ABEELA	49	Delhi	1500.00
3	RANIA	45	Kota	2000.00
4	KAVYA	47	Mumbai	6500.00
5	NAGINA	41	Bhopal	8500.00
6	NAJAH	48	Jaipur	
7	RAMEESHA	50	Indore	

→Now, following is the usage of **IS NOT NULL** operator:

```
SQL> SELECT ID, NAME, AGE, ADDRESS, SALARY
     FROM CUSTOMERS
     WHERE SALARY IS NOT NULL;
```

→This would produce the following result:

ID	NAME	AGE	ADDRESS	SALARY
1	ANUPAMA	52	Ahmadabad	2000.00
2	ABEELA	49	Delhi	1500.00
3	RANIA	45	Kota	2000.00
4	KAVYA	47	Mumbai	6500.00
5	NAGINA	41	Bhopal	8500.00

→Now, following is the usage of **IS NULL** operator:

```
SQL> SELECT ID, NAME, AGE, ADDRESS, SALARY
     FROM CUSTOMERS
     WHERE SALARY IS  NULL;
```

→This would produce the following result:

ID	NAME	AGE	ADDRESS	SALARY
6	NAJAH	48	Jaipur	
7	RAMEESHA	50	Indore	

1. Comparison Using Null Values

→It is difficult to perform comparison of valid values with **NULL** values if two valued Logic **TRUE** or **FALSE** is used. Therefore to avoid this issue three valued logic **TRUE, FALSE** or **UNKNOWN** must be used with **NULL** value.

→Consider a comparison such as **rating = 8.** If this is applied to the row for **Dan**, is this condition **TRUE** or **FALSE?** Since **Dan's rating** is UNKNOWN, it is reasonable to say that this comparison should evaluate to the value **UNKNOWN.** In fact, this is the case for the comparisons rating > 8 and rating < 8 as well. If we compare two null values using <, <, =, <> and so on, the result is always **UNKNOWN.**

→For example, if we have null in two distinct rows of the sailor relation, any comparison returns **UNKNOWN.**

→SQL provides a special comparison operator **IS NULL** to test whether a column value is null; for example, we can say **rating IS NULL**, which would evaluate to **TRUE** on the row representing **Dan**. We can also say **rating IS NOT NULL**, which would evaluate to **FALSE** on the row for **Dan.**

2. Logical Connectives AND, OR, and NOT:

→Logical connectives with **NULL** values must be defined using three valued logic wherein expressions evaluates to three values (i.e., to **TRUE, FALSE,** or **UNKNOWN**).

→Now, the Boolean expressions such as **rating = 8 OR age < 40 and rating = 8 AND age < 40.** Considering the row for **Danage < 40**, the first expression evaluates to **TRUE** the value of **rating**; the second can only say **UNKNOWN.**

→The given **table** will give you a better understanding of logical operators when used with **NULL** values. Point to note here is that we are using a three valued logic **TRUE, FALSE** or **UNKNOWN** i.e., the logical condition applied may evaluate to any one of them (**UNKNOWN** is used in case of **NULL** values).

S.No.	Operation	Result	Reason
1)	X and Y	TRUE	If both X and Y are TRUE
		FALSE	If either X or Y is FALSE
		UNKNOWN	If either X or Y is UNKNOWN (NULL values)
2)	X or Y	TRUE	If either of them (X or Y) is TRUE
		FALSE	If both of them are FALSE
		UNKNOWN	If one of the arguments is FALSE and other is UNKNOWN
3)	NOT X	TRUE	If X is FALSE
		FALSE	If X is TRUE
		UNKNOWN	If X is UNKNOWN

Table. Logical Operators

3. Impact on SQL Constructs:

→As many Boolean expressions are used in SQL, it is necessary to understand the impact of NULL values on these constructs.

STUDENT_ID	STD_NAME	COURSE_ID	CLASS	GROUP
1	A	101	2	B
2	B	102	3	B
3	C	103	2	B
4	D	104	4	B
5	E	105	5	B
6	F	106	3	B
7	G	107	6	B

Table. Student Table

→Example:
List all names of students who belongs to group 'B'

```
SELECT   *
FROM     STUDENT S
WHERES.group = 'B';
```

→This solution will result in the set of tuples that satisfies the '**WHERE'** condition and all other tuples that does not satisfy this condition are ignored in addition to these tuples. Tuples with **NULL** values are also ignored because for them the condition evaluates to **FALSE** or **UNKNOWN**. This elimination of rows that resulted unknown, makes the queries that involves **EXISTS** and/or **UNIQUE** much more simple, easy to understand and makes the evaluation of these queries (nested queries especially) much easier.
→We know that the comparison of any two fields with NULL values for equality is an **UNKNOWN** value. But when it comes to **(=)** equality operator, the two **NULL** value attributes are treated as equal. If a field contains two **NULL** values then that is considered as duplicate values. Two tuples are said to be duplicates if they hold the same value or if they hold **NULL** values. So, the comparison of **NULL** values with the **"="** operator always results in **TRUE**.
→The result of all the **arithmetic operators (+, -, %, /, *)** results in an **UNKNOWN** value (**NULL**) if any one of the argument is a NULL value. Similarly, with all the aggregate operators the result is NULL if these operators are applied a NULL value. Aggregate functions simply delete the **NULL** values and then returns the result of aggregate operators i.e., **SUM, AVG, MIN, MAX, COUNT(DISTICT)** i.e., simply delete/ignore the NULL values and returns the result of other **NOT NULL** tuples. Only exception in aggregate operator is **COUNT (*)** which does not ignore/delete the **NULL** values, it counts them and then return the number of tuples in the table.

4. Outer Joins: (**********)
→We need to use outer joins to include all the tuples from the participating relations in the resulting relation.
→This is the special case of **"join"** operator which considers the NULL values. Generally **"join**" operations performs the cross product of two tables and apply certain join condition. Then it selects those rows from the cross product that satisfied the given condition. But with outer joins, DBMS allows to us select those rows which are common (satisfies the given) and even those rows that does not satisfies the given condition.

→To understand this, consider simple **instances of Project and Department** as shown in **table.**

DEPARTMENT D1			PROJECT P1	
Dept_id	Dept_no	Project_no	Project_no	Project_name
100001	16	111	444	K
100002	4	222	111	N
100003	14	333	222	R

TABLE . Intances of PROJECT and DEPARTMENT Table.

→If we perform join operation on these two tables,

```
SELECT   * D1, * P1
FROM     DEPARTMENT D1, PROJECT P1
WHERE    D1.Project_no = P1.Project_no;
```

→The result of this statement is shown in **Table 1.**

Dept_id	Dept_no	Project_no	Project_no	Project_name
100001	16	111	111	N
100002	4	222	222	R

TABLE 1. Table Showing the Simple Join Operation.

→The **Table 1** shows the simple join operation of two tables, only those rows are selected that satisfied the condition. However, if we want to include those rows that do not satisfy the condition, then we can use the concept of **OUTER JOINS**.

→There are three types of **OUTER JOINS**. They are,

1. LEFT OUTER JOIN
2. RIGHT OUTER JOIN
3. FULL OUTER JOIN

1. LEFT OUTER JOIN:

→**LEFT OUTER JOIN** lists all those rows which are common to both the tables and also all those unmatched rows of the table which is specified at the left hand side.

Example:

```
SELECT   * D1, * P1
FROM     DEPARTMENT D1   LEFT OUTER JOIN PROJECT P1
WHERE    D1.Project_no = P1.Project_no;
```

→The result of this statement is shown in **Table 1A.**

DEPARTMENT D1			PROJECT P1	
Dept_id	Dept_no	Project_no	Project_no	Project_name
100001	16	111	111	N
100002	4	222	222	R
100003	14	333	NULL	NULL

TABLE 1A. Table Showing the LEFT OUTER JOIN Operation.

→So, the **LEFT OUTER JOIN** resulted in relations that have common rows from both the tables and also the row which does not have match in the other table. The values of the attributes corresponding to second table are **NULL** values.

2. RIGHT OUTER JOIN:

→**RIGHT OUTER JOIN** is same as the **LEFT OUTER JOIN** but the only difference is the unmatched rows of second table (specified on the right hand side) are listed along with the common rows of both the tables.

```
SELECT   * D1, * P1
FROM     DEPARTMENT D1   RIGHT OUTER JOIN PROJECT P1
WHERE    D1.Project_no = P1.Project_no;
```

→The result of this statement is shown in **Table 2B.**

DEPARTMENT D1			PROJECT P1	
Dept_id	Dept_no	Project_no	Project_no	Project_name
NULL	NULL	NULL	444	K
100001	16	111	111	N
100002	4	222	222	R

TABLE 2B. Table Showing the RIGHT OUTER JOIN.

→The values of attributes for the first table are declared as **NULL**.

3. FULL OUTER JOIN:

→**FULL OUTER JOIN** is same as the **RIGHT OUTER JOIN** and **LEFT OUTER JOIN** but only difference is unmatched rows of both tables are listed along with the common rows of the tables.

```
SELECT   * D1, * P1
FROM     DEPARTMENT D1   FULL OUTER JOIN PROJECT P1
WHERE    D1.Project_no = P1.Project_no;
```

→The result of this statement is shown in **Table 2C.**

DEPARTMENT D1			PROJECT P1	
Dept_id	Dept_no	Project_no	Project_no	Project_name
100001	16	111	111	N
100002	4	222	222	R
100003	14	333	NULL	NULL
NULL	NULL	NULL	444	K

TABLE 3C. Table Showing the FULL OUTER JOIN.

→In this relation as you can see all the matched and unmatched columns of both the tables are displayed, the values for the unmatched attributes are entered as **NULL**.

5. Disallowing Null Values:

→These fields can take on NULL values, if they are not declared as **NOT NULL**. We can restrict the insertion of NULL values for the field by declaring that field as **NOT NULL**. This means that the field cannot take **NULL** values. For the **PRIMARY KEY** Constraint i.e., the field which is declared as **PRIMARY KEY** is also declared as **NOT NULL**. This declaration is implicit declaration done by DBMS.

```
CREATE TABLE STUDENT (Sid INT NOT NULL,
                      Sname CHAR(10) NOT NULL
                      Project VARCHAR2 (15),
                      Class INT,
                      PRIMARY KEY(Sid));
```

→In this declaration i.e., creation of **STUDENT Table,** Sid is the **PRIMARY KEY** hence it must be UNIQUE and it should not be **NULL**. Project field indicates the Project taken up by the student .This field can take **NULL** values as it is possible

6. EmbeddedSQL: (**********)

→**SQL** provides a powerful declarative query language. Writing queries in SQL is usually much easier than coding the same queries in a general - purpose programming language. However, a programmer must have access to a database from a general purpose programming language for at least two reasons:

- Not all queries can be expressed in SQL, since SQL does not provide the full expressive power of a general-purpose language. That is, there exist queries that can be expressed in a language such as C, Java, or COBOL that cannot be expressed in SQL. To write such queries, we can embed SQL within a more powerful language.

- Non-declarative actions—such as printing a report, interacting with a user, or sending the results of a query to a graphical user interface—cannot be done from within SQL. Applications usually have several components, and querying or updating data is only one component; other components are written in general-purpose programming languages. For an integrated application, the programs written in the programming language must be able to access the database.

1. Declaring Variables and Exceptions:

→The SQL standard defines embeddings of SQL in a variety of programming languages such as **C, Java,** and **COBOL**.

→A language to which SQL queries are embedded is referred to as a **host language**, and the SQL structures permitted in the host language comprise embedded SQL.

→SQL statements can refer to variables defined in the host program. Such host-language variables must be prefixed by a colon (:) in SQL statements and must be declared between the commands **EXEC SQL BEGIN DECLARE SECTION** and **EXEC SQL END DECLARE SECTION.** The declarations are similar to how they would look in a C program and, as usual in C, are separated by semicolons.

→**For example,** we can declare variables **c_sname, c_sid, c_rating, and c_age** (with the initial c used as a naming convention to emphasize that these are host language variables) as follows:

```
EXEC SQL BEGIN DECLARE SECTION
char c_sname[20];
long c_sid;
short c_rating;
float c_age;
EXEC SQL END DECLARE SECTION
```

2. Embedding SQL Statements:

→All SQL statements that are embedded within a host program must be clearly marked, with the details dependent on the host language; in **C**, SQL statements must be pre- fixed by **EXEC SQL**. An SQL statement can essentially appear in any place in the host language program where a host language statement can appear.

→**As a simple example**, the following embedded SQL statement inserts a row, whose column values are based on the values of the host language variables contained in it, into the **Sailors relation**:

```
EXEC SQL INSERT INTO Sailors VALUES (:c_sname, :c_sid, :c_rating, :c_age);
```

→The **SQLSTATE** variable should be checked for errors and exceptions after each embedded SQL statement. SQL provides the **WHENEVER** command to simplify this tedious task:

```
EXEC SQL WHENEVER [ SQLERROR | NOT FOUND ] [ CONTINUE | GOTO stmt ]
```

→The intent is that after each embedded SQL statement is executed, the value of **SQLSTATE** should be checked. If **SQLERROR** is specified and the value of **SQLSTATE** indicates an exception, control is transferred to stmt, which is presumably responsible for error/exception handling. Control is also transferred to stmt if **NOT FOUND** is specified and the value of S**QLSTATE** is 02000, which denotes **NO DATA**.

7. Dynamic SQL: (****************)

→The dynamic SQL component of SQL allows programs to construct and submit SQL queries at run time.

→Using dynamic SQL, programs can create SQL queries as strings at run time (perhaps based oninput from the user) and can either have them executed immediately or have they prepared for subsequent use. Preparing a dynamic SQL statement compiles it, and subsequent uses of the prepared statement use the compiled version.

→SQL defines standards for embedding dynamic SQL calls in a host language, such as C, as in the following example.

```
Char * sqlprog = " update account set balance = balance*1.05
                  where account_number =?"
EXEC SQL PREPARE dynprog from: sqlprog;
Char account [10] ="A-101";
EXEC SQL EXECUTE dynprog using: account;
```

→The dynamic SQL program contains a? which is a place holder for a value that is provided when the SQL program is executed?

8. CURSORS: (************)

→A major problem in embedding SQL statements in a host language like C is that an impedance mismatch occurs because SQL operates on sets of records, whereas languages like C do not cleanly support a set-of-records abstraction. The solution is to essentially provide a mechanism that allows us to retrieve rows one at a time from a relation. This mechanism is called a **cursor.**

→A **cursor** is a temporary work area created in the system memory when a SQL statement is executed. A **cursor** contains information on a select statement and the rows of data accessed by it.

→This temporary work area is used to store the data retrieved from the database, and manipulate this data. A **cursor** can hold more than one row, but can process only one row at a time. The set of rows the cursor holds is called the ***active* set**.

→We can declare a **cursor** on any relation or on any SQL query (because every query returns a set of rows).

→Once a **cursor** is **declared**, we can **open** it (which positions the cursor just before the first row); **fetch** the next row; **move** the cursor (to the next row, to the row after the next n, to the first row, or to the previous row, etc., by specifying additional parameters for the **FETCH** command); or close the cursor.

1. Basic Cursor Definition and Usage:

→Cursors enable us to examine in the host language program a collection of rows computed by an embedded SQL statement:

- We usually need to open a cursor if the embedded statement is a **SELECT** (i.e., a query)..
- **INSERT, DELETE,** and **UPDATE** statements typically don't require a cursor, although some variants of DELETE and UPDATE do use a cursor.

→**As an example,** we can **find the name and age of a sailor**, specified by assigning a value to the host variable **c_sid** as follows:

```
EXEC SQL SELECT S.sname, S.age
         INTO    :c_sname, :c_age
         FROM    Sailors S
         WHERE   S.sid = :c_sid;
```

→The INTO clause allows us to assign the columns of the single answer row to the host variables **c_sname** and **c_age.**

→Computes the names and ages of all sailors with a rating greater than the current value of the host variable c_minrating?

```
SELECT S.sname, S.age
FROM   Sailors S
WHERE  S.rating > :c_minrating
```

→This query returns a collection of rows, not just one row. The solution is to use a **cursor:**

```
DECLARE sinfo CURSOR FOR
SELECT S.sname, S.age
FROM   Sailors S
WHERE  S.rating > :c_minrating;
```

→This code can be included in a C program, and once it is executed, the cursor **sinfo** is defined. Subsequently, we can **open the cursor**:

```
OPEN sinfo;
```

→ We can use the **FETCH** command to read the first row of **cursorsinfo** into host language variables:

```
FETCH sinfo INTO :c_sname, :c_age;
```

→When we are done with a cursor**, we can close it:**

```
CLOSE sinfo;
```

2. Properties of Cursors:

→The general form of a cursor declaration is:

```
DECLARE cursorname [INSENSITIVE] [SCROLL] CURSOR FOR
        some query
        [ ORDER BY order-item-list ]
        [ FOR READ ONLY | FOR UPDATE ]
```

→ A **cursor** can be declared to be a read-only cursor (**FOR READ ONLY**) or, if it is a cursor on a base relation or an updatable view, to be an updatable cursor (**FOR UPDATE**).
→If it is updatable, simple variants of the **UPDATE** and **DELETE** commands allow us to update or delete the row on which the cursor is positioned.

→**For example**, if **sinfo** is an updatable cursor and is open, we can execute the following statement:

```
UPDATE Sailors S
SET    S.rating = S.rating - 1
WHERE  CURRENT of sinfo;
```

→ This embedded SQL statement **modifies** the rating value of the row currently pointed to by cursor sinfo; similarly, we can delete this row by executing the next statement:

```
DELETE Sailors S
WHERE  CURRENT of sinfo;
```

→A cursor is updatable by default unless it is a **scrollable** or **insensitive cursor**, in which case it is read-only by default.

→If the keyword **SCROLL** is specified, the cursor is scrollable, which means that variants of the FETCH command can be used to position the cursor in very flexible ways; otherwise, only the basic FETCH command, which retrieves the next row, is allowed
→If the keyword **INSENSITIVE** is specified, the cursor behaves as if it is ranging over a private copy of the collection of answer rows.

→**For example,** while we are fetching rows using the sinfo cursor, we might modify rating values in Sailor rows by concurrently executing the command:

```
UPDATE Sailors S
SET    S.rating = S.rating - 1
```

→The order-item-list is a list of order-items; an order-item is a column name, optionally followed by one of the keywords ASC or DESC.

→Suppose that a cursor is opened on this query, with the clause:

```
ORDER BY minage ASC, rating DESC
```

The answer is sorted first in ascending order by **minage**, and if several rows have the same **minage** value, these rows are sorted further in descending order by rating. The cursor would fetch the rows in the order shown in **Figure 5.18.**

rating	*minage*
8	25.5
3	25.5
7	35.0

Figure 5.18 Order In which Tuples Are Fetched

9. COMPLEX INTEGRITY CONSTRAINTS IN SQL:

Integrity constraints need not only be applied on single columns, they can also be applied on single table or group of tables (called **assertions**).

Constraints over a Single Table:

We can specify complex constraints over a single table using table constraints, which have the form CHECK conditional-expression. For example, to ensure that rating must be an integer in the range 1 to 10, we could use:

```
CREATE TABLE Sailors ( sid     INTEGER,
                       sname   CHAR(10),
                       rating  INTEGER,
                       age     REAL,
                       PRIMARY KEY (sid),
                       CHECK ( rating >= 1 AND rating <= 10 ))
```

To enforce the constraint that Interlake boats cannot be reserved, we could use:

```
CREATE TABLE Reserves ( sid     INTEGER,
                        bid     INTEGER,
                        day     DATE,
                        FOREIGN KEY (sid) REFERENCES Sailors
                        FOREIGN KEY (bid) REFERENCES Boats
                        CONSTRAINT noInterlakeRes
                        CHECK ( 'Interlake' <>
                                ( SELECT B.bname
                                  FROM   Boats B
                                  WHERE  B.bid = Reserves.bid )))
```

When a row is inserted into Reserves or an existing row is modified, the conditional expression in the CHECK constraint is evaluated. If it evaluates to false, the command is rejected.

Domain Constraints: (**********)

→A user can define a new domain using the CREATE DOMAIN statement, which makes use of CHECK constraints.

→The syntax for creating a new domain is,

CREATE DOMAIN Domain_name Source_domain (DEFAULT value) CHECK (VALUE)

CREATE DOMAIN: A statement or keyword used to define a new domain.
Domain_name : Name of the new domain.
Source_domain: Name of the source domain from which new domain is derived.
DEFAULT value: We can also provide default values for the domains.
CHECK: This option is used to restrict the values in the particular field (for which a new domain is specified). This option provides a condition that must be checked by all the tuples of the column.
VALUE: The key word is used to provide a value to a domain variable.

Example:

```
CREATE DOMAIN ratingval INTEGER DEFAULT 0
                    CHECK ( VALUE >= 1 AND VALUE <= 10 )
```

INTEGER is the base type for the domain **ratingval**, and every **ratingval** value must be of this type. Values in **ratingval** are further restricted by using a **CHECK** constraint; in defining this constraint, we use the keyword **VALUE** to refer to a value in the domain.

Assertions: ICs over Several Tables:

→**Assertions** are group of tables on which a constraint is applied. Unlike table constraints which are applied on single table, assertions are applied on multiple tables.

Example:
As an example, suppose that we wish to enforce the constraint that the number of boats plus the number of sailors should be less than 100. We could try the following table constraint:

```
CREATE TABLE Sailors ( sid     INTEGER,
                       sname   CHAR(10),
                       rating  INTEGER,
                       age     REAL,
                       PRIMARY KEY (sid),
                       CHECK ( rating >= 1 AND rating <= 10)
                       CHECK ( ( SELECT COUNT (S.sid) FROM Sailors S )
                               + ( SELECT COUNT (B.bid) FROM Boats B )
                               < 100 ))
```

→This solution suffers from two drawbacks. It is associated with **Sailors**, although it involves **Boats** in a completely symmetric way. More important, if the **Sailors** table is empty, this constraint is defined (as per the semantics of table constraints) to always hold, even if we have more than 100 rows in **Boats**! We could extend this constraint specification to check that Sailors is nonempty, but this approach becomes very cumbersome. The best solution is to create an **assertion**, as follows:

```
CREATE ASSERTION smallClub
CHECK ( ( SELECT COUNT (S.sid) FROM Sailors S )
        + ( SELECT COUNT (B.bid) FROM Boats B)
        < 100 )
```

10. TRIGGERS AND ACTIVE DATABASES:(*****)

→A **trigger** is a procedure that is automatically invoked by the DBMS in response to specified changes to the database, and is typically specified by the DBA.
→A database that has a set of associated triggers is called an **active database.**
→A **trigger** description contains three parts:

1. **Event**
2. **Condition**
3. **Action**

1. **Event:** A change to the database that activates the trigger.

→Event describes the modifications done to the database which lead to the activation of trigger. The following are fall under the category of events,

i) Inserting, updating, deleting columns of the tables or rows of tables may activate the trigger.
ii) Creating, altering or dropping any database object may also lead to activation of triggers.
iii) An error message or user log-on or log-off may also activate the trigger.

2. **Condition:** A query or test that is run when the trigger is activated.

→Conditions are used to specify whether the particular action must be performed or not. If the condition is evaluated to true then the respective action is taken otherwise the action is rejected.

3. **Action:** A procedure that is executed when the trigger is activated and its condition is true.

→The examples shown in **Figure 5.19**
→The trigger called **init_count** initializes a counter variable before every execution of an INSERT statement that adds tuples to the Students relation.
→The trigger called **incr_count** increments the counter for each inserted tuple that satisfies the condition age < 18.

```
CREATE TRIGGER init_count BEFORE INSERT ON Students          /* Event */
    DECLARE
        count INTEGER;
    BEGIN                                                    /* Action */
        count := 0;
    END

CREATE TRIGGER incr_count AFTER INSERT ON Students           /* Event */
    WHEN (new.age < 18)          /* Condition: 'new' is just-inserted tuple */
    FOR EACH ROW
    BEGIN              /* Action; a procedure in Oracle's PL/SQL syntax */
        count := count + 1;
    END
```

Figure 5.19 Examples Illustrating Triggers

→A **row-level trigger** is activated for each modified record, a **statement-level trigger** is activated only once per INSERT command.

Advantages of trigger:

1) Triggers can be used as an alternative method for implementing referential integrity constraints.

2) By using triggers, business rules and transactions are easy to store in database and can be used consistently even if there are future updates to the database.

3) It controls on which updates are allowed in a database.

4) When a change happens in a database a trigger can adjust the change to the entire database.

5) Triggers are used for calling stored procedures.

10. DESIGNING ACTIVE DATABASES:

→Active database contains a set of triggers and therefore it becomes quite difficult to maintain active database.
→What triggers are activated in what order can be hard to understand because a statement can activate more than one trigger and the action of one trigger can activate other triggers.

Why Triggers Can Be Hard to Understand:

→In an active database system, when the DBMS is about to execute a statement that modifies the database, it checks whether some trigger is activated by the statement. If so, the DBMS processes the trigger by evaluating its condition part, and then (if the condition evaluates to true) executing its action part.
→If a statement activates more than one trigger, the DBMS typically processes all of them, in some arbitrary order. The execution of this action part of a trigger may in turn activate another trigger. In particular, the execution of the action part of a trigger could again activate the same trigger; such triggers are called **recursive triggers**. The potential for such **chain activations**, and the unpredictable order in which a DBMS processes **activated triggers**, can make it difficult to understand the effect of a collection of triggers.

Constraints versus Triggers:

→Triggers are more flexible than integrity constraints and the potential uses of triggers go beyond maintaining database integrity.
→Triggers are used to maintain the data integrity in the database. Whenever a change (update, insert or delete) is done in a database, a trigger can be used to indicate that change. There are several uses of triggers.

- To maintain data integrity.
- To identity the unusual events that occurs in a database.
- For security checks and also for auditing.

UNIT IV

→A schema can be defined as a complete description of database. The specifications for database schema are provided during the database design stage and this schema does not change frequently.

→**Schema Refinement** is a technique of organizing the data in the database. It is a systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics like Insertion, Update and Deletion Anomalies.
→**Schema refinement** is the process that re-defines (refining) the schema of a relation so as to solve the problems caused by redundantly storing the information.

→**Redundancy** refers to repetition of same data or duplicate copies of same data stored in different locations.
→The **Schema Refinement** refers to refine the schema by using some technique. The best technique of schema refinement is **decomposition**.
→**Decomposition** can eliminate the redundancy.

1. Problems Caused by Redundancy :((**********)
→Redundancy is a data organization issue. It allows unnecessary duplication of data to be stored within the database. If modifications are performed to redundant data, then it is necessary to perform the same modification in multiple fields of database.
→Storing the same information redundantly, that is, in more than one place within a database, can lead to several problems:

1. **Redundant storage.**
2. **Update anomalies.**
3. **Insertion anomalies.**
4. **Deletion anomalies.**

→Schema diagram for Employee database is as follows,
→**Example:**

Employee Emp_id, Emp_name, Emp_section_id, Job_section, grade.

Emp_id	Emp_name	Emp_section_id	Job_section	grade
100001	ANUPAMA	111	CLERK	D
100002	RAMEESHA	222	Secretary	B
100002	RAMEESHA	222	Secretary	B
100002	RAMEESHA	222	Secretary	B
100003	NAGINA	333	MANAGER	A
100001	ANUPAMA	111	CLERK	D
100004	KAVYA	444	Asst.Manager	C

TABLE. An Instance of the Employee relation.

→Consider the above database **table**. The three tuples with **Emp_id100002** and two tuples with **Emp_id100001** repeat the same name and same job section information. The repetition wastes space as well as causes data inconsistency i.e., this redundant data may lead to **loss of data integrity.**

→For example, some update operation is being carried out, entering new record for an employee with id **100002.** This must be done multiple time i.e., it must be done for each file witch stores the employees details. This leads to redundant storage i.e., the same information is stored multiple times.

1. Redundant storage: Some information is stored repeatedly.

2. Update anomalies: If one copy of such repeated data is updated, an inconsistency is created unless all copies are similarly updated.

→If the update operation is performed, for example, the Emp_section_id 268 is updated to 520 and this correction is made only to the first record of the database, then this may lead to inconsistent data unless all the copies in the database are updated. This is referred to as update anomalies. The changes must be done to all the copies of data.

3. Insertion anomalies: It may not be possible to store some information unless some other information is stored as well.
→For example, if a new employee record is being entered, who has not yet assigned an Emp_id, now if we assume that the null values are not allowed, then it impossible to enter the new record unless the new employee has been assigned an Emp_id. This is called insertion anomalies.

4. Deletion anomalies: It may not be possible to delete some information without losing some other information as well.
→For example, if we want to delete the grade entries where grade is equal to 'A' then all the information of Emp_section_id 268 will be deleted/loss.

2. Use of Decompositions:
→**Decomposion** is the solution to the problem caused by data redundancy. **Decomposition** means breaking up the large schema into smaller multiple Schemas. **Decomposition** helps to remove all the anomalies and helps to maintain data integrity.

→We can restrict redundancy in **Employee** database by dividing it into two smaller relations/Schemas as in **table1R** and **Table2R.**

→Now we can easily update **Emp_section_id** in the **Schema Section** without bothering about the updations in the other tuples. To insert a new tuple, we can directly insert the new record in the Schema section (With the help of **Emp_section-id**) even if the new employee has not yet been assigned the **Emp_id**. To delete the entry with the **grade** equal to 'A', we can do it directly on the **Section schema** which does not lead to loss of other information. Thus, **decomposioneliminates the Problems caused by different anomalies**

Emp_id	Emp_name	Job_section	grade
100001	ANUPAMA	CLERK	D
100002	RAMEESHA	Secretary	B
100002	RAMEESHA	Secretary	B
100002	RAMEESHA	Secretary	B
100003	NAGINA	MANAGER	A
100001	ANUPAMA	CLERK	D
100004	KAVYA	Asst.Manager	C

TABLE1R. An Instance of the Employee relation.

Emp_section_id	grade
100001	D
100002	B
100003	A
100004	C

TABLE 2R. An Instance of the Section Relation

Functional Dependency

The functional dependency is a relationship that exists between two attributes. It typically exists between the primary key and non-key attribute within a table.

1. X → Y

The left side of FD is known as a determinant, the right side of the production is known as a dependent.

For example:

Assume we have an employee table with attributes: Emp_Id, Emp_Name, Emp_Address.

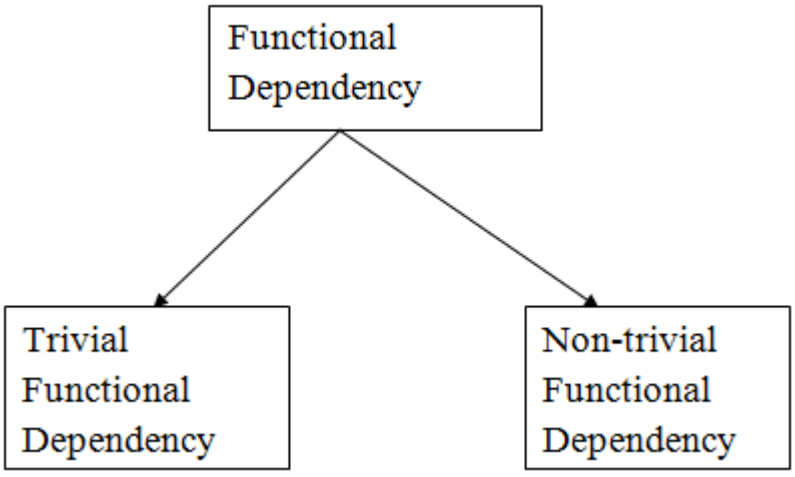
Here Emp_Id attribute can uniquely identify the Emp_Name attribute of employee table because if we know the Emp_Id, we can tell that employee name associated with it.

Functional dependency can be written as:

1. Emp_Id → Emp_Name

We can say that Emp_Name is functionally dependent on Emp_Id.

Types of Functional dependency



1. Trivial functional dependency

- A → B has trivial functional dependency if B is a subset of A.
- The following dependencies are also trivial like: A → A, B → B

Example:

1. Consider a table with two columns Employee_Id and Employee_Name.
2. {Employee_id, Employee_Name} → Employee_Id is a trivial functional dependency as
3. Employee_Id is a subset of {Employee_Id, Employee_Name}.
4. Also, Employee_Id → Employee_Id and Employee_Name → Employee_Name are trivial dependencies too.

2. Non-trivial functional dependency

- A → B has a non-trivial functional dependency if B is not a subset of A.
- When A intersection B is NULL, then A → B is called as complete non-trivial.

Example:

1. ID → Name,
2. Name → DOB

Inference Rule (IR):

- The Armstrong's axioms are the basic inference rule.
- Armstrong's axioms are used to conclude functional dependencies on a relational database.
- The inference rule is a type of assertion. It can apply to a set of FD(functional dependency) to derive other FD.
- Using the inference rule, we can derive additional functional dependency from the initial set.

The Functional dependency has 6 types of inference rule:

1. Reflexive Rule (IR_1)

In the reflexive rule, if Y is a subset of X, then X determines Y.

1. If X ⊇ Y then X → Y

Example:

1. X = {a, b, c, d, e}
2. Y = {a, b, c}

2. Augmentation Rule (IR_2)

The augmentation is also called as a partial dependency. In augmentation, if X determines Y, then XZ determines YZ for any Z.

1. If X → Y then XZ → YZ

Example:

1. For R(ABCD), **if** A → B then AC → BC

3. Transitive Rule (IR_3)

In the transitive rule, if X determines Y and Y determine Z, then X must also determine Z.

1. If X → Y and Y → Z then X → Z

4. Union Rule (IR_4)

Union rule says, if X determines Y and X determines Z, then X must also determine Y and Z.

1. If X → Y and X → Z then X → YZ

Proof:

1. X → Y (given)
2. X → Z (given)
3. X → XY (using IR_2 on 1 by augmentation with X. Where XX = X)
4. XY → YZ (using IR_2 on 2 by augmentation with Y)
5. X → YZ (using IR_3 on 3 and 4)

5. Decomposition Rule (IR_5)

Decomposition rule is also known as project rule. It is the reverse of union rule.

This Rule says, if X determines Y and Z, then X determines Y and X determines Z separately.

1. If X → YZ then X → Y and X → Z

Proof:

1. X → YZ (given)
2. YZ → Y (using IR_1 Rule)
3. X → Y (using IR_3 on 1 and 2)

6. Pseudo transitive Rule (IR_6)

In Pseudo transitive Rule, if X determines Y and YZ determines W, then XZ determines W.

1. If X → Y and YZ → W then XZ → W

Proof:

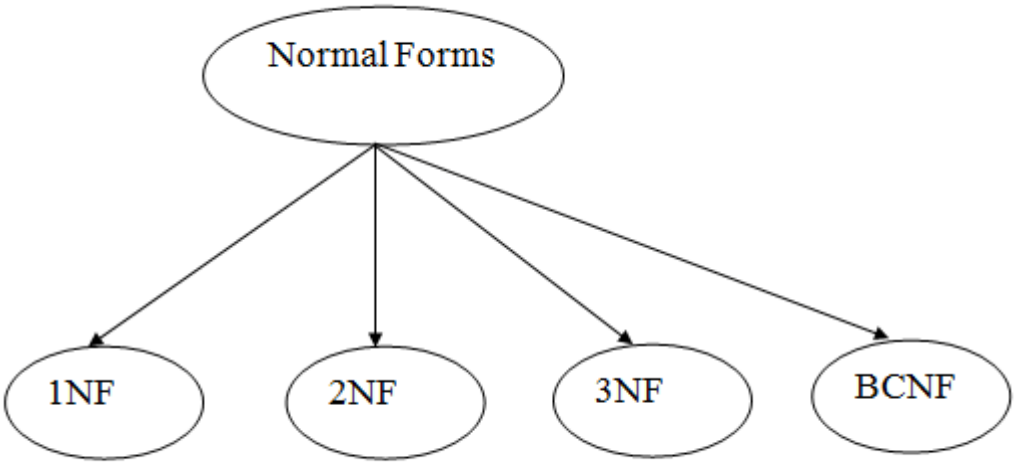
1. X → Y (given)
2. WY → Z (given)
3. WX → WY (using IR_2 on 1 by augmenting with W)
4. WX → Z (using IR_3 on 3 and 2)

Normalization

- Normalization is the process of organizing the data in the database.
- Normalization is used to minimize the redundancy from a relation or set of relations. It is also used to eliminate the undesirable characteristics like Insertion, Update and Deletion Anomalies.
- Normalization divides the larger table into the smaller table and links them using relationship.
- The normal form is used to reduce redundancy from the database table.

Types of Normal Forms

There are the four types of normal forms:



Normal Form	Description
1NF	A relation is in 1NF if it contains an atomic value.
2NF	A relation will be in 2NF if it is in 1NF and all non-key attributes are fully functional dependent on the primary key.
3NF	A relation will be in 3NF if it is in 2NF and no transition dependency exists.
4NF	A relation will be in 4NF if it is in Boyce Codd normal form and has no multi-valued dependency.
5NF	A relation is in 5NF if it is in 4NF and not contains any join dependency and joining should be lossless.

First Normal Form (1NF)

- A relation will be 1NF if it contains an atomic value.
- It states that an attribute of a table cannot hold multiple values. It must hold only single-valued attribute.
- First normal form disallows the multi-valued attribute, composite attribute, and their combinations.

Example: Relation EMPLOYEE is not in 1NF because of multi-valued attribute EMP_PHONE.

EMPLOYEE table:

EMP_ID	EMP_NAME	EMP_PHONE	EMP_STATE
14	John	7272826385, 9064738238	UP
20	Harry	8574783832	Bihar
12	Sam	7390372389, 8589830302	Punjab

The decomposition of the EMPLOYEE table into 1NF has been shown below:

EMP_ID	EMP_NAME	EMP_PHONE	EMP_STATE

14	John	7272826385	UP
14	John	9064738238	UP
20	Harry	8574783832	Bihar
12	Sam	7390372389	Punjab
12	Sam	8589830302	Punjab

Second Normal Form (2NF)

- In the 2NF, relational must be in 1NF.
- In the second normal form, all non-key attributes are fully functional dependent on the primary key

Example: Let's assume, a school can store the data of teachers and the subjects they teach. In a school, a teacher can teach more than one subject.

TEACHER table

TEACHER_ID	SUBJECT	TEACHER_AGE
25	Chemistry	30
25	Biology	30
47	English	35
83	Math	38
83	Computer	38

In the given table, non-prime attribute TEACHER_AGE is dependent on TEACHER_ID which is a proper subset of a candidate key. That's why it violates the rule for 2NF.

To convert the given table into 2NF, we decompose it into two tables:

TEACHER_DETAIL table:

TEACHER_ID	TEACHER_AGE
25	30

47	35
83	38

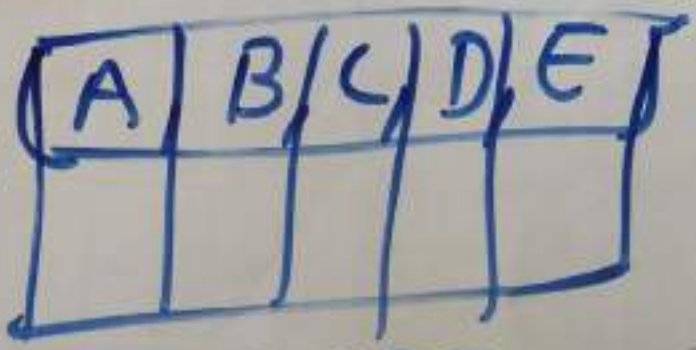
TEACHER_SUBJECT table:

TEACHER_ID	SUBJECT
25	Chemistry
25	Biology
47	English
83	Math
83	Computer

2NF:

1. Satisfying the 1NF
2. Every non key Attribute is Fully Functionally dependent on any key of Relation.

R



FD: A → BCDE

FD: BC → ADE

FD: C → E ✓

Keys: A, BC

Non-keys: D, E

R(A,B,C,D,E) A → BCDE A, BC

BC → ADE

C → E X

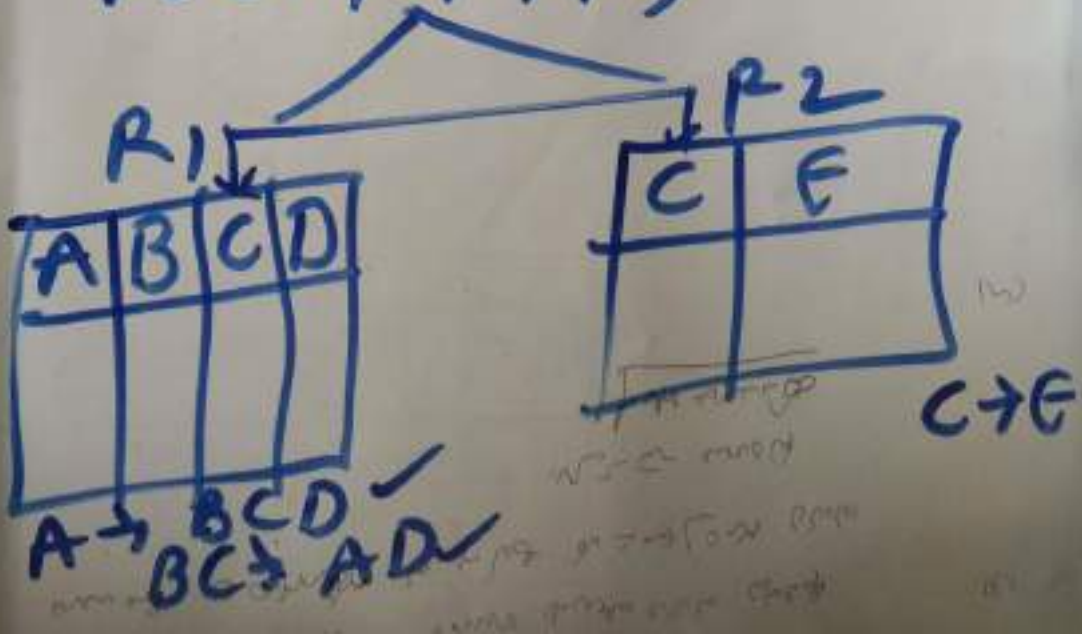
FULLY FD ✓

If we are using total set of keys

partial FD

subset of key is used to find the data

R(A,B,C,D,E)



Third Normal Form (3NF)

- A relation will be in 3NF if it is in 2NF and not contain any transitive partial dependency.
- 3NF is used to reduce the data duplication. It is also used to achieve the data integrity.
- If there is no transitive dependency for non-prime attributes, then the relation must be in third normal form.

A relation is in third normal form if it holds atleast one of the following conditions for every non-trivial function dependency X → Y.

1. X is a super key.
2. Y is a prime attribute, i.e., each element of Y is part of some candidate key.

Example:

EMPLOYEE_DETAIL table:

EMP_ID	EMP_NAME	EMP_ZIP	EMP_STATE	EMP_CITY
222	Harry	201010	UP	Noida
333	Stephan	02228	US	Boston
444	Lan	60007	US	Chicago
555	Katharine	06389	UK	Norwich
666	John	462007	MP	Bhopal

Super key in the table above:

1. {EMP_ID}, {EMP_ID, EMP_NAME}, {EMP_ID, EMP_NAME, EMP_ZIP}....so on

Candidate key: {EMP_ID}

Non-prime attributes: In the given table, all attributes except EMP_ID are non-prime.

Here, EMP_STATE & EMP_CITY dependent on EMP_ZIP and EMP_ZIP dependent on EMP_ID. The non-prime attributes (EMP_STATE, EMP_CITY) transitively dependent on super key(EMP_ID). It violates the rule of third normal form.

That's why we need to move the EMP_CITY and EMP_STATE to the new <EMPLOYEE_ZIP> table, with EMP_ZIP as a Primary key.

EMPLOYEE table:

EMP_ID	EMP_NAME	EMP_ZIP
222	Harry	201010
333	Stephan	02228
444	Lan	60007
555	Katharine	06389
666	John	462007

EMPLOYEE_ZIP table:

EMP_ZIP	EMP_STATE	EMP_CITY
201010	UP	Noida
02228	US	Boston
60007	US	Chicago
06389	UK	Norwich
462007	MP	Bhopal

3NF:

- if it is said to be 3NF FIRST satisfies the 2NF
- and NO NON Prime attribute should be transitively dependent on candidate key.

(or)

- should not occur the case one non-Prime attribute determines another NPA

3NF Ex:

R(sno, sname, city, DISTRICT, STATE, ZIP)

FD: sno → sname, CITY, DISTRICT, STATE, ZIP

BUT ZIP → CITY, DISTRICT, STATE

So TABLE DIVIDED INTO TWO TABLES.

R1

(sno, sname, ZIP)

R2

(ZIP, CITY, DISTRICT, CITY)

Ex:

R1 (A, B, C, D, E)

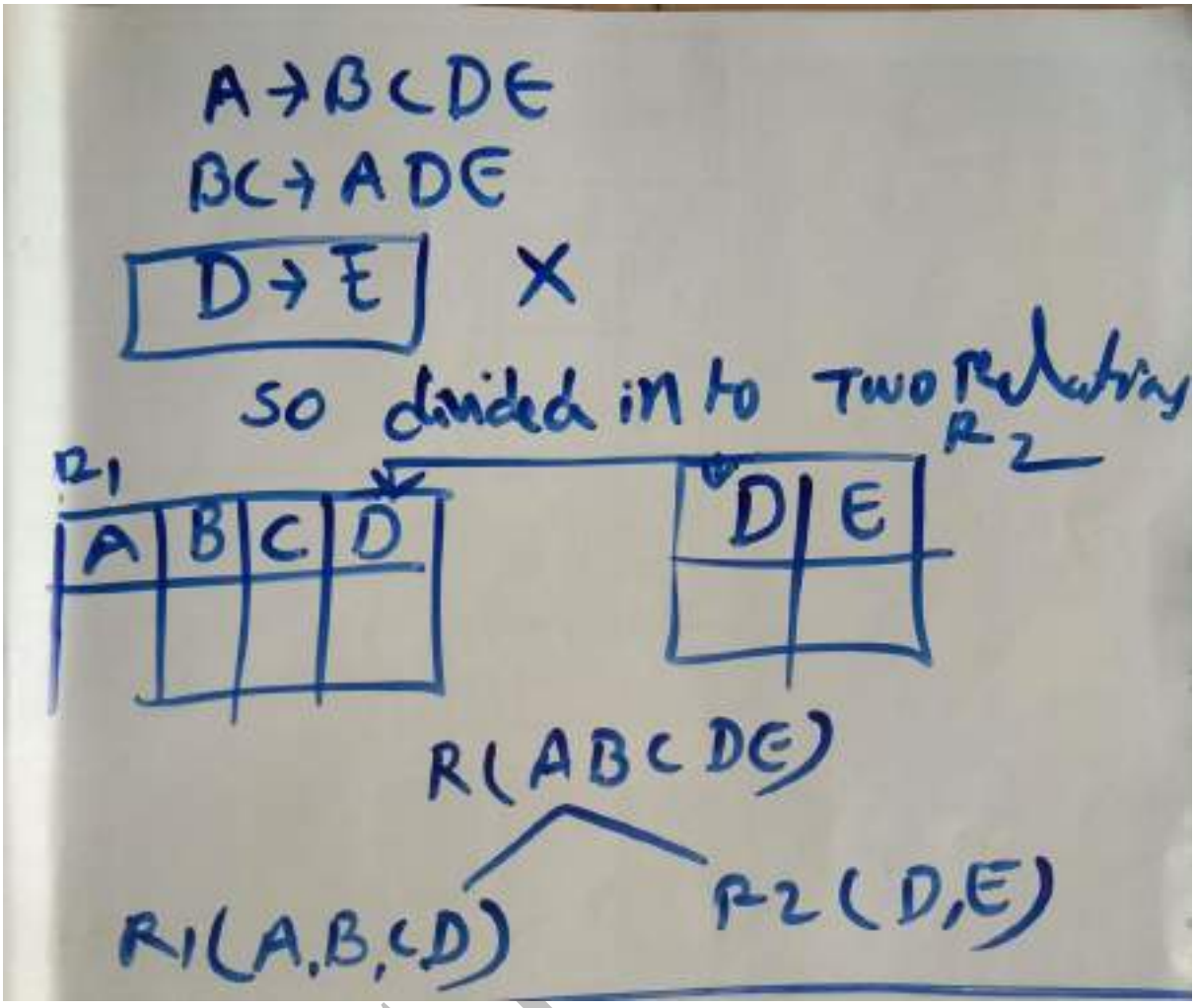
A → BCDE

BC → ADE

D → E

PK: A, BC

NPK: D, E



Boyce Codd normal form (BCNF)

- BCNF is the advance version of 3NF. It is stricter than 3NF.
- A table is in BCNF if every functional dependency X → Y, X is the super key of the table.
- For BCNF, the table should be in 3NF, and for every FD, LHS is super key.

Example: Let's assume there is a company where employees work in more than one department.

EMPLOYEE table:

EMP_ID	EMP_COUNTRY	EMP_DEPT	DEPT_TYPE	EMP_DEPT_NO
264	India	Designing	D394	283

264	India	Testing	D394	300
364	UK	Stores	D283	232
364	UK	Developing	D283	549

In the above table Functional dependencies are as follows:

1. EMP_ID → EMP_COUNTRY
2. EMP_DEPT → {DEPT_TYPE, EMP_DEPT_NO}

Candidate key: {EMP-ID, EMP-DEPT}

The table is not in BCNF because neither EMP_DEPT nor EMP_ID alone are keys.

To convert the given table into BCNF, we decompose it into three tables:

EMP_COUNTRY table:

EMP_ID	EMP_COUNTRY
264	India
264	India

EMP_DEPT table:

EMP_DEPT	DEPT_TYPE	EMP_DEPT_NO
Designing	D394	283
Testing	D394	300
Stores	D283	232
Developing	D283	549

EMP_DEPT_MAPPING table:

EMP_ID	EMP_DEPT
D394	283
D394	300

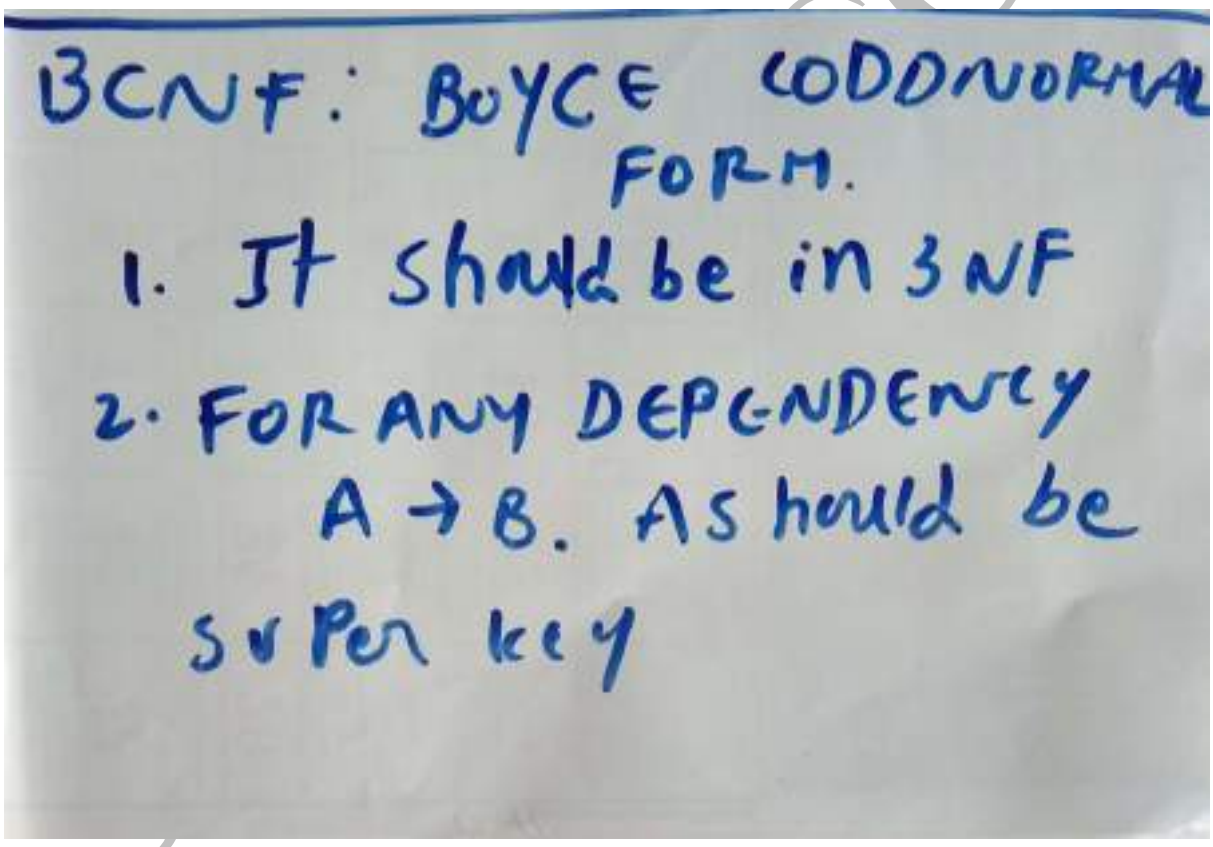
D283	232
D283	549

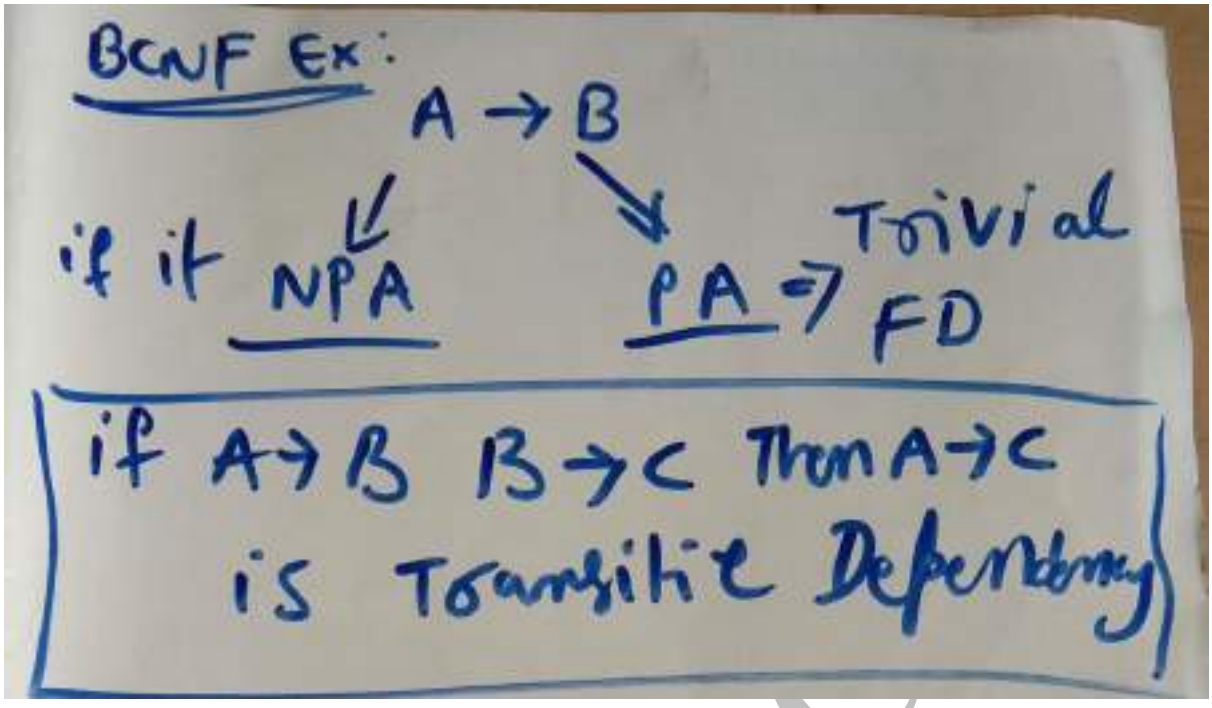
Functional dependencies:

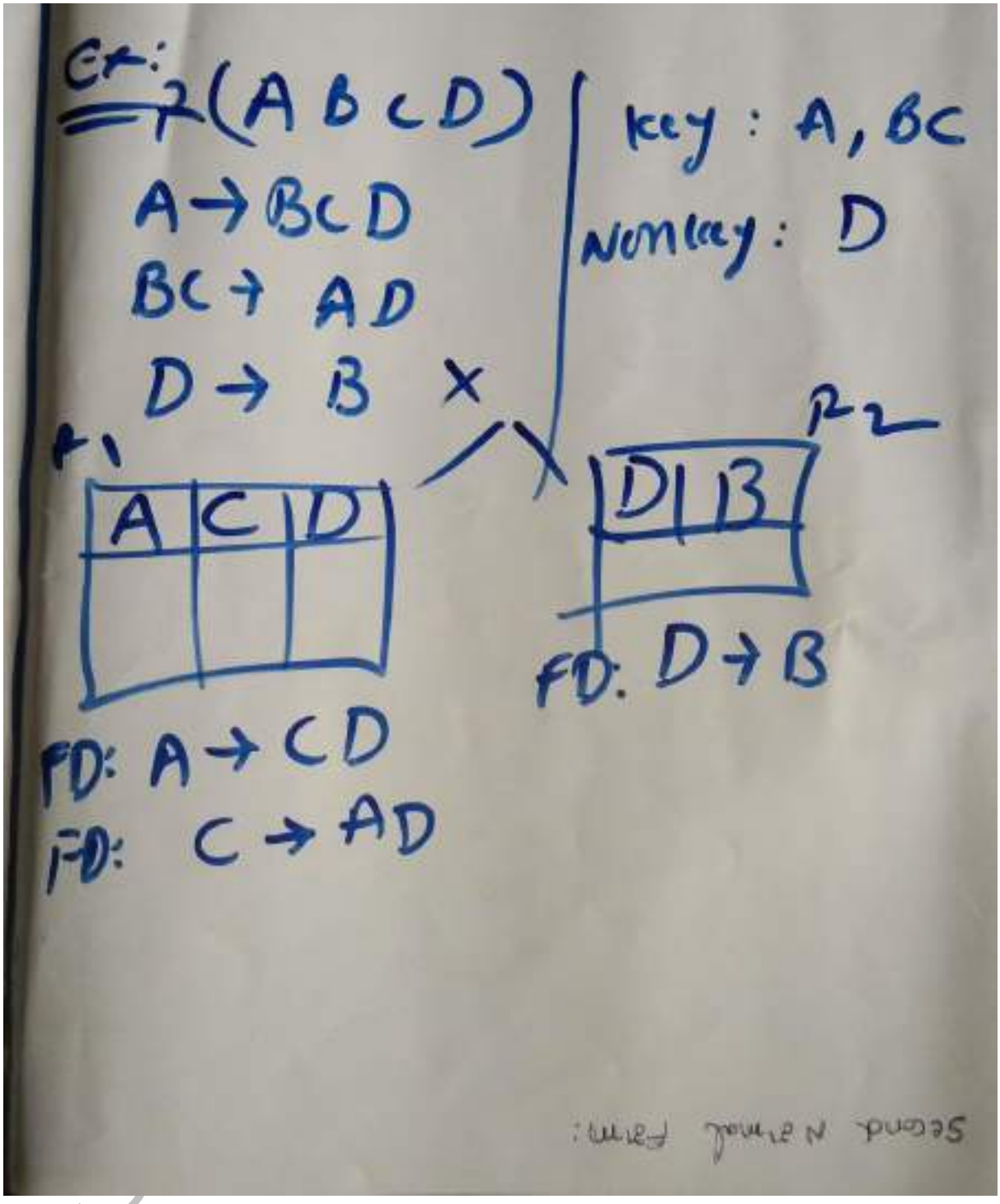
1. EMP_ID → EMP_COUNTRY
2. EMP_DEPT → {DEPT_TYPE, EMP_DEPT_NO}

Candidate keys:

For the first table: EMP_ID
For the second table: EMP_DEPT
For the third table: {EMP_ID, EMP_DEPT}




BCNF Ex:
A → B
if it NPA
PA ⇒ Trivial FD
if A→B B→C Then A→C
is Transitive Dependency



Fourth normal form (4NF)

- A relation will be in 4NF if it is in Boyce Codd normal form and has no multi-valued dependency.
- For a dependency A → B, if for a single value of A, multiple values of B exists, then the relation will be a multi-valued dependency.

Example

STUDENT

STU_ID	COURSE	HOBBY
21	Computer	Dancing
21	Math	Singing
34	Chemistry	Dancing
74	Biology	Cricket
59	Physics	Hockey

The given STUDENT table is in 3NF, but the COURSE and HOBBY are two independent entity. Hence, there is no relationship between COURSE and HOBBY.

In the STUDENT relation, a student with STU_ID, **21** contains two courses, **Computer** and **Math** and two hobbies, **Dancing** and **Singing**. So there is a Multi-valued dependency on STU_ID, which leads to unnecessary repetition of data.

So to make the above table into 4NF, we can decompose it into two tables:

STUDENT_COURSE

STU_ID	COURSE
21	Computer
21	Math
34	Chemistry
74	Biology
59	Physics

STUDENT_HOBBY

STU_ID	HOBBY
21	Dancing
21	Singing
34	Dancing

74	Cricket
59	Hockey

1NF : Remove multiple values, NULL values

2NF : PA → NPA

subset of PA→NPA X
Eliminates Partial dependency

3NF : NPA → NPA X

Eliminates Transitive dependency

BCNF : NPA → PA X

Eliminates Trivial FD

4NF : Eliminates MVD

5NF : Eliminates JoinDependency

4NF:

(1) satisfies the BCNF

(2) A Relation may not contain more than one multivalue attribute.

Ex:

Sid	Subject	Activity
10	CS	Running
10	CN	Jogging
10	CS	Jogging
10	CN	Running
11	CN	Running

FD: sid → → subject

MVD: sID → → Activity

Sid	Sub
10	CS
10	CN
11	CN

Sid	Activity
10	Running
10	Jogging
11	Running

Eliminates MVD

Multi value dependency.

Ex: Bikes

Model	Year manufacturing	color
2010	2008	SILVER
2010	2008	Black
2010	2009	SILVER
2010	2009	Black

MVD: model →→ Year manufact

MVD: model →→

Fifth normal form (5NF)

- A relation is in 5NF if it is in 4NF and not contains any join dependency and joining should be lossless.
- 5NF is satisfied when all the tables are broken into as many tables as possible in order to avoid redundancy.
- 5NF is also known as Project-join normal form (PJ/NF).

Example

SUBJECT	LECTURER	SEMESTER
Computer	Anshika	Semester 1
Computer	John	Semester 1
Math	John	Semester 1
Math	Akash	Semester 2
Chemistry	Praveen	Semester 1

In the above table, John takes both Computer and Math class for Semester 1 but he doesn't take Math class for Semester 2. In this case, combination of all these fields required to identify a valid data.

Suppose we add a new Semester as Semester 3 but do not know about the subject and who will be taking that subject so we leave Lecturer and Subject as NULL. But all three columns together acts as a primary key, so we can't leave other two columns blank.

So to make the above table into 5NF, we can decompose it into three relations P1, P2 & P3:

P1

SEMESTER	SUBJECT
Semester 1	Computer
Semester 1	Math
Semester 1	Chemistry
Semester 2	Math

P2

SUBJECT	LECTURER
Computer	Anshika
Computer	John
Math	John
Math	Akash
Chemistry	Praveen

P3

SEMSTER	LECTURER
Semester 1	Anshika
Semester 1	John
Semester 1	John
Semester 2	Akash
Semester 1	Praveen

UNIT V

Transaction Concept: Transaction State, Implementation of Atomicity and Durability, Concurrent Executions, Serializability, Recoverability, Implementation of Isolation, Testing for Serializability, Failure Classification, Storage, Recovery and Atomicity, Recovery algorithm. Indexing Techniques: B+ Trees: Search, Insert, Delete algorithms, File Organization and Indexing, Cluster Indexes, Primary and Secondary Indexes , Index data Structures, Hash Based Indexing: Tree base Indexing ,Comparison of File Organizations, Indexes and Performance Tuning

Transaction

- The transaction is a set of logically related operation. It contains a group of tasks.
- A transaction is an action or series of actions. It is performed by a single user to perform operations for accessing the contents of the database.

Example: Suppose an employee of bank transfers Rs 800 from X's account to Y's account. This small transaction contains several low-level tasks:

X's Account

1. Open_Account(X)
2. Old_Balance = X.balance
3. New_Balance = Old_Balance - 800
4. X.balance = New_Balance
5. Close_Account(X)

Y's Account

1. Open_Account(Y)
2. Old_Balance = Y.balance
3. New_Balance = Old_Balance + 800
4. Y.balance = New_Balance
5. Close_Account(Y)

Operations of Transaction:

Following are the main operations of transaction:

Read(X): Read operation is used to read the value of X from the database and stores it in a buffer in main memory.

Write(X): Write operation is used to write the value back to the database from the buffer.

Let's take an example to debit transaction from an account which consists of following operations:

1. 1. R(X);
2. 2. X = X - 500;
3. 3. W(X);

Let's assume the value of X before starting of the transaction is 4000.

- The first operation reads X's value from database and stores it in a buffer.
- The second operation will decrease the value of X by 500. So buffer will contain 3500.
- The third operation will write the buffer's value to the database. So X's final value will be 3500.

But it may be possible that because of the failure of hardware, software or power, etc. that transaction may fail before finished all the operations in the set.

For example: If in the above transaction, the debit transaction fails after executing operation 2 then X's value will remain 4000 in the database which is not acceptable by the bank.

To solve this problem, we have two important operations:

Commit: It is used to save the work done permanently.

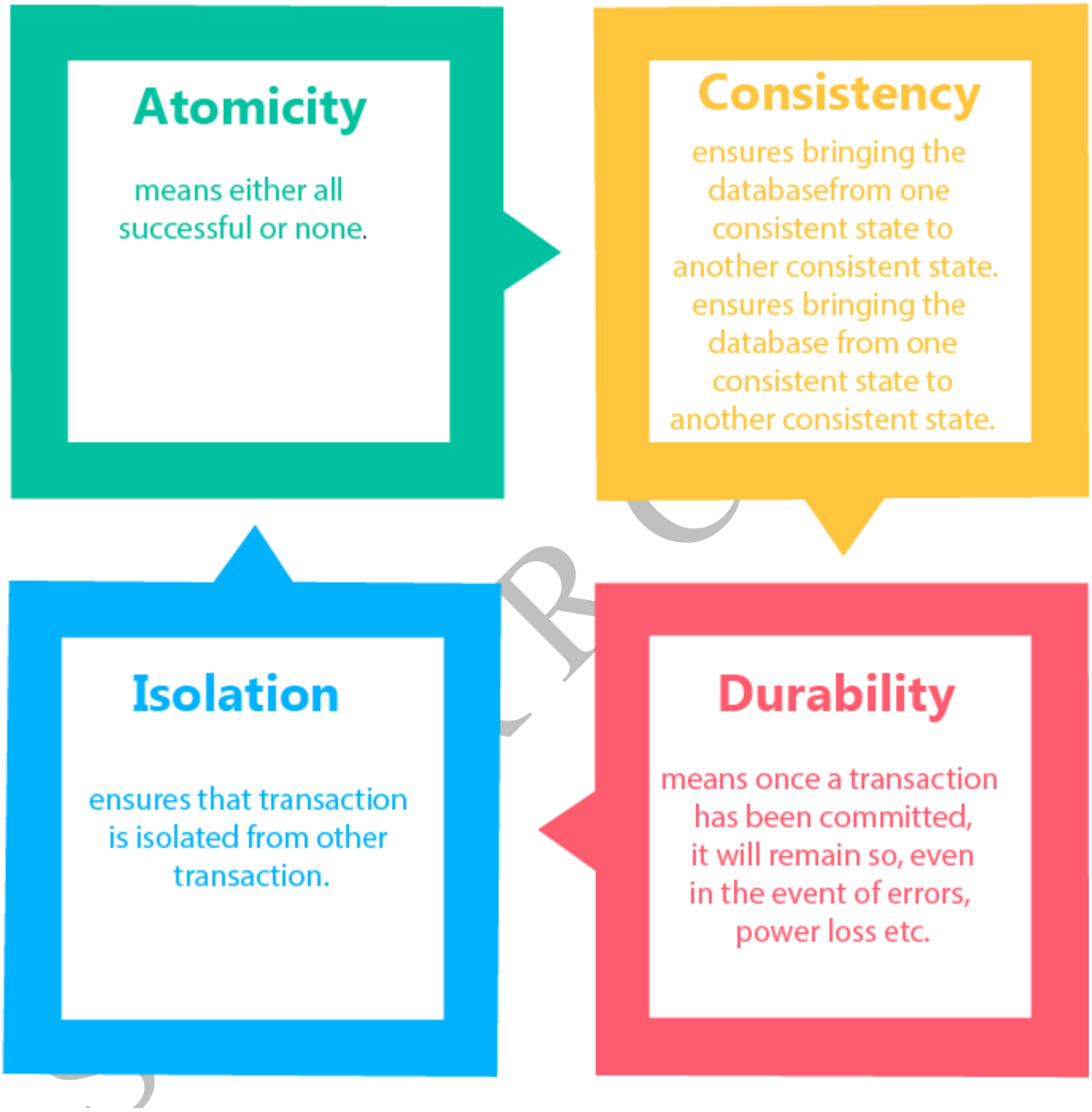
Rollback: It is used to undo the work done.

Transaction property

The transaction has the four properties. These are used to maintain consistency in a database, before and after the transaction.

Property of Transaction

1. Atomicity
2. Consistency
3. Isolation
4. Durability



Atomicity

- It states that all operations of the transaction take place at once if not, the transaction is aborted.
- There is no midway, i.e., the transaction cannot occur partially. Each transaction is treated as one unit and either run to completion or is not executed at all.

Atomicity involves the following two operations:

Abort: If a transaction aborts then all the changes made are not visible.

Commit: If a transaction commits then all the changes made are visible.

Example: Let's assume that following transaction T consisting of T1 and T2. A consists of Rs 600 and B consists of Rs 300. Transfer Rs 100 from account A to account B.

T1	T2
Read(A) A:= A-100 Write(A)	Read(B) Y:= Y+100 Write(B)

After completion of the transaction, A consists of Rs 500 and B consists of Rs 400.

If the transaction T fails after the completion of transaction T1 but before completion of transaction T2, then the amount will be deducted from A but not added to B. This shows the inconsistent database state. In order to ensure correctness of database state, the transaction must be executed in entirety.

Consistency

- The integrity constraints are maintained so that the database is consistent before and after the transaction.
- The execution of a transaction will leave a database in either its prior stable state or a new stable state.
- The consistent property of database states that every transaction sees a consistent database instance.
- The transaction is used to transform the database from one consistent state to another consistent state.

For example: The total amount must be maintained before or after the transaction.

1. Total before T occurs = 600+300=900
2. Total after T occurs= 500+400=900

Therefore, the database is consistent. In the case when T1 is completed but T2 fails, then inconsistency will occur.

Isolation

- It shows that the data which is used at the time of execution of a transaction cannot be used by the second transaction until the first one is completed.

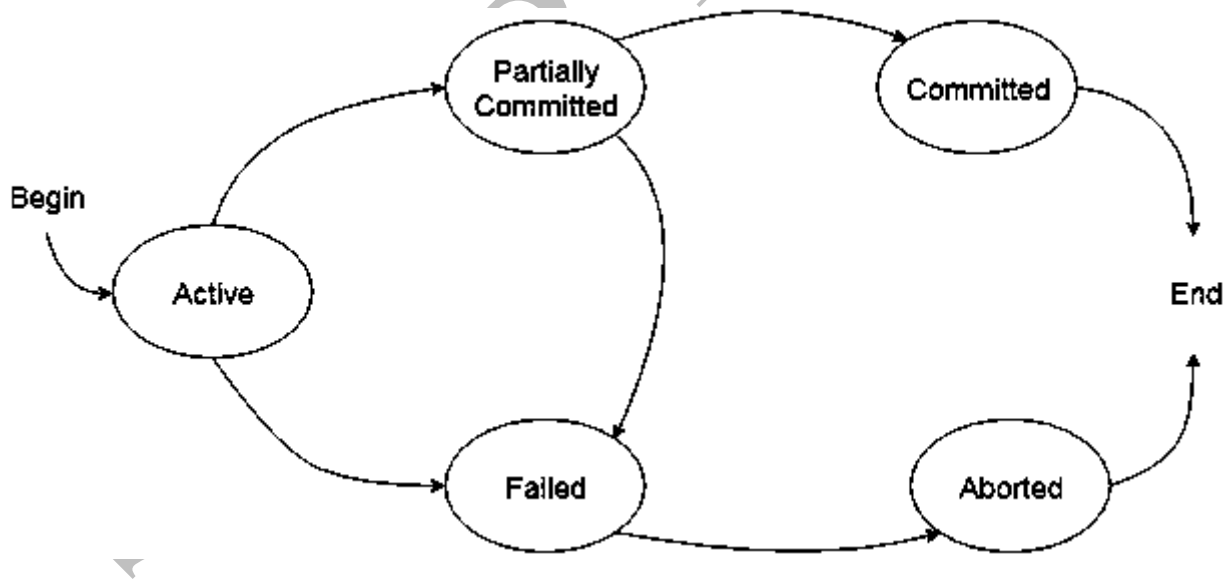
- In isolation, if the transaction T1 is being executed and using the data item X, then that data item can't be accessed by any other transaction T2 until the transaction T1 ends.
- The concurrency control subsystem of the DBMS enforced the isolation property.

Durability

- The durability property is used to indicate the performance of the database's consistent state. It states that the transaction made the permanent changes.
- They cannot be lost by the erroneous operation of a faulty transaction or by the system failure. When a transaction is completed, then the database reaches a state known as the consistent state. That consistent state cannot be lost, even in the event of a system's failure.
- The recovery subsystem of the DBMS has the responsibility of Durability property.

States of Transaction

In a database, the transaction can be in one of the following states -



Active state

- The active state is the first state of every transaction. In this state, the transaction is being executed.
- For example: Insertion or deletion or updating a record is done here. But all the records are still not saved to the database.

Partially committed

- In the partially committed state, a transaction executes its final operation, but the data is still not saved to the database.
- In the total mark calculation example, a final display of the total marks step is executed in this state.

Committed

A transaction is said to be in a committed state if it executes all its operations successfully. In this state, all the effects are now permanently saved on the database system.

Failed state

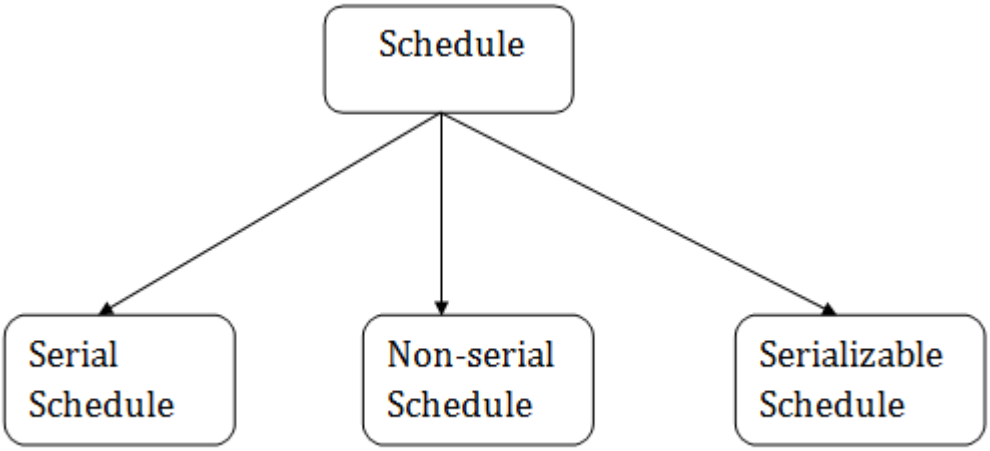
- If any of the checks made by the database recovery system fails, then the transaction is said to be in the failed state.
- In the example of total mark calculation, if the database is not able to fire a query to fetch the marks, then the transaction will fail to execute.

Aborted

- If any of the checks fail and the transaction has reached a failed state then the database recovery system will make sure that the database is in its previous consistent state. If not then it will abort or roll back the transaction to bring the database into a consistent state.
- If the transaction fails in the middle of the transaction then before executing the transaction, all the executed transactions are rolled back to its consistent state.
- After aborting the transaction, the database recovery module will select one of the two operations:
    1. Re-start the transaction
    2. Kill the transaction

Schedule

A series of operation from one transaction to another transaction is known as schedule. It is used to preserve the order of the operation in each of the individual transaction.



1. Serial Schedule

The serial schedule is a type of schedule where one transaction is executed completely before starting another transaction. In the serial schedule, when the first transaction completes its cycle, then the next transaction is executed.

For example: Suppose there are two transactions T1 and T2 which have some operations. If it has no interleaving of operations, then there are the following two possible outcomes:

1. Execute all the operations of T1 which was followed by all the operations of T2.
2. Execute all the operations of T1 which was followed by all the operations of T2.

- In the given (a) figure, Schedule A shows the serial schedule where T1 followed by T2.
- In the given (b) figure, Schedule B shows the serial schedule where T2 followed by T1.

2. Non-serial Schedule

- If interleaving of operations is allowed, then there will be non-serial schedule.
- It contains many possible orders in which the system can execute the individual operations of the transactions.
- In the given figure (c) and (d), Schedule C and Schedule D are the non-serial schedules. It has interleaving of operations.

3. Serializable schedule

- The serializability of schedules is used to find non-serial schedules that allow the transaction to execute concurrently without interfering with one another.
- It identifies which schedules are correct when executions of the transaction have interleaving of their operations.
- A non-serial schedule will be serializable if its result is equal to the result of its transactions executed serially.

(a)

Time

T1	T2

read(A);
A := A – N;
write(A);
read(B);
B := B + N;
write(B);	

	read(A);
	A := A + M;
	write(A);

Schedule A

(b)

Time

T1	T2

	read(A);
	A := A + M;
	write(A);

read(A);
A:= A - N;
write(A);
read(B);
B:= B +N;
write(B);	

Schedule B

(c)

Time

T1	T2
read(A); A := A – N;	
	read(A); A := A + M;
write(A); read(B);	
	write(A);
B := B + N; write(B);	

Schedule C

(d)

Time

T1	T2

read(A);
A := A – N;
write(A);	

	read(A);
	A := A + M;
	write(A);

read(B);
B := B + N;
write(B);	

Schedule D

Here,

Schedule A and Schedule B are serial schedule.

Schedule C and Schedule D are Non-serial schedule.

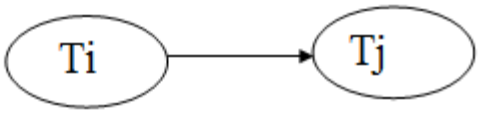
Testing of Serializability

Serialization Graph is used to test the Serializability of a schedule.

Assume a schedule S. For S, we construct a graph known as precedence graph. This graph has a pair G = (V, E), where V consists a set of vertices, and E consists a set of edges. The set of vertices is used to contain all the transactions participating in the schedule. The set of edges is used to contain all edges Ti ->Tj for which one of the three conditions holds:

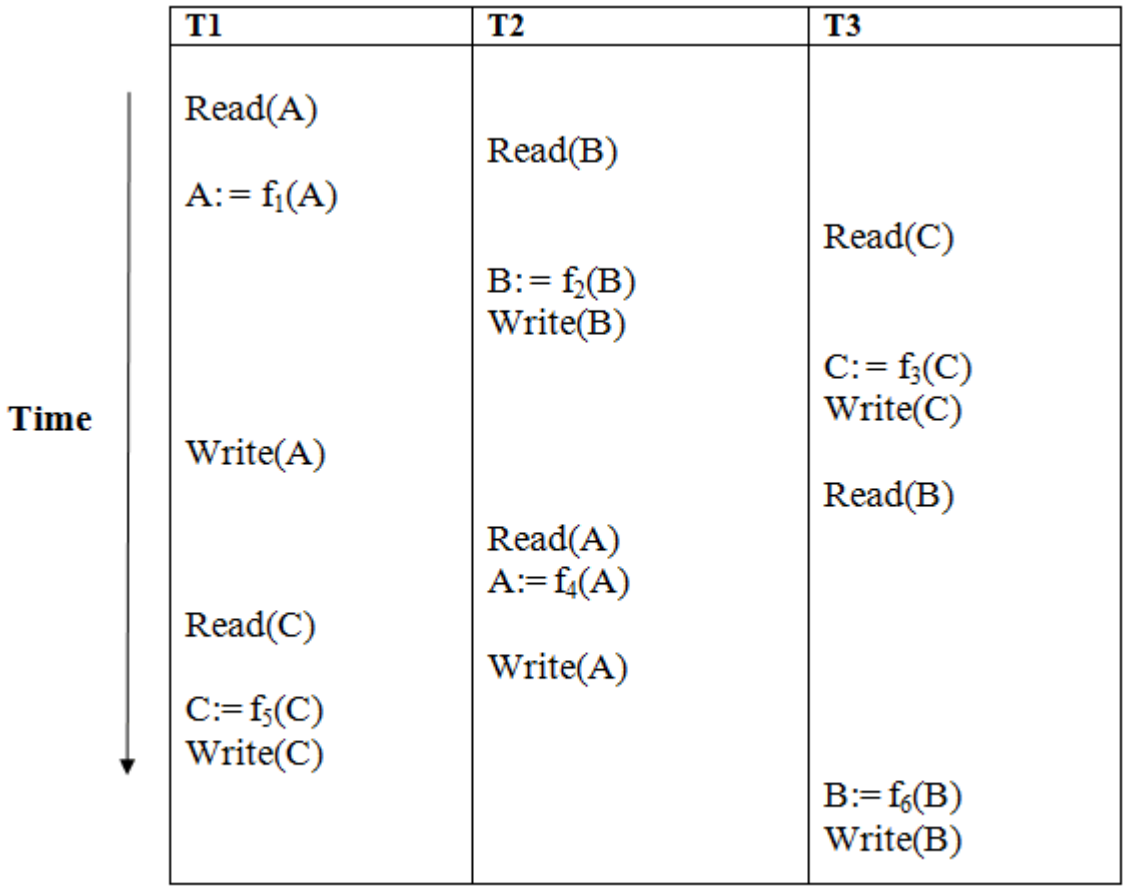
1. Create a node Ti → Tj if Ti executes write (Q) before Tj executes read (Q).
2. Create a node Ti → Tj if Ti executes read (Q) before Tj executes write (Q).
3. Create a node Ti → Tj if Ti executes write (Q) before Tj executes write (Q).

Precedence graph for Schedule S



- If a precedence graph contains a single edge Ti → Tj, then all the instructions of Ti are executed before the first instruction of Tj is executed.
- If a precedence graph for schedule S contains a cycle, then S is non-serializable. If the precedence graph has no cycle, then S is known as serializable.

For example:

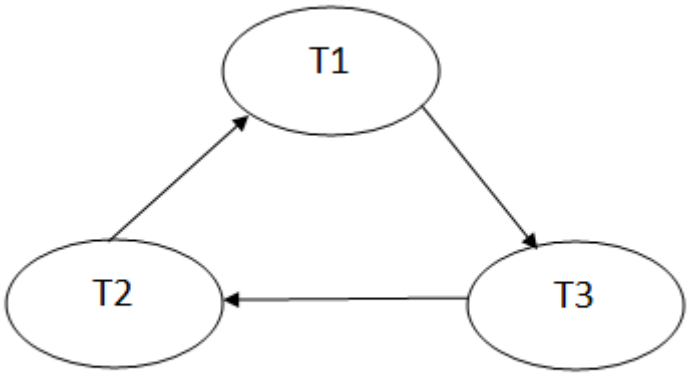


Schedule S1

Explanation:

Read(A): In T1, no subsequent writes to A, so no new edges
Read(B): In T2, no subsequent writes to B, so no new edges
Read(C): In T3, no subsequent writes to C, so no new edges
Write(B): B is subsequently read by T3, so add edge T2 → T3
Write(C): C is subsequently read by T1, so add edge T3 → T1
Write(A): A is subsequently read by T2, so add edge T1 → T2
Write(A): In T2, no subsequent reads to A, so no new edges
Write(C): In T1, no subsequent reads to C, so no new edges
Write(B): In T3, no subsequent reads to B, so no new edges

Precedence graph for schedule S1:



The precedence graph for schedule S1 contains a cycle that's why Schedule S1 is non-serializable.

T4	T5	T6
Read(A)		
A:= f1(A)		
Read(C)		
Write(A)		
A:= f2(C)		
	Read(B)	
Write(C)		
	Read(A)	
		Read(C)
	B:= f3(B)	
	Write(B)	
		C:= f4(C)
		Read(B)
		Write(C)
	A:=f5(A)	
	Write(A)	
		B:= f6(B)
		Write(B)

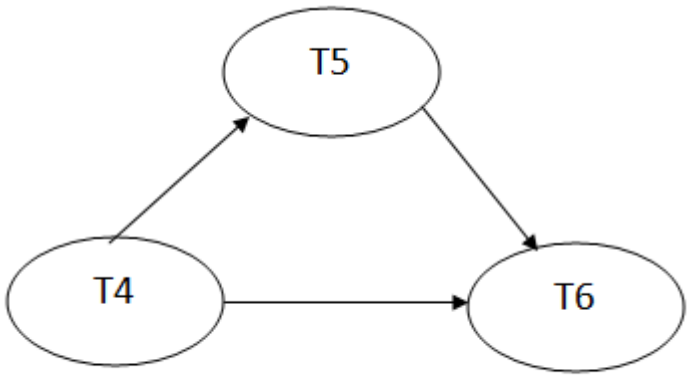
Time

Schedule S2

Explanation:

Read(A): In T4,no subsequent writes to A, so no new edges
Read(C): In T4, no subsequent writes to C, so no new edges
Write(A): A is subsequently read by T5, so add edge T4 → T5
Read(B): In T5,no subsequent writes to B, so no new edges
Write(C): C is subsequently read by T6, so add edge T4 → T6
Write(B): A is subsequently read by T6, so add edge T5 → T6
Write(C): In T6, no subsequent reads to C, so no new edges
Write(A): In T5, no subsequent reads to A, so no new edges
Write(B): In T6, no subsequent reads to B, so no new edges

Precedence graph for schedule S2:



The precedence graph for schedule S2 contains no cycle that's why ScheduleS2 is serializable.

Conflict Serializable Schedule

- A schedule is called conflict serializability if after swapping of non-conflicting operations, it can transform into a serial schedule.
- The schedule will be a conflict serializable if it is conflict equivalent to a serial schedule.

Conflicting Operations

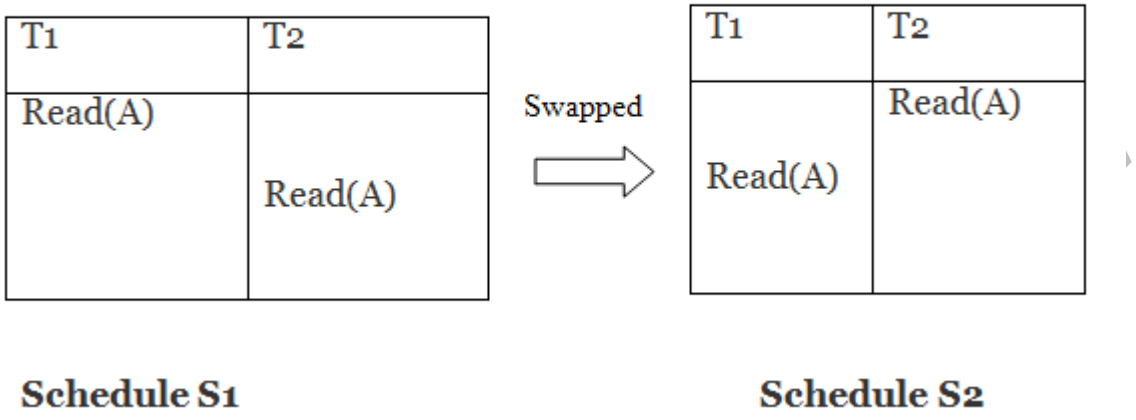
The two operations become conflicting if all conditions satisfy:

1. Both belong to separate transactions.
2. They have the same data item.
3. They contain at least one write operation.

Example:

Swapping is possible only if S1 and S2 are logically equal.

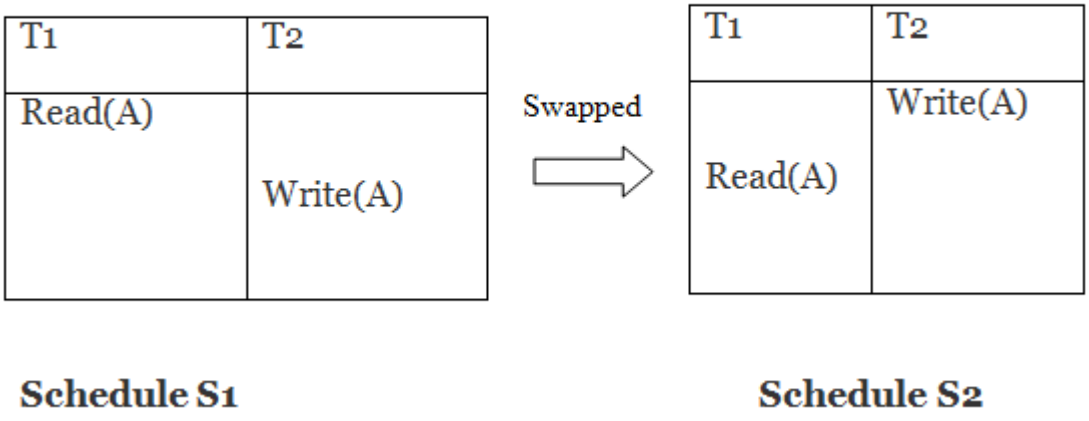
1. T1: Read(A) T2: Read(A)



Schedule S1 **Schedule S2**

Here, S1 = S2. That means it is non-conflict.

2. T1: Read(A) T2: Write(A)



Schedule S1 **Schedule S2**

Here, S1 ≠ S2. That means it is conflict.

Conflict Equivalent

In the conflict equivalent, one can be transformed to another by swapping non-conflicting operations. In the given example, S2 is conflict equivalent to S1 (S1 can be converted to S2 by swapping non-conflicting operations).

Two schedules are said to be conflict equivalent if and only if:

1. They contain the same set of the transaction.
2. If each pair of conflict operations are ordered in the same way.

Example:

Non-serial schedule

T1	T2
Read(A)	
Write(A)	
	Read(A)
	Write(A)
Read(B)	
Write(B)	
	Read(B)
	Write(B)

Schedule S1

Serial Schedule

T1	T2
Read(A)	
Write(A)	
Read(B)	
Write(B)	
	Read(A)
	Write(A)
	Read(B)
	Write(B)

Schedule S2

Schedule S2 is a serial schedule because, in this, all operations of T1 are performed before starting any operation of T2. Schedule S1 can be transformed into a serial schedule by swapping non-conflicting operations of S1.

After swapping of non-conflict operations, the schedule S1 becomes:

T1	**T2**
Read(A) Write(A) Read(B) Write(B)	

	Read(A) Write(A) Read(B) Write(B)

Since, S1 is conflict serializable.

View Serializability

- A schedule will view serializable if it is view equivalent to a serial schedule.
- If a schedule is conflict serializable, then it will be view serializable.
- The view serializable which does not conflict serializable contains blind writes.

View Equivalent

Two schedules S1 and S2 are said to be view equivalent if they satisfy the following conditions:

1. Initial Read

An initial read of both schedules must be the same. Suppose two schedule S1 and S2. In schedule S1, if a transaction T1 is reading the data item A, then in S2, transaction T1 should also read A.

T1	T2
Read(A)	
	Write(A)

Schedule S1

T1	T2
	Write(A)
Read(A)	

Schedule S2

Above two schedules are view equivalent because Initial read operation in S1 is done by T1 and in S2 it is also done by T1.

2. Updated Read

In schedule S1, if Ti is reading A which is updated by Tj then in S2 also, Ti should read A which is updated by Tj.

T1	T2	T3
Write(A)		
	Write(A)	
		Read(A)

Schedule S1

T1	T2	T3
	Write(A)	
Write(A)		
		Read(A)

Schedule S2

Above two schedules are not view equal because, in S1, T3 is reading A updated by T2 and in S2, T3 is reading A updated by T1.

3. Final Write

A final write must be the same between both the schedules. In schedule S1, if a transaction T1 updates A at last then in S2, final writes operations should also be done by T1.

T1	T2	T3
Write(A)		
	Read(A)	
		Write(A)

Schedule S1

T1	T2	T3
	Read(A)	
Write(A)		
		Write(A)

Schedule S2

Above two schedules is view equal because Final write operation in S1 is done by T3 and in S2, the final write operation is also done by T3.

Example:

T1	T2	T3
Read(A) Write(A)	Write(A)	Write(A)

Schedule S

With 3 transactions, the total number of possible schedule

1. = 3! = 6
2. S1 = <T1 T2 T3>
3. S2 = <T1 T3 T2>
4. S3 = <T2 T3 T1>
5. S4 = <T2 T1 T3>
6. S5 = <T3 T1 T2>
7. S6 = <T3 T2 T1>

Taking first schedule S1:

T1	T2	T3
Read(A) Write(A)	Write(A)	Write(A)

Schedule S1

Step 1: final updation on data items

In both schedules S and S1, there is no read except the initial read that's why we don't need to check that condition.

Step 2: Initial Read

The initial read operation in S is done by T1 and in S1, it is also done by T1.

Step 3: Final Write

The final write operation in S is done by T3 and in S1, it is also done by T3. So, S and S1 are view Equivalent.

The first schedule S1 satisfies all three conditions, so we don't need to check another schedule.

Hence, view equivalent serial schedule is:

1. T1 → T2 → T3

Recoverability of Schedule

Sometimes a transaction may not execute completely due to a software issue, system crash or hardware failure. In that case, the failed transaction has to be rollback. But some other transaction may also have used value produced by the failed transaction. So we also have to rollback those

tr

T1	T1's buffer space	T2	T2's buffer space	Database
				A = 6500
Read(A);	A = 6500			A = 6500
A = A - 500;	A = 6000			A = 6500
Write(A);	A = 6000			A = 6000
		Read(A);	A = 6000	A = 6000
		A =A + 1000;	A = 7000	A = 6000
		Write(A);	A = 7000	A = 7000
		Commit;		
Failure Point				
Commit;				

ansactions.

The above table 1 shows a schedule which has two transactions. T1 reads and writes the value of A and that value is read and written by T2. T2 commits but later on, T1 fails. Due to the failure, we have to rollback T1. T2 should also be rollback because it reads the value written by T1, but T2 can't be rollback because it already committed. So this type of schedule is known as irrecoverable schedule.

Irrecoverable schedule: The schedule will be irrecoverable if Tj reads the updated value of Ti and Tj committed before Ti commit.

T1	T1's buffer space	T2	T2's buffer space	Database
				A = 6500
Read(A);	A = 6500			A = 6500
A = A - 500;	A = 6000			A = 6500
Write(A);	A = 6000			A = 6000
		Read(A);	A = 6000	A = 6000
		A =A + 1000;	A = 7000	A = 6000
		Write(A);	A = 7000	A = 7000
Failure Point				
Commit;				
		Commit;		

The above table 2 shows a schedule with two transactions. Transaction T1 reads and writes A, and that value is read and written by transaction T2. But later on, T1 fails. Due to this, we have to rollback T1. T2 should be rollback because T2 has read the value written by T1. As it has not committed before T1 commits so we can rollback transaction T2 as well. So it is recoverable with cascade rollback.

Recoverable with cascading rollback: The schedule will be recoverable with cascading rollback if Tj reads the updated value of Ti. Commit of Tj is delayed till commit of Ti.

T1	T1's buffer space	T2	T2's buffer space	Database
				A = 6500
Read(A);	A = 6500			A = 6500
A = A - 500;	A = 6000			A = 6500
Write(A);	A = 6000			A = 6000
Commit;		Read(A);	A = 6000	A = 6000
		A =A + 1000;	A = 7000	A = 6000
		Write(A);	A = 7000	A = 7000
		Commit;		

The above Table 3 shows a schedule with two transactions. Transaction T1 reads and write A and commits, and that value is read and written by T2. So this is a cascade less recoverable schedule.

Failure Classification

To find that where the problem has occurred, we generalize a failure into the following categories:

1. Transaction failure
2. System crash
3. Disk failure

1. Transaction failure

The transaction failure occurs when it fails to execute or when it reaches a point from where it can't go any further. If a few transaction or process is hurt, then this is called as transaction failure.

Reasons for a transaction failure could be -

1. **Logical errors:** If a transaction cannot complete due to some code error or an internal error condition, then the logical error occurs.
2. **Syntax error:** It occurs where the DBMS itself terminates an active transaction because the database system is not able to execute it. **For example,** The system aborts an active transaction, in case of deadlock or resource unavailability.

2. System Crash

- System failure can occur due to power failure or other hardware or software failure. **Example:** Operating system error.

  Fail-stop assumption: In the system crash, non-volatile storage is assumed not to be corrupted.

3. Disk Failure

- It occurs where hard-disk drives or storage drives used to fail frequently. It was a common problem in the early days of technology evolution.
- Disk failure occurs due to the formation of bad sectors, disk head crash, and unreachability to the disk or any other failure, which destroy all or part of disk storage.

Log-Based Recovery

- The log is a sequence of records. Log of each transaction is maintained in some stable storage so that if any failure occurs, then it can be recovered from there.
- If any operation is performed on the database, then it will be recorded in the log.
- But the process of storing the logs should be done before the actual transaction is applied in the database.
-

Let's assume there is a transaction to modify the City of a student. The following logs are written for this transaction.

- When the transaction is initiated, then it writes 'start' log.

1. <Tn, Start>

- When the transaction modifies the City from 'Noida' to 'Bangalore', then another log is written to the file.

1. <Tn, City, 'Noida', 'Bangalore' >

- When the transaction is finished, then it writes another log to indicate the end of the transaction.

1. <Tn, Commit>

There are two approaches to modify the database:

1. Deferred database modification:

- The deferred modification technique occurs if the transaction does not modify the database until it has committed.
- In this method, all the logs are created and stored in the stable storage, and the database is updated when a transaction commits.

2. Immediate database modification:

- The Immediate modification technique occurs if database modification occurs while the transaction is still active.
- In this technique, the database is modified immediately after every operation. It follows an actual database modification.

Recovery using Log records

When the system is crashed, then the system consults the log to find which transactions need to be undone and which need to be redone.

1. If the log contains the record <Ti, Start> and <Ti, Commit> or <Ti, Commit>, then the Transaction Ti needs to be redone.
2. If log contains record<T_n, Start> but does not contain the record either <Ti, commit> or <Ti, abort>, then the Transaction Ti needs to be undone.

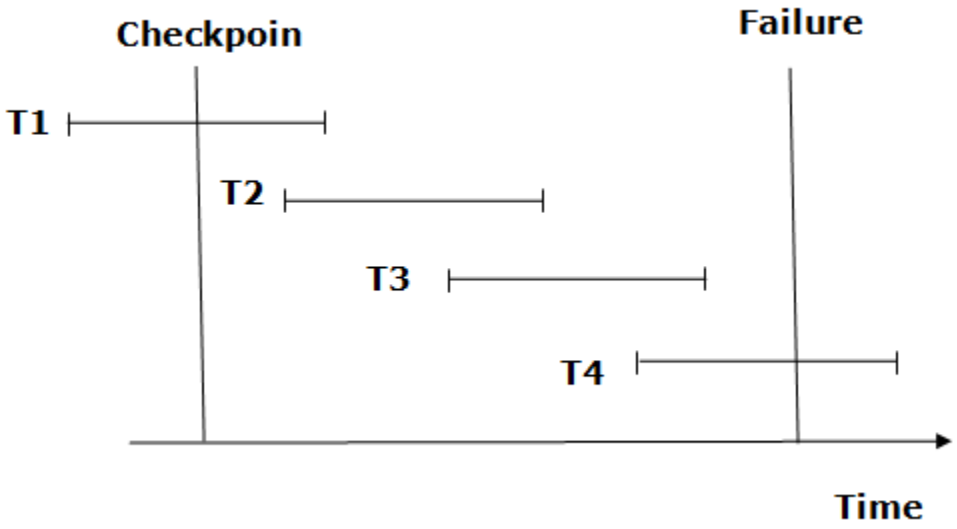
Checkpoint

- The checkpoint is a type of mechanism where all the previous logs are removed from the system and permanently stored in the storage disk.

- The checkpoint is like a bookmark. While the execution of the transaction, such checkpoints are marked, and the transaction is executed then using the steps of the transaction, the log files will be created.
- When it reaches to the checkpoint, then the transaction will be updated into the database, and till that point, the entire log file will be removed from the file. Then the log file is updated with the new step of transaction till next checkpoint and so on.
- The checkpoint is used to declare a point before which the DBMS was in the consistent state, and all transactions were committed.

Recovery using Checkpoint

In the following manner, a recovery system recovers the database from this failure:



- The recovery system reads log files from the end to start. It reads log files from T4 to T1.
- Recovery system maintains two lists, a redo-list, and an undo-list.
- The transaction is put into redo state if the recovery system sees a log with <Tn, Start> and <Tn, Commit> or just <Tn, Commit>. In the redo-list and their previous list, all the transactions are removed and then redone before saving their logs.
- **For example:** In the log file, transaction T2 and T3 will have <Tn, Start> and <Tn, Commit>. The T1 transaction will have only <Tn, commit> in the log file. That's why the transaction is committed after the checkpoint is crossed. Hence it puts T1, T2 and T3 transaction into redo list.

- The transaction is put into undo state if the recovery system sees a log with <Tn, Start> but no commit or abort log found. In the undo-list, all the transactions are undone, and their logs are removed.
- **For example:** Transaction T4 will have <Tn, Start>. So T4 will be put into undo list since this transaction is not yet complete and failed amid.

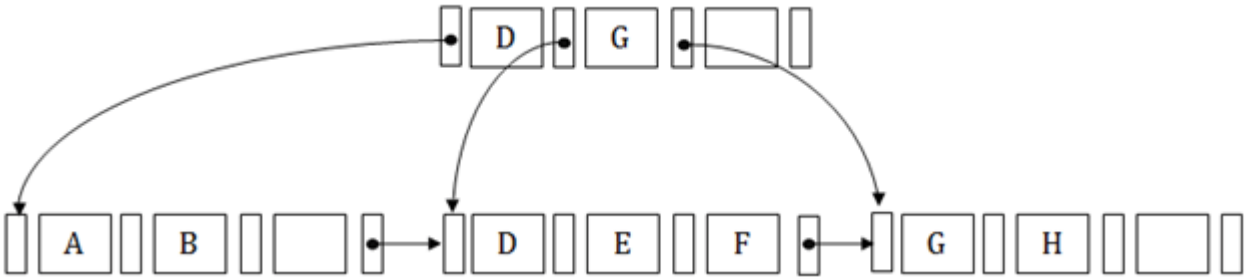
INDEXING TECHNIQUES:

B+ Tree

- The B+ tree is a balanced binary search tree. It follows a multi-level index format.
- In the B+ tree, leaf nodes denote actual data pointers. B+ tree ensures that all leaf nodes remain at the same height.
- In the B+ tree, the leaf nodes are linked using a link list. Therefore, a B+ tree can support random access as well as sequential access.

Structure of B+ Tree

- In the B+ tree, every leaf node is at equal distance from the root node. The B+ tree is of the order n where n is fixed for every B+ tree.
- It contains an internal node and leaf node.



Internal node

- An internal node of the B+ tree can contain at least n/2 record pointers except the root node.
- At most, an internal node of the tree contains n pointers.

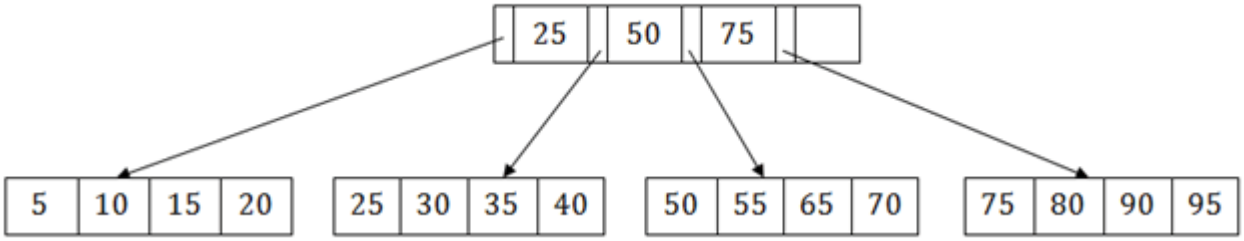
Leaf node

- The leaf node of the B+ tree can contain at least n/2 record pointers and n/2 key values.
- At most, a leaf node contains n record pointer and n key values.
- Every leaf node of the B+ tree contains one block pointer P to point to next leaf node.

Searching a record in B+ Tree

Suppose we have to search 55 in the below B+ tree structure. First, we will fetch for the intermediary node which will direct to the leaf node that can contain a record for 55.

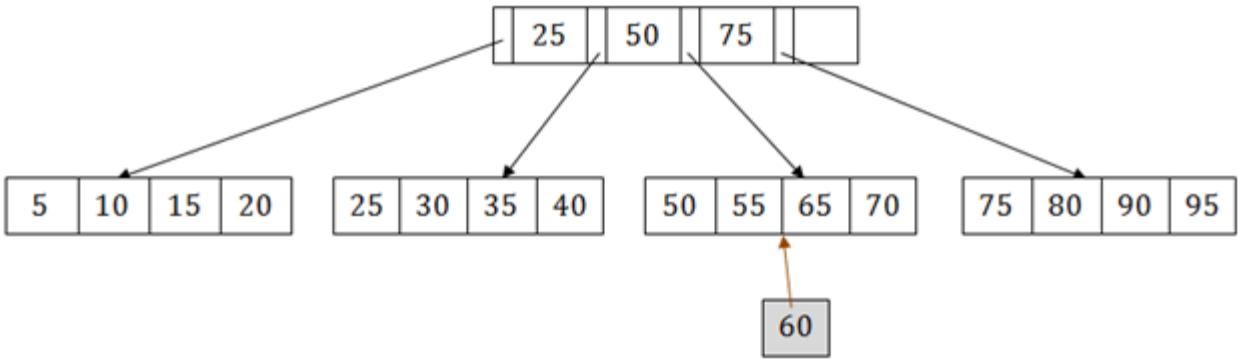
So, in the intermediary node, we will find a branch between 50 and 75 nodes. Then at the end, we will be redirected to the third leaf node. Here DBMS will perform a sequential search to find 55.



B+ Tree Insertion

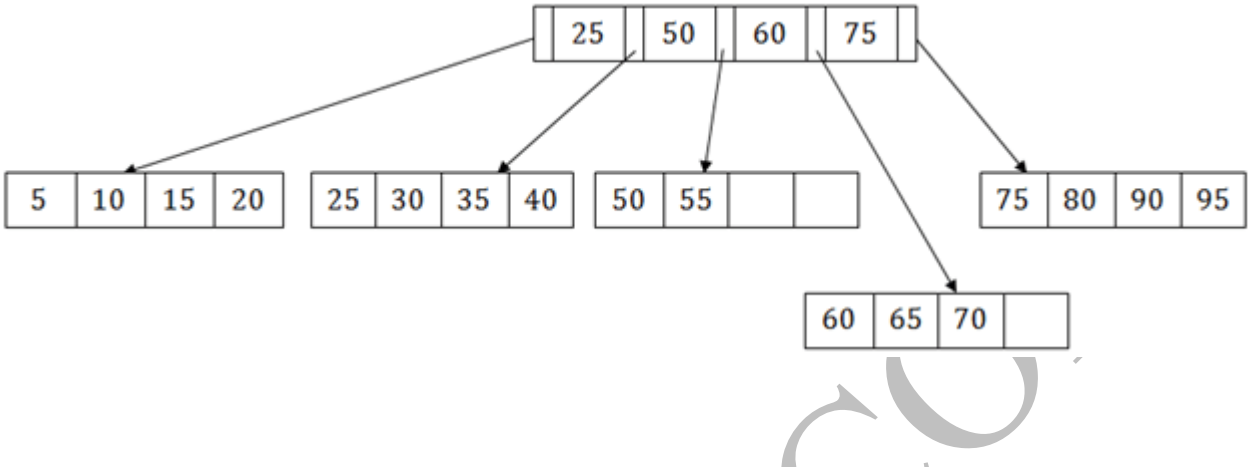
Suppose we want to insert a record 60 in the below structure. It will go to the 3rd leaf node after 55. It is a balanced tree, and a leaf node of this tree is already full, so we cannot insert 60 there.

In this case, we have to split the leaf node, so that it can be inserted into tree without affecting the fill factor, balance and order.



The 3rd leaf node has the values (50, 55, 60, 65, 70) and its current root node is 50. We will split the leaf node of the tree in the middle so that its balance is not altered. So we can group (50, 55) and (60, 65, 70) into 2 leaf nodes.

If these two has to be leaf nodes, the intermediate node cannot branch from 50. It should have 60 added to it, and then we can have pointers to a new leaf node.

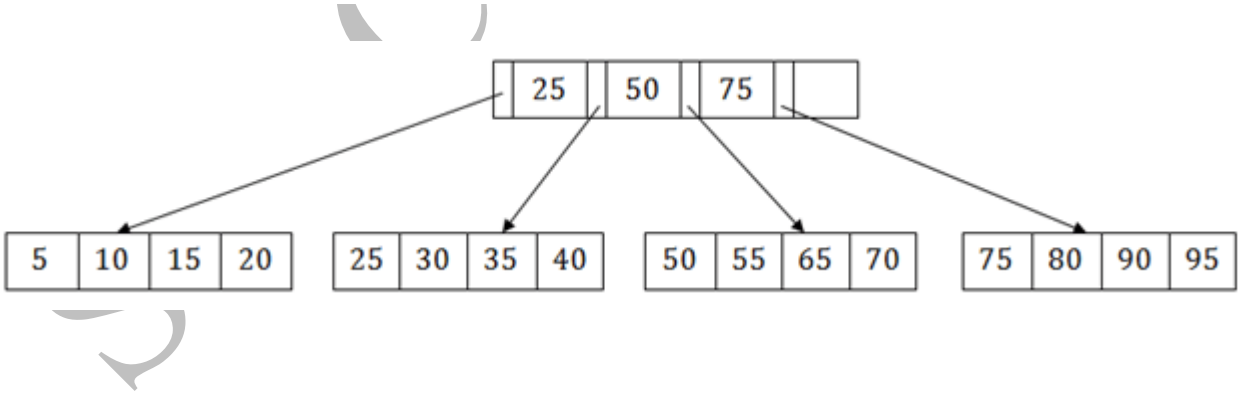


This is how we can insert an entry when there is overflow. In a normal scenario, it is very easy to find the node where it fits and then place it in that leaf node.

B+ Tree Deletion

Suppose we want to delete 60 from the above example. In this case, we have to remove 60 from the intermediate node as well as from the 4th leaf node too. If we remove it from the intermediate node, then the tree will not satisfy the rule of the B+ tree. So we need to modify it to have a balanced tree.

After deleting node 60 from above B+ tree and re-arranging the nodes, it will show as follows:



File Organization

- The **File** is a collection of records. Using the primary key, we can access the records. The type and frequency of access can be determined by the type of file organization which was used for a given set of records.
- File organization is a logical relationship among various records. This method defines how file records are mapped onto disk blocks.

- File organization is used to describe the way in which the records are stored in terms of blocks, and the blocks are placed on the storage medium.
- The first approach to map the database to the file is to use the several files and store only one fixed length record in any given file. An alternative approach is to structure our files so that we can contain multiple lengths for records.
- Files of fixed length records are easier to implement than the files of variable length records.

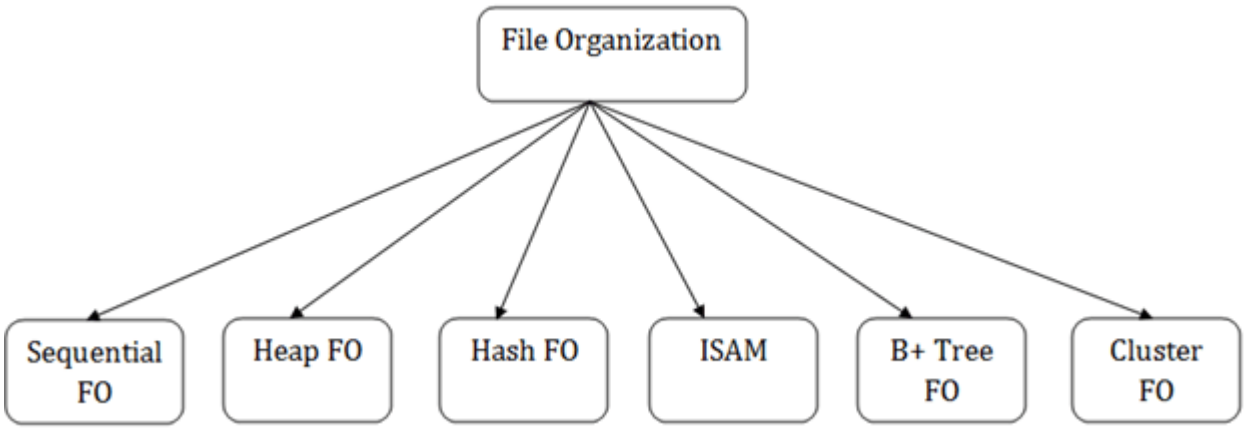
Objective of file organization

- It contains an optimal selection of records, i.e., records can be selected as fast as possible.
- To perform insert, delete or update transaction on the records should be quick and easy.
- The duplicate records cannot be induced as a result of insert, update or delete.
- For the minimal cost of storage, records should be stored efficiently.

Types of file organization:

File organization contains various methods. These particular methods have pros and cons on the basis of access or selection. In the file organization, the programmer decides the best-suited file organization method according to his requirement.

Types of file organization are as follows:



- Sequential file organization
- Heap file organization
- Hash file organization

- B+ file organization
- Indexed sequential access method (ISAM)
- Cluster file organization

Sequential File Organization

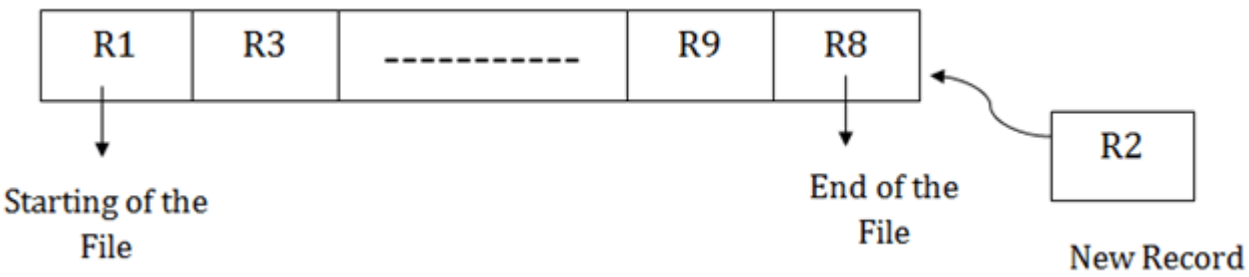
This method is the easiest method for file organization. In this method, files are stored sequentially. This method can be implemented in two ways:

1. Pile File Method:

- It is a quite simple method. In this method, we store the record in a sequence, i.e., one after another. Here, the record will be inserted in the order in which they are inserted into tables.
- In case of updating or deleting of any record, the record will be searched in the memory blocks. When it is found, then it will be marked for deleting, and the new record is inserted.

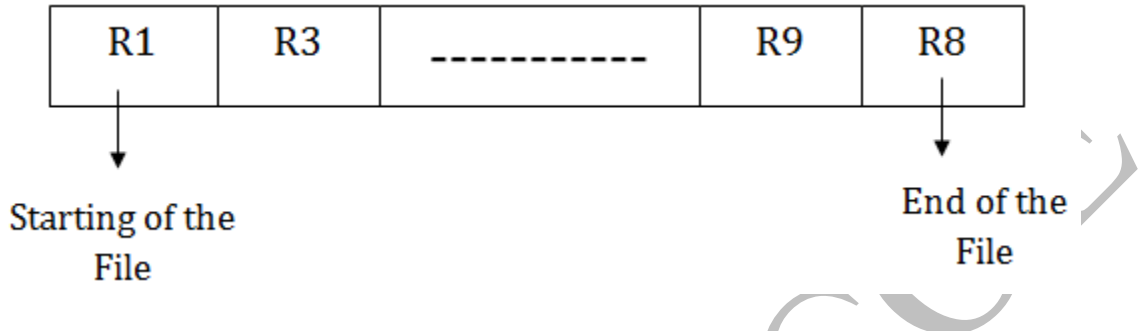
Insertion of the new record:

Suppose we have four records R1, R3 and so on upto R9 and R8 in a sequence. Hence, records are nothing but a row in the table. Suppose we want to insert a new record R2 in the sequence, then it will be placed at the end of the file. Here, records are nothing but a row in any table.



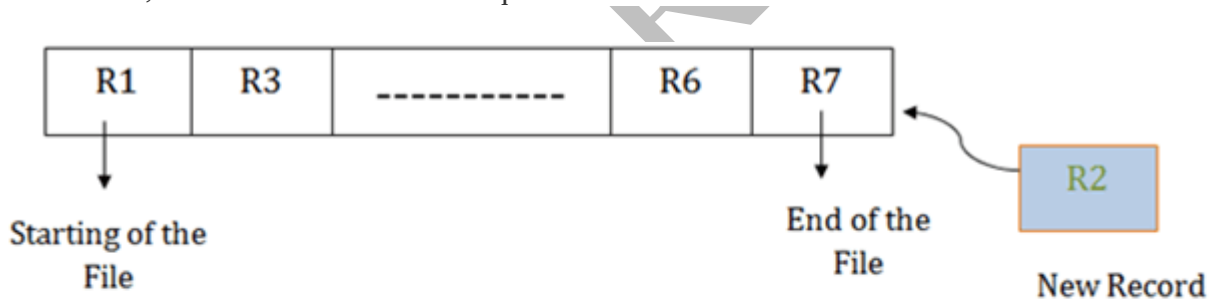
2. Sorted File Method:

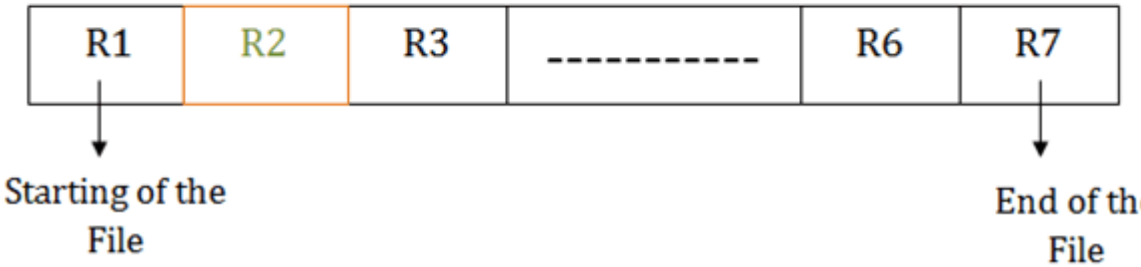
- In this method, the new record is always inserted at the file's end, and then it will sort the sequence in ascending or descending order. Sorting of records is based on any primary key or any other key.
- In the case of modification of any record, it will update the record and then sort the file, and lastly, the updated record is placed in the right place.



Insertion of the new record:

Suppose there is a preexisting sorted sequence of four records R1, R3 and so on upto R6 and R7. Suppose a new record R2 has to be inserted in the sequence, then it will be inserted at the end of the file, and then it will sort the sequence.





Pros of sequential file organization

- It contains a fast and efficient method for the huge amount of data.
- In this method, files can be easily stored in cheaper storage mechanism like magnetic tapes.
- It is simple in design. It requires no much effort to store the data.

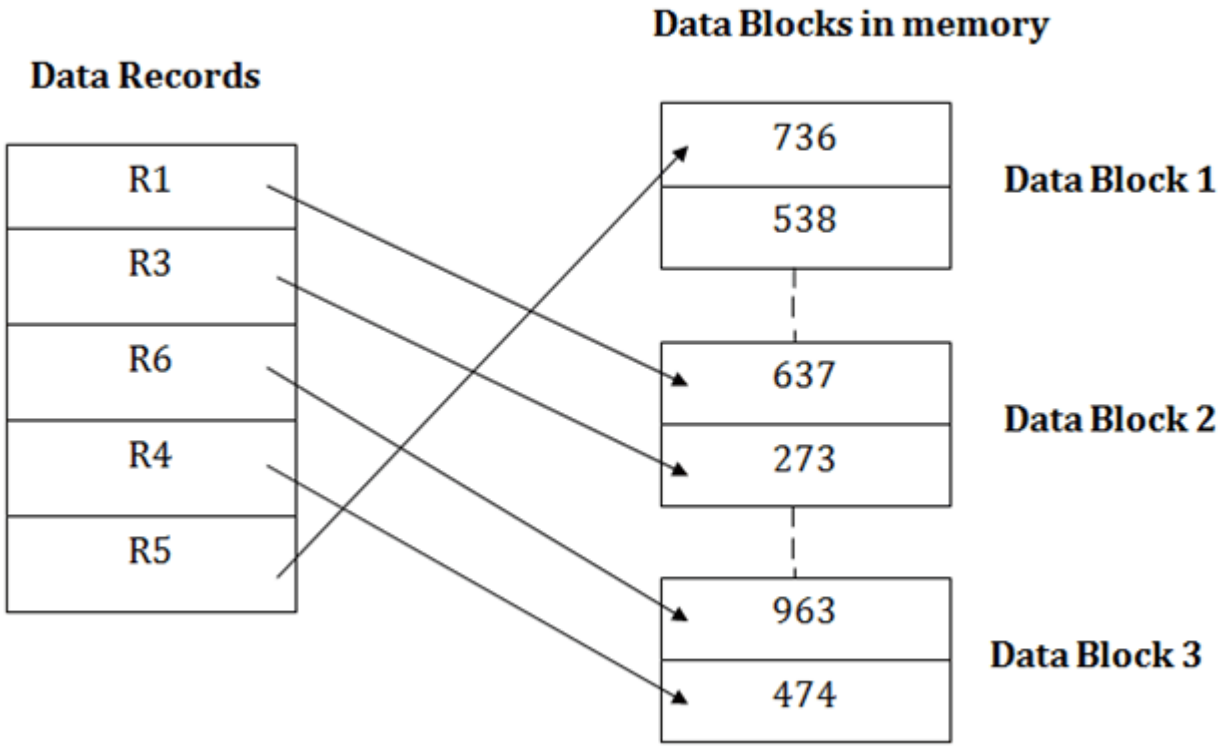
- This method is used when most of the records have to be accessed like grade calculation of a student, generating the salary slip, etc.
- This method is used for report generation or statistical calculations.

Cons of sequential file organization

- It will waste time as we cannot jump on a particular record that is required but we have to move sequentially which takes our time.
- Sorted file method takes more time and space for sorting the records.

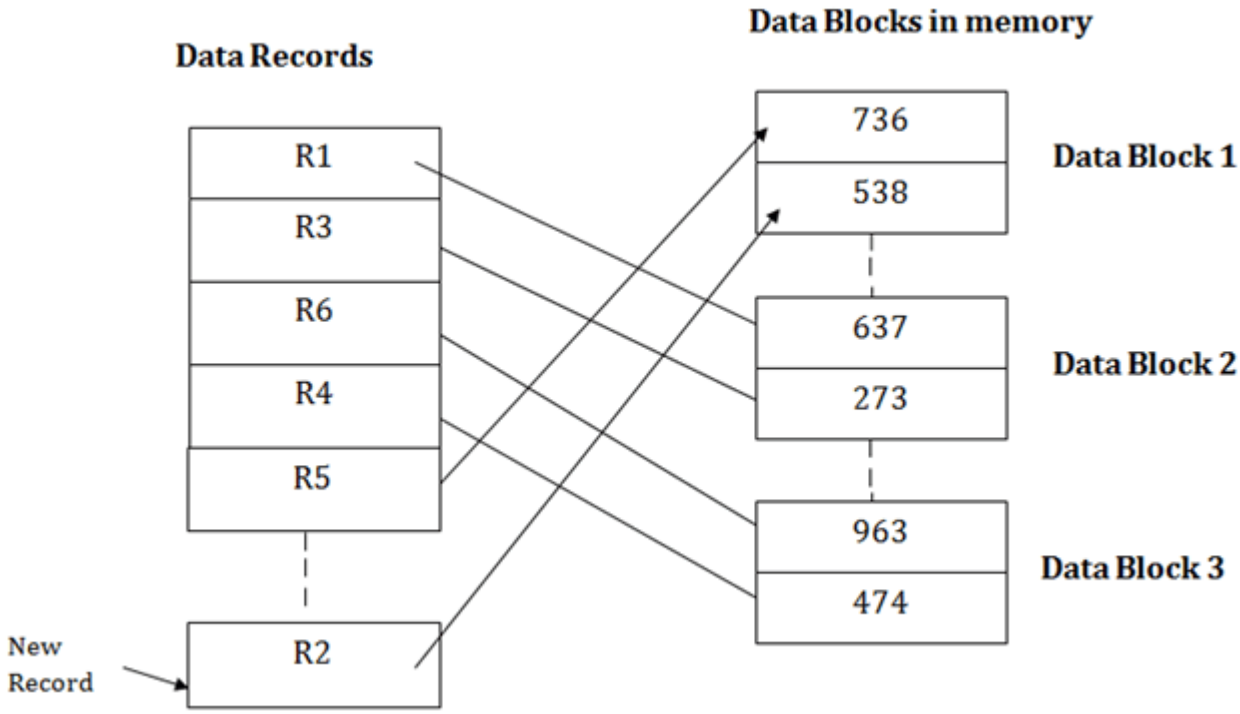
Heap file organization

- It is the simplest and most basic type of organization. It works with data blocks. In heap file organization, the records are inserted at the file's end. When the records are inserted, it doesn't require the sorting and ordering of records.
- When the data block is full, the new record is stored in some other block. This new data block need not to be the very next data block, but it can select any data block in the memory to store new records. The heap file is also known as an unordered file.
- In the file, every record has a unique id, and every page in a file is of the same size. It is the DBMS responsibility to store and manage the new records.



Insertion of a new record

Suppose we have five records R1, R3, R6, R4 and R5 in a heap and suppose we want to insert a new record R2 in a heap. If the data block 3 is full then it will be inserted in any of the database selected by the DBMS, let's say data block 1.



If we want to search, update or delete the data in heap file organization, then we need to traverse the data from staring of the file till we get the requested record.

If the database is very large then searching, updating or deleting of record will be time-consuming because there is no sorting or ordering of records. In the heap file organization, we need to check all the data until we get the requested record.

Pros of Heap file organization

- It is a very good method of file organization for bulk insertion. If there is a large number of data which needs to load into the database at a time, then this method is best suited.
- In case of a small database, fetching and retrieving of records is faster than the sequential record.

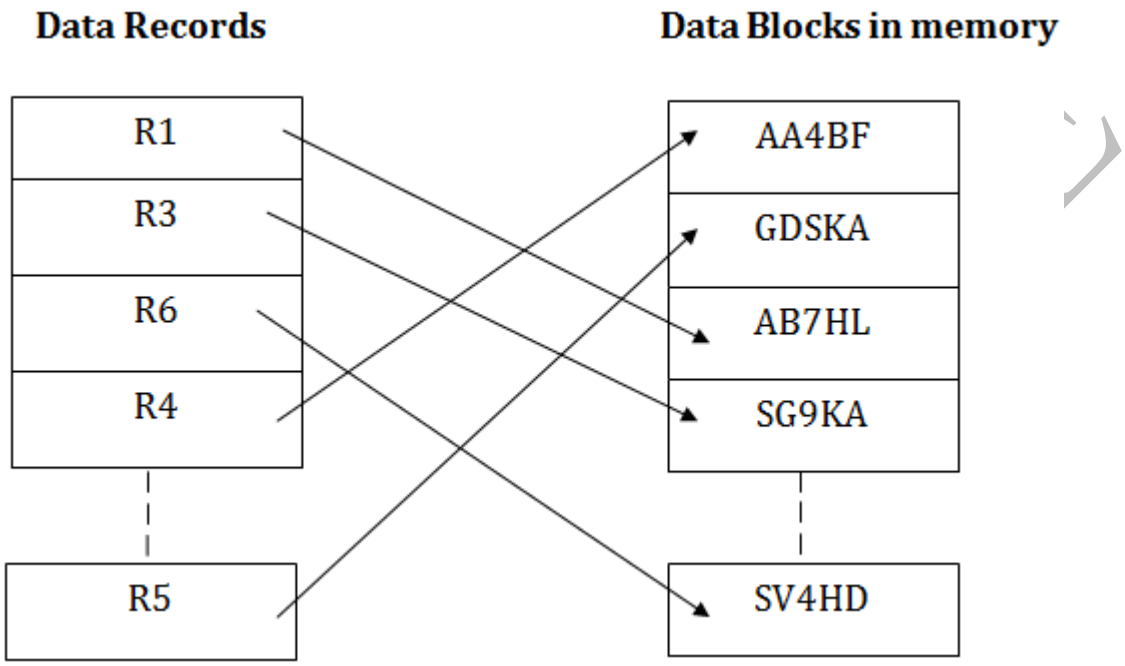
Cons of Heap file organization

- This method is inefficient for the large database because it takes time to search or modify the record.
-

- This method is inefficient for large databases.

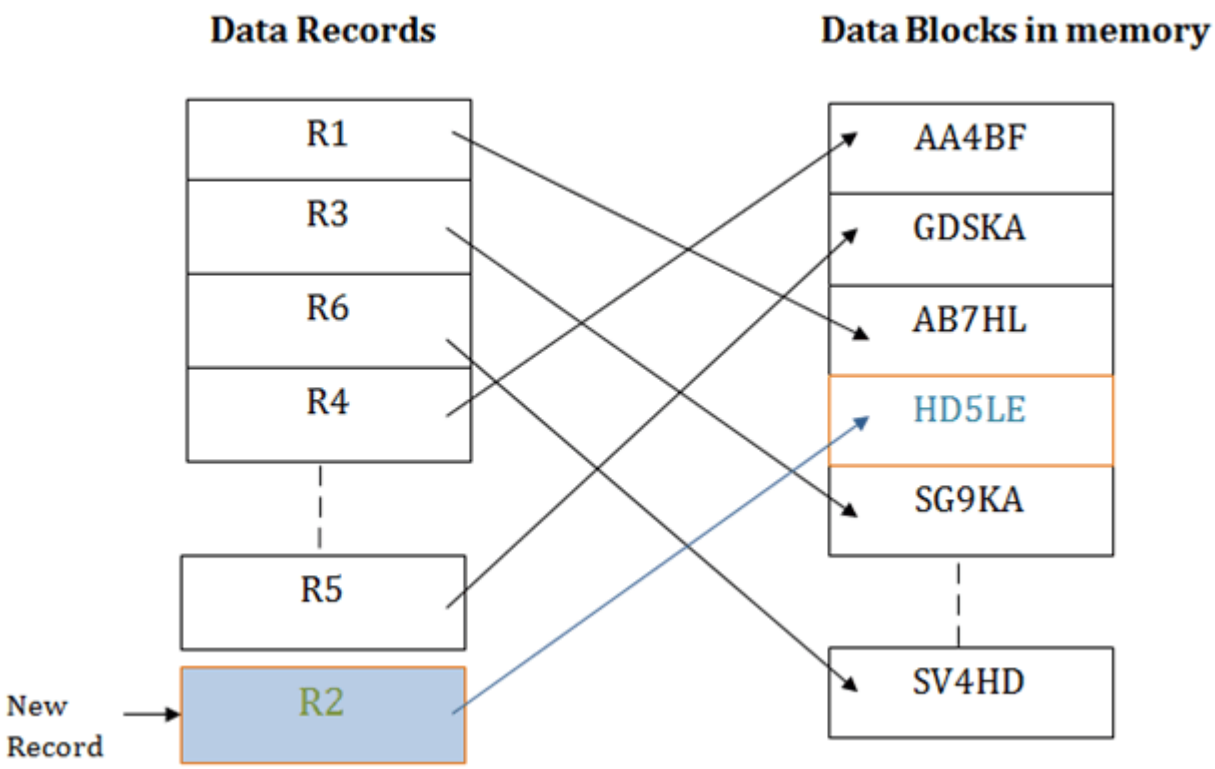
Hash File Organization

Hash File Organization uses the computation of hash function on some fields of the records. The hash function's output determines the location of disk block where the records are to be placed.



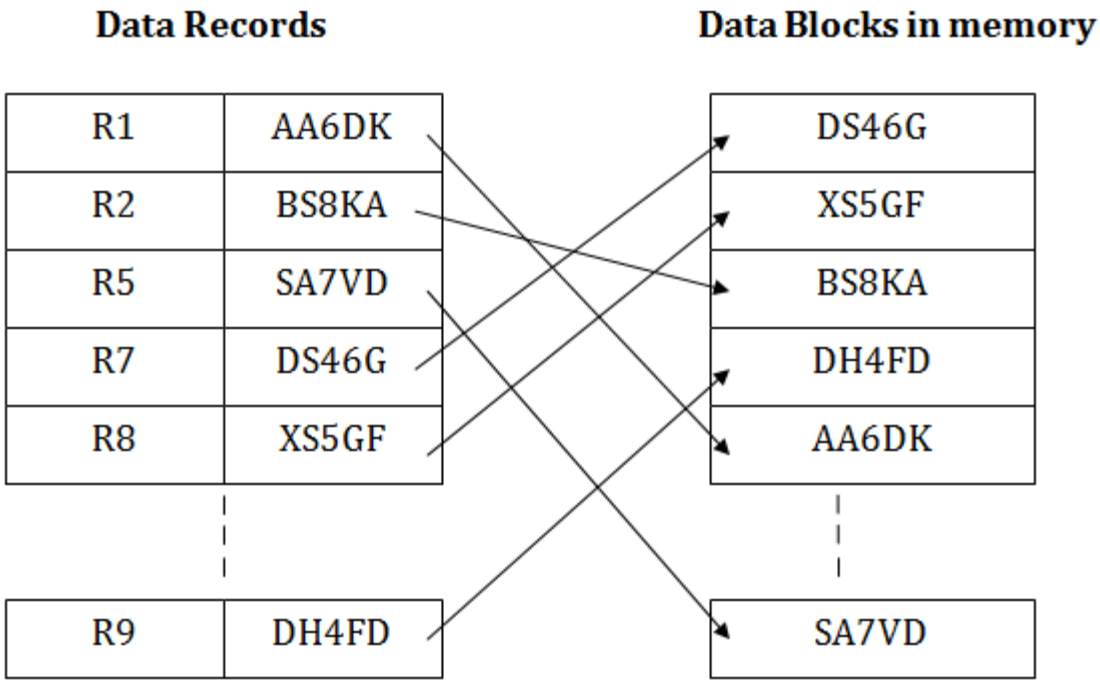
When a record has to be received using the hash key columns, then the address is generated, and the whole record is retrieved using that address. In the same way, when a new record has to be inserted, then the address is generated using the hash key and record is directly inserted. The same process is applied in the case of delete and update.

In this method, there is no effort for searching and sorting the entire file. In this method, each record will be stored randomly in the memory.



Indexed sequential access method (ISAM)

ISAM method is an advanced sequential file organization. In this method, records are stored in the file using the primary key. An index value is generated for each primary key and mapped with the record. This index contains the address of the record in the file.



If any record has to be retrieved based on its index value, then the address of the data block is fetched and the record is retrieved from the memory.

Pros of ISAM:

- In this method, each record has the address of its data block, searching a record in a huge database is quick and easy.
- This method supports range retrieval and partial retrieval of records. Since the index is based on the primary key values, we can retrieve the data for the given range of value. In the same way, the partial value can also be easily searched, i.e., the student name starting with 'JA' can be easily searched.

Cons of ISAM

- This method requires extra space in the disk to store the index value.
- When the new records are inserted, then these files have to be reconstructed to maintain the sequence.
- When the record is deleted, then the space used by it needs to be released. Otherwise, the performance of the database will slow down.

Cluster file organization

- When the two or more records are stored in the same file, it is known as clusters. These files will have two or more tables in the same data block, and key attributes which are used to map these tables together are stored only once.
- This method reduces the cost of searching for various records in different files.
- The cluster file organization is used when there is a frequent need for joining the tables with the same condition. These joins will give only a few records from both tables. In the given example, we are retrieving the record for only particular departments. This method can't be used to retrieve the record for the entire department.

EMPLOYEE

EMP_ID	EMP_NAME	ADDRESS	DEP_ID
1	John	Delhi	14
2	Robert	Gujarat	12
3	David	Mumbai	15
4	Amelia	Meerut	11
5	Kristen	Noida	14
6	Jackson	Delhi	13
7	Amy	Bihar	10
8	Sonoo	UP	12

DEPARTMENT

DEP_ID	DEP_NAME
10	Math
11	English
12	Java
13	Physics
14	Civil
15	Chemistry

Cluster Key

DEP_ID	DEP_NAME	EMP_ID	EMP_NAME	ADDRESS
10	Math	7	Amy	Bihar
11	English	4	Amelia	Meerut
12	Java	2	Robert	Gujarat
12		8	Sonoo	UP
13	Physics	6	Jackson	Delhi
14	Civil	1	John	Delhi
14		5	Kristen	Noida
15	Chemistry	3	David	Mumbai

In this method, we can directly insert, update or delete any record. Data is sorted based on the key with which searching is done. Cluster key is a type of key with which joining of the table is performed.

Types of Cluster file organization:

Cluster file organization is of two types:

1. Indexed Clusters:

In indexed cluster, records are grouped based on the cluster key and stored together. The above EMPLOYEE and DEPARTMENT relationship is an example of an indexed cluster. Here, all the records are grouped based on the cluster key- DEP_ID and all the records are grouped.

2. Hash Clusters:

It is similar to the indexed cluster. In hash cluster, instead of storing the records based on the cluster key, we generate the value of the hash key for the cluster key and store the records with the same hash key value.

Pros of Cluster file organization

- The cluster file organization is used when there is a frequent request for joining the tables with same joining condition.
- It provides the efficient result when there is a 1:M mapping between the tables.

Cons of Cluster file organization

- This method has the low performance for the very large database.
- If there is any change in joining condition, then this method cannot use. If we change the condition of joining then traversing the file takes a lot of time.
- This method is not suitable for a table with a 1:1 condition.

Indexing in DBMS

- Indexing is used to optimize the performance of a database by minimizing the number of disk accesses required when a query is processed.
- The index is a type of data structure. It is used to locate and access the data in a database table quickly.

Index structure:

Indexes can be created using some database columns.

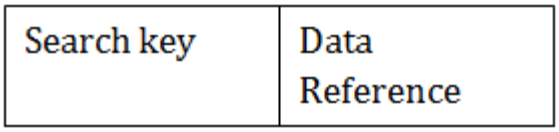
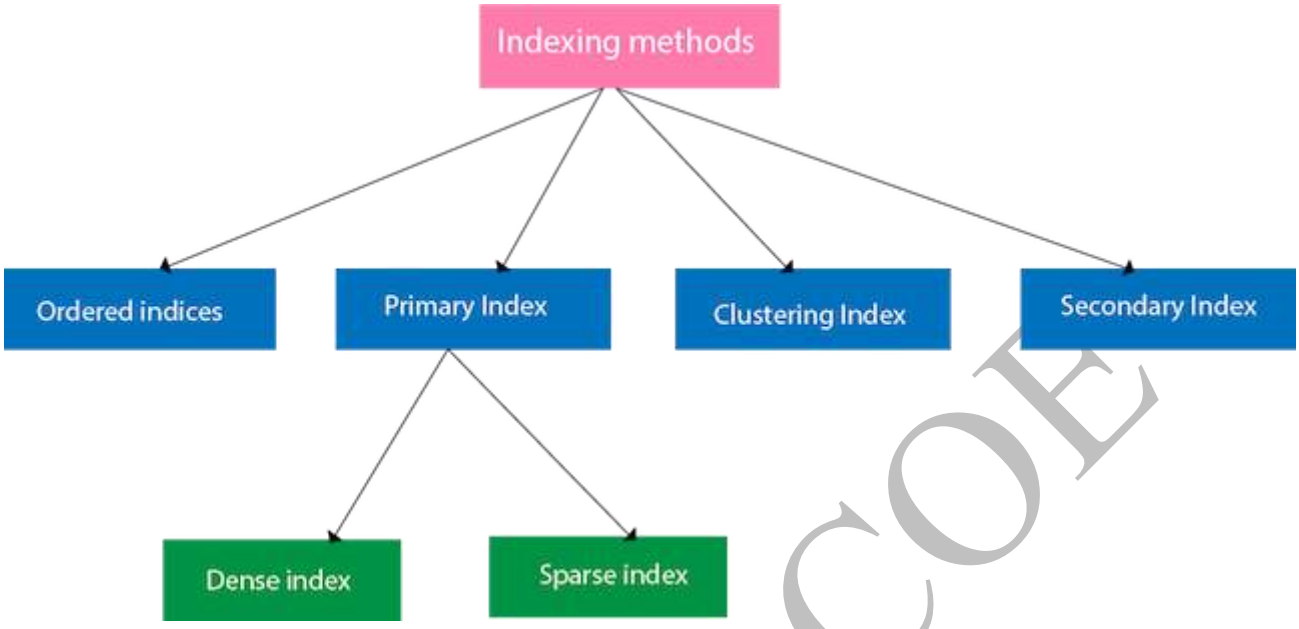


Fig: Structure of Index

- The first column of the database is the search key that contains a copy of the primary key or candidate key of the table. The values of the primary key are stored in sorted order so that the corresponding data can be accessed easily.
- The second column of the database is the data reference. It contains a set of pointers holding the address of the disk block where the value of the particular key can be found.

Indexing Methods



Ordered indices

The indices are usually sorted to make searching faster. The indices which are sorted are known as ordered indices.

Example: Suppose we have an employee table with thousands of record and each of which is 10 bytes long. If their IDs start with 1, 2, 3....and so on and we have to search student with ID-543.

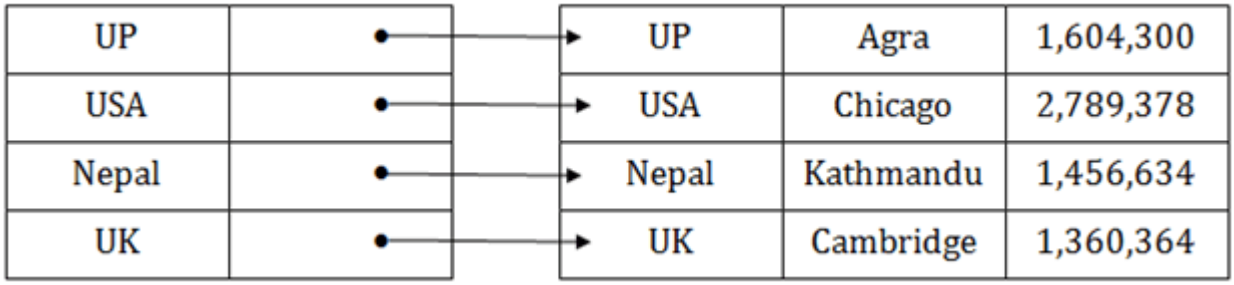
- In the case of a database with no index, we have to search the disk block from starting till it reaches 543. The DBMS will read the record after reading 543*10=5430 bytes.
- In the case of an index, we will search using indexes and the DBMS will read the record after reading 542*2= 1084 bytes which are very less compared to the previous case.

Primary Index

- If the index is created on the basis of the primary key of the table, then it is known as primary indexing. These primary keys are unique to each record and contain 1:1 relation between the records.
- As primary keys are stored in sorted order, the performance of the searching operation is quite efficient.
- The primary index can be classified into two types: Dense index and Sparse index.

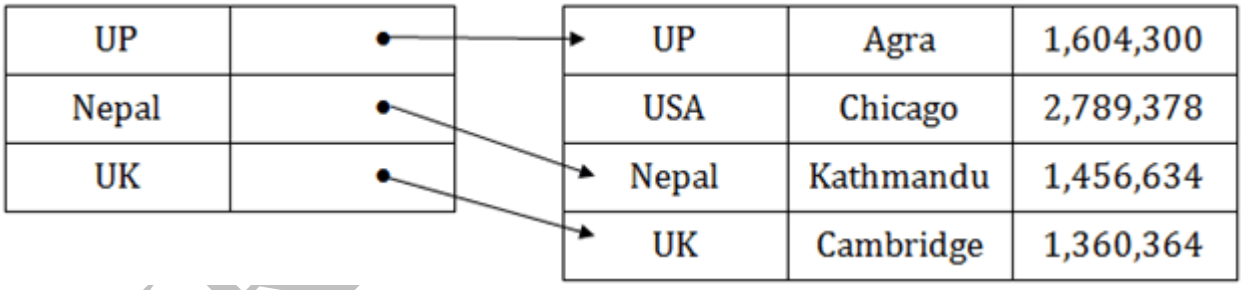
Dense index

- The dense index contains an index record for every search key value in the data file. It makes searching faster.
- In this, the number of records in the index table is same as the number of records in the main table.
- It needs more space to store index record itself. The index records have the search key and a pointer to the actual record on the disk.



Sparse index

- In the data file, index record appears only for a few items. Each item points to a block.
- In this, instead of pointing to each record in the main table, the index points to the records in the main table in a gap.

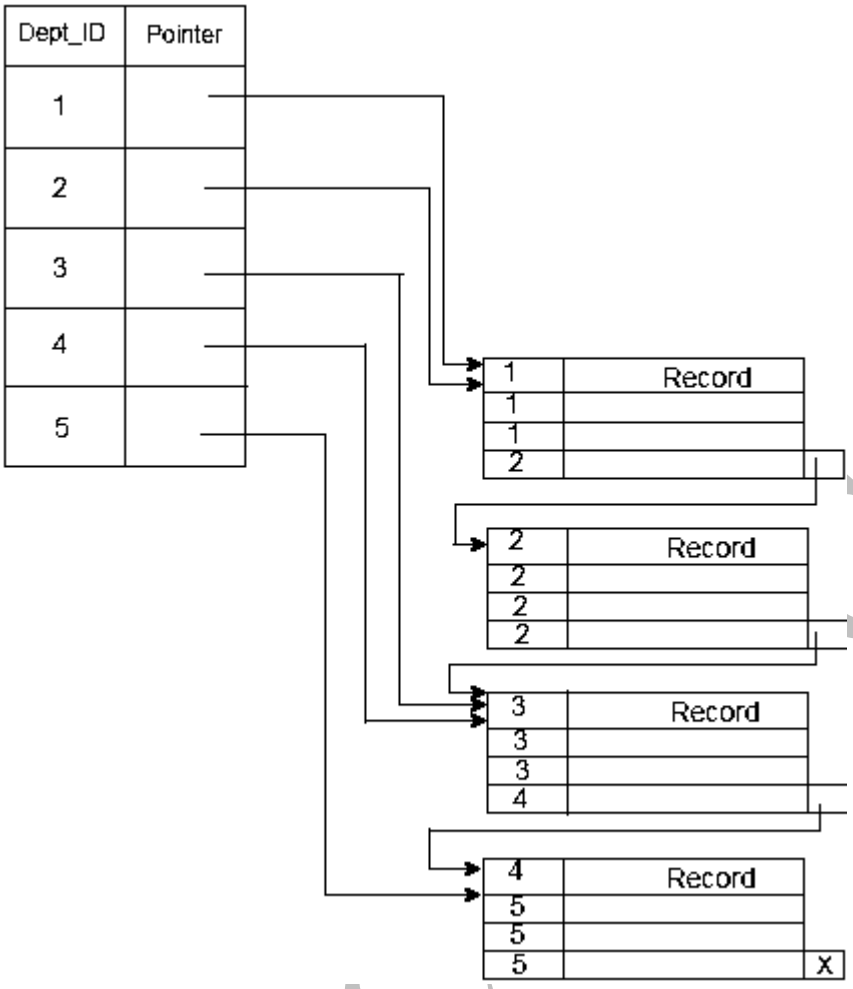


Clustering Index

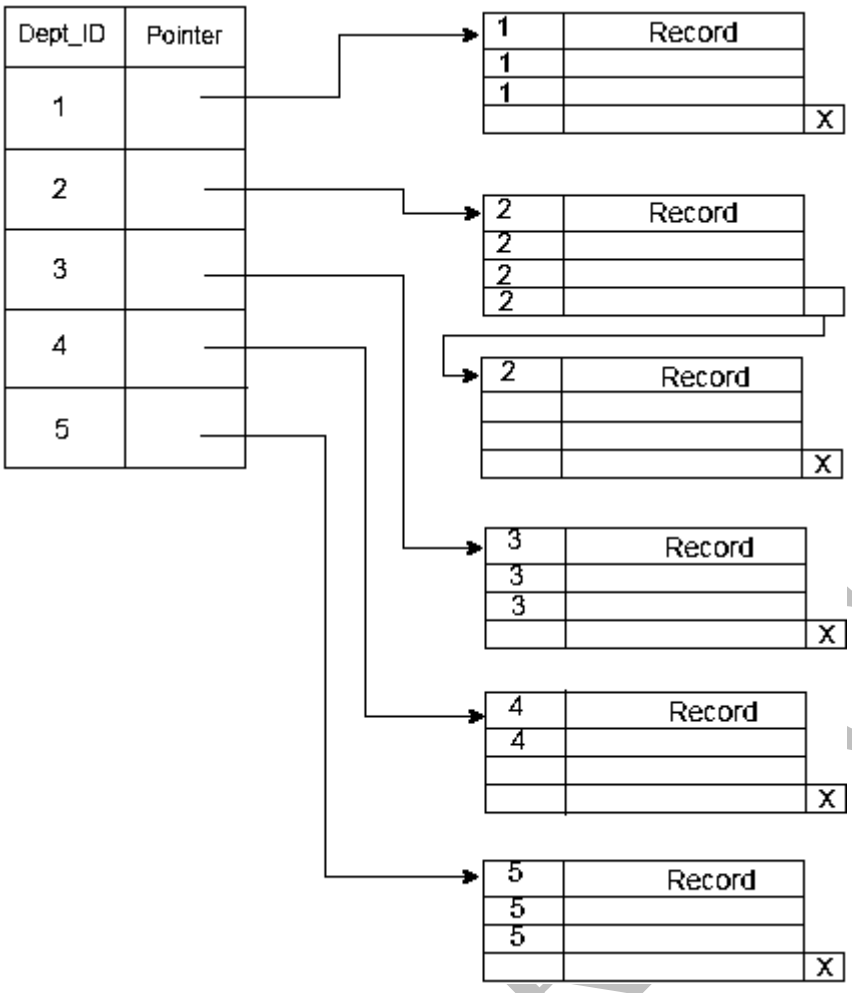
- A clustered index can be defined as an ordered data file. Sometimes the index is created on non-primary key columns which may not be unique for each record.
- In this case, to identify the record faster, we will group two or more columns to get the unique value and create index out of them. This method is called a clustering index.
- The records which have similar characteristics are grouped, and indexes are created for these group.

Example: suppose a company contains several employees in each department. Suppose we use a clustering index, where all employees which belong to the same Dept_ID are

considered within a single cluster, and index pointers point to the cluster as a whole. Here Dept_Id is a non-unique key.



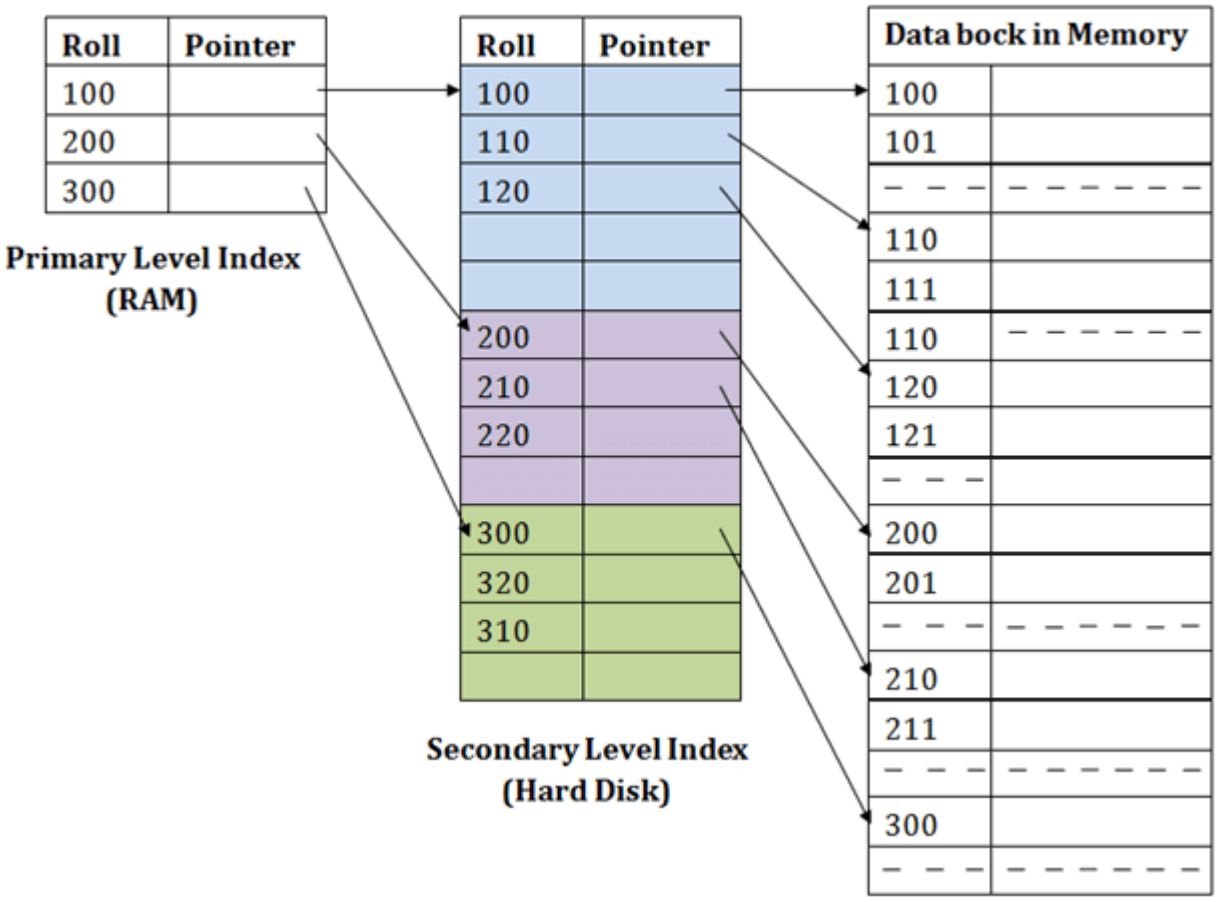
The previous schema is little confusing because one disk block is shared by records which belong to the different cluster. If we use separate disk block for separate clusters, then it is called better technique.



Secondary Index

In the sparse indexing, as the size of the table grows, the size of mapping also grows. These mappings are usually kept in the primary memory so that address fetch should be faster. Then the secondary memory searches the actual data based on the address got from mapping. If the mapping size grows then fetching the address itself becomes slower. In this case, the sparse index will not be efficient. To overcome this problem, secondary indexing is introduced.

In secondary indexing, to reduce the size of mapping, another level of indexing is introduced. In this method, the huge range for the columns is selected initially so that the mapping size of the first level becomes small. Then each range is further divided into smaller ranges. The mapping of the first level is stored in the primary memory, so that address fetch is faster. The mapping of the second level and actual data are stored in the secondary memory (hard disk).



For example:

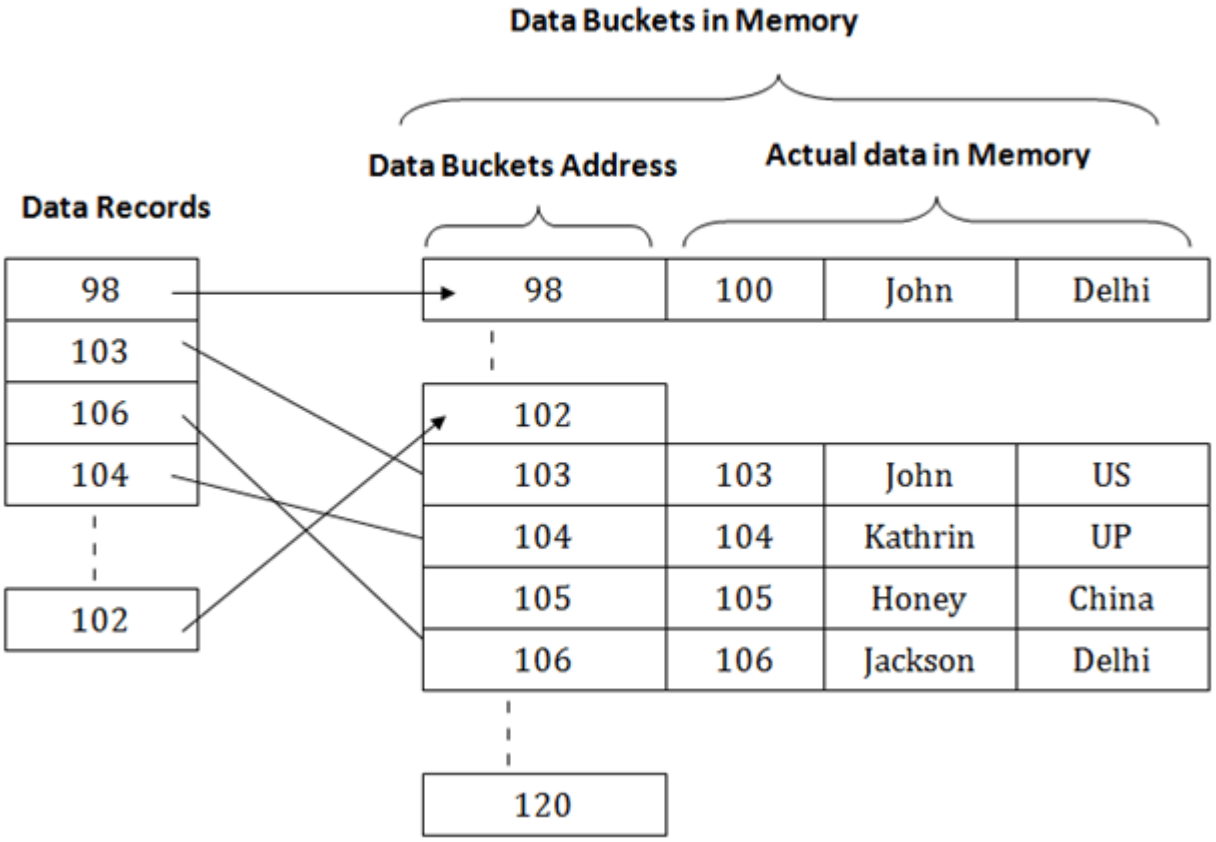
- If you want to find the record of roll 111 in the diagram, then it will search the highest entry which is smaller than or equal to 111 in the first level index. It will get 100 at this level.
- Then in the second index level, again it does max (111) <= 111 and gets 110. Now using the address 110, it goes to the data block and starts searching each record till it gets 111.
- This is how a search is performed in this method. Inserting, updating or deleting is also done in the same manner.

Hashing

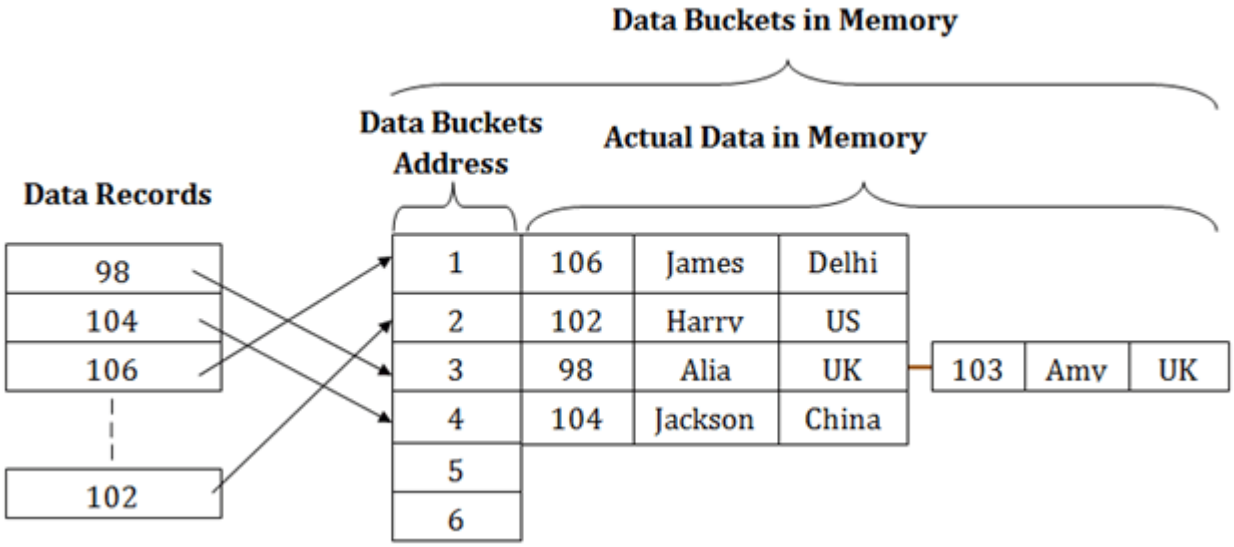
In a huge database structure, it is very inefficient to search all the index values and reach the desired data. Hashing technique is used to calculate the direct location of a data record on the disk without using index structure.

In this technique, data is stored at the data blocks whose address is generated by using the hashing function. The memory location where these records are stored is known as data bucket or data blocks.

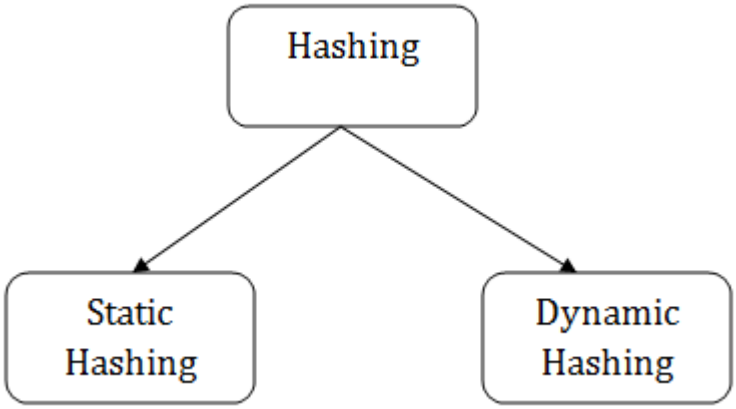
In this, a hash function can choose any of the column value to generate the address. Most of the time, the hash function uses the primary key to generate the address of the data block. A hash function is a simple mathematical function to any complex mathematical function. We can even consider the primary key itself as the address of the data block. That means each row whose address will be the same as a primary key stored in the data block.



The above diagram shows data block addresses same as primary key value. This hash function can also be a simple mathematical function like exponential, mod, cos, sin, etc. Suppose we have mod (5) hash function to determine the address of the data block. In this case, it applies mod (5) hash function on the primary keys and generates 3, 3, 1, 4 and 2 respectively, and records are stored in those data block addresses.



Types of Hashing:

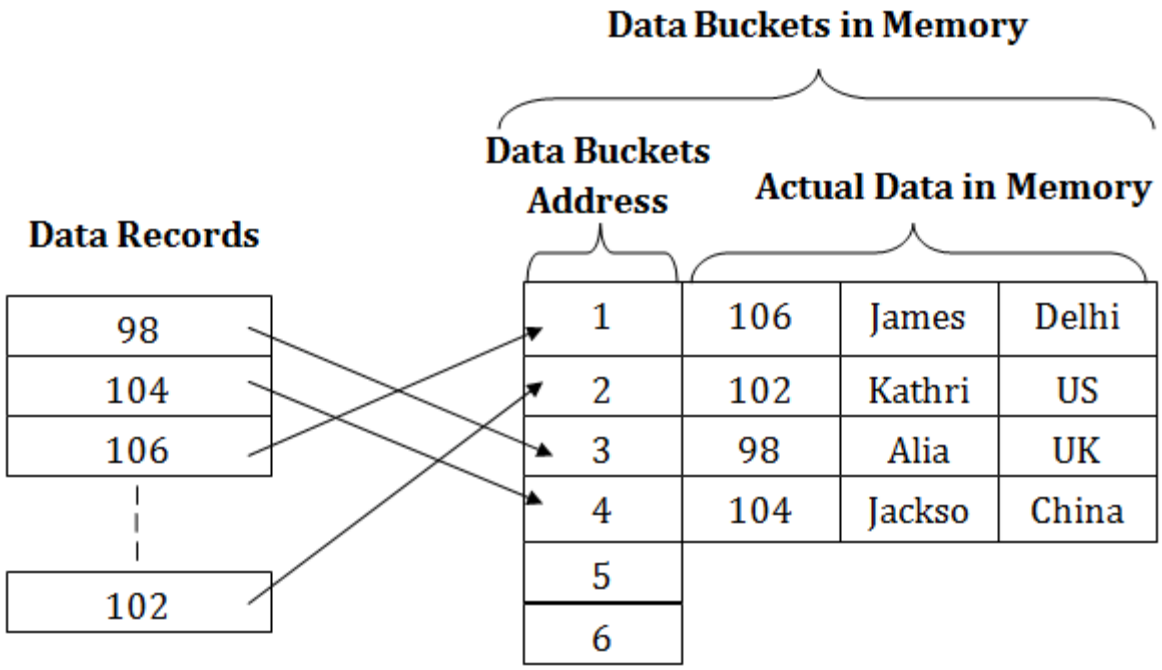


- Static Hashing
- Dynamic Hashing

Static Hashing

In static hashing, the resultant data bucket address will always be the same. That means if we generate an address for EMP_ID =103 using the hash function mod (5) then it will always result in same bucket address 3. Here, there will be no change in the bucket address.

Hence in this static hashing, the number of data buckets in memory remains constant throughout. In this example, we will have five data buckets in the memory used to store the data.



Operations of Static Hashing

- **Searching a record**

When a record needs to be searched, then the same hash function retrieves the address of the bucket where the data is stored.

- **Insert a Record**

When a new record is inserted into the table, then we will generate an address for a new record based on the hash key and record is stored in that location.

- **Delete a Record**

To delete a record, we will first fetch the record which is supposed to be deleted. Then we will delete the records for that address in memory.

- **Update a Record**

To update a record, we will first search it using a hash function, and then the data record is updated.

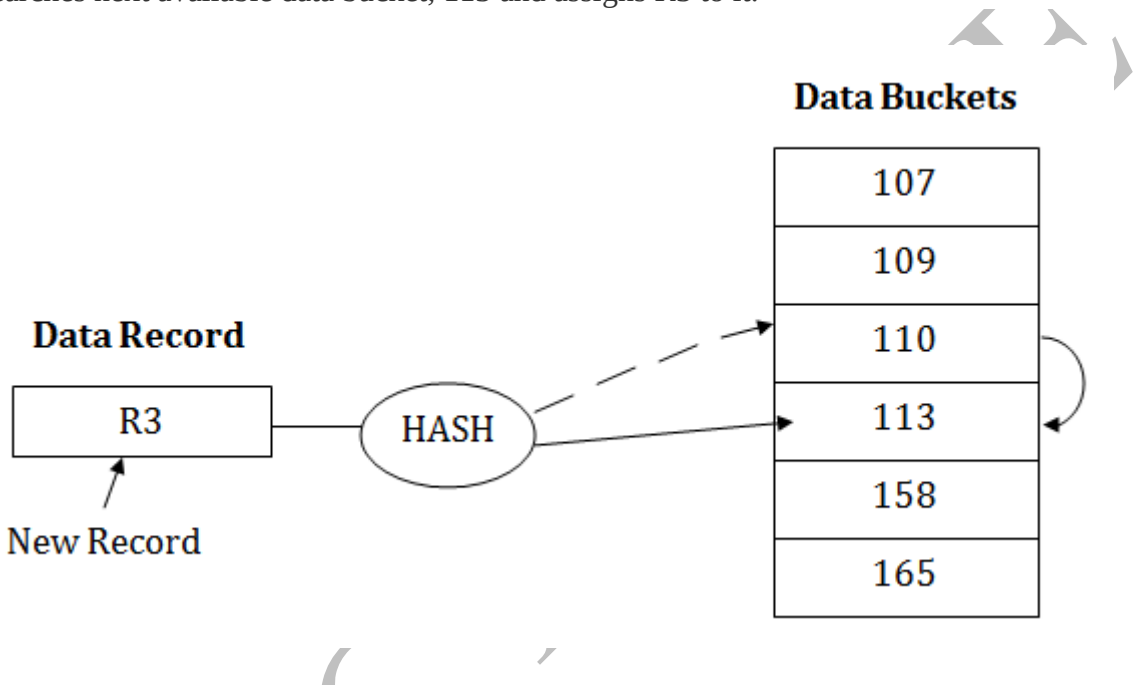
If we want to insert some new record into the file but the address of a data bucket generated by the hash function is not empty, or data already exists in that address. This situation in the static hashing is known as **bucket overflow**. This is a critical situation in this method.

To overcome this situation, there are various methods. Some commonly used methods are as follows:

1. Open Hashing

When a hash function generates an address at which data is already stored, then the next bucket will be allocated to it. This mechanism is called as **Linear Probing**.

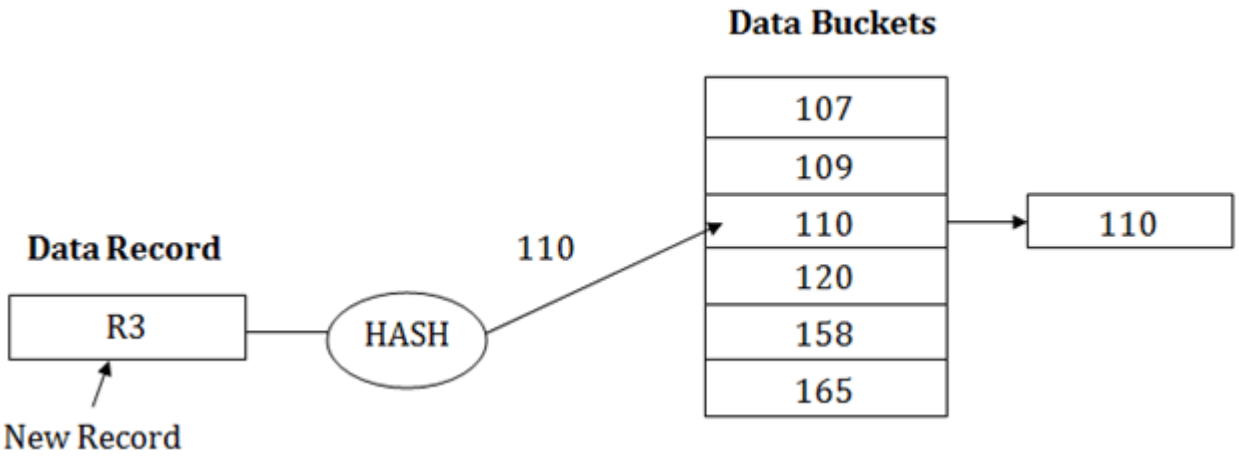
For example: suppose R3 is a new address which needs to be inserted, the hash function generates address as 112 for R3. But the generated address is already full. So the system searches next available data bucket, 113 and assigns R3 to it.



2. Close Hashing

When buckets are full, then a new data bucket is allocated for the same hash result and is linked after the previous one. This mechanism is known as **Overflow chaining**.

For example: Suppose R3 is a new address which needs to be inserted into the table, the hash function generates address as 110 for it. But this bucket is full to store the new data. In this case, a new bucket is inserted at the end of 110 buckets and is linked to it.



Dynamic Hashing

- The dynamic hashing method is used to overcome the problems of static hashing like bucket overflow.
- In this method, data buckets grow or shrink as the records increases or decreases. This method is also known as Extendable hashing method.
- This method makes hashing dynamic, i.e., it allows insertion or deletion without resulting in poor performance.

How to search a key

- First, calculate the hash address of the key.
- Check how many bits are used in the directory, and these bits are called as i.
- Take the least significant i bits of the hash address. This gives an index of the directory.
- Now using the index, go to the directory and find bucket address where the record might be.

How to insert a new record

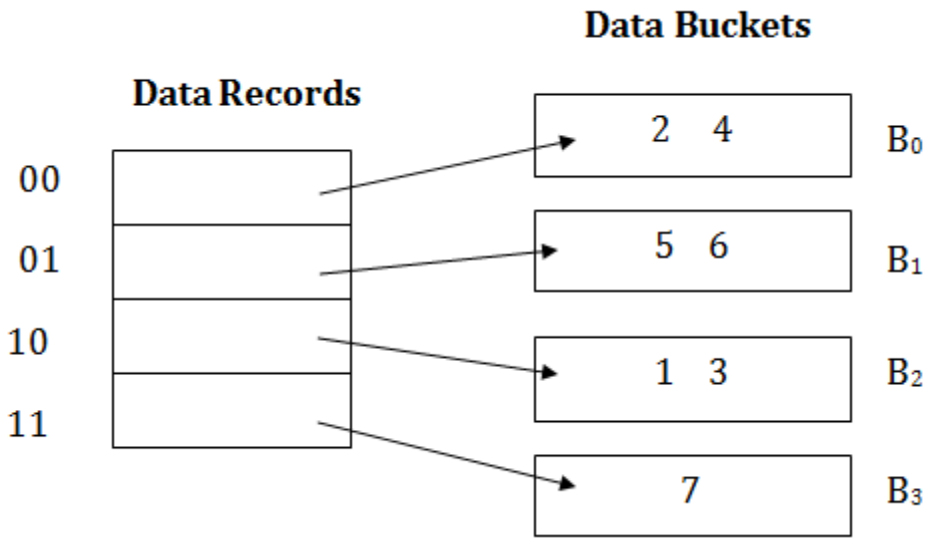
- Firstly, you have to follow the same procedure for retrieval, ending up in some bucket.
- If there is still space in that bucket, then place the record in it.
- If the bucket is full, then we will split the bucket and redistribute the records.

For example:

Consider the following grouping of keys into buckets, depending on the prefix of their hash address:

Key	Hash address
1	11010
2	00000
3	11110
4	00000
5	01001
6	10101
7	10111

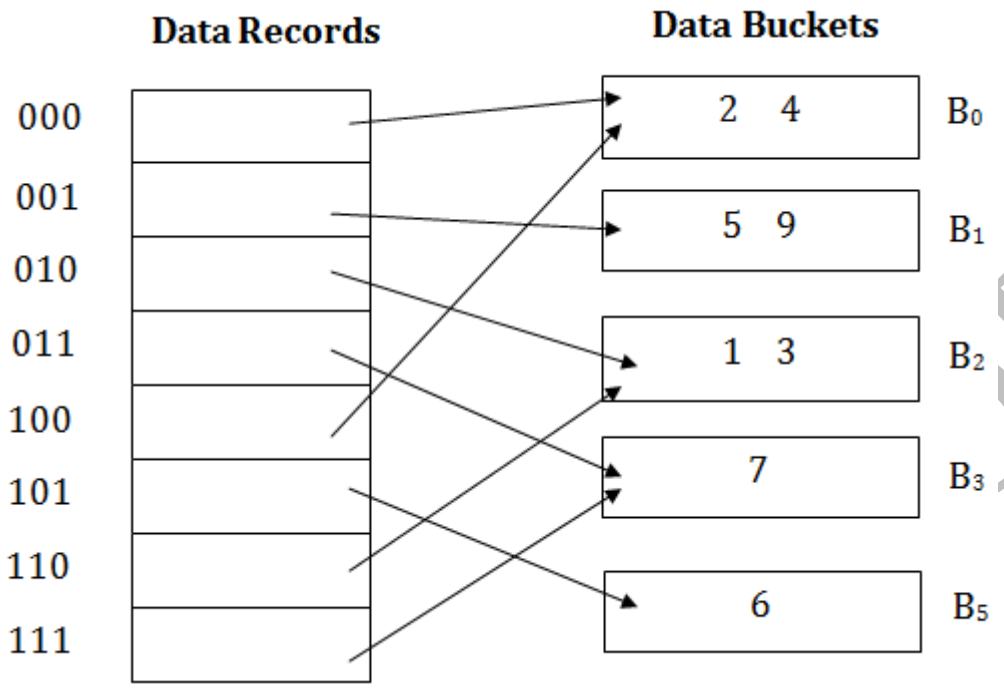
The last two bits of 2 and 4 are 00. So it will go into bucket B0. The last two bits of 5 and 6 are 01, so it will go into bucket B1. The last two bits of 1 and 3 are 10, so it will go into bucket B2. The last two bits of 7 are 11, so it will go into B3.



Insert key 9 with hash address 10001 into the above structure:

- Since key 9 has hash address 10001, it must go into the first bucket. But bucket B1 is full, so it will get split.
- The splitting will separate 5, 9 from 6 since last three bits of 5, 9 are 001, so it will go into bucket B1, and the last three bits of 6 are 101, so it will go into bucket B5.
- Keys 2 and 4 are still in B0. The record in B0 pointed by the 000 and 100 entry because last two bits of both the entry are 00.
- Keys 1 and 3 are still in B2. The record in B2 pointed by the 010 and 110 entry because last two bits of both the entry are 10.

- Key 7 are still in B3. The record in B3 pointed by the 111 and 011 entry because last two bits of both the entry are 11.



Advantages of dynamic hashing

- In this method, the performance does not decrease as the data grows in the system. It simply increases the size of memory to accommodate the data.
- In this method, memory is well utilized as it grows and shrinks with the data. There will not be any unused memory lying.
- This method is good for the dynamic database where data grows and shrinks frequently.

Disadvantages of dynamic hashing

- In this method, if the data size increases then the bucket size is also increased. These addresses of data will be maintained in the bucket address table. This is because the data address will keep changing as buckets grow and shrink. If there is a huge increase in data, maintaining the bucket address table becomes tedious.
- In this case, the bucket overflow situation will also occur. But it might take little time to reach this situation than static hashing.