

UNIT I: Introduction

Syllabus:

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.

Objectives:

After studying this unit, you will be able to:

- Define database management system
- Explain database system applications
- State the characteristics and the database approach
- Understand different data models
- Discuss the advantages and disadvantages of database architecture

Introduction

The information storage and retrieval has become very important in our day-to-day life. The old era of manual system is no longer used in most of the places. For example, to book your airline tickets or to deposit your money in the bank the database systems may be used. The database system makes most of the operations automated. A very good example for this is the billing system used for the items purchased in a super market. Obviously this is done with the help of a database application package. Inventory systems used in a drug store or in a manufacturing industry are some more examples of database. We can add similar kind of examples to this list.

Apart from these traditional database systems, more sophisticated database systems are used in the Internet where a large amount of information is stored and retrieved with efficient search engines. For instance, <http://www.google.com> is a famous web site that enables users to search for their favorite information on the net. In a database we can store starting from text data to very complex data like audio, video, etc.

Database Management Systems (DBMS)

A database is a collection of related data stored in a standard format, designed to be shared by multiple users. A database is defined as “A collection of interrelated data items that can be processed by one or more application programs”.

A database can also be defined as “A collection of persistent data that is used by the application systems of some given enterprise”. An enterprise can be a single individual (with a small personal database), or a complete corporation or similar large body (with a large shared database), or anything in between.

Example: A Bank, a Hospital, a University, a Manufacturing company

Data

Data is the raw material from which useful information is derived. The word data is the plural of Datum. Data is commonly used in both singular and plural forms. It is defined as raw facts or observations. It takes variety of forms, including numeric data, text and voice and images. Data is a collection of facts, which is unorganized but can be made organized into useful information. The term Data and Information come across in our daily life and are often interchanged.

Example: Weights, prices, costs, number of items sold etc.

Information

Data that have been processed in such a way as to increase the knowledge of the person who uses the data. The term data and information are closely related. Data are raw material resources that are processed into finished information products. The information as data that has been processed in such way that it can increase the knowledge of the person who uses it.

In practice, the database today may contain either data or information.

Data Processing

The process of converting the facts into meaningful information is known as data processing. Data processing is also known as information processing.

Metadata

Data that describe the properties or characteristics of other data.

Data is only become useful when placed in some context. The primary mechanism for providing context for data is Metadata. Metadata are data that describe the properties, or characteristics of other data. Some of these properties include data definition, data structures and rules or constraints. The Metadata describes the properties of data but do not include that data.

Database System Applications

Databases are widely used. Here are some representative applications:

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.

File Systems Versus A DBMS (Characteristics)

In earlier days, the databases were created directly on top of file systems. File system has many disadvantages.

1. Not enough primary memory to process large data sets. If data is maintained in other storage devices like disks, tapes and bringing relevant data to main memory, it increases the cost of performance. Problem in accessing the large data due to addressing the data using 32 bit or 64 bit mode addressing mechanism.
2. Programs must be written to process the user request to process the data stored in files which are complex in nature because of large volume of data to be searched.
3. Inconsistent data and complexity in providing concurrent accesses.
4. Not sufficiently flexible to enforce security policies in which different users have permission to access different subsets of the data.

A DBMS is a piece of software that is designed to make the preceding tasks easier. By storing data in a DBMS, rather than as a collection of operating system Files, we can use the DBMS's features to manage the data in a robust and efficient manner.

Advantages of DBMS

One of the main advantages of using a database management system is that the organization can exert via the DBA, centralized management and control over the data. The database administrator is the focus of the centralized control.

The following are the major advantages of using a Database Management System (DBMS):

Data independence: Application programs should be as independent as possible from details of data representation and storage. The DBMS can provide an abstract view of the data to insulate application code from such details.

Efficient data access: A DBMS utilizes a variety of sophisticated techniques to store and retrieve data efficiently. This feature is especially important if the data is stored on external storage devices.

Data integrity and security: The DBMS can enforce integrity constraints on the data. The DBMS can enforce access controls that govern what data is visible to different classes of users.

Data administration: When several users share the data, centralizing the administration of data can offer significant improvements. It can be used for organizing the data representation to minimize redundancy and for fine-tuning the storage of the data to make retrieval efficient.

Concurrent access and crash recovery: A DBMS schedules concurrent accesses to the data in such a manner that users can think of the data as being accessed by only one user at a time. Further, the DBMS protects users from the effects of system failures.

Reduced application development time: Clearly, the DBMS supports many important functions that are common to many applications accessing data stored in the DBMS.

Disadvantages of DBMS

The disadvantage of the DBMS system is **overhead cost**. The processing overhead introduced by the DBMS to implement security, integrity, and sharing of the data causes a degradation of the response and throughput times. An additional cost is that of migration from a traditionally separate application environment to an integrated one.

Even though centralization reduces duplication, the lack of duplication requires that the database be adequately backup so that in the case of failure the data can be recovered.

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

Data Models

A **data model** is a collection of high-level data description constructs that hide many low-level storage details. A DBMS allows a user to define the data to be stored in terms of a data model. Most database management systems today are based on the **relational data model**.

A schema is a description of a particular collection of data, using the given data model. The relational model of data is the most widely used model today.

Main concept: relation, basically a table with rows and columns.

Every relation has a schema, which describes the columns, or fields.

Data Model is a collection of high-level data description constructs that hide many low-level storage details. A DBMS allows a user to define the data to be stored in terms of a data model. Most database management systems today are based on the Relational data model. Relational models include – IBM's DB2, Informix, Oracle, Sybase, Microsoft's Access, Foxbase, Paradox, Tandem and Teradata.

Categories of data models

- **Conceptual (high-level, semantic) data models:** Provide concepts that are close to the way many users perceive data (Also called entity-based or object-based data models).
- **Physical (low-level, internal) data models:** Provide concepts that describe details of how data is stored in the computer.
- **Implementation (representational) data models:** Provide concepts that fall between the above two.

1. Hierarchical models:

Advantages:

- Hierarchical model is simple to construct and operate on.
- Corresponds to a number of natural hierarchical organized domains – e.g., assemblies in manufacturing, personal organization in companies.
- Language is simple; uses constructs like GET, GET UNIQUE, GET NEXT, GET NEXT WITHIN PARENT etc.,

Disadvantages:

- Navigational and procedural nature of processing.
- Database is visualized as a linear arrangement of records.
- Little scope for “query optimization”.
- One-to-many relationships.

2. Network model:

Advantages:

- Network model is able to model complex relationships and represents semantics of add/delete on the relationships.
- Can handle most situations for modeling using record types and relationship types.
- Language is navigational; uses constructs like FIND, FIND member, FIND owner, FIND NEXT within set, GET etc. Programmers can do optimal navigation through the database.

Disadvantages:

- Navigational and procedural nature of processing.
- Database contains a complex array of pointers that are expensive and difficult to update when inserting and deleting.
- Little scope for automated “query optimization”.

3. **Relational model:**

- A relation, basically a table with rows and columns.
- Every relation has a schema, which describes the columns, or fields.
- Student information in a university database may be stored in a relation with the following schema
- Students (sid: string, name: string, login: string, age: integer, gpa: real)

Levels of Abstraction in a DBMS (Three tier schema architecture)

The data in a DBMS is described at three levels of abstraction.

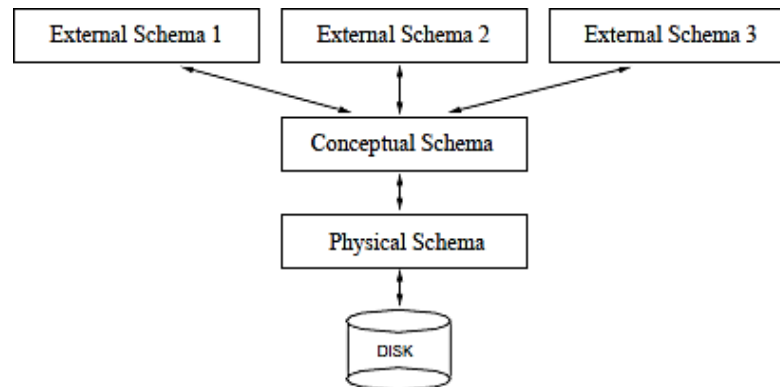
The database description consists of a schema at each of these three levels of abstraction.

External, Conceptual and Physical

Views describe how users see the data.

Conceptual schema defines logical structure.

Physical schema describes the files and indexes used.



Conceptual schema:

- The conceptual schema(also called as logical schema) describes the stored data in terms of the data model of the DBMS.
- In a relational DBMS, the conceptual schema describes all relations that are stored in the database.
- In our sample university database, these relations contain information about entities, such as students and faculty, and about relationships, such as students' enrollment in courses.

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

Faculty(fid: string, fname: string, salary : real)

Courses(cid: string, cname: string, credits: integer)

Rooms(nw: integer, address: string, capacity: integer)

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

Teaches (fid: string, cid: string)

The choice of relations, and the choice of fields for each relation, is not always obvious, and the process of arriving at a good conceptual schema is called conceptual database design.

Physical Schema:

- The physical schema specifies storage details.
- It summarizes how the relations described in the conceptual schema are actually stored on secondary storage devices such as disks and tapes.
- Decides what file organizations to use to store the relations and create auxiliary data structures, called indexes, to speed up data retrieval operations.
- A sample physical schema for the university database is to store all relations as unsorted files of records.
 - Create indexes on the first column of the students, faculty and courses relations, the salary column of faculty, and the capacity of column of rooms.

External Schema:

- This schema allows data access to be customized at the level of individual users or groups of users.
- A database has exactly one conceptual schema and one physical schema, but it may have several external schemas.
- An external schema is a collection of one or more views and relations from the conceptual schema.
- A view is conceptually a relation, but the records in a view are not stored in the DBMS.

Data Independence

Application programs are insulated from changes in the way the data is structured and stored.

Data independence is achieved through use of the three levels of data abstraction.

Logical data independence: users can be shielded from changes in the logical structure of the data, or changes in the choice of relations to be stored. This is the independence to change the conceptual schema without having to change the external schemas and their application programs.

Physical data independence: the conceptual schema insulated users from changes in physical storage details. This is the independence to change the internal schema without having to change the conceptual schema.

Architecture of a DBMS

The functional components of a database system can be broadly divided into query processor components and storage manager components. The query processor includes:

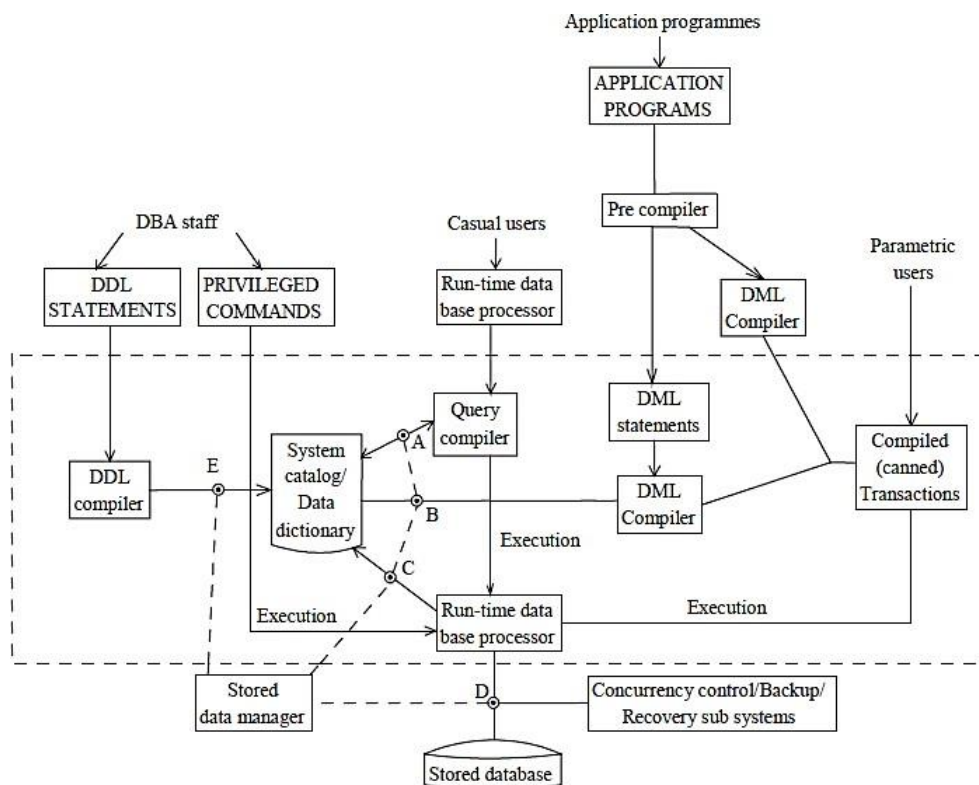
1. **DML Compiler:** It translates DML statements in a query language into low-level instructions that the query evaluation engine understands.
2. **Embedded DML Pre-compiler:** It converts DML statements embedded in an application program to normal procedure calls in the host language. The pre-compiler must interact with the DML compiler to generate the appropriate code.
3. **DDL Interpreter:** It interprets DDL Stateline its and records them in a set of tables containing

metadata.

4. **Transaction Manager:** Ensures that the database remains in a consistent (correct) state despite system failures, and that concurrent transaction executions proceed without conflicting.
5. **File Manager:** Manages the allocation of space on disk storage and the data structures used to represent information stored on disk.
6. **Buffer Manager:** Is responsible for fetching data from disk storage into main memory and deciding what data to cache in memory.

Also some data structures are required as part of the physical system implementation:

1. **Data Files:** The data files store the database by itself.
2. **Data Dictionary:** It stores metadata about the structure of the database, as it is used heavily.
3. **Indices:** It provides fast access to data items that hold particular values.
4. **Statistical Data:** It stores statistical information about the data in the database. This information used by the query processor to select efficient ways to execute a query.



People Who Deal With Databases

Quite a variety of people are associated with the creation and use of databases. Obviously, there are **database implementors**, who build DBMS software, and **end users** who wish to store and use data in a DBMS.

Database implementors work for vendors such as IBM or Oracle. End users come from a diverse and increasing number of fields.

In addition to end users and implementors, two other classes of people are associated with a DBMS: **application programmers** and **database administrators (DBAs)**.

Database application programmers develop packages that facilitate data access for end users, who are usually not computer professionals, using the host or data languages and software tools that DBMS vendors provide.

The task of designing and maintaining the database is entrusted to a professional called the **database administrator**.

The DBA is responsible for many critical tasks:

- **Design of the conceptual and physical schemas:** The DBA is responsible for interacting with the users of the system to understand what data is to be stored in the DBMS and how it is likely to be used. Based on this knowledge, the DBA must design the conceptual schema (decide what relations to store) and the physical schema (decide how to store them).
- **Security and authorization:** The DBA is responsible for ensuring that unauthorized data access is not permitted. In general, not everyone should be able to access all the data. In a relational DBMS, users can be granted permission to access only certain views and relations.
- **Data availability and recovery from failures:** The DBA must take steps to ensure that if the system fails, users can continue to access as much of the uncorrupted data as possible.
- **Database tuning:** The needs of users are likely to evolve with time. The DBA is responsible for modifying the database, in particular the conceptual and physical schemas, to ensure adequate performance as user requirements change.

Database Environment

A database management system (DBMS) is a collection of programs that enables users to create and maintain a database. The DBMS is hence a *general-purpose software system* that facilitates the processes of *defining, constructing, manipulating, and sharing* databases among various users and applications.

Defining a database involves specifying the data types, structures, and constraints for the data to be stored in the database.

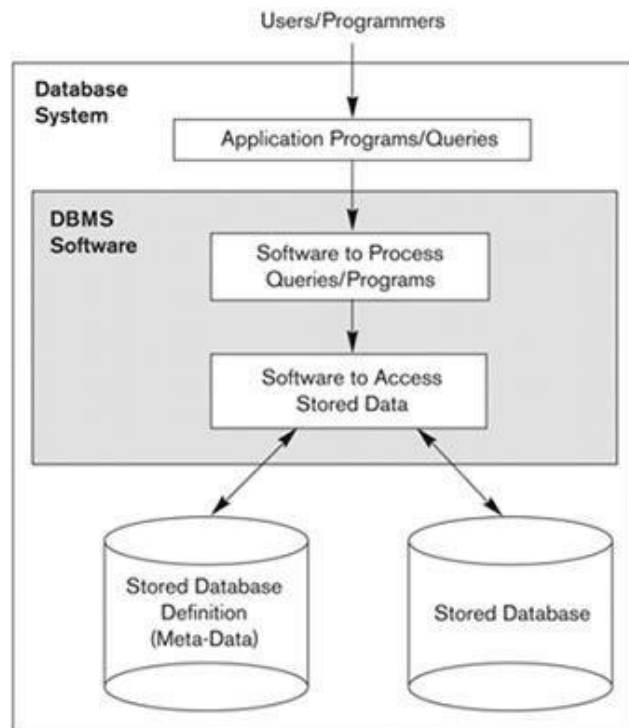
Constructing the database is the process of storing the data itself on some storage medium that is controlled by the DBMS.

Manipulating a database includes such functions as querying the database to retrieve specific data, updating the database to reflect changes in the mini world, and generating reports from the data.

Sharing a database allows multiple users and programs to access the database concurrently.

Other important functions provided by the DBMS include *protecting* the database and *maintaining* it over a long period of time.

Protection includes both *system protection* against hardware or software malfunction (or crashes), and *security protection* against unauthorized or malicious access. A typical large database may have a life cycle of many years, so the DBMS must be able to maintain the database system by allowing the system to evolve as requirements change over time. We can call the database and DBMS software together a database system.



Database Architecture

Database architecture uses programming languages to design a particular type of software for businesses or organizations. Database architecture focuses on the design, development, implementation and maintenance of computer programs that store and organize information for businesses, agencies and institutions.

The architecture of a DBMS can be seen as either single tier or multi-tier. The tiers are classified as follows:

1-tier architecture

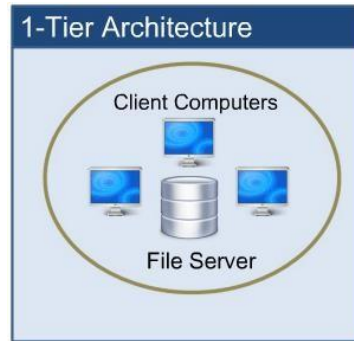
2-tier architecture

3-tier architecture

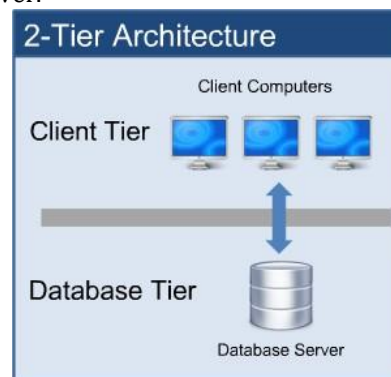
n-tier architecture

1-tier architecture:

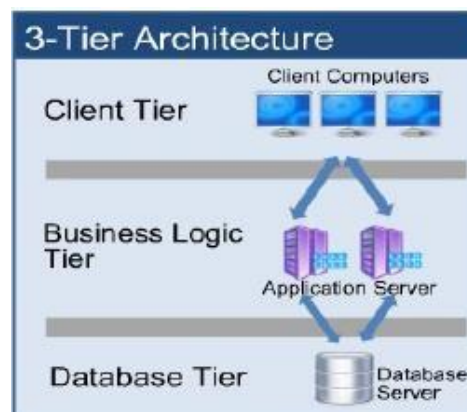
One-tier architecture involves putting all of the required components for a software application or technology on a single server or platform.

**2-tier architecture:**

The two-tier is based on Client Server architecture. The two-tier architecture is like client server application. The direct communication takes place between client and server. There is no intermediate between client and server.

**3-tier architecture:**

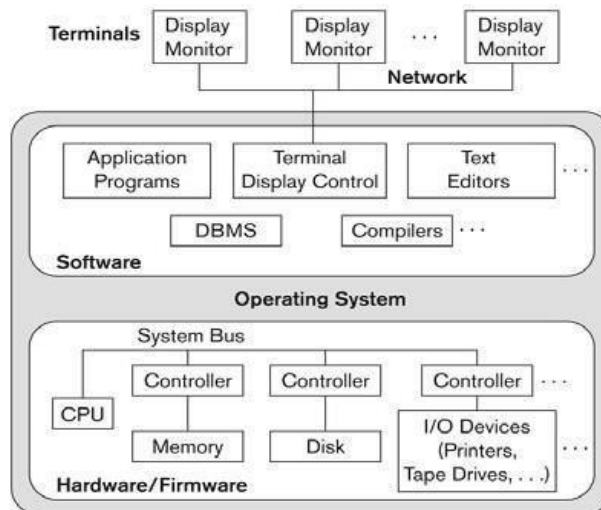
A 3-tier architecture separates its tiers from each other based on the complexity of the users and how they use the data present in the database. It is the most widely used architecture to design a DBMS.

**Centralized DBMS Architecture**

Architectures for DBMSs have followed trends similar to those for general computer system architectures. Earlier architectures used mainframe computers to provide the main processing for all functions of the system, including user application programs and user interface programs, as well as all the DBMS functionality.

As prices of hardware declined, most users replaced their terminals with personal computers (PCs) and workstations. At first, database systems used these computers in the same way as they had used display terminals, so that the DBMS itself was still a centralized DBMS in which all the DBMS functionality, application program execution, and user interface processing were carried out on one machine.

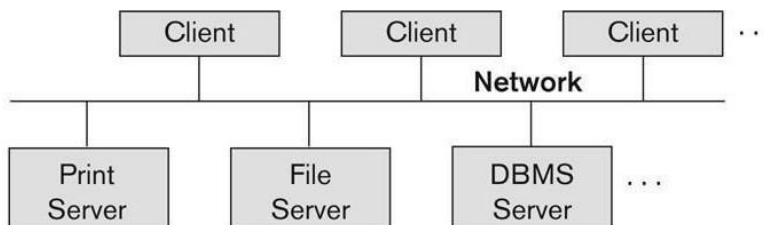
Gradually, DBMS systems started to exploit the available processing power at the user side, which led to client/server DBMS architectures.



Client/Server Architecture:

The client/server architecture was developed to deal with computing environments in which a large number of PCs, workstations, file servers, printers, database servers, Web servers, and other equipment are connected via a network. The idea is to define specialized servers with specific functionalities.

The resources provided by specialized servers can be accessed by many client machines. The client machines provide the user with the appropriate interfaces to utilize these servers, as well as with local processing power to run local applications. This concept can be carried over to software, with specialized software-such as a DBMS or a CAD (computer-aided design) package being stored on specific server machines and being made accessible to multiple clients.

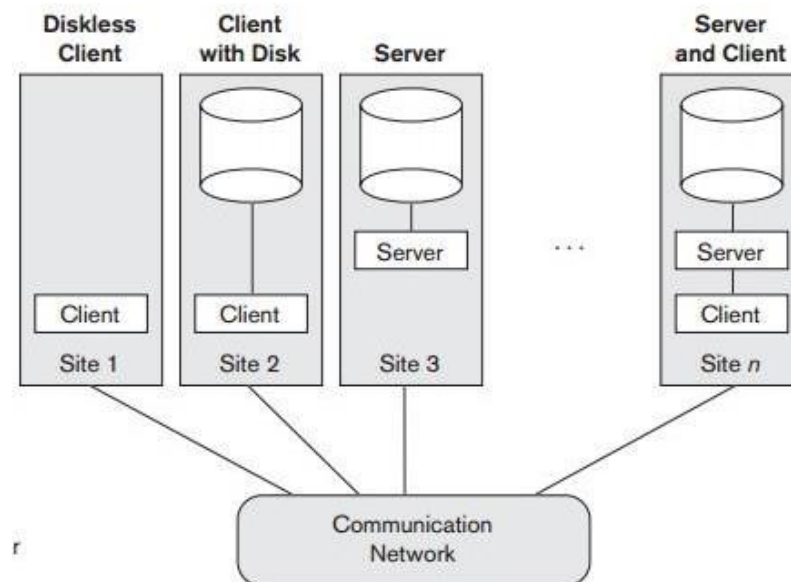


The concept of client/server architecture assumes an underlying framework that consists of many PCs and workstations as well as a smaller number of mainframe machines, connected via local area networks and other types of computer networks. A client in this framework is typically a user machine that provides user interface capabilities and local processing. When a client

requires access to additional functionality-such as database access-that does not exist at that machine, it connects to a server that provides the needed functionality.

A server is a machine that can provide services to the client machines, such as file access, printing, archiving, or database access. In the general case, some machines install only client software, others only server software, and still others may include both client and server software. However, it is more common that client and server software usually run on separate machines.

In client/server architecture, the user interface programs and application programs can run on the client side. When DBMS access is required, the program establishes a connection to the DBMS (which is on the server side); once the connection is created, the client program can communicate with the DBMS. A standard called Open Database Connectivity (ODBC) provides an application programming interface (API), which allows client-side programs to call the DBMS, as long as both client and server machines have the necessary software installed. Most DBMS vendors provide ODBC drivers for their systems.



Review Questions

1. Distinguish between database systems and file systems.
2. Discuss about the client server architecture of the database.
3. Define DBMS. Explain database users in detail.
4. What are advantages of DBMS? Explain.
5. With a neat diagram, explain the structure of Database Management System.
6. What is data independence and how does a DBMS support it? Explain.
7. What is a Database model? List out various database models and explain any two of them.
8. Explain the difference between external, logical and physical level schemas. How are these different schema layers related to the concepts of logical and physical data independence?
9. Compare and contrast various Data Models.
10. Demonstrate data abstraction implementation in DBMS.
11. List and explain various data models used for database design.
12. Explain about Buffer management in DBMS.

13. Who are the different database users? Explain their interfaces to database management system.
14. Describe the client server architecture for the database with necessary diagram.
15. Define Schema. Explain three level architecture in DBMS.
16. Explain Data Independence and its types in detail.
17. How does DBMS provide data abstraction? Explain the concept of data independence.
18. With a neat diagram describe the overall system structure of DBMS.

References:

- Raghurama Krishnan, Johannes Gehrke, *Database Management Systems*, 3rd Edition, Tata McGraw Hill.
- C.J. Date, *Introduction to Database Systems*, Pearson Education.
- Elmasri Navrate, *Fundamentals of Database Systems*, Pearson Education.

DATABASEMANAGEMENTSYSTEMS

UNITII:RelationalModel&BasicSQL

Syllabus:

RelationalModel: Introduction to relational model, concepts of domain, attribute, tuple, relation, importance of null values, constraints (Domain, Key constraints, integrity constraints) and their importance

BASICSQL: 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).

Objectives:

After studying this unit, you will be able to:

- Describe Relational model
- Describe SQL and data types
- Explain the basic structures of SQL queries
- Know how to create tables
- Realise aggregate functions and null values

Introduction

Relational Model was proposed by E.F Codd to model data in the form of relations or tables. After designing the conceptual model of database using ER diagram, we need to convert the conceptual model in the relational model which can be implemented using any RDBMS (Relational Data Base Management System) like SQL, MY SQL etc.

The relational model is very simple and elegant; a database is a collection of one or more relations, where each relation is a table with rows and columns.

This simple tabular representation enable even new user to understand the content of a database, and it permits the use of simple, high-level language to query the data.

Relational Model

Relational Model represents how data is stored in relational databases.

Relational database stores data in the form of relations (tables).

Consider a relation STUDENT with attributes ROLL_NO, NAME, ADDRESS, PHONE and AGE as shown in table.

ROLL_NO	NAME	ADDRESS	PHONE	AGE
1	Nishma	Hyderabad	9455123451	28
2	Sai	Guntur	9652431843	27
3	Swetha	Nellore	9156253131	26
4	Raji	Ongole	9215635311	25

Attribute: Attributes are the properties that define a relation. Ex:

ROLL_NO, NAME

Tuple: Each row in a relation is known as a tuple.

Ex:

1	Nishma	Hyderabad	9455123451	28
---	--------	-----------	------------	----

Degree: The number of attributes in the relation is known as degree.

Ex: The degree of the given STUDENT table is 5.

Column: Column represent the set of values for a particular attribute. The column ROLL_NO is extracted from the relation STUDENT.

Ex:

ROLL_NO
1

Null values: The value which is not known or unavailable is called NULL VALUE. It is represented by blank space.

Cardinality: The number of tuples present in the relation is called as its cardinality.

Ex: The Cardinality of the STUDENT table is 4.

ConceptOfDomain

The domain of a database is the set of all allowable values (or) attributes of the database.

Ex: Gender (Male, Female, Others).

Relation

- A relation is defined as a set of tuples and attributes.
- A relation consists of a relation schema and a relation instance.
- Relation schema: A relation schema represents the name of the relation with its attributes.

Ex: STUDENT (ROLL_NO, NAME, ADDRESS, PHONE and AGE) is a relation schema for STUDENT.

- Relation instance: These tuples of a relation at a particular instance of time are called Relation Instance.

An instance of „Employee„ relation

Emp_code	Emp_Name	Dept_Name
01234	John	HR
12567	Smith	Sales
21678	Sai	Production
12456	Jay	Design

Importance of Null values:

- SQL supports a special value known as NULL which is used to represent the values of attributes that may be unknown or not apply to a tuple.
- For example, the apartment_number attribute of an address applies only to the address that is in apartment buildings and not to other types of residences.
- It is important to understand that a NULL value is different from a Zero value.
- A NULL value is used to represent a missing value, but that is usually has one of the following interpretations:
 - Value unknown (Value exists but it is unknown)
 - Value not available (exists but it is purposely withheld)
 - Attribute not applicable (undefined for this tuple)
- It is often not possible to determine which of the meanings is intended.

Constraints

- On modeling the design of the relational database, we can put some rules (conditions) like what values are allowed to be inserted in the relation
- Constraints are the rules enforced on the data columns of a table. These are used to limit the type of data that can go in to a table
- This ensures the accuracy and reliability of the data in the database. Constraints could be either on a column level or on a table level.

DomainConstraintsInDBMS

- InDBMStableisviewedasacombinationofrowsandcolumns
- Forexample,ifyouarehavingacolumncalledmonthandyouwantonly(jan,feb, march.....) as values allowed to be entered for that particular column which is referred to as domain for that particular column

Definition: Domain constraint ensures two things it makes sure that the data value entered for that particularcolumnmatcheswiththedatatypesdefinedbythatcolumn

Itshowsthattheconstraints(NOTNULL/UNIQUE/PRIMARYKEY/FOREIGN KEY/CHECK/DEFAULT)

Domain Constraints

ROLL_NO	NAME	AGE
1	Arya	21
2	Bravo	19
3	John	24
4	Max	24

Domain

Age must be greater than 18 and should be an integer

↓
Check(age>=18)

Domainconstraint=datatypecheckforthecolumn+constraints.

Example;wewanttocreateatable"STUDENT"with"stu_id"fieldhavingavaluegreater than 100, can create a domain and table like this.

- Createdomainid_valueintconstraintid_testcheck(value>=100);
- CREATEtableSTUDENT(stu_ididvalueprimarykey,stu_namevarchar (30), stu_age int);

KeyconstraintsinDBMS:

- Constraints arenothing butthe rules that areto be followed while entering data into columnsof thedatabase table.
- Constraints ensure that the data entered by the user into columns must be within the criteria specified bythe condition.
- Wehave6typesofkeyconstraintsinDBMS
 1. NotNull
 2. Unique
 3. Default
 4. Check
 5. Primarykey
 6. Foreignkey

1. NotNull:

- Nullrepresentsarecordwheredatamaybemissingdataordataforthatrecordmaybeoptional.
- Oncenotnullisappliedtoaparticularcolumn,youcannotenternullvaluestothatcolumn.
- Anotnullconstraintcannotbeappliedatablelevel.

ROLL_NO	NAME	AGE	PHONE
1	Arya	21	7491901512
2	Bravo	19	8491901000
3	John	24	929108403
4	Max	24	7903084562

NOT NULL (does not allow
null value)

(NULL values are allowed)

Example:

Create table EMPLOYEE (id int Not null, name varchar Not null, Age int not null, address char (25), salary decimal (18,2), primary key(id));

- Intheaboveexamplewehaveappliednotnullonthreecolumnsid,nameandagewhich means whenever a record is entered using insert statement all three columns should contain a value other than null.
- We have two other columns address and salary, where not null is not applied which means that you can leave the row as empty.

2. Unique:

Sometimesweneedtomaintainonly.Uniquedatain thecolumnofadatabasetable,thisis possible by using a Unique constraint.

- Uniqueconstraintensures thatallvaluesinacolumnareUnique.

Example:

CreatetablePERSONS(idintunique,last_namevarchar(25)notnull, First name varchar (25), age int);

- Intheaboveexample,aswehaveuseduniqueconstraintonIDcolumnwearenotsupposed toenterthedata thatisalready present,simplynotwoIDvaluesaresame.

ROLL_NO	NAME	AGE	PHONE
1	Arya	21	7491901512
2	Bravo	19	8491901000
3	John	24	929108403
4	Max	24	7903084562

UNIQUE (All values are different)

(allow duplicate values)

3. Default:

Default in SQL is used to add default data to the columns.

- When a column is specified as default with same value then all the rows will use the same value i.e., each and every time while entering the data we need not enter that value.
- But default column value can be customised i.e., it can be overridden when inserting data for that row based on the requirement.

ID_NO	NAME	COMPANY
1	Arya	abc
2	Bravo	abc
3	John	abc
4	Max	abc

(Row with default values "abc")

Example:

Create table **EMPLOYEE** (id int Not null, last_name varchar (25) Not null, first_name varchar (25), Age int, city varchar (25) Default Hyderabad);

- As a result, whenever you insert a new row each time you need not enter a value for this default column that is entering a column value for a default column is optional.

4. Check:

- Check constraint ensures that the data entered by the user for that column is within the range of values or possible values specified.

Example: Create table **STUDENT** (id int, name varchar (25), age int, check (age >= 18));

ROLL_NO	NAME	AGE
1	Arya	21
2	Bravo	19
3	John	24
4	Max	24

Check(age >= 18)

- As we have used a check constraint as (age >= 18) which means value entered by user for this age column while inserting the data must be less than or equal to 18.

5. PrimaryKey:

- A primary key is a constraint in a table which uniquely identifies each row record in a database table by enabling one or more columns in the table as primary key.

ROLL_NO	NAME	AGE	GPA
1	Arya	21	4
2	Bravo	19	3
3	John	24	4.3
4	Max	24	1

Creating a primary key:

- A particular column is made as a primary key column by using the primary key keyword followed by the column name.

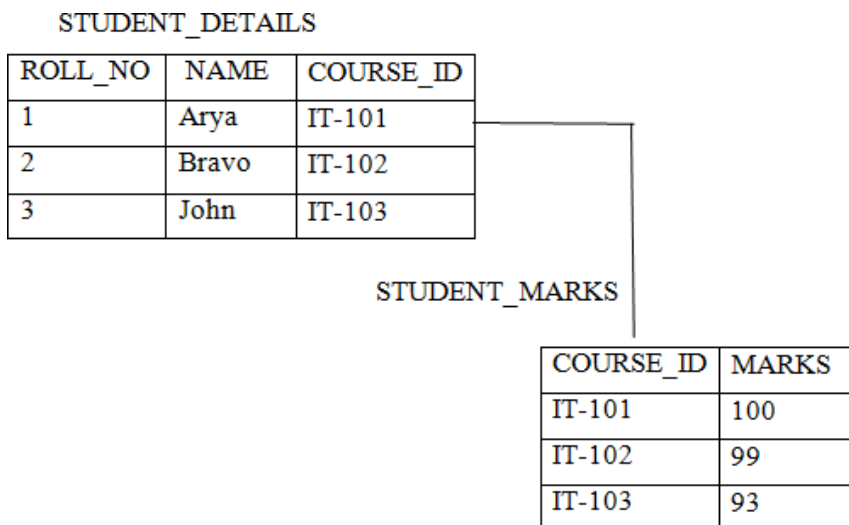
Example:

```
CREATE TABLE EMP (ID INT, name VARCHAR(20), age INT, course VARCHAR(10), PRIMARY KEY (ID));
```

- Here we have used the primary key on ID column then ID column must contain unique values i.e., one ID cannot be used for another student.

6. Foreign Key:

- The foreign key constraint is a column or list of columns which points to the primary key column of another table.
- The main purpose of the foreign key is only those values are allowed in the present table that will match to the primary key column of another table.



From the above two tables, COURSE_ID is a primary key of the table STUDENT_MARKS and also behaves as a foreign key as it is same in STUDENT_DETAILS and STUDENT_MARKS.

Example:

(ReferenceTable)

```
CreatetableCUSTOMER1 (idint,namevarchar(25),coursevarchar(10), primary
key (ID));
```

(Childtable)

```
CreatetableCUSTOMER2 (idint,marksint,referencescustomer1 (ID));
```

IntegrityConstraints inDBMS:

- Therearetwotypesofintegrityconstraints
 1. EntityIntegrityConstraints
 2. ReferentialIntegrityConstraints

EntityIntegrityconstraints:

- These constraints are used to ensure the uniqueness of each record or row in the data table.
- EntityIntegrityconstraintssaysthatnoprimarykeycantakeNULLVALUE,since usingprimarykeyweidentifyeachtupleuniquelyinarelation.

Example:

EID	NAME	PHONE
01	Sony	7002494274
02	Pinky	6026526747
NULL	Lalli	9234567892

Explanation:

- In the above relation, EID is made primary key, and the primary key can'ttakeNULLvaluesbutinthe3rdtuple,theprimarykeyisNULL,soitis violating Entity integrity constraints.

ReferentialIntegrityconstraints:

- Thereferentialintegrityconstraintisspecifiedbetweentworelationsortablesandused to maintain the consistency among the tuples in two relations.
- This constraint is enforced through foreign key, when an attribute in the foreign key of relationR1havethesamedomainasprimarykeyofrelationR2,thentheforeignkeyof R1 is said to reference or refer to the primary key of relation R2.
- The values of the foreign key in a tuple of relation R1 can either take the values of the primarykeyforsometupleinRelationR2,orcantakeNULLvalues,butcan'tbeempty.

EID	NAME	DNO
01	DIVIN	12
02	DYNO	22
03	VIVI	14

Foreign key

Primary key

DNO	PLACE
12	Jaipur
13	Mumbai
14	Delhi

Explanation:

- In the above, DNO of the first relation is the foreign key and DNO in the second relation is the primary key
- DNO=22 in the foreign key of the first relation is not available in the second relation so, since DNO=22 is not defined in the primary key of the second relation therefore Referential integrity constraints is violated here.

BasicSQL(introduction)

- SQL stands for Structure Query Language it is used for storing and managing data in relational database management system.
- It is standard language for relational database system. It enables a user to create, read, update and delete relational databases and tables.
- All the RDBMS like MySQL, Oracle, Microsoft Access and SQL Server use SQL as their standard database language.
- SQL allows users to query the database in a number of ways using statements like common English.

Rules: SQL follows following rules

- SQL is not a case sensitive. Generally, keywords are represented in UPPERCASE.
- Using the SQL statements, you can perform most of the actions in a database.
- Statements of SQL are dependent on text lines. We can use a single SQL statement on one or multiple text line.

SQLProcess:

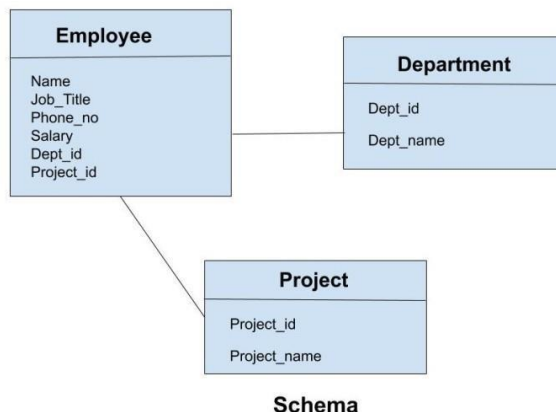
- When an SQL command is executing for any RDBMS, then the system figure out the best way to carry out the request and the sql engine determines that how to interrupt the task.
- In the process, various components are included. These components can be optimization engine, query engine, query dispatcher etc.,
- All the non-sql queries are handled by the classic query engine, but sql query engine won't handle logical files.

CharacteristicsofSQL:

- SQL is easy to learn.
- SQL is used to access data from relational database management system.
- SQL is used to describe the data.
- SQL is used to create and drop the database and table.
- SQL allows users to set permissions on tables, procedures and views.

Simple databaseSchema:

- A database schema is a structure that represents the logical storage of the data in the database.
- It represents the organization of data and provides information about the relationships between the tables in a given database.
- A database schema is the logical representation of a database, which shows how the data is stored logically in the entire database.
- It contains list of attributes and instruction that inform the database engine that how the data is organized and how the elements are related to each other.
- A database schema contains schema objects that may include tables, fields, packages, views, relationship, primary key, foreign key.
- In actual, the data is physically stored in files that may be in unstructured form, but to retrieve it and use it, we need to keep them in a structured manner. To do this a database schema is used. It provides knowledge about how the data is organized in a database and how it is associated with other data.
- A database schema object includes the following:
 - Consistent formatting for all data entries.
 - Database objects and unique keys for all data entries.
 - Tables with multiple columns, and each column contains its names and data types.
 - The given diagram is an example of a database schema that contains three tables, their data types. This also represents the relationships between the tables and primary keys as well as foreign keys.



Schema

SQLCommands:

SQLcommandsarecategorizedinto threetypes.

1. **Data Definition Language (DDL):** used to create (define) a table.
2. **Data Manipulation Language (DML):** used to update, store and retrieve data from tables.
3. **Data Control Language (DCL):** used to control the access of database created using DDL and DML.

DDL - Data Definition Language

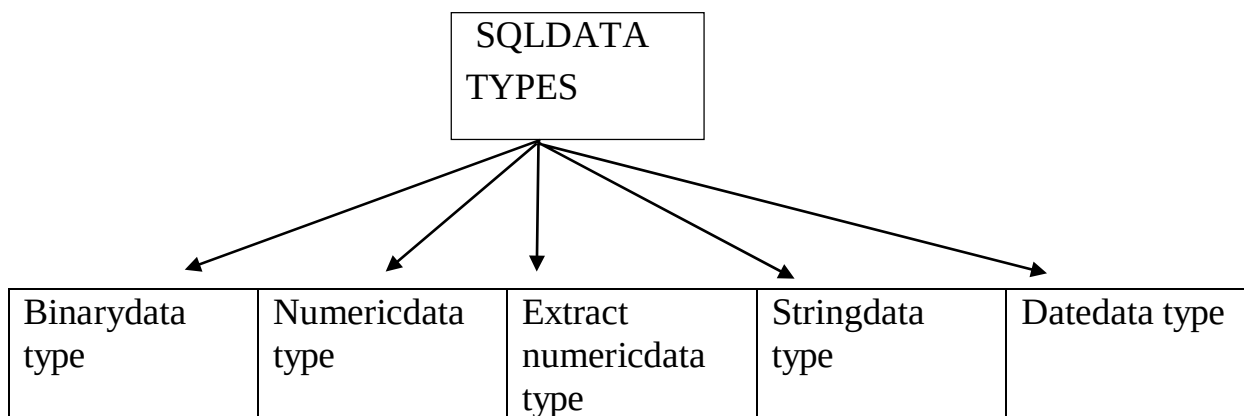
Command	Description
CREATE	Creates a new table, a view of a table, or other object in the database.
ALTER	Modifies an existing database object, such as a table.
DROP	Deletes an entire table, a view of a table or other objects in the database.

SQLDATATYPES:

SQL datatype is used to define the values that a column can contain

Every column is required to have a name and data type in the database

table. DATATYPES OF SQL:

**1. BINARYDATATYPES:**

There are three types of binary data types which are given below

DATATYPE	DESCRIPTION
Binary	It has a maximum length of 800 bytes. It contains a fixed-length binary data
Varbinary	It has a maximum length of 800 bytes. It contains a variable - length binary data
Image	It has a maximum length of 2,147,483,647 bytes. It contains a variable-length binary data

2. NUMERIC DATATYPE:

DATATYPE	FROM	TO	DESCRIPTION
Float	-1.79E+308	1.79E+308	It is used to specify a floating-point value. Ex: 6.2, 2.9 etc
Real	-3.40E+38	3.40E+38	It specifies a single precision floating point number.

3. EXACT NUMERIC DATATYPE:

DATATYPE	DESCRIPTION
Int	It is used to specify an integer value
Smallint	It is used to specify a small integer value
Bit	It has the number of bits to store
Decimal	It specifies a numeric value that can have a decimal number
Numeric	It is used to specify a numeric value

4. DATE AND TIME DATATYPES:

DATATYPE	DESCRIPTION
Date	It is used to store the year, month, and days value
Time	It is used to store the hour, minute, and seconds value
Timestamp	It stores the year, month, hour, minute, and the seconds value

5. STRING DATATYPE:

DATATYPE	DESCRIPTION
Char	It has a maximum length of 8000 characters. It contains fixed-length non-Unicode characters.
Varchar	It has a maximum length of 8000 characters. It contains variable-length non-Unicode characters.
Text	It has a maximum length of 2,147,483,647 characters. It contains variable-length non-Unicode characters.

TABLE DEFINITIONS: (CREATE, ALTER)

SQLTABLE: SQLtable isacollectionofdata whichisorganizedintermsof rows and columns.

- InDBMS,thetableisknownasrelationandrowasatuple
- Let"sseeanexampleofthe"EMPLOYEE"table

EMP_ID	EMP_NAME	CITY	PHONE_ID
1	Kristen	Washington	7289201223
2	Anna	Franklin	9378282882
3	Jackson	California	9264783838
4	Daniel	Hawaii	9638482678

- In the above table, "EMPLOYEE" is the table name, "EMP_ID", "EMP_NAME", "CITY", "PHONE-NO" are the column names.
- Thecombinationofdataofmultiplecolumnsformsarow

EG:1,"Kristen","Washington"and"7289201223"arethedataofonerow

OPERATIONS ON TABLE:

1. Createtable
2. Altertable
3. Droptable

1. **Create table:** SQL create table is used to create a table in the database. To define the table, you should define the name of the table and also define its column and column"s datatype.

SYNTAX:

```

Createtabletable_name ("column1" "datatype",
                        "column2" "datatype",
                        "column3" "datatype",
                        ...
                        "columnN" "datatype");

```

EXAMPLE:

```

SQL >create table employee (emp_id int, emp_name varchar (25), phone_no int,
address char (30));

```

- Ifyoucreatethetablesuccessfully,youcanverifythetablebylookingatthemessagebythe sql server. else you can use DESC command as follows

SQL>DESCemployee;

FIELD	TYPE	NULL	DEFAULT	EXTRA
Emp_id	Int(11)	No	NULL	
Emp_name	Varchar(25)	No	NULL	
Phone_no	No	Int(11)	NULL	
address	yes		NULL	Char(30)

2. ALTER TABLE:

- The alter table command adds, delete or modifies columns in a table
- The alter table command also adds and deletes various constraints in a table
- The following SQL adds an "EMAIL" column to the "EMPLOYEE" table

SYNTAX:

ALTER table *table_name* **add** column *column1* *data type*;

EXAMPLE:

ALTER table *employee* **add** email *varchar* (255) ;

SQL>DESC employee;

FIELD	TYPE	NULL	DEFAULT	EXTRA
Emp_id	Int(11)	No	NULL	
Emp_name	Varchar(25)	No	NULL	
Phone_no	No	Int(11)	NULL	
Address	Yes		NULL	Char(30)
Email	Varchar(255)		NULL	

3. DROP TABLE:

- The drop table command deletes a table in the database
- The following example SQL deletes the table "EMPLOYEE"

SYNTAX:

DROP table *table_name*;

EXAMPLE:

DROP table *employee*;

- Dropping a table results in loss of all information stored in the table.

Different DML Operations (insert, delete, update):

- DML-Data Manipulation Language.
- Data Manipulation Commands are used to manipulate data to the database.
- Some of the data manipulation commands are
 1. Insert
 2. Update
 3. Delete

1. Insert:

SQL insert statement is a SQL query. It is used to insert a single or multiple records in a table.

Syntax:

Insertintotablenamevalues (value1,value2,value 3) ;

Let'stakeanexampleoftablewhichhas3recordswithinit.

- insertintostudentvalues („alekhya",501,"hyderabad");
- insertintostudentvalues („deepti",502,"guntur");
- insertintostudentvalues („ramya",503,"nellore");

The following table will be as follows:

NAME	ID	CITY
Alekhya	501	Hyderabad
Deepti	502	Guntur
Ramya	503	Nellore

2. Update:

- TheSQLCommandsupdateareusedtomodifythedata thatisalreadyinthedatabase.
- SQLUpdatestatementisusedtochangethedataofrecords heldbytableswhichrowsisto be update, it is decided by condition to specify condition, we use "WHERE" clause.
 - Theupdatestatementcanbewritteninfollowingform:

Syntax:

Updatetable_name setcolumn_name=expressionwherecondition;

Example:

Let'stakeanexample:herewearegoingtoupdateanentryinthetable.

Updatestudents setname="rasi"whereid=503;

Afterupdatethetableisasfollows:

NAME	ID	CITY
Alekhya	501	Hyderabad
Deepti	502	Guntur
Rasi	503	Nellore

3. Delete:

- TheSQLdeletestatementisusedtodeleterowsfromatable.
- Generally,deletestatementremovesoneormore recordsfromatable.

Syntax:

deletefromtable_name [wherecondition] ;

Example:

Letustakeatablenamed"student"table

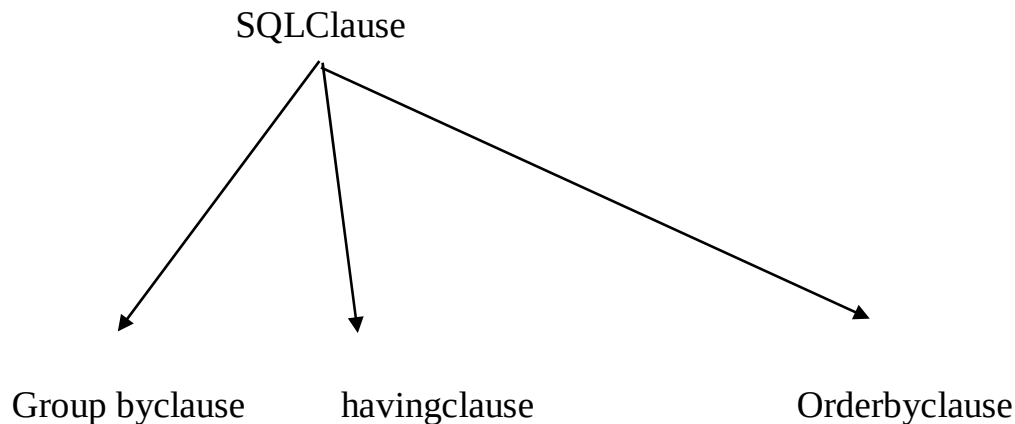
Delete from students where id=501;

Resulting table after the query:

NAME	ID	CITY
Deepti	502	Guntur
Rasi	503	Nellore

Basic SQL querying (select and project) using where clause:

- The following are the various SQL clauses:



1. Groupby:

- SQL group by statement is used to arrange identical data into groups.
- The group by statement is used with the SQL select statement.
- The group by statement follows the WHERE clause in a SELECT statement and precedes the ORDER BY clause.

Syntax:

Select column from table_name where column group by column, order by column;

Sample table: product

PRODUCT	COMPANY	QTY	RATE	COST
Item1	Com1	2	10	20
Item2	Com2	3	25	75
Item3	Com1	2	30	60
Item4	Com3	5	10	50
Item5	Com2	2	20	40

Example:

- **Select company count(*) from product group by company;**

Output:

Com1	2
Com2	3
Com3	5

2. Havingclause:

- Havingclauseisusedtospecifyasearchconditionforagrouporanaggregate.

Havingclause isused ina group byclause, if youare notusinggroup byclause thenyoucan use having function like a where clause.

Syntax:

Selectcolumn1,column2fromtable_name

Whereconditions

Groupbycolumn1,column2

Having conditions

Orderbycolumn1,column2;

Example:

- selectcompanycount(*)fromproduct

Group by company

Havingcount(*)>2;

Output:

Com3	5
Com2	2

3. Orderbyclause:

Theorderbyclausesortstheresult_setinascendingordescendingorder.

Syntax:

Selectcolumn1,column2,fromtable_name

Wherecondition

Orderbycolumn1,column2...asc;

Sampletable:

Takeastudenttable

Example:

Select*fromstudentorderby name;

Output:

NAME	ID	CITY
Alekhya	501	Hyderabad
Deepti	502	Guntur
Rasi	503	Nellore

SQLWhereclause:

- AwhereclauseinSQLisadatamanipulationlanguagestatement.
- WhereclausesarenotmandatoryclausesofSQLDMLstatementsbutitcanbeusedtolimit the number of rows affected by a SQL DML statement or returned by query.
- Actually,itfollowstherecords.itreturnsonlythosequerieswhichthespecificconditions.

Syntax:

Selectcolumn1,column2,.....columnfromtable_namewhere[condition];

- Whereclauseusessameconditionalselection.

=	EqualTo
>	Greaterthan
<	Lessthan
>=	Greaterthanorequalto
<=	Lessthanorequalto
<>	Notequalto

Arithmeticandlogicaloperations:**SQLoperators:**

- SQLstatementsgenerallycontainsomereservedwordsorcharactersthatareusedto perform operations such as arithmetic and logical operations etc. Their reserved words are known as operators.

SQLarithmeticoperator:

- Wecanusevariousarithmeticoperatorsonthedatastoredintables.
- Arithmeticoperatorsare:

+	Addition
-	Subtraction
/	Division
*	Multiplication

%	modulus
---	---------

1. Addition(+):

It is used to perform addition operation on data items.

Sample table:

EMP_ID	EMP_NAME	SALARY
1	Alex	25000
2	John	55000
3	Daniel	52000
4	Sam	12312

- `select emp_id, emp_name, salary, salary+100 as "salary+100" from addition;`

Output:

EMP_ID	EMP_NAME	SALARY	SALARY+100
1	Alex	25000	25100
2	John	55000	55100
3	Daniel	52000	52100
4	Sam	12312	12412

- Here we have done addition of 100 to each emp's salary.

2. Subtraction(-):

- It is used to perform subtraction on the data items.

Example:

`Select emp_id, emp_name, salary, salary-100 as "salary-100" from subtraction;`

EMP_ID	EMP_NAME	SALARY	SALARY-100
1	Alex	25000	24900
2	John	55000	54900
3	Daniel	52000	51900
4	Sam	90000	89900

Here we have done subtraction of 100 for each emp's salary.

3. Division(/):

- The division function is used to integer division (x is divided by y). an integer value is returned.

Example:

- `Select emp_id, emp_name, salary, salary/100 as "salary/100" from division;`

EMP_ID	EMP_NAME	SALARY	Salary/100
1	Alex	25000	250
2	John	55000	550
3	Daniel	52000	520
4	Sam	90000	900

4. **Multiplication(*):**

- It is used to perform multiplication of data items.
- **Select emp_id, emp_name, salary, salary*100 as "salary*100" from multiplication;**

EMP_ID	EMP_NAME	SALARY	SALARY*100
1	Alex	25000	2,500,000
2	John	55000	5,500,000
3	Daniel	52000	5,200,000
4	Sam	90000	9,000,000

- Here we have done multiplication of 100 to each emp's salary.

5. **Modulus(%):**

- It is used to get remainder when one data is divided by another.
- **Select emp_id, emp_name, salary, salary%25000 as "salary%25000" from modulus;**

Output:

EMP_ID	EMP_NAME	SALARY	SALARY%25000
1	Alex	25000	0
2	John	55000	5000
3	Daniel	52000	2000
4	Sam	90000	15000

- Here we have done modulus operation to each emp's salary.

Logical operations:

- Logical operations allow you to test for the truth of a condition.
- The following table illustrates the SQL logical operator.

OPERATOR	MEANING
ALL	Return true if all comparisons are true
AND	Return true if both expressions are true
ANY	Return true if any one of the comparisons is true
BETWEEN	Return true if the operand is within a range
IN	Return true if the operand is equal to one of the values in a list
EXISTS	Return true if the subquery contains any rows

1. **AND:**

The AND operator allows you to construct multiple condition in the WHERE clause of an SQL statement such as select.

- The following example finds all employees where salaries are greater than the 5000 and less than 7000.
- **Select first_name, last_name, salary from employees where salary > 5000 AND salary < 7000 order by salary;**

Output:

FIRST_NAME	LAST_NAME	SALARY
John	Wesley	6000
Eden	Daniel	6000
Luis	Popp	6900
Shanta	Suji	6500

2. ALL:

The ALL operator compares a value to all values in another value set.

- The following example finds all employees whose salaries are greater than all salaries of employees.

EX:

```
select first_name, last_name, salary from employees where salary >= ALL
(select salary from employees where department_id=8) order by salary DESC;
```

Output:

FIRST_NAME	LAST_NAME	SALARY
Steven	King	24000
John	Russel	17000
Neena	Kochhar	14000

3. ANY:

The ANY operator compares a value to any value in a set ascending to condition.

The following example statement finds all employees whose salaries are greater than the average salary of every department.

EX:

```
select first_name, last_name, salary from employees where salary > ANY
(select avg(salary) from employees group by department_id) order by
first_name, last_name;
```

Output:

FIRST_NAME	LAST_NAME	SALARY
Alexander	Hunold	9000.00
Charles	Johnson	6200.00
David	Austin	4800.00
Eden	Flip	9000.00

4. Between:

- The between operator searches for values that are within a set of values.
- For example, the following statement finds all employees where salaries are between 9000 and 12000.

EX:

```
select first_name, last_name, salary from employees where salary between 9000
AND 12000 order by salary;
```

Output:

FIRST_NAME	LAST_NAME	SALARY
Alexander	Hunold	9000.00
Den	Richards	10000.00
Nancy	Prince	12000.00

5. IN:

- TheINoperatorcomparesavaluetolistofspecifiedvalues.TheINoperatorreturntrueif compared value matches at least one value in the list.
- The following statement finds all employees who work in department_id 8 or 9.EX:

```
selectfirst_name,last_name,department_idfromemployeeswhere
department_id IN (8,9) order by department_id;
```

Output:

FIRST_NAME	LAST_NAME	DEPARTMENT_ID
John	Russel	8
Jack	Livingstone	8
Steven	King	9
Neena	Kochhar	9

6. Exists:

- TheEXISTSoperator tests if a subquery contains any rows.
- For example, the following statement finds all employees who have dependents.
- ```
selectfirst_name,last_namefromemployeeswhereEXISTS(selectl from
dependent d where d.employee_id=e.employee_id);
```

| FIRST_NAME | LAST_NAME |
|------------|-----------|
| Steven     | King      |
| Neena      | Kochhar   |
| Alexander  | Hunold    |

**SQLFUNCTIONS(Date&Time,Numeric,Aggregate,Stringconversions):****DATE&TIMEFUNCTIONS:**

| Function          | Example                                                                             | Result                                 | Description                                                                                        |
|-------------------|-------------------------------------------------------------------------------------|----------------------------------------|----------------------------------------------------------------------------------------------------|
| ADD_MONTHS        | ADD_MONTHS( DATE '2016-02-29', 1 )                                                  | 31-MAR-16                              | Add a number of months (n) to a date and return the same day which is n of months away.            |
| CURRENT_DATE      | SELECT CURRENT_DATE FROM dual                                                       | 06-AUG-2017 19:43:44                   | Return the current date and time in the session time zone                                          |
| CURRENT_TIMESTAMP | SELECT CURRENT_TIMESTAMP FROM dual                                                  | 06-AUG-17 08.26.52.742000000 PM -07:00 | Return the current date and time with time zone in the session time zone                           |
| DBTIMEZONE        | SELECT DBTIMEZONE FROM dual;                                                        | -07:00                                 | Get the current database time zone                                                                 |
| EXTRACT           | EXTRACT(YEAR FROM SYSDATE)                                                          | 2017                                   | Extract a value of a date time field e.g., YEAR, MONTH, DAY, ... from a date time value.           |
| FROM_TZ           | FROM_TZ(TIMESTAMP '2017-08-08 08:09:10', '-09:00')                                  | 08-AUG-17 08.09.10.000000000 AM -07:00 | Convert a timestamp and a time zone to a TIMESTAMP WITH TIME ZONE value                            |
| LAST_DAY          | LAST_DAY( DATE '2016-02-01' )                                                       | 29-FEB-16                              | Gets the last day of the month of a specified date.                                                |
| LOCALTIMESTAMP    | SELECT LOCALTIMESTAMP FROM dual                                                     | 06-AUG-17 08.26.52.742000000 PM        | Return a TIMESTAMP value that represents the current date and time in the session time zone.       |
| MONTHS_BETWEEN    | MONTHS_BETWEEN( DATE '2017-07-01', DATE '2017-01-01' )                              | 6                                      | Return the number of months between two dates.                                                     |
| NEW_TIME          | NEW_TIME( TO_DATE( '08-07-2017 01:30:45', 'MM-DD-YYYY HH24:MI:SS' ), 'AST', 'PST' ) | 06-AUG-2017 21:30:45                   | Convert a date in one time zone to another                                                         |
| NEXT_DAY          | NEXT_DAY( DATE '2000-01-01', 'SUNDAY' )                                             | 02-JAN-00                              | Get the first weekday that is later than a specified date.                                         |
| ROUND             | ROUND( DATE '2017-07-16', 'MM' )                                                    | 01-AUG-17                              | Return a date rounded to a specific unit of measure.                                               |
| SESSIONTIMEZONE   | SELECT SESSIONTIMEZONE FROM dual;                                                   | -07:00                                 | Get the session time zone                                                                          |
| SYSDATE           | SYSDATE                                                                             | 01-AUG-17                              | Return the current system date and time of the operating system where the Oracle Database resides. |

|              |                                            |                                               |                                                                                                                           |
|--------------|--------------------------------------------|-----------------------------------------------|---------------------------------------------------------------------------------------------------------------------------|
| SYSTIMESTAMP | SELECT SYSTIMESTAMP<br>FROM dual;          | 01-AUG-17<br>01.33.57.929000000 PM -<br>07:00 | Return the system date and<br>time that includes fractional<br>seconds and time zone.                                     |
| TO_CHAR      | TO_CHAR( DATE'2017-01-<br>01', 'DL' )      | Sunday, January 01, 2017                      | Convert a <a href="#">DATE</a> or<br>an <a href="#">INTERVAL</a> value to a<br>character string in a specified<br>format. |
| TO_DATE      | TO_DATE( '01 Jan 2017', 'DD<br>MON YYYY' ) | 01-JAN-17                                     | Convert a date which is in the<br>character string to<br>a DATE value.                                                    |
| TRUNC        | TRUNC( DATE '2017-07-16',<br>'MM' )        | 01-JUL-17                                     | Return a date truncated to a<br>specific unit of measure.                                                                 |
| TZ_OFFSET    | TZ_OFFSET( 'Europe/London'<br>)            | +01:00                                        | Get time zone offset of a time<br>zone name from UTC                                                                      |

### Some important date and time functions are below:

**Sysdate:** It generates the system date.

**Ex:** `Select sysdate from dual;`

Output: 05-DEC-2021.

**ADD\_MONTHS:** This function returns a date after adding data with specified no of months. **EX:** `Select`

`ADD_MONTHS (, '2017-02-29', 1) from dual;`

Output: 31-MAR-17.

`Select add_months(sysdate, 3) from dual;`

Output: 05-MAR-22.

**CURRENT\_DATE:** This function displays the current date.

**Ex:** `Select CURRENT_DATE from dual;`

Output: 05-DEC-2021.

**NEXT\_DAY:** This function represents both day and date and returns the day of the next given day.

**EX:** `Select NEXT_DAY(SYSDATE, 'MONDAY') from dual;`

Output: 07-DEC-21.

**LAST\_DAY:** This function returns a day corresponding to the last day of months.

**EX:** `Select LAST_DAY(sysdate) from dual;`

Output: 31-DEC-21.

**MONTHS\_BETWEEN:** It is used to find the number of months between two given dates.

EX: `Select MONTHS_BETWEEN ("16-APRIL-2021", "16-AUGUST-2021") from dual;`

**Output:-4.**

**ROUND:** It gives the nearest value or round off value for the argument passed. (or) It returns a date rounded to a specific unit of measure.

EX: `Select ROUND ("26-NOV-21", "YYYY") from dual;`

**Output:01-JAN-22.**

**TRUNC:** This function returns the date with the time (co-efficient) portion of the date truncated to the unit specified.

EX: `Select TRUNC (sysdate, "MM") from dual;`

**Output:01-DEC-21.**

**TO\_DATE:** This function converts a date which is in the character string to a date value.

EX: `Select TO_DATE ("01-jan-2017", "DDMMYYYY") from dual;`

**Output:01-JAN-17.**

**TO\_CHAR:** This function converts DATE or an INTERVAL value to a character string in a specified format.

EX: `Select TO_CHAR (sysdate, "DDMMYYYY") from dual;`

**Output:0512 2021.**

**LEAST:** This function displays the oldest date present in the argument list.

EX: `Select LEAST ("01-march-2021", "16-feb-2021", "28-dec-2021") from dual;`

**Output:01-MAR-21.**

**GREATEST:** This function displays the latest date present in the argument list.

EX: `Select GREATEST ("01-march-2021", "16-feb-2021", "28-dec-2021") from dual;`

**Output:28-DEC-21.**

### **Aggregate Functions:**

Aggregate Function takes a collection of values as input and returns a single value.

1. Count()

2. Sum()

3. Avg()

4. Max()

5. Min()

1. **Count()**: This function returns number of rows returned by a query.

**Syntax:** Select count (column\_name)

From table\_name

Where condition);

**Example:** Select count (distinct manager\_id) from employees;

2. **Sum()**: It will add/sum all the column values in the query.

**Syntax:** Select sum (column\_name)

From table\_name

Where condition);

**Example:** Select sum (salaries) from employees;

3. **Avg()**: Avg function used to calculate average values of the set of rows.

**Syntax:** Select avg (column\_name)

From table\_name

Where condition);

**Example:** Select avg (salary) from employees;

4. **Max()**: This function is used to find maximum value from the set of values.

**Syntax:** Select max (column\_name)

From table\_name

Where condition);

**Example:** Select max (salary) from employees;

5. **Min()**: This function is used to find minimum value from the set of values.

**Syntax:** Select min (column\_name)

From table\_name

Where condition);

**Example:** Select min (salary) from employees;

### **SQLNUMERICFUNCTIONS:**

| Function           | Input Argument                                                      | Value Returned                                     |
|--------------------|---------------------------------------------------------------------|----------------------------------------------------|
| ABS ( m )          | m = value                                                           | Absolute value of m                                |
| MOD ( m, n )       | m = value, n = divisor                                              | Remainder of m divided by n                        |
| POWER ( m, n )     | m = value, n = exponent                                             | m raised to the nth power                          |
| ROUND ( m [, n ] ) | m = value, n = number of decimal places, default 0                  | m rounded to the nth decimal place                 |
| TRUNC ( m [, n ] ) | m = value, n = number of decimal places, default 0                  | m truncated to the nth decimal place               |
| SIN ( n )          | n = angle expressed in radians                                      | sine (n)                                           |
| COS ( n )          | n = angle expressed in radians                                      | cosine (n)                                         |
| TAN ( n )          | n = angle expressed in radians                                      | tan (n)                                            |
| ASIN ( n )         | n is in the range -1 to +1                                          | arc sine of n in the range $-\pi/2$ to $+\pi/2$    |
| ACOS ( n )         | n is in the range -1 to +1                                          | arc cosine of n in the range 0 to $\pi$            |
| ATAN ( n )         | n is unbounded                                                      | arc tangent of n in the range $-\pi/2$ to $+\pi/2$ |
| SINH ( n )         | n = value                                                           | hyperbolic sine of n                               |
| COSH ( n )         | n = value                                                           | hyperbolic cosine of n                             |
| TANH ( n )         | n = value                                                           | hyperbolic tangent of n                            |
| SQRT ( n )         | n = value                                                           | positive square root of n                          |
| EXP ( n )          | n = value                                                           | e raised to the power n                            |
| LN ( n )           | n > 0                                                               | natural logarithm of n                             |
| LOG ( n2, n1 )     | base n2 any positive value other than 0 or 1, n1 any positive value | logarithm of n1, base n2                           |
| CEIL ( n )         | n = value                                                           | smallest integer greater than or equal to n        |
| FLOOR ( n )        | n = value                                                           | greatest integer smaller than or equal to n        |
| SIGN ( n )         | n = value                                                           | -1 if n < 0, 0 if n = 0, and 1 if n > 0            |

Numeric functions are used to perform operations on numbers and return

numbers. Following are some of the Numeric functions

1. **ABS ()**: It returns the absolute value of a number.

EX: **select ABS (-243.5) from dual;**

OUTPUT: 243.5

2. **ACOS ()**: It returns the cosine of a number.

EX: **select ACOS (0.25) from dual;**

OUTPUT: 1.318116071652818

3. **ASIN ()**: It returns the arc sine of a number.

EX: **select ASIN (0.25) from dual;**

OUTPUT: 0.253680255142

4. **CEIL ()**: It returns the smallest integer value that is a greater than or equal to a number. EX: **select CEIL (25.77) from dual;**

OUTPUT: 26

5. **FLOOR ()**: It returns the largest integer value that is a less than or equal to a number.EX:**select FLOOR (25.75) from dual;**  
OUTPUT:25
6. **TRUNCATE()**:ThisdoesnotworkforSQLserver.Itreturnsthetruncatedto2placesrightofthe decimal point.  
EX:**selectTRUNCATE (7.53635,2)from dual;**  
OUTPUT:7.53
7. **MOD ()**: It returns the remainder when two numbers are divided.EX: **select MOD (55,2) from dual;**  
OUTPUT:1.
8. **ROUND()**:Thisfunctionroundsthegivenvaluetogivennumberofdigitsofprecision. EX: **select ROUND (14.5262,2) from dual;**  
OUTPUT:14.53.
9. **POWER()**:Thisfunctiongivesthevalueofmraisedtothepowerofn. EX: **selectPOWER(4,9)fromdual;**  
OUTPUT:262144.
10. **SQRT()**:Thisfunctiongivesthesquarerootofthegivenvaluen. EX: **SelectSQRT (576) fromdual;**  
OUTPUT:24.
11. **LEAST ()**: This function returns least integer from given set of integers.  
EX: **select LEAST (1,9,2,4,6,8,22) from dual;**  
OUTPUT:1.
12. **GREATEST ()**: This function returns greatest integer from given set of integers.EX: **select GREATEST (1,9,2,4,6,8,22) from dual;**  
OUTPUT:22

### **STRINGCONVERSIONFUNCTIONSOFSQL:**

| Function              | Input Argument                                            | Value Returned                                                                                 |
|-----------------------|-----------------------------------------------------------|------------------------------------------------------------------------------------------------|
| INITCAP ( s )         | s = character string                                      | First letter of each word is changed to uppercase and all other letters are in lower case.     |
| LOWER ( s )           | s = character string                                      | All letters are changed to lowercase.                                                          |
| UPPER ( s )           | s = character string                                      | All letters are changed to uppercase.                                                          |
| CONCAT ( s1, s2 )     | s1 and s2 are character strings                           | Concatenation of s1 and s2. Equivalent to s1    s2                                             |
| LPAD ( s1, n [, s2] ) | s1 and s2 are character strings and n is an integer value | Returns s1 right justified and padded left with n characters from s2; s2 defaults to space.    |
| RPAD ( s1, n [, s2] ) | s1 and s2 are character strings and n is an integer value | Returns s1 left justified and padded right with n characters from s2; s2 defaults to space.    |
| LTRIM ( s [, set ] )  | s is a character string and set is a set of characters    | Returns s with characters removed up to the first character not in set; defaults to space      |
| RTRIM ( s [, set ] )  | s is a character string and set is a set of characters    | Returns s with final characters removed after the last character not in set; defaults to space |

|                                              |                                                                                           |                                                                                                                |
|----------------------------------------------|-------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------|
| REPLACE ( s,<br>search_s [,<br>replace_s ] ) | s = character string, search_s = target<br>string, replace_s = replacement string         | Returns s with every occurrence of search_s in s replaced by<br>replace_s; default removes search_s            |
| SUBSTR ( s, m [, n ] )                       | s = character string, m = beginning<br>position, n = number of characters                 | Returns a substring from s, beginning in position m and n<br>characters long; default returns to end of s.     |
| LENGTH ( s )                                 | s = character string                                                                      | Returns the number of characters in s.                                                                         |
| INSTR ( s1, s2 [, m [,<br>n ] ] )            | s1 and s2 are character strings, m =<br>beginning position, n = occurrence of<br>s2 in s1 | Returns the position of the nth occurrence of s2 in s1,<br>beginning at position m, both m and n default to 1. |

**String Functions are used to perform an operation on an input string and return the output string. Following are the string functions**

**1. CONCAT():** This function is used to add two words (or) strings.

EX: `select „database“ || „ „ || „management system“ from dual;`

OUTPUT: „database management system“

**2. INSTR():** This function is used to find the occurrence of an

alphabet. EX: `instr („database system“, „a“) from dual;`

OUTPUT: 2 (the first occurrence of „a“)

**3. LOWER():** This function is used to convert the given string into

lowercase. EX: `select lower („DATABASE“) from dual;`

OUTPUT: database

**4. UPPER():** This function is used to convert the lowercase string into uppercase.

EX: `select upper („database „) from dual;`

OUTPUT: DATABASE

**5. LPAD():** This function is used to make the given string of the given size by adding the given symbol. EX: `> lpad`

`( „system „, 8, „0“) from dual;`

OUTPUT: 00system

**6. RPAD():** This function is used to make the given string as long as the given size by adding the given symbol on the right.

EX: `rpadd („system“, 8, „0“) from dual;`

OUTPUT: system00

**7. LTRIM():** This function is used to cut the given substring from the original string. EX:

`ltrim („database „, „data „) from dual;`

OUTPUT:base

**8. RTRIM ():** This function is used to cut the given substring from the original

string. EX: `rtrim („database „, „base „) from dual;`

OUTPUT:data.

**9. INITCAP():** This function returns the string with first letter of each word starts with uppercase.

EX: `Select INITCAP („database management system“) from dual;`

OUTPUT: DataBase Management System.

**10. LENGTH():** This function returns the length of the given string.

EX: `select LENGTH („SQL LANGUAGE“) from dual;`

OUTPUT:11.

**11. SUBSTR ():** This function returns a portion of a string beginning at the character position. EX:

`select SUBSTR („MY WORLD IS AMAZING“,12,3) from dual;`

OUTPUT:AM.

**12. TRANSLATE ():** This function returns a string after replacing some set of characters into another set. EX:

`select TRANSLATE („Delhi is the capital of India“, „i“, „a“) from dual;`

OUTPUT: Delhaas the capatal andaa.

## Review Questions:

1. Why NULL values are needed in databases?
2. Discuss GROUP BY and HAVING clauses with an example. And also give the constraints related to their usage.
3. Consider the SAILOR DATABASE

Sailors (Sid: string, sname: string, rating: integer, age: real)

Boats (bid: integer, bname: string, colour: string)

Reserves (Sid: integer, bid: integer, day: date)

Based on the above schema, write the corresponding SQL queries for the following? Find the colours of boats reserved by Lubber.

Find the names of sailors who have reserved at least one boat. Find the names of sailors who have reserved a red or green boat.

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

Findnamesofsailorswhohavereservedallboats.

4. Writeabouttheusabilityof„groupby“and„having“clausesInSQL.
5. How would you use the operators IN, EXISTS, UNIQUE,ANY and ALL in writing nestedqueries? Why are they useful? Explain with an example.
6. ExplaincommandswithrespecttoSQL:(i)Rename(ii)Alter(iii)View
7. ExplaintwoaggregatefunctionsofSQL.
8. ExplainthefollowingSQL constructswithexamples:  
(1)orderby(2)groupbyandhaving(3)aselect(4)schema

9. WriteSQL Queriesforfollowingset of tables:

EMPLOYEE (EmpNo, Name, DoB, Address, Gender, Salary, DNumber)

DEPARTMENT (DNumber, Dname, ManagerEmpNo,ManagerStartDate).

Display the Age of „male“ employees.

Display all employees in Department named „Marketing“.

Display the name of highest salary paid „female“ employee.

Which employee is oldest manger in company?

Displaythenameofdepartmentoftheemployee„SMITH.

10. DescribecreatingandmodifyingrelationsusingSQL.Giveexamplesforeach.
11. Explainaboutintegrityconstraintsoverrelations.
12. Considerthefollowinginsurancedatabase,wheretheprimarykeysareunderlined.

Construct thefollowing SQLqueries forthis relationaldatabase.person (driver-id#,name, address)

car(license,model,year)

accident (report-number, date, location)

owns (driver-id#, license)

participated(driver-id,car,report-number,damage-amount)

- a. Find the total numberof people who owned cars that were involved in accidents in1989.
- b. Findthenumberofaccidentsinwhichthecarsbelongingto“JohnSmith” were involved.
- c. Addanewaccidenttothedatabase;assumeanyvaluesforrequiredattributes.
- d. DeletetheMazdabelongingto “JohnSmith”.
- e. Update the damage amount for the car with licensenumber “AABB2000” in theaccident with report number “AR2197” to \$3000.

13. ListtheSQLfunctionsforstringconversions.

14. WriteSQLstatementsforfollowing:

Student(Enrno,name,courseId,emailId,cellno)

Course (courseId, course\_nm, duration)

- i. Addacolumncityinstudenttable.
  - ii. Findoutlistof studentswhohaveenrolledin“computer”course.
  - iii. Listnameofallcourseswiththeirduration.
  - iv. Listnameofallstudentsstartwith„a“.
  - v. ListemailIdandcellnoofallmechanicalengineeringstudents.
15. Howtodefineadomainconstraint?Giveanexample.
16. Whatisanintegrityconstraint?ExplainitsenforcementbyDBMSwithillustrative
17. ListthedataotypessupportedbySQL.
18. DemonstratetheuseofDISTINCTkeywordinSQLselectstatement.
19. ConsiderthefollowingdatabaseschematowritenestedqueriesinSQL
- Supplier (id, name, city)
- Parts (pno, pname, pdescription)
- Supply (id, pno, cost)
- i. Findthenamesofthepartssuppliedby“RamRaj”
  - ii. Findthenamesofthesupplierswhosupply“Nuts”
  - iii. FindthecostofboltsbeingsuppliedbyNagpursuppliers.

**References:**

- Raghurama Krishnan, Johannes Gehrke, *Database Management Systems*, 3rd Edition, TataMcGraw Hill.
- C.J.Date,*IntroductiontoDatabaseSystems*,PearsonEducation.
- ElmasriNavrate,*FundamentalsofDatabaseSystems*,PearsonEducation.

# DATABASEMANAGEMENTSYSTEMS

## UNITIII:EntityRelationshipModel (PART - 1)

### Syllabus:

**EntityRelationshipModel:**Introduction,Representationofentities,attributes,entityset, relationship,relationshipset,constraints,subclasses,superclass,inheritance,specialization, generalization using ER Diagrams.

**SQL:**Creatingtableswithrelationship,implementationofkeyandintegrityconstraints, nestedqueries,subqueries,grouping,aggregation,ordering,implementationofdifferent types of joins, view(updatable and non-updatable), relational set operations.

### Objectives:

Afterstudying this unit, youwill be ableto:

- Explainentityrelationshipmodel
- Knowtheconstraints
- DescribeE-RdiagramsandextendedERfeatures
- Describesubqueriesandnestedsubqueries
- Explaincomplexqueries
- Defineviews
- Discussjoinedrelations

## Introduction

The *entity-relationship (ER) data model* allows us to describe the data involved in a real-world enterprise in terms of objects and their relationships and is widely used to develop an initial database design. The ER model is important primarily for its role in database design. It provides useful concepts that allow us to move from an informal description of what users want from their database to a more detailed and precise description that can be implemented in a DBMS. Even though the ER model describes the physical database model, it is basically useful in the design and communication of the logical database model.

## Overview of Database Design

Our primary focus is the design of the database. The database design process can be divided into six steps:

### **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 and an analysis of any available documentation on existing applications and so on.

### **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 conditions known to hold this data. The goal is to create a description of the data that matches both—how users and developers think of the data (and the people and processes to be represented in the data). This facilitates 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.

### **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 relational DBMSs, and therefore, the task in the logical design step is to convert the conceptual database design in the form of E-R Schema (Entity-Relationship Schema) into a relational database schema.

### **Schema Refinement**

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

### PhysicalDatabaseDesign

This step may simply involve building indexes on some tables and clustering some tables, or it may involve redesign of part of the database schema obtained from the earlier design steps.

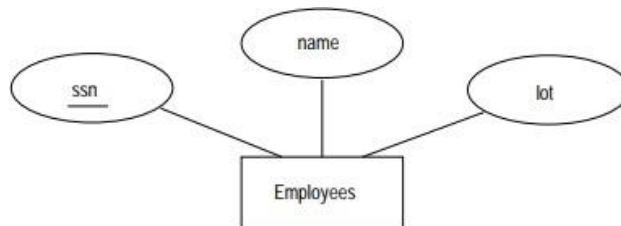
### ApplicationandSecurityDesign

Any software project that involves a DBMS must consider applications that involve processes and identify the entities.

### Entities,AttributesandEntitySets

**Entity:** An entity is an object in the real world that is distinguishable from other objects.

**Entity set:** An entity set is a collection of similar entities. The Employees entity set with attributes ssn, name, and lot is shown in the following figure.



**Attribute:** An attribute describes a property associated with entities. An attribute will have a name and a value for each entity.

**Domain:** A domain defines a set of permitted values for an attribute.

**Entity Relationship Model:** An ERM is a theoretical and conceptual way of showing data relationships in software development. It is a database modeling technique that generates an abstract diagram or visual representation of a system's data that can be helpful in designing a relational database.

ER model allows us to describe the data involved in a real-world enterprise in terms of objects and their relationships and is widely used to develop an initial database design.

### RepresentationofEntitiesandAttributes

**ENTITIES:** Entities are represented by using rectangular boxes. These are named with the entity name that they represent.

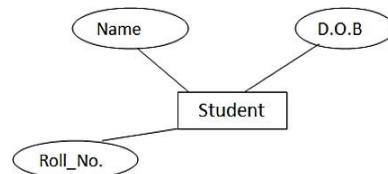


Fig: Student and Employee entities

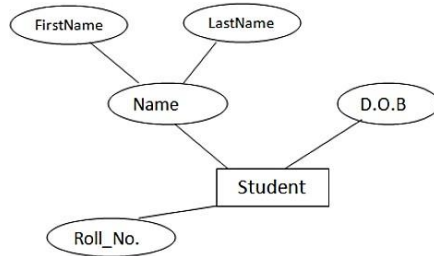
**ATTRIBUTES:** Attributes are the properties of entities. Attributes are represented by means of ellipses. Every ellipse represents one attribute and is directly connected to its entity.

### Typesofattributes:

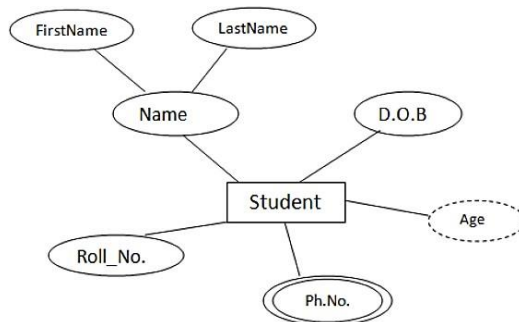
- **Simple attribute**– Simple attributes are atomic values, which cannot be divided further. For example, a student's roll number is an atomic value.



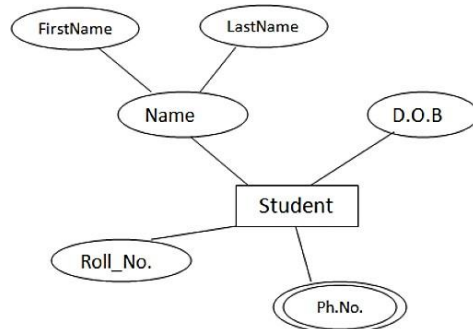
- **Composite attribute** – Composite attributes are made of more than one simple attribute. For example, a student's complete name may have first\_name and last\_name.



- **Derived attribute**– Derived attributes are the attributes that do not exist in the physical database, but their values are derived from other attributes present in the database. For example, average\_salary in a department should not be saved directly in the database, instead it can be derived. For another example, age can be derived from data\_of\_birth.



- **Single-value attribute**– Single-value attributes contain single value. For example – Social\_Security\_Number.
- **Multi-value attribute**– Multi-value attributes may contain more than one values. For example, a person can have more than one phone number, email\_address, etc.



### Relationship and Relationshipset

**Relationships** are represented by diamond-shaped box. Name of the relationship is written inside the diamond-box. All the entities (rectangles) participating in a relationship, are connected to it by a line.

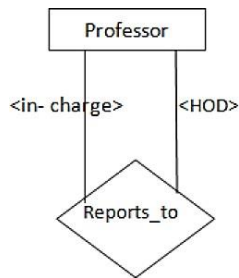
#### **Types of relationships:**

Degree of Relationship is the number of participating entities in a relationship. It defines the degree of the relationship. Based on degree, the relationships are categorized as

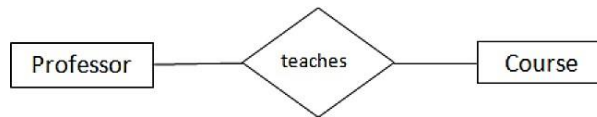
- Unary=degree 1
- Binary=degree 2

- Ternary=degree3
- n-array=degree

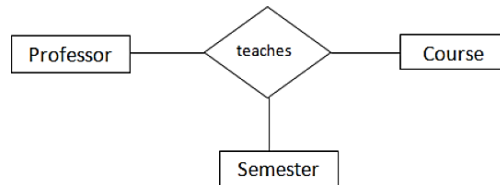
**Unary Relationship:** A relationship with one entity set. It is like a relationship among 2 entities of same entity set. Example: A professor (in-charge) reports to another professor (Head of the Dept).



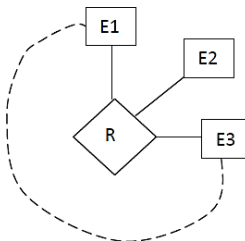
**Binary Relationship:** A relationship among 2 entity sets. Example: A professor teaches a course and a course is taught by a professor.



**Ternary Relationship:** A relationship among 3 entity sets. Example: A professor teaches a course in so and so semester.



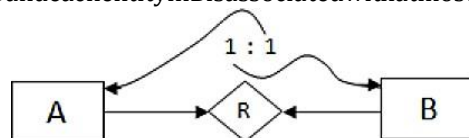
**n-array Relationship:** A relationship among n entity sets.



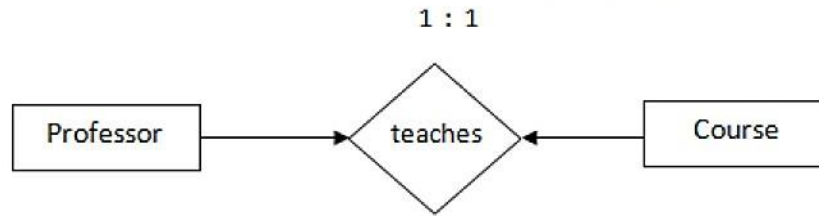
### Cardinality:

Defines the number of entities in one entity set, which can be associated with the number of entities of the other set via the relationship set. Cardinality ratios are categorized into 4. They are.

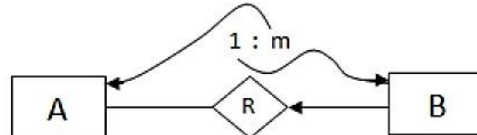
- One-to-One relationship:** When only one instance of entity A is associated with the relationship, then the relationship is one-to-one relationship. Each entity in A is associated with at most one entity in B and each entity in B is associated with at most one entity in A.



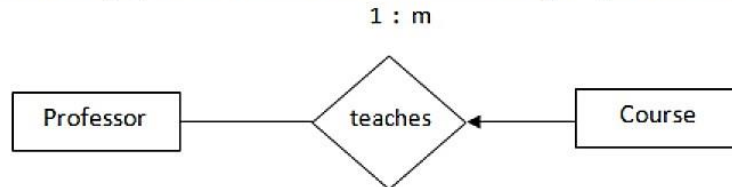
Each professor teaches one course and each course is taught by one professor.



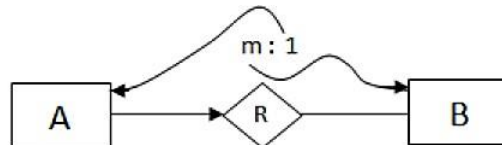
2. **One-to-many relationship:** When more than one instance of an entity is associated with a relationship, then the relationship is one-to-many relationship. Each entity in A is associated with zero or more entities in B and each entity in B is associated with at most one entity in A.



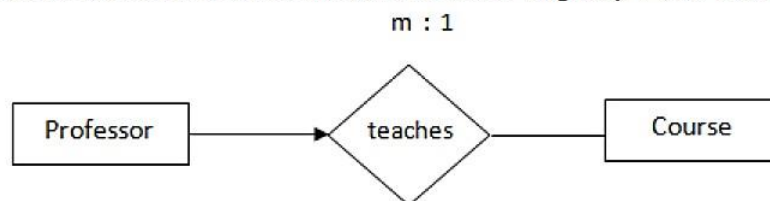
Each professor teaches 0 (or) more courses and each course is taught by at most one professor.



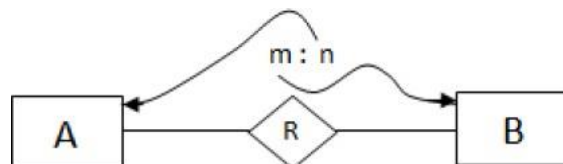
3. **Many-to-one relationship:** When more than one instance of entity is associated with the relationship, then the relationship is many-to-one relationship. Each entity in A is associated with at most one entity in B and each entity in B is associated with 0 (or) more entities in A.



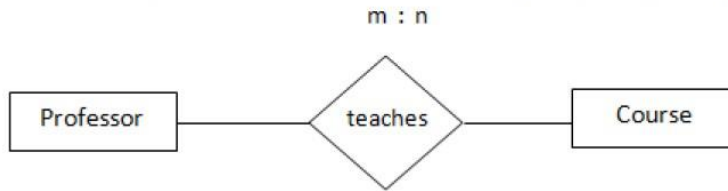
Each professor teaches at most one course and each course is taught by 0 (or) more professors.



4. **Many-to-Many relationship:** If more than one instance of an entity on the left and more than one instance of an entity on the right can be associated with the relationship, then it depicts many-to-many relationship. Each entity in A is associated with 0 (or) more entities in B and each entity in B is associated with 0 (or) more entities in A.



Each professor teaches 0 (or) more courses and each course is taught by 0 (or) more professors.



### RelationshipSet:

A set of relationships of similar type is called a relationship set. Like entities, a relationship too can have attributes. These attributes are called descriptive attributes.

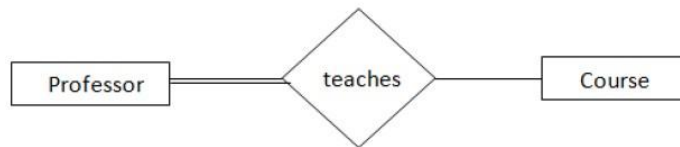
### ParticipationConstraints:

- **Total Participation** – If Each entity in the entity set is involved in the relationship then the participationoftheentitysetissaidtobetotal.Totalparticipationisrepresentedbydouble lines.
- **Partial participation** – If, Not all entities of the entity set are involved in the relationship thensuchaparticipationissaidtobepartial.Partialparticipationisrepresentedbysinglelines.

### Example:

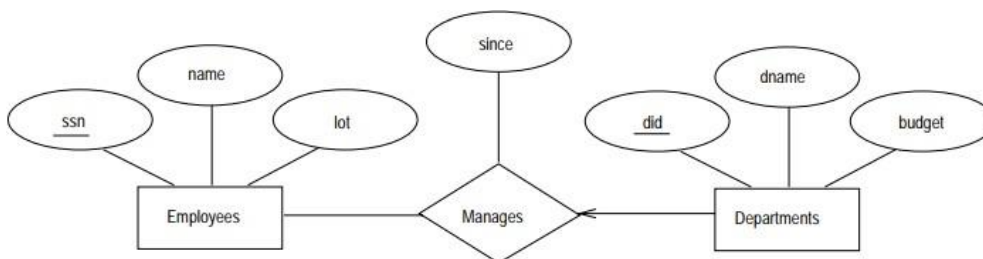
Each Professor teaches at least one course.

min=1 (Total Participation)  
max=many (No key)



### AdditionalFeaturesOfTheERModel Key Constraints

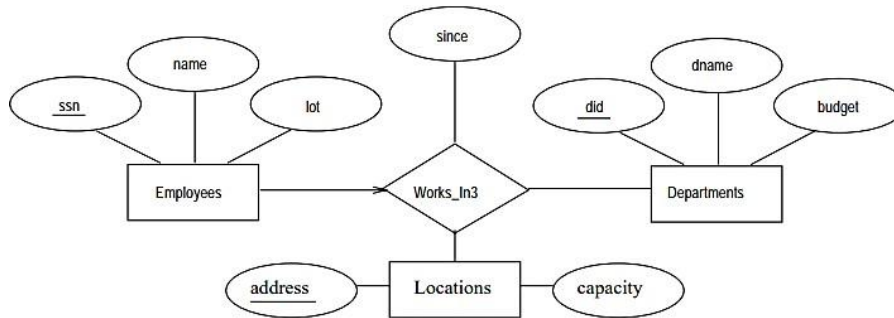
ConsiderarelationshipsetcalledManagesbetweentheEmployeesandDepartmentsentitysets suchthateachdepartmenthasatmostonemanager,althoughasingleemployeeisallowedto managemorethanonedepartment.Therestrictionthateachdepartmenthasatmostonemanager isanexampleofa**keyconstraint**,anditimpliesthateachDepartmentsentityappearsinatmost oneManagesrelationshipinanyallowableinstanceofManages.Thisrestrictionisindicatedinthe ERdiagramofbelowFigurebyusinganarrowfromDepartmentstoManages.Intuitively,the arrowstatesthatgivenaDepartmentsentity,wecanuniquelydeterminetheManagesrelationship in which it appears.



## Key Constraints for Ternary Relationships

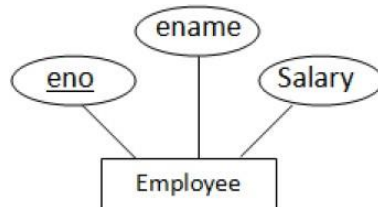
If an entity set  $E$  has a key constraint in a relationship set  $R$ , each entity in an instance of  $E$  appears in at most one relationship in (a corresponding instance of)  $R$ . To indicate a key constraint on entity set  $E$  in relationship set  $R$ , we draw an arrow from  $E$  to  $R$ .

Below figure show a ternary relationship with key constraints. Each employee works in at most one department, and at a single location.

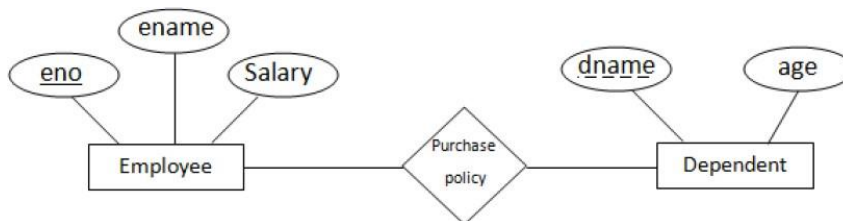


## Weak Entities

**Strong Entity set:** If each entity in the entity set is distinguishable or it has a key then such an entity set is known as strong entity set.



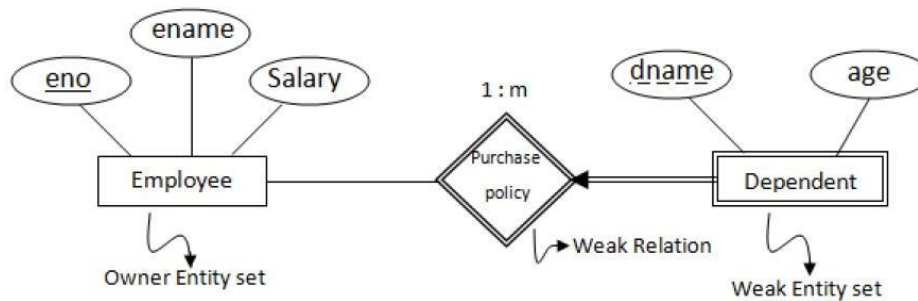
**Weak Entity set:** If each entity in the entity set is not distinguishable or it doesn't has a key then such an entity set is known as weak entity set.



$eno$  is key so it is represented by solid underline.  $dname$  is partial key. It can't distinguish the tuples in the Dependent entity set. so  $dname$  is represented by dashed underline.

Weak entity set is always in total participation with the relation. If entity set is weak then the relationship is also known as weak relationship, since the dependent relation is no longer needed when the owner left.

Ex: policy dependent details are not needed when the owner (employee) of that policy left or fired from the company or expired. The detailed ER Diagram is as follows.



The cardinality of the owner entity set with weak relationship is 1:m. Weak entity set is uniquely identifiable by partial key and key of the owner entity set.

Dependent entity set is key to other relation because all the tuples of weak entity set are associated with the owner entity set tuples.

Dependents 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.

### E-R Diagrams Implementation

Now we are in a position to write the ER diagram for the Company database which was introduced in the beginning of this unit. The readers are strictly advised to follow the steps shown in this unit to design an ER diagram for any chosen problem.

#### **Step 1: Identify the Strong and Weak Entity Sets**

After careful analysis of the problem we come to a conclusion that there are four possible entity sets as shown below:

- |                |                   |
|----------------|-------------------|
| 1. Employees   | Strong Entity Set |
| 2. Departments | Strong Entity Set |
| 3. Projects    | Strong Entity Set |
| 4. Dependents  | Weak Entity Set   |

#### **Step 2: Identify the Relevant Attributes**

The next step is to get all the attributes that are most applicable for each entity set. Do this work by considering each entity set in mind and also the type of attributes. Next job is to pick the primary key for strong entity sets and partial key for weak entity sets.

**Example:** Following are the attributes:

- |                      |                                             |
|----------------------|---------------------------------------------|
| 1. Employees         | SSN, Name, Addr, Date Of Birth, Sex, Salary |
| 2. Departments       | DNo, DName, DLocation                       |
| 3. Projects          | PNo, PName, PLocation                       |
| 4. Dependents (weak) | DepName, Date Of Birth, Sex, Relationship   |

The underlined attributes are the primary keys and DepName is the partial key of Dependents. Also, DLocation may be treated as a multivalued attribute.

### Step3:IdentifytheRelationshipSets

In this step we need to find all the meaningful relationship sets among possible entity sets. This step is very tricky, as redundant relationships may lead to complicated design and in turn a bad implementation.

**Example:** Let us show below what the possible relationship sets are:

- |                              |            |
|------------------------------|------------|
| 1. Employees and Departments | WorksFor   |
| 2. Employees and Departments | Manages    |
| 3. Departments and Projects  | Controls   |
| 4. Projects and Employees    | WorksOn    |
| 5. Dependents and Employees  | Has        |
| 6. Employees and Employees   | Supervises |

Some problems may not have recursive relationship sets but some do have. In fact, our Company database has one such relationship set called Supervises. You can complete this step adding possible descriptive attributes of the relationship sets (Manages has StartDate and WorksOn has Hours).

### Step4:IdentifytheCardinalityRatioandParticipationConstraints

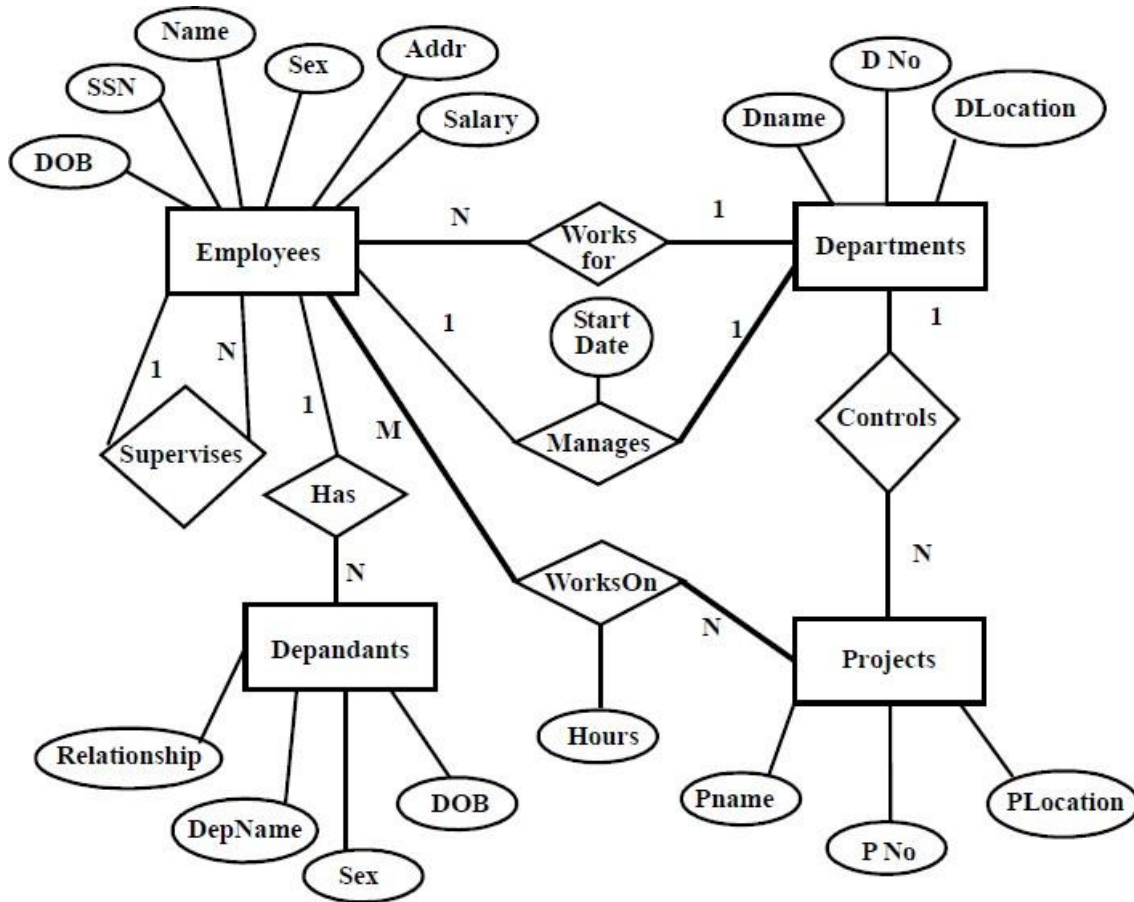
This step is relatively as simple one. Simply apply the business rules and your common sense. So, we write the structural constraints for our example as follows:

- |             |                                                       |
|-------------|-------------------------------------------------------|
| 1. WorksFor | N:1 Total on either side                              |
| 2. Manages  | 1:1 Total on Employees and Partial on Departmentsside |
| 3. Controls | 1:N Total on either side                              |
| 4. WorksOn  | M:N Total on either side                              |
| 5. Has      | 1:M Total on Dependents and Partial on Employees      |

### Step5:IdentifytheIS-AandHas-ARelationshipSets

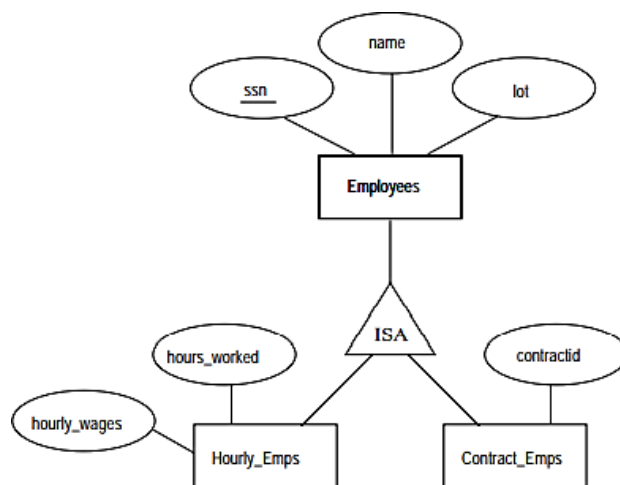
The last step is to look for “is-a” and “has-a” relationship sets for the given problem. As far as the Company database is concerned, there are no generalization and aggregation relationships in the Company database.

The complete single ER diagram by combining all the above five steps is shown in figure



### 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 Contract-Emps entity set to distinguish the basis on which they are paid. Then the class hierarchy is illustrated as follows.



This class hierarchy illustrates the inheritance concept. Where, the subclass attributes ISA (read as :isa) superclass attributes; indicating the “isa” 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 entity set are the attributes of Contract-Emps plus attributes of Employees.

### Class Hierarchy based on Sub-superSet

1. **Specialization:** Specialization is the process of identifying subsets (subclasses) of an entity set (superclass) that share some special distinguishable characteristic. Here, the superclass (Employee) is defined first, then the subclasses (Hourly-Emps, Contract-Emps, etc.) are defined next.

In short, Employees is specialized into subclasses.

2. **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 (possesses) these common characteristics. Here, the subclasses (Hourly-Emps, Contract-Emps, etc.) are defined first, then the Superclass (Employee) is defined, next.

In short, Hourly-Emps and Contract-Emps are generalized by Employees.

### Class Hierarchy based on Constraints

1. **Overlap 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 specialisation hierarchy property. We denote this by writing "Contract-Emps OVERLAP Senior-Emps".

2. **Covering Constraints:** Covering constraints determine whether the entities in the subclasses collectively include all entities in the superclass.

**Example:** Should every Employee be a Hourly-Emps or a 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

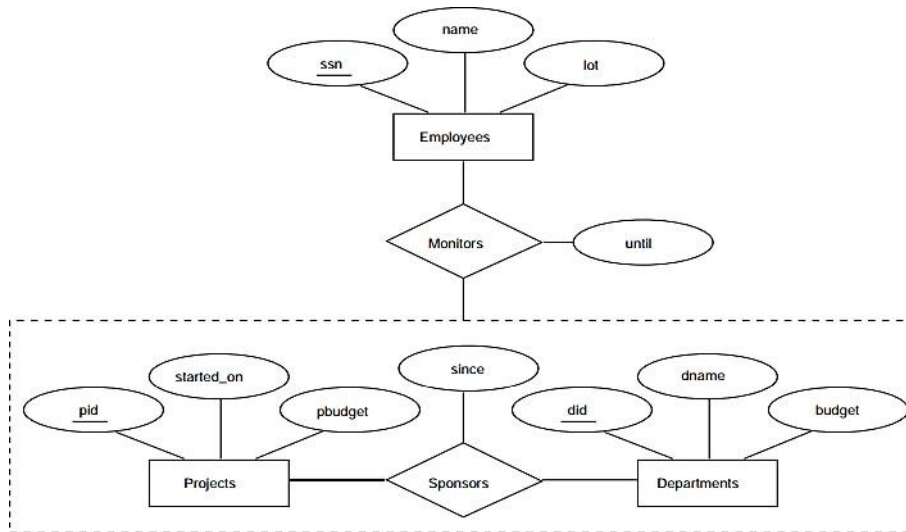
Aggregation allows us to indicate that a relationship set (identified through a dashed box) participates in another relationship sets. That is, a relationship set in an association between entity sets. Sometimes we have to model a relationship between a collection of entities and relationships.

*Example:* Suppose that we have an entity set called Project and that each Project entity is sponsored by one or more departments. Thus, the sponsors relationship set captures this information but, a department that sponsors a project, might assign employee to monitor the sponsorship. Therefore, Monitor should be a relationship set that associates a sponsor's relationship (rather

thanaProjectorDepartmententity)withanEmployeesentity.However,againwehavetodefine relationships to associate two or more entities.

### UseofAggregation

Weusean aggregation,when weneedto express arelationship among relationships.Thus,there arereallytwodistinctrelationships,SponsorsandMonitors,eachwithitsownattributes.



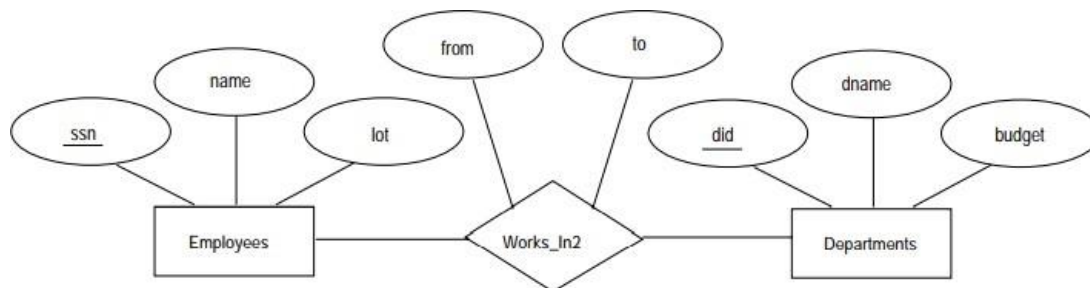
### ConceptualDatabaseDesign WithTheERModel(ERDesignIssues)

ThefollowingaretheERdesignissues:

1. Useentrysetsattributes
2. UseofEntitysetsorrelationshipsets
3. Binaryversusentryrelationshipsets
4. Aggregationversusternaryrelationship.

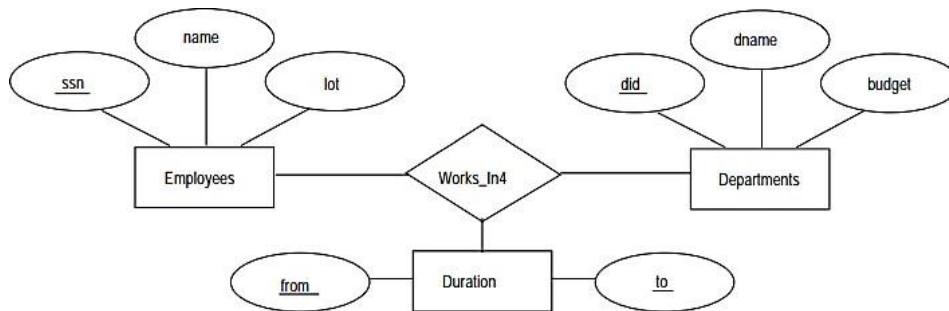
#### 1. UseofEntitySetsversusAttributes

Considertherelationshipset(calledWorksIn2)showninFigure



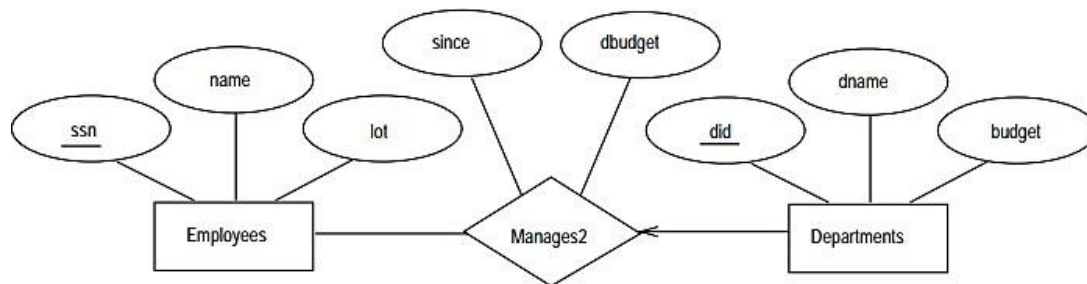
Intuitively, it records the interval during which an employee works for a department. Now suppose that it is possible for an employee to work in a given department over more than one period.

This possibility is ruled out by the ER diagram's semantics. The problem is that we want to record several values for the descriptive attributes for each instance of the Works\_In2 relationship. (This situation is analogous to wanting to record several addresses for each employee.) We can address this problem by introducing an entity set called, say, Duration, with attributes from and to, as shown in Figure



## 2. Entity versus Relationship

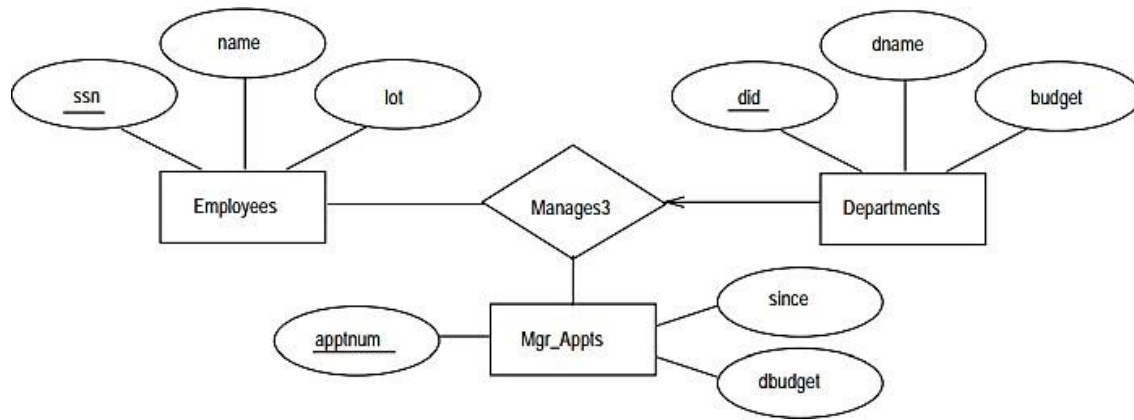
Consider the relationship set called Manages that each department manager is given a discretionary budget (dbudget), as shown in below figure, in which we have also renamed the relationship set to Manages2.



There is at most one employee managing a department, but a given employee could manage several departments; we store the starting date and discretionary budget for each manager- department pair. This approach is natural if we assume that a manager receives a separate discretionary budget for each department that he or she manages.

But what if the discretionary budget is a sum that covers all departments managed by that employee? In this case each Manages2 relationship that involves a given employee will have the same value in the dbudget field. In general such redundancy could be significant and could cause a variety of problems. Another problem with this design is that it is misleading.

We can address these problems by associating dbudget with the appointment of the employee as manager of a group of departments. In this approach, we model the appointment as an entity set, say Mgr\_Appts, and use a ternary relationship, say Manages3, to relate a manager, an appointment, and a department. The details of an appointment (such as the discretionary budget) are not repeated for each department that is included in the appointment now, although there is still one Manages3 relationship instance per such department. Further, note that each department has at most one manager, as before, because of the key constraint. This approach is illustrated in below Figure.

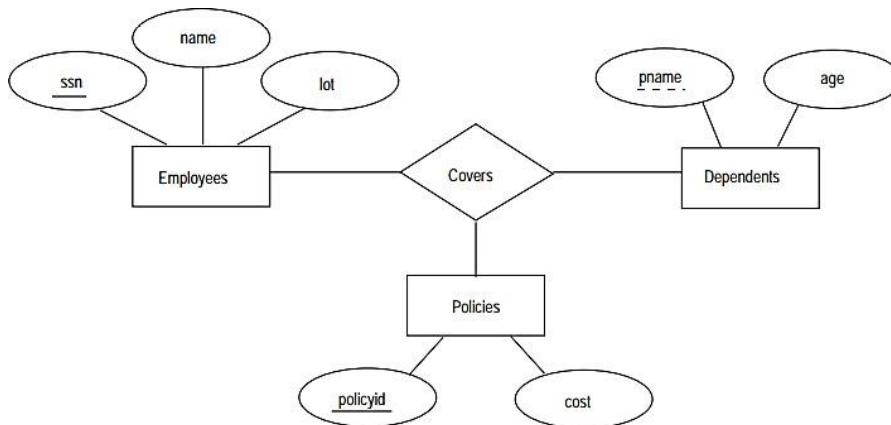


### 3. BinaryversusTernaryRelationships

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

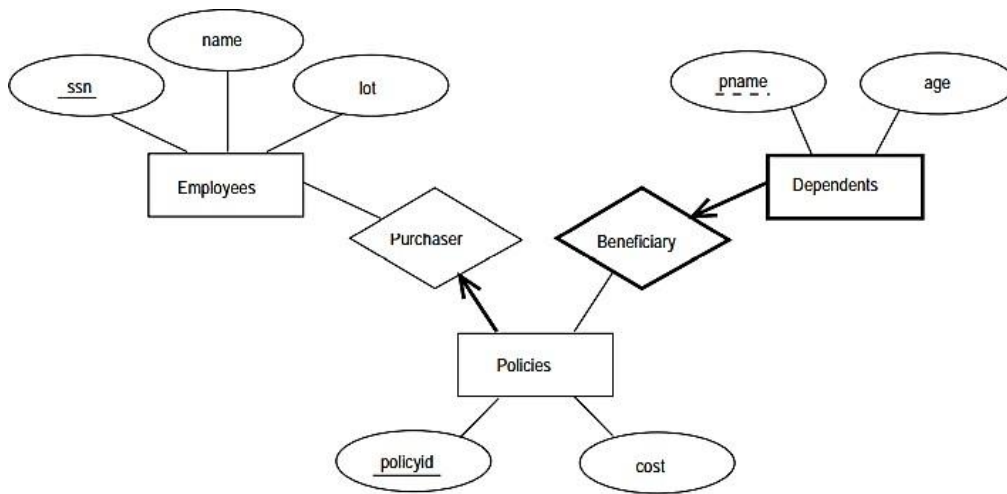
Suppose that we have the following additional requirements:

- A policy cannot be owned jointly by two or more employees.
- Every policy must be owned by some employee.
- Dependents is a weak entity set, and each dependent entity is uniquely identified by taking pname in conjunction with the policyid of a policy entity (which, intuitively, covers the given dependent).



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

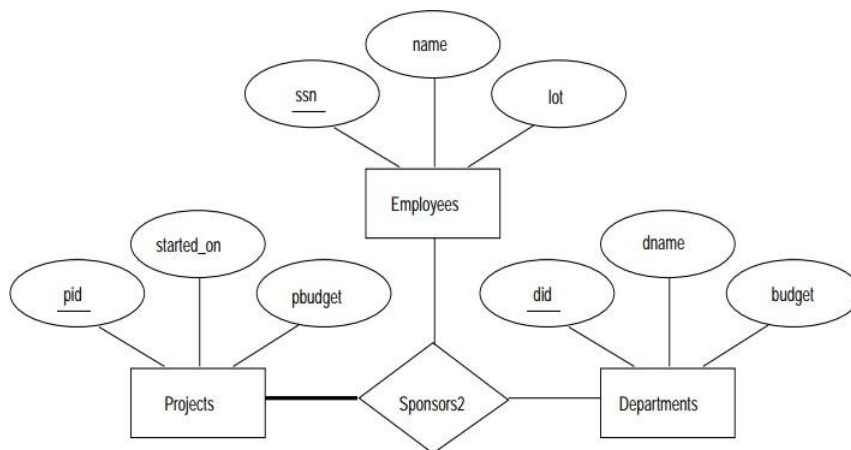
Even ignoring the third point above, the best way to model this situation is to use two binary relationships, as shown in below figure.



#### 4. AggregationversusTernaryRelationships

The choice between using aggregation or a ternary relationship is mainly determined by the existence of relationship that relates a relationship set to an entity set (or second relationship set). The choice may also be guided by certain integrity constraints to we want to express.

Considertheconstraintthateachsponsorship(ofaprojectbyadepartment)bemonitoredbyat most one employee. We cannot express this constraint in terms of the Sponsors2 relationship set. Also we can express the constraint by drawing an arrow from the aggregated relationship. Sponsorstothetherelationship Monitors.Thus,the presenceof such aconstraint serves asanother reason for using aggregation rather than a ternary relationship set.



#### ReviewQuestions

1. Definethefollowingterms andgive examples  
(i)cardinality(ii)unaryrelationships(iii)aggregation(iv)specialization
2. What is Entity set?andalsodefineRelationship set.List andexplain thesymbolsusedto draw ER Diagram.
3. Designadatabaseforanairline.Thedatabasemustkeeptrackofcustomersandtheir

reservations, flights and their status, seat assignments on individual flights, and the schedule and routing of future flights. Your design should include an E-R diagram, a set of relational schemas, and a list of constraints, including primary-key and foreign-key constraints.

4. Discuss the representation of total participation and multivalued attribute in an E/R diagram.
5. What is an Entity Relationship diagram and why is it useful?
6. What is a weak entity in ER diagram?
7. Give the diagrammatic representation of recursive relationship in an ER diagram and also explain the importance of role names in representing a recursive relationship by taking a real time example.
8. Consider a database used to record the marks that students get in different exams of different course offerings.
  - a. Construct an E-R diagram that models exams as entities, and uses a ternary relationship, for the above database.
  - b. Construct an alternative E-R diagram that uses only a binary relationship between students and course-offerings. Make sure that only one relationship exists between a particular student and course-offering pair, yet you can represent the marks that a student gets in different exams of a course offering.
9. Explain about relationship sets in ER model with examples.
10. Explain about ER model design issues.

### **References:**

- Raghurama Krishnan, Johannes Gehrke, *Database Management Systems*, 3rd Edition, Tata McGraw Hill.
- C.J.Date, *Introduction to Database Systems*, Pearson Education.
- Elmasri Navrate, *Fundamentals of Database Systems*, Pearson Education.

## UNIT- 3PART-IISQL

- **Creating Tables with relationship**
- **Implementation of key and integrity constraints**
- **Relational Set Operations**
- **Implementation of different types of Joins**
- **Views (Updatable and Non Updatable)**
- **Grouping, Aggregation, Ordering**
- **Nested queries, Subqueries**

## I.INTRODUCTION TO CREATING TABLES WITH RELATIONSHIP

### Creating tables using CREATE command:

This command is used to create a database and its objects such as Tables, Views, Procedures, Triggers etc. It defines each column of the table uniquely. Each column has a minimum of **three** attributes, a column name, data type and size.

```
Syntax: CREATETABLE <table_name>
 (
 column_name1DATATYPE1 (SIZE),
 column_name2DATATYPE2(SIZE),
 :
 column_namenDATATYPEN(SIZE));
```

## CREATING TABLES WITH RELATIONSHIP

When we want to create tables with relationship, we need to use **Referential integrity constraints**. The referential integrity constraint **enforces relationship** between tables.

- It designates a column or combination of columns as a **Foreign key**.

-The foreign key establish a **relationship** with a specified primary or unique key in another table called the **Referenced key**.

- Whenreferentialintegrityisenforced,itpreventsfrom..

- 1) Adding records to a related table if there is no associated record in the primary table.
- 2) Changing values in a primary table that result in orphaned records in a related table.
- 3) Deleting records from a primary table if there are matching related records.

Note: The table containing the foreign key is called the child table and the table containing the referenced key is called the Parent table.

**SYNTAX:**   **CREATETABLE**<tablename>(

**col\_name1datatype[size],**

**col\_name2datatype[size] ,**

**:**

**col\_namendatatype[size],**

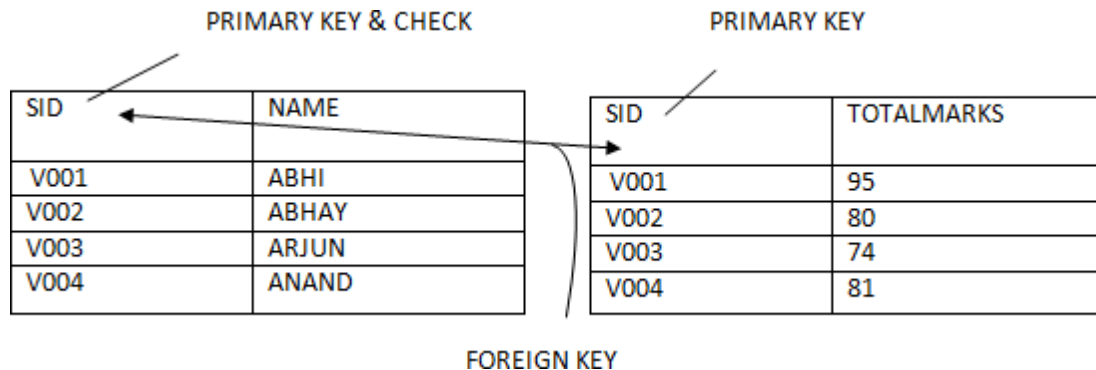
**FOREIGNKEY(column name)REFERENCES     <parent table name>(column name));**

**EX:** SQL>CREATETABLE marks(

```

sid VARCHAR2(4),
marks NUMBER(3),
PRIMARY KEY(sid),
FOREIGN KEY(sid) REFERENCES student1(sid));

```



## II.IMPLEMENTATIONOFKEYANDINTEGRITYCONSTRAINTS

**Data constraints:** All business of the world run on business data being gathered, stored and analyzed. Business managers determine a set of business rules that must be applied to their data prior to it being stored in the database/table of ensure its integrity.

For instance , no employee in the sales department can have a salary of less than Rs.1000/- .Such rules have to be enforced on data stored. If not, inconsistent data is maintained in database.

Note:ItisusedtoimposebusinessrulesonDBs.  
Itallowstoenteronlyvaliddata.

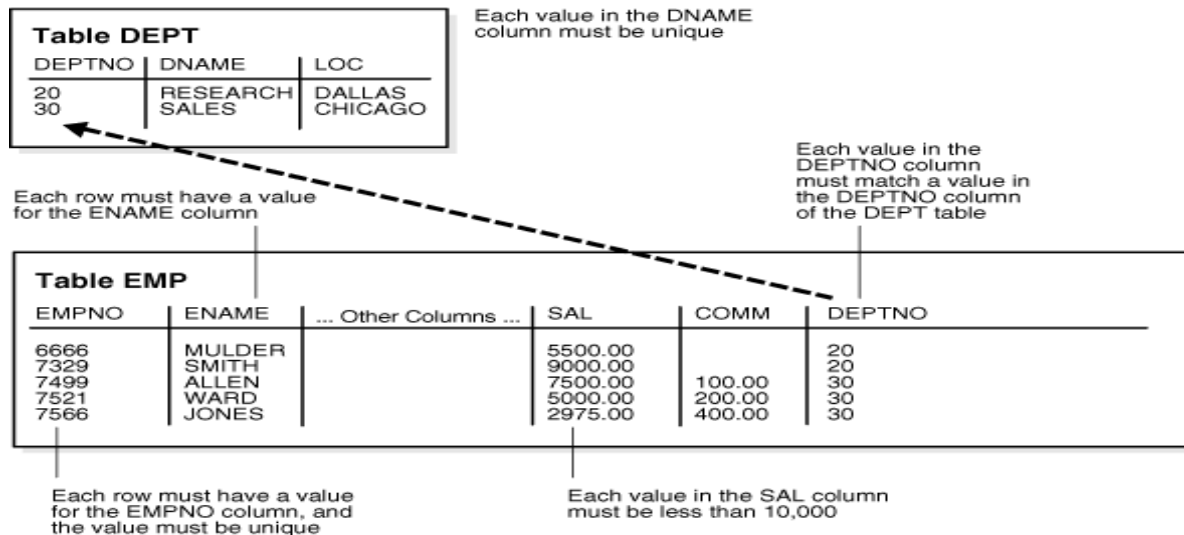
### VarioustypesofIntegrityConstraints:

Integrity constraints are the **rules** in real life, which are to be imposed on the data. If the data is not satisfying the constraints then it is considered as **inconsistent**. These rules are to be enforced on data because of the presence of these rules in real life. These rules are called integrity constraints. Every DBMS software must enforce integrity constraints, otherwise inconsistent data is generated.

### You can use constraint to do the following:

- To prevent invalid data entry into tables.
- To **Enforce rules** on the data in a table whenever a row is inserted, updated, or deleted from that table. The constraint must be satisfied for the operation to succeed.
- To **Prevent the deletion** of a record from a table if there are dependencies.

### Example for Integrity Constraints:-



Constraints are categorized as follows.

**1. Domain integrity constraints** - A domain means a set of values assigned to a column. i.e. a set of permitted values. Domain constraints are handled by

- Defining proper **datatype**
- Specifying **not null** constraint
- Specifying **check** constraint.
- Specifying **default** constraint

**Not null** - indicates that a column cannot store NULL value.

**Check** - Ensure that the value in a column meets a specific condition.

**Default** - prevents null value in a column when a value is not provided in a column by user. So, it assigns default value or globally assigned value.

**2. Entity integrity constraints** - are TWO types

**Unique constraint** - It defines an entity or column as a UNIQUE for a particular table.

And ensure that each row of a column must have a UNIQUE value or name.

**Primary key constraint** - This avoids duplicate and null values. It is a combination of a NOT NULL and UNIQUE.

**3. Referential integrity constraints**

**Foreign key** - indicates the relationship between child and parent tables.

This constraint is always attached to a column in a table.

We can add constraints in **two** ways.

**Column level:-**

- Constraint is declared immediately declaring column.
- Define with each column
- Use column level to declare constraint for a single column.
- Composite key cannot be defined at column level.

**Table level:-**

- constraint is declared after declaring all columns.
- use table level to declare constraint for combination of columns. (i.e. composite key)
- **not null** cannot be

defined. Another type is possible at **Alter**

**level**

- constraint is declared with ALTER command.
- When we use this, make sure that the table should not contain data.

To add these constraints, we can use constraint with label or without label. TWO

BASIC TYPES -----

1. Constraint WITH NAME
2. constraints WITHOUT NAME.

i)

Declaring Constraint at "TABLE" level (Constraints with label) Syntax

```
:- CREATE TABLE <table name>
 (
 col_name1 DATATYPE(SIZE),
 ,
 col_nameN DATATYPE(SIZE) ,
 CONSTRAINT <cons_label> NAME_OF_THE_CONSTRAINT[column_list]);
```

ii) Declaring Constraint at "Column" level (Constraints with label) Synt

ax :-

```
col_name DATATYPE(SIZE) CONSTRAINT <cons_label> NAME_OF_THE_CONSTRAINT
```

iii)

Declaring Constraint at "TABLE" level (Constraints without label) Syntax

:-

```
CREATE TABLE <tablename> (
 col_name1 DATATYPE(SIZE),
 ,
 col_nameN DATATYPE(SIZE) ,
 NAME_OF_THE_CONSTRAINT[column_list]);
```

iv)

Declaring Constraint at "Column" level (Constraints without label) Syntax

:-

```
col_name DATATYPE(SIZE) NAME_OF_THE_CONSTRAINT
```

v) Adding Constraint to a table at "Alter" level (Constraints with label)

A constraint can be added to a table at any time after the table was created by using ALTER TABLE statement, using **ADD** clause.

Syntax:

```
ALTER TABLE <table_name> ADD CONSTRAINT cont_label NAME OF _
THE _CONSTRAINT(column);
```



- Value entered in the column **should not violate the condition**.
- Check constraint allows **null** values.
- Check constraint can be declared at **table level** or **column level**.
- There is no limit to the number of CHECK constraints that can be defined on a condition.

#### Limitations:-

- Conditions should not contain/not applicable to pseudocolumns like ROWNUM, SYSDATE etc.
- Conditions should not access columns of another table

#### **//CONSTRAINT@COLUMNLEVEL**

**SYNTAX:** column\_name DATATYPE (SIZE) **CHECK** (condition) //without label

Here, we are creating a table with two columns such as Sid, sname.

Ex: 

```
CREATE TABLE check_column(
 sid VARCHAR2(4) CHECK(sid LIKE 'C%' AND LENGTH(sid)=4), //without label
 sname VARCHAR2(10));
```

Here, **sid** should start with 'C' and a length of sid is exactly 4 characters.

SQL> SELECT \* FROM check\_column;

SID SNAME

-----

C501 MANI

C502 DHANA

C503 RAVI

C504 RAJA

#### **//with label**

**Syntax:** Column\_name DATATYPE (SIZE) CONSTRAINT constraint\_label  
CHECK (condition)

```
CREATE TABLE check_column(
 sid VARCHAR2(4) CONSTRAINT ck CHECK(sid LIKE 'C%' AND LENGTH(sid)=4), sname
 VARCHAR2(10));
```

#### **//CHECK@TABLELEVEL**

```
CREATE TABLE check_table(
 sid VARCHAR2(4) ,
 sname VARCHAR2(10),
 CHECK (sid LIKE 'C%' AND LENGTH(sid)=4),
 CHECK (sname LIKE '%A'));
```

Here, **sid** should start with 'C' and a length of sid is exactly 4 characters. And **sname** should end with letter 'A'

SQL>SELECT\*FROMcheck\_table;

SIDSNAME

-----

C401 ABHILA

C401 ANITHA

C403NANDHITHA

C522 LOHITHA

//withlabel

CONSTRAINTck1CHECK(sidLIKE'C%'ANDLENGTH(sid)=4), CONSTRAINT  
ck2 CHECK(sname LIKE '%A'));

**@ALTER LEVEL**

**Here,weaddcheckconstrainttonew tablewithcolumns.**

SQL>CREATETABLEcheck\_alter(sidVARCHAR2(4),  
Sname VARCHAR2(10));

**//CHECK@ALTERLEVEL://CONSTRAINTWITHOUTNAME**

**SQL> ALTER TABLE <table name>ADDCHECK (condition );**

**SQL>ALTERTABLEcheck\_alterADDCHECK(sidlike'C%');**

**SYNTAX: ALTERTABLE<table\_name>ADDCONSTRAINTcont\_name  
CHECK(cond);**

SQL>ALTERTABLEcheck\_alterADDCONSTRAINTckCHECK(sidLIKE'C%');

**ANOTHEREXAMPLEFORTABLELEVEL CONSTRAINT**

**Here,WecreatetablewithTHREEcolumns**

**ADDCHECKCONSTRAINT@TABLELEVEL(ATTHEENDOFTABLE  
DEFINITION)**

**MARKSINBETWEEN0AND100.**

**SQL>CREATETABLEmarks2(sid VARCHAR2(4),  
sec VARCHAR2(2),  
marks NUMBER(3),  
CHECK(marks>0ANDmarks<=100)); DROP**

**@ CHECK CONSTRAINT**

**SYNTAX:ALTERTABLE<table\_name>DROPCONSRAINTcont\_name;**

SQL>ALTERTABLEcheck\_table DROPCONSTRAINTck;

## DEFAULT

- If values are not provided for table column, default will be considered.
- This prevents NULL values from entering the columns, if a row is inserted without a value for a column.
- The default value can be a literal, an expression, or a SQL function.
- The default expression must match the data type of the column.
- The DEFAULT constraint is used to provide a default value for a column.
- The default value will be added to all new records IF no other value is specified.

**Syntax:** Column\_name data type (size) **DEFAULT** <value/expression/function> Ex:

```
MIDDLENAME VARCHAR(10) DEFAULT 'UNAVAILABLE'

CONTACTNO NUMBER(10) DEFAULT 9999999999
```

This defines what value the column should use when no value has been supplied explicitly when inserting a record in the table.

```
CREATE TABLE tab_default(sid NUMBER(10),
 contactno NUMBER(10) DEFAULT 9999999999);
```

**Add data to table:**

```
Insert into tab_default(sid,contactno) values(501,9493949312);
Insert into tab_student(sid) values(502);
Insert into tab_student(sid) values(503);
Insert into tab_student(sid,sname) values(504,9393949412);
```

**Select \* from tab\_default;**

**SID CONTACTNO**

```

5019493949312
5029999999999
5039999999999
5049393949412
```

## UNIQUE

- Columns declared with UNIQUE constraint **does not accept duplicate values**.
- One table can have a number of unique keys.
- Unique key can be defined on more than one column. i.e. composite unique key
- A composite key UNIQUE key is always defined at the table level only.
- By default UNIQUE columns accept null values unless declared with NOT NULL constraint
- Oracle automatically creates UNIQUE index on the column declared with UNIQUE constraint

- UNIQUE constraint can be declared at column level and table level.

### UNIQUE@COLUMN LEVEL

**SYNTAX:** column\_name DATA\_TYPE(SIZE) UNIQUE

```
CREATE TABLE table_unique(
 sid NUMBER(4) UNIQUE,
 sname VARCHAR2(10));
```

### //UNIQUE@TABLE LEVEL

**SYNTAX:** UNIQUE(COLUMN\_LIST);

```
CREATE TABLE table_unique2(
 sid NUMBER(4),
 sname VARCHAR2(10),
 UNIQUE(sid, sname));
```

```
SQL> SELECT * FROM TABLE_UNIQUE2;
SID
SNAME
```

-----

401 RAMU

402 SITA

//Here, these two records are distinct not the same.

402 GITHA

403 GITHA

404 RAMU

### Unique@ALTER level:

```
Alter table table_unique ADD UNIQUE(sid) //without label
```

```
Alter table table_unique ADD CONSTRAINT Tuq UNIQUE(sid) //with label
```

### DROP UNIQUE@TABLE LEVEL

```
SQL> ALTER TABLE table_unique2 DROP UNIQUE(sid, sname);
```

Now, we removed unique constraint, so now this table consists duplicated data.

//UNIQUE@ALTER LEVEL (here, the table contains duplicates, so it is not works)

//deleted data from table\_unique2

```
SQL> DELETE FROM table_unique2;
```

### PRIMARY KEY constraint:-

PRIMARY KEY is one of the **candidate** keys, which uniquely identifies a record in a table.

-used to define key column of a table.

-it is provided with an automatic index.

-A primary key constraint combines a NOT NULL and UNIQUE behavior in one declaration.

### Characteristics of PRIMARY KEY:-

➔ There should be at the most one Primary Key or Composite primary key per table.

- PK column does not accept null values.
- PK column does not accept duplicate values.
- RAW, LONG RAW, VARRAY, NESTED TABLE, BFILE columns cannot be declared with PK
- If PK is composite then uniqueness is determined by the combination of columns.
- A composite primary key cannot have more than 32 columns
- It is recommended that PK columns should be short and numeric.
- Oracle automatically creates Unique Index on PK column

EX:

| Table DEPT |          |         |
|------------|----------|---------|
| DEPTNO     | DNAME    | LOC     |
| 20         | RESEARCH | DALLAS  |
| 30         | SALES    | CHICAGO |

**PRIMARY KEY**  
(no row may duplicate a value in the key and no null values are allowed)

//PRIMARY KEY@COLUMN LEVEL

**SYNTAX: column\_name DATA\_TYPE(SIZE) PRIMARY KEY**

```
SQL> CREATE TABLE student1(
 sid VARCHAR2(4) PRIMARY KEY
 CHECK(sid LIKE 'V%' AND LENGTH(sid)=4), name
 VARCHAR2(10));
```

```
SQL> DESC student1;
```

| Name | Null?    | Type         |
|------|----------|--------------|
| SID  | NOT NULL | VARCHAR2(4)  |
| NAME |          | VARCHAR2(10) |

**CASE2: ADD PRIMARY KEY@ALTER LEVEL**

```
SQL> CREATE TABLE student2(sid VARCHAR2(4),
 name VARCHAR2(10));
```

**SYNTAX: ALTER TABLE <tablename> ADD PRIMARY KEY(col\_name);**

**SQL> ALTER TABLE student2 ADD PRIMARY KEY(sid);**

Table altered.

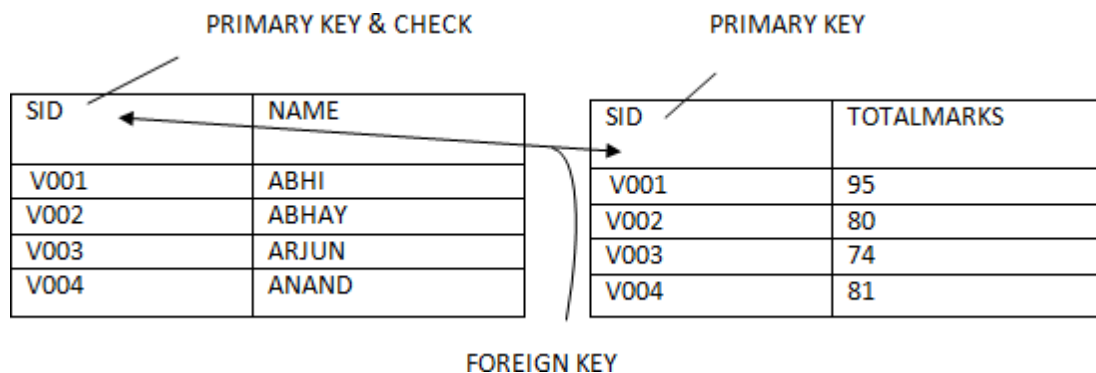
```
SQL> DESC student2;
```

| Name | Null?    | Type         |
|------|----------|--------------|
| SID  | NOT NULL | VARCHAR2(4)  |
| NAME |          | VARCHAR2(10) |



- Foreignkeyisusedtoestablishrelationshipbetweentables.
- Foreignkeyisacolumninonetablethatrefers primarykey/unique columns of another or same table.
- Values of foreign key should match with values of primary key/unique or foreign key can be null.
- ForeignkeycolumnallowsnullvaluesunlessitisdeclaredwithNOT NULL.
- ForeginkeycolumnallowsduplicatesunlessitisdeclaredwithUNIQUE
- Bydefaultoracleestablish1:Mrelationshipbetweentwotables.
- To establish 1:1 relationship between twotables declare foreign keywith unique constraint
- Foreignkeycanbedeclaredatcolumnlevelortable level.
- CompositeforeignkeymustrefercompositeprimarykeyorComposite unique key.

**EX: TABLES:STYDENT1&MARKS1**



**CHILDTABLE'MARKS1':**

**ADDINGPRIMARYANDCHECKCONSTRAINT@CREATELEVEL**

```
SQL>CREATETABLEmarks1(
 sidVARCHAR2(4)PRIMARYKEYCHECK(sidLIKE'V%'AND LENGTH(sid)=4),
 marksNUMBER(3));
```

**//ADDINGPRIMARYANDFOREIGNKEY@TABLE/CREATELEVEL**

**SYNTAX: CREATETABLE<tablename>(**

```

col_name1datatype[size],
col_name2datatype[size] ,
:
col_namendatatype[size],
FOREIGN KEY(column_name) REFERENCES <parent_table_name>(column_name));

```

**EX:** SQL> CREATE TABLE marks3(  
sid VARCHAR2(4),  
marks NUMBER(3),  
PRIMARYKEY(sid),  
FOREIGNKEY(sid)REFERENCESstudent1(sid));

SQL>DESC MARKS3;

| Name  | Null?    | Type        |
|-------|----------|-------------|
| SID   | NOT NULL | VARCHAR2(4) |
| MARKS |          | NUMBER(3)   |

Query:ADDCHECKCONSTRAINT@ALTERLEVEL(ONEXISTINGTABLE) MARKS  
IN BETWEEN 0 AND 100.

**SQL>ALTERTABLEmarks3ADDCHECK(marks>0ANDmarks<=100);**

ADDCHECKCONSTRAINT@TABLELEVEL(ATTHEENDOFTABLE  
DEFINITION)  
MARKSINBETWEEN0AND100.

**SQL>CREATETABLEmarks3(sid VARCHAR2(4),  
sec VARCHAR2(2),  
marks NUMBER(3),  
CHECK(marks>0ANDmarks<=100));**

//ADDINGFOREIGNKEY@ALTERLEVEL

SQL>ALTERTABLEmarks1ADDFOREIGNKEY(sid)REFERENCES  
STUDENT1(sid);

SQL>desc marks1;

| Name  | Null?    | Type        |
|-------|----------|-------------|
| SID   | NOT NULL | VARCHAR2(4) |
| MARKS |          | NUMBER(3)   |

**Note:-**

- PRIMARYKEYcannotbedroppedifitreferencedbyanyFOREIGNKEYconstraint.
- IfPRIMARYKEYisdroppedwithCASCADEoptionthenalongwithPRIMARYKEYreferencing FOREIGN KEY is also dropped.
- PRIMARYKEYcolumncannotbedroppedifitisreferencedbysomeFOREIGN KEY.
- PRIMARYKEYtablecannotbedroppedifitisreferencedbysomeFOREIGNKEY.

→ PRIMARY KEY table cannot be truncated if it is referenced by some FOREIGN KEY.

**Note:** Once the primary key and foreign key relationship has been created then you can not remove any parent record if the dependent child exists.

## USING ON DELETE CASCADE

By using this clause you can remove the parent record even if child exists. Because when ever you remove parent record oracle automatically removes all its dependent records from child table, if this clause is present while creating foreign key constraint.

**Ex: Consider two tables dept(parent) and emp(child) tables.**

### TABLE LEVEL

SQL> create table emp(empno number(2), ename varchar(10), deptno number(2), primary key(empno), foreign key(deptno) references dept(deptno) on delete cascade); // without label

SQL> create table emp(empno number(2), ename varchar(10), deptno number(2), constraint pk primary key(empno), constraint fk foreign key(deptno) references dept(deptno) on delete cascade); // with label

### ALTER LEVEL

SQL> alter table emp add foreign key(deptno) references dept(deptno) on delete cascade;

SQL> alter table emp add constraint fk foreign key(deptno) references dept(deptno) on delete cascade;

### Enabling/Disabling a Constraint:

If the constraints are present, then for each DML operation constraints are checked by executing certain codes internally. It may slow down the DML operation marginally. For massive DML operations, such as transferring data from one table to another because of the presence of constraint, the speed will be considered slower. To improve the speed in such cases, the following methods are adopted:

- Disable constraint
- Performing the DML operation
- Enable constraint

### Disabling Constraint:-

#### Syntax :-

ALTER TABLE <tablename> DISABLE CONSTRAINT <constraint\_name>;

#### Example :-

SQL> ALTER TABLE student1 DISABLE CONSTRAINT ck ;

SQL> ALTER TABLE mark1 DISABLE PRIMARY KEY CASCADE;

**NOTE:-**

If constraint is disabled with CASCADE then PK is disabled with FK.

**Enabling Constraint:-****Syntax :-**

**ALTER TABLE <tablename> ENABLE CONSTRAINT <name>**

**Example:-**

**SQL> ALTER TABLE student1 ENABLE CONSTRAINT ck;**

**III.SET OPERATIONS IN SQL**

SQL supports few Set operations to be performed on table data. These are used to get meaningful results from data, under different special conditions. The SET operators combine the results of two or more component queries into one result. Queries containing SET operators are called Compound Queries.

The number of columns and data types of the columns being selected must be identical in all the SELECT statements used in the query. The names of the columns need not be identical.

All SET operators have equal precedence. If a SQL statement contains multiple SET operators, the Oracle server evaluates them from left (top) to right (bottom) if no parentheses explicitly specify another order.

**Introduction**

SQL set operators allow combine results from two or more SELECT statements. At first sight this looks similar to SQL joins although there is a big difference. SQL joins tend to combine columns i.e. with each additionally joined table it is possible to select more and more columns. SQL set operators on the other hand combine rows from different queries with strong preconditions .

- Retrieve the same number of columns and
- The data types of corresponding columns in each involved SELECT must be compatible (either the same or with possibility implicitly convert to the data types of the first SELECT statement).

**Set operator types**

According to SQL Standard there are following Set operator types:

- UNION---returns all rows selected by either query. To return all rows from multiple tables and eliminates any duplicate rows.
- UNIONALL--returns all rows from multiple tables including duplicates.
- INTERSECT--returns all rows common to multiple queries.
- MINUS---returns rows from the first query that are not present in second query.

Note: Whenever these operators used select statement must have

- Equal no. of columns.
- Similar data type columns.

### **Syntax:-**

**SELECT statement1**

**UNION/UNIONALL/INTERSECT/MINUS**

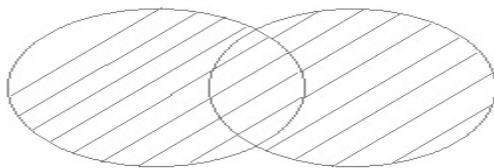
**SELECT statement2; Rules**

**∴**

- 1 No of columns returned by first query must be equal to no of columns returned by second query
- 2 Corresponding columns data type must be same.

## **1. UNION**

- UNION operator combines data returned by two SELECT statement.
- Eliminates duplicates.
- Sorts result.
- This will combine the records of multiple tables having the same structure.



### **Example:-**

- 1 **SQL>SELECT job FROM Emp WHERE deptno=10  
UNION  
SELECT job FROM Emp WHERE deptno=20;**
- 2 **SQL>SELECT job, sal FROM Emp WHERE deptno=10**

## UNION

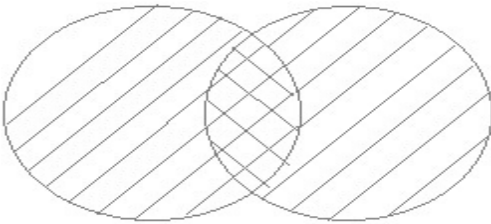
**SELECT** job,sal **FROM** emp **WHERE** deptno=20 **ORDER BY** sal;

**NOTE:-** ORDER BY clause must be used with last query.

3 SQL> select \* from student1 **union** select \* from student2;

## 2. UNION ALL

This will combine the records of multiple tables having the same structure but including duplicates. It is similar to UNION but it includes duplicates.



**Example:-**

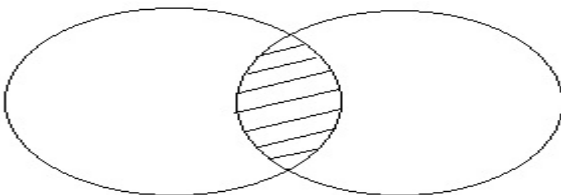
SQL> **SELECT** job **FROM** emp **WHERE** deptno=10  
UNION ALL  
SELECT job **FROM** emp **WHERE** deptno=20;

SQL> select \* from student1 **unionall** select \* from student2;

## 3. INTERSECT

This will give the common records of multiple tables having the same structure.

INTERSECT operator returns common values from the result of two SELECT statements.



**Example:-**

Display common jobs belong to 10<sup>th</sup> and 20<sup>th</sup> departments?

EX1: SQL> **SELECT** job **FROM** emp **WHERE** deptno=10

## INTERSECT

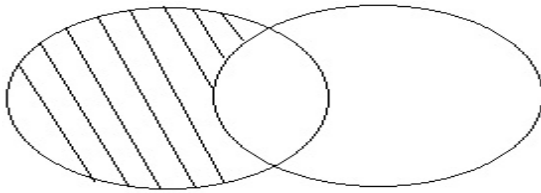
```
SELECT job FROM emp WHERE deptno=20;
```

**EX2:SQL>select\*fromstudent1intersectselect\*fromstudent2;**

## 4. MINUS

This will give the records of a table whose records are not in other tables having the same structure.

MINUS operator returns values present in the result of first SELECT statement and not present in the result of second SELECT statement.



### Example:-

Display jobs in 10<sup>th</sup> dept and not in 20<sup>th</sup> dept?

```
EX1:SQL>SELECT job FROM emp WHERE deptno=10
MINUS
SELECT job FROM emp WHERE deptno=20;
```

**Ex2:SQL>select\*fromstudent1minusselect\*fromstudent2;**

### UNION vs JOIN:-

#### UNION

Union combines data

Union is performed on similar structures  
on

#### JOIN

Join relates data

Join can be performed also on dissimilar  
structures also

## V. SQL JOINS

A SQL JOIN is an Operation, used to retrieve data from multiple tables. It is performed whenever two or more tables are joined in a SQL statement. so, SQL Join clause is used to combine records from two or more tables in a database. A JOIN is a means for combining fields from two tables by using values common to each. Several operators can be used to join tables,

such as =, <>, <=, >=, !=, BETWEEN, LIKE, and NOT; these all to be used to join tables. However, the most common operator is the equal symbol.

### **SQL Join Types:**

There are different types of joins available in SQL:

- **INNER JOIN:** Returns rows when there is a match in both tables.
- **OUTER JOIN:** Returns all rows even there is a match or no match in tables.
  - **LEFT JOIN/LEFT OUTER JOIN:** Returns all rows from the left table, even if there are no matches in the right table.
  - **RIGHT JOIN/RIGHT OUTER JOIN:** Returns all rows from the right table, even if there are no matches in the left table.
  - **FULL JOIN/FULL OUTER JOIN :** Returns rows when there is a match in one of the tables.
- **SELF JOIN:** It is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.
- **CARTESIAN JOIN or CROSS JOIN:** It returns the Cartesian product of the set of records from the two or more joined tables.

Based on **Operators**, **The Join** can be classified as

- **Inner join or Equi Join**
- **Non-Equi Join**
- **NATURAL JOIN:** It is performed only when common column name is same. In this, no need to specify join condition explicitly, ORACLE automatically performs join operation on the column with same name.

#### **1. SQL INNER JOIN (simple join)**

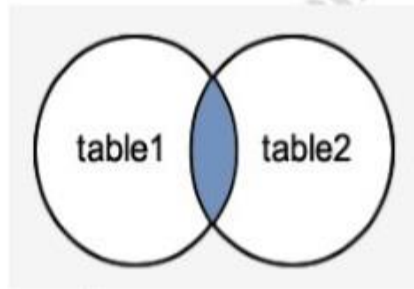
It is the most common type of SQL join. SQL INNER JOINs return all rows from multiple tables where the join condition is met.

#### **Syntax**

**SELECT columns FROM table1 INNER JOIN table2 ON table1.column = Table2.column; Visual**

#### **Illustration**

In this visual diagram, the SQL INNER JOIN returns the shaded area:



The SQL INNER JOIN would return the records where table1 and table2 intersect. Let's look at some data to explain how the INNER JOINS work with example.

We have a table called SUPPLIERS with two fields (supplier\_id and supplier\_name). It contains the following data:

| supplier_id | supplier_name  |
|-------------|----------------|
| 10000       | ibm            |
| 10001       | hewlettpackard |
| 10002       | microsoft      |
| 10003       | nvidia         |

We have another table called ORDERS with three fields (order\_id, supplier\_id, and order\_date).

It contains the following data:

| order_id | supplier_id | order_date |
|----------|-------------|------------|
| 500125   | 10000       | 2003/05/12 |
| 500126   | 10001       | 2003/05/13 |
| 500127   | 10004       | 2003/05/14 |

#### Example of INNER JOIN:

Q: List supplier id, name and order id of supplier.

```
SELECT s.supplier_id, s.supplier_name, od.order_date FROM suppliers s INNER JOIN
orders od ON s.supplier_id = od.supplier_id;
```

This SQL INNER JOIN example would return all rows from the suppliers and orders tables where there is a matching supplier\_id value in both the suppliers and orders tables. Our result set would look like this:

| supplier_id | name           | order_date |
|-------------|----------------|------------|
| 10000       | ibm            | 2003/05/12 |
| 10001       | hewlettpackard | 2003/05/13 |

TherowsforMicrosoftandNVIDIAfromthesuppliortablewouldbeomitted,sincethe supplier\_id's 10002 and 10003 do not exist in both tables.

Therowfor500127(order\_id)fromtheorderstablewouldbeomitted,sincethe supplier\_id 10004 does not exist in the suppliers table.

## **2. OUTER JOIN:**

Inner/Equijoinreturnsonlymatchingrecordsfromboththetablesbutnotunmatchedrecord, An Outer join retrieves all row even when one of the column metjoin condition.

### **Types of outer join:**

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

#### **LEFT OUTER JOIN**

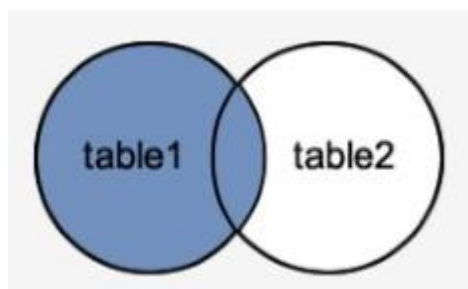
This type of join returns all rows from the LEFT-hand table specified in the ON condition and only those rows from the other table where the joined fields are equal (join condition is met).

### **Syntax**

```
SELECT columns FROM table1 LEFT [OUTER] JOIN table2
ON table1.column = table2.column;
```

### **Visual Illustration**

In this visual diagram, the SQL LEFT OUTER JOIN returns the shaded area:



The SQL LEFT OUTER JOIN would return all records from table1 and only those records from table2 that intersect with table1.

### **Example**

```
SELECT suppliers.supplier_id, suppliers.supplier_name, orders.order_date FROM
suppliers LEFT OUTER JOIN orders ON suppliers.supplier_id = orders.supplier_id;
```

This LEFT OUTER JOIN example would return all rows from the supplier table and only those rows from the orders table where the joined fields are equal.

| supplier_id | supplier_name  | order_date |
|-------------|----------------|------------|
| 10000       | ibm            | 2003/05/12 |
| 10001       | hewlettpackard | 2003/05/13 |
| 10002       | microsoft      | <null>     |
| 10003       | nvidia         | <null>     |

TherowsforMicrosoftandNVIDIAwouldbeincludedbecauseaLEFTOUTERJOIN wasused.However,youwillnoticethattheorder\_datefieldforthoserecordscontainsa <null> value.

## 2.2SQLRIGHTOUTERJOIN

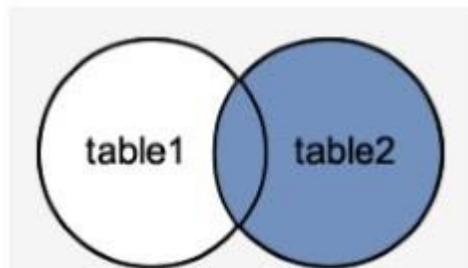
This type of join returns all rows from the RIGHT-hand table specified in the ON conditionandonlythoserowsfromtheothertablewherethejoinedfieldsareequal(join condition is met).

### Syntax

SELECTcolumnsFROMtable1RIGHT[OUTER]JOINTable2ONtable1.column= table2.column;

Insome databases,theRIGHTOUTERJOINkeywordsarereplacedwithRIGHTJOIN. Visual Illustration

Inthisvisual diagram,theSQLRIGHTOUTERJOINreturnstheshaded area:



TheSQLRIGHTOUTERJOINwouldreturntheallrecordsfromtable2andonlythose records from table1 that intersect with table2.

### Example

SELECTorders.order\_id,orders.order\_date,suppliers.supplier\_nameFROMsuppliers  
RIGHT OUTER JOIN orders ON suppliers.supplier\_id = orders.supplier\_id;

This RIGHT OUTER JOIN example would return all rows from the order table and only those rows from the suppliers table where the joined fields are equal.

If a supplier\_id value in the order table does not exist in the suppliers table, all fields in the suppliers table will display as <null> in the result set.

| order_id | order_date | supplier_name  |
|----------|------------|----------------|
| -----    | -----      | -----          |
| 500125   | 2013/05/12 | ibm            |
| 500126   | 2013/05/13 | hewlettpackard |
| 500127   | 2013/05/14 | <null>         |

The row for 500127 (order\_id) would be included because a RIGHT OUTER JOIN was used. However, you will notice that the supplier\_name field for that record contains a <null> value.

### SQL FULL OUTER JOIN

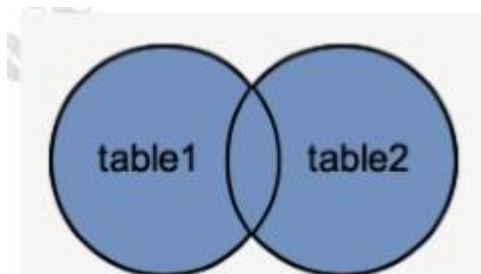
This type of join returns all rows from the LEFT-hand table and RIGHT-hand table with nulls in place where the join condition is not met.

#### Syntax

SELECT columns FROM table1 FULL [OUTER] JOIN table2 ON table1.column = table2.column; In some databases, the FULL OUTER JOIN keywords are replaced with FULL JOIN.

#### Visual Illustration

In this visual diagram, the SQL FULL OUTER JOIN returns the shaded area:



The SQL FULL OUTER JOIN would return all records from both table1 and table2. Example  
Here is an example of a SQL FULL OUTER JOIN:

Query: Find supplier id, supplier name and order date of suppliers who have ordered.

**SELECT suppliers.supplier\_id, suppliers.supplier\_name, orders.order\_date FROM suppliersFULLOUTERJOINordersONsuppliers.supplier\_id=orders.supplier\_id;**

ThisFULLOUTERJOINexamplewouldreturnallrowsfromthesupplierstableandall rows from the orders table and whenever the join condition is not met, <null> would be extended to those fields in the result set.

If a supplier\_id value in the suppliers table does not exist in the orders table, all fields in theorderstablewilldisplayas<null>intheresultset.Ifasupplier\_idvalueintheorders table does not exist in the suppliers table, all fields in the suppliers table will display as <null>intheresultset.

| supplier_id | supplier_name  | order_date |
|-------------|----------------|------------|
| -----       | -----          | -----      |
| 10000       | ibm            | 2013/05/12 |
| 10001       | hewlettpackard | 2013/05/13 |
| 10002       | microsoft      | <null>     |
| 10003       | nvidia         | <null>     |
| <null>      | <null>         | 2013/05/14 |

The rows for Microsoft and NVIDIA would be included because a FULLOUTERJOIN was used. However, you will notice that the order\_date field for those records contains a <null> value.

The row for supplier\_id 10004 would be also included because a FULLOUTERJOIN was used. However, you will notice that the supplier\_id and supplier\_name field for those records contain a <null> value.

### **Equi join:**

When the Join Condition is based on EQUALITY(=) operator, the join is said to be an Equi join. It is also called as **Inner Join**.

### **Syntax**

**Select col1,col2,...From<table1>,<table2>Where<join condition with '='>.**

**Ex. Query : Finds supplierid, suppliername and order date of suppliers who have ordered. select s.supplierid, s.suppliername ,o.order\_date from suppliers s, orders o where s.supplierid =o.supplierid.**

| supplier_id | name           | order_date |
|-------------|----------------|------------|
| 10000       | ibm            | 2003/05/12 |
| 10001       | hewlettpackard | 2003/05/13 |

### NonEquiJoin:-

When the join condition is based on other than equality operator, the join is said to be a Non-Equi join.

### Syntax:-

Select col1,col2,.....

From <table1>,<table2>

Where <joincondition>[AND <joincond>AND <cond> ----- ]

→ In NON- EQUI JOIN, JOINCOND is not based on = operator. It is based on other than = operator, usually BETWEEN or > or < operators.

**Query: Find supplier id, supplier name and order date in between 50025 and 500127.**

```
sql>select s.supplier_id,s.supplier_name,o.order_date from suppliers s,orders o where o.order_id
between 500125 and 500127;
```

SUPPLIER\_ID SUPPLIER\_N ORDER\_DAT

```

10000 ibm 12-may-03
10000 ibm 13-may-03
10000 ibm 14-may-03
10001 hewlett 12-may-03
10001 hewlett 13-may-03
10001 hewlett 14-may-03
10002 microsoft 12-may-03
10002 microsoft 13-may-03
10002 microsoft 14-may-03
10003 nvidia 12-may-03
10003 nvidia 13-may-03
10003 nvidia 14-may-03
```

**Query: Find supplier id, supplier name and order date above 500126.**

```
sql>select s.supplier_id,s.supplier_name,o.order_date from suppliers s,orders o where o.order_id
>500126;
```

SUPPLIER\_ID SUPPLIER\_N ORDER\_DAT

```

10000 ibm 14-may-03
10001 hewlett 14-may-03
10002 microsoft 14-may-03
10003 nvidia 14-may-03
```

### **SelfJoin:-**

Joining a table to itself is called SelfJoin.

- SelfJoin is performed when tables having self-referential integrity.
- To perform SelfJoin, a table must be listed twice with different aliases.
- SelfJoin is EquiJoin within the table.

It is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.

### **Syntax:**

(Here T1 and T2 refers same table)

**SELECT <collist> From Table1 T1, Table1 T2 Where  
T1.Column1=T2.Column2;**

### **Example:**

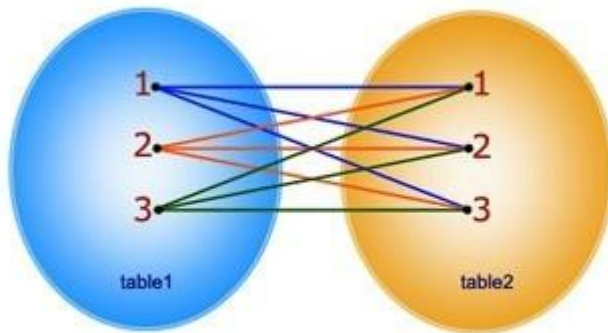
select s1.supplier\_id, s1.supplier\_name, s2.supplier\_id from suppliers s1, suppliers s2 where  
s1.supplier\_id=s2.supplier\_id;

| supplier_id | supplier_name  | supplier_id |
|-------------|----------------|-------------|
| 10000       | ibm            | 10000       |
| 10001       | hewlettpackard | 10001       |
| 10002       | microsoft      | 10002       |
| 10003       | nvidia         | 10003       |

### **CROSSJOIN:**

It returns the Cartesian product of the sets of records from the two or more joined tables. In Cartesian product, each element of one set is combined with every element of another set to form the resultant elements of Cartesian product.

Syntax: **SELECT \* FROM <tablename1> CROSSJOIN <tablename2>**



In CROSS JOIN, each row from 1st table joins with all the rows of another table.  
If 1st table contain x rows and y rows in 2nd one the result set will be x \* y rows.

- CROSSJOIN returns cross product of two tables.
- Each record of one table is joined to each and every record of another table.
- If table1 contains 10 records and table2 contains 5 records then CROSS JOIN between table1 and table2 returns 50 records.
- ORACLE performs CROSSJOIN when we submit query without JOIN COND.

Example: sql> SELECT \* FROM suppliers CROSS JOIN orders; supplier\_id  
supplier\_norder\_id supplier\_id order\_dat

|                |        |                |
|----------------|--------|----------------|
| 10000ibm       | 500125 | 1000012-may-03 |
| 10000ibm       | 500126 | 1000113-may-03 |
| 10000ibm       | 500127 | 1000314-may-03 |
| 10001hewlett   | 500125 | 1000012-may-03 |
| 10001hewlett   | 500126 | 1000113-may-03 |
| 10001hewlett   | 500127 | 1000314-may-03 |
| 10002microsoft | 500125 | 1000012-may-03 |
| 10002microsoft | 500126 | 1000113-may-03 |
| 10002microsoft | 500127 | 1000314-may-03 |
| 10003nvidia    | 500125 | 1000012-may-03 |
| 10003nvidia    | 500126 | 1000113-may-03 |

#### NATURAL JOIN:

- NATURAL JOIN is possible in ANSI SQL/92 standard.
- NATURAL JOIN is similar to EQUI JOIN.
- NATURAL JOIN is performed only when common column name is same.
- In NATURAL JOIN, no need to specify join condition explicitly, ORACLE automatically performs join operation on the column with same name.

Syntax:SELECT<columnlist>FROMtable1NATURALJOINtable2;

**Example:(Sailors table)**

**SELECTsid,sname,sidFROMsailorsNATURALJOINreserves;//bothtableshave  
samecolumn name.**

| SID SNAME | SID |
|-----------|-----|
| -----     |     |
| 22 DUSTIN | 22  |
| 22 DUSTIN | 22  |
| 22 DUSTIN | 22  |
| 22 DUSTIN | 22  |
| 31 LUBBER | 31  |
| 31 LUBBER | 31  |
| 31 LUBBER | 31  |
| 64 HORTIO | 64  |
| 64 HORTIO | 64  |
| 74 HORTIO | 74  |

**VI. VIEWS**

A view in SQL is a logical subset of data from one or more tables. View is used to restrict data access. Data abstraction is usually required after a table is created and populated with data. Data held by some tables might require restricted access to prevent all users from accessing all columns of a table, for data security reasons. Such a security issue can be solved by creating several tables with appropriate columns and assigning specific users to each such table, as required. This answers data security requirements very well but gives rise to a great deal of redundant data being resident in tables, in the database. To reduce redundant data to the minimum possible, Oracle provides Virtual tables which are Views.

**View Definition:-**

→ A View is a virtual table based on the result returned by a SELECT query.

→ The most basic purpose of a view is restricting access to specific column/rows from a table thus allowing different users to see only certain rows or columns of a table.

**Composition Of View:-**

→ A view is composed of rows and columns, very similar to table. The fields in a view are fields from one or more database tables in the database.

→ SQL functions, WHERE clauses and JOIN statements can be applied to a view in the same manner as they are applied to a table.

**View storage:-**

→ Oracle does not store the view data. It recreates the data, using the view's SELECT statement, every time a user queries a view.

→ A view is stored only as a definition in Oracle's system catalog.

→ When a reference is made to a view, its definition is scanned, the base table is opened and the view is created on top of the base table. This, therefore, means that a view never holds data, until a specific call to the view is made. This reduces redundant data on the HDD to a very large extent.

### **Advantages Of View:-**

Security:- Each user can be given permission to access only a set of views that contain specific data.

Query simplicity:- A view can draw from several different tables and present it as a single table turning multiple table queries into single table queries against the view.

Data Integrity:- If data is accessed and entered through a view, the DBMS can automatically check the data to ensure that it meets specified integrity constraints.

### **Disadvantage of View:-**

Performance:- Views only create the appearance of the table but the RDBMS must still translate queries against the views into the queries against the underlined source tables. If the view is defined on a complex multiple table query then even a simple query against the view becomes a complicated join and takes a long time to execute.

### **Types of Views:-**

- Simple Views
- Complex Views

### **Simple Views:-**

a view based on a single table is called a simple view.

### **Syntax:-**

```
CREATE VIEW <ViewName> AS
SELECT <ColumnName1>, <ColumnName2> ..
FROM <TableName>
[WHERE <COND>]
[WITH CHECK OPTION]
[WITH READ ONLY]
```

### **Example :-**

```
SQL> CREATE VIEW Wemp_v
AS
SELECT empno, ename, sal FROM emp;
```

→ Views can also be used for manipulating the data that is available in the base tables [i.e. the user can perform the Insert, Update and Delete operations through view.

→ Views on which data manipulation can be done are called Updateable Views.

→ If an Insert, Update or Delete SQL statement is fired on a view, modifications to data in the view are passed to the underlying base table.

→ For a view to be updatable, it should meet the following criteria:

→ Views defined from a single table.

→ If the user wants to INSERT records with the help of a view, then the PRIMARY KEY column(s) and all the NOT NULL columns must be included in the view.

### **Inserting record through view:-**

```
SQL>INSERT INTO emp_v VALUES(1,'A',5000,200);
```

### **Updating record through view:-**

*Updating a View:*

A view can be updated under certain conditions:

- The SELECT clause may not contain the keyword DISTINCT.
- The SELECT clause may not contain summary functions.
- The SELECT clause may not contain set functions.
- The SELECT clause may not contain set operators.
- The SELECT clause may not contain an ORDER BY clause.
- The FROM clause may not contain multiple tables.
- The WHERE clause may not contain subqueries.
- The query may not contain GROUP BY or HAVING.
- Calculated columns may not be updated.
- All NOT NULL columns from the base table must be included in the view in order for the INSERT query to function.

So if a view satisfies all the above-mentioned rules then you can update a view.

**EX:** SQL>UPDATE emp\_v SET sal=2000 WHERE empno=1;

### **Deleting record through view:-**

```
SQL>DELETE FROM emp_v WHERE empno=1;
```

### **With Check Option :-**

If a VIEW is created with WITH CHECK OPTION then any DML operation through that view violates where condition then that DML operation returns error.

### **Example :-**

```
SQL>CREATE VIEW V2
AS
SELECT empno,ename,sal,deptno FROM emp WHERE
deptno=10
WITH CHECK OPTION ;
```

Then insert the record into emp table through view V2

**SQL>INSERT INTO V2 VALUES(2323,'RAJU',4000,20);**

The above INSERT returns error because DML operation violating WHERE clause.

### **Complex Views:-**

A view is said to be a complex view

→ If it is based on more than one table

→ Query contains

AGGREGATE functions

DISTINCT clause

GROUP BY clause

HAVING clause

Sub-queries

Constants

Strings or Values Expressions

UNION, INTERSECT, MINUS operators.

#### **Example 1:-**

SQL>CREATE VIEW V3

AS

SELECT E.empno, E.ename, E.sal, D.dname, D.loc

FROM emp E JOIN dept D

USING(deptno);

### **NON-UPDATABLE VIEWS:**

We cannot perform insert or update or delete operations on a base table through complex views. Complex views are not updatable views.

#### **Example 2:-**

SQL>CREATE VIEW V2

AS

SELECT deptno, SUM(sal) AS sumsal

FROM EMP

GROUP BY deptno;

### **Destroying a View:-**

The DROP VIEW command is used to destroy a view from the database.

#### **Syntax:-**

DROP VIEW <viewName>

#### **Example :-**

SQL>DROP VIEW emp\_v;

DIFFERENCES BETWEEN SIMPLE AND COMPLEX VIEWS:

| SIMPLE                          | COMPLEX                         |
|---------------------------------|---------------------------------|
| Created from one table          | Created from one or more tables |
| Does not contain functions      | Contains functions              |
| Does not contain groups of data | Contains groups of data         |

## MATERIALIZED VIEW: @DATA WAREHOUSE SYSTEMS

A materialized view in Oracle is a database object that contains the results of a query. They are local copies of data located remotely, or are used to create summary tables based on aggregations of a table's data. Materialized views, which store data based on remote tables are also, known as snapshots.

A materialized view can query tables, views, and other materialized views. Collectively these are called master tables (a replication term) or detail tables (a data warehouse term).

For replication purposes, materialized views allow you to maintain copies of remote data on your local node. These copies are read-only. If you want to update the local copies, you have to use the Advanced Replication feature. You can select data from a materialized view as you would from a table or view.

For data warehousing purposes, the materialized views commonly created are aggregate views, single-table aggregate views, and join views.

In replication environments, the materialized views commonly created are primary key, rowid, and subquery materialized views.

### SYNTAX:

```
CREATE MATERIALIZED VIEW view-name
BUILD [IMMEDIATE | DEFERRED]
REFRESH [FAST | COMPLETE | FORCE]
ON [COMMIT | DEMAND]
[[ENABLE|DISABLE] QUERY REWRITE]
[ON PREBUILT TABLE]
AS
SELECT COLUMN_LIST FROM TABLE_NAME;
```

The BUILD clause options are shown below.

- IMMEDIATE: The materialized view is populated immediately.
- DEFERRED: The materialized view is populated on the first requested refresh.

The following refresh types are available.

- **FAST**: A fast refresh is attempted. If materialized view logs are not present against the source tables in advance, the creation fails.
- **COMPLETE**: The table segments supporting the materialized view are truncated and repopulated completely using the associated query.
- **FORCE**: A fast refresh is attempted. If one is not possible, a complete refresh is performed.

A refresh can be triggered in one of two ways.

- **ON COMMIT**: The refresh is triggered by a committed data change in one of the dependent tables.
- **ON DEMAND**: The refresh is initiated by a manual request or a scheduled task.

The **QUERY REWRITE** clause tells the optimizer if the materialized view should be considered for query rewrite operations. An example of the query rewrite functionality is shown below.

The **ON PREBUILT TABLE** clause tells the database to use an existing table segment, which must have the same name as the materialized view and support the same column structure as the query.

Example:

The following statement creates the rowid materialized view on table emp located on a remote database:

```
SQL> CREATE MATERIALIZED VIEW mv_emp_rowid
 REFRESH WITH ROWID
 AS SELECT * FROM emp@remote_db;
```

Materialized view log created.

## **VII. ORDERING**

USING “**ORDER BY**” clause:

This will be used to ordering the column's data (ascending or descending).

Syntax 1: (simple form)

**select \* from <table\_name> order by <col> desc;**

Note: By default Oracle will use ascending order.

If you want output in descending order, you have to use the desc keyword after the column. Ex:

SQL> select \* from student order by no;                      SQL> select \* from student order by no desc;

SQL> select \* from student order by no;    SQL> select \* from student order by no desc;

| NO  | NAME   | MARKS | NO  | NAME   | MARKS |
|-----|--------|-------|-----|--------|-------|
| --- | -----  | ----- | --- | -----  | ----- |
| 1   | Sudha  | 100   | 6   | Rattu  |       |
| 1   | Jagan  | 300   | 5   | Visu   |       |
| 2   | Saketh | 200   | 4   | Madhu  |       |
| 2   | Naren  | 400   | 3   | Ramesh |       |
| 3   | Ramesh |       | 2   | Saketh | 200   |
| 4   | Madhu  |       | 2   | Naren  | 400   |
| 5   | Viru   |       | 1   | Sudha  | 100   |
| 6   | Rattu  |       | 1   | Jagan  | 300   |

The order of rows returned in a query result is undefined. The ORDER BY clause can be used to sort the rows. If you use the ORDER BY clause, it must be the last clause of the SQL statement. You can specify an expression, or an alias, or column position in ORDER BY clause.

#### Syntax2:(complex form)

**SELECT** *expr* **FROM** *table*  
**[WHERE** *condition(s)*  
**[ORDER BY {column,expr}{ASC | DESC}];**

In the syntax,

**ORDER BY:** specifies the order in which the retrieved rows are displayed. orders  
the rows in ascending order ( default order)  
order the rows in descending order

#### Ordering of Data:-

- Numeric values are displayed with the lowest values first      for example    1–999.
- Date values are displayed with the earliest value first for example 01-JAN-92 before 01- JAN-95.
- Character values are displayed in alphabetical order — for example, A first and Z last.
- Null values are displayed last for ascending sequences and first for descending sequences.

#### Examples:-

Arrange employee records in ascending order of their sal?

**SQL>SELECT\*FROMempORDER BYsal;**

Arrangeemployee records in descending order of their sal?

**SQL>SELECT\*FROMempORDER BYsalDESC;**

Display employee records working for 10<sup>th</sup> dept and arrange the result in ascending order of their sal ?

**SQL>SELECT\*FROMempWHEREdeptno=10ORDER BYsal;**

Arrange employee records in ascending order of their deptno and within dept arrange records in descending order of their sal ?

**SQL>SELECT\*FROMempORDER BYdeptno,salDESC;**

In ORDER BY clause we can use column name or column position, for example

**SQL>SELECT\*FROMempORDER BY5DESC;**

In the above example records are sorted based on the fifth column in emp table.

Arrange employee records in descending order of their comm. If comm. is null then arrange those records last ?

**SQL>SELECT\*FROMempORDER BYcommDESCNULLSLAST; VIII.GROUP BY**

### **AND HAVING CLAUSE**

#### **GROUP BY clause**

Using group by, we can create groups of related information. Columns used in select must be used with group by, otherwise it was not a group by expression.

**SELECT        [DISTINCT]select-list**

**FROMfrom-list**

**WHERE    qualification**

**GROUP BYgrouping-list**

**HAVING    group-qualification**

- TheselectlistintheSELECTclause contain
  1. Alistofcolumnnames
  2. Alistoftermshavingtheformaggop(aggregateoperators)Every column that appear in (1) must also appear in **grouping-list**
- Theexpressionappearinginthe **group-qualification**intheHAVINGclausemusthavea single value per group.

Ex:SQL>selectdeptno,sum(sal)fromempgroupbydeptno;

SQL> select deptno, sum(sal) from emp group by deptno;

DEPTNO SUM(SAL)

-----

10 8750

20 10875

30 9400

SQL>selectdeptno,job,sum(sal)fromempgroupbydeptno,job;

Sql>Find the age of the youngest sailor for each rating level.

SQL>Selects.rating,MIN(s.age)fromsailorssGROUPBYs.rating;

Findtheageoftheyoungestsailorwhoiseligibletovoteforeachratinglevelwithatleasttwo such sailors ?

SQL>selects.rating,MIN(s.age)asminagefromsailorsswheres.age>=18

GROUP BY s.rating

HAVINGCOUNT(\*)>1;

Foreachredboatfindthenumberofreservationsforthisboat?

SQL>Selectb.bid,COUNT(\*)ASreservationcountfromboatsb,

reserves r where r.bid=b.bid and b.color='red'

GROUPBYb.bid;

Findtheaverageage ofsailorsforeachratinglevelthathasatleasttwosailors?

SQL>Selects.rating,AVG(s.age)ASavgagefromsailorss

GROUPBYs.rating

HAVINGCOUNT(\*)>1;

## IX.AGGREGATION

Itisagroupoperation, whichwillbeworksonallrecordsofatable. Todothis,Group functions required toprocess group of rows and Returns one value from that group.

→ These functions are also called **AGGREGATE** functions or **GROUP** functions

- **Aggregate functions-max(),min(),sum(),avg(),count(),count(\*)**.

Group functions will be applied on all the rows but produce single output.

a) SUM

This will give the sum of the values of the specified column. Syntax:

sum (column)

Ex: SQL> select sum(sal) from emp;

b) AVG

This will give the average of the values of the specified column. Syntax:

avg (column)

Ex: SQL> select avg(sal) from emp;

c) MAX

This will give the maximum of the values of the specified column. Syntax:

max (column)

Ex: SQL> select max(sal) from emp;

d) MIN

This will give the minimum of the values of the specified column. Syntax:

min (column)

Ex: SQL> select min(sal) from emp;

e) COUNT

This will give the count of the values of the specified column. Syntax:

count (column)

Ex: SQL> select count(sal), count(\*) from emp;

## **X.SUB QUERIES**

What is subquery in SQL?

A subquery is a SQL query nested inside a larger query.

- A subquery may occur in:
  - A SELECT clause
  - A FROM clause

- AWHERE clause

- The subquery can be nested inside a SELECT, INSERT, UPDATE, or DELETE statement or inside another subquery.
- A subquery is usually added within the WHERE clause of another SQL SELECT statement.
- You can use the comparison operators, such as >, <, or =. The comparison operator can also be a multiple-row operator, such as IN, ANY, or ALL.
- A subquery is also called an inner query or inner select, while the statement containing a subquery is also called an outer query or outer select.
- The inner query executes first before its parent query so that the results of the inner query can be passed to the outer query.

You can use a subquery in a SELECT, INSERT, DELETE, or UPDATE statement to perform the following tasks :

- Compare an expression to the result of the query.
- Determine if an expression is included in the results of the query.
- Check whether the query selects any rows.

**Syntax:**

```
SELECT select_list
FROM table
WHERE expr operator
 (SELECT select_list
 FROM table);
```

- The subquery (inner query) executes once before the main query (outer query) executes.
- The main query (outer query) uses the subquery result.

### SQL Subqueries Example:

In this section, you will learn the requirements of using subqueries. We have the following two tables 'student' and 'marks' with common field 'SID'.

**SQL>select \* from student1;**

```
SID NAME

```

v001abhi  
v002abhay  
v003arjun  
v004anand  
**SQL>select\*from marks;**

**SID      TOTALMARKS**

-----

|      |    |
|------|----|
| v001 | 95 |
| v002 | 80 |
| v003 | 74 |
| v004 | 81 |

Nowwewanttowriteaquery**to identify all students who get better marks than that of the student who's StudentID is 'V002'**, but we do not know the marks of 'V002'.

-To solve this problem, we require **two** queries.

One query return the marks (stored in Totalmarks field) of 'V002' and a second query identifies the students who get better marks than the result of the first query.

**SQL>select\*from markswheresid='v002';**

**Query Result:**

**SID      TOTALMARKS**

-----

|             |           |
|-------------|-----------|
| <b>v002</b> | <b>80</b> |
|-------------|-----------|

The result of query is **80**.

-Using the result of this query, here we have written another query to identify the students who get better marks than 80. Here is the query :

**Second query:**

**SQL>selects.sid,s.name,m.totalmarksfrom student1s,marksmwheres.sid=m.sidand m.totalmarks>80;**

**SIDNAME      TOTALMARKS**

-----

|                   |           |
|-------------------|-----------|
| <b>v001abhi</b>   | <b>95</b> |
| <b>v004 anand</b> | <b>81</b> |

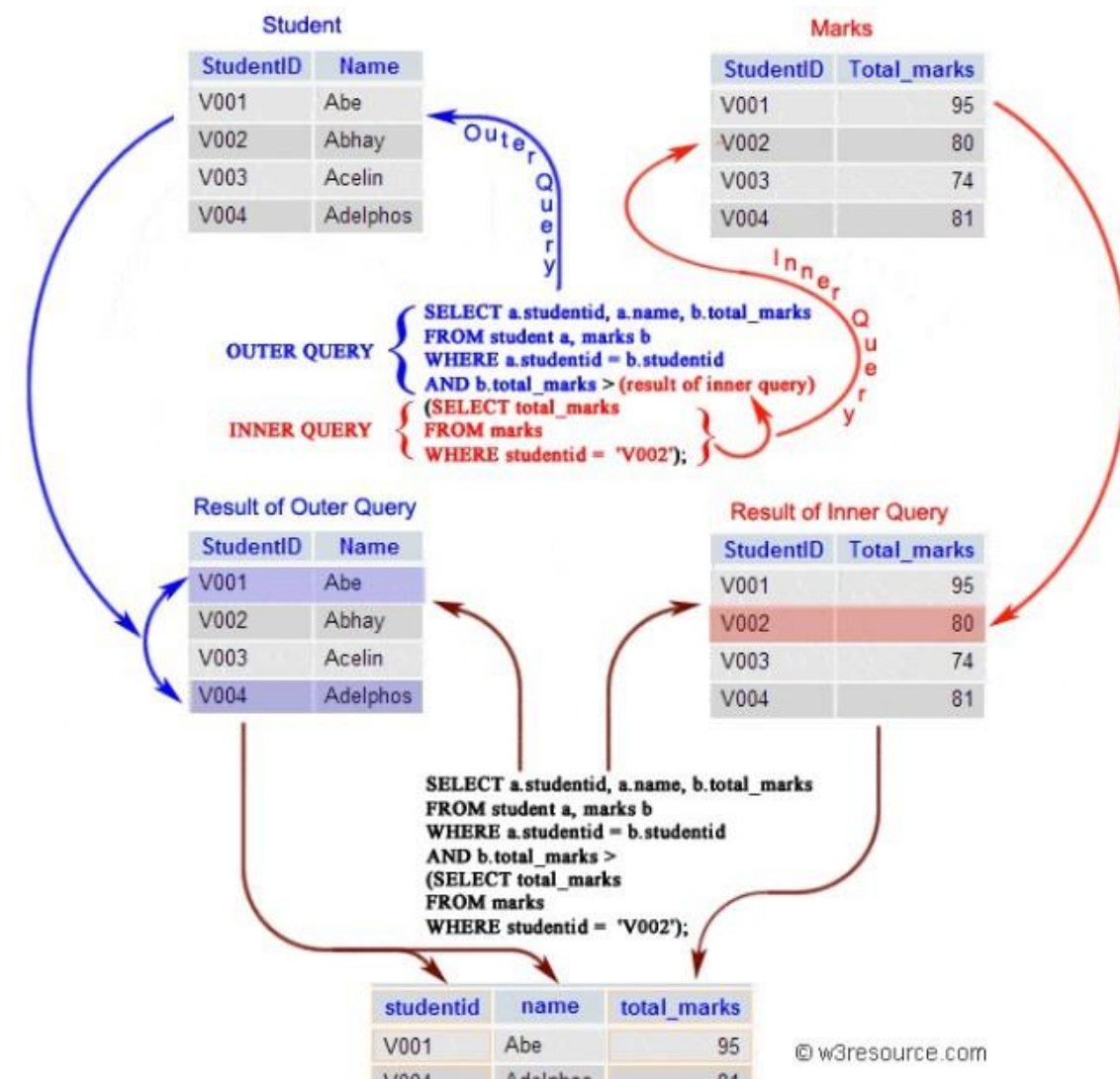
Above two queries identified students who get better number than the student who's StudentID is 'V002' (Abhi).

You can combine the above two queries by placing one query inside the other. The subquery (also called the 'inner query') is the query inside the parentheses. See the following code and query result :

**SQL>select s.sid,s.name,m.total\_marks from student1s,marks m where s.sid=m.sid and m.total\_marks >(select total\_marks from marks where sid='v002');**

**SID NAME TOTAL MARKS**

-----  
v001 abhi 95  
v004 anand 81



## Subqueries: Guidelines

There are some guidelines to consider when using subqueries:

- A subquery must be enclosed in parentheses.
- A subquery must be placed on the right side of the comparison operator.
- Subqueries cannot manipulate their results internally, therefore ORDER BY clause cannot be added into a subquery. You can use an ORDER BY clause in the main SELECT statement (outer query) which will be the last clause.
- Use single-row operators with single-row subqueries.
- If a subquery (inner query) returns a null value to the outer query, the outer query will not return any rows when using certain comparison operators in a WHERE clause.

## Type of Subqueries

- **Singlerow subquery:** Returns zero or one row.
- **Multiplerow subquery:** Returns one or more rows.
- **Multiplecolumn subquery:** Returns one or more columns.
- **Correlated subqueries:** Reference one or more columns in the outer SQL statement. The subquery is known as a correlated subquery because the subquery is related to the outer SQL statement.
- **Nested subqueries:** Subqueries are placed within another subquery.

### 1) SINGLEROW SUBQUERIES:- Returns zero or one row.

If an inner query returns only one row then it is called a singlerow subquery.

### Syntax:-

```
SELECT <collist> FROM <tablename>
 WHERE colname OPERATOR (SELECT statement)
```

Operator can be      <> <=> == <>

### Examples:- (on Emp Table)

Q: Display employee records whose job equals to job of SMITH?

```
SQL> SELECT * FROM Emp
 WHERE job = (SELECT job FROM Emp WHERE ename = 'SMITH');
```

Q: Display employee name earning maximum salary?

```
SQL> SELECT ename FROM Emp
 WHERE sal = (SELECT MAX(sal) FROM Emp);
```

Example2:(onSAILORS\_BOAT\_RESERVATIONDATABASE)

**SQL>SELECT\*FROMSAILORS;**

|  | <b>SIDSNAME</b> | <b>RATING</b> | <b>AGE</b> |
|--|-----------------|---------------|------------|
|--|-----------------|---------------|------------|

|    |         |    |      |
|----|---------|----|------|
| 22 | DUSTIN  | 7  | 45   |
| 29 | BRUTUS  | 1  | 33   |
| 31 | LUBBER  | 8  | 55.5 |
| 32 | CANDY   | 8  | 25.5 |
| 58 | RUSTY   | 10 | 35   |
| 64 | HORATIO | 7  | 35   |
| 71 | ZOBRA   | 10 | 16   |
| 74 | HORATIO | 9  | 35   |
| 85 | ART     | 3  | 25.5 |
| 95 | BOB     | 3  | 63.5 |

**SQL>SELECT\*FROMBOATS;**

|  | <b>BID BNAME</b> | <b>COLOR</b> |
|--|------------------|--------------|
|--|------------------|--------------|

|     |              |       |
|-----|--------------|-------|
| 101 | INTERLAKE    | BLUE  |
| 102 | INTERLAKERED |       |
| 103 | CLIPPER      | GREEN |
| 104 | MARINE       | RED   |

**SQL>SELECT\*FROMRESERVES;**

| <b>SID</b> | <b>BID</b> | <b>DAY</b> |
|------------|------------|------------|
|------------|------------|------------|

|    |     |           |
|----|-----|-----------|
| 22 | 101 | 10-OCT-17 |
| 22 | 102 | 10-OCT-17 |
| 22 | 103 | 10-OCT-17 |
| 22 | 104 | 10-JUL-17 |
| 31 | 102 | 11-OCT-17 |
| 31 | 103 | 11-JUN-17 |
| 31 | 104 | 11-DEC-17 |
| 64 | 101 | 09-MAY-17 |
| 64 | 102 | 09-AUG-17 |
| 74 | 103 | 09-AUG-17 |

**Q:Findthesailor'sIDwhosenameisequalto'DUSTIN'**

**SQL>SELECTSIDFROMSAILORSWHEREID=(SELECTSIDFROM SAILORS  
WHERE SNAME='DUSTIN');**

| <b>SID</b> |
|------------|
|------------|

|    |
|----|
| 22 |
|----|

**Q:Find sailorsrecordswhosenameequalsto'DUSTIN'?**

**SQL>SELECT\*FROMSAILORSWHEREID=(SELECTSIDFROM  
SAILORS WHERE SNAME='DUSTIN');**

| SID | SNAME  | RATING | AGE |
|-----|--------|--------|-----|
| 22  | DUSTIN | 7      | 45  |

**Q:Findtheratingofasailor whosenameis 'DUSTIN'.**

**SQL>SELECTRATINGFROMSAILORSWHEREID=(SELECTSIDFROMSAILORSWHERE SNAME='DUSTIN');**

RATING  
-----  
7

**Q:Findthesailorsrecordswhosesidisgeater than 'dustin'?**

**SQL>SELECT\*FROMSAILORSWHEREID>(SELECTSIDFROM  
SAILORS WHERE SNAME='DUSTIN');**

| SID | SNAME   | RATING | AGE  |
|-----|---------|--------|------|
| 29  | BRUTUS  | 1      | 33   |
| 31  | LUBBER  | 8      | 55.5 |
| 32  | CANDY   | 8      | 25.5 |
| 58  | RUSTY   | 10     | 35   |
| 64  | HORATIO | 7      | 35   |
| 71  | ZOBRA   | 10     | 16   |
| 74  | HORATIO | 9      | 35   |
| 85  | ART     | 3      | 25.5 |
| 95  | BOB     | 3      | 63.5 |

**Q:Findthesailorsrecords,whosesailors'havingmaximumrating.**

**SQL>SELECT\*FROMSAILORSWHEREERATING=(SELECT  
MAX(RATING) FROM SAILORS);**

| SID | SNAME | RATING | AGE |
|-----|-------|--------|-----|
| 58  | RUSTY | 10     | 35  |
| 71  | ZOBRA | 10     | 16  |

**Q:Find therecordsofsailorswhoseratingissameas'DUSTIN'**

**SQL>SELECT\*FROMSAILORSWHEREERATING=(SELECTRATING FROM  
SAILORS WHERE SNAME='DUSTIN');**

| SID | SNAME  | RATING | AGE |
|-----|--------|--------|-----|
| 22  | DUSTIN | 7      | 45  |

64 HORATIO 7 35  
**Q:Findtherecordsofsailorswhoseratingishigherthan ‘DUSTIN’**

**SQL>SELECT\*FROMSAILORSWHERE RATING>(SELECT  
 MAX(RATING) FROM SAILORS WHERE SNAME='DUSTIN');**

| SID | SNAME   | RATING | AGE  |
|-----|---------|--------|------|
| 31  | LUBBER  | 8      | 55.5 |
| 32  | CANDY   | 8      | 25.5 |
| 58  | RUSTY   | 10     | 35   |
| 71  | ZOBRA   | 10     | 16   |
| 74  | HORATIO | 9      | 35   |

#### **MULTIROWSUBQUERIES:**

ifinnerqueryreturnsmorethanonerowthenitiscalledmultirowsubquery.

#### **Syntax:-**

**SQL>SELECT<collist>FROM<tabname>  
 WHEREcolnameOPERATOR(SELECTstatement);**  
 Here,OPERATORmustbeIN,NOTIN, ANY,ALL

#### **INoperator:-**

Totestforvaluesinaspecifiedlistofvalues,useINoperator.TheINoperatorcanbeusedwith any data type. If characters or dates are used in the list, they must be enclosed in single quotation marks (").

#### **Syntax:-**

**IN (V1,V2,V3----- );**

#### **Note:-**

IN ( ... ) is actually translated by Oracle server to a set of 'OR' conditions: a =value1 OR a = value2ORa=value3.SousingIN(...)hasnoperformancebenefits,anditisusedforlogical simplicity.

#### **Example:-**

**Q:DisplayemployeerecordsworkingasCLERKORMANAGER?**

**SQL>SELECT\*FROMempWHEREjobIN('CLERK','MANAGER');**

Q: Find the name of sailors who have reserved boat 103

**SQL> SELECT S.SNAME FROM SAILORS S WHERE S.SID IN (SELECT R.SID FROM RESERVES R WHERE R.BID=103);**

**SNAME**  
-----  
**DUSTIN**  
**LUBBER**  
**HORATIO**

Q: Find the name of sailors who have reserved a red boat

**SQL> 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'));**

**SNAME**  
-----  
**DUSTIN**  
**LUBBER**  
**HORATIO**

Q: 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'));**

**SNAME**  
-----  
**BRUTUS**  
**CANDY**  
**RUSTY**  
**ZOBRA**  
**HORATIO**  
**ART**  
**BOB**

**Using EXISTS Operator:-**

→ EXISTS operator returns TRUE or FALSE.

→ If inner query returns at least one record then EXISTS returns TRUE otherwise returns FALSE.

→ ORACLE recommends EXISTS and NOT EXISTS operators instead of IN and NOT IN. Q: Find the name of sailors who have reserved boat 103

**SQL> SELECT S.SNAME FROM SAILORS S WHERE EXISTS (SELECT \* FROM RESERVES R WHERE R.BID=103 AND R.SID = S.SID);**

**SNAME**

-----

**DUSTIN**

**LUBBER**

**HORATIO**

Q: Find the name of sailors who have not reserved boat 103

**SQL>SELECT S.SNAME FROM SAILORSS WHERE NOT EXISTS (SELECT \* FROM  
RESERVES R WHERE R.BID=103 AND R.SID = S.SID) ;**

**SNAME**

-----

**BRUTUS**

**CANDY**

**RUSTY**

**HORATIO**

**ZOBRA**

**ART**

**BOB**

**ANY Operator:-**

Compares a value to each value in a list or returned by a query. Must be preceded by =, !=, >, <, <=, >=. Evaluate to FALSE if the query returns no rows.

Select employees whose salary is greater than any salesman's salary?

**SQL>SELECT ename FROM Emp**

**WHERE SAL > ANY (SELECT sal FROM Emp WHERE job = 'SALESMAN');**

Q: Find sailors whose rating is better than some sailor called Horatio?

**SQL>SELECT S.SID FROM SAILORSS WHERE S.RATING > ANY (SELECT S2.RATING  
FROM SAILORS S2 WHERE S2.SNAME = 'HORATIO') ;**

**SID**

-----

**58**

**71**

**74**

**31**

**32**

**ALL Operator:-**

Compares a value to every value in a list or returned by a query. Must be preceded by =, !=, >, <, <=, >=. Evaluate to TRUE if the query returns no rows.

**Example:-**

Select employees whose salary is greater than every salesman's salary?

**SQL>SELECTenameFROMemp**

**WHEREsal>ALL(SELECTsalFROMempWHEREjob='SALESMAN');**

Q: FindsailorswhoseratingisbetterthaneversailorcalledHoratio?

**SQL>SELECTS.SIDFROMSAILORSSWHERE.S.RATING>ALL(SELECT S2.RATING  
FROM SAILORS S2 WHERE S2.SNAME='HORATIO') ;**

**SID**

-----

**58**

**71**

**MultiColumnSubqueries:-**

IfinnerqueryreturnsmorethanonecolumnvaluethenitiscalledMULTICOLUMN subquery.

**Example:-**

Displayemployeenamesearningmaximumsalariesintheirdept?

**SQL>SELECTenameFROMemp**

**WHERE(deptno,sal)IN(SELECTdeptno,MAX(sal)**

**FROM emp**

**GROUPBYdeptno);**

**SQL>SELECTSNAMEFROMSAILORSWHERE(RATING,AGE)IN(SELECT  
RATING,MAX(AGE) FROM SAILORS GROUP BY RATING);**

**SNAME**

-----

**DUSTIN**

**BRUTUS**

**LUBBER**

**RUSTY**

**HORATIO**

**BOB**

**SQL>SELECTSID,SNAMEFROMSAILORSWHERE(RATING,AGE)IN(SELECT  
RATING,MAX(AGE) FROM SAILORS GROUP BY RATING);**

**SID SNAME**

-----

**22 DUSTIN**

**29 BRUTUS**

**31 LUBBER**

**58 RUSTY**

**74 HORATIO**

**95 BOB**

**NestedQueries:-**

→AsubqueryembeddedinanothersubqueryiscalledNESTED QUERY.

→ Queries can be nested up to 255 levels.

**Example:-**

Display employee names earning second maximum salary?

```
SQL>SELECT ename FROM Emp
 WHERE sal = (SELECT MAX(sal) FROM Emp
 WHERE sal < (SELECT MAX(sal) FROM Emp)) ;
```

Q: Find the names of sailors who have not reserved a red boat.

```
SELECT S.SNAME FROM SAILOR SS WHERE S.SID NOT IN (SELECT TR.SID FROM
RESERVES R WHERE R.BID IN (SELECT B.BID FROM BOATS B WHERE B.COLOR
='RED'));
```

```
 SNAME

 BRUTUS
 CANDY
 RUSTY
 ZOBRA
 HORATIO
 ART
 BOB
```

**CORRELATED SUBQUERIES:**

In the Co-Related subquery, a parent query will be executed first and based on the output of the outer query, the inner query executes.

If a parent query returns N rows, the inner query executes for N times.

If a subquery references one or more columns of a parent query, it is called a CO-RELATED subquery because it is related to the outer query. This subquery executes once for each and every row of the main query.

**Example1:-**

→ Display employee names earning more than the avg(sal) of their dept?

```
SQL>SELECT ename FROM Emp x
 WHERE sal > (SELECT AVG(sal) FROM Emp
 WHERE deptno = x.deptno);
```

**Example2:** Find sailors whose rating is more than the avg(rating) of their id.

```
SQL>SELECT S.SNAME FROM SAILOR SS WHERE RATING > (SELECT AVG(RATING) FROM SAILORS
WHERE SID = S.SID);
```

norows selected.

#### **SUBQUERIESWITHSET OPERATORS:**

Q1) Findthenamesofsailorswhohavereservedaredoragreenboat?

**SQL>Selects.snamefromsailorss,reservesr,boatsbwheres.sid=r.sidand r.bid=b.bid and (b.color = 'red' or b.color= 'green');**

**Or**

**SQL>Selects.snamefromsailorss,reservesr,boatsbwheres.sid=r.sidand r.bid=b.bid and b.color='red'**

#### **UNION**

**Selects.snamefromsailorss,reservesr,boatsbwheres.sid=r.sidand r.bid=b.bid and b.color='green';**

| SNAME   |
|---------|
| Dustin  |
| Lubber  |
| Horatio |

Q2) Findthenamesofsailorswhohavereservedaredandagreenboat?

**SQL>Selects.snamefromsailorss,reservesr,boatsbwheres.sid=r.sid and r.bid=b.bid and b.color='red'**

#### **INTERSECT**

**Selects.snamefromsailorss,reservesr,boatsbwheres.sid=r.sidand r.bid=b.bid and b.color='green';**

| SNAME  |
|--------|
| Dustin |
| Lubber |

Q3) Find the names of sailors who have reserved a red boat but not a green boat?

**SQL>Select s.sname from sailor s, reserves r, boats b where s.sid=r.sid and  
r.bid=b.bid and b.color='red'**

**MINUS**

**Select s.sname from sailor s, reserves r, boats b where s.sid=r.sid and r.bid=b.bid  
and b.color='green';**

**NOROWSSELECTED**

Q4) Find all IDs of sailors who have a rating of 10 or reserved boat 104?

**SQL>select s.sid from sailor s where s.rating=10**

**UNION**

**Select r.sid from reserves r where r.bid=104;**

| SID |
|-----|
| 22  |
| 31  |
| 58  |
| 71  |

# DATABASEMANAGEMENTSYSTEMS

## UNITIV:SchemaRefinement(Normalization)

### Syllabus:

**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).

### Objectives:

After studying this unit, you will be able to:

- Discuss the different types of anomalies in a database
- State what is functional dependency
- List the different forms of normalization
- Differentiate among different types of normalization

## INTRODUCTION TO SCHEMA REFINEMENT

The Schema Refinement refers to refining the schema by using some technique. The best technique of schema refinement is **decomposition**.

**Normalization** means “split the tables into small tables which will contain less number of attributes in such a way that table design must not contain any problem of inserting, deleting, updating anomalies and guarantees no redundancy”.

**Normalization or 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.

**Redundancy:** refers to repetition of same data or duplicate copies of same data stored in different locations.

**Anomalies:** Anomalies refers to the problems occurred after poorly planned and normalized databases where all the data is stored in one table which is sometimes called a flat file database.

### Anomalies or problems facing without normalization (problems due to redundancy):

Anomalies refers to the problems occurred after poorly planned and unnormalized databases where all the data is stored in one table which is sometimes called a flat file database. Let us consider such type of schema

| SID                         | Sname | CID | Cname | FEE |
|-----------------------------|-------|-----|-------|-----|
| S1                          | A     | C1  | C     | 5k  |
| S2                          | A     | C1  | C     | 5k  |
| S1                          | A     | C2  | C     | 10k |
| S3                          | B     | C2  | C     | 10k |
| S3                          | B     | C2  | JAVA  | 15k |
| <b>Primary Key(SID,CID)</b> |       |     |       |     |

Here all the data is stored in a single table which causes redundancy of data or say anomalies as SID and Sname are repeated once for same CID. Let us discuss anomalies one by one. Due to redundancy of data we may get the following problems, those are-

**1. insertion anomalies:** It may not be possible to store some information unless some other information is stored as well.

**2. redundant storage:** some information is stored repeatedly

**3. update anomalies:** If one copy of redundant data is updated, then inconsistency is created unless all redundant copies of data are updated.

**4. deletion anomalies:** It may not be possible to delete some information without losing some other information as well.

**Problem in updation/ updation anomaly**– If there is updation in the fee from 5000 to 7000, then we have to update FEE column in all the rows, else data will become inconsistent.

| SID | Sname | CID | Cname | FEE |
|-----|-------|-----|-------|-----|
| S1  | A     | C1  | C     | 5k  |
| S2  | A     | C1  | C     | 5k  |
| S1  | A     | C2  | C     | 10k |
| S3  | B     | C2  | C     | 10k |
| S3  | B     | C2  | JAVA  | 15k |

7k  
7k

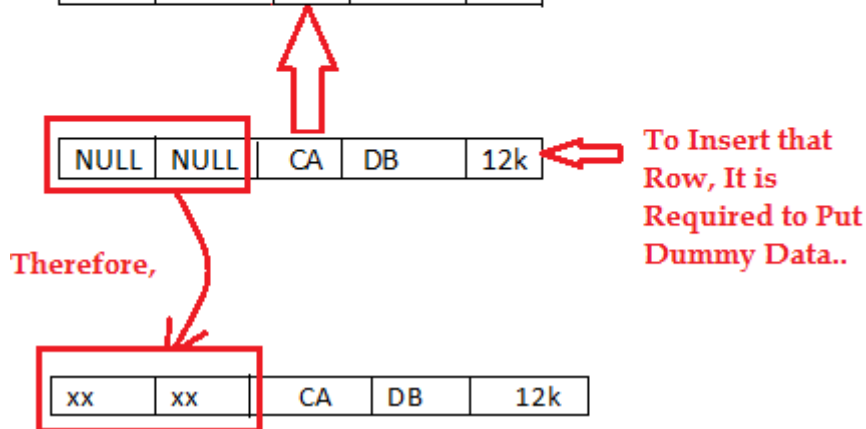
Costly Operation

More IO Cost

**Insertion Anomaly and Deletion Anomaly**- These anomalies exist only due to redundancy, otherwise they do not exist.

**Insertion Anomalies:** New course is introduced C4, But no student is there who is having C4 subject.

| SID | Sname | CID | Cname | FEE |
|-----|-------|-----|-------|-----|
| S1  | A     | C1  | C     | 5k  |
| S2  | A     | C1  | C     | 5k  |
| S1  | A     | C2  | C     | 10k |
| S3  | B     | C2  | C     | 10k |
| S3  | B     | C2  | JAVA  | 15k |



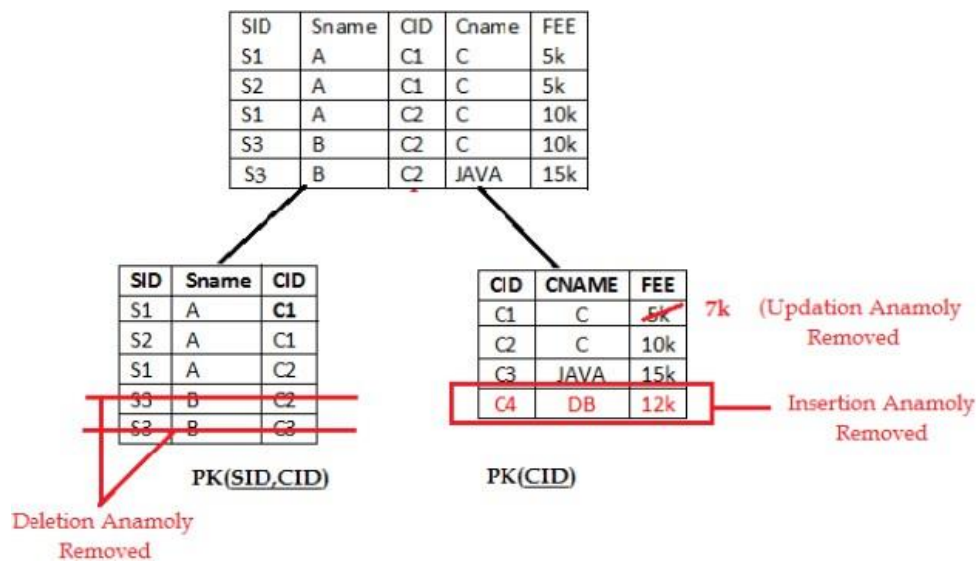
Because of insertion of some data, it is forced to insert some other dummy data.

**Deletion Anomaly:**

Deletion of S3 student causes the deletion of course. Because of deletion of some data forced to delete some other useful data.

| SID           | Sname        | CID           | Cname           | FEE            |
|---------------|--------------|---------------|-----------------|----------------|
| S1            | A            | C1            | C               | 5k             |
| S2            | A            | C1            | C               | 5k             |
| S1            | A            | C2            | C               | 10k            |
| <del>S3</del> | <del>B</del> | <del>C2</del> | <del>C</del>    | <del>10k</del> |
| <del>S3</del> | <del>B</del> | <del>C2</del> | <del>JAVA</del> | <del>15k</del> |

SolutionsToAnomalies:DecompositionofTables–SchemaRefinement,shownbelow.



### Purposeof Normalization:

- Minimize theredundancyin data.
- Removeinsert,update,anddeleteanomaliesduringthedatabaseactivities.
- Reducetheneedtoorganizethedatawhenitismodifiedorenhanced.
- Normalizationreducesacomplex user view to aset of small and subgroupsof fields or relations. This process helps to design a logical data model known as conceptual data model.

### AdvantagesofNormalization:

1. Greateroveralldatabaseorganizationwillbegained.
2. Theamountofunnecessaryredundantdatareduced.
3. Dataintegrityiseasilymaintainedwithinthedatabase.
4. Thedatabase&applicationdesignprocessesaremuchforflexible.
5. Securityiseasiertomaintainormanager.

### DisadvantagesofNormalization:

1. Thedisadvantageofnormalizationisthatitproducesalotoftableswitharelatively small number of columns. These columns then have to be joined using their primary/foreign key relationship.
2. Thishastwodisadvantages.

**Performance:** all the joins required to merge data slow processing & place additional stress on your hardware.

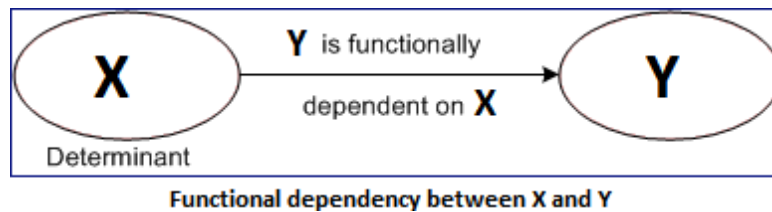
**Complex queries:** developers have to code complex queries in order to merge data from different tables.

## Concept of Functional Dependency:

**Functional Dependencies** are fundamental to the process of Normalization i.e., Functional Dependency plays key role in differentiating good database design from bad database designs. A functional dependency is a “**type of constraint that is a generalization of the notation of the key**”.

Functional Dependency describes the **relationship between attributes (columns) in a table**. Functional dependency is represented by an arrow sign ( $\rightarrow$ ).

**In other words, a dependency FD: “ $X \rightarrow Y$ ”** means that the values of Y are determined by the values of X. Two tuples sharing the same values of X will necessarily have the same values of Y. An attribute on left hand side is known as “Determinant”. Here X is a Determinant.



### Example [Identifying the FD's]

| A  | B  | C  | D  |
|----|----|----|----|
| A1 | B1 | C1 | D1 |
| A1 | B2 | C1 | D2 |
| A2 | B2 | C2 | D2 |
| A2 | B2 | C2 | D3 |
| A3 | B3 | C2 | D4 |

#### Case 1: $A \rightarrow B$

Here A1 belongs to B1 & B2. So A1 does not have unique value in B. So it is not in FD.

#### Case 1: $A \rightarrow C$

Here  $A1 \rightarrow C1$  and  $A2, A3 \rightarrow C2$ . So A has unique values in C. So it is in FD.

**Note:** try to find all the possibilities. i.e.,  $A \rightarrow D$ ,  $B \rightarrow C$ ,  $B \rightarrow D$ , and  $C \rightarrow D$

## Reasoning about functional dependencies:

**Armstrong Axioms (Inference Rules) :** The term Armstrong axioms refers to the sound and complete set of inference rules or axioms, introduced by William W. Armstrong, that is used to test logical implication of **functional dependencies**.

Armstrong axioms define the set of rules for reasoning about functional dependencies and also to infer all the functional dependencies on a relational database.

## Variousaxioms rulesor inferencerules:

Primaryaxioms:

|               |                                                                                                                                                                                                                                  |
|---------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Rule 1</b> | <b>Reflexivity</b><br>If A is a set of attributes and B is a subset of A, then A holds B. $\{ A \rightarrow B \}$                                                                                                                |
| <b>Rule 2</b> | <b>Augmentation</b><br>If A hold B and C is a set of attributes, then AC holds BC. $\{AC \rightarrow BC\}$<br>It means that attribute in dependencies does not change the basic dependencies.                                    |
| <b>Rule 3</b> | <b>Transitivity</b><br>If A holds B and B holds C, then A holds C.<br>If $\{A \rightarrow B\}$ and $\{B \rightarrow C\}$ , then $\{A \rightarrow C\}$<br>A holds B $\{A \rightarrow B\}$ means that A functionally determines B. |

Secondaryorderivedaxioms:

|               |                                                                                                                                                                  |
|---------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Rule 1</b> | <b>Union</b><br>If A holds B and A holds C, then A holds BC.<br>If $\{A \rightarrow B\}$ and $\{A \rightarrow C\}$ , then $\{A \rightarrow BC\}$                 |
| <b>Rule 2</b> | <b>Decomposition</b><br>If A holds BC and A holds B, then A holds C.<br>If $\{A \rightarrow BC\}$ and $\{A \rightarrow B\}$ , then $\{A \rightarrow C\}$         |
| <b>Rule 3</b> | <b>Pseudo Transitivity</b><br>If A holds B and BC holds D, then AC holds D.<br>If $\{A \rightarrow B\}$ and $\{BC \rightarrow D\}$ , then $\{AC \rightarrow D\}$ |

## ClosureofaSetofAttributes:

Attributeclosureofanattributesetcanbedefinedassetofattributeswhichcanbe functionally determined from it.

The set of FD's that is logically implied by F is called the closure of F and written as  $F^+$ . And it is defined as "If F is a set FD's on a relation R, the  $F^+$ , the closure of F by using the inferences axioms that are not contained in  $F^+$ .

**Example:** $R(A,B,C,D)$ andsetofFunctionalDependenciesare $A \rightarrow B, B \rightarrow D, C \rightarrow B$ thenwhat is the Closure of A, B, C, D?

**Solution:** $A^+$ is

$A^+ \rightarrow \{A, B, D\}$  i.e.,  $A \rightarrow B, B \rightarrow D$  is exists and C is not FD on A. So it is eliminated.

$B^+ \rightarrow \{B, D\}$  i.e.,  $B \rightarrow D$  is exists and A, C is not FD on A. So it is eliminated.

$C^+ \rightarrow \{C, B, D\}$  i.e.,  $C \rightarrow B, B \rightarrow D$  is exists and A is not FD on C. So it is eliminated.

Thealgorithmforcomputingtheattribute closure ofasetXofattributesisshownbelow

```
closure = X;
repeat until there is no change: {
 if there is an FD $U \rightarrow V$ in F such that $U \subseteq closure$,
 then set $closure = closure \cup V$
}
```

## Types of functional dependencies:

1. **Fully Functional Dependency:** A functional dependency is said to be full dependency “if and only if the determinant of the functional dependency is either candidate key or superkey, and the dependent can be either prime or non-prime attribute”.

(OR)

Let's take the functional dependency  $X \rightarrow Y$  (i.e., X determines Y). Here Y is said to be fully determinant, if it cannot determine any subset of X.

**Example:** Consider the following determinant  $ABC \rightarrow D$  i.e., ABC determines D but D is not determined by any subset of A/ BC/ C/ B/ AB i.e.,  $BC \rightarrow D$ ,  $C \rightarrow D$ ,  $A \rightarrow D$  Functional dependencies are not exists. So Dis Fully Functional Dependent.

2. **Partial Functional Dependency:** If a non-prime attribute of the relation is getting derived by only a part of the candidate key, then such dependency is known as Partial Dependency.

(OR)

In a relation having more than one key field, a subset of non key fields may depend on all key fields but another subset or a particular non-key field may depend on only one of the key fields. Such dependency is defined as **Partial Dependency**.

**Example:** Consider the following determinants  $AC \rightarrow P$ ,  $A \rightarrow D$ ,  $D \rightarrow P$ . From these determinants P is not fully FD on AC. Because, If we find A+ (means A's Closure)  $A \rightarrow D$ ,  $D \rightarrow P$  i.e.,  $A \rightarrow P$ . But we don't have any requirement of C. Attribute is removed completely. So P is Partially Dependent on AC.

**Under the following conditions a table cannot have partial F.D**

- (1) If primary key consists as single attribute
- (2) If table consists only two attributes
- (3) If all the attributes in the table are part of the primary key

3. **Transitive Functional Dependency:** If a non-prime attribute of a relation is getting derived by either another non-prime attribute or the combination of the part of the candidate key along with non-prime attribute, then such dependency is defined as **Transitive dependency**. i.e., in a relation, there may be dependency among non-key fields. Such dependency is called Transitive Functional Dependency.

**Example:**  $X \rightarrow Y$ , and  $Y \rightarrow Z$  then we can determine  $X \rightarrow Z$  holds.

**Under the following Circumstances, a table cannot have transitive F.D**

- (1) If table consists only two attributes
- (2) If all the attributes in the table are part of the primary key.

4. **Trivial Functional Dependency:** It is basically related to Reflexive rule. i.e., if X is a set of attributes, and Y is subset of X then  $X \rightarrow Y$  holds.

**Example:**  $ABC \rightarrow BC$  is a Trivial Dependency.

5. **Multi-ValuedDependency:** Consider 3 fields X,Y,and Z in a relation. If for each value of X, there is a well-defined set of values Y and Well-defined set of values of Z and set of values of Y is independent of the set values of Z. This dependency is Multi-valued Dependency. i.e.,  $X \twoheadrightarrow Y / Z$ .

### Prime and non-prime attributes

Attributes which are parts of any candidate key of relation are called as prime attribute, others are non-prime attributes.

### Candidate Key:

Candidate Key is minimal set of attributes of a relation which can be used to identify a tuple uniquely.

Consider student table: student(sno, sname, sphone, age)

we can take **sno** as candidate key. we can have more than 1 candidate key in a table. types of candidate keys:

1. simple (having only one attribute)
2. composite (having multiple attributes as candidate key)

### Super Key:

Super Key is set of attributes of a relation which can be used to identify a tuple uniquely.

- Adding zero or more attributes to candidate key generates super key.
- A candidate key is a super key but vice versa is not true.

Consider student table: student(sno, sname, sphone, age)

we can take **sno**, (**sno, sname**) as super key

### Operations performed on functional dependencies (applications of closure set of attributes):

- (1) To identify the additional F.D's.
- (2) To identify the keys.
- (3) To identify the equivalences of the F.D's
- (4) To identify irreducible set (minimal set) of F.D's or canonical forms of F.D's or standard form of F.D's.

#### (1) To identify the additional F.D's:

To check any F.D's like  $A \rightarrow B$  can be determined from F1 or not. Complete  $A^+$  from F1 is  $A^+$  includes B also then  $A \rightarrow B$  can be derived as a F.D in F1.

### Examples:

1. In a schema with attributes A,B,C,D and E the following set of attributes are given  $A \rightarrow B, A \rightarrow C, CD \rightarrow E, B \rightarrow D, E \rightarrow A$ . Find  $CD \rightarrow AC$  determines from the given FDs or not.

**Sol:** Given FD is  $CD \rightarrow AC$  find the closure set of CD.

$CD^+ = CDE$  ( $\because CD \rightarrow E$ )

$= CDEA$  ( $\because E \rightarrow A$ )

$= CDEAB$  ( $\because A \rightarrow B$ )

From the closure set the attributes AC are determined by CD so  $CD \rightarrow AC$ .

**2. Check  $D \rightarrow A$  can be derived from the following FDs or not  $AB \rightarrow C$ ,  $BC \rightarrow AD$ ,  $D \rightarrow E$ ,  $CF \rightarrow B$ .**

**(2) Identification of key by using closure set as attributes:**

**A key attribute:** An attribute that is capable of identifying all other attributes in a given table.

- (i) **Primary key:** It is a unique value attribute in a table to enforce entity integrity and to identify rows in the table uniquely.
- (ii) **Composite Primary Key:** Sometimes single attribute is not sufficient to identify uniquely the rows in the table so, we combine 2 or more attributes to identify the rows uniquely.
- (iii) **Candidate keys:** Sometimes 2 or more independent attribute or attributes can be used to identify the rows uniquely Eg : (veh no, veng no, purchase date) Either vehicle no or vehicle engine no can be used as a key attribute then they are called as candidate keys one of the candidate key can be elected as primary key.

**Example 1: Find candidate keys for the relation R(ABCD) having following FD's  $AB \rightarrow CD$ ,  $C \rightarrow A$ ,  $D \rightarrow A$ .**

**Sol:** From the given FD's, the attribute B is key attribute because it is not in RHS of functional dependency.

$B^+ = B$  (not a candidate key, find the combinations of B)  $AB^+ =$

$ABCD$  ( $\because AB \rightarrow CD$ )

$BC^+ = BCAD$  ( $\because C \rightarrow A$ ,  $AB \rightarrow CD$ )

$BD^+ = BDA$  ( $\because D \rightarrow A$ )

$CD^+ = CDA$  ( $\because D \rightarrow A$ )

$AC^+ = AC$

$AD^+ = AD$

From the above attributes AB and BC determines all attributes. AB, BC are candidate keys.

**Example2: Find candidate keys for the relation R(ABCDE) having following FD's  $A \rightarrow BC$ ,  $CD \rightarrow E$ ,  $B \rightarrow D$ ,  $E \rightarrow A$ .**

**Sol:** From the given FD's, no attribute is key attribute because all are in RHS of functional dependency. So check for all attributes of LHS.

$A^+ = ABC (\because A \rightarrow BC)$   
 $= ABCD (\because B \rightarrow D)$   
 $= ABCDE (\because CD \rightarrow E)$   
 $B^+ = BD (\because B \rightarrow D)$   
 $E^+ = EA (\because E \rightarrow A)$   
 $= EABC (\because A \rightarrow BC)$   
 $= EABCD (\because B \rightarrow D)$   
 $C^+ = C$   
 $D^+ = D$   
 $CD^+ = CDE (\because CD \rightarrow E)$   
 $= CDEA (\because E \rightarrow A)$   
 $= CDEAB (\because A \rightarrow BC)$   
 $BC^+ = BCD (\because B \rightarrow D)$   
 $= BCDE (\because CD \rightarrow E)$   
 $= BCDEA (\because E \rightarrow A)$

From the above attributes A, E, CD and BC determines all attributes. A, E, CD, BC are candidate keys.

**Question 2: Given a relation R(ABCDEF) having FDs  $\{AB \rightarrow C, C \rightarrow DE, E \rightarrow F, C \rightarrow B\}$  Identify the prime attributes and non prime attributes.**

**Solution :**

$(AB)^+ : \{A B C D E F\}$   
 $(A)^+ : \{A\}$   
 $(B)^+ : \{B\}$   
 $(AB) \Rightarrow (AC), (AC)^+ : \{ABCDEF\}$   
 $(C)^+ : \{DECBF\}$   
 $\Rightarrow$  Candidate Keys  $\{AB, AC\}$   
 $\Rightarrow$  Prime Attributes  $\{A, B, C\}$   
 $\Rightarrow$  Non Prime Attributes  $\{D, E, F\}$

**Question 1 :** Given a relation R(ABCDEF) having FDs {AB→C, C→D, D→E, F→B, E→F} Identify the prime attributes and non prime attributes .

**Solution :**

(AB)<sup>+</sup> : {ABCDEF} ⇒ Super Key

(A)<sup>+</sup> : {A} ⇒ Not Super Key

(B)<sup>+</sup> : {B} ⇒ Not Super Key

**Prime Attributes : {A,B}**

(AB) → Candidate Key

↓ (as F → B)

(AF)<sup>+</sup> : {AFBCDE}

(A)<sup>+</sup> : {A} ⇒ Not Super key

(F)<sup>+</sup> : {FB} ⇒ Not Super Key

(AF) → Candidate Key

↓

(AE)<sup>+</sup> : {AEFBCD}

(A)<sup>+</sup> : {A} ⇒ Not Super key

(E)<sup>+</sup> : {EFB} ⇒ Not Super key

(AE) → Candidate Key

↓

(AD)<sup>+</sup> : {ADEFB}

(A)<sup>+</sup> : {A} ⇒ Not Super key

(D)<sup>+</sup> : {DEFB} ⇒ Not Super key

(AD) → Candidate Key

↓

(AC)<sup>+</sup> : {ACDEFB}

(A)<sup>+</sup> : {A} ⇒ Not Super Key

(C)<sup>+</sup> : {DCEFB} ⇒ Not Super Key

⇒ Candidate Keys {AB, AF, AE, AD, AC}

⇒ Prime Attributes {A,B,C,D,E,F}

⇒ Non Prime Attributes {}

### (3) To identify equivalence of F.D's:

Different database designers may define different F.D's sets from the same requirements. To evaluate whether they are equivalent if we are able to derive all F.D's in G from F and vice-versa.

#### Q1 Consider the following two sets of FDs

F =  
 $A \rightarrow C$   
 $AC \rightarrow D$   
 $E \rightarrow AD$   
 $E \rightarrow H$

G =  
 $A \rightarrow CD$   
 $E \rightarrow AH$

Find the equivalence of two sets of FDs.

**Sol:**

**Step1:** TakesetFandencloseallFD'sinGthatcanbederivedfromF.  $A \rightarrow CD$

$A^+$ fromF

$=A$

$=AC(\because A \rightarrow C)$

$=ACD(\because AC \rightarrow D)$

$\therefore A \rightarrow CD$  canbederivedfromF

$E \rightarrow AH$

$E^+$ fromF

$=E$

$=EAD$

$=EADH$

$\therefore E \rightarrow AH$  canbederivedfromF

**Step2:** TakesetGandencloseallF.D'sinFthatcanbederived from

$G.A \rightarrow C$

$A^+$ fromG

$=A$

$=ACD$

$A \rightarrow C$  canbederivedfromG

$E \rightarrow AD$

$E^+$ fromG

$=E$

$=EAH$

$=EAHCD$

$E \rightarrow AH$  &  $E \rightarrow AD$  canbederivedfromG

$\therefore G$  and  $F$  areequivalent.

**Q2 Consider two sets of FDs on the attributes ABCDE**

$B \rightarrow CD$

F=  $AD \rightarrow E$

$B \rightarrow A$

$B \rightarrow CDE$

G=  $B \rightarrow ABC$

$AD \rightarrow E$

**Find whether they are equivalent or not**

**(4) To identify the irreducible form of FD's/canonical Form (minimal cover):**

We try to minimize the functional dependency. The minimized FD should be equivalent to original FD,

**Procedure to find minimal set:**

**Step 1:** Have single attributes on the RHS for every FD.

**Step 2:** Evaluate all F.D's in step 1 for their necessity. If they are not necessary, remove them from the list.

**Step 3:** Evaluate the necessity of the LHS attributes in FD's obtained from step 2. If they are not necessary, remove from FD.

**Step 4:** Apply the union rule for common to LHS attribute in the FD's obtained from step 3. Then we will get irreducible set.

**Q1 Find the irreducible set from the following FDs**

F=

$A \rightarrow B$

$C \rightarrow B$

$D \rightarrow ABC$

$AC \rightarrow D$

**Sol:**

**Step 1:**

(1)  $A \rightarrow B$

(2)  $C \rightarrow B$

(3)  $D \rightarrow A$

(4)  $D \rightarrow B$

(5)  $D \rightarrow C$

(6)  $AC \rightarrow D$

**Step 2:**

Remove 1 & compute  $A^+$  from 2, 3, 4, 5, 6

$A^+ = A$

∴ We need 1

Remove 2 and compute 1, 3, 4, 5 & 6

$C^+ = C$

∴ We need 2.

Remove 3 and compute  $D^+$  from 1, 2, 4, 5 & 6

$D^+ = DBC$

∴ We need 3.

Remove 3 and compute  $D^+$  from 1, 2, 4, 5 & 6

Remove 4 and compute  $D^+$  from 1, 2, 4, 5 & 6

$D^+ = ADCB$

$\therefore D \rightarrow B$  can be removed.

Remove 5 and compute  $D^+$  from 1, 2, 3, 4 & 6

$D^+ = ABD$

$\therefore$  We need 5.

Remove 6 and compute  $D^+$  from 1, 2, 3, 4, 5

$AC^+ = ACB$

$\therefore$  We need 6.

**Step 3:**

$A \rightarrow B$

$C \rightarrow B$

$D \rightarrow A$

$D \rightarrow C$

$AC \rightarrow D$

Remove  $A_j$

|                   |                    |
|-------------------|--------------------|
| $A \rightarrow B$ | $A \rightarrow B$  |
| $C \rightarrow B$ | $C \rightarrow B$  |
| $D \rightarrow A$ | $D \rightarrow A$  |
| $D \rightarrow C$ | $D \rightarrow C$  |
| $C \rightarrow D$ | $AC \rightarrow D$ |
| $C^+ = CDAB$      | $C^+ = CB$         |

$\therefore C^+ \neq C^+$

Remove  $C_j$

|                   |                    |
|-------------------|--------------------|
| $A \rightarrow B$ | $A \rightarrow B$  |
| $C \rightarrow B$ | $C \rightarrow B$  |
| $D \rightarrow A$ | $D \rightarrow A$  |
| $D \rightarrow C$ | $D \rightarrow C$  |
| $A \rightarrow D$ | $AC \rightarrow D$ |
| $A^+ = ADCB$      | $A^+ = AB$         |

$A^+ \neq A^+$

**Step4:** $A \rightarrow B$  $C \rightarrow B$  $D \rightarrow A$  $D \rightarrow C$  $AC \rightarrow D$  $A \rightarrow B$  $C \rightarrow B$  $D \rightarrow AC$  $AC \rightarrow D$ 

Therefore, it is an irreducible F.D.

**Q2 Consider Universal relation with attributes ABC and FDs** $AB \rightarrow C$  $C \rightarrow B$  $A \rightarrow B$ 

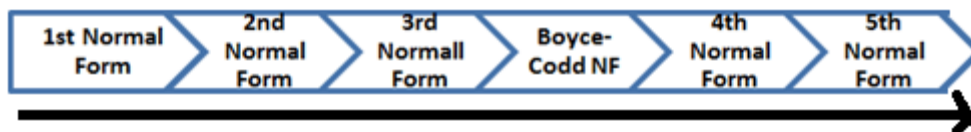
Find the Irreducible set

## Normal forms based on functional dependency (1NF, 2NF and 3 NF, Boyce-Codd normal form (BCNF), 4NF)

*Normalization* means “split the tables into small tables which will contain less number of attributes in such a way that table design must not contain any problem of inserting, deleting, updating anomalies and guarantees no redundancy”.

The evolution of Normalization theories / Steps of Normalization / Different Normal Forms is illustrated below-

1. First Normal Form (1NF)
2. Second Normal Form (2NF)
3. Third Normal Form (3NF)
4. Boyce-Codd Normal Form (BCNF)
5. Fourth Normal Form (4NF)
6. Fifth Normal Form (5NF).



### Points to Remember

- 1NF is a mandatory NF and remaining are the optional
- If you construct E-R diagrams into the tables, then 4NF and 5NF need not be applied on the table.
- Practically applied normalization is up to 3NF and very rarely we will go beyond that.
- 2NF dealing with the partial dependencies and 3NF is dealing with transitive dependencies.

**First Normal Form (1NF):** A relation is said to be in 1NF if it is already in un-normalized form and it satisfies the following conditions or rules or qualifications are:

1. Each attribute name must be unique.
2. Each attribute value must be single or atomic i.e., Single Valued Attributes.
3. Each row/record must be unique.
4. There is no repeating group's.

**Example:** How do we bring an un-normalized table into first normal form? Consider the following relation:

**TABLE\_PRODUCT**

| Product ID | Color        | Price |
|------------|--------------|-------|
| 1          | red, green   | 15.99 |
| 2          | yellow       | 23.99 |
| 3          | green        | 17.50 |
| 4          | yellow, blue | 9.99  |
| 5          | red          | 29.99 |

**Solution:** This table is not in first normal form because the [Color] column can contain multiple values. For example, the first row includes values "red" and "green." To bring this table to first normal form, we split the table into two tables and now we have the resulting tables:

**TABLE\_PRODUCT\_PRICE**

| Product ID | Price |
|------------|-------|
| 1          | 15.99 |
| 2          | 23.99 |
| 3          | 17.50 |
| 4          | 9.99  |
| 5          | 29.99 |

**TABLE\_PRODUCT\_COLOR**

| Product ID | Color  |
|------------|--------|
| 1          | red    |
| 1          | green  |
| 2          | yellow |
| 3          | green  |
| 4          | yellow |
| 4          | blue   |
| 5          | red    |

**Second Normal Form (2NF):** A relation is said to be in 2NF, if it is already in 1st NF and it has no Partial Dependency i.e., no non-prime attribute is dependent on the only a part of the candidate key.

(OR)

A relation is in second normal form if it satisfies the following conditions:

- It is in first normal form
- All non-key attributes are fully functional dependent on the primary key.

**Note: Partial Functional Dependency:** If a non-prime attribute of the relation is getting derived by only a part of the candidate key, then such dependency is known as Partial Dependency

**Example:** Consider the following relation

**TABLE\_PURCHASE\_DETAIL**

| Customer ID | Store ID | Purchase Location |
|-------------|----------|-------------------|
| 1           | 1        | Los Angeles       |
| 1           | 3        | San Francisco     |
| 2           | 1        | Los Angeles       |
| 3           | 2        | New York          |
| 4           | 3        | San Francisco     |

→ This table has a composite primary key [Customer ID, Store ID]. The non-key attribute is [Purchase Location]. In this case, [Purchase Location] only depends on [Store ID], which is only part of the primary key. Therefore, this table does not satisfy second normal form.

→ To bring this table to second normal form, we break the table into two tables, and now we have the following:

**TABLE\_PURCHASE**

| Customer ID | Store ID |
|-------------|----------|
| 1           | 1        |
| 1           | 3        |
| 2           | 1        |
| 3           | 2        |
| 4           | 3        |

**TABLE\_STORE**

| Store ID | Purchase Location |
|----------|-------------------|
| 1        | Los Angeles       |
| 2        | New York          |
| 3        | San Francisco     |

**Q1 Given relation R(ABCD) and F: {AB → C, B → D} Decompose into 2NF.**

from the given FDs determine primary key. Necessary attributes to include in the key are A, B (because these attributes are not in RHS of FD).

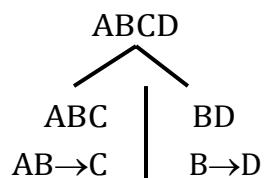
Find the closure set of AB

$$AB^+ = ABC$$

$$= ABCD (\because B \rightarrow D)$$

∴ AB is a primary key.

From the FDs B → D is partially dependent on AB. So decompose the table. (D is a non-prime attribute derived by a part of the key)



**Q2** Consider the relation  $R=ABCDEF$  and set of FDs are  $A \rightarrow FC, C \rightarrow D, B \rightarrow E$ . Find the key and normalize into 2NF.

**Third Normal Form (3NF):** A database is in third normal form if it satisfies the following conditions:

- It is in 2NF.
- There is no transitive functional dependency.
  - By **transitive functional dependency**, we mean we have the following relationships in the table: A is functionally dependent on B, and B is functionally dependent on C. In this case, C is transitively dependent on A via B. **and a non-key attribute is depending on a non-key attribute.**

**Example:** Consider the following relation.

**TABLE\_BOOK\_DETAIL**

| Book ID | Genre ID | Genre Type | Price |
|---------|----------|------------|-------|
| 1       | 1        | Gardening  | 25.99 |
| 2       | 2        | Sports     | 14.99 |
| 3       | 1        | Gardening  | 10.00 |
| 4       | 3        | Travel     | 12.99 |
| 5       | 2        | Sports     | 17.99 |

- In the table above, [Book ID] determines [Genre ID], and [Genre ID] determines [Genre Type]. Therefore, [Book ID] determines [Genre Type] via [Genre ID] and we have transitive functional dependency, and this structure does not satisfy third normal form.
- To bring this table to third normal form, we split the table into two as follows:

TABLE\_BOOK

| Book ID | Genre ID | Price |
|---------|----------|-------|
| 1       | 1        | 25.99 |
| 2       | 2        | 14.99 |
| 3       | 1        | 10.00 |
| 4       | 3        | 12.99 |
| 5       | 2        | 17.99 |

TABLE\_GENRE

| Genre ID | Genre Type |
|----------|------------|
| 1        | Gardening  |
| 2        | Sports     |
| 3        | Travel     |

**Q1 Given relation R(ABCDE) and F: {AB → C, B → D, D → E} Decompose into 3NF.**

from the given FDs determine primary key. Necessary attributes to include in the key are A, B (because these attributes are not in RHS of FD).

Find the closure set of AB

$$AB^+ = ABC$$

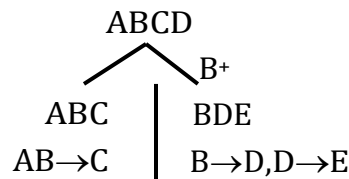
$$= ABCD \quad (\because B \rightarrow D)$$

$$= ABCDE \quad (\because D \rightarrow E)$$

$\therefore$  AB is a primary key.

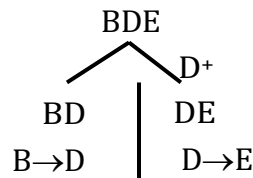
From the FDs  $B \rightarrow D$  is partially depending on AB. So decompose the table. (D is a non-prime attribute derived by a part of the key)

$$B^+ = BDE$$



$\therefore$  table is in 2NF but not in 3NF. Because  $D \rightarrow E$  is a transitive dependency. (No non-key attribute should determine a non-key attribute)

$$D^+ = DE$$



$\therefore$  Table is 3NF.

The relations after decomposing into 3NF.

R1: ABC

R2: BD

R3: DE

**Q2** Given relation  $R=ABCDEFGHIJ$  and the set of FDs are  $AB \rightarrow C$ ,  $A \rightarrow DE$ ,  $B \rightarrow F$ ,  $F \rightarrow GH$ ,  $D \rightarrow IJ$  Decompose  $R$  into 3NF.

**Q3(a)** Given a set of FDs for the relation schema  $R(ABCD)$  with primary key  $AB$  under which  $R$  is 1NF but not in 2NF

**(b)** Find FDs such that  $R$  is in 2NF but not in 3NF

**Sol:**  $R=ABCD$   
 $Key=AB$

(a) Atomic values are allowed in 1NF and partial dependency is not allowed in 2NF.

The following FDs are allowed.

$B \rightarrow C, A \rightarrow C, B \rightarrow D, A \rightarrow D$

(show the FDs which is having partial dependency)

(b) According to question partial dependencies are not allowed and transitivity dependency is allowed. The following FDs are allowed.

$C \rightarrow D, D \rightarrow C$

**Boyce-Coddnormalform(BCNF):**ArelationissaidtobeinBCNF,ifandonlyifevery determinant should be a candidate key.

- ✓ BCNFistheadvanceversionof3NF.Itisstricterthan3NF.
- ✓ Atableisin3NFifforeveryfunctionaldependency $X \rightarrow Y$ ,Xisthesuperkeyofthetable.
- ✓ ForBCNF,thetable shouldbein3NFandforeveryFD,LHSissuperkey.

**Example:**Let'sassumethereisacompanywhereemployeesworkinmorethanone department. EMPLOYEE table:

| emp_id | emp_nationality | emp_dept                     | dept_type | dept_no_of_emp |
|--------|-----------------|------------------------------|-----------|----------------|
| 1001   | Austrian        | Productionandplanning        | D001      | 200            |
| 1001   | Austrian        | stores                       | D001      | 250            |
| 1002   | American        | design and technical support | D134      | 100            |
| 1002   | American        | Purchasingdepartment         | D134      | 600            |

→ IntheabovetableFunctionaldependenciesareasfollows:EMP\_ID→EMP\_COUNTRY and EMP\_DEPT → {DEPT\_TYPE, EMP\_DEPT\_NO} Candidate key: {EMP-ID, EMP-DEPT}

→ ThetableisnotinBCNFbecauseneitherEMP\_DEPTnorEMP\_IDalonearekeys.To convertthegiventableintoBCNF,wedecomposeitintothreetables:

EMP\_COUNTRY table:

| EMP_ID | EMP_COUNTRY |
|--------|-------------|
| 264    | India       |
| 264    | India       |

EMP\_DEPT table:

| EMP_DEPT  | DEPT_TYPE | EMP_DEPT_NO |
|-----------|-----------|-------------|
| Designing | D394      | 283         |
| 283       | D394      | 300         |
| Stores    | D283      | 232         |
| 549       | D283      | 549         |

EMP\_DEPT\_MAPPING table:

| EMP_ID | EMP_DEPT |
|--------|----------|
| D394   | 283      |
| D394   | 300      |
| D283   | 232      |
| D283   | 549      |

### FORBCNFproblemsreferyournotebook.

**Q1** Considertherelationschema $R(A,B,C)$ ,whichhastheFDB $\rightarrow C$ .IfAisacandidatekeyforR, is itpossible for R to be in BCNF? Ifso, under what conditions? If not, explain why not.

**Sol:** TheonlywayRcouldbeinBCNFisifBincludesakey,i.e.BisakeyforR

**Fourth Normal Form (4NF):**Arelationsaidtobein4NFifitisinBoyceCoddnormal form and should have no multi-valued dependency.

✓ Foradependency $A \rightarrow B$ ,ifforasinglevalueofA,multiplevalueofBexiststhen the relation will be multi-valued dependency.

✓ **Note: Multi Valued Dependency:**A table is said to have multi-valued dependency, if the following conditions are true,

1. For a dependency  $A \rightarrow B$ , if for a single value of A, multiple value of B exists, then the table may have multi-valued dependency.
2. Also,atableshouldhaveat-least3columnsforittohaveamulti-valueddependency.
3. And, for a relation R (A, B, C), if there is a multi-valued dependency between, A andB, then B and C should be independent of each other.

■ Ifalltheseconditionsaretrueforanyrelation(table),itissaidtohavemulti-valued dependency.

### Example

| 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, student with STU\_ID, 21 contains two courses, Computer and Math and two hobbies, Dancing andSinging. So there is aMulti-valued dependency on STU\_ID,which leads to un-necessary repetition of data.
- Sotomaketheabovetableinto4NF,wecandecomposeitintotwotables:  
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  |

**Concept of Surrogate Key:**

- ✓ Alternate of Primary Key that allows duplication of data's/records.
- ✓ Surrogate key is a unique identification key, it is like an artificial key to production key, because the production key may be alphanumeric or composite key but the surrogate key is always single numeric key.
- ✓ A surrogate key has the following characteristics:
  - i. The value is never reused and is unique within the whole system.
  - ii. It is system generated and an integer.
  - iii. The value cannot be manipulated by the user or application.
  - iv. The value is not an amalgam of different values from multiple domains.
- ✓ A Surrogate Key can be generated in a variety of ways, and most databases offer ways to generate surrogate keys.

Example: Oracle uses **SEQUENCE**,  
 MYSQL uses **Auto\_Increment**,  
 and SQL Server uses **IDENTITY**.

**Lossless join and Dependency preserving decomposition:**

**Decomposition:** Decomposition of a relation is done when a relation in relational model is not in appropriate normal form. Relation R is decomposed into two or more relations if decomposition is lossless join as well as dependency preserving.

→ If we decompose a relation R into relations R1 and R2, There are 2 types of decomposition:

1. **Decomposition is lossy**, if  $R1 \bowtie R2 \supset R$ .

- ✓ The decomposition of relation R into R1 and R2 is lossy when the join of R1 and R2 does not yield the same relation as in R." One of the disadvantages of decomposition into two or more relational schemes (or tables) is that some information is lost during retrieval of original relation or table.
- ✓ Consider that we have table STUDENT with three attribute roll\_no, sname and department.

| Roll_no | Sname   | Dept       |
|---------|---------|------------|
| 111     | parimal | COMPUTER   |
| 222     | parimal | ELECTRICAL |

- ✓ This relation is decomposed into two relation no\_name and name\_dept:

| Roll_no | Sname   |
|---------|---------|
| 111     | parimal |
| 222     | parimal |

| Sname   | Dept       |
|---------|------------|
| parimal | COMPUTER   |
| parimal | ELECTRICAL |

- ✓ In lossy decomposition, spurious tuples are generated when a natural join is applied to the relations in the decomposition.

stu\_joined :

| Roll_no | Sname   | Dept       |
|---------|---------|------------|
| 111     | parimal | COMPUTER   |
| 111     | parimal | ELECTRICAL |
| 222     | parimal | COMPUTER   |
| 222     | parimal | ELECTRICAL |

- ✓ The above decomposition is a bad decomposition or Lossy decomposition.

2. **Lossless Join Decomposition:** Decomposition is lossless if  $R1 \bowtie R2 = R$

- ✓ This is also referred as **non-additive decomposition**.
- ✓ The lossless-join decomposition is always defined with respect to a specific set F of dependencies
- ✓ **To check for lossless join decomposition using FD set, following conditions must hold:**
  1. Union of Attributes of R1 and R2 must be equal to attribute of R. Each attribute of R must be either in R1 or in R2.

$$\text{Att}(R1) \cup \text{Att}(R2) = \text{Att}(R).$$

2. Intersection of Attributes of R1 and R2 must not be NULL.

$$\text{Att}(R1) \cap \text{Att}(R2) \neq \Phi.$$

3. Common attribute must be a key for at least one relation (R1 or R2)

$$\text{Att}(R1) \cap \text{Att}(R2) \rightarrow \text{Att}(R1) \text{ or } \text{Att}(R1) \cap \text{Att}(R2) \rightarrow \text{Att}(R2).$$

- ✓ **Example 1.:** A relation R (A, B, C, D) with FD set{A->BC} is decomposed into R1(ABC) and R2(AD) which is a lossless join decomposition as:

1. First condition holds true as  $\text{Att}(R1) \cup \text{Att}(R2) = (ABC) \cup (AD) = (ABCD) = \text{Att}(R)$ .
2. Second condition holds true as  $\text{Att}(R1) \cap \text{Att}(R2) = (ABC) \cap (AD) \neq \Phi$
3. Third condition holds true as  $\text{Att}(R1) \cap \text{Att}(R2) = A$  is a key of R1(ABC) because A->BC is given.

- ✓ **Example 2:** Consider that we have table STUDENT with three attribute roll\_no, sname and department.  
STUDENT:

| Roll_no | Sname   | Dept       |
|---------|---------|------------|
| 111     | parimal | COMPUTER   |
| 222     | parimal | ELECTRICAL |

This relation is decomposed into two relation Stu\_name and Stu\_dept:

| Roll_no | Sname   |
|---------|---------|
| 111     | parimal |
| 222     | parimal |

| Roll_no | Dept       |
|---------|------------|
| 111     | COMPUTER   |
| 222     | ELECTRICAL |

Now, when these two relations are joined on the common column 'roll\_no', the resultant relation will look like stu\_joined.

| Roll_no | Sname   | Dept       |
|---------|---------|------------|
| 111     | parimal | COMPUTER   |
| 222     | parimal | ELECTRICAL |

In lossless decomposition, no any spurious tuples are generated when a natural joined is applied to the relations in the decomposition.

➤ **Dependency Preserving Decomposition:**

- ✓ If we decompose a relation R into relations R1 and R2, All dependencies of R either must be a part of R1 or R2 or must be derivable from combination of FD's of R1 and R2.
- ✓ The dependency preservation decomposition is another property of decomposed relational database schema D in which each functional dependency  $X \rightarrow Y$  specified in F either appeared directly in one of the relation schemas Ri in the decomposed D or could be inferred from the dependencies that appear in some Ri.
- ✓ Decomposition  $D = \{R_1, R_2, R_3, \dots, R_n\}$  of R is said to be dependency-preserving with respect to F if the union of the projections of F on each Ri, in D is equivalent to F. In other words,  $R \subseteq \text{join of } R_i, R_i \text{ over } X$ . The dependencies are preserved because each dependency in F represents a constraint on the database. If decomposition is not dependency-preserving, some dependency is lost in the decomposition.
- ✓ **Example 1:** A relation R (A, B, C, D) with FD set  $\{A \rightarrow BC\}$  is decomposed into R1(ABC) and R2(AD) which is dependency preserving because FD  $A \rightarrow BC$  is a part of R1(ABC).
- ✓ **Example 2:** Let a relation R(A,B,C,D) and set a FDs  $F = \{A \rightarrow B, A \rightarrow C, C \rightarrow D\}$  are given. A relation R is decomposed into –

$R_1 = (A, B, C)$  with FDs  $F_1 = \{A \rightarrow B, A \rightarrow C\}$ , and  
 $R_2 = (C, D)$  with FDs  $F_2 = \{C \rightarrow D\}$ .  
 $F' = F_1 \cup F_2 = \{A \rightarrow B, A \rightarrow C, C \rightarrow D\}$   
 So,  $F' = F$ .  
 And so,  $F'^+ = F^+$ .

Thus, the decomposition is dependency preserving decomposition.

**Review Questions**

1. What is Functional Dependency? Explain types and properties of FD's.
2. What is a normal form? Explain about various normal forms with examples.
3. What is normalization? Differentiate between second normal form and third normal form.
4. Explain briefly about 3NF, 4NF with suitable examples?
5. Explain about Boyce Codd normal form with an example.
6. Why is normalization needed? Explain the process of normalization.
7. How to compute closure of set of functional dependencies? Explain with a suitable example schema.
8. What is multivalued dependency? State and explain fourth normal form based on this concept.
9. List and explain the inference rules of functional dependencies.
10. Explain insertion, deletion, and modification anomalies.
11. What is the importance of dependency preservation during decomposition? How to achieve it?
12. Consider the relation R on attributes (ABCDE) with functional dependencies:  
 $AB \rightarrow CDE, AC \rightarrow BDF, B \rightarrow C, C \rightarrow B, C \rightarrow D, B \rightarrow E$ 
  - i) Determine a Key for relation R
  - ii) Find 3NF decomposition for R using normalization process
13. Give a set of FDs for the relation schema R(A,B,C,D) with primary key AB under which R is in 1NF but not in 2NF.
14. Why is a relation that is in 3NF generally considered good? Explain.
15. Discuss about 4NF with suitable example.
16. What are the problems caused by redundantly storing information? Explain.
17. Given Relation,  $R=(A,B,C,D,E,F,G)$  and Functional Dependencies  
 $F=\{ \{A,B\} \rightarrow \{C\}, \{A,C\} \rightarrow \{B\}, \{A,D\} \rightarrow \{E\}, \{B\} \rightarrow \{D\}, \{B,C\} \rightarrow \{A\}, \{E\} \rightarrow \{F\} \}$ .  
 Check whether the following decomposition of R into  $R_1=(A,B,C), R_2=(A,C,D,E)$  and  $R_3=(A,D,F)$  is satisfying the lossless Decomposition property.
18. What is dependency preservation property for decomposition? Explain why it is important.
19. Given a Relation  $R=(X,Y,Z)$  and Functional Dependencies are  $F=\{ \{X,Y\} \rightarrow \{Z\}, \{Z\} \rightarrow \{X\} \}$   
 Determine all Candidate keys of R and the normal form of R with proper explanation.
20. Define functional dependency? How can you compute the minimal cover for a set of functional dependencies? Explain it with an example.

21. Consider schema  $R = (A, B, C, G, H, I)$  and the set of functional dependencies  $\{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$ . Compute the candidate keys of the schema. Compute the closure of the same.
22. Explain 3NF & BCNF. What is the difference between them?
23. What is functional dependency? Explain its usage in database design.
24. What is a surrogate key? How can it be used for schema refinement?
25. How to compute closure of set of functional dependencies? Explain with a suitable example schema.
26. What is multivalued dependency? State and explain fourth normal form based on this concept.
27. Given a set of FDs for the relation schema  $R(A, B, C, D)$  with Primary key AB, and  $D \rightarrow C$  or  $C \rightarrow D$  or  $AC \rightarrow D$  or  $AD \rightarrow C$  or  $BC \rightarrow D$  or  $BD \rightarrow C$ . In which normal form is R?
28. Discuss the problems caused by redundancy and the purpose of normalization.
29. Give relation schemas for the following normal forms
  - i) 2NF but not in 3NF
  - ii) 3NF but not in BCNF

### **References:**

- Raghurama Krishnan, Johannes Gehrke, *Database Management Systems*, 3rd Edition, Tata McGraw Hill.
- C.J.Date, *Introduction to Database Systems*, Pearson Education.
- Elmasri Navrate, *Fundamentals of Database Systems*, Pearson Education.

# DATABASE MANAGEMENT SYSTEMS

## UNIT V: Transaction Concept & Indexing Techniques

### Syllabus:

**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.

### Objectives:

After studying this unit, you will be able to:

- Discuss the different types of ACID properties and its implementation.
- Describe concurrent Execution, Serializability and Recoverability.
- Understand physical design of a database system, by discussing Database indexing techniques and storage techniques
- Examine issues in data storage and query processing and can formulate appropriate solutions

### Introduction

- A **transaction** is a unit of program execution that accesses and possibly updates various data items.
- The transaction consists of all operations executed between the statements **begin** and **end** of the transaction
- **Transaction operations:** Access to the database is accomplished in a transaction by the following two operations:
  - *read (X): Performs the reading operation of data item X from the database*
  - *write (X): Performs the writing operation of data item X to the database*
- A transaction must see a consistent database
- During transaction execution the database may be inconsistent
- When the transaction is *committed*, the database must be consistent
- Two main issues to deal with:
  - *Failures, e.g. hardware failures and system crashes*
  - *Concurrency, for simultaneous execution of multiple transactions*

### ACID Properties

- To preserve integrity of data, the database system must ensure:
- **Atomicity:** Either all operations of the transaction are properly reflected in the database or none are
- **Consistency:** Execution of a transaction in isolation preserves the consistency of the database
- **Isolation:** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions; intermediate transaction results must be hidden from other concurrently executed transactions
- **Durability:** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures

**Example of Fund Transfer:** Let  $T_i$  be a transaction that transfers 50 from account A to B. This transaction can be illustrated as follows

**Transfer \$50 from account A to B:**

$T_i:$     **read(A)**

**$A := A - 50$**

**write(A)**

**read(B)**

**$B := B + 50$**

**write(B)**

- **Consistency:** the sum of A and B is unchanged by the execution of the transaction.
- **Atomicity:** if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will

result.

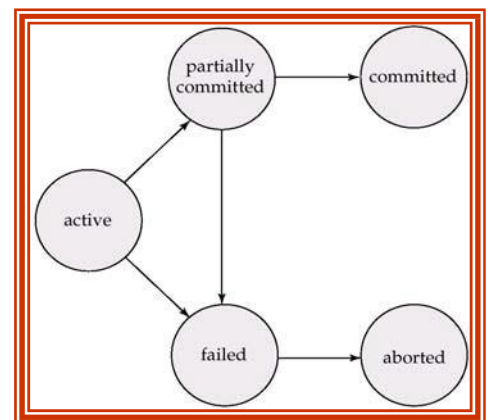
- **Durability:** once the user has been notified that the transaction has completed, the updates to the database by the transaction must persist despite failures.
- **Isolation:** between steps 3 and 6, no other transaction should access the partially updated database, or else it will see an inconsistent state (the sum  $A + B$  will be less than it should be).

### Transaction and Schedules

- A transaction is seen by the DBMS as a series, or list of actions. We therefore establish a simple transaction model named as *transaction states*.

**Transaction State:** A transaction must be one of the following states:

- **Active**, the initial state; the transaction stays in this state while it is executing
- **Partially committed**, after the final statement has been executed.
- **Committed**, after *successful completion*.
- **Failed**: after the discovery that *normal execution* can no longer proceed.
- **Aborted**: after the transaction has been rolled back and the database restored to its state prior to the start of the transaction.



### Concurrent Execution and Schedules

**Concurrent execution:** executing transactions simultaneously has the following advantages:

- *increased processor and disk utilization, leading to better throughput*
- *one transaction can be using the CPU while another is reading from or writing to the disk*
- *reduced average response time for transactions: short transactions need not wait behind long ones*

**Concurrency control schemes:** these are mechanisms to achieve isolation

- *to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database*

**Schedules:** sequences that indicate the chronological order in which instructions of concurrent transactions are executed

- *a schedule for a set of transactions must consist of all instructions of those transactions*
- *must preserve the order in which the instructions appear in each individual transaction*

### Example Schedules

- Let  $T_1$  transfer \$50 from  $A$  to  $B$ , and  $T_2$  transfer 10% of the balance from  $A$  to  $B$ . The following is a serial schedule (Schedule 1 in the text), in which  $T_1$  is followed by  $T_2$ .

| $T_1$                                                                                                            | $T_2$                                                                                                                                                          |
|------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\text{read}(A)$<br>$A := A - 50$<br>$\text{write}(A)$<br>$\text{read}(B)$<br>$B := B + 50$<br>$\text{write}(B)$ | $\text{read}(A)$<br>$\text{temp} := A * 0.1$<br>$A := A - \text{temp}$<br>$\text{write}(A)$<br>$\text{read}(B)$<br>$B := B + \text{temp}$<br>$\text{write}(B)$ |

**Schedule 1**

- Let  $T_1$  and  $T_2$  be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to above Schedule.

| $T_1$                                                  | $T_2$                                                                                       |
|--------------------------------------------------------|---------------------------------------------------------------------------------------------|
| $\text{read}(A)$<br>$A := A - 50$<br>$\text{write}(A)$ | $\text{read}(A)$<br>$\text{temp} := A * 0.1$<br>$A := A - \text{temp}$<br>$\text{write}(A)$ |
| $\text{read}(B)$<br>$B := B + 50$<br>$\text{write}(B)$ | $\text{read}(B)$<br>$B := B + \text{temp}$<br>$\text{write}(B)$                             |

**Schedule 2**

- The following concurrent schedule does not preserve the value of the sum  $A + B$

| $T_1$                                                                       | $T_2$                                                                                                           |
|-----------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------|
| $\text{read}(A)$<br>$A := A - 50$                                           | $\text{read}(A)$<br>$\text{temp} := A * 0.1$<br>$A := A - \text{temp}$<br>$\text{write}(A)$<br>$\text{read}(B)$ |
| $\text{write}(A)$<br>$\text{read}(B)$<br>$B := B + 50$<br>$\text{write}(B)$ | $B := B + \text{temp}$<br>$\text{write}(B)$                                                                     |

**Schedule 3**

### Serializable Schedule

- A *serializable schedule* over a set  $S$  of committed transactions is a schedule whose effect on any consistent database is guaranteed to be identical to that of some complete serial schedule over  $S$ . i.e., even though the actions of transactions are interleaved, the result of executing transactions serially in different order may produce different results.
- **Example:** The schedule shown in the following figure is serializable.

| $T_1$   | $T_2$   |
|---------|---------|
| R(A)    |         |
| W(A)    |         |
|         | R(A)    |
|         | W(A)    |
| R(B)    |         |
| W(B)    |         |
|         | R(B)    |
|         | W(A)    |
|         | Committ |
| Committ |         |

Even though the actions of  $T_1$  and  $T_2$  are interleaved, the result of this schedule is equivalent to first running  $T_1$  entirely and then running  $T_2$  entirely. Actually  $T_1$ 's read and write of B is not influenced by  $T_2$ 's actions on B, and the net effect is the same if these actions are the serial schedule First  $T_1$ , then  $T_2$ . This schedule is also serializable if first  $T_2$ , then  $T_1$ . Therefore if  $T_1$  and  $T_2$  are submitted concurrently to a DBMS, either of these two schedules could be chosen as first

- A DBMS might sometimes execute transactions which is not a serial execution i.e., not serializable.
- This can be happen for two reasons:
  - First the DBMS might use a concurrency control method that ensures the executed schedule itself.
  - Second, SQL gives programmers the authority to instruct the DBMS to choose non-serializable schedule.

### Anomalies due to Interleaved execution

- There are three main situations when the actions of two transactions  $T_1$  and  $T_2$  conflict with each other in the interleaved execution on the same data object.
  - **Write-Read (WR) Conflict:** Reading Uncommitted data.
  - **Read-Write (RW) Conflict:** Unrepeatable Reads
  - **Write-Write (WW) Conflict:** Overwriting Uncommitted Data.

### □ Reading Uncommitted Data (WR Conflicts)

- **Dirty Read:** The first source of anomalies is that a transaction  $T_2$  could read a database object A that has been just modified by another transaction  $T_1$ , which has not yet committed, such a read is called a *dirty read*.
- **Example:** Consider two transactions  $T_1$  and  $T_2$ , where  $T_1$  stands for transferring \$100 from A to B and  $T_2$  stands for incrementing both A and B by 6% of their accounts. Suppose that their actions are interleaved as follows:
  - (i)  $T_1$  deducts \$100 from account A, then immediately
  - (ii)  $T_2$  reads accounts of A and B adds 6% interest to each, and then,
  - (iii)  $T_1$  adds \$100 to account B.

*This corresponding schedule is illustrated as follows:*

| $T_1$                                  | $T_2$                                                                  |
|----------------------------------------|------------------------------------------------------------------------|
| R(A)<br>A: = A - 100<br>W(A)           | R(A)<br>A: = A + 0.06<br>AW(A)<br>R(B)<br>B:= B+.06<br>BW(B)<br>Commit |
| R(B)<br>B: = B + 100<br>W(B)<br>Commit |                                                                        |

The problem here is  $T_2$  has added incorrect 6% interest to each A and B. Because before commitment that \$100 is deducted from A, it has added 6% to account A before commitment that \$100 is credited to B, it has added 6% to account B. thus, the result of this schedule is different from the result of the other schedule which is serializable: first  $T_1$  then  $T_2$ .

### □ Unrepeatable Reads (RW Conflicts)

- The second source of anomalies is that a transaction  $T_2$  could change the value of an object A that has been read by a transaction  $T_1$  and  $T_1$  is still in progress. This situation causes a problem that, if  $T_1$  tries to read the value of A again, it will get a different result, even though it has not modified A in the meantime. But, this situation could not arise in a serial execute of two transactions: this, it is called as *unrepeatable read*.
- **Example:** Suppose that both  $T_1$  and  $T_2$  reads the same value of A, say 5. Then  $T_1$  has incremented A value to 6 but before commitment as A value 6,  $T_2$  has decremented A value from 5 to 4. Thus, instead of answer of A value as 5, i.e., from 5 we got an answer 4 which is incorrect.

### □ Overwriting Uncommitted Data (WW Conflicts)

- The third source of anomalies is that a transaction  $T_2$  could overwrite the value of an object A, which has already been modified by a transaction  $T_1$ , while  $T_1$  is still in progress.
- **Example:** Suppose that A and B are two employees, and their salaries must be kept equal. Transaction  $T_1$  sets their salaries to \$1000 and transaction  $T_2$  sets their salaries to \$2000.

*The following interleaving of the actions  $T_1$  and  $T_2$  occurs:*

- i)  $T_1$  sets A's salary to \$1000, at the same time,  $T_2$  sets B's salary to \$2000.
- ii)  $T_1$  sets B's salary is set to \$2000, at the same time,  $T_2$  sets A's salary to \$2000.

As a result A's salary is set to \$2000 and B's salary is set to \$1000, i.e., the result is not identical

- **Blind-Write:** Neither transaction reads a value before writing it-such a write is called a *blind-write*.

*The above example is the best example of blind write because  $T_1$  and  $T_2$*

*are concentrating only on writing but not on reading.*

### Schedules involving aborted Transactions

- All transactions of aborted transactions are to be undone, and we can therefore imagine that they were never carried out to begin with.
- **Example:** Suppose that transaction  $T_1$  deducts \$100 from account A then immediately before committing A's new value the transaction  $T_2$  reads the current values of accounts A and B and adds 6% interest to each, then commits, but incidentally  $T_1$  is aborted. So, we get incorrect result of transaction  $T_2$  because  $T_1$  was aborted in the middle of the process and  $T_2$  has taken incorrect value of A by  $T_1$  and added 6%. We say that such a schedule is *Unrecoverable Schedule*. The corresponding schedule is shown as follows:

| $T_1$                       | $T_2$                                                              |
|-----------------------------|--------------------------------------------------------------------|
| R(A)<br>A: = A -<br>100W(A) | R(A)<br>A: = A +<br>0.06 A<br>R(B)<br>B:= B+.06<br>BW(B)<br>Commit |
| Abort                       |                                                                    |

- Whereas, a recoverable schedule is one in which transactions read only the changes of committed transactions.

## Serializability

- Basic Assumption – Each transaction, on its own, preserves database consistency
  - i.e. serial execution of transactions preserves database consistency*
- A (possibly concurrent) schedule is *serializable* if it is equivalent to a serial schedule
- Different forms of schedule equivalence give rise to the notions of *conflict serializability* and *view serializability*
- Simplifying assumptions:
  - ignore operations other than read and write instructions*
  - assume that transactions may perform arbitrary computations on data in local buffers between reads and writes*
  - simplified schedules consist only of reads and writes*
- Conflict Serializability**
  - Instructions  $li$  and  $lj$  of transactions  $Ti$  and  $Tj$  respectively, **conflict** if and only if there exists some item  $Q$  accessed by both  $li$  and  $lj$ , and at least one of these instructions wrote  $Q$ .
    - $li = \text{read}(Q), lj = \text{read}(Q)$ .  $li$  and  $lj$  don't conflict.
    - $li = \text{read}(Q), lj = \text{write}(Q)$ . They conflict.
    - $li = \text{write}(Q), lj = \text{read}(Q)$ . They conflict
    - $li = \text{write}(Q), lj = \text{write}(Q)$ . They conflict
  - Intuitively, a conflict between  $li$  and  $lj$  forces a (logical) temporal order between them
  - If  $li$  and  $lj$  are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the ordering
  - If a schedule  $S$  can be transformed into a schedule  $S'$  by a series of swaps of non-conflicting instructions, we say that  $S$  and  $S'$  are **conflict equivalent**.
  - We say that a schedule  $S$  is **conflict serializable** if it is conflict equivalent to a serial schedule

- Example of a schedule that is not conflict serializable:*

| $T_3$    | $T_4$    |
|----------|----------|
| read(Q)  |          |
|          | write(Q) |
| write(Q) |          |

We are unable to swap instructions in the above schedule to obtain either the serial schedule  $\langle T_3, T_4 \rangle$ , or the serial schedule  $\langle T_4, T_3 \rangle$ .

- Schedule 3 below can be transformed into Schedule 1, a serial schedule where  $T_2$  follows  $T_1$ , by series of swaps of

| $T_1$    | $T_2$    |
|----------|----------|
| read(A)  |          |
| write(A) |          |
|          | read(A)  |
|          | write(A) |
| read(B)  |          |
| write(B) |          |
|          | read(B)  |
|          | write(B) |

non-conflicting instructions.

Therefore Schedule 3 is conflict serializable.

### • View Serializability

- Let  $S$  and  $S'$  be two schedules with the same set of transactions.  $S$  and  $S'$  are *view equivalent* if the following three conditions are met, where  $Q$  is a data item and  $T_i$  is a transaction:

1. If  $T_i$  reads the initial value of  $Q$  in schedule  $S$ , then  $T_i$  must, in schedule  $S'$ , also read the initial value of  $Q$
2. If  $T_i$  executes  $\text{read}(Q)$  in schedule  $S$ , and that value was produced by transaction  $T_j$  (if any), then transaction  $T_i$  must in schedule  $S'$  also read the value of  $Q$  that was produced by transaction  $T_j$
3. The transaction (if any) that performs the final write( $Q$ ) operation in schedule  $S$  (for any data item  $Q$ ) must perform the final write( $Q$ ) operation in schedule  $S'$

NB: View equivalence is also based purely on **reads** and **writes**

- A schedule  $S$  is *view serializable* if it is view equivalent to a serial schedule
- Every conflict serializable schedule is also view serializable
- Schedule 9 (from book) — a schedule which is view-serializable but *not* conflict serializable

| $T_3$        | $T_4$        | $T_6$        |
|--------------|--------------|--------------|
| read( $Q$ )  |              |              |
| write( $Q$ ) | write( $Q$ ) |              |
|              |              | write( $Q$ ) |

Every view serializable schedule that is not conflict serializable has *blind writes*

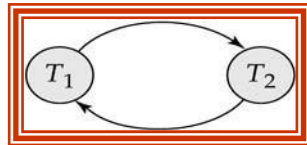
### Other Notions of Serializability

- This schedule produces the same outcome as the serial schedule  $\langle T_1, T_5 \rangle$
- However it is not conflict equivalent or view equivalent to it
- Determining such equivalence requires analysis of operations other than read and write

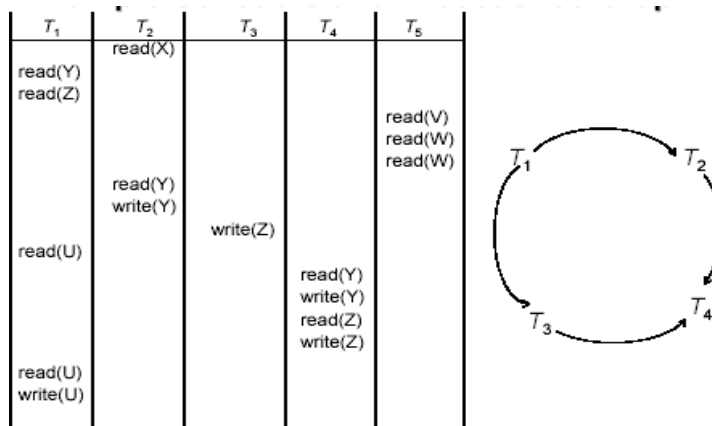
| $T_1$         | $T_5$         |
|---------------|---------------|
| read( $A$ )   |               |
| $A := A - 50$ |               |
| write( $A$ )  |               |
|               | read( $B$ )   |
|               | $B := B - 10$ |
|               | write( $B$ )  |
| read( $B$ )   |               |
| $B := B + 50$ |               |
| write( $B$ )  |               |
|               | read( $A$ )   |
|               | $A := A + 10$ |
|               | write( $A$ )  |

### Testing for Serializability

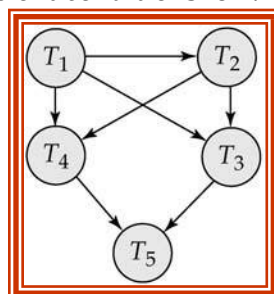
- Consider some schedule of a set of transactions  $T_1, T_2, \dots, T_n$
- Precedence graph:** a directed graph where the vertices are transaction names
- We draw an arc from  $T_i$  to  $T_j$  if the two transaction conflict, and  $T_i$  accessed the data item before  $T_j$
- We may label the arc by the item that was accessed
- Example:**



### Example Schedule and Precedence Graph



- A schedule is **conflict serializable** if and only if its precedence graph is acyclic
  - Cycle-detection algorithms exist which take order  $n^2$  time, where  $n$  is the number of vertices in the graph
  - If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph. This is a linear order consistent with the partial order of the graph. For example, a serializability order for this graph is  $T_2 \rightarrow T_1$   
 $\square T_3 \rightarrow T_4 \rightarrow T_5$
- The precedence graph test for conflict serializability must be modified to apply to a test for **view serializability**
  - The problem of checking if a schedule is view serializable is *NP*-complete. Thus existence of an efficient algorithm is unlikely. However practical algorithms that just check some *sufficient conditions* for view serializability can still be used



*Example of an acyclic precedence graph*

• **Concurrency Control vs. Serializability Tests**

- Goal – to develop concurrency control protocols that will ensure serializability
- These protocols will impose a discipline that avoids nonserializable schedules
- A common concurrency control protocol uses *locks*
  - while one transaction is accessing a data item, no other transaction can modify it
  - require a transaction to lock the item before accessing it
  - two standard lock modes are “shared” (read-only) and “exclusive” (read-write)

**Recoverability**

- Need to address the effect of transaction failures on concurrently running transactions.
- *Recoverable schedule*: if a transaction  $T_j$  reads a data item previously written by a transaction  $T_i$ , the commit operation of  $T_i$  appears before the commit operation of  $T_j$
- The following schedule (Schedule 11) is not recoverable if  $T_9$  commits immediately after the read

| $T_8$    | $T_9$   |
|----------|---------|
| read(A)  |         |
| write(A) |         |
|          | read(A) |
| read(B)  |         |

- If  $T_8$  should abort,  $T_9$  would have read (and possibly shown to the user) an inconsistent database state. Hence database must ensure that schedules are recoverable
- *Cascading rollback* – a single transaction failure leads to a series of transaction rollbacks
- Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)
- If  $T_{10}$  fails,  $T_{11}$  and  $T_{12}$  must also be rolled back
- Can lead to the undoing of a significant amount of work

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

- *Cascadeless schedules* — cascading rollbacks cannot occur; for each pair of transactions  $T_i$  and  $T_j$  such that  $T_j$  reads a data item previously written by  $T_i$ , the commit operation of  $T_i$  appears before the read operation of  $T_j$

- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

### Implementation of Atomicity and Durability

- The **recovery-management** component of a database system implements the support for atomicity and durability.

E.g. the *shadow-database* scheme:

all updates are made on a *shadow copy* of the database

**db\_pointer** is made to point to the updated shadow copy after

the transaction reaches partial commit and all updated pages have been flushed to disk.

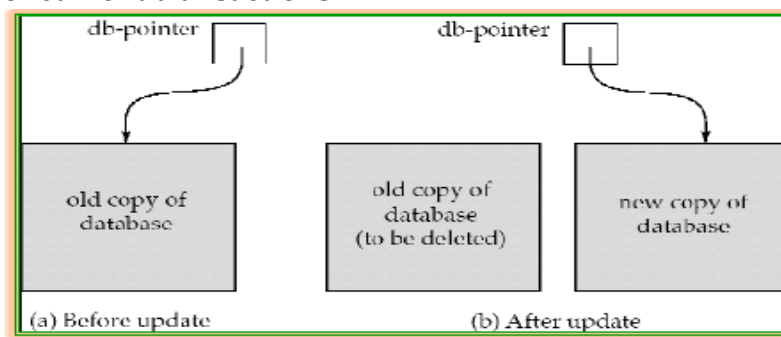
**db\_pointer** always points to the current consistent copy of the database.

In case transaction fails, old consistent copy pointed to by **db\_pointer** can be used, and the shadow copy can be deleted.

The shadow-database scheme:

Assumes that only one transaction is active at a time. Assumes disks do not fail

Does not handle concurrent transactions



### Storage Structure

- **Volatile storage:**
  - does not survive system crashes
  - examples: main memory, cache memory
- **Nonvolatile storage:**
  - survives system crashes
  - examples: disk, tape, flash memory, non-volatile (battery backed up) RAM
- **Stable storage:**
  - a mythical form of storage that survives all failures
  - approximated by maintaining multiple copies on distinct nonvolatile media

## Stable-Storage Implementation

- Maintain multiple copies of each block on separate disks
  - ▣ copies can be at remote sites to protect against disasters such as fire or flooding.
- Failure during data transfer can still result in inconsistent copies: Block transfer can result in
  - ▣ Successful completion
  - ▣ Partial failure: destination block has incorrect information
  - ▣ Total failure: destination block was never updated
- Protecting storage media from failure during data transfer (one solution):
  - ▣ Execute output operation as follows (assuming two copies of each block):
    1. Write the information onto the first physical block.
    2. When the first write successfully completes, write the same information onto the second physical block.
    3. The output is completed only after the second write successfully completes.

## File Organization

A **file** is organized logically as a sequence of records. These records are mapped onto disk blocks. Files are provided as a basic construct in operating systems, so we shall assume the existence of an underlying *file system*.

Each file is also logically partitioned into fixed-length storage units called **blocks**, which are the units of both storage allocation and data transfer.

Most databases use block sizes of 4 to 8 kilobytes by default, but many databases allow the block size to be specified when a database instance is created.

Two possible approaches to store records: • Record size is fixed • Record size is variable

## Fixed-Length Records

As an example, let us consider a file of *instructor* records for our university database. Each record of this file is defined (in pseudocode) as:

```
type instructor = record
 ID varchar (5);
 name varchar(20);
 dept_name varchar (20);
 salary numeric (8,2);
end
```

We allocate the maximum number of bytes that each attribute can hold. Then, the instructor record is 53 bytes long.

A simple approach is to use the first 53 bytes for the first record, the next 53 bytes for the second record, and so on as shown below figure.

|           |       |            |            |       |
|-----------|-------|------------|------------|-------|
| record 0  | 10101 | Srinivasan | Comp. Sci. | 65000 |
| record 1  | 12121 | Wu         | Finance    | 90000 |
| record 2  | 15151 | Mozart     | Music      | 40000 |
| record 3  | 22222 | Einstein   | Physics    | 95000 |
| record 4  | 32343 | El Said    | History    | 60000 |
| record 5  | 33456 | Gold       | Physics    | 87000 |
| record 6  | 45565 | Katz       | Comp. Sci. | 75000 |
| record 7  | 58583 | Califieri  | History    | 62000 |
| record 8  | 76543 | Singh      | Finance    | 80000 |
| record 9  | 76766 | Crick      | Biology    | 72000 |
| record 10 | 83821 | Brandt     | Comp. Sci. | 92000 |
| record 11 | 98345 | Kim        | Elec. Eng. | 80000 |

There are two problems with this simple approach:

1. Unless the block size happens to be a multiple of 53 (which is unlikely), some records will cross block boundaries. That is, part of the record will be stored in one block and part in another. It would thus require two block accesses to read or write such a record.
2. It is difficult to delete a record from this structure. The space occupied by the record to be deleted must be filled with some other record of the file, or we must have a way of marking deleted records so that they can be ignored.

To avoid the first problem, we allocate only as many records to a block as would fit entirely in the block (this number can be computed easily by dividing the block size by the record size, and discarding the fractional part). Any remaining bytes of each block are left unused.

To avoid the second problem, When a record is deleted, we could move the record that came after it into the space formerly occupied by the deleted record, and so on, until every record following the deleted record has been moved ahead shown in below figure.

|           |       |            |            |       |
|-----------|-------|------------|------------|-------|
| record 0  | 10101 | Srinivasan | Comp. Sci. | 65000 |
| record 1  | 12121 | Wu         | Finance    | 90000 |
| record 2  | 15151 | Mozart     | Music      | 40000 |
| record 4  | 32343 | El Said    | History    | 60000 |
| record 5  | 33456 | Gold       | Physics    | 87000 |
| record 6  | 45565 | Katz       | Comp. Sci. | 75000 |
| record 7  | 58583 | Califieri  | History    | 62000 |
| record 8  | 76543 | Singh      | Finance    | 80000 |
| record 9  | 76766 | Crick      | Biology    | 72000 |
| record 10 | 83821 | Brandt     | Comp. Sci. | 92000 |
| record 11 | 98345 | Kim        | Elec. Eng. | 80000 |

Such an approach requires moving a large number of records. It might be easier simply to move the final record of the file into the space occupied by the deleted record shown in below figure.

|           |       |            |            |       |
|-----------|-------|------------|------------|-------|
| record 0  | 10101 | Srinivasan | Comp. Sci. | 65000 |
| record 1  | 12121 | Wu         | Finance    | 90000 |
| record 2  | 15151 | Mozart     | Music      | 40000 |
| record 11 | 98345 | Kim        | Elec. Eng. | 80000 |
| record 4  | 32343 | El Said    | History    | 60000 |
| record 5  | 33456 | Gold       | Physics    | 87000 |
| record 6  | 45565 | Katz       | Comp. Sci. | 75000 |
| record 7  | 58583 | Califieri  | History    | 62000 |
| record 8  | 76543 | Singh      | Finance    | 80000 |
| record 9  | 76766 | Crick      | Biology    | 72000 |
| record 10 | 83821 | Brandt     | Comp. Sci. | 92000 |

It is undesirable to move records to occupy the space freed by a deleted record, since doing so requires additional block accesses. Since insertions tend to be more frequent than deletions, it is acceptable to leave open the space occupied by the deleted record, and to wait for a subsequent insertion before reusing the space.

A simple marker on a deleted record is not sufficient, since it is hard to find this available space when an insertion is being done. Thus, we need to introduce an additional structure.

At the beginning of the file, we allocate a certain number of bytes as a **file header**. The header will contain a variety of information about the file. For now, all we need to store there is the address of the first record whose contents are deleted.

We use this first record to store the address of the second available record, and so on. Intuitively, we can think of these stored addresses as *pointers*, since they point to the location of a record. The deleted records thus form a linked list, which is often referred to as a **free list**.

Below figure shows with the free list, after records 1, 4, and 6 have been deleted. On insertion of a new record, we use the record.

|           |       |            |            |       |
|-----------|-------|------------|------------|-------|
| header    |       |            |            |       |
| record 0  | 10101 | Srinivasan | Comp. Sci. | 65000 |
| record 1  |       |            |            |       |
| record 2  | 15151 | Mozart     | Music      | 40000 |
| record 3  | 22222 | Einstein   | Physics    | 95000 |
| record 4  |       |            |            |       |
| record 5  | 33456 | Gold       | Physics    | 87000 |
| record 6  |       |            |            |       |
| record 7  | 58583 | Califieri  | History    | 62000 |
| record 8  | 76543 | Singh      | Finance    | 80000 |
| record 9  | 76766 | Crick      | Biology    | 72000 |
| record 10 | 83821 | Brandt     | Comp. Sci. | 92000 |
| record 11 | 98345 | Kim        | Elec. Eng. | 80000 |

## Variable-Length Records

Variable-length records arise in database systems in several ways:

- Storage of multiple record types in a file.
- Record types that allow variable lengths for one or more fields.
- Record types that allow repeating fields, such as arrays or multisets.

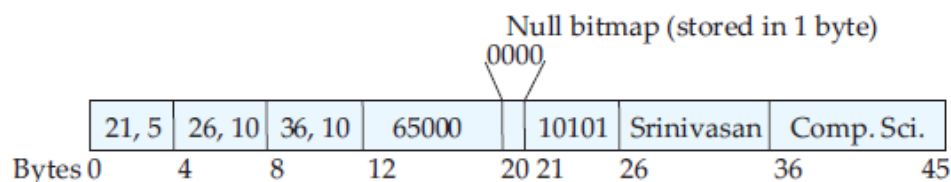
Different techniques for implementing variable-length records exist. Two different problems must be solved by any such technique:

- How to represent a single record in such a way that individual attributes can be extracted easily.
- How to store variable-length records within a block, such that records in a block can be extracted easily.

## Representation of variable-length record

The representation of a record with variable-length attributes typically has two parts: an initial part with fixed length attributes, followed by data for variable length attributes. Fixed-length attributes, such as numeric values, dates, or fixed length character strings are allocated as many bytes as required to store their value. Variable-length attributes, such as varchar types, are represented in the initial part of the record by a pair (*offset*, *length*), where *offset* denotes where the data for that attribute begins within the record, and *length* is the length in bytes of the variable-sized attribute. The values for these attributes are stored consecutively, after the initial fixed-length part of the record. Thus, the initial part of the record stores a fixed size of information about each attribute, whether it is fixed-length or variable-length.

An example of such a record representation is shown in below figure. The figure shows an *instructor* record, whose first three attributes *ID*, *name*, and *dept name* are variable-length strings, and whose fourth attribute *salary* is a fixed-sized number. We assume that the offset and length values are stored in two bytes each, for a total of 4 bytes per attribute. The *salary* attribute is assumed to be stored in 8 bytes, and each string takes as many bytes as it has characters.



The figure also illustrates the use of a **null bitmap**, which indicates which attributes of the record have a null value. In this particular record, if the salary were null, the fourth bit of the bitmap would be set to 1, and the salary value stored in bytes 12 through 19 would be ignored.

## Storing variable-length records in a block

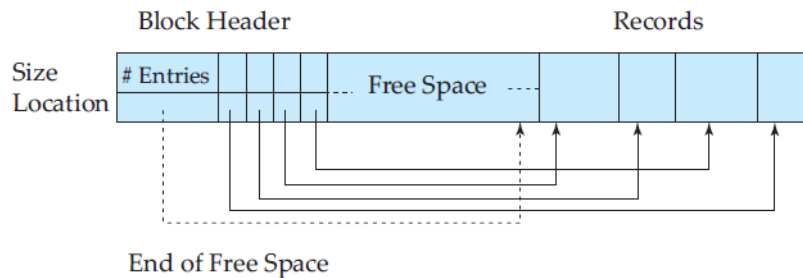
The **slotted-page structure** is commonly used for organizing records within a block. There is a header at the beginning of each block, containing the following information:

1. The number of record entries in the header.
2. The end of free space in the block.
3. An array whose entries contain the location and size of each record.

The actual records are allocated *contiguously* in the block, starting from the end of the block. The free space in the block is contiguous, between the final entry in the header array,

and the first record. If a record is inserted, space is allocated for it at the end of free space, and an entry containing its size and location is added to the header.

If a record is deleted, the space that it occupies is freed, and its entry is set to *deleted* (its size is set to -1, for example). Further, the records in the block before the deleted record are moved, so that the free space created by the deletion gets occupied, and all free space is again between the final entry in the header array and the first record. The end-of-free-space pointer in the header is appropriately updated as well. Records can be grown or shrunk by similar techniques, as long as there is space in the block. The cost of moving the records is not too high, since the size of a block is limited: typical values are around 4 to 8 kilobytes.



### Organization of Records in Files

A relation is a set of records. Given a set of records, the next question is how to organize them in a file. Several of the possible ways of organizing records in files are:

- **Heap file organization.** Any record can be placed anywhere in the file where there is space for the record. There is no ordering of records. Typically, there is a single file for each relation.
- **Sequential file organization.** Records are stored in sequential order, according to the value of a “search key” of each record.
- **Hashing file organization.** A hash function is computed on some attribute of each record. The result of the hash function specifies in which block of the

### Sequential File Organization

A **sequential file** is designed for efficient processing of records in sorted order based on some search key. A **search key** is any attribute or set of attributes; it need not be the primary key, or even a superkey. To permit fast retrieval of records in search-key order, we chain together records by pointers. The pointer in each record points to the next record in search-key order. Furthermore, to minimize the number of block accesses in sequential file processing, we store records physically in search-key order, or as close to search-key order as possible.

Blow figure shows a sequential file of *instructor* records taken from our university example. In that example, the records are stored in search-key order, using *ID* as the search key.

|       |            |            |       |  |
|-------|------------|------------|-------|--|
| 10101 | Srinivasan | Comp. Sci. | 65000 |  |
| 12121 | Wu         | Finance    | 90000 |  |
| 15151 | Mozart     | Music      | 40000 |  |
| 22222 | Einstein   | Physics    | 95000 |  |
| 32343 | El Said    | History    | 60000 |  |
| 33456 | Gold       | Physics    | 87000 |  |
| 45565 | Katz       | Comp. Sci. | 75000 |  |
| 58583 | Califieri  | History    | 62000 |  |
| 76543 | Singh      | Finance    | 80000 |  |
| 76766 | Crick      | Biology    | 72000 |  |
| 83821 | Brandt     | Comp. Sci. | 92000 |  |
| 98345 | Kim        | Elec. Eng. | 80000 |  |



It is difficult, however, to maintain physical sequential order as records are inserted and deleted, since it is costly to move many records as a result of a single insertion or deletion. We can manage deletion by using pointer chains, as we saw previously. For insertion, we apply the following rules:

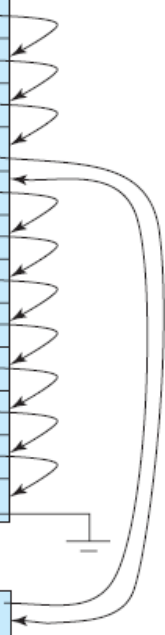
1. Locate the record in the file that comes before the record to be inserted in search-key order.
2. If there is a free record (that is, space left after a deletion) within the same block as this record, insert the new record there. Otherwise, insert the new record in an *overflow block*. In either case, adjust the pointers so as to chain together the records in search-key order.

Below figure shows the record after the insertion of the record (32222, Verdi, Music, 48000).

|       |            |            |       |  |
|-------|------------|------------|-------|--|
| 10101 | Srinivasan | Comp. Sci. | 65000 |  |
| 12121 | Wu         | Finance    | 90000 |  |
| 15151 | Mozart     | Music      | 40000 |  |
| 22222 | Einstein   | Physics    | 95000 |  |
| 32343 | El Said    | History    | 60000 |  |
| 33456 | Gold       | Physics    | 87000 |  |
| 45565 | Katz       | Comp. Sci. | 75000 |  |
| 58583 | Califieri  | History    | 62000 |  |
| 76543 | Singh      | Finance    | 80000 |  |
| 76766 | Crick      | Biology    | 72000 |  |
| 83821 | Brandt     | Comp. Sci. | 92000 |  |
| 98345 | Kim        | Elec. Eng. | 80000 |  |

|       |       |       |       |  |
|-------|-------|-------|-------|--|
| 32222 | Verdi | Music | 48000 |  |
|-------|-------|-------|-------|--|



## Introduction to Indexing

An index for a file in a database system works in much the same way as the index in this textbook. If we want to learn about a particular topic (specified by a word or a phrase) in this textbook, we can search for the topic in the index at the back of the book, find the pages where it occurs, and then read the pages to find the information for which we are looking.

Database-system indices play the same role as book indices in libraries. For example, to retrieve a student record given an ID, the database system would look up an index to find on which disk block the corresponding record resides, and then fetch the disk block, to get the appropriate student record.

There are two basic kinds of indices:

- **Ordered indices.** Based on a sorted ordering of the values.
- **Hash indices.** Based on a uniform distribution of values across a range of buckets. The bucket to which a value is assigned is determined by a function, called a *hash function*.

## Index Evaluation Metrics

- **Access types:** The types of access that are supported efficiently. Access types can include finding records with a specified attribute value and finding records whose attribute values fall in a specified range.
- **Access time:** The time it takes to find a particular data item, or set of items, using the technique in question.
- **Insertion time:** The time it takes to insert a new data item. This value includes the time it takes to find the correct place to insert the new data item, as well as the time it takes to update the index structure.
- **Deletion time:** The time it takes to delete a data item. This value includes the time it takes to find the item to be deleted, as well as the time it takes to update the index structure.
- **Space overhead:** The additional space occupied by an index structure. Provided that the amount of additional space is moderate, it is usually worthwhile to sacrifice the space to achieve improved performance.

An attribute or set of attributes used to look up records in a file is called a **search key**.

## Ordered Indices

To gain fast random access to records in a file, we can use an index structure. Each index structure is associated with a particular search key. Just like the index of a book or a library catalog, an ordered index stores the values of the search keys in sorted order, and associates with each search key the records that contain it.

A file may have several indices, on different search keys. If the file containing the records is sequentially ordered, a **clustering index** is an index whose search key also defines the sequential order of the file.

Clustering indices are also called **primary indices**; the term primary index may appear to denote an index on a primary key, but such indices can in fact be built on any search key.

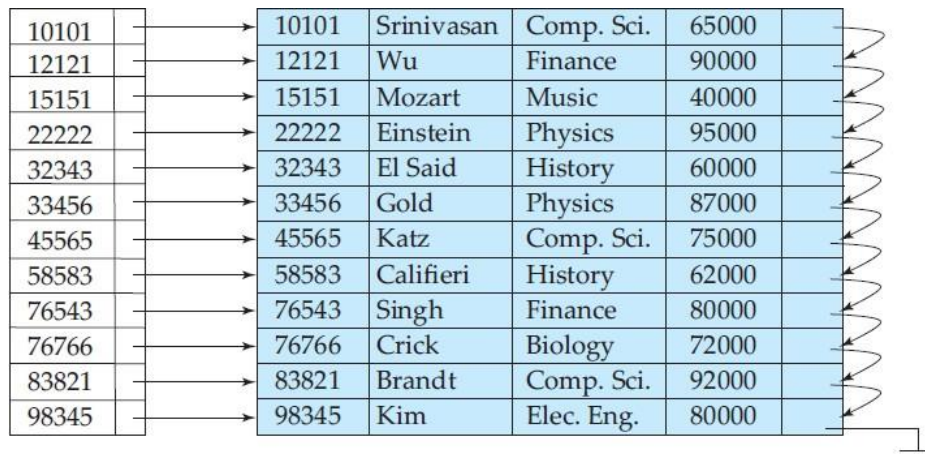
Indices whose search key specifies an order different from the sequential order of the file are called **nonclustering indices**, or **secondary indices**.

## Dense and Sparse Indices

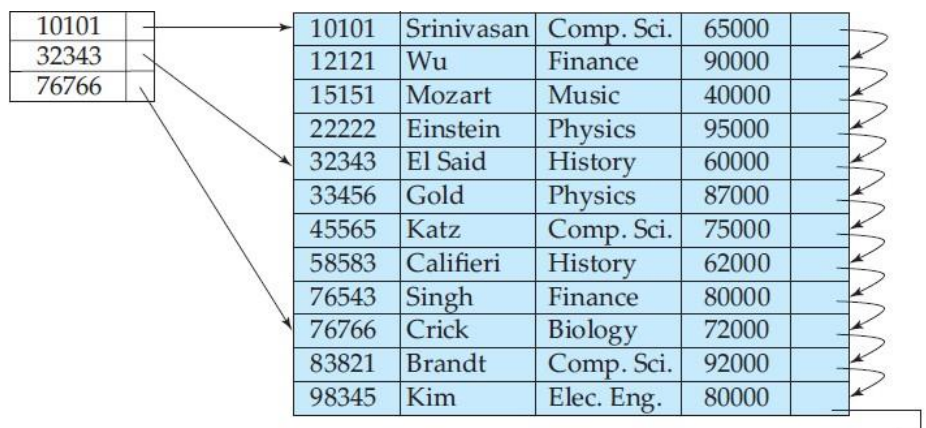
An **index entry**, or **index record**, consists of a search-key value and pointers to one or more records with that value as their search-key value. The pointer to a record consists of the identifier of a disk block and an offset within the disk block to identify the record within the block.

There are two types of ordered indices that we can use:

- **Dense index:** In a dense index, an index entry appears for every search-key value in the file. In a dense clustering index, the index record contains the search-key value and a pointer to the first data record with that search-key value. The rest of the records with the same search-key value would be stored sequentially after the first record, since, because the index is a clustering one, records are sorted on the same search key. In a dense nonclustering index, the index must store a list of pointers to all records with the same search-key value.
- **Sparse index:** In a sparse index, an index entry appears for only some of the search-key values. Sparse indices can be used only if the relation is stored in sorted order of the search key, that is, if the index is a clustering index. As is true in dense indices, each index entry contains a search-key value and a pointer to the first data record with that search-key value. To locate a record, we find the index entry with the largest search-key value that is less than or equal to the search-key value for which we are looking. We start at the record pointed to by that index entry, and follow the pointers in the file until we find the desired record.



Dense index

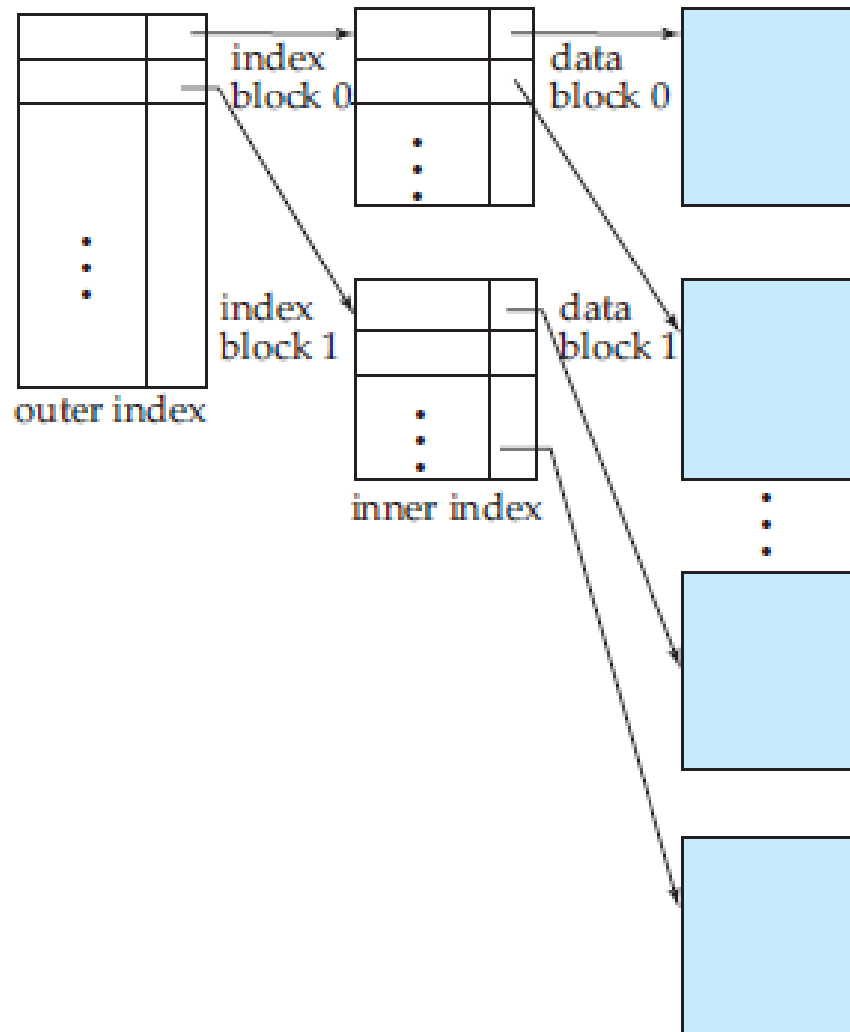


Sparse index

## Multilevel Indices

If an index is small enough to be kept entirely in main memory, the search time to find an entry is low. However, if the index is so large that not all of it can be kept in memory, index blocks must be fetched from disk when required. (Even if an index is smaller than the main memory of a computer, main memory is also required for a number of other tasks, so it may not be possible to keep the entire index in memory.) The search for an entry in the index then requires several disk-block reads.

We treat the index just as we would treat any other sequential file, and construct a sparse outer index on the original index, which we now call the inner index, as shown in below figure. Note that the index entries are always in sorted order, allowing the outer index to be sparse. To locate a record, we first use binary search on the outer index to find the record for the largest search-key value less than or equal to the one that we desire. The pointer points to a block of the inner index. We scan this block until we find the record that has the largest search-key value less than or equal to the one that we desire. The pointer in this record points to the block of the file that contains the record for which we are looking.



## Two level sparse index

## Index Update

Regardless of what form of index is used, every index must be updated whenever a record is either inserted into or deleted from the file.

We first describe algorithms for updating single-level indices.

- **Insertion.** First, the system performs a lookup using the search-key value that appears in the record to be inserted. The actions the system takes next depend on whether the index is dense or sparse:

- **Dense indices:**

1. If the search-key value does not appear in the index, the system inserts an index entry with the search-key value in the index at the appropriate position.
2. Otherwise the following actions are taken:
  - a. If the index entry stores pointers to all records with the same search key value, the system adds a pointer to the new record in the index entry.
  - b. Otherwise, the index entry stores a pointer to only the first record with the search-key value. The system then places the record being inserted after the other records with the same search-key values.

- **Sparse indices:** We assume that the index stores an entry for each block. If the system creates a new block, it inserts the first search-key value (in search-key order) appearing in the new block into the index. On the other hand, if the new record has the least search-key value in its block, the system updates the index entry pointing to the block; if not, the system makes no change to the index.

- **Deletion.** To delete a record, the system first looks up the record to be deleted. The actions the system takes next depend on whether the index is dense or sparse:

- **Dense indices:**

1. If the deleted record was the only record with its particular search-key value, then the system deletes the corresponding index entry from the index.
2. Otherwise the following actions are taken:
  - a. If the index entry stores pointers to all records with the same search key value, the system deletes the pointer to the deleted record from the index entry.
  - b. Otherwise, the index entry stores a pointer to only the first record with the search-key value. In this case, if the deleted record was the first record with the search-key value, the system updates the index entry to point to the next record.

- **Sparse indices:**

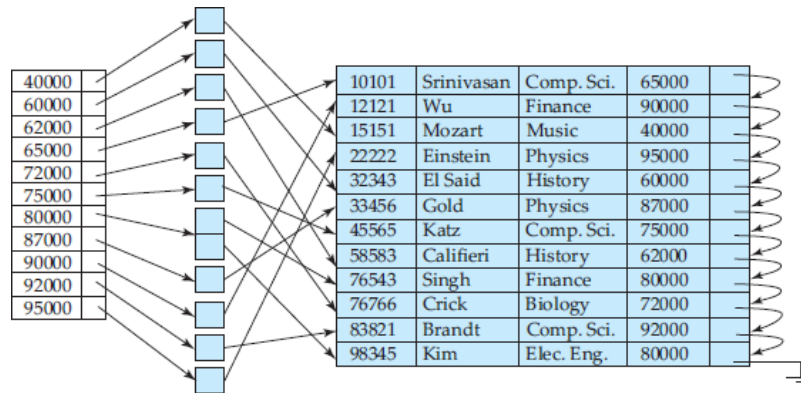
1. If the index does not contain an index entry with the search-key value of the deleted record, nothing needs to be done to the index.
2. Otherwise the system takes the following actions:
  - a. If the deleted record was the only record with its search key, the system replaces the corresponding index record with an index record for the next search-key value (in search-key order). If the next search-key value already has an index entry, the entry is deleted instead of being replaced.
  - b. Otherwise, if the index entry for the search-key value points to the record being deleted, the system updates the index entry to point to the next record with the same search-key value.

## Secondary Indices

Secondary indices must be dense, with an index entry for every search-key value, and a pointer to every record in the file. A clustering index may be sparse, storing only some of the search-key values, since it is always possible to find records with intermediate search-key values by a sequential access to a part of the file, as described earlier. If a secondary index stores only some of the search-key values, records with intermediate search-key values may be anywhere in the file and, in general, we cannot find them without searching the entire file.

We can use an extra level of indirection to implement secondary indices on search keys that are not candidate keys. The pointers in such a secondary index do not point directly to the file. Instead, each points to a bucket that contains pointers to the file.

Below figure shows the structure of a secondary index that uses an extra level of indirection on the *instructor* file, on the search key *salary*.



## B+ Tree Index Files

The main disadvantage of the index-sequential file organization is that performance degrades as the file grows, both for index lookups and for sequential scans through the data.

The **B+ tree** index structure is the most widely used of several index structures that maintain their efficiency despite insertion and deletion of data.

A B+ tree index takes the form of a **balanced tree** in which every path from the root of the tree to a leaf of the tree is of the same length. Each non leaf node in the tree has between  $n/2$  and  $n$  children, where  $n$  is fixed for a particular tree.

### Structure of a B+ Tree

A B<sup>+</sup>-tree index is a multilevel index, but it has a structure that differs from that of the multilevel index-sequential file. Figure 11.7 shows a typical node of a B<sup>+</sup>-tree. It contains up to  $n - 1$  search-key values  $K_1, K_2, \dots, K_{n-1}$ , and  $n$  pointers  $P_1, P_2, \dots, P_n$ . The search-key values within a node are kept in sorted order; thus, if  $i < j$ , then  $K_i < K_j$ .

We consider first the structure of the **leaf nodes**. For  $i = 1, 2, \dots, n-1$ , pointer  $P_i$  points to a file record with search-key value  $K_i$ . Pointer  $P_n$  has a special purpose that we shall discuss shortly.

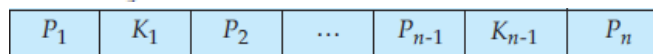


Figure 11.7 Typical node of a B<sup>+</sup>-tree.

Figure 11.8 shows one leaf node of a B<sup>+</sup>-tree for the *instructor* file, in which we have chosen  $n$  to be 4, and the search key is *name*.

Now that we have seen the structure of a leaf node, let us consider how search-key values are assigned to particular nodes. Each leaf can hold up to  $n - 1$  values. We allow leaf nodes to contain as few as  $\lceil (n - 1)/2 \rceil$  values. With  $n = 4$  in our example B<sup>+</sup>-tree, each leaf must contain at least 2 values, and at most 3 values.

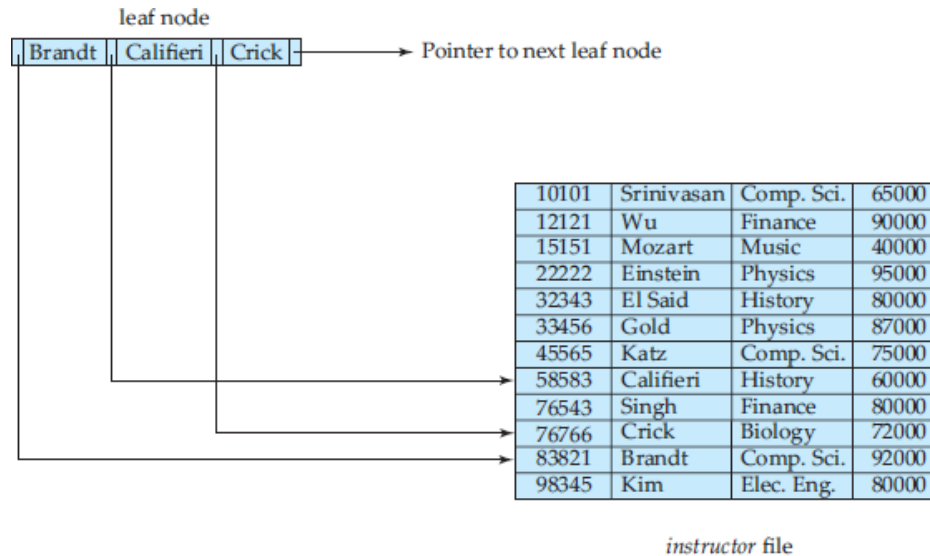


Figure 11.8 A leaf node for *instructor* B<sup>+</sup>-tree index ( $n = 4$ ).

The **nonleaf nodes** of the B<sup>+</sup>-tree form a multilevel (sparse) index on the leaf nodes. The structure of nonleaf nodes is the same as that for leaf nodes, except that all pointers are pointers to tree nodes. A nonleaf node may hold up to  $n$  pointers, and *must* hold at least  $\lceil n/2 \rceil$  pointers. The number of pointers in a node is called the *fanout* of the node. Nonleaf nodes are also referred to as **internal nodes**.

Let us consider a node containing  $m$  pointers ( $m \leq n$ ). For  $i = 2, 3, \dots, m - 1$ , pointer  $P_i$  points to the subtree that contains search-key values less than  $K_i$  and greater than or equal to  $K_{i-1}$ . Pointer  $P_m$  points to the part of the subtree that contains those key values greater than or equal to  $K_{m-1}$ , and pointer  $P_1$  points to the part of the subtree that contains those search-key values less than  $K_1$ .

Unlike other nonleaf nodes, the root node can hold fewer than  $\lceil n/2 \rceil$  pointers; however, it must hold at least two pointers, unless the tree consists of only one node. It is always possible to construct a B<sup>+</sup>-tree, for any  $n$ , that satisfies the preceding requirements.

Figure 11.9 shows a complete B<sup>+</sup>-tree for the *instructor* file (with  $n = 4$ ). We have shown instructor names abbreviated to 3 characters in order to depict the tree clearly; in reality, the tree nodes would contain the full names. We have also omitted null pointers for simplicity; any pointer field in the figure that does not have an arrow is understood to have a null value.

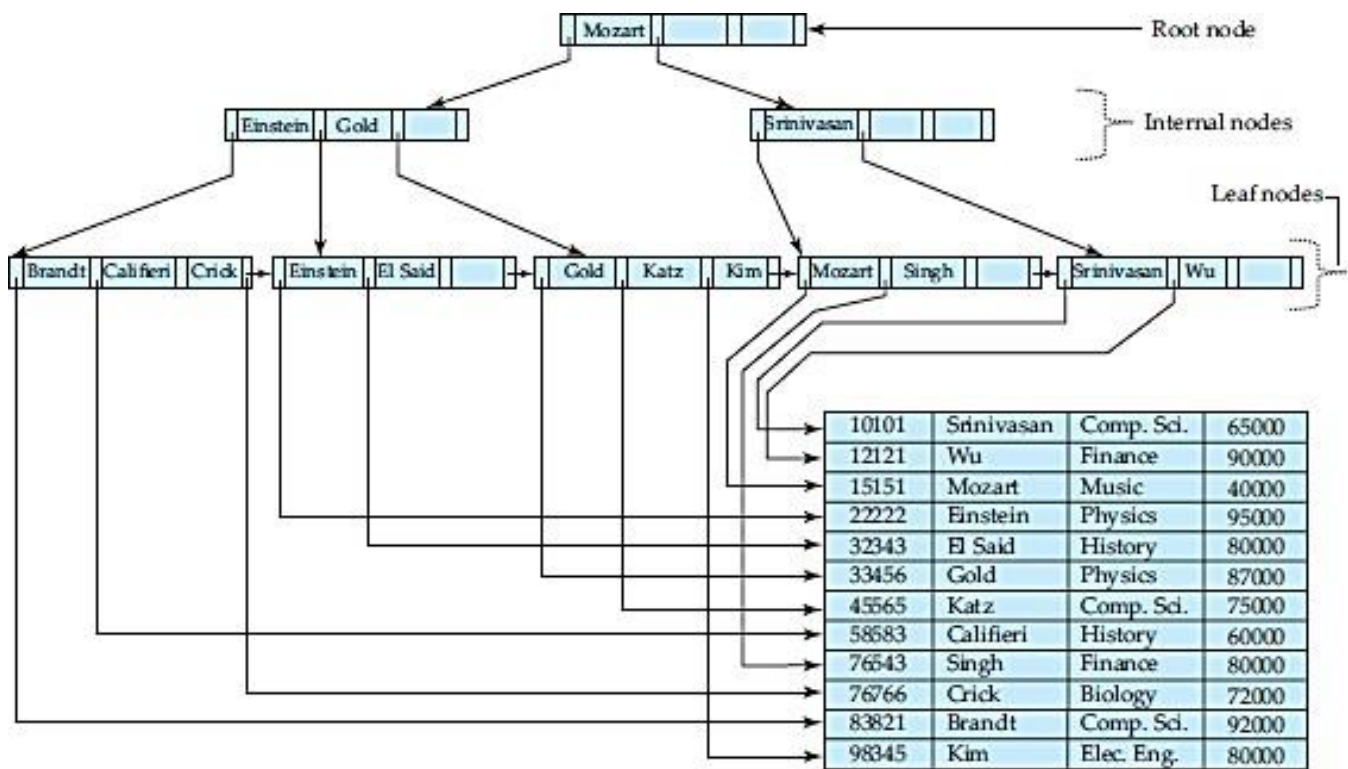


Figure 11.9 B<sup>+</sup>-tree for *instructor* file ( $n = 4$ ).

## Queries on B<sup>+</sup> Trees

Let us consider how we process queries on a B<sup>+</sup>-tree. Suppose that we wish to find records with a search-key value of  $V$ .

Intuitively, the function starts at the root of the tree, and traverses the tree down until it reaches a leaf node that would contain the specified value if it exists in the tree. Specifically, starting with the root as the current node, the function repeats the following steps until a leaf node is reached. First, the current node is examined, looking for the smallest  $i$  such that search-key value  $K_i$  is greater than or equal to  $V$ . Suppose such a value is found; then, if  $K_i$  is equal to  $V$ , the current node is set to the node pointed to by  $P_{i+1}$ , otherwise  $K_i > V$ , and the current node is set to the node pointed to by  $P_i$ . If no such value  $K_i$  is found, then clearly  $V > K_{m-1}$ , where  $P_m$  is the last non null pointer in the node. In this case the current node is set to that pointed to by  $P_m$ . The above procedure is repeated, traversing down the tree until a leaf node is reached.

At the leaf node, if there is a search-key value equal to  $V$ , let  $K_i$  be the first such value; pointer  $P_i$  directs us to a record with search-key value  $K_i$ . The function then returns the leaf node  $L$  and the index  $i$ . If no search-key with value  $V$  is found in the leaf node, no record with key value  $V$  exists in the relation, and function find returns null, to indicate failure.

```

function find(value V)
/* Returns leaf node C and index i such that C.Pi points to first record
* with search key value V */
 Set C = root node
 while (C is not a leaf node) begin
 Let i = smallest number such that V ≤ C.Ki
 if there is no such number i then begin
 Let Pm = last non-null pointer in the node
 Set C = C.Pm
 end
 else if (V = C.Ki)
 then Set C = C.Pi+1
 else C = C.Pi /* V < C.Ki */
 end
 /* C is a leaf node */
 Let i be the least value such that Ki = V
 if there is such a value i
 then return (C, i)
 else return null ; /* No record with key value V exists */

```

### Updates on B<sup>+</sup> Trees

When a record is inserted into, or deleted from a relation, indices on the relation must be updated correspondingly.

### Insertion

We now consider an example of insertion in which a node must be split. Assume that a record is inserted on the *instructor* relation, with the *name* value being Adams. We then need to insert an entry for “Adams” into the B<sup>+</sup>-tree of Figure 11.9. Using the algorithm for lookup, we find that “Adams” should appear in the leaf node containing “Brandt”, “Califieri”, and “Crick.” There is no room in this leaf to insert the search-key value “Adams.” Therefore, the node is *split* into two nodes. Figure 11.12 shows the two leaf nodes that result from the split of the leaf node on inserting “Adams”. The search-key values “Adams” and “Brandt” are in one leaf, and “Califieri” and “Crick” are in the other. In general, we take the *n* search-key values (the *n* – 1 values in the leaf node plus the value being inserted), and put the first  $\lceil n/2 \rceil$  in the existing node and the remaining values in a newly created node.

Having split a leaf node, we must insert the new leaf node into the B<sup>+</sup>-tree structure. In our example, the new node has “Califieri” as its smallest search-key value. We need to insert an entry with this search-key value, and a pointer to the new node, into the parent of the leaf node that was split. The B<sup>+</sup>-tree of Figure 11.13 shows the result of the insertion. It was possible to perform this insertion with no further node split, because there was room in the parent node for the new entry. If there were no room, the parent would have had to be split, requiring an entry to be added to its parent. In the worst case, all nodes along the path to the root must be split. If the root itself is split, the entire tree becomes deeper.

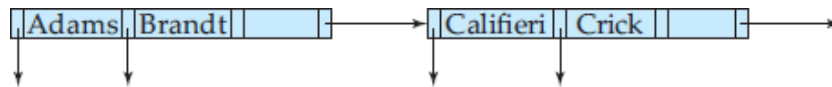


Figure 11.12 Split of leaf node on insertion of “Adams”

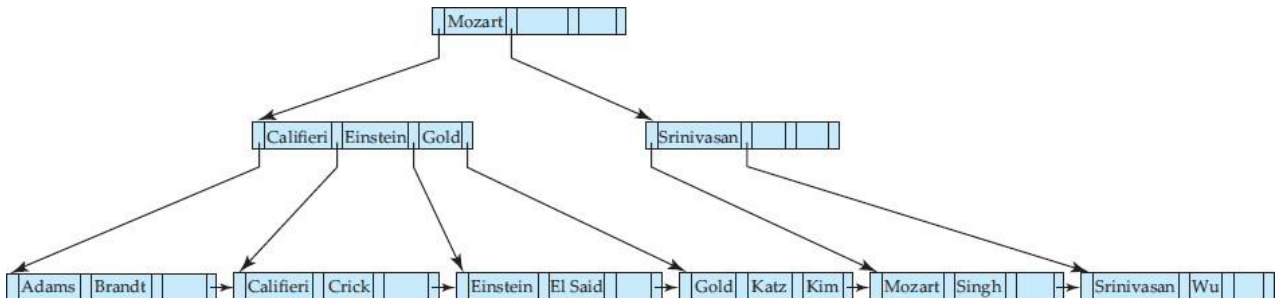


Figure 11.13 Insertion of “Adams” into the B<sup>+</sup>-tree of Figure 11.9.

```

procedure insert(value K, pointer P)
 if (tree is empty) create an empty leaf node L, which is also the root
 else Find the leaf node L that should contain key value K
 if (L has less than $n - 1$ key values)
 then insert_in_leaf (L, K, P)
 else begin /* L has $n - 1$ key values already, split it */
 Create node L'
 Copy $L.P_1 \dots L.K_{n-1}$ to a block of memory T that can
 hold n (pointer, key-value) pairs
 insert_in_leaf (T, K, P)
 Set $L'.P_n = L.P_n$; Set $L.P_n = L'$
 Erase $L.P_1$ through $L.K_{n-1}$ from L
 Copy $T.P_1$ through $T.K_{\lceil n/2 \rceil}$ from T into L starting at $L.P_1$
 Copy $T.P_{\lceil n/2 \rceil + 1}$ through $T.K_n$ from T into L' starting at $L'.P_1$
 Let K' be the smallest key-value in L'
 insert_in_parent(L, K', L')
 end

procedure insert_in_leaf (node L, value K, pointer P)
 if ($K < L.K_1$)
 then insert P, K into L just before $L.P_1$
 else begin
 Let K_i be the highest value in L that is less than K
 Insert P, K into L just after $T.K_i$
 end

```

```

procedure insert_in_parent(node N , value K' , node N')
 if (N is the root of the tree)
 then begin
 Create a new node R containing N , K' , N' /* N and N' are pointers */
 Make R the root of the tree
 return
 end
 Let $P = \text{parent}(N)$
 if (P has less than n pointers)
 then insert (K' , N') in P just after N
 else begin /* Split P */
 Copy P to a block of memory T that can hold P and (K' , N')
 Insert (K' , N') into T just after N
 Erase all entries from P ; Create node P'
 Copy $T.P_1 \dots T.P_{\lceil n/2 \rceil}$ into P
 Let $K'' = T.K_{\lceil n/2 \rceil}$
 Copy $T.P_{\lceil n/2 \rceil + 1} \dots T.P_{n+1}$ into P'
 insert_in_parent(P , K'' , P')
 end

```

## Deletion

We now consider deletions that cause tree nodes to contain too few pointers. First, let us delete “Srinivasan” from the  $B^+$ -tree of Figure 11.13. The resulting  $B^+$ -tree appears in Figure 11.16. We now consider how the deletion is performed. We first locate the entry for “Srinivasan” by using our lookup algorithm. When we delete the entry for “Srinivasan” from its leaf node, the node is left with only one entry, “Wu”. Since, in our example,  $n = 4$  and  $1 < \lceil (n - 1)/2 \rceil$ , we must either merge the node with a sibling node, or redistribute the entries between the nodes, to ensure that each node is at least half-full. In our example, the underfull node with the entry for “Wu” can be merged with its left sibling node. We merge the nodes by moving the entries from both the nodes into the left sibling, and deleting the now empty right sibling. Once the node is deleted, we must also delete the entry in the parent node that pointed to the just deleted node.

In our example, the entry to be deleted is (Srinivasan,  $n3$ ), where  $n3$  is a pointer to the leaf containing “Srinivasan”. (In this case the entry to be deleted in the nonleaf node happens to be the same value as that deleted from the leaf; that would not be the case for most deletions.) After deleting the above entry, the parent node, which had a search key value “Srinivasan” and two pointers, now has one pointer (the leftmost pointer in the node) and no search-key values. Since  $1 < \lceil n/2 \rceil$  for  $n = 4$ , the parent node is underfull. (For larger  $n$ , a node that becomes underfull would still have some values as well as pointers.)

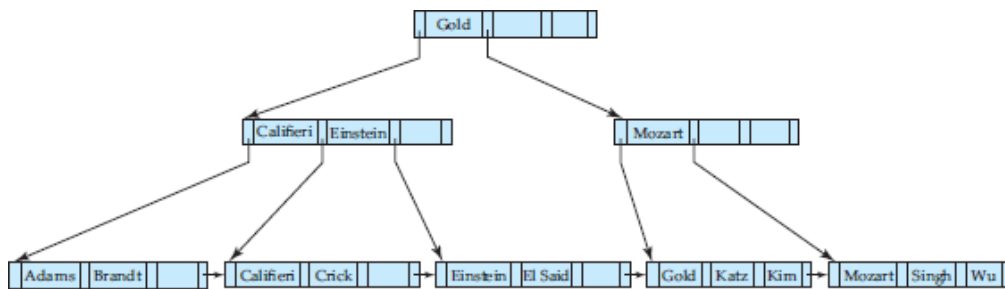


Figure 11.16 Deletion of “Srinivasan” from the B<sup>+</sup>-tree of Figure 11.13.

```

procedure delete(value K, pointer P)
 find the leaf node L that contains (K, P)
 delete_entry(L, K, P)

procedure delete_entry(node N, value K, pointer P)
 delete (K, P) from N
 if (N is the root and N has only one remaining child)
 then make the child of N the new root of the tree and delete N
 else if (N has too few values/pointers) then begin
 Let N' be the previous or next child of parent(N)
 Let K' be the value between pointers N and N' in parent(N)
 if (entries in N and N' can fit in a single node)
 then begin /* Coalesce nodes */
 if (N is a predecessor of N') then swap_variables(N, N')
 if (N is not a leaf)
 then append K' and all pointers and values in N to N'
 else append all (Ki, Pi) pairs in N to N'; set N'.Pn = N.Pn
 delete_entry(parent(N), K', N); delete node N
 end
 else begin /* Redistribution: borrow an entry from N' */
 if (N' is a predecessor of N) then begin
 if (N is a nonleaf node) then begin
 let m be such that N'.Pm is the last pointer in N'
 remove (N'.Km-1, N'.Pm) from N'
 insert (N'.Pm, K') as the first pointer and value in N,
 by shifting other pointers and values right
 replace K' in parent(N) by N'.Km-1
 end
 else begin
 let m be such that (N'.Pm, N'.Km) is the last pointer/value
 pair in N'
 remove (N'.Pm, N'.Km) from N'
 insert (N'.Pm, N'.Km) as the first pointer and value in N,
 by shifting other pointers and values right
 replace K' in parent(N) by N'.Km
 end
 end
 else ... symmetric to the then case ...
 end
 end

```

## Hash Organization

For a huge database structure, it can be almost next to impossible to search all the index values through all its level and then reach the destination data block to retrieve the desired data. Hashing is an effective technique to calculate the direct location of a data record on the disk without using index structure.

Hashing uses hash functions with search keys as parameters to generate the address of a data record.

### Hash Organization

- **Bucket** – A hash file stores data in bucket format. Bucket is considered a unit of storage. A bucket typically stores one complete disk block, which in turn can store one or more records.
- **Hash Function** – A hash function, **h**, is a mapping function that maps all the set of search-keys **K** to the address where actual records are placed. It is a function from search keys to bucket addresses.

### Static Hashing

In static hashing, when a search-key value is provided, the hash function always computes the same address. For example, if mod-4 hash function is used, then it shall generate only 5 values. The output address shall always be same for that function. The number of buckets provided remains unchanged at all times.

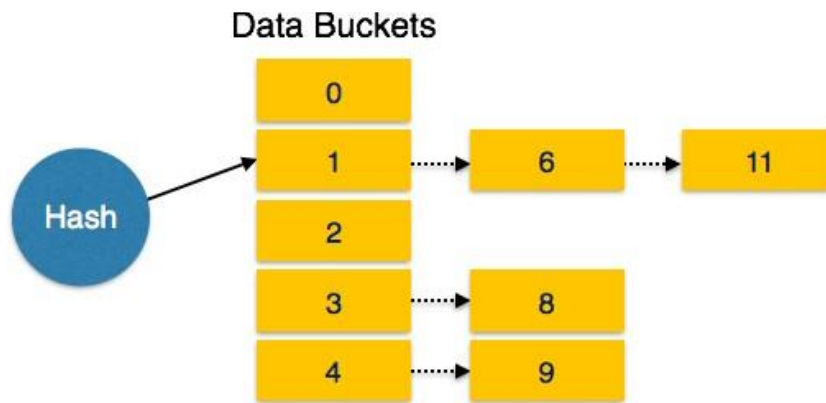
### Operation

- **Insertion** – When a record is required to be entered using static hash, the hash function **h** computes the bucket address for search key **K**, where the record will be stored.  
$$\text{Bucket address} = h(K)$$
- **Search** – When a record needs to be retrieved, the same hash function can be used to retrieve the address of the bucket where the data is stored.
- **Delete** – This is simply a search followed by a deletion operation.

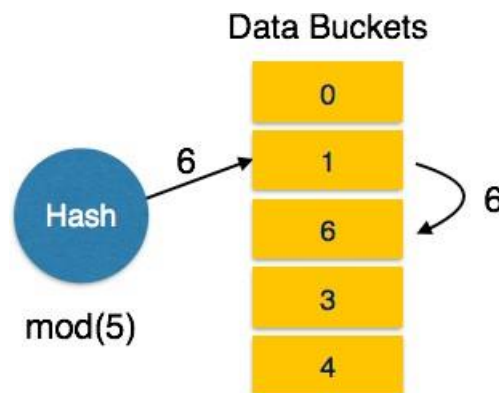
### Bucket Overflow

The condition of bucket-overflow is known as **collision**. This is a fatal state for any static hash function. In this case, overflow chaining can be used.

- **Overflow Chaining** – When buckets are full, a new bucket is allocated for the same hash result and is linked after the previous one. This mechanism is called **Closed Hashing**.



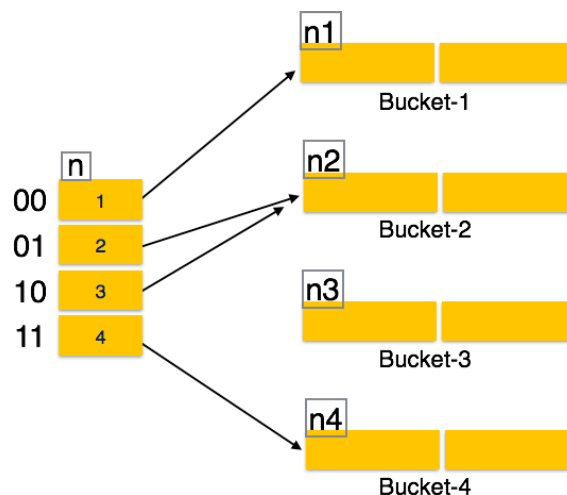
- **Linear Probing** – When a hash function generates an address at which data is already stored, the next free bucket is allocated to it. This mechanism is called **Open Hashing**.



### Dynamic Hashing

The problem with static hashing is that it does not expand or shrink dynamically as the size of the database grows or shrinks. Dynamic hashing provides a mechanism in which data buckets are added and removed dynamically and on-demand. Dynamic hashing is also known as **extended hashing**.

Hash function, in dynamic hashing, is made to produce a large number of values and only a few are used initially.



## COMPARISON OF THREE FILE ORGANIZATIONS

Refer text book

Database Management Systems, 3/e, Raghurama Krishnan, Johannes Gehrke, TMH

### **Chapter 8 File Organizations & Indexes**

**Pages from 232 to 236.**

#### **Review Questions**

1. Explain about the measures that are to be considered for comparing the performance of various file organization techniques.
2. Explain in detail B+ tree file organization.
3. Write short notes on: i) Primary index ii) Clustered index iii) Secondary index.
4. Explain various anomalies that arise due to interleaved execution of transactions with suitable examples.
5. What is static hashing? What rules are followed for index selection?
6. Define transaction and explain desirable properties of transactions.
7. What is database Recovery? Explain Shadow paging in detail.
8. Explain about Conflict Serializability and view serializability.
9. Explain the following a) Concurrent executions, b) Transaction states.

#### **References:**

- Raghurama Krishnan, Johannes Gehrke, *Database Management Systems*, 3rd Edition, Tata McGraw Hill.
- C.J. Date, *Introduction to Database Systems*, Pearson Education.
- Elmasri Navrate, *Fundamentals of Database Systems*, Pearson Education.