DATABASE MANAGEMENT SYSTEMS

UNIT-1

1.1 Introduction:

Database: The collections of related data are called as database.

Database management system: It is a collection of interrelated data and a set of programs to access those data.

FILE PROCESSING SYSTEM: It is supported by a conventional operating system. The system stores permanent records in various file. It's having some disadvantages,

1. Data redundancy and inconsistency
2. Difficulty in accessing data
3. Data isolation
4. Integrity problems
5. Atomicity problems
6. Concurrent access anomalies
7. Security problems

1.2 DATABASE MANAGEMENT SYSTEMS APPLICATIONS:

Railway Reservation System

Database is required to keep record of ticket booking, train's departure and arrival status. Also if trains get late then people get to know it through database update.

Library Management System

There are thousands of books in the library so it is very difficult to keep record of all the books in a copy or register. So DBMS used to maintain all the information relate to book issue dates, name of the book, author and availability of the book.

Banking

We make thousands of transactions through banks daily and we can do this without going to the bank. So how banking has become so easy that by sitting at home we can send or get money through banks. That is all possible just because of DBMS that manages all the bank transactions.

Universities and colleges

Examinations are done online today and universities and colleges maintain all these records through DBMS. Student's registrations details, results, courses and grades all the information is stored in database.

Social Media Sites

We all are on social media websites to share our views and connect with our friends. Daily millions of users signed up for these social media accounts like face book, twitter, interest and Google plus. But how all the information of users are stored and how we become able to connect to other people, yes this all because DBMS.

Telecommunications

Any telecommunication company cannot even think about their business without DBMS. DBMS is must for these companies to store the call details and monthly post paid bills.

Finance

Those days have gone far when information related to money was stored in registers and files. Today the time has totally changed because there are lots f thing to do with finance like storing sales, holding information and finance statement management etc.

Military

Military keeps records of millions of soldiers and it has millions of files that should be keep secured and safe. As DBMS provides a big security assurance to the military information so it is widely used in militaries. One can easily search for all the information about anyone within seconds with the help of DBMS.

1.3 DATABASE SYSTEM VS FILE SYSTEM:

S no	Database	File system
1	Data are not redundant and consistent	Data are redundant and inconsistent
2	Easy to access	Difficult to access
3	It ensures integrity, atomicity and security	It doesn't ensures it
4	Data can be concurrently accessed	Concurrent access is not possible
5	Easy to retrieve appropriate data	Difficult to retrieve appropriate data

Disadvantages of file management systems:

1. **Data Mapping and Access: -** Although all the related information's are grouped and stored in different files in same directory, there is no mapping between any two files.
2. **Data Redundancy: -** There are no methods to validate the insertion of duplicate data in file system. Decentralization of data leads to data redundancy.
3. **Data Dependence: -** It is to store the data in flat files. So there is no dependent on files.
4. **Data inconsistency: -** It is due to decentralization of data[i.e. no updating of the data]

5. **Data Isolation:** separation of data [individuality]

6. **Security: -** Each file can be password protected. But what if have to give access to only few records in the file? For example, user has to be given access to view only their bank account information in the file. This is very difficult in the file system.
7. **Integrity:** In FMS it does not check the rules.
8. **Atomicity: -** It depends on transaction which means set of actions. Transaction is done or undone is atomicity.
9. **Concurrent Access: -** In FMS, multiple users cannot access same file at same time.

NOTE: All these disadvantages in reverse become the advantages of Database management systems.

1.4 VIEW OF DATA: The database system provides an abstract view of data to users. The system hides details/ of how the data is stored and maintained. Here we are including 2 things,

1.4.1. **Data abstraction**: Data is hidden through various levels of abstraction such as,

a. Physical level

b. Logical level

c. View level

a) **Physical level**- The lowest level of abstraction describes how the data are physically stored. The physical level describes complex low level data structure.

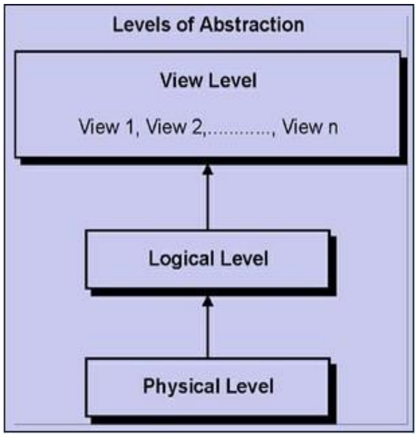


Figure-1.1

b) **Logical level**- The next level of abstraction describes about data is stored in the database, and what relationship exists among those data. It describes the entire database in terms of a small number of relatively simple structures.

c) **View level**- The highest level of abstraction describes only part of the entire database. The view level of abstraction exists to simplify their interaction with the system. The system may provide many views for the same database.

1.5 Instances and schema: The collection of information stored in the database at a particular moment is called an instance of the database. The overall design of the database is called the database schema. Types of schemas are

1. **Physical schema**- describes the database design at the physical level.
2. **Logical schema**-describes the data database design at the logical level.
3. **Sub level**- describes the data database design at the view level.

Data Independence:

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

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

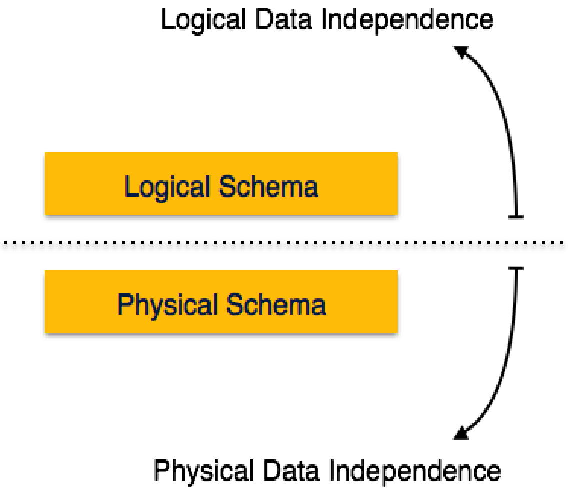


Figure-1.2

Logical Data Independence

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

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

Physical Data Independence

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

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

1.6 DATA MODELS: A data model is a collection of concepts that can be used to describe the structure of a database.

It provides the necessary means to achieve the abstraction.

By structure of a database we mean the data types, relationships and constraints that apply to data.

Design of database is called data model

Data models are classified into three types:

1. **Object based model**
    1. Object oriented model
    2. E-R model
    3. Functional model
2. **Record based model**
    1. Relational model
    2. Network model
    3. Hierarchical model
3. **Physical model**
    1. Unify model
    2. Frame model

Object oriented model:

This data model is another method of representing real world objects. It considers each object in the world as objects and isolates it from each other. It groups its related functionalities together and allows inheriting its functionality to other related sub-groups.

1.7 E-R model:

In the below diagram, Entities or real world objects are represented in a rectangular box. Their attributes are represented in ovals. Primary keys of entities are underlined. All the entities are mapped using diamonds. This is one of the methods of representing ER model. There are many different forms of representation. Basically, ER model is a graphical representation of real world objects with their attributes and relationship. It makes the system easily understandable. This model is considered as a top down approach of designing a requirement.

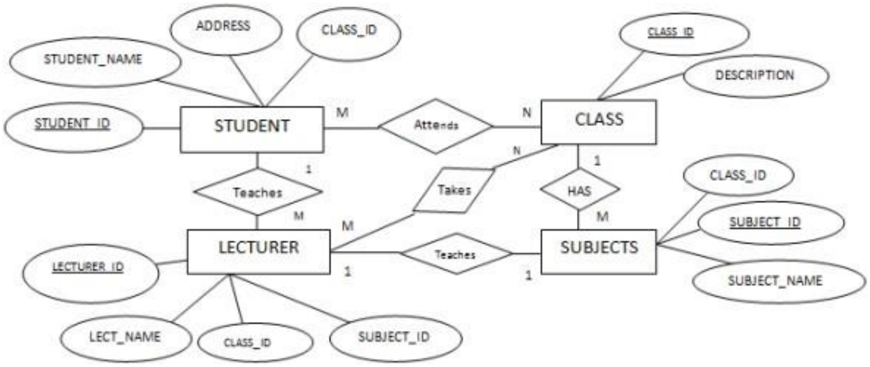


Figure 1.3

Functional model:

In functional model the data is stored and accessed in form of functions.

Hierarchical model:

In this data model, the entities are represented in a hierarchical fashion. Here we identify a parent entity, and its child entity. Again we drill down to identify next level of child entity and so on. This model can be imagined as folders inside a folder.

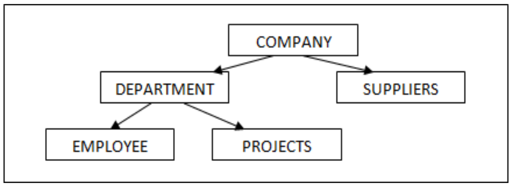


Figure 1.4

Network model:

This is the enhanced version of hierarchical data model. It is designed to address the drawbacks of the hierarchical model. It helps to address M:N relationship. This data model is also represented as hierarchical, but this model will not have single parent concept. Any child in the tree can have multiple parents here.

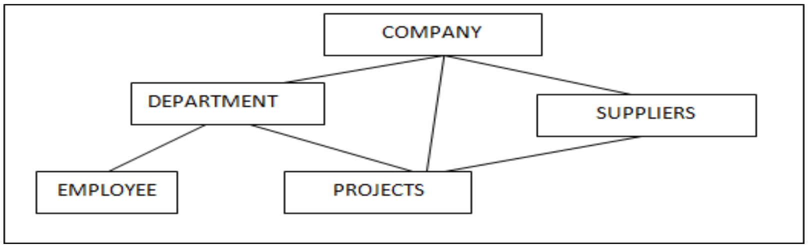
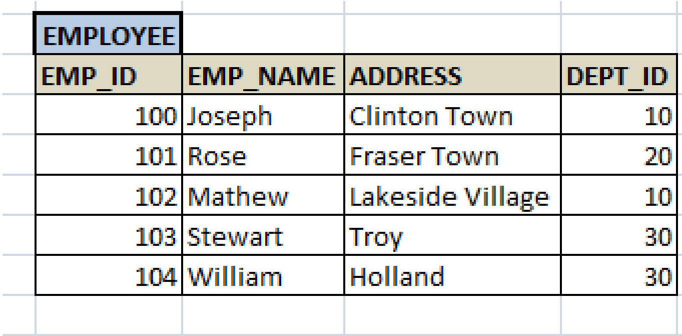


Figure-1.5

Relational model:

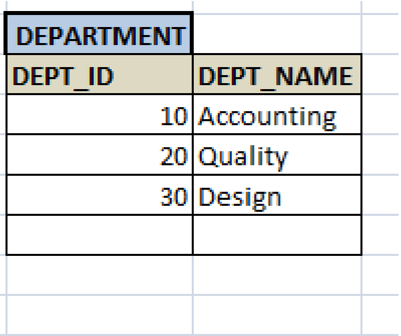
This model is designed to overcome the drawbacks of hierarchical and network models. It is designed completely different from those two models. Those models define how they are structured in the database physically and how they are inter-related. But in the relational model, we are least bothered about how they are structured. It purely based on how the records in each table are related. It purely isolates physical structure from the logical structure. Logical structure is defines records are grouped and distributed.



EMPLOYEE

EMP_ID	EMP_NAME	ADDRESS	DEPT_ID
100	Joseph	Clinton Town	10
101	Rose	Fraser Town	20
102	Mathew	Lakeside Village	10
103	Stewart	Troy	30
104	William	Holland	30

Table 1.1



DEPARTMENT

DEPT_ID	DEPT_NAME
10	Accounting
20	Quality
30	Design

Table 1.2

Physical model:

Physical data model represent the model where it describes how data are stored in computer memory, how they are scattered and ordered in the memory, and how they would be retrieved from memory. Basically physical data model represents the data at data layer or internal layer. It represents each table, their columns and specifications, constraints like primary key, foreign key etc. It basically represents how each tables are built and related to each other in DB.

In short we can say a physical data model has

- Tables and its specifications – table names and their columns. Columns are represented along with their data types and size. In addition primary key of each table is shown at the top of the column list.
- Foreign keys are used to represent the relationship between the tables. Mapping between the tables are represented using arrows between them.
- Physical data model can have denormalized structure based on the user requirement. The tables might not be in normalized forms.

1.8 DATA BASE ACCESS FOR APPLICATION PROGRAMS:

- ➢ Data base will not interact directly it will use the software. It is called the embedded software
- ➢ The software used is sql.
- ➢ This gives the commands to the front end people(navy users)
- ➢ The people who interact directly with data base they are known as **data base administrators** .They have all rights on data base
- ➢ The sql statements are combined with the host users ie: front end is called **" embedded sql"**

Data base users:

1. Application programmers: Through application statements interacts with DML statements.

- ❖ Embedded to host language

2. Sophisticated users: Through direct SQLstatements.

3. Specialize users: Similar to sophisticated users but they write special data base programs like by PL/SQL.

4. Naive users or unsophisticated: Interact with standalone permanent application programs

5. Data base administrators: create, resolve the data .Grant permission to users. Storage can be checked. Backup and recovery

DBA can check both technical and non-technical details of the data base.

TRANSACTION MANAGEMENT:

If the transaction is not done properly data will lose the consistency

- Eg: Booking the ticket

- The sequence of operation in an atomic in nature is called a transaction.

- Once the transaction is completed the reflection on data should be done immediately if it is not done consistency will lose.
- Each transaction should follow the **ACID** properties

  A-Atomicity

  C-Consistency

  I-Isolation

  D-Durability
- **Atomicity** − this property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none. There must be no state in a database where a transaction is left partially completed. States should be defined either before the execution of the transaction or after the execution/abortion/failure of the transaction.
- **Consistency** − the database must remain in a consistent state after any transaction. No transaction should have any adverse effect on the data residing in the database. If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.
- **Durability** − the database should be durable enough to hold all its latest updates even if the system fails or restarts. If a transaction updates a chunk of data in a database and commits, then the database will hold the modified data. If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system brings back into action.
- **Isolation** − In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.

Every transaction have 5 states:

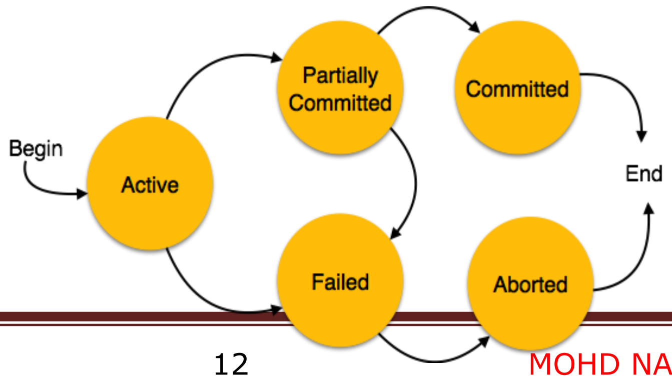


Figure-1.6

- **Active** − in this state, the transaction is being executed. This is the initial state of every transaction.
- **Partially Committed** − When a transaction executes its final operation, it is said to be in a partially committed state.
- **Failed** − A transaction is said to be in a failed state if any of the checks made by the database recovery system fails. A failed transaction can no longer proceed further.
- **Aborted** − If any of the checks fails and the transaction has reached a failed state, then the recovery manager rolls back all its write operations on the database to bring the database back to its original state where it was prior to the execution of the transaction. Transactions in this state are called aborted. The database recovery module can select one of the two operations after a transaction aborts −
  - Re-start the transaction
  - Kill the transaction
- **Committed** − If a transaction executes all its operations successfully, it is said to be committed. All its effects are now permanently established on the database system.

1.9 DATA BASE SYSTEM ARCHITECTURE:

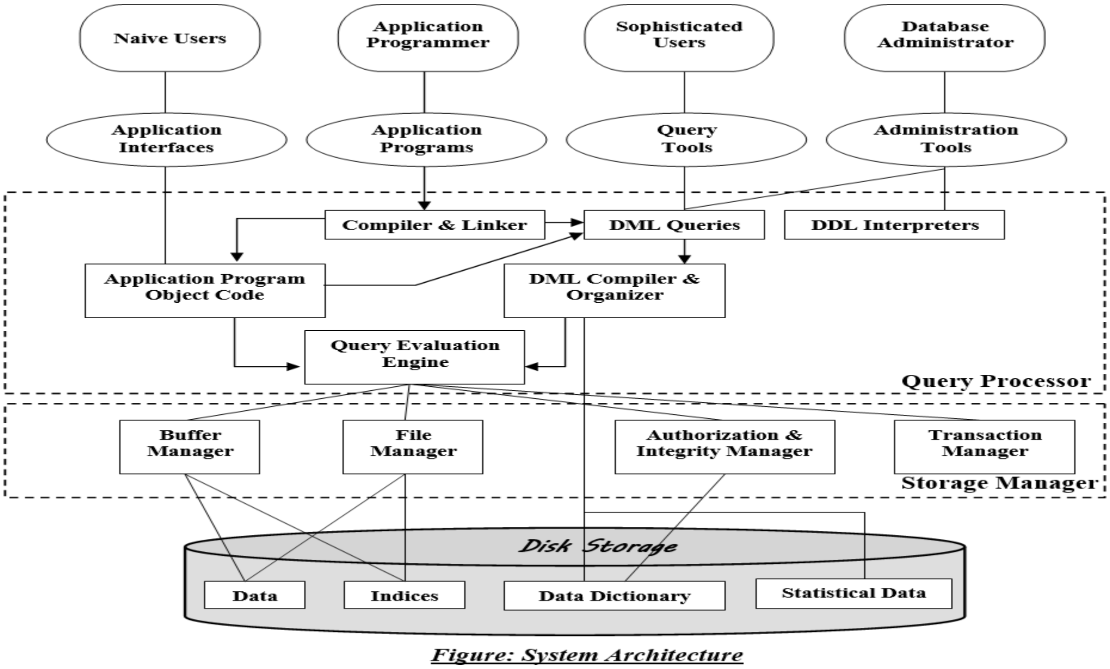


Figure-1.7

DBMS architecture is mainly divided into 2 modules

1. Query processor
2. Storage manager

1.9.1. QUERY PROCESSOR:

The query processor components in dbms accepts sql commands generated from variety of users interfaces ,produce query evaluation plan executes these plans with the data base and return results.

The components are:

DML compiler:

It is which translates DML statements in query language, low level instruction that the query evolution engine understands.

Embedded DML pre-compiler:

Sql commands can be embedded in host language application programs (c, c++, JAVA) which can be converted into DML statements by this embedded DML precompiler. The precompiler interact with the DML compiler to generate appropriate code.

DDL interpreter:

Which interprets the DDL statements and records them in a set of tables containing metadata

Application program object code:

It is which is low level instructions of a program written by no voice users/naive users which the query evaluation engine understands and execute them

Query evaluation engine:

It is which executes low level instructions generated by DML compiler.

1.9.2. STORAGE MANAGER:

The storage manager components in dbms provide the interface between the physical level data stored in data base and the program on queries requesting the stored data from data base.

The storage manager components are:

Authorization and integrity manager:

It is which test for the satisfaction of integrity constraints and checks the authority of user to access the data.

Recovery manager:

It is which is responsible for maintaining a log and restoring the system to a consistent state, after system failure.

Transaction manager:

It is which ensure that the data base remains in consistent state despite of the system failure and the concurrent transaction execution proceed without conflicting the transaction.

Lock manager:

A lock is a mechanism used to control access to database objects.

Two kinds of locks are commonly supported by a DBMS: – Shared locks on an object can be held by two different transactions at the same time.

Exclusive lock on an object ensures that no other transaction holds any lock on this object.

Every transaction begins by obtaining a shared lock on each data object that it needs to read and an exclusive lock on each data that it needs to modify, and then release all its locks after completing all actions.

It is which keeps track of requests for locks and grant locks on data in the data base.

Buffer manager:

It is which is responsible for fetching the data from disk storage into main memory and deciding what data to cache memory.

Disk space manager:

It is which manages allocation of space on disk storage and the data structures used to represent the stored information on disk.

Disk storage components are:

Data files:

Which stores the data base itself i.e. The actual data in the data base.

Data dictionary:

It is which stores the Meta data about the structure of data base .The data dictionary is used very heavily. Therefore great emphasis should be placed on developing a good design and efficient implementation of the data dictionary.

Indices:

It is which provides the fast access to the data items that hold the particular value.

Statistical data:

It stores the statistical data about the data in the database .This information is used by the query processor to select efficient ways and to execute user queries.

DATABASE DESIGN WITH E-R MODEL

1.10 E-R model :

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

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

Rectangle: Represents Entity sets.

Ellipses: Attributes

Diamonds: Relationship Set

Lines: They link attributes to Entity Sets and Entity sets to Relationship set.

Double Ellipses: Multivalued Attributes

Dashed Ellipses: Derived Attributes

Double Rectangles: Weak Entity Sets

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

A sample E-R Diagram:

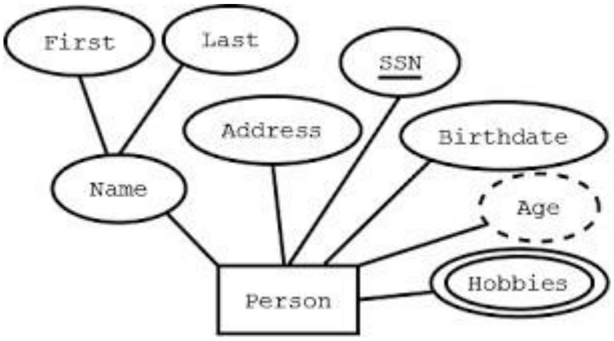


Figure-1.8

What is Database Design?

Database Design is a collection of processes that facilitate the designing, development, implementation and maintenance of enterprise data management systems

It helps produce database systems

1. That meet the requirements of the users
2. Have high performance.

The main objectives of database designing are to produce logical and physical designs models of the proposed database system.

The logical model concentrates on the data requirements and the data to be stored independent of physical considerations. It does not concern itself with how the data will be stored or where it will be stored physically.

The physical data design model involves translating the logical design of the database onto physical media using hardware resources and software systems such as database management systems (DBMS).

1.11 Attributes:

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

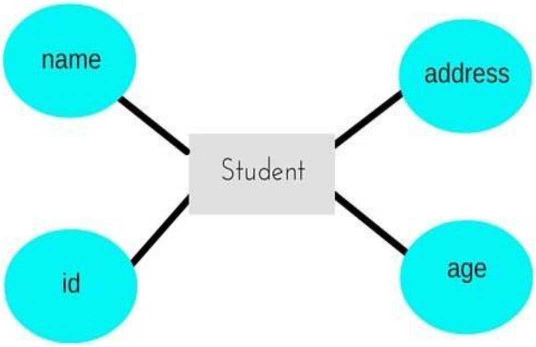


Figure-1.9

Key Attribute:

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

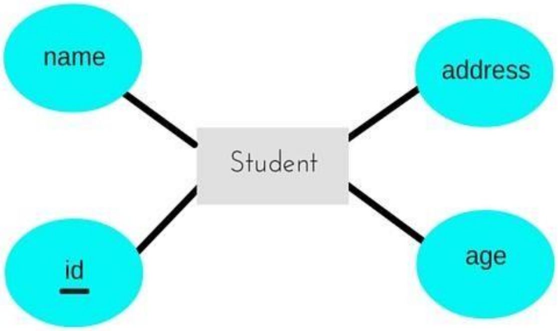


Figure-1.10

Composite Attribute:

An attribute can also have their own attributes. These attributes are known as **Composite** attribute.

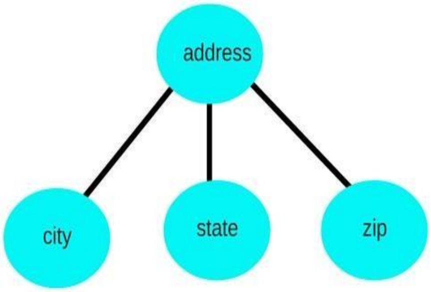


Figure-1.11

Derived Attribute:

The last category that attributes can be defined is called a derived attribute, where one attribute is calculated from another attribute.

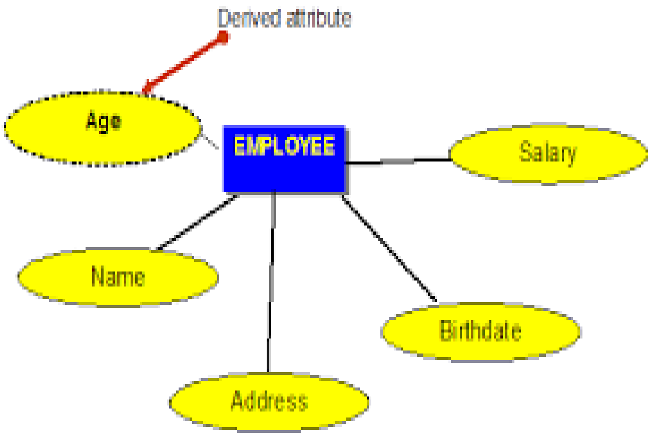


Figure-1.12

Multivalued attribute:

multi-valued attributes usually can store multiple data in them.

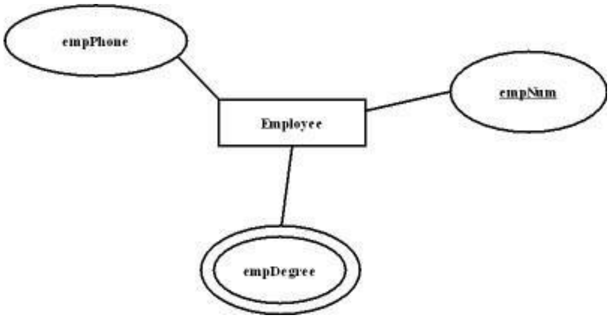


Figure-1.13

Descriptive Attribute:

. **Attributes** of the relationship is called **descriptive attribute**.

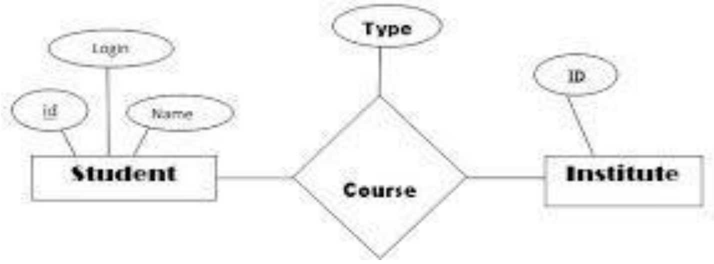


Figure-1.14

1.12 Entities: An **entity set** is a set of entities of the same type (e.g., all persons having an account at a bank).

The entity set which does not have sufficient attributes to form a primary key is called as Weak entity set.

An entity set that has a primary key is called as Strong entity set.

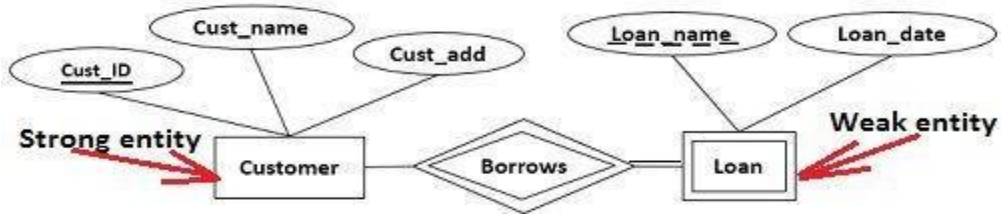


Figure-1.14

Relationship

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.

Binary Relationship and Cardinality:

A relationship where two entities are participating is called a **binary relationship**. Cardinality is the number of instance of an entity from a relation that can be associated with the relation.

- **One-to-one** – When only one instance of an entity is associated with the relationship, it is marked as '1:1'. The following image reflects that only one instance of each entity should be associated with the relationship. It depicts one-to-one relationship.

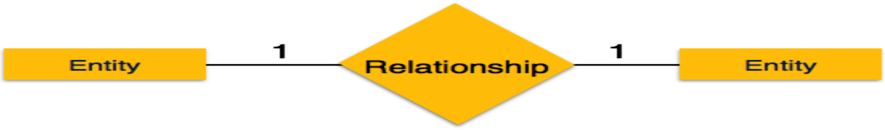


Figure-1.15

- **One-to-many** – When more than one instance of an entity is associated with a relationship, it is marked as '1:N'. The following image reflects that only one instance of entity on the left and more than one instance of an entity on the right can be associated with the relationship. It depicts one-to-many relationship.

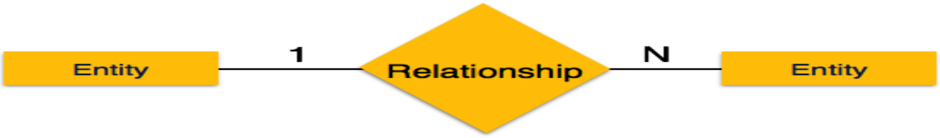


Figure-1.15

- **Many-to-one** – When more than one instance of entity is associated with the relationship, it is marked as 'N:1'. The following image reflects that more than one instance of an entity on the left and only one instance of an entity on the right can be associated with the relationship. It depicts many-to-one relationship.

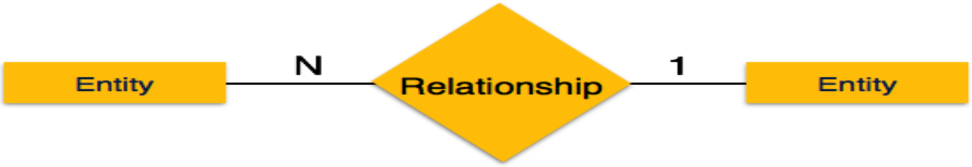


Figure-1.16

- **Many-to-many** – The following image reflects that 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. It depicts many-to-many relationship.

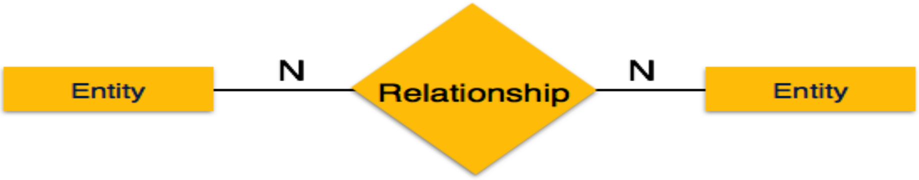


Figure-1.17

Participation Constraints

- **Total Participation** – Each entity is involved in the relationship. Total participation is represented by double lines.
- **Partial participation** – Not all entities are involved in the relationship. Partial participation is represented by single lines.

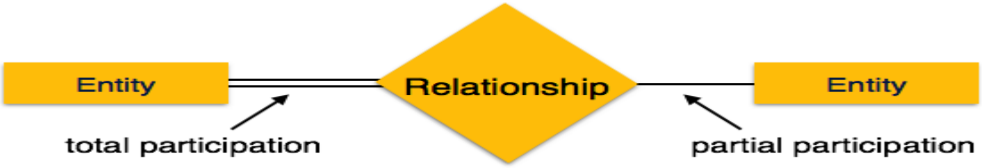
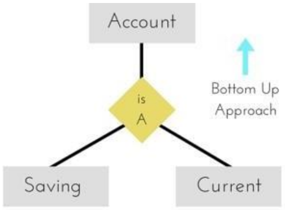


Figure-1.18

1.13 Additional features of ER Model:

Generalization:

Generalization is a bottom-up approach in which two lower level entities combine to form a higher level entity. In generalization, the higher level entity can also combine with other lower level entity to make further



higher level entity.

Figure-1.19

Specialization:

Specialization is opposite to Generalization. It is a top-down approach in which one higher level entity can be broken down into two lower level entity. In specialization, some higher level entities may not have lower-level entity sets at all.

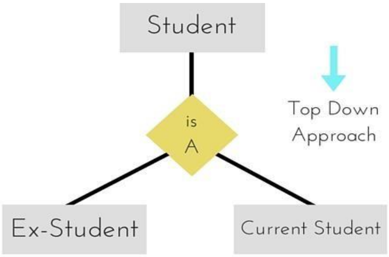


Figure-1.20

Aggregation:

Aggregation is a process when relation between two entity is treated as a single entity. Here the relation between Center and Course, is acting as an Entity in relation with Visitor.

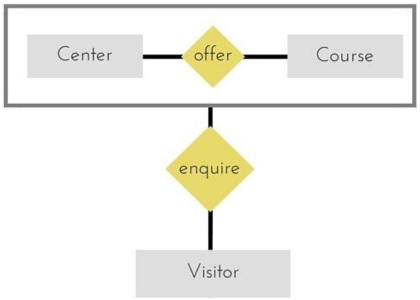


Figure-1.21

1.14 Conceptual Design:

• Conceptual design follows requirements analysis, Yields a high-level description of data to be stored

• ER model popular for conceptual design. Constructs are expressive, close to the way people think about their applications.

• Basic constructs: *entities*, *relationships*, and *attributes* (of entities and relationships)

• Constraints

• Some additional constructs: *weak entities*, *ISA hierarchies*

• Note: There are many variations on ER model

TEXT BOOK:

1. Raghurama Krishnan, Johannes Gehrke, Database Management Systems, 3rd Edition, TATA McGraw hill.
2. Web pages

UNIT - II

2.1 Introduction to Relational Model:

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

We use the example of student information:

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

This says, for instance, that the field named *sid* has a domain named string. The set of values associated with domain string is the set of all character strings.We now turn to the instances of a relation. An **instance** of a relation is a set of **tuples**, also called **records**, in which each tuple has the same number of fields as the relation schema. A relation instance can be thought of as a *table* in which each tuple is a *row*, and all rows have the same number of fields. (The term *relation instance* is often abbreviated to just *relation*, when there is no confusion with other aspects of a relation such as its schema.)

Fields (Attributes, columns) Field names

sid	name	login	age	gpa
50000	Jones	Jones@cs	19	1.8
53666	Smith	Smith@cs	21	2.0
53668	Roy	Roy@cs	18	3.2
54532	Mealey	Mealey@cs	20	3.8
54566	Rony	Rony@cs	14	3.4
53452	Guldu	Guldu@cs	12	3.3

An instance of the Students relation Tuples (Records, rows)

Table 2.1

The instance *S*1 contains six tuples and has, as we expect from the schema, five fields. Note that no two rows are identical. This is a requirement of the relational model each relation is defined to be a *set* of unique tuples or rows.1. The order in which the rows are listed is not important. Thus *sid* is field 1 of Students, *login* is field 3,and so on. If this convention is used, the order of fields is significant.

Most database systems use a combination of these conventions. For example, in SQL the named fields convention is used in statements that retrieve tuples, and the ordered fields convention is commonly used when inserting tuples.

A relation schema specifies the domain of each field or column in the relation instance.

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

The **degree**, also called **arity**, of a relation is the number of fields. The **cardinality** of a relation instance is the number of tuples in it. In Figure the degree of the relation (the number of columns) is five, and the cardinality of this instance is six. A **relational database** is a collection of relations with distinct relation names. The **relational database schema** is the collection of schemas for the relations in the database.

Creating and Modifying relations using SQL:

The SQL that supports the creation, deletion, and modification of tables is called the **Data Definition Language (DDL)**. Now, we will just consider domains that are built-in types, such as integer.

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

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

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

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

We can optionally omit the list of column names in the INTO clause and list the values in the appropriate order, but it is good style to be explicit about column names. We can delete tuples using the DELETE command. We can delete all Students tuples with *name* equal to Smith using the command:

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

We can modify the column values in an existing row using the UPDATE command. For example, we can increment the age and decrement the gpa of the student with *sid* 53668:

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

These examples illustrate some important points. The WHERE clause is applied first and determines which rows are to be modified. The SET clause then determines how these rows are to be modified. If the column that is being modified is also used to determine the new value, the value used in the expression on the right side of equals (=) is the *old* value, that is, before the modification. To illustrate these points further,

consider the following variation of the previous query:

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

If this query is applied on the instance *S*1 of Students shown in Figure, we obtain the instance shown in following figure

sno	name	login	age	gpa
50000	Jones	Jones@cs	19	3.2
53666	Smith	Smith@cs	21	3.3
53668	Roy	Roy@cs	13	3.2
54532	Mealey	Mealey@cs	15	3.7
54566	Rony	Rony@cs	11	1.8
53452	Guldu	Guldu@cs	12	2.0

Students Instance after update

Table 2.2

2.2 INTEGRITY CONSTRAINTS OVER RELATIONS:

A database is only as good as the information stored in it, and a DBMS must therefore help prevent the entry of incorrect information. An **integrity constraint (IC)** is a condition that is specified on a database schema, and restricts the data that can be stored in an instance of the database. If a database instance satisfies all the integrity constraints specified on the database schema, it is a **legal** instance. A DBMS **enforces** integrity constraints, in that it permits only legal instances to be stored in the database.

Integrity constraints are specified and enforced at different times:

1. When the DBA or end user defines a database schema, he or she specifies the ICs that must hold on any instance of this database.

2. When a database application is run, the DBMS checks for violations and disallows changes to the data that violate the specified ICs. (In some situations, rather than disallow the change, the DBMS might instead make some compensating changes to the data to ensure that the database instance satisfies all ICs. In any case, changes to the database are not allowed to create an instance that violates any IC.)

Many kinds of integrity constraints can be specified in the relational model.

TYPES OF INTEGRITY CONSTRAINTS

Various types of integrity constraints are-

1. Domain Integrity
2. Entity Integrity Constraint
3. Referential Integrity Constraint
4. Key Constraints

2.2.1. Domain Integrity- Domain integrity means the definition of a valid set of values for an attribute. You define data type, length or size, is null value allowed, is the value unique or not for an attribute, the default value, the range (values in between) and/or specific values for the attribute.

2.2.2. Entity Integrity Constraint- This rule states that in any database relation value of attribute of a primary key can't be null.

EXAMPLE- Consider a relation "STUDENT" Where "Stu_id" is a primary key and it must not contain any null value whereas other attributes may contain null value e.g "Branch" in the following relation contains one null value.

Stu_id	Name	Branch
113421	Ajay	IT
113564	Sandy	CSE
112453	Sujith	
114532	Rose	CIVIL
113422	Jack	IT

Table 2.3

2.2.3. Referential Integrity Constraint-It states that if a foreign key exists in a relation then either the foreign key value must match a primary key value of some tuple in its home relation or the foreign key value must be null.

The rules are:

1. You can't delete a record from a primary table if matching records exist in a related table.
2. You can't change a primary key value in the primary table if that record has related records.
3. You can't enter a value in the foreign key field of the related table that doesn't exist in the primary key of the primary table.
4. However, you can enter a Null value in the foreign key, specifying that the records are unrelated.

EXAMPLE-Consider 2 relations "stu" and "stu_1" Where "Stu_id " is the primary key in the "stu" relation and foreign key in the "stu_1" relation.

Relation "stu"

Stu_id	Name	Branch
113421	Ajay	IT
113564	Sandy	CSE
112453	Sujith	ECE
114532	Rose	CIVIL
113422	Jack	IT

Table 2.4

Relation "stu_1"

Stu_id	Course	Duration
113421	B.Tech	4 years
113564	B.Tech	4 years
112453	B.Tech	4 years
114532	B.Tech	4 years
113422	B.Tech	4 years

Table 2.5

Examples:

Rule 1. You can't delete any of the rows in the ”stu” relation that are visible since all the ”stu” are in use in the “stu_1” relation.

Rule 2. You can't change any of the ”Stu_id” in the “stu” relation since all the “Stu_id” are in use in the ”stu_1” relation.

Rule 3. The values that you can enter in the” Stu_id” field in the “stu_1” relation must be in the” Stu_id” field in the “stu” relation.

Rule 4 You can enter a null value in the "stu_1" relation if the records are unrelated.

2.2.4. Key Constraints- A Key Constraint is a statement that a certain minimal subset of the fields of a relation is a unique identifier for a tuple.

SQL Constraints are rules used to limit the type of data that can go into a table, to maintain the accuracy and integrity of the data inside table.

Constraints can be divided into following two types,

- **Column level constraints:** limits only column data. The style is called Inline specification.
- **Table level constraints:** limits whole table data. The style is called out of line specification

Constraints are used to make sure that the integrity of data is maintained in the database. Following are the most used constraints.

- NOT NULL
- UNIQUE
- PRIMARY KEY
- FOREIGN KEY
- UNIQUE

2.2.4.1 NOT NULL Constraint:

NOT NULL constraint restricts a column from having a NULL value. Once **NOT NULL** constraint is applied to a column, you cannot pass a null value to that column. It enforces a column to contain a proper value. One important point to note about NOT NULL constraint is that it cannot be defined at table level.

Example using NOT NULL constraint:

```
CREATE table Student (s_id int NOT NULL, Name varchar(60), Age int);
```

The above query will declare that the **s_id** field of **Student** table will not take NULL value.

2.2.4.2UNIQUE Constraint:

UNIQUE constraint ensures that a field or column will only have unique values. A UNIQUE constraint field will not have duplicate data. UNIQUE constraint can be applied at column level or table level.

Example using UNIQUE constraint when creating a Table (Table Level)

```
CREATE table Student(s_id int NOT NULL UNIQUE, Name varchar(60), Age int);
```

The above query will declare that the **s_id** field of **Student** table will only have unique values and won't take NULL value.

Example using UNIQUE constraint after Table is created (Column Level)

```
ALTER table Student add UNIQUE(s_id);
```

The above query specifies that **s_id** field of **Student** table will only have unique value.

2.2.4.3 Primary Key Constraint:

Primary key constraint uniquely identifies each record in a database. A Primary Key must contain unique value and it must not contain null value. Usually Primary Key is used to index the data inside the table.

Example using PRIMARY KEY constraint at Table Level:

```
CREATE table Student (s_id int PRIMARY KEY, Name varchar(60) NOT NULL, Age int);
```

The above command will creates a PRIMARY KEY on the s_id.

Example using PRIMARY KEY constraint at Column Level:

```
ALTER table Student add PRIMARY KEY (s_id);
```

The above command will creates a PRIMARY KEY on the s_id.

2.2.4.4 Foreign Key Constraint:

FOREIGN KEY is used to relate two tables. FOREIGN KEY constraint is also used to restrict actions that would destroy links between tables. To understand FOREIGN KEY, let's see it using two table.

Customer_detail table:

C_id	Customer_name	Address
101	Ajay	Delhi
102	Adam	Kolkata
103	Anand	Noidak

Table 2.6

Order_detail table:

O_id	Order_name	C_id
10	Order1	101
11	Order2	102
12	Order3	103

Table 2.7

In **Customer_Detail** table, c_id is the primary key which is set as foreign key in

Order_Detail table. The value that is entered in c_id which is set as foreign key in

Order_Detail table must be present in **Customer_Detail** table where it is set as primary key. This prevents invalid data to be inserted into c_id column of **Order_Detail** table.

Example using FOREIGN KEY constraint at Table Level:

```
CREATE table Order_Detail(order_id int PRIMARY KEY,order_name varchar(60) NOT NULL,
c_id int FOREIGN KEY REFERENCES Customer_Detail(c_id));
```

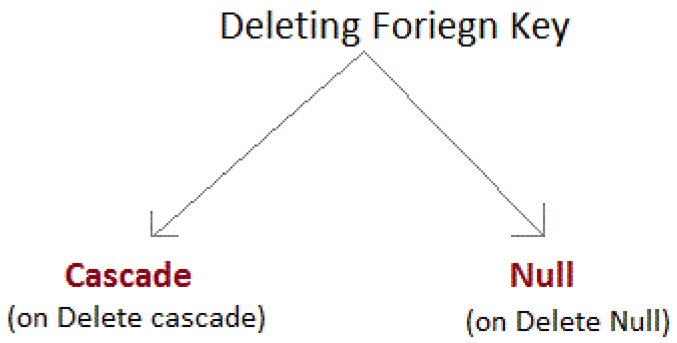
In this query, c_id in table Order_Detail is made as foriegn key, which is a reference of c_id column of Customer_Detail.

Example using FOREIGN KEY constraint at Column Level:

```
ALTER table Order_Detail add FOREIGN KEY (c_id) REFERENCES Customer_Detail(c_id);
```

Behaviour of Foriegn Key Column on Delete:

There are two ways to maintin the integrity of data in Child table, when a particular record is deleted in main table. When two tables are connected with Foriegn key, and certain data in the main table is deleted, for which record exit in child table too, then we must have some mechanism to save the integrity of data in child table.



- **On Delete Cascade :** This will remove the record from child table, if that value of foriegn key is deleted from the main table.
- **On Delete Null :** This will set all the values in that record of child table as NULL, for which the value of foriegn key is deleted from the main table.
- If we don't use any of the above, then we cannot delete data from the main table for which data in child table exists. We will get an error if we try to do so.

ERROR : Record in child table exist

2.2.4.5 CHECK Constraint:

CHECK constraint is used to restrict the value of a column between a range. It performs check on the values, before storing them into the database. Its like condition checking before saving data into a column.

Example using CHECK constraint at Table Level:

```
create table Student(s_id int NOT NULL CHECK(s_id > 0),
Name varchar(60) NOT NULL, Age int);
```

The above query will restrict the s_id value to be greater than zero.

Example using CHECK constraint at Column Level:

```
ALTER table Student add CHECK(s_id > 0);
```

2.3 QUERYING RELATIONAL DATA

A **relational database query** is a question about the data, and the answer consists of a new relation containing the result. For example, we might want to find all students younger than 18 or all students enrolled in Reggae203. A **query language** is a specialized language for writing queries.

SQL is the most popular commercial query language for a relational DBMS. We now present some SQL examples that illustrate how easily relations can be queried. Consider the instance of the Students relation shown figure . We can retrieve rows corresponding to students who are younger than 18 with the following SQL query:

```
SELECT *FROM Students S
WHERE S.age < 18
```

The symbol * means that we retain all fields of selected tuples in the result. To understand this query, think of S as a variable that takes on the value of each tuple in Students, one tuple after the other. The condition *S.age < 18* in the WHERE clause specifies that we want to select only tuples in which the *age* field has a value less than 18. This query evaluates to the relation shown in below figure.

sid	name	login	age	gpa
54566	Rony	Rony@cs	14	3.4
53452	Guldu	Guldu@cs	12	3.3

Table 2.8

Students with *age* < 18 on Instance S1

This example illustrates that the domain of a field restricts the operations that are permitted on field values, in addition to restricting the values that can appear in the field. The condition *S.age < 18* involves an arithmetic comparison of an *age* value with an integer and is permissible because the domain of *age* is the set of integers. In addition to selecting a subset of tuples, a query can extract a subset of the fields of each selected tuple. We can compute the names and logins of students who are younger than 18 with the following query:

```
SELECT S.name, S.login FROM Students S WHERE S.age < 18
```

The solution to this query is the following figure:

name	Login
Rony	Rony@cs
Guldu	Guldu@cs

2.4 LOGICAL DATABASE DESIGN: ER TO RELATIONAL

The ER model is convenient for representing an initial, high-level database design. Given an ER diagram describing a database, there is a standard approach to generating a relational database schema that closely approximates the ER design. It is discussed in 2nd unit.

SQL Alias:

Alias is used to give an alias name to a table or a column. This is quite useful in case of large or complex queries. Alias is mainly used for giving a short alias name for a column or a table with complex names.

Syntax of Alias for table names,

SELECT column-name from *table-name*as **alias-name**

Following is an Example using Alias,

SELECT * from Employee_detail as **ed**;

Alias syntax for columns will be like,

SELECT *column-name* as **alias-name** from*table-name*

Example using alias for columns,

SELECT customer_id as **cid** from Emp;

2.5 Introduction to views:

A view is nothing more than a SQL statement that is stored in the database with a associated name. A view is actually a composition of a table in the form of a predefined SQL query.

A view can contain all rows of a table or select rows from a table. A view can be created from one or many tables which depends on the written SQL query to create a view.

Views, which are a type of virtual tables allow users to do the following −

- Structure data in a way that users or classes of users find natural or intuitive.
- Restrict access to the data in such a way that a user can see and (sometimes) modify exactly what they need and no more.
- Summarize data from various tables which can be used to generate reports.
  - ➢ View is a virtual table or logical table or it acts like a mirror image.
  - ➢ View in practically contains no data by itself.
  - ➢ The table upon which a view is based are called as base tables.
  - ➢ Views can be created as object views or relational views.
  - ➢ View is simply a select statement.

Creating Views:

Database views are created using the **CREATE VIEW** statement. Views can be created from a single table, multiple tables or another view.

To create a view, a user must have the appropriate system privilege according to the specific implementation.

The basic **CREATE VIEW** syntax is as follows −

```
CREATE VIEW view_name AS SELECT column1, column2.....
          FROM table_name WHERE [condition];
```

You can include multiple tables in your SELECT statement in a similar way as you use them in a normal SQL SELECT query.

Example: Consider the CUSTOMERS table having the following records −

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

Following is an example to create a view from the CUSTOMERS table. This view would be used to have customer name and age from the CUSTOMERS table.

```
SQL > CREATE VIEW CUSTOMERS_VIEW AS SELECT name, age FROM  CUSTOMERS;
```

Now, you can query CUSTOMERS_VIEW in a similar way as you query an actual table. Following is an example for the same.

```
SQL > SELECT * FROM CUSTOMERS_VIEW;
```

This would produce the following result.

```
+----------+-----+
| name     | age |
+----------+-----+
| Ramesh   |  32 |
| Khilan   |  25 |
| kaushik  |  23 |
| Chaitali |  25 |
| Hardik   |  27 |
| Komal    |  22 |
| Muffy    |  24 |
+----------+-----+
```

WITH CHECK OPTION

The WITH CHECK OPTION is a CREATE VIEW statement option. The purpose of the WITH CHECK OPTION is to ensure that all UPDATE and INSERTs satisfy the condition(s) in the view definition.

If they do not satisfy the condition(s), the UPDATE or INSERT returns an error.

The following code block has an example of creating same view CUSTOMERS_VIEW with the WITH CHECK OPTION.

```
CREATE VIEW CUSTOMERS_VIEW AS
SELECT name, age
FROM  CUSTOMERS
WHERE age IS NOT NULL
WITH CHECK OPTION;
```

The WITH CHECK OPTION in this case should deny the entry of any NULL values in the view's AGE column, because the view is defined by data that does not have a NULL value in the AGE column.

Force View Creation

Force keyword is used while creating a view. This keyword force to create View even if the table does not exist. After creating a force View if we create the base table and enter values in it, the view will be automatically updated.

Syntax for forced View is,

```
CREATE or REPLACE force view view_name AS
SELECT column_name(s)
FROM table_name
WHERE condition
```

Read-Only View

We can create a view with read-only option to restrict access to the view.

Syntax to create a view with Read-Only Access

```
CREATE or REPLACE force view view_name AS
SELECT column_name(s)
FROM table_name
WHERE condition with read-only
```

The above syntax will create view for read-only purpose, we cannot Update or Insert data into read-only view. It will throw an error.

Updating a View

A view can be updated under certain conditions which are given below −

- 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 that view. The following code block has an example to update the age of Ramesh.

```
SQL > UPDATE CUSTOMERS_VIEW    SET AGE = 35   WHERE name = 'Ramesh';
```

This would ultimately update the base table CUSTOMERS and the same would reflect in the view itself. Now, try to query the base table and the SELECT statement would produce the following result.

```
+......+...............+.......+.................+...............+
| ID | NAME      | AGE | ADDRESS    | SALARY    |
+......+...............+.......+.................+...............+
|  1 | Ramesh    |  35 | Ahmedabad |  2000.00 |
|  2 | Khilan    | 25 | Delhi       |  1500.00 |
|  3 | kaushik  |  23 | Kota        |  2000.00 |
|  4 | Chaitali |  25 | Mumbai     |  6500.00 |
|  5 | Hardik    |  27 | Bhopal     |  8500.00 |
|  6 | Komal     |  22 | MP          |  4500.00 |
|  7 | Muffy     |  24 | Indore     | 10000.00 |
+......+...............+.......+.................+...............+
```

Inserting Rows into a View

Rows of data can be inserted into a view. The same rules that apply to the UPDATE command also apply to the INSERT command.

Here, we cannot insert rows in the CUSTOMERS_VIEW because we have not included all the NOT NULL columns in this view, otherwise you can insert rows in a view in a similar way as you insert them in a table.

Deleting Rows from a View

Rows of data can be deleted from a view. The same rules that apply to the UPDATE and INSERT commands apply to the DELETE command.

Following is an example to delete a record having AGE = 22.

```
SQL > DELETE FROM CUSTOMERS_VIEW   WHERE age = 22;
```

This would ultimately delete a row from the base table CUSTOMERS and the same would reflect in the view itself. Now, try to query the base table and the SELECT statement would produce the following result.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+------------+-----+---------------+------------+
|  1 | Ramesh   |  35 | Ahmedabad |  2000.00 |
|  2 | Khilan    |  25   | Delhi          |  1500.00 |
|  3 | kaushik   |  23 | Kota              |  2000.00 |
|  4 | Chaitali   |  25 | Mumbai         |  6500.00 |
|  5 | Hardik      |  27 | Bhopal          |  8500.00 |
|  7 | Muffy       |  24 | Indore            | 10000.00 |
+.......+................+.......+..................+................+
```

Dropping Views

Obviously, where you have a view, you need a way to drop the view if it is no longer needed. The syntax is very simple and is given below −

```
DROP VIEW view_name;
```

Following is an example to drop the CUSTOMERS_VIEW from the CUSTOMERS table.

```
DROP VIEW CUSTOMERS_VIEW;
```

Types of View

There are two types of view,

- Simple View
- Complex View

Simple View	**Complex View**
Created from one table	Created from one or more table
Does not contain functions	Contain functions
Does not contain groups of data	Contains groups of data

2.6 DESTROYING/ALTERING TABLES AND VIEWS:

If we decide that we no longer need a base table and want to destroy it, we can use the DROP TABLE command. For example, DROP TABLE Students RESTRICT destroys the Students table unless some view or integrity constraint refers to Students; if so, the command fails. If the keyword RESTRICT is replaced by CASCADE, Students is dropped and any referencing views or integrity constraints are (recursively) dropped as well; one of these two keywords must always be specified. A view can be dropped using the DROP VIEW command, which is just like DROP TABLE.ALTER TABLE modifies the structure of an existing table. To add a column called *maiden-name* to Students, for example, we would use the following command:

```
ALTER TABLE Students ADD COLUMN maiden-name char(10)
```

The definition of Students is modified to add this column, and all existing rows are padded with *null* values in this column. ALTER TABLE can also be used to delete columns and to add or drop integrity constraints on a table; we will not discuss these aspects of the command beyond remarking that dropping columns is treated very similarly to dropping tables or views.

Relational database systems are expected to be equipped with a query language that can assist its users to query the database instances. There are two kinds of query languages − relational algebra and relational calculus proposed by E. F. Codd in 1976.

2.7 RELATIONAL ALGEBRA:

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

Terminology

- Relation – a set of tuples.
- Tuple – a collection of attributes which describe some real world entity.
- Attribute – a real world role played by a named domain.
- Domain – a set of atomic values.
- Set – a mathematical definition for a collection of objects which contains no duplicates.

Unary operations: Operates on one relation. These include:

- ***Selection***
- ***Projection***
- ***Renaming***

Selection Operation (σ):

It selects tuples that satisfy the given predicate from a relation.

Notation − $\sigma_p(r)$

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

For example −

$\sigma_{subject = "database"}(\text{Books})$

Output − Selects tuples from books where subject is 'database'.

$\sigma_{\text{subject = "database" and price = "450"}}(\text{Books})$

Output – Selects tuples from books where subject is 'database' and 'price' is 450.

$\sigma_{\text{subject = "database" and price = "450" or year > "2010"}}(\text{Books})$

Output – Selects tuples from books where subject is 'database' and 'price' is 450 or those books published after 2010.

Project Operation (∏):

It projects column(s) that satisfy a given predicate.

Notation – $\prod_{A1, A2, An}$ (r) Where A_1, A_2, A_n are attribute names of relation **r**.

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

For example –

$\prod_{\text{subject, author}}$ (Books)

Selects and projects columns named as subject and author from the relation Books.

Rename Operation (ρ):

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

Notation – ρ_{x} (E)

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

Binary operations: Operates on pairs of relation. These include:

- Union
- Set Difference
- Cartesian product
- Division
- Joins

Union Operation (∪***):***

It performs binary union between two given relations and is defined as –

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

Notation –: r ∪ s

Where **r** and **s** are either database relations or relation result set (temporary relation).

For a union operation to be valid, the following conditions must hold –

- **r**, and **s** must have the same number of attributes.
- Attribute domains must be compatible.
- Duplicate tuples are automatically eliminated.

Set Difference (−):

The result of set difference operation is tuples, which are present in one relation but are not in the second relation.

Notation –: (**r − s)**

Finds all the tuples that are present in **r** but not in **s**.

∏ $_{\text{author}}$ (Books) − ∏ $_{\text{author}}$ (Articles)

Output − Provides the name of authors who have written books but not articles.

Cartesian Product (X):

Combines information of two different relations into one.

Notation − r X s

Where **r** and **s** are relations and their output will be defined as −

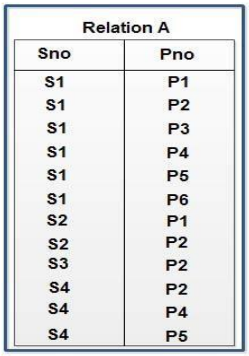
r X s = { q t | q ∈ r and t ∈ s}

σ$_{\text{author = 'tutorial'}}$(Books X Articles)

Output − Yields a relation, which shows all the books and articles written by tutorial.

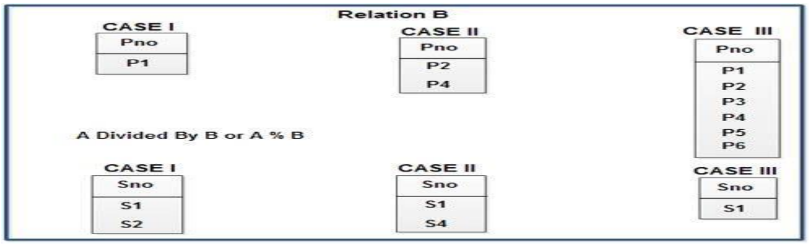
Division:

The division operator divides a dividend relation A of degree (means number of columns in a relation) m + n by a divisor relation B of degree n, and produces a\ resultant relation of degree m. It is denoted by ÷.



Relation A

Sno	Pno
S1	P1
S1	P2
S1	P3
S1	P4
S1	P5
S1	P6
S2	P1
S2	P2
S3	P2
S4	P2
S4	P4
S4	P5



Relation B

CASE I

Pno
P1

CASE II

Pno
P2
P4

CASE III

Pno
P1
P2
P3
P4
P5
P6

A Divided By B or A % B

CASE I

Sno
S1
S2

CASE II

Sno
S1
S4

CASE III

Sno
S1

In this example dividend relation A has two attributes of Sno, Pno (of degree 2) and division relation B has only one attribute Pno (of degree 1). Then, A divide by B gives a resultant relation of degree 1. It means it has only one attribute of Sno.

A = SNO* SNO = SNO. B = PNO

The resultant relation has those tuples that are common values of those attributes, which appears in the resultant attribute sno. *For* example, in CASE II,

P2 has Snos. -- SI, S2, S3,S4 P4 has Snos. -- SI, S4

S1, S4 are the common supplier who supply both P2 and P4. So the resultant relation has tuples S1 and S4.

In CASE III : There is only one supplier S1 who supply all the parts from PI to P6.

Joins:

We understand the benefits of taking a Cartesian product of two relations, which gives us all the possible tuples that are paired together. But it might not be feasible for us in certain cases to take a Cartesian product where we encounter huge relations with thousands of tuples having a considerable large number of attributes.

Join is a combination of a Cartesian product followed by a selection process. A Join operation pairs two tuples from different relations, if and only if a given join condition is satisfied.

We will briefly describe various join types in the following sections.

Theta (θ) Join:

Theta join combines tuples from different relations provided they satisfy the theta condition. The join condition is denoted by the symbol **θ**.

Notation

$R1 \bowtie_{\theta} R2$

R1 and R2 are relations having attributes (A1, A2, .., An) and (B1, B2,.. ,Bn) such that the attributes don’t have anything in common, that is R1 ∩ R2 = Φ.

Theta join can use all kinds of comparison operators.

Student		
SID	**Name**	**Std**
101	Alex	10
102	Maria	11

Subjects	
Class	**Subject**
10	Math
10	English
11	Music
11	Sports

Student_Detail −

$STUDENT \bowtie_{Student.Std = Subject.Class} SUBJECT$

Student_detail

SID	Name	Std	Class	Subject
101	Alex	10	10	Math
101	Alex	10	10	English
102	Maria	11	11	Music
102	Maria	11	11	Sports

Equijoin:

When Theta join uses only **equality** comparison operator, it is said to be equijoin. The above example corresponds to equijoin.

Natural Join (⋈):

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

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

Courses

CID	Course	Dept
CS01	Database	CS
ME01	Mechanics	ME
EE01	Electronics	EE

HoD

Dept	Head
CS	Alex
ME	Maya
EE	Mira

Courses ⋈ HoD

Dept	CID	Course	Head
CS	CS01	Database	Alex
ME	ME01	Mechanics	Maya
EE	EE01	Electronics	Mira

Outer Joins: Theta Join, Equijoin, and Natural Join are called inner joins. An inner join includes only those tuples with matching attributes and the rest are discarded in the resulting relation. Therefore, we need to use outer joins to include all the tuples from the participating relations in the resulting relation. There are three kinds of outer joins − left outer join, right outer join, and full outer join.

Left Outer Join(R ⟕S):

All the tuples from the Left relation, R, are included in the resulting relation. If there are tuples in R without any matching tuple in the Right relation S, then the S-attributes of the resulting relation are made NULL.

Left	
A	**B**
100	Database
101	Mechanics
102	Electronics

Right	
A	**B**
100	Alex
102	Maya
104	Mira

Courses ⟕HoD			
A	**B**	**C**	**D**
100	Database	100	Alex
101	Mechanics	---	---
102	Electronics	102	Maya

Right Outer Join: (R ⟖S):

All the tuples from the Right relation, S, are included in the resulting relation. If there are tuples in S without any matching tuple in R, then the R-attributes of resulting relation are made NULL.

Courses ⟖HoD			
A	**B**	**C**	**D**
100	Database	100	Alex
102	Electronics	102	Maya

---	---	104	Mira

Full Outer Join: (R ⟗S):

All the tuples from both participating relations are included in the resulting relation. If there are no matching tuples for both relations, their respective unmatched attributes are made NULL.

Courses ⟗HoD

A	**B**	**C**	**D**
100	Database	100	Alex
101	Mechanics	---	---
102	Electronics	102	Maya
---	---	104	Mira

TEXT BOOK:

1. Raghurama Krishnan, Johannes Gehrke, Database Management Systems, 3rd Edition, TATA McGraw hill.
2. Web pages

UNIT-3

3.1 Examples of Basic SQL Queries:

SQL is a language which is used to interact with relational database management system. Before we learn what is a SQL and what we can do with the SQL, lets learn the basics first.

What is SQL?

1. SQL stands for Structured Query Language, which is a standardised language for interacting with RDBMS (Relational Database Management System). Some of the popular relational database example are: MySQL, Oracle, mariaDB, postgreSQL etc.

2. SQL is used to perform C.R.U.D (Create, Retrieve, Update & Delete) operations on relational databases.

3. SQL can also perform administrative tasks on database such as database security, backup, user management etc.

4. We can create databases and tables inside database using SQL.

3.1.1 Types of Structured Query Language(SQL):

In the above section, we learned what we do with the database using SQL. SQL is basically combination of four different languages, they are –

DQL (Data Query Language): DQL is used to fetch the information from the database which is already stored there.

DDL (Data Definition Language): DDL is used to define table schemas.

DCL (Data Control Language): DCL is used for user & permission management. It controls the access to the database.

DML (Data Manipulation Language): DML is used for inserting, updating and deleting data from the database.

SQL CREATE Database:

The SQL CREATE DATABASE statement is used to create new SQL database.

Syntax: Basic syntax of CREATE DATABASE statement is as follows:

```
CREATE DATABASE DatabaseName;
```

Always database name should be unique within the RDBMS.

Example:

If you want to create new database <testDB>, then CREATE DATABASE statement would be as follows:

```
SQL> CREATE DATABASE testDB;
```

Make sure you have admin privilege before creating any database. Once a database is created, you can check it in the list of databases as follows:

```
SQL> SHOW DATABASES;
```

SQL CREATE Table:

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

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

Syntax:

Basic syntax of CREATE TABLE statement is as follows:

```
CREATE TABLE table_name(  column1 datatype,
                          column2 datatype,
                          column3 datatype,
                          .....
                          columnN datatype,
                          PRIMARY KEY( one or more columns )
                         );
```

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

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

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

Create Table Using another Table:

A copy of an existing table can be created using a combination of the CREATE TABLE statement and the SELECT statement.

The new table has the same column definitions. All columns or specific columns can be selected.

When you create a new table using existing table, new table would be populated using existing values in the old table.

Syntax:

The basic syntax for creating a table from another table is as follows:

```
CREATE TABLE NEW_TABLE_NAME AS
         SELECT [ column1, column2...columnN ]
          FROM EXISTING_TABLE_NAME
         [ WHERE ]
```

Here, column1, column2...are the fields of existing table and same would be used to create fields of new table.

Example:

Following is an example, which would create a table SALARY using CUSTOMERS table and having fields customer ID and customer SALARY:

SQL> CREATE TABLE SALARY AS SELECT ID, SALARY FROM CUSTOMERS;

This would create new table SALARY, which would have the following records:

SQL DROP or DELETE Table:

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

NOTE: You have to be careful while using this command because once a table is deleted then all the information available in the table would also be lost forever.

Syntax:

Basic syntax of DROP TABLE statement is as follows:

```
DROP TABLE table_name;
```

Example:

Let us first verify CUSTOMERS table and then we would delete it from the database:

SQL> DESC CUSTOMERS;

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

This means CUSTOMERS table is available in the database, so let us drop it as follows:

SQL> DROP TABLE CUSTOMERS;

```
Query OK, 0 rows affected (0.01 sec)
```

Now, if you would try DESC command, then you would get error as follows:

SQL> DESC CUSTOMERS;

```
ERROR 1146 (42S02): Table 'TEST.CUSTOMERS' doesn't exist
```

Here, TEST is database name which we are using for our examples.

SQL INSERT Query:

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

Syntax:

There are two basic syntaxes of INSERT INTO statement as follows:

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

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

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

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

Example:

Following statements would create six records in CUSTOMERS table:

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
        VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
        VALUES (2, 'Khilan', 25, 'Delhi', 1500.00 );
```

we can create a record in CUSTOMERS table using second syntax as follows:

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

SQL SELECT Query:

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

Syntax:

The basic syntax of SELECT statement is as follows:

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

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

```
SELECT * FROM table_name;
```

Example:

Following is an example, which would fetch ID, Name and Salary fields of the customers available in CUSTOMERS table:

SQL> SELECT ID, NAME, SALARY FROM CUSTOMERS;

SQL WHERE Clause:

The SQL WHERE clause is used to specify a condition while fetching the data from single table or joining with multiple tables.

If the given condition is satisfied, then only it returns specific value from the table. You would use WHERE clause to filter the records and fetching only necessary records.

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

Syntax:

The basic syntax of SELECT statement with WHERE clause is as follows:

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

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

SQL AND and OR Operators:

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

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

The AND Operator:

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

Syntax:

The basic syntax of AND operator with WHERE clause is as follows:

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

Example:

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

SQL UPDATE Query:

The SQL UPDATE Query is used to modify the existing records in a table.

You can use WHERE clause with UPDATE query to update selected rows, otherwise all the rows would be affected.

Syntax:

The basic syntax of UPDATE query with WHERE clause is as follows:

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

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

Example:

Following is an example, which would update ADDRESS for a customer whose ID is 6:

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

SQL DELETE Query

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

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

Syntax:

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

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

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

Example:

Following is an example, which would DELETE a customer, whose ID is 6:

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

SQL LIKE Clause:

The SQL LIKE clause is used to compare a value to similar values using wildcard operators. There are two wildcards used in conjunction with the LIKE operator:

The percent sign (%)

The underscore (_)

The percent sign represents zero, one, or multiple characters. The underscore represents a single number or character. The symbols can be used in combinations.

Syntax:

The basic syntax of % and _ is as follows:

```
SELECT FROM table_name
        WHERE column LIKE 'XXXX%'
or
 SELECT FROM table_name
        WHERE column LIKE '%XXXX%'
or
SELECT FROM table_name
        WHERE column LIKE 'XXXX_'
or
SELECT FROM table_name
        WHERE column LIKE '_XXXX'
or
SELECT FROM table_name
        WHERE column LIKE '_XXXX_'
```

You can combine N number of conditions using AND or OR operators. Here, XXXX could be any numeric or string value.

Example:

```
SQL> SELECT * FROM CUSTOMERS
                WHERE SALARY LIKE '200%';
```

SQL ORDER BY Clause:

The SQL ORDER BY clause is used to sort the data in ascending or descending order, based on one or more columns. Some database sorts query results in ascending order by default.

Syntax:

The basic syntax of ORDER BY clause is as follows:

```
SELECT column-list  FROM table_name [WHERE condition]
        [ORDER BY column1, column2, .. columnN] [ASC | DESC];
```

You can use more than one column in the ORDER BY clause. Make sure whatever column you are using to sort, that column should be in column-list.

Example: Following is an example, which would sort the result in descending order by NAME:

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

```
ORDER BY NAME DESC;
```

SQL Group By:

The SQL GROUP BY clause is used in collaboration with the SELECT statement to arrange identical data into groups.

The GROUP BY clause follows the WHERE clause in a SELECT statement and precedes the ORDER BY clause.

Syntax:

The basic syntax of GROUP BY clause is given below. The GROUP BY clause must follow the conditions in the WHERE clause and must precede the ORDER BY clause if one is used.

```
SELECT column1, column2
        FROM table_name
        WHERE [ conditions ]
        GROUP BY column1, column2
        ORDER BY column1, column2
```

Example:

```
SQL> SELECT NAME, SUM(SALARY) FROM CUSTOMERS
                GROUP BY NAME;
```

SQL Distinct Keyword:

The SQL DISTINCT keyword is used in conjunction with SELECT statement to eliminate all the duplicate records and fetching only unique records.

There may be a situation when you have multiple duplicate records in a table. While fetching such records, it makes more sense to fetch only unique records instead of fetching duplicate records.

Syntax:

The basic syntax of DISTINCT keyword to eliminate duplicate records is as follows:

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

Example:

First, let us see how the following SELECT query returns duplicate salary records:

```
SQL> SELECT SALARY FROM CUSTOMERS
                ORDER BY SALARY;
```

SQL SORTING Results:

The SQL ORDER BY clause is used to sort the data in ascending or descending order, based on one or more columns. Some databases sort query results in ascending order by default.

Syntax:

The basic syntax of ORDER BY clause which would be used to sort result in ascending or descending order is as follows:

```
SELECT column-list
        FROM table_name
        [WHERE condition]
        [ORDER BY column1, column2, .. columnN] [ASC | DESC];
```

You can use more than one column in the ORDER BY clause. Make sure whatever column you are using to sort, that column should be in column-list.

Example:

```
SQL> SELECT * FROM CUSTOMERS
                ORDER BY NAME, SALARY;
```

3.2 Nested Queries and sub queries:

A Subquery or Inner query or a Nested query is a query within another SQL query and embedded within the WHERE clause.

A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.

Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.

Syntax:

```
select columnlist
from tablename
where column operator (select column from tablename
                where column operator …..));
```

There are a few rules that subqueries must follow:−

- Subqueries must be enclosed within parentheses.
- A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns.
- An ORDER BY command cannot be used in a subquery, although the main query can use an ORDER BY. The GROUP BY command can be used to perform the same

function as the ORDER BY in a subquery.

- Subqueries that return more than one row can only be used with multiple value operators such as the IN operator.
- A subquery cannot be immediately enclosed in a set function.
- The BETWEEN operator cannot be used with a subquery. However, the BETWEEN operator can be used within the subquery.

- A subquery can a part of column in the select statement is called <u>SUBSET.</u>
- A <u>INLINE VIEW</u> is a select statement in the FROM clause of another select statement

  Example:

  ```
  select * from (select deptno , count(*) emp_count
          from emp
          group by deptno ) emp , dept
          where dept.deptno = emp.deptno;
  ```

- A subquery in the where clause of the select statement is called <u>NESTED SUBQUERY</u>

  Example:

  ```
  select *from emp
          where sal > (select sal from emp where empno = 7566);
  ```

3.2.1 Types of subqueries:

I. Single row subquery: The query returns only one row from the inner select statement. The operators used in this query are >, <, =, <=, >=.

Example:

Display the emp details whose salary is equal to minimum salary in company.

```
select ename,job,sal from emp
        where sal = (select min(sal) from emp);
```

II. Multiple row subquery: The query returns more than one row from inner select statement. The operators used in these queries are ANY, ALL, IN.

Example:

Display who are getting minimum salary of each dept

```
select ename, sal from emp
        where sal in (select min(sal) from emp group by deptno);
```

III. Multiple column subquery: The query returns more than one column from the inner select statement. The column comparison in a multiple column subquery can be 1) pair wise comparison   2) Non-pair wise comparison.

IV. Correlated subquery: It is another way of performing queries upon the data with simulation of joins. In this information from outer select statement participates as the condition in the inner select statement. The operators used in these queries are exists, not exists.

Steps to be performed in correlated subqueries:

1. First the outer query is executed.
2. Passes the qualified column value to the inner queries "where" clause.
3. Then the inner query or candidate query is executed and the result is passed is passed to the outer query where clause.
4. Depending upon the separate values the condition is qualified for the specific record.

Example: Display deptno, dept name there exist at least one employee in dept

```
select d.deptno,d.dname from dept d
        where exist (select * from emp e where d.deptno = e.deptno);
```

Set Comparison Operators: We have seen the set comparison operators EXISTS, IN &UNIQUE along with their negation. ALL, ANY are the arithmetic operators (<, >, <=, >=, <>, =).

Example: Find sailors whose rating is better than some sailor called horatio

```
select s.sid from sailors s where s.rating > any(select s2.rating
                                   from sailors s2 where s2.name = 'horatio');
```

3.2.3 Aggregative Operators: The aggregative functions are used in the subqueries like max(), min(), avg()…. It is used in the in the multiple row subqueries because in these queries returns more than one row for the outer query.

Example: Display who are getting the minimum salary of each department.

```
select ename, sal from emp
        where sal in (select min(sal) from emp group by deptid);
```

NULL VALUES:

SQL provides a special column value called null to use null when the column value is unknown or inapplicable.

Example: create table student (sid integer constraint student_sid_nn not null,

```
                name char(20));
```

In the above example sid doesn't allow the null values in the column.

Impact on SQL Constructs: Boolean expressions arise in many contexts in SQL, and the impact of null values must be recognized. In the presence of null value any row that evaluates to false or unknown is eliminated so that there is an impact on the nested queries.

The arithmetic operators all return null if one of their arguments is null. The aggregate operators discard the null values.

3.2.4 Outer Joins:

Some interesting variants of the join operation that rely on null values called outer joins.

Left Outer Join:

To retrieve the missing records from the left side table of the join condition.

Example: Display name, dept no, dept name from tables

```
select e.ename, d.deptno, d.dname from emp e, dept d
        where e.deptno = d.deptno(+);
```

Right Outer Join:

To retrieve the missing records from the right side table of the join condition.

Example: Display name, dept name, dept no from tables.

```
select e.ename,e.deptno,d.dname   from emp e, dept d
        where e.deptno(+) = d.deptno;
```

Full Outer Join:

To retrieve the missing records from the both side table of the join condition.

3.2.5 Complex Integrity Constraints in SQL:

Constraints over single table:

Constraints can specify over a single table using table constraints by CHECK conditional expression.

Domain Constraints and distinct types:

A user can define a new domain using the CREATE DOMAIN statement which uses CHECK constraints.

3.2.6 Triggers:

A trigger is a procedure that is automatically invoked by the DBMS in response to special changes to the database and DBA. A database that has a set of associated triggers is called an ACTIVE database.

- Event: A changes to the data base that activates the trigger.
- Condition: A query or test that is run when the trigger is activated.
- Action: A procedure that is executed when the trigger is activated and the condition is true.

A trigger can be thought of as a daemon that monitors a database and executed when the database is modified in a way that matches the event specification

Eg: Write a trigger to display salary difference of old and new values.

```
        set serveroutput on;
        create or replace trigger emp_diff
        after insert or delete or update on emp      for each row
        declare diff number;
    begin
        diff := :new.sal – :old.sal;
        dbms_output.put_line(‘old salary’: || :old.sal);
        dbms_output.put_line(‘new salary’: || :new.sal);
        dbms_output.put_line(‘ salary difference’: || diffl);
    end;
```

Examples of sub queries

1. Display the details of all employees
2. Display the depart information from department table
3. Display the name and job for all the employees
4. Display the name and salary for all the employees
5. Display the employee no and total salary for all the employees
6. Display the employee name and annual salary for all employees.
7. Display the names of all the employees who are working in depart number 10.
8. Display the names of all the employees who are working as clerks and drawing a salary more than 3000.
9. Display the employee number and name who are earning comm.
10. Display the employee number and name who do not earn any comm.
11. Display the names of employees who are working as clerks, salesman or analyst and drawing a salary more than 3000.
12. Display the names of the employees who are working in the company for the past 5 years;
13. Display the list of employees who have joined the company before 30-JUN-90 or after 31-DEC-90.
14. List the emps who are joined in the year 1981
15. List the emps who are joined in the month of Aug 1980

16. List the emps whose annul sal ranging from 22000 and 45000

17. Display the names of employees working in depart number 10 or 20 or 40 or employees working as CLERKS, SALESMAN or ANALYST.
18. Display the names of employees whose name starts with alphabet S.
19. Display the Employee names for employees whose name ends with Alphabet S.
20. Display the names of employees whose names have second alphabet A in their names.
21. Select the names of the employee whose names is exactly five characters in length.
22. Display the names of the employee who are not working as MANAGERS.
23. Display the names of the employee who are not working as SALESMAN OR CLERK OR ANALYST.
24. Display the total number of employee working in the company.
25. Display the total salary being paid to all employees.
26. Display the maximum salary and maximum salary and average salary from emp table.
27. Display the maximum salary being paid to CLERK.
28. Display the maximum salary being paid to depart number 20.
29. Display the minimum salary being paid to any SALESMAN.
30. Display the average salary drawn by MANAGERS.
31. Display the total salary drawn by ANALYST working in depart number 30
32. Display empno,ename,deptno,sal sort the output first base on name and within name by deptno and with in deptno by sal.
33. Display department numbers and total number of employees working in each department.
34. Display the various jobs and total number of employees within each job group.
35. Display the depart numbers and total salary for each department.
36. Display the depart numbers and max salary for each department.
37. Display the various jobs and total salary for each job
38. Display the depart numbers with more than three employees in each dept.
39. Display the various jobs along with total salary for each of the jobs where total salary is greater than 40000.
40. Display the various jobs along with total number of employees in each job. The output should contain only those jobs with more than three employees.
41. Display the jobs found in department 10 and 20 Eliminate duplicate jobs.
42. Display the distinct jobs in the department 10.

43. Display those department whose name start with "S" while the location name ends with "K".
44. Display those employees whose salary is more than 3000 after giving 20% increment.
45. Display those employees who are not working under any manager.
46. Select count of employee in each department where count greater than 3?
47. List out employee name and salary increased by 15% and expressed as whole number of Dollars?
48. Find out the average salary and average total remuneration for each job type
49. List the emps whose annul sal ranging from 22000 and 45000
50. List the emps who joined in January
51. select second max salary from emp

TEXT BOOK:

1. Raghurama Krishnan, Johannes Gehrke, Database Management Systems, 3rd Edition, TATA McGraw hill.
2. Web pages

Unit – 4

Schema Refinement and Normal forms

Schema:

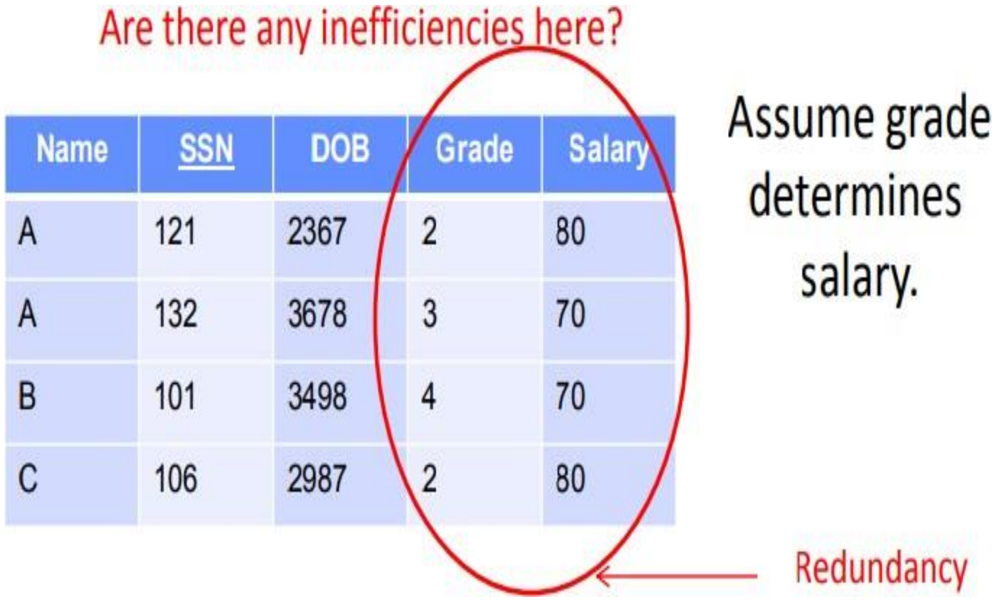
A Schema can be defined as a complete description of database. The Specifications for database schema is provided during the database design and this schema does not change frequently.

4.1 Schema Refinement:

Schema Refinement is a process of refining the schema so as to solve the problems caused by redundantly storing the information.

Redundancy means duplication of data. Redundancy is at the root of several problems associated with relational schemas. Some of them are

- Redundant storage,
- Insert/delete/update anomalies.



Are there any inefficiencies here?

Name	SSN	DOB	Grade	Salary
A	121	2367	2	80
A	132	3678	3	70
B	101	3498	4	70
C	106	2987	2	80

Assume grade determines salary.

Redundancy

Figure 4.1

4.2 Problems caused by Redundancy:

4.2.1. Data Inconsistency

4.2.2. Memory Fragmentation

4.2.3. Anomalies:

There are three types of anomalies:

i. update,

ii. deletion and

iii. insertion anomalies.

Let us consider an example :

Each employee in a company has a department associated with them as well as the student group they participate in.

Employee_ID	**Name**	**Department**	**Student_Group**
123	James	Accounting	Pst club
234	B. Rech	Marketing	Marketing Club
234	B. Rech	Marketing	Management Club
456	Anand	CIS	Technology Org.
456	Anand	CIS	Pst club

4.3.2.1 Update Anomaly:

An update anomaly is a data inconsistency that results from data redundancy and a partial update.

For example, if Anand's department is an error it must be updated at least 2 times or there will be inconsistent data in the database. If the user performing the update does not realize the data is stored redundantly the update will not be done properly.

4.3.2.2 Deletion anomaly :

A deletion anomaly is the unintended loss of data due to deletion of other data.

For example, if the student group Pst club is disbanded and was deleted from the table above, James and the Accounting department would cease to exist. This results in database inconsistencies and is an example of how combining information that does not really belong together into one table can cause problems.

4.3.4.3 Insertion anomaly :

An insertion anomaly is the inability to add data to the database due to absence of other data.

For example, assume Student_Group is defined so that null values are not allowed. If a new employee is hired but not immediately assigned to a Student_Group then this employee could not be entered into the database. This results in database inconsistencies due to omission.

Update, deletion, and insertion anomalies are very undesirable in any database. Anomalies are avoided by the process of normalization.

Ways to avoid Data Anomalies:

There are two ways to avoid data anomalies. They are :

1. Normalization
2. Decomposition

4.4 Functional Dependency :

A Functional dependency is defined as the relationship between the attributes that correspond to a single relation.

A functional dependency (FD) has the form X -> Y (read as X functionally determines Y) where X and Y are sets of attributes in a relation R. Here X is used to determine the value of Y,so it is said that Y is functionally dependent on X.

Example : A student can have only one birth year : **S → B**

Functional Dependencies in entity sets :

An entity set can have the following functional dependencies :

4.4.1. Fully Functional Dependency:

If x and y are attributes of an entity set in a table such that y is functionally dependent only on x, but not on any proper subset of x,then this type of dependency is called as Fully Functional Dependency.

Eg : RollNo, SubName -> Marks.

4.4.2. Partial Functional Dependency:

If x and y are attributes of an entity set in a table such that y is functionally dependent only on x and elimination of some attributes from x does not affect the dependency, then this type of dependency is called as Partial Functional Dependency.

Eg :emp_id, emp_name -> salary.

4.4.3. Transitive Functional Dependency:

If x,y,z are attributes of an entity set in a table such that x is functionally dependent on y and y is functionally dependent on z, then z will be transitively dependent on x through y.

Eg: Students -> Teachers
Teachers -> Management
Management -> Students

4.5 Reasoning about FD's:

Armstrong's Axioms:

William W. Armstrong established a set of rules which can be used to infer the functional dependencies in a relational database (from umbc.edu - no external linking, Google Database Design UMBC):

- **Reflexivity rule**:

  If A is a set of attributes, and B is a set of attributes that are completely contained in A, then A implies B.

- **Augmentation rule:**

If A implies B, and C is a set of attributes, then if A implies B, then AC implies BC.

- **Transitivity rule:**

  If A implies B and B implies C, then A implies C.

These can be simplified if we also use:

- **Union rule:** If A implies B and A implies C, the A implies BC.
- **Decomposition rule**: If A implies BC then A implies B and A implies C.
- **Pseudo transitivity rule**: If A implies B and CB implies D, then AC implies D.

4.6 Normalization:

Normalization is a relational database management system design concept which is a process of designing the database structure such that it minimizes the data redundancy and also data anomalies.

The process of normalization includes a series of stages known as Normal Forms. Normalization rules are divided into following normal form :

1. First Normal Form (1 NF)
2. Second Normal Form (2 NF)
3. Third Normal Form (3 NF)
4. Boyce-Codd Normal Form (BCNF)

4.6.1 First Normal Form (1 NF) :

A relation is in first Normal Form if and only if all underlying domains contain atomic values only. In other words, a relation doesn't have multivalve attributes.

For example:

Consider a ***STUDENT (Sid, Sname, Cname)*** relation.

Student :		
SID	**Sname**	**Cname**
S1	A	C,C++
S2	B	C++,DB
S3	A	DB
SID : Primary Key		

⇐

Due to occurrence of MVA, the above relation is not in 1 NF.

<table>
<tr><td>U</td><td colspan="2"><b>Solution :</b>
Removal of MVA by inserting more rows</td></tr>
<tr><td>
<table>
<tr><td colspan="3"><b>Student :</b></td></tr>
<tr><td><b>SID</b></td><td><b>Sname</b></td><td><b>Cname</b></td></tr>
<tr><td>S1</td><td>A</td><td>C</td></tr>
<tr><td>S1</td><td>A</td><td>C++</td></tr>
<tr><td>S2</td><td>B</td><td>C++</td></tr>
<tr><td>S2</td><td>B</td><td>DB</td></tr>
<tr><td>S3</td><td>A</td><td>DB</td></tr>
<tr><td colspan="3"><b>SID : Primary Key</b></td></tr>
</table>
</td><td>⇐</td><td>The relation is in 1NF</td></tr>
</table>

4.6.2 2NF – Second Normal Form :

Relation ***R*** is in Second Normal Form (2NF) only iff :

1. ***R*** should be in 1NF and
2. ***R*** should not contain any *Partial Dependency*

Partial Dependency :
Let ***R*** be a relational Schema and ***X,Y,A*** be the attribute sets over ***R***. ***X:*** Any Candidate Key ***Y:*** Proper Subset of Candidate Key ***A:*** Non Key Attribute If Y → A exists in R, then R is not in 2 NF. ***(Y → A)*** is a Partial dependency only if ***Y:*** Proper subset of Candidate Key ***A:*** Non Prime Attribute

Removal of Partial Dependency

If there is any partial dependency, remove partially dependent attributes from original table, place them in a separate table along with a copy of its determinant.

Example 1 :Consider the relation

Student(SID, Sname, Cname) which is in 1 NF (No Multi-Valued-Attributes) :

Student :

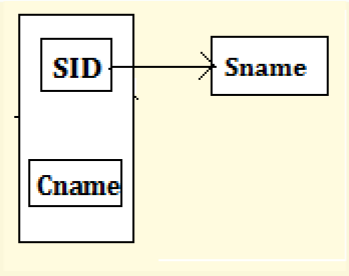
SID	**Sname**	**Cname**
S1	A	C
S1	A	C++
S2	B	C++
S2	B	DB
S3	A	DB

{SID,Cname} : Primary Key

<u>Functional Dependencies</u>:

{SID,Cname} → Sname

SID → Sname



Partial Dependencies :

SID → Sname {as SID is a Proper Subset of Candidate Key {SID,Cname}.

Solution : Removal of Partial Dependency by creating separate table

R1 :

SID	**Sname**
S1	A
S2	B
S3	A

**SID : **Primary Key

U

R2 :

SID	**Cname**
S1	C
S1	C++
S2	C++
S2	DB
S3	DB

{SID,Cname} : Primary Key

The above two relations R1 and R2 are Lossless Join and Dependency Preserving . They were in 2NF

* There is less redundancy in 2NF rather than in 1 NF, but 2NF is not free from redundancy.

4.6.3 3NF – Third Normal Form:

Let ***R*** be the relational schema, ***R*** is in 3NF only if :

- ***R*** should be in 2NF.
- ***R*** should not contain *transitive dependencies.*

Transitive Dependency

Let ***R*** be a relational Schema and ***X,Y,Z*** be the attribute sets over ***R.***

If ***X*** is functionally dependent on ***Y (X → Y)***

and ***Y*** is functionally dependent on ***Z (Y → Z)***

then ***X*** is transitive dependent on ***Z (X → Z)***

Removal of Transitive Dependency

If there is any transitive dependency in the relation, then

- Create a separate relation and copy the dependent attribute along with a copy of its determinant. and remove these determinants from the original table.
- Mark dependent attribute as a foreign key in the original relation and Mark dependent attribute as a Primary key in the separate relation

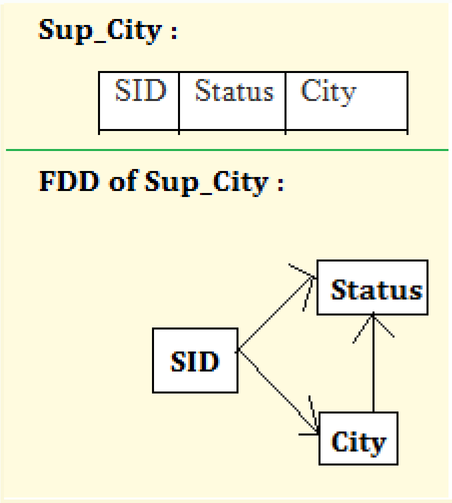
Example of 3NF :

Let us consider the relation **Sup_City(SID, Status, City)** :

Sup_City :

SID	Status	City
S1	30	Delhi
S2	10	Karnal
S3	40	Rohtak
S4	30	Delhi

SID : Primary Key



Transitive Dependency :

SID → City {As SID → City and City → Status}

Solution :

Removal of Transitive Dependency by creating separate table

U

SC :	
SID	**City**
S1	Delhi
S2	Karnal
S3	Rohtak
S4	Delhi
SID : Primary Key	

CS :	
City	**Status**
Delhi	30
Karnal	10
Rohtak	40
City : Primary Key	

The relations SC and CS are in 3NF as they doesn't contain any transitive dependencies.

4.6.4 BCNF – Boyce Codd Normal Form:

Let ***R*** be the relational schema, ***R*** is in BCNF only if :

- ***R*** should be in 3NF.
- Every Functional Dependency will have a Superkey on the LHS or all determinants are the superkeys.

Example :

Consider the following relationship ***R(ABCD)*** having following functional dependencies:

F = {A → BCD, BC → AD, D → B}		
Candidate Keys are : $(A)^+$ = {ABCD} $(BC)^+$ = {BCAD} $(DC)^+$ = {DCBA}		
Functional Dependency	**Is FD in BCNF or not ?**	**Reason ?**
A → BCD	Yes	A is a super key
BC → AD	Yes	BC is also a super key
D → A	No	D is not super key, it is part of key

Solution :

Decomposition in BCNF:

The relation ***R(ABCD)*** is decomposed into two relations ***R1*** and ***R2*** such that :

R1(A,D,C) and **R2(D,B).**

The above two relations ***R1*** and ***R2***

1. Lossless Join
2. BCNF Decomposition
3. But Not Dependency Preserving

Redundancy in BCNF

There will be 0% redundancy, because of Single Valued Functional Dependency. Redundancy may exist because of **Multivalved Dependency.**

Multivalued Dependency

Consider a relation Faculty (FID, Course, Book) which consists of two multivalued attributes valued attributes are independent of each other.

FID	Course	Book
1	C1/C2	B1/B2
2	C1	B1

Differences between 3NF and BCNF :

S.NO	3NF	BCNF
1.	It concentrates on Primary Key	It concentrates on Candidate Key.
2.	Redundancy is high as compared to BCNF	0% redundancy
3.	It may preserve all the dependencies	It may not preserve the dependencies.
4.	A dependency X → Y is allowed in 3NF if X is a super key or Y is a part of some key.	A dependency X → Y is allowed if X is a super key

Decomposition:

Decomposition is the process of dividing longer relations into smaller relations. This division is based on functional and other dependencies which are specified by database designer.

The decomposition of a relation scheme R consists of replacing the relation schema by two or more relation schemas that each contain a subset of the attributes of R and together include all attributes in R.

Problems related to Decomposition:

Decomposing a relation schema can create more problems unless we handle them carefully.

- Two important questions must be avoided repeatedly.They are:
    1. Do we need to decompose a relation ?
    2. What problems does a given decomposition cause ?
- To answer the first question, several normal forms have been proposed for relations. If a relation schema is in one of these normal forms, we know that certain kinds of problems cannot arise.
- To answer the second question, two properties of decomposition arte of particular interest :
    1. Loss-less join decomposition
    2. Dependency Preserving Decomposition

4.7 Lossless join Decomposition:

Lossy Decomposition :

"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.

STUDENT:

Roll_no	Sname	Dept
111	parimal	COMPUTER
222	parimal	ELECTRICAL

This relation is decomposed into two relation no_name and name_dept :

No_name:

Roll_no	Sname
111	parimal

Sname	Dept
parimal	COMPUTER
parimal	ELECTRICAL

222	parimal

Name_dept :

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.

Lossless Join Decomposition :

"The decomposition of relation R into R1 and R2 is **lossless** when the join of R1 and R2 yield the same relation as in R."

A relational table is decomposed (or factored) into two or more smaller tables, in such a way that the designer can capture the precise content of the original table by joining the decomposed parts. This is called lossless-join (or non-additive join) decomposition.

This is also referred as **non-additive** decomposition.

The lossless-join decomposition is always defined with respect to a specific set F of dependencies.

Example:

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 :

Stu_name:

Roll_no	Sname
111	parimal
222	parimal

Stu_dept :

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.

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.

Tests for lossless join decomposition:

To Identify whether a decomposition is lossy or lossless, it must satisfy the following conditions :

a) $R_1 \cup R_2 = R$

i. 2. $R_1 \cap R_2 \neq \Phi$ and

ii. 3. $R_1 \cap R_2 \rightarrow R_1$ or $R_1 \cap R_2 \rightarrow R_2$

Example 1 :

Let R(ABC) F = {A → B, A → C} decomposed into D = R_1(AB), R_2(BC) Find whether D is Lossless or Lossy ?

Solution :

Given D = {AB, BC}

Step 1: AB ∪ BC = ABC

Step 2: AB ∩ BC = B //Intersection

Step 3: B^+ = {B} //Not a super key of R_1 or R_2

Therefore, the given Decomposition is lossy.

Example 2 :

Let R(ABCDEF) F = {A → B, B → C, C → D, E → F} decomposed into D = R_1(AB), R_2(BCD), R_3(DEF). Find whether D is Lossless or Lossy ?

Solution :

Step 1:

AB ∪ BCD ∪ DEF = ABCDEF = R // Condition 1 satisfies

step 2:

AB ∩ BCD = B

B^+ = {BCD} //superkey of R_2

⇒ R_{12}(ABCD)

ABCD ∩ DEF = D

D^+ = {D} // Not a superkey of R_{12} or R_3

Therfore, the given decomposition is Lossy.

4.8 Dependency Preserving Decomposition:

Another property of decomposition is **Dependency Preserving Decomposition**.

If the original table is decomposed into multiple fragments, then somehow, we suppose to get all original FDs from these fragments. In other words, every dependency in original table must be preserved or say, every dependency must be satisfied by at least one decomposed table.

Let R be the original relational schema having FD set F. Let R1 and R2 having FD set F1 and F2 respectively, are the decomposed sub-relations of R. then there can be three cases:

- F1 U F2 = F > Decomposition is dependency preserving.
- F1 U F2 is a subset of F > Not Dependency preserving.
- F1 U F2 is a super set of F > This case is not possible.

Example:

Let a relation R (A, B, C, D) and functional dependency {AB –> C, C –> D, D –> A}. Relation R is decomposed into R1(A, B, C) and R2(C, D). Check whether decomposition is dependency preserving or not.

Solution:

Given R1(A, B, C) and R2(C, D)

Let us find closure of F1 and F2:

To find closure of F1, consider all combination of ABC. i.e., find closure of A, B, C, AB, BC and AC. Note that ABC is not considered as it is always ABC

closure(A) = { A } // Trivial
closure(B) = { B } // Trivial
closure(C) = {C, A, D} but D can't be in closure as D is not present R1.
= {C, A}
C--> A // Removing C from right side as it is trivial attribute

closure(AB) = {A, B, C, D} = {A, B, C}
AB --> C // Removing AB from right side as these are trivial attributes

closure(BC) = {B, C, D, A} = {A, B, C}
BC --> A // Removing BC from right side as these are trivial attributes

closure(AC) = {A, C, D}
AC --> D // Removing AC from right side as these are trivial attributes

F1 {C--> A, AB --> C, BC --> A}.
Similarly F2 { C--> D }

In the original Relation Dependency {AB --> C , C --> D , D --> A}.
AB --> C is present in F1.
C --> D is present in F2.
D --> A is not preserved.
F1 U F2 is a subset of F.
Therefore, the given decomposition is **not dependency preserving**.

TEXT BOOK:

1. Raghurama Krishnan, Johannes Gehrke, Database Management Systems, 3rd Edition, TATA McGraw hill.
2. Web pages

Unit-5

5.1 Transaction management:

A **transaction** is a set of logically related operations. For example, you are transferring money from your bank account to your friend's account, the set of operations would be like this:

5.1.1 The ACID Properties:

A transaction is a single logical unit of work which accesses and possibly modifies the contents of a database. Transactions access data using read and write operations.
In order to maintain consistency in a database, before and after the transaction, certain properties are followed. These are called ACID properties.

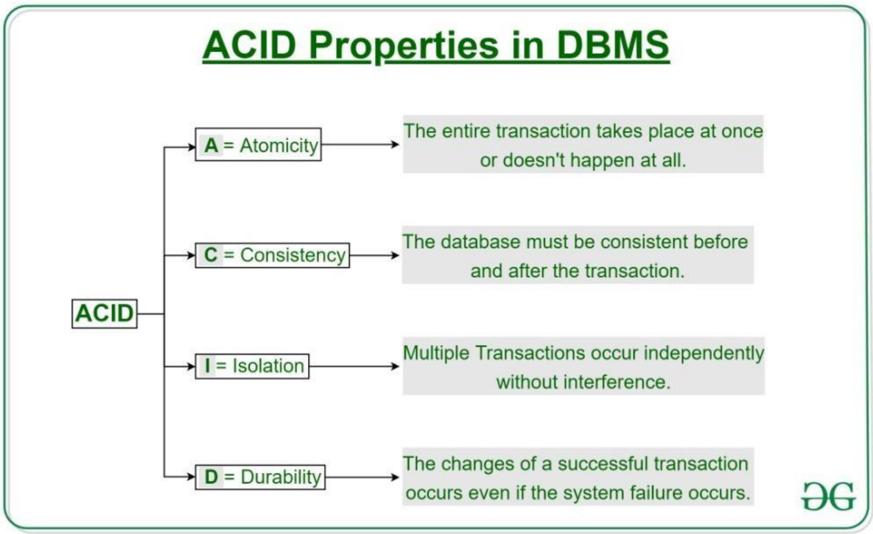

ACID Properties in DBMS
ACID
A = Atomicity
The entire transaction takes place at once or doesn't happen at all.
C = Consistency
The database must be consistent before and after the transaction.
I = Isolation
Multiple Transactions occur independently without interference.
D = Durability
The changes of a successful transaction occurs even if the system failure occurs.

Figure 5.1

5.1.1.1 Atomicity:

By this, we mean that either the entire transaction takes place at once or doesn't happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves the following two operations.

—Abort: If a transaction aborts, changes made to database are not visible.

—Commit: If a transaction commits, changes made are visible.

Atomicity is also known as the 'All or nothing rule'.

5.1.1.2 Consistency:

This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to the correctness of a database. Referring to the example above, The total amount before and after the transaction must be maintained.

Total before T occurs = 500 + 200 = 700.

Total after T occurs = 400 + 300 = 700.

Therefore, database is consistent. Inconsistency occurs in case T1 completes but T2 fails. As a result T is incomplete.

5.1.1.3 Isolation:

This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of database state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed. This property ensures that the execution of transactions concurrently will result in a state that is equivalent to a state achieved these were executed serially in some order.

5.1.1.4 *Durability*:

This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if a system failure occurs. These updates now become permanent and are stored in non-volatile memory. The effects of the transaction, thus, are never lost.

5.1.2 Transactions in DBMS:

A transaction is an action or series of actions that are being performed by a single user or application program, which reads or updates the contents of the database.

A transaction can be defined as a logical unit of work on the database. This may be an entire program, a piece of a program, or a single command (like the SQL commands such as INSERT or UPDATE), and it may engage in any number of operations on the database. In the database context, the execution of an application program can be thought of as one or more transactions with non-database processing taking place in between.

Example of Transaction in DBMS:

A simple example of a transaction will be dealing with the bank accounts of two users, let say Karlos and Ray. A simple transaction of moving an amount of 5000 from Karlos to Ray engages many low-level jobs. As the amount of Rs. 5000 gets transferred from the Karlos's account to Ray's account, a series of tasks gets performed in the background of the screen.
This straightforward and small transaction includes several
	Open_Acc (Karlos)
	OldBal = Karlos.bal
	NewBal = OldBal - 5000
	Ram.bal = NewBal
	CloseAccount(Karlos)

steps: decrease Karlos's bank account from 5000:	

we can say, the transaction involves many tasks, such as opening the account of Karlos, reading the old balance, decreasing the specific amount of 5000 from that account, saving new balance to an account of Karlos, and finally closing the transaction session.
For adding amount 5000 in Ray's account, the same sort of tasks needs to be done:
	OpenAccount(Ray)
	Old_Bal = Ray.bal
	NewBal = OldBal + 1000
	Ahmed.bal = NewBal
	CloseAccount(B)

5.1.3 Schedules in DBMS:

When multiple transactions are being executed by the operating system in a multiprogramming environment, there are possibilities that instructions of one transactions are interleaved with some other transaction.

Schedule − A chronological execution sequence of a transaction is called a schedule. A schedule can have many transactions in it, each comprising of a number of instructions/tasks.

Serial Schedule − It is a schedule in which transactions are aligned in such a way that one transaction is executed first. When the first transaction completes its cycle, then the next transaction is executed. Transactions are ordered one after the other. This type of schedule is called a serial schedule, as transactions are executed in a serial manner.

5.1.3.1 Types of Schedules in DBMS

Schedule, as the name suggests, is a process of lining the transactions and executing them one by one. When there are multiple transactions that are running in a concurrent manner and the order of operation is needed to be set so that the operations do not overlap each other, Scheduling is brought into play and the transactions are timed accordingly.

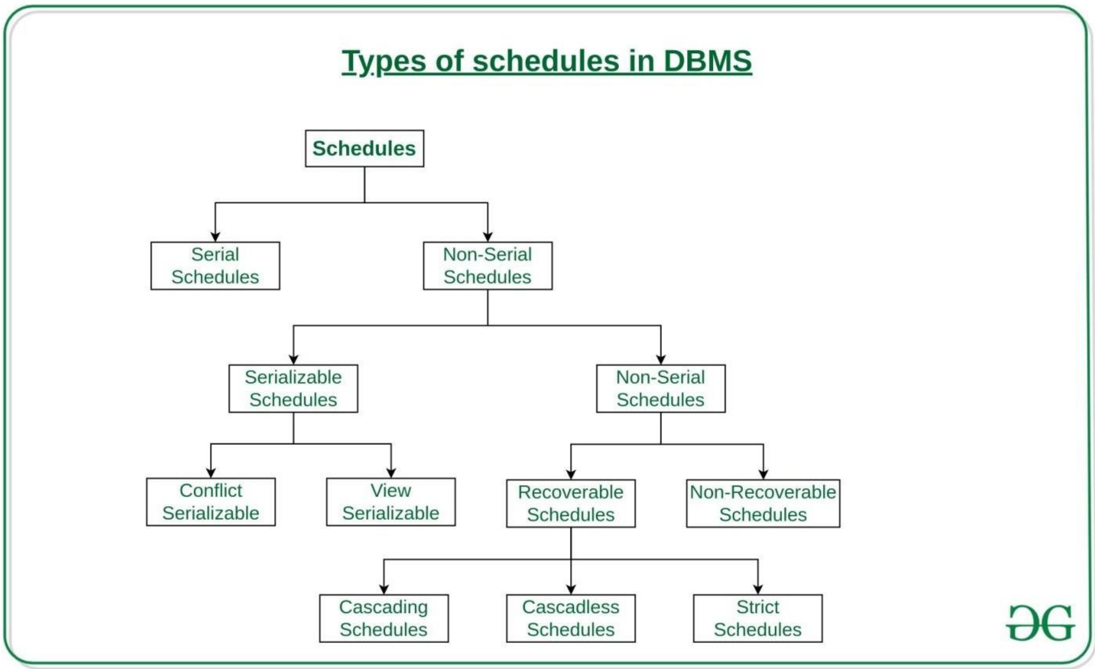


Figure 5.2

5.1.3.1.1. Serial Schedules:

Schedules in which the transactions are executed non-interleaved, i.e., a serial schedule is one in which no transaction starts until a running transaction has ended are called serial schedules.

Example: Consider the following schedule involving two transactions T_1 and T_2.

where R(A) denotes that a read operation is performed on some data item 'A'

This is a serial schedule since the transactions perform serially in the order T_1 —> T_2

T_1	**T_2**
R(A)	
W(A)	
R(B)	
	W(B)
	R(A)
	R(B)

5.1.3.1.2. Non-Serial Schedule:

This is a type of Scheduling where the operations of multiple transactions are interleaved. This might lead to a rise in the concurrency problem. The transactions are executed in a non-serial manner, keeping the end result correct and same as the serial schedule.

The Non-Serial Schedule can be divided further into Serializable and Non-Serializable.

a. Serializable:

This is used to maintain the consistency of the database. It is mainly used in the Non-Serial scheduling to verify whether the scheduling will lead to any inconsistency or not.
A serializable schedule helps in improving both resource utilization and CPU throughput. These are of two types:

i. Conflict Serializable:

A schedule is called conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations. Two operations are said to be conflicting if all conditions satisfy:

- They belong to different transactions
- They operate on the same data item
- At Least one of them is a write operation

ii. View Serializable:

A Schedule is called view serializable if it is view equal to a serial schedule (no overlapping transactions). A conflict schedule is a view serializable but if the serializability contains blind writes, then the view serializable does not conflict serializable.

b. Non-Serializable:

The non-serializable schedule is divided into two types, Recoverable and Non-recoverable Schedule.

i. *Recoverable Schedule:*

Schedules in which transactions commit only after all transactions whose changes they read commit are called recoverable schedules. if some transaction T_j is reading value updated or written by some other transaction T_i, then the commit of T_j must occur after the commit of T_i.

Example – Consider the following schedule involving two transactions T_1 and T_2.

This is a recoverable schedule since T_1 commits before T_2, that makes the value read by T_2 correct.

T_1	T_2
R(A)	
W(A)	
	W(A)
	R(A)
commit	
	commit

iii. *Non-Recoverable Schedule:*

Example: Consider the following schedule involving two transactions T_2 read the value of A written by T_1, and committed. T_1 later aborted, therefore the value read by T_2 is wrong, but since T_2 committed, this schedule is **non-recoverable**.

T_1	T_2
R(A)	
W(A)	
	W(A)
	R(A)
	commit
abort	

5.1.4 Concurrent Execution of Transactions:

An important task of a DBMS is to schedule concurrent accesses to data so that each user can safely ignore the fact that others are accessing the data concurrently. The importance of this ta.sk cannot be underestimated because a database is typically shared by a large number of users, who submit their requests to the DBMS independently and simply cannot be expected to deal with arbitrary changes being made concurrently by other users. A DBMS allows users to think of their programs &'3 if they were executing in isolation, one after the other in some order chosen by the DBJ\;:IS. For example, if a program that deposits cash into an account is submitted to the DBMS at the same time as another program that debits money from the same account, either of these programs could be run first by the DBMS, but their steps will not be interleaved in such a way that they interfere with each other.

A locking protocol is a set of rules to be followed by each transaction (and enforced by the DBMS) to ensure that, even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions in some serial order. A lock is a mechanism used to control access to database objects. Two kinds of locks are commonly supported by a DBMS: shared locks on an object can be held by two different transactions at the same time, but an exclusive lock on an object ensures that no other transactions hold any lock on this object.

Suppose that the following locking protocol is followed: Every transaction begins by obtaining a shared lock on each data object that it needs to read and an exclusive lock on each data object that it needs to rnod~fy, then releases all its locks after completing all actions. Consider two transactions T1 and T2 such that T1 wants to modify a data object and T2 wants to read the same object.

Intuitively, if T1's request for an exclusive lock on the object is granted first, T2 cannot proceed until T1 relea..':les this lock, because T2's request for a shared lock will not be granted by the DBMS until then. Thus, all of T1's actions will be completed before any of T2's actions are initiated.

5.1.5 Concurrent Execution of Database:

Multiple transactions are allowed to run concurrently in the system. Concurrent execution of database is meant by execution of database in parallel. I.e. each transaction must behave in isolation. This means that the concurrent execution does not result an inconsistent state. Ensuring consistency in spite of concurrent execution of transactions requires is very complex

Advantages are:

- increased processor and disk utilization, leading to better transaction throughput
- 4 E.g. 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.

Timestamp Protocol: Timestamp is a unique identifier to identify a transaction. Timestamp can be considered to be as the transaction start time. It determines the concurrent execution such that the timestamp determines the serializability order. Time stamp holds two timestamp values:

- W-timestamp() is the largest time-stamp of any transaction that executed write() successfully.
- R-timestamp () is the largest time-stamp of any transaction that executed read() successfully.

Suppose that transaction Ti issues write (Q).

If TS (Ti) < R-timestamp (Q), then the value of Q that Ti is producing was needed previously, and the system assumed that that value would never be produced. Hence, the write operation is rejected, and Ti is rolled back.

If TS (Ti) < W-timestamp (Q), then Ti is attempting to write an obsolete value of Q. Hence, this write operation is rejected, and Ti is rolled back.

Otherwise, the write operation is executed, and W-timestamp(Q) is set to TS(Ti).The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form. Timestamp protocol ensures freedom from deadlock as no transaction ever waits.

5.1.6 Concurrency Control:

Concurrency control is the procedure in DBMS for managing simultaneous operations without conflicting with each another. Concurrent access is quite easy if all users are just reading data.

Concurrency control is used to address such conflicts which mostly occur with a multi-user system. It helps you to make sure that database transactions are performed concurrently without violating the data integrity of respective databases.

Why use Concurrency method?

Reasons for using Concurrency control method is DBMS:

To apply Isolation through mutual exclusion between conflicting transactions

To resolve read-write and write-write conflict issues

To preserve database consistency through constantly preserving execution obstructions

The system needs to control the interaction among the concurrent transactions. This control is achieved using concurrent-control schemes.

Concurrency control helps to ensure serializability

5.1.6.1 Concurrency Control Protocols:

Different concurrency control protocols offer different benefits between the amount of concurrency they allow and the amount of overhead that they impose.

- Lock-Based Protocols
- Two Phase
- Timestamp-Based Protocols
- Validation-Based Protocols

Lock-based Protocols:

A lock is a data variable which is associated with a data item. This lock signifies that operations that can be performed on the data item. Locks help synchronize access to the database items by concurrent transactions.

All lock requests are made to the concurrency-control manager. Transactions proceed only once the lock request is granted.

Binary Locks: A Binary lock on a data item can either locked or unlocked states.

Shared/exclusive: This type of locking mechanism separates the locks based on their uses. If a lock is acquired on a data item to perform a write operation, it is called an exclusive lock.

1. Shared Lock (S):

A shared lock is also called a Read-only lock. With the shared lock, the data item can be shared between transactions. This is because we will never have permission to update data on the data item.

For example, consider a case where two transactions are reading the account balance of a person. The database will let them read by placing a shared lock. However, if another transaction wants to update that account's balance, shared lock prevent it until the reading process is over.

2. Exclusive Lock (X):

With the Exclusive Lock, a data item can be read as well as written. This is exclusive and can't be held concurrently on the same data item. X-lock is requested using lock-x instruction. Transactions may unlock the data item after finishing the 'write' operation.

For example, when a transaction needs to update the account balance of a person. You can allows this transaction by placing X lock on it. Therefore, when the second transaction wants to read or write, exclusive lock prevent this operation.

Two Phase Locking (2PL) Protocol:

Two-Phase locking protocol which is also known as a 2PL protocol. It is also called P2L. In this type of locking protocol, the transaction should acquire a lock after it releases one of its locks.

This locking protocol divides the execution phase of a transaction into three different parts.

- In the first phase, when the transaction begins to execute, it requires permission for the locks it needs.
- The second part is where the transaction obtains all the locks. When a transaction releases its first lock, the third phase starts.
- In this third phase, the transaction cannot demand any new locks. Instead, it only releases the acquired locks.

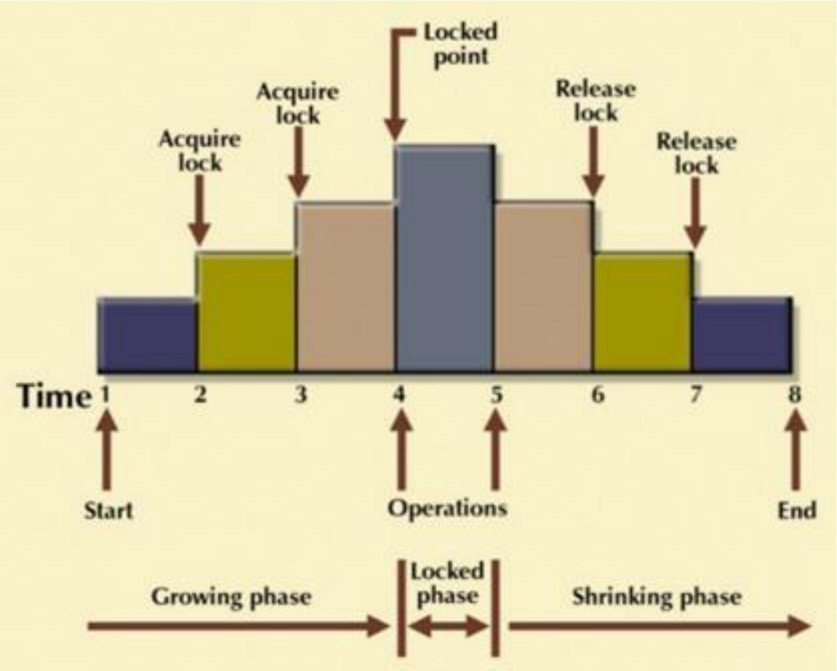


Figure 5.2

The Two-Phase Locking protocol allows each transaction to make a lock or unlock request in two steps:

- **Growing Phase**: In this phase transaction may obtain locks but may not release any locks.
- **Shrinking Phase**: In this phase, a transaction may release locks but not obtain any new lock

It is true that the 2PL protocol offers serializability. However, it does not ensure that deadlocks do not happen.

In the above-given diagram, you can see that local and global deadlock detectors are searching for deadlocks and solve them with resuming transactions to their initial states.

Strict Two-Phase Locking Method:

Strict-Two phase locking system is almost similar to 2PL. The only difference is that Strict-2PL never releases a lock after using it. It holds all the locks until the commit point and releases all the locks at one go when the process is over.

Centralized 2PL

In Centralized 2 PL, a single site is responsible for lock management process. It has only one lock manager for the entire DBMS.

Primary copy 2PL

Primary copy 2PL mechanism, many lock managers are distributed to different sites. After that, a particular lock manager is responsible for managing the lock for a set of data items. When the primary copy has been updated, the change is propagated to the slaves.

Distributed 2PL

In this kind of two-phase locking mechanism, Lock managers are distributed to all sites. They are responsible for managing locks for data at that site. If no data is replicated, it is equivalent to primary copy 2PL. Communication costs of Distributed 2PL are quite higher than primary copy 2PL

Timestamp-based Protocols:

The timestamp-based algorithm uses a timestamp to serialize the execution of concurrent transactions. This protocol ensures that every conflicting read and write operations are executed in timestamp order. The protocol uses the **System Time or Logical Count as** a Timestamp.

The older transaction is always given priority in this method. It uses system time to determine the time stamp of the transaction. This is the most commonly used concurrency protocol.

Lock-based protocols help you to manage the order between the conflicting transactions when they will execute. Timestamp-based protocols manage conflicts as soon as an operation is created.

Example:

Suppose there are there transactions T1, T2, and T3.

T1 has entered the system at time 0010

T2 has entered the system at 0020

T3 has entered the system at 0030

Priority will be given to transaction T1, then transaction T2 and lastly Transaction T3.

5.2 Concurrency Control:

5.2.1 Serializability:

- Serializability is a concurrency scheme where the concurrent transaction is equivalent to one that executes the transactions serially.
- A schedule is a list of transactions.
- Serial schedule defines each transaction is executed consecutively without any interference from other transactions.

The main objective of serializability is to find non-serial schedules that allow transactions to execute concurrently without interference and produce a database state that could be produced by a serial execution.

5.2.1.1. Conflict Serializability:

- Conflict serializability defines two instructions of two different transactions accessing the same data item to perform a read/write operation.
- It deals with detecting the instructions that are conflicting in any way and specifying the order in which the instructions should execute in case there is any conflict.
- A conflict serializability arises when one of the instruction is a write operation.

The following rules are important in Conflict Serializability:

1. If two transactions are both read operation, then they are not in conflict.
2. If one transaction wants to perform a read operation and other transaction wants to perform a write operation, then they are in conflict and cannot be swapped.
3. If both the transactions are for write operation, then they are in conflict, but can be allowed to take place in any order, because the transactions do not read the value updated by each other.

5.2.1.2. View Serializability:

- View serializability is the another type of serializability.
- It can be derived by creating another schedule out of an existing schedule and involves the same set of transactions.

Example : Let us assume two transactions T1 and T2 that are being serialized to create two different schedules SH1 and SH2, where T1 and T2 want to access the same data item. Now there can be three scenarios

1. If in SH1, T1 reads the initial value of data item, then in SH2 , T1 should read the initial value of that same data item.

2. If in SH2, T1 writes a value in the data item which is read by T2, then in SH2, T1 should write the value in the data item before T2 reads it.

3. If in SH1, T1 performs the final write operation on that data item, then in SH2, T1 should perform the final write operation on that data item.

If a concurrent schedule is view equivalent to a serial schedule of same transaction then it is said to be **View serializable.**

5.2.2 Recoverability:

a transaction may not execute completely due to hardware failure, system crash or software issues. In that case, we have to roll back the failed transaction. But some other transaction may also have used values produced by the failed transaction. So we have to roll back those transactions as well.

Recoverable Schedules:

Schedules in which transactions commit only after all transactions whose changes they read commit are called recoverable schedules. In other words, if some transaction T_j is reading value updated or written by some other transaction T_i, then the commit of T_j must occur after the commit of T_i.

Example 1:

S1: R1(x), W1(x), R2(x), R1(y), R2(y),
W2(x), W1(y), C1, C2;

Given schedule follows order of **Ti->Tj => C1->C2**. Transaction T1 is executed before T2 hence there is no chances of conflict occur. R1(x) appears before W1(x) and transaction T1 is committed before T2 i.e. completion of first transaction performed first update on data item x, hence given schedule is recoverable.

Example 2:

	T_1	T_2
Consider the following schedule involving two transactions T_1 and T_2. This is a recoverable schedule since T_1 commits before T_2, that makes the value read by T_2 correct.	R(A)	
	W(A)	
		W(A)
		R(A)
	commit	
		commit

5.2.3 Introduction to Lock:

Transaction processing systems usually allow multiple transactions to run concurrently. By allowing multiple transactions to run concurrently will improve the performance of the system in terms of increased throughout or improved response time, but this allows causes several complications with consistency of the data.

A lock is a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied to it. Generally, there is one lock for each data item in the database. Locks are used as a means of synchronizing the access by concurrent transactions to the database item.

Types of Locks:

Several types of locks are used in concurrency control.

Binary Locks:

A binary lock can have two states or values: locked and unlocked.

A binary lock can have two states or values: locked and unlocked.

A distinct lock is associated with each database item *A.* If the value of the lock on *A* is 1, item *A* cannot be accessed by a database operation that requests the item. If the value of the lock on *A* is 0 then item can be accessed when requested. We refer to the current value of the lock associated with item *A* as *LOCK (A).* There are two operations, lock item and unlock item are used with binary locking A transaction requests access to an item *A* by first issuing a lock *item (A)* operation. If LOCK (A) = 1, the transaction is forced to wait. If LOCK (A) = 0 it is set to 1 (the transaction locks the item) and the transaction is allowed to access item *A.* When the transaction is through using the item, it issues an unlock *item (A)* operation, which sets *LOCK (A)* to 0 (unlocks the item) so that *A* may be accessed by other transactions. Hence binary lock enforces mutual exclusiol1 on the data item.

Share/Exclusive (for Read/Write) Locks

We should allow several transactions to access the same item *A* if they all access *A'* for reading purposes only. However, if a transaction is to write an item A, it must have exclusive access to *A.* For this purpose, a different type of lock called a multiple-mode lock is used. In this scheme there are shared/exclusive or read/write locks are used.

Shared lock:

- It is also known as a Read-only lock. In a shared lock, the data item can only read by the transaction.
- It can be shared between the transactions because when the transaction holds a lock, then it can't update the data on the data item.

Exclusive lock:

- In the exclusive lock, the data item can be both reads as well as written by the transaction.
- This lock is exclusive, and in this lock, multiple transactions do not modify the same data simultaneously.

5.2.4 Dealing with Deadlock:

In a database, a deadlock is an unwanted situation in which two or more transactions are waiting indefinitely for one another to give up locks.

Example – let us understand the concept of Deadlock with an example : Suppose, Transaction T1 holds a lock on some rows in the Students table and **needs to update** some rows in the Grades table. Simultaneously, Transaction **T2 holds** locks on those very rows (Which T1 needs to update) in the Grades table **but needs** to update the rows in the Student table **held by Transaction T1**.

Now, the main problem arises. Transaction T1 will wait for transaction T2 to give up lock, and similarly transaction T2 will wait for transaction T1 to give up lock. As a consequence, All activity comes to a halt and remains at a standstill forever unless the DBMS detects the deadlock and aborts one of the transactions.

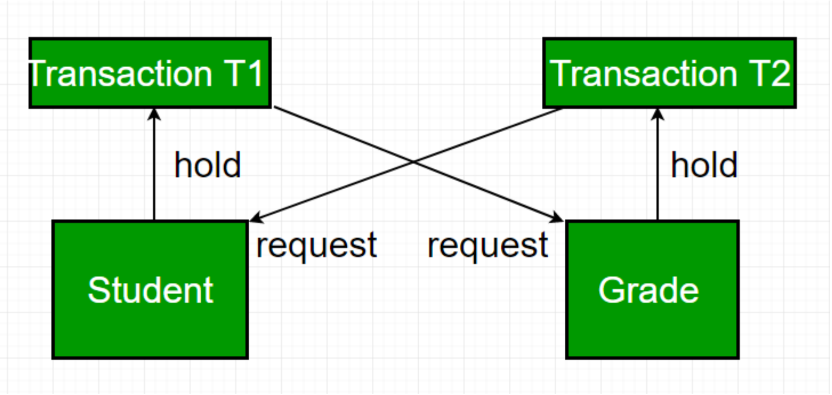


Figure 5.3

Deadlock prevention –

For large database, deadlock prevention method is suitable. A deadlock can be prevented if the resources are allocated in such a way that deadlock never occur. The DBMS analyzes the operations whether they can create deadlock situation or not, If they do, that transaction is never allowed to be executed.

Deadlock prevention mechanism proposes two schemes :

i.Wait-Die Scheme:

In this scheme, If a transaction request for a resource that is locked by other transaction, then the DBMS simply checks the timestamp of both transactions and allows the older transaction to wait until the resource is available for execution.

Suppose, there are two transactions T1 and T2 and Let timestamp of any transaction T be TS (T). Now, If there is a lock on T2 by some other transaction and T1 is requesting for resources held by T2, then DBMS performs following actions:

Checks if TS (T1) < TS (T2) – if T1 is the older transaction and T2 has held some resource, then it allows T1 to wait until resource is available for execution. That means if a younger transaction has locked some resource and older transaction is waiting for it, then older transaction is allowed wait for it till it is available. If T1 is older transaction and has held some resource with it and if T2 is waiting for it, then T2 is killed and restarted latter with random delay but with the same timestamp. i.e. if the older transaction has held some resource and younger transaction waits for the resource, then younger transaction is killed and restarted with very minute delay with same timestamp. This scheme allows the older transaction to wait but kills the younger one.

Wound Wait Scheme –

In this scheme, if an older transaction requests for a resource held by younger transaction, then older transaction forces younger transaction to kill the transaction and release the resource. The younger transaction is restarted with minute delay but with same timestamp. If the younger transaction is requesting a resource which is held by older one, then younger transaction is asked to wait till older releases it.

Deadlock Detection –

When a transaction waits indefinately to obtain a lock, The database managememt system should detect whether the transaction is involved in a deadlock or not.

Wait-for-graph is one of the methods for detecting the deadlock situation. This method is suitable for smaller database. In this method a graph is drawn based on the transaction and their lock on the resource. If the graph created has a closed loop or a cycle, then there is a deadlock.

For the above mentioned scenario the Wait-For graph is drawn below

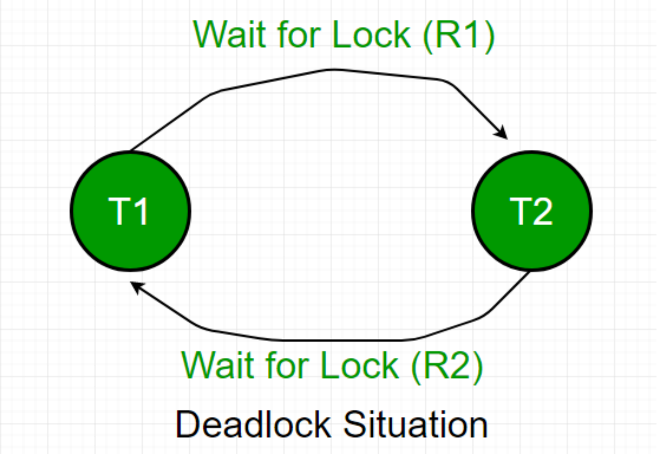


Figure 5.4

Deadlock Avoidance –

When a database is stuck in a deadlock, It is always better to avoid the deadlock rather than restarting or aborting the database. Deadlock avoidance method is suitable for smaller database whereas deadlock prevention method is suitable for larger database. One method of avoiding deadlock is using application consistent logic. In the above given example, Transactions that access Students and Grades should always access the tables in the same order. In this way, in the scenario described above, Transaction T1 simply waits for transaction T2 to release the lock on Grades before it begins. When transaction T2 releases the lock, Transaction T1 can proceed freely.

Another method for avoiding deadlock is to apply both row level locking mechanism and

READ COMMITTED isolation level. However, It does not guarantee to remove deadlocks completely.

5.3 Crash Recovery:

5.3.1 Introduction to ARIES:

Algorithm for Recovery and Isolation Exploiting Semantics (ARIES) is based on the Write Ahead Log (WAL) protocol. Every update operation writes a log record which is one of the following :

1. **Undo-only log record:**
   Only the before image is logged. Thus, an undo operation can be done to retrieve the old data.
2. **Redo-only log record:**
   Only the after image is logged. Thus, a redo operation can be attempted.
3. **Undo-redo log record:**
   Both before images and after images are logged.

The recovery process actually consists of 3 phases:

1. **Analysis:**
   The recovery subsystem determines the earliest log record from which the next pass must start. It also scans the log forward from the checkpoint record to construct a snapshot of what the system looked like at the instant of the crash.
2. **Redo:**
   Starting at the earliest LSN, the log is read forward and each update redone.
3. **Undo:**
   The log is scanned backward and updates corresponding to loser transactions are undone.

5.3.2 Failure Classification:

To see where the problem has occurred we generalize the failure into various categories, as follows:

TRANSACTION FAILURE :

When a transaction is failed to execute or it reaches a point after which it cannot be completed successfully it has to abort. This is called transaction failure. Where only few transaction or process are hurt.

Reason for transaction failure could be:

- **Logical errors:** where a transaction cannot complete because of it has some code error or any internal error condition
- **System errors:** where the database system itself terminates an active transaction because DBMS is not able to execute it or it has to stop because of some system condition. For example, in case of deadlock or resource unavailability systems aborts an active transaction.

5.3.3 SYSTEM CRASH:

There are problems, which are external to the system, which may cause the system to stop abruptly and cause the system to crash. For example interruption in power supply, failure of underlying hardware or software failure.

- Examples may include operating system errors.

DISK FAILURE:

In early days of technology evolution, it was a common problem where hard disk drives or storage drives used to fail frequently.

- Disk failures include formation of bad sectors, unreachability to the disk, disk head crash or any other failure, which destroys all or part of disk storage

Storage Structure:

We have already described storage system here. In brief, the storage structure can be divided in various categories:

- **Volatile storage:** As name suggests, this storage does not survive system crashes and mostly placed very closed to CPU by embedding them onto the chipset itself for examples: main memory, cache memory. They are fast but can store a small amount of information.

- **Nonvolatile storage:** These memories are made to survive system crashes. They are huge in data storage capacity but slower in accessibility. Examples may include, hard disks, magnetic tapes, flash memory, non-volatile (battery backed up) RAM.

5.3.4 Recovery and Atomicity:

When a system crashes, it many have several transactions being executed and various files opened for them to modifying data items. As we know that transactions are made of various operations, which are atomic in nature. But according to ACID properties of DBMS, atomicity of transactions as a whole must be maintained that is, either all operations are executed or none.

When DBMS recovers from a crash it should maintain the following:

- It should check the states of all transactions, which were being executed.
- A transaction may be in the middle of some operation; DBMS must ensure the atomicity of transaction in this case.
- It should check whether the transaction can be completed now or needs to be rolled back.
- No transactions would be allowed to left DBMS in inconsistent state.

There are two types of techniques, which can help DBMS in recovering as well as maintaining the atomicity of transaction:

- Maintaining the logs of each transaction, and writing them onto some stable storage before actually modifying the database.
- Maintaining shadow paging, where are the changes are done on a volatile memory and later the actual database is updated.

5.3.4.1 Log-Based Recovery

- Log is a sequence of records, which maintains the records of actions performed by a transaction. It is important that the logs are written prior to actual modification and stored on a stable storage media, which is failsafe.
- Log based recovery works as follows:
- The log file is kept on stable storage media
- When a transaction enters the system and starts execution, it writes a log about it.

<Tn, Start>

- When the transaction modifies an item X, it write logs as follows:

<Tn, X, V1, V2>

It reads Tn has changed the value of X, from V1 to V2.

- When transaction finishes, it logs:

<Tn, commit>

- Database can be modified using two approaches:

1. **Deferred database modification:** All logs are written on to the stable storage and database is updated when transaction commits.

2. **Immediate database modification:** Each log follows an actual database modification. That is, database is modified immediately after every operation.

5.3.4.2 Recovery with concurrent transactions:

When more than one transactions are being executed in parallel, the logs are interleaved. At the time of recovery it would become hard for recovery system to backtrack all logs, and then start recovering. To ease this situation most modern DBMS use the concept of 'checkpoints'.

5.3.5 CHECKPOINT:

Keeping and maintaining logs in real time and in real environment may fill out all the memory space available in the system. At time passes log file may be too big to be handled at all. Checkpoint is a mechanism where all the previous logs are removed from the system and stored permanently in storage disk. Checkpoint declares a point before which the DBMS was in consistent state and all the transactions were committed.

RECOVERY

When system with concurrent transaction crashes and recovers, it does behave in the following manner:

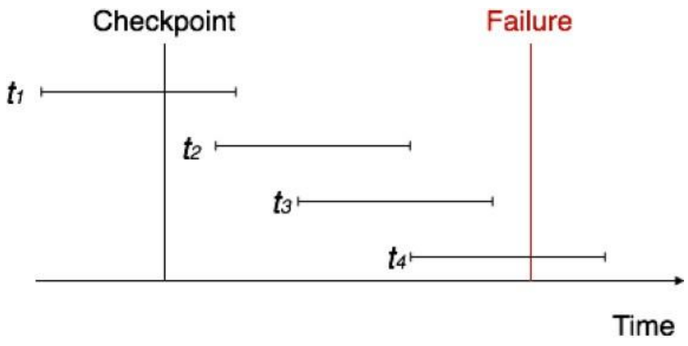


Figure 5.5

- The recovery system reads the logs backwards from the end to the last Checkpoint.
- It maintains two lists, undo-list and redo-list.
- If the recovery system sees a log with <Tn, Start> and <Tn, Commit> or just

<Tn, Commit>, it puts the transaction in redo-list.

- If the recovery system sees a log with <Tn, Start> but no commit or abort log found, it puts the transaction in undo-list.

All transactions in undo-list are then undone and their logs are removed. All transaction in redo-list, their previous logs are removed and then redone again and log saved.

5.3.6 Aries Algorithm:

ARIES (Algorithm for Recovery and Isolation Exploiting Semantics) recovery is based on the Write Ahead Logging (WAL) protocol. Every update operation writes a log record which is one of

1. An undo-only log record: Only the before image is logged. Thus, an undo operation can be done to retrieve the old data.
2. A redo-only log record: Only the after image is logged. Thus, a redo operation can be attempted.
3. An undo-redo log record. Both before image and after images are logged.

Every log record is assigned a unique and monotonically increasing log sequence number (LSN). Every data page has a page LSN field that is set to the LSN of the log record corresponding to the last update on the page. WAL requires that the log record corresponding to an update make it to stable storage before the data page corresponding to that update is written to disk. For performance reasons, each log write is not immediately forced to disk. A log tail is maintained in main memory to buffer log writes. The log tail is flushed to disk when it gets full. A transaction cannot be declared committed until the commit log record makes it to disk.

Once in a while the recovery subsystem writes a checkpoint record to the log. The checkpoint record contains the transaction table (which gives the list of active transactions) and the dirty page table (the list of data pages in the buffer pool that have not yet made it to disk). A master log record is maintained separately, in stable storage, to store the LSN of the latest checkpoint record that made it to disk. On restart, the recovery subsystem reads the master log record to find the checkpoint's LSN, reads the checkpoint record, and starts recovery from there on.

The actual recovery process consists of three passes:

1. Analysis: The recovery subsystem determines the earliest log record from which the next pass must start. It also scans the log forward from the checkpoint record to construct a snapshot of what the system looked like at the instant of the crash.
2. Redo: Starting at the earliest LSN determined in pass (1) above, the log is read forward and each update redone.
3. Undo: The log is scanned backward and updates corresponding to loser

transactions are undone.

UNIT – VI

6.1 Overview of Indexing:

We know that data is stored in the form of records. Every record has a key field, which helps it to be recognized uniquely.

Indexing is a way to optimize the performance of a database by minimizing the number of disk accesses required when a query is processed. It is a data structure technique which is used to quickly locate and access the data in a database.

Indexes are created using a few database columns.

- The first column is the **Search key** that contains a copy of the primary key or candidate key of the table. These values are stored in sorted order so that the corresponding data can be accessed quickly.

  Note: The data may or may not be stored in sorted order.
- The second column is the **Data Reference** or **Pointer** which contains a set of pointers holding the address of the disk block where that particular key value can be found.

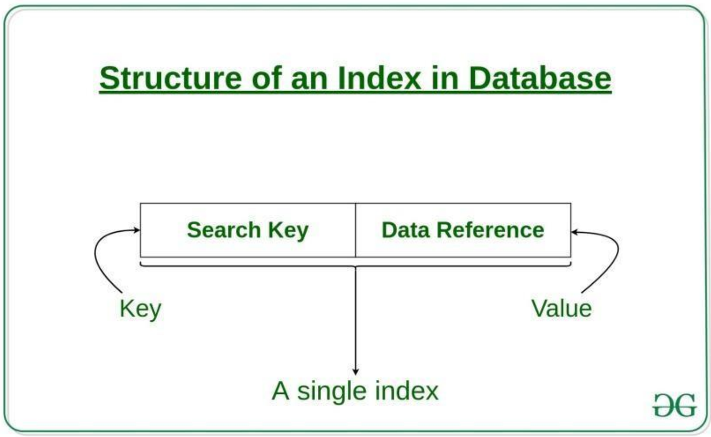


Figure 6.1

The indexing has various attributes:

- **Access Types**: This refers to the type of access such as value based search, range access, etc.
- **Access Time**: It refers to the time needed to find particular data element or set of elements.
- **Insertion Time**: It refers to the time taken to find the appropriate space and insert a new data.
- **Deletion Time**: Time taken to find an item and delete it as well as update the index structure.
- **Space Overhead**: It refers to the additional space required by the index.

In general, there are two types of file organization mechanism which are followed by the indexing methods to store the data:

6.1.1 Sequential File Organization or Ordered Index File: In this, the indices are based on a sorted ordering of the values. These are generally fast and a more traditional type of storing mechanism. These Ordered or Sequential file organization might store the data in a dense or sparse format:

- **Dense Index:**
  - For every search key value in the data file, there is an index record.
  - This record contains the search key and also a reference to the first data record with that search key value.

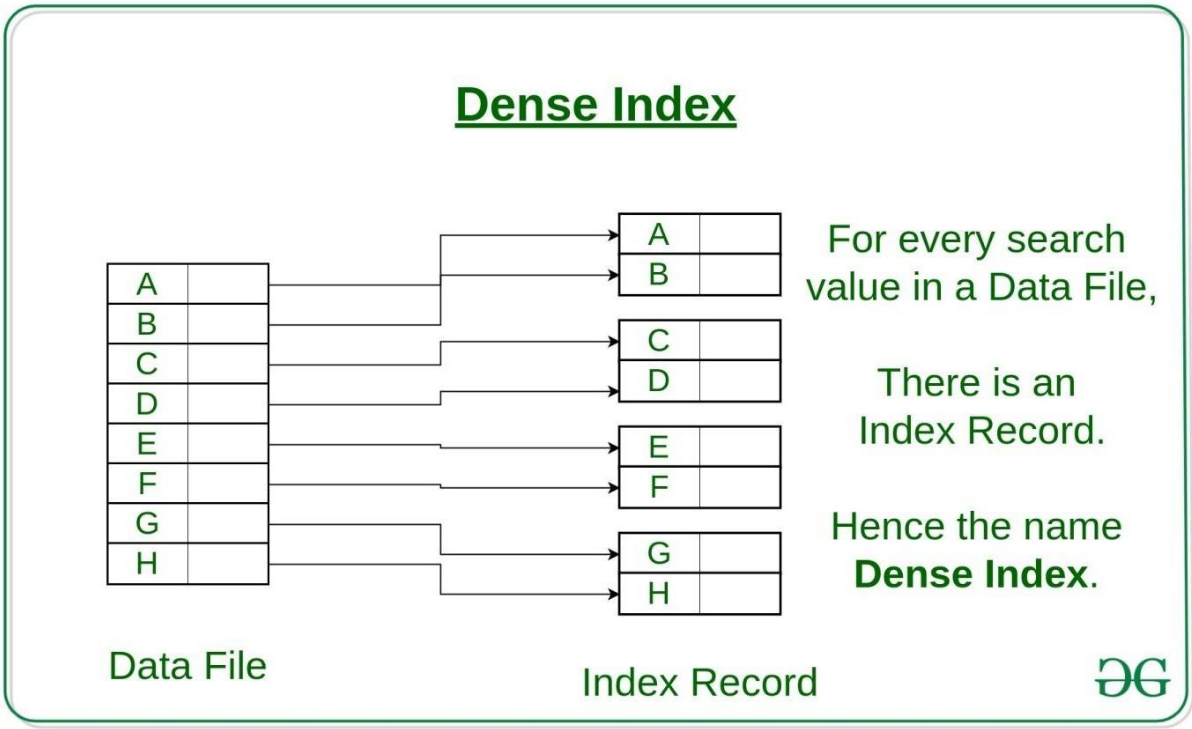

Dense Index
A
B
C
D
E
F
G
H
For every search value in a Data File,
There is an Index Record.
Hence the name Dense Index.
Data File
Index Record

Figure 6.2

Sparse Index:

- The index record appears only for a few items in the data file. Each item points to a block as shown.
- To locate a record, we find the index record with the largest search key value less than or equal to the search key value we are looking for.
- We start at that record pointed to by the index record, and proceed along with the pointers in the file (that is, sequentially) until we find the desired record.

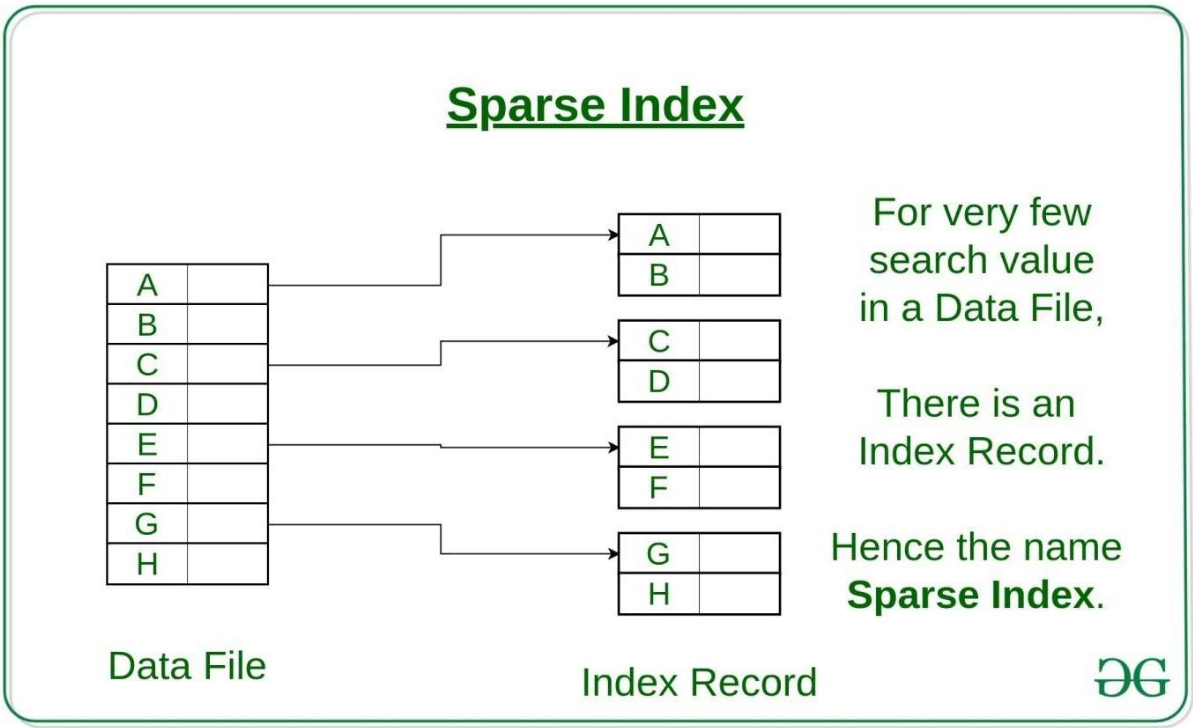

Sparse Index
A
B
C
D
E
F
G
H
Data File
A
B
C
D
E
F
G
H
Index Record
For very few search value in a Data File,
There is an Index Record.
Hence the name Sparse Index.

Figure 6.3

6.1.2 Hash File organization: Indices are based on the values being distributed uniformly across a range of buckets. The buckets to which a value is assigned is determined by a function called a hash function.

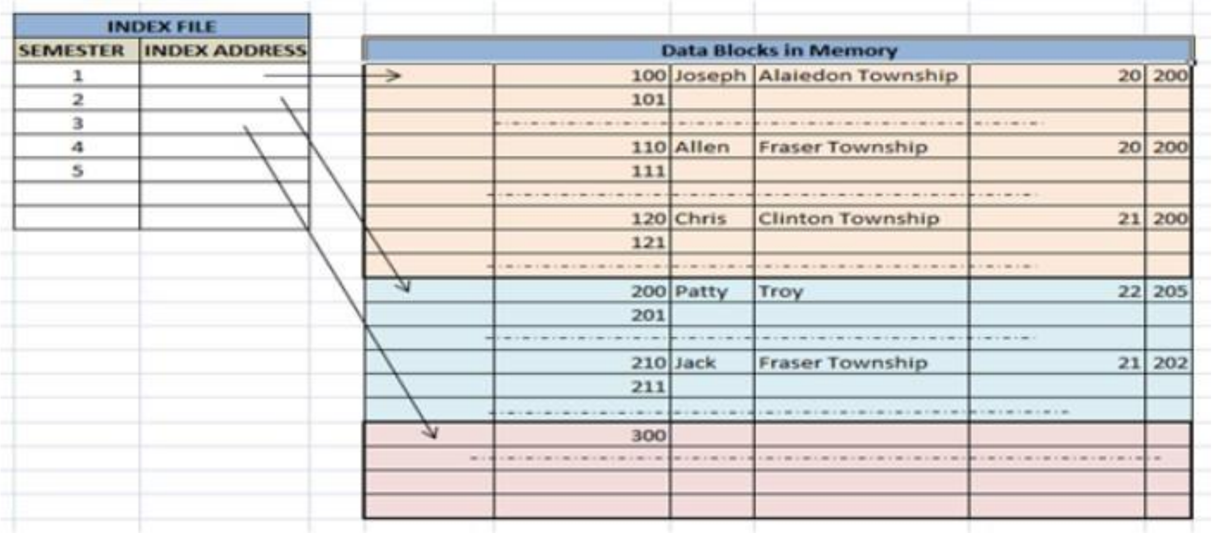
There are primarily three methods of indexing:

- Clustered Indexing
- Non-Clustered or Secondary Indexing
- Multilevel Indexing

6.1.2.1 Clustered Indexing:

When more than two records are stored in the same file these types of storing known as cluster indexing. By using the cluster indexing we can reduce the cost of searching reason being multiple records related to the same thing are stored at one place and it also gives the frequent joing of more than two tables(records).

Clustering index is defined on an ordered data file. The data file is ordered on a non-key field. In some cases, the index is created on non-primary key columns which may not be unique for each record. In such cases, in order to identify the records faster, we will group two or more columns together to get the unique values and create index out of them. This method is known as the clustering index. Basically, records with similar characteristics are grouped together and indexes are created for these groups. For example, students studying in each semester are grouped together. i.e. 1st Semester students, 2nd semester students, 3rd semester students etc are grouped.



INDEX FILE	
SEMESTER	INDEX ADDRESS
1	
2	
3	
4	
5	

Data Blocks in Memory					
	100	Joseph	Alaiedon Township	20	200
	101				
	110	Allen	Fraser Township	20	200
	111				
	120	Chris	Clinton Township	21	200
	121				
	200	Patty	Troy	22	205
	201				
	210	Jack	Fraser Township	21	202
	211				
	300				

Figure 6.4

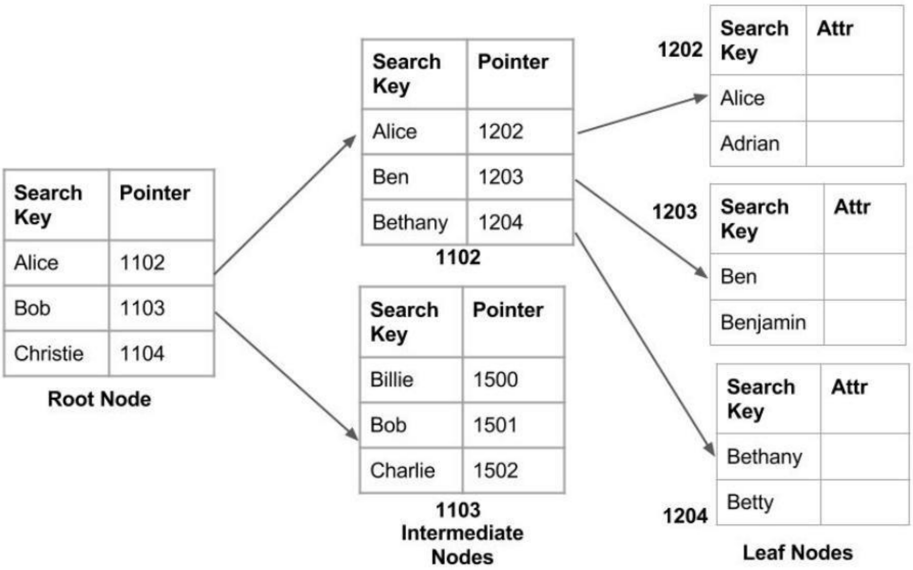
Clustered index sorted according to first name (Search key)

Primary Indexing:

This is a type of Clustered Indexing wherein the data is sorted according to the search key and the primary key of the database table is used to create the index. It is a default format of indexing where it induces sequential file organization. As primary keys are unique and are stored in a sorted manner, the performance of the searching operation is quite efficient.

6.1.2.2 Non-clustered or Secondary Indexing:

A non clustered index just tells us where the data lies, i.e. it gives us a list of virtual pointers or references to the location where the data is actually stored. Data is not physically stored in the order of the index. Instead, data is present in leaf nodes. For eg. the contents page of a book. Each entry gives us the page number or location of the information stored. The actual data here(information on each page of the book) is not organized but we have an ordered reference(contents page) to where the data points actually lie. We can have only dense ordering in the non-clustered index as sparse ordering is not possible because data is not physically organized accordingly. It requires more time as compared to the clustered index because some amount of extra work is done in order to extract the data by further following the pointer. In the case of a clustered index, data is directly present in front of the index.



Search Key	Pointer
Alice	1102
Bob	1103
Christie	1104

Root Node

Search Key	Pointer
Alice	1202
Ben	1203
Bethany	1204

1102

Search Key	Pointer
Billie	1500
Bob	1501
Charlie	1502

1103 Intermediate Nodes

1202

Search Key	Attr
Alice	
Adrian	

1203

Search Key	Attr
Ben	
Benjamin	

Search Key	Attr
Bethany	
Betty	

1204

Leaf Nodes

Non clustered index

Figure 6.5

iii.Multilevel Indexing:

With the growth of the size of the database, indices also grow. As the index is stored in the main memory, a single-level index might become too large a size to store with multiple disk accesses. The multilevel indexing segregates the main block into various smaller blocks so that the same can stored in a single block. The outer blocks are divided into inner blocks which in turn are pointed to the data blocks. This can be easily stored in the main memory with fewer overheads.

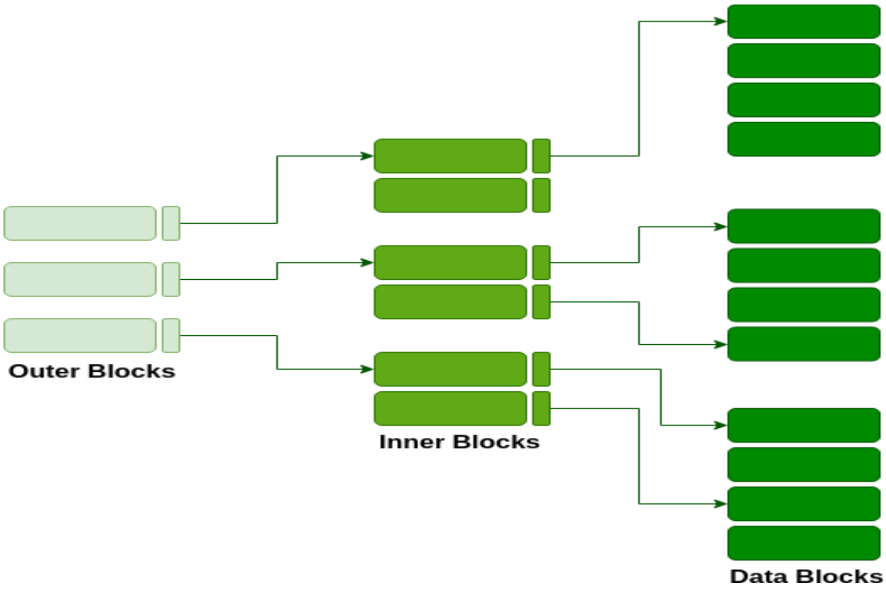


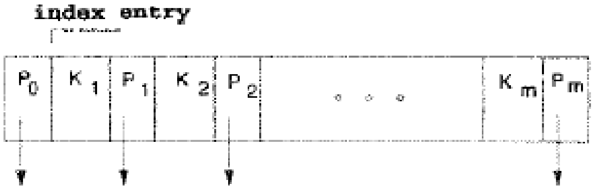
Figure 6.6

6.2. Tree Structured Indexing:

6.2.1 Intuition for Tree Indexes:

Consider a file of Students recorcls sorted by *gpa.* To answer a range selection such as "Find all students with a gpa higher than 3.0," we must identify the first such student by doing a binary search of the file and then scan the file from that point on. If the file is large, the initial binary search can be quite expensive, since cost is proportional to the number of pages fetched.

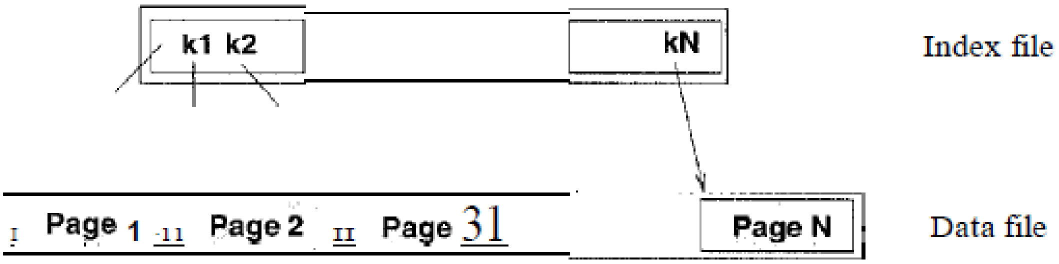
One idea is to create a second file with One record per page in the original (data) file, of the form (first key on page, pointer to page), again sorted by the key attribute (which is gpa in our example). The format of a page in the second index file is illustrated in Figure.



Format of an Index Page

Figure 6.7

We refer to pairs of the form (key, pointer) as indx entries or just entries when the ontext is clear. Note that each index page contains One pointer more than the number of keys --- each key serves as a separator- for the contents of the pages pointed to by the pointers to its left and right.The simple index file data structure is illustrated in Figure



One-Level Index Structure

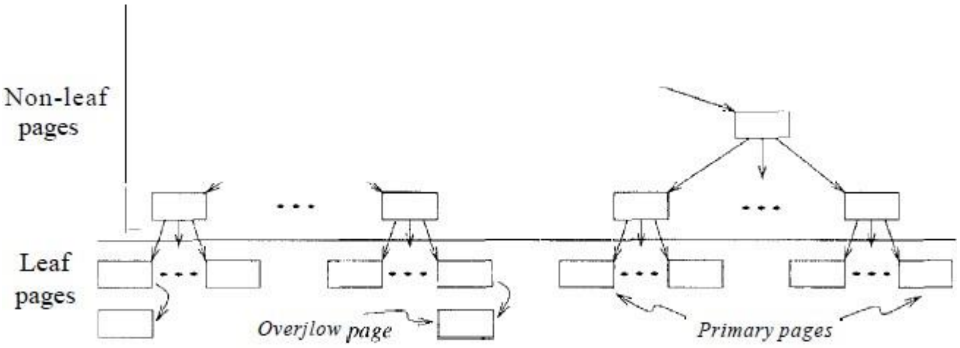
Figure 6.8

We can do a binary search of the index file to identify the page containing the first key (gpa.) value that satisfies the range selection (in our example, the first student with gpa over 3.0) and follow the pointer to the page containing the first data. record with that key value. We can then scan the data file sequentially from that point on to retrieve other qualifying records. This example uses the index to find the first data page containing a Students record with gpa greater than 3.0, and the data file is scanned from that point on to retrieve other such Students records.

6.2.2 Indexed Sequential Access Methods (ISAM):

ISAM method is an advanced sequential file organization. In this method, records are stored in the file using the primary key. An index value is generated for each primary key and mapped with the record. This index contains the address of the record in the file.

The data entries of the ISAM index are in the leaf pages of the tree and additional overflow pages chained to some leaf page. Database systems carefully organize the layout of pages so that page boundaries correspond closely to the physical characteristics of the underlying storage device. The ISAM structure is completely static (except for the overflow pages, of which it is hoped, there will be few) and facilitates such low-level optimizations. The ISAM data structure is illustrated in Figure



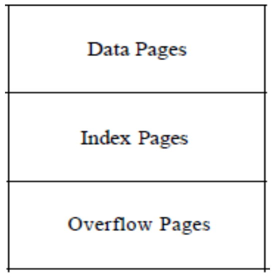
ISAM Index Structure

Figure 6.9

Each tree node is a disk page, and all the data resides in the leaf pages. This orresponds to an index that uses Alternative .

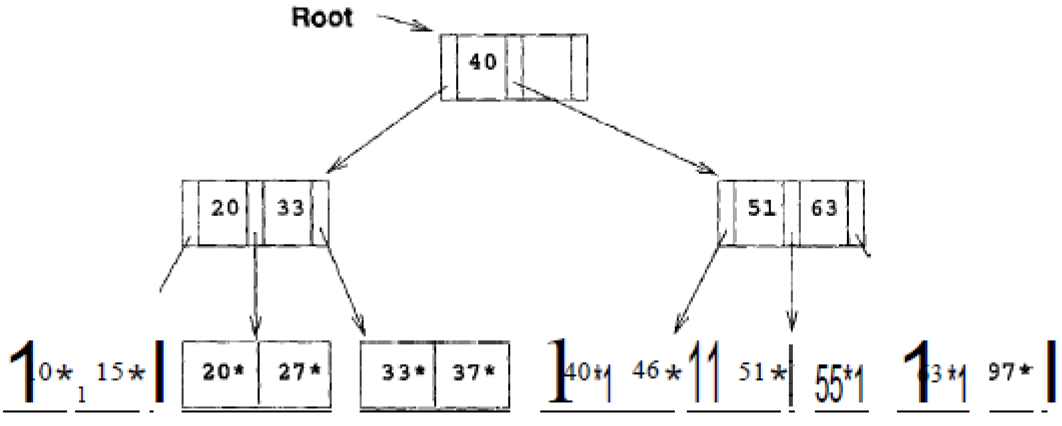
(1) for data entries, in terms of the alternatives we can create an index with Alternative

(2) by storing t.he data records in a separate file and storing (key, rid) pairs in the leaf pages of the ISAM index.

If there are several inserts to the file subsequently, so that more entries are inserted into a leaf than will fit onto a single page, additional pages are needed because the index structure is static. These additional pages are allocated from an overflow area. The allocation of pages is illustrated	 Page Allocation in ISAM

The basic operations of insertion, deletion, and search are all quite straightforward. For an equality selection search, we start at the root node and determine which sub tree to search by comparing the value in the search field of the given record with the key values in the node.

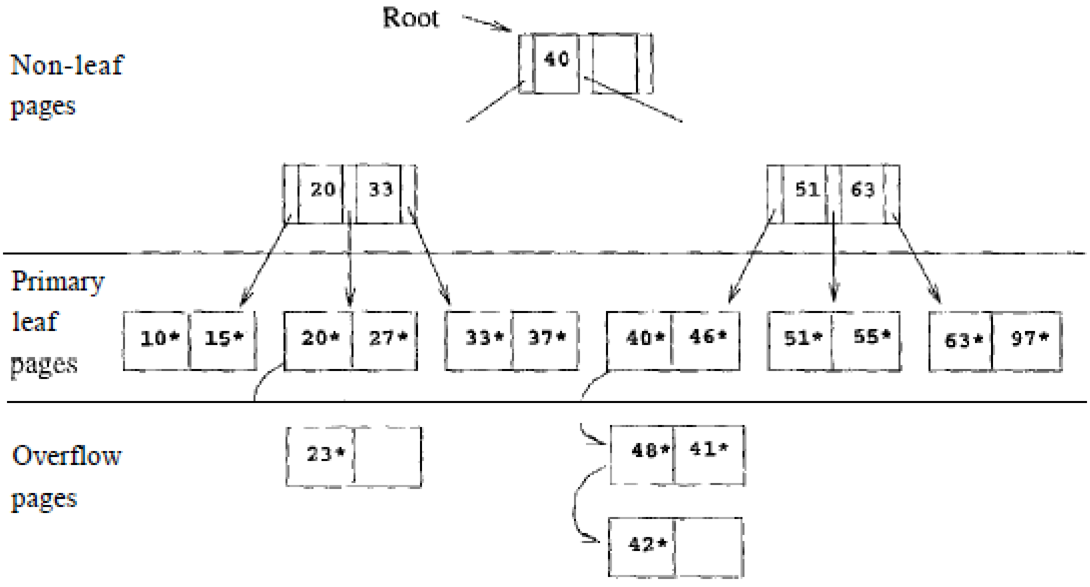
The following example illustrates the ISAM index structure. Consider the tree shown below



Sample ISAM Tree

Figure 6.10

All searches begin at the root. For example, to locate a record with the key value 27, we start at the root and follow the left pointer, since 27 < 40. We then follow the middle pointer, since 20 <= 27 < 33. For a range search, we find the first qualifying data entry as for an equality selection and then retrieve primary leaf pages sequentially The primary leaf pages are assumed to be allocated sequentially this assumption is reasonable because the number of such pages is known when the tree is created and does not change subsequently under inserts and deletes-and so no 'next leaf page' pointers are needed.



ISAM Tree after Inserts

Figure 6.11

We assume that each leaf page can contain two entries. If we now insert a record with key value 23, the entry 23* belongs in the second data page, which already contains

20* and 27* and has no more space. We deal with this situation by adding an overflow page and putting 23* in the overflow page. Chains of overflow pages can easily develop. For instance, inserting 48*, 41* and 42* leads to an overflow chain of two pages. all these insertions shown in Figure 6.11

The deletion of an entry h is handled by simply removing the entry. If this entry is on an overflow page and the overflow page becomes empty, the page can be removed. If the entry is on a primary page and deletion makes the primary page empty, the simplest approach is to simply leave the empty primary page as it is; it serves as a placeholder for future insertions. Thus, the number of primary leaf pages is fixed at file creation time.

6.2.3 B+ Trees:

6.2.3.1 Introduction of B+ Tree:

To implement dynamic multilevel indexing, B-tree and B+ tree are generally employed. B+ tree eliminates the drawback of storing the data pointer by storing data pointers only at the leaf nodes of the tree. Thus, the structure of leaf nodes of a B+ tree is quite different from the structure of internal nodes of the B tree.

B+ Tree:

The B+ tree is a balanced binary search tree. It follows a multi-level index format.

In the B+ tree, leaf nodes denote actual data pointers. B+ tree ensures that all leaf nodes remain at the same height.

In the B+ tree, the leaf nodes are linked using a link list. Therefore, a B+ tree can support random access as well as sequential access.

Structure of B+ Tree:

In the B+ tree, every leaf node is at equal distance from the root node. The B+ tree is of the order n where n is fixed for every B+ tree.

It contains an internal node and leaf node.

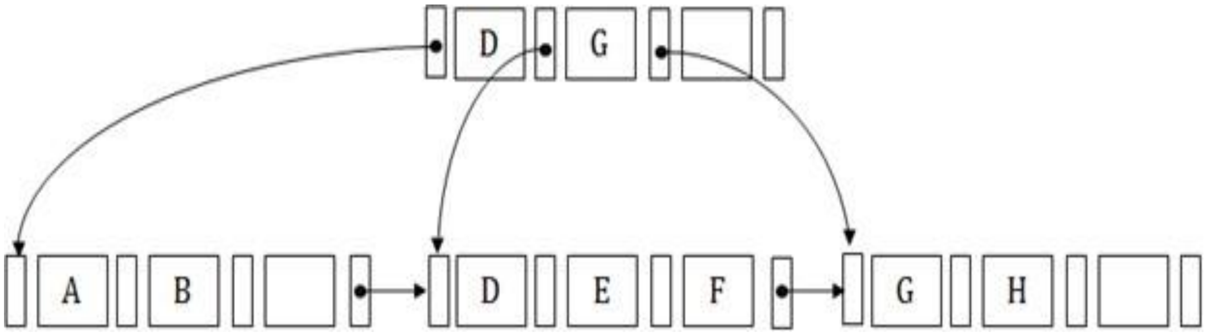


Figure 6.12

Internal node:

An internal node of the B+ tree can contain at least n/2 record pointers except the root node. At most, an internal node of the tree contains n pointers.

Leaf node:

The leaf node of the B+ tree can contain at least n/2 record pointers and n/2 key values. At most, a leaf node contains n record pointer and n key values.

Every leaf node of the B+ tree contains one block pointer P to point to next leaf node.

6.2.3.2 Searching a record in B+ Tree:

Suppose we have to search 55 in the below B+ tree structure. First, we will fetch for the intermediary node which will direct to the leaf node that can contain a record for 55.

So, in the intermediary node, we will find a branch between 50 and 75 nodes. Then at the end, we will be redirected to the third leaf node. Here DBMS will perform a sequential search to find 55.

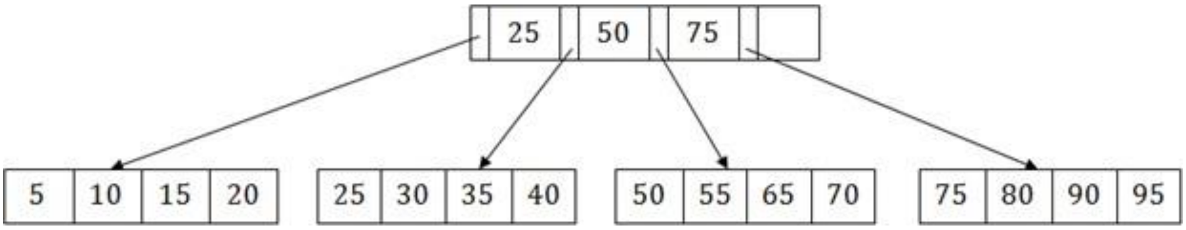


Figure 6.13

6.2.3.3 B+ Tree Insertion:

Suppose we want to insert a record 60 in the below structure. It will go to the 3rd leaf node after 55. It is a balanced tree, and a leaf node of this tree is already full, so we cannot insert 60 there.

In this case, we have to split the leaf node, so that it can be inserted into tree without affecting the fill factor, balance and order.

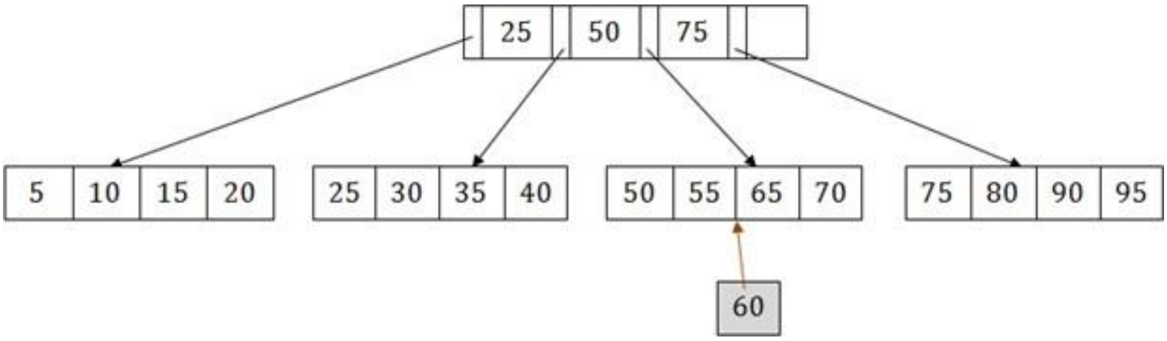


Figure 6.14

The 3rd leaf node has the values (50, 55, 60, 65, 70) and its current root node is 50. We will split the leaf node of the tree in the middle so that its balance is not altered. So we can group (50, 55) and (60, 65, 70) into 2 leaf nodes.

If these two has to be leaf nodes, the intermediate node cannot branch from 50. It should have 60 added to it, and then we can have pointers to a new leaf node.

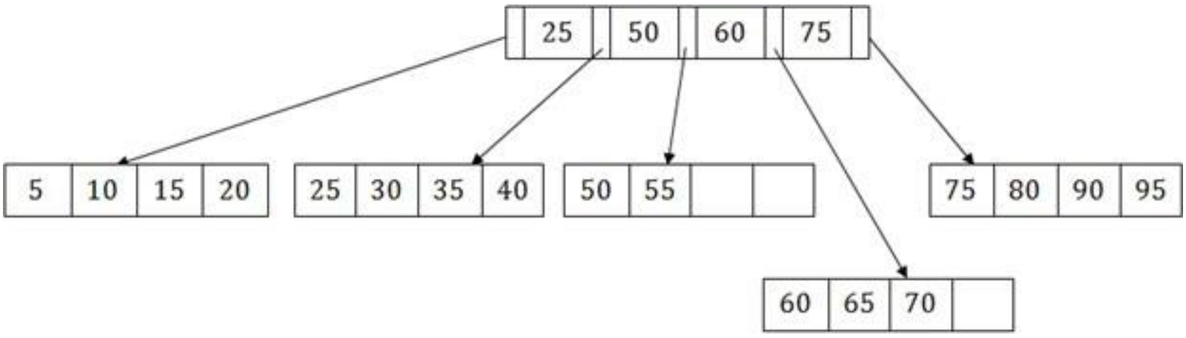


Figure 6.15

This is how we can insert an entry when there is overflow. In a normal scenario, it is very easy to find the node where it fits and then place it in that leaf node.

6.2.3.4 B+ Tree Deletion:

Suppose we want to delete 60 from the above example. In this case, we have to remove 60 from the intermediate node as well as from the 4th leaf node too. If we remove it from the intermediate node, then the tree will not satisfy the rule of the B+ tree. So we need to modify it to have a balanced tree.

After deleting node 60 from above B+ tree and re-arranging the nodes, it will show as follows:

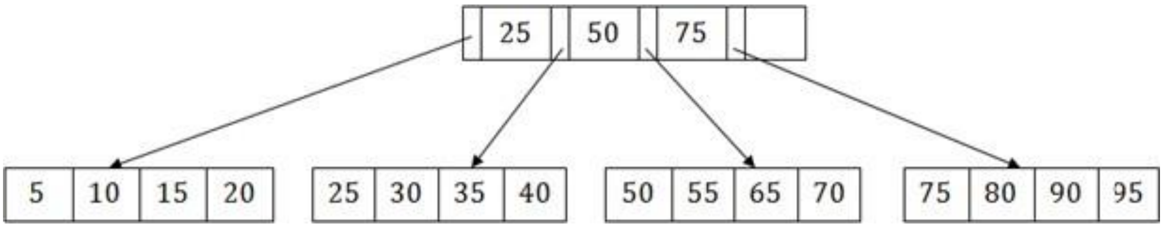


Figure 6.16

Advantages:

- Automatically Adjust the nodes to fit the new record. Similarly it re-organizes the nodes in the case of delete, if required. Hence it does not alter the definition of B+ tree.
- No file degradation problem

- Is suitable for partial and range search too

6.3 Hash Based Indexing:

Introduction:

In DBMS, hashing is a technique to directly search the location of desired data on the disk without using index structure. Data is stored in the form of data blocks whose address is generated by applying a hash function in the memory location where these records are stored known as a data block or data bucket.

- Hashing maps a search key directly to the pid of the containing page/page-overflow chain
- Doesn’t require intermediate page fetches for internal “steering nodes” of tree-based indices
- Hash-based indexes are best for equality selections.They do not support efficient range searches.
- Static and dynamic hashing techniques exist with trade-offs similar to ISAM vs. B+ trees.

Important Terminologies using in Hashing:

Here, are important terminologies which are used in Hashing:

- **Data bucket** – Data buckets are memory locations where the records are stored. It is also known as Unit Of Storage.
- **Key**: A DBMS key is an attribute or set of an attribute which helps you to identify a row(tuple) in a relation(table). This allows you to find the relationship between two tables.
- **Hash function**: A hash function, is a mapping function which maps all the set of search keys to the address where actual records are placed.
- **Linear Probing** – Linear probing is a fixed interval between probes. In this method, the next available data block is used to enter the new record, instead of overwriting on the older record.
- **Quadratic probing**- It helps you to determine the new bucket address. It helps you to add Interval between probes by adding the consecutive output of quadratic polynomial to starting value given by the original computation.
- **Hash index** – It is an address of the data block. A hash function could be a simple mathematical function to even a complex mathematical function.
- **Double Hashing** –Double hashing is a computer programming method used in hash tables to resolve the issues of has a collision.
- **Bucket Overflow**: The condition of bucket-overflow is called collision. This is a fatal stage for any static has to function.

There are mainly two types of SQL hashing methods:

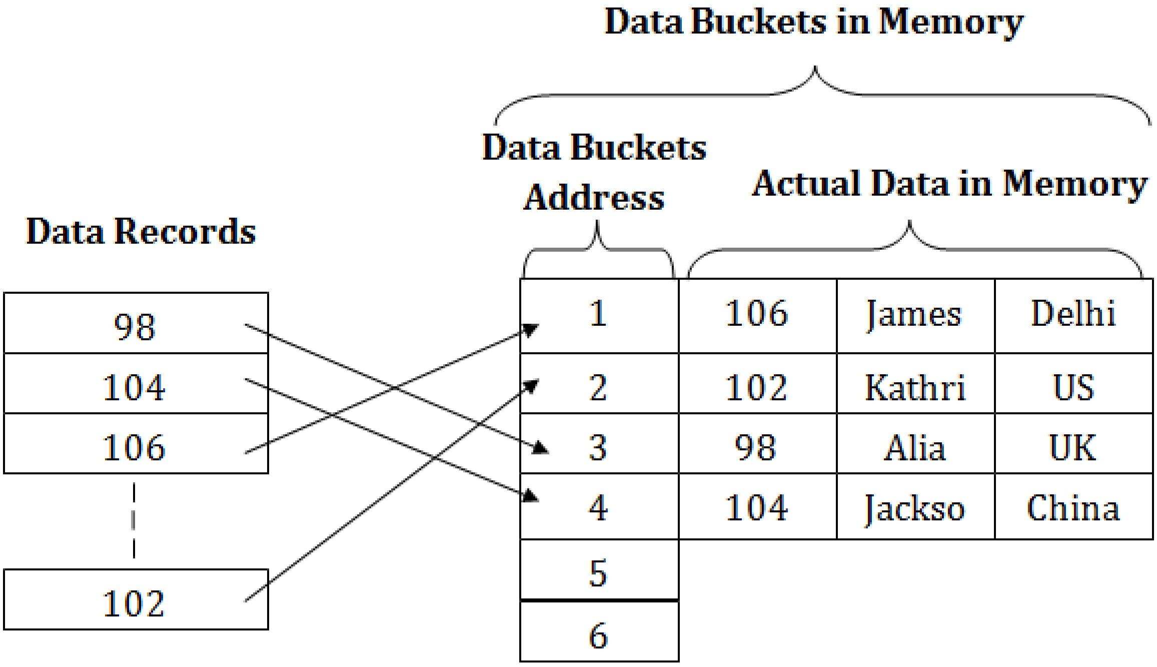
1. Static Hashing
2. Dynamic Hashing

6.3.1 Static Hashing

In the static hashing, the resultant data bucket address will always remain the same. Therefore, if you generate an address for say **Student_ID = 10** using hashing function **mod(3)**, the resultant bucket address will always be **1**. So, you will not see any change in the bucket address.

Therefore, in this static hashing method, the number of data buckets in memory always remains constant.

Hence in this static hashing, the number of data buckets in memory remains constant throughout. In this example, we will have five data buckets in the memory used to store the data.



1	106	James	Delhi
2	102	Kathri	US
3	98	Alia	UK
4	104	Jackso	China
5			
6			

Figure 6.17

Static Hash Functions

- **Inserting a record**: When a new record requires to be inserted into the table, you can generate an address for the new record using its hash key. When the address is generated, the record is automatically stored in that location.
- **Searching**: When you need to retrieve the record, the same hash function should be helpful to retrieve the address of the bucket where data should be stored.
- **Delete a record**: Using the hash function, you can first fetch the record which is you wants to delete. Then you can remove the records for that address in memory.

Static hashing is further divided into:

1. *Open hashing*

2. *Close hashing.*

1. Open Hashing: In Open hashing method, Instead of overwriting older one the next available data block is used to enter the new record, This method is also known as linear probing.

For example: suppose R3 is a new address which needs to be inserted, the hash function generates address as 112 for R3. But the generated address is already full. So the system searches next available data bucket, 113 and assigns R3 to it.

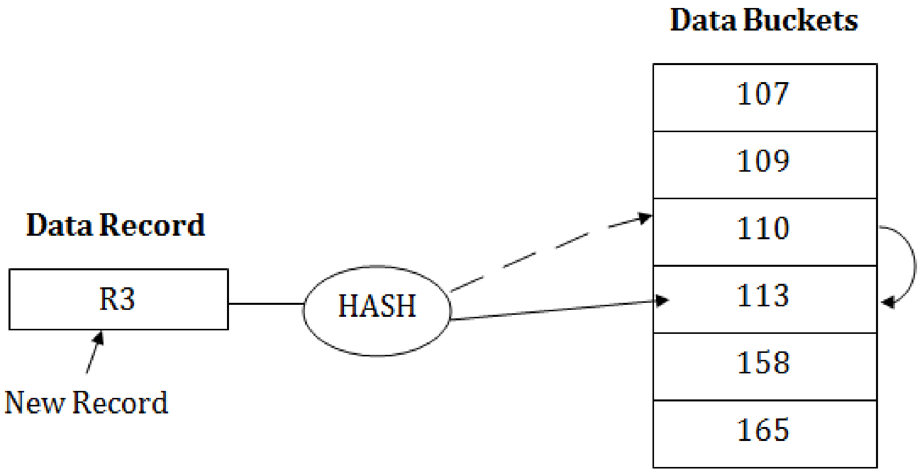


Figure 6.18

***2.* Close Hashing:**

When buckets are full, then a new data bucket is allocated for the same hash result and is linked after the previous one. This mechanism is known as **Overflow chaining**.

For example: Suppose R3 is a new address which needs to be inserted into the table, the hash function generates address as 110 for it. But this bucket is full to store the new data. In this case, a new bucket is inserted at the end of 110 buckets and is linked to it.

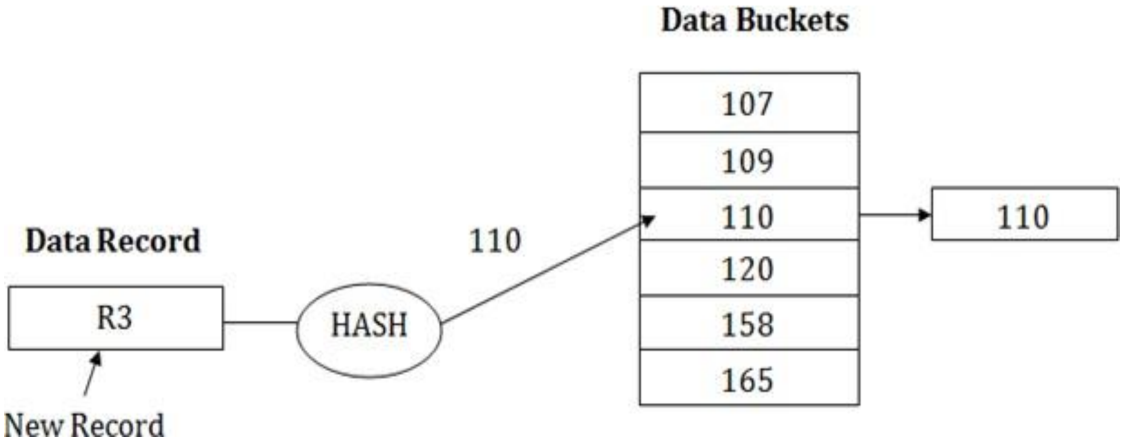


Figure 6.19

6.3.2 Dynamic Hashing:

- The dynamic hashing method is used to overcome the problems of static hashing like bucket overflow.
- In this method, data buckets grow or shrink as the records increases or decreases. This method is also known as Extendable hashing method.
- This method makes hashing dynamic, i.e., it allows insertion or deletion without resulting in poor performance.

How to search a key

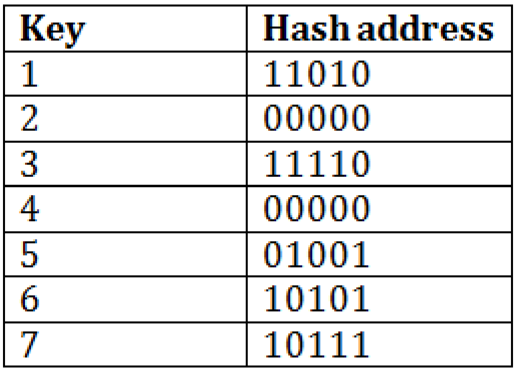
- First, calculate the hash address of the key.
- Check how many bits are used in the directory, and these bits are called as i.
- Take the least significant i bits of the hash address. This gives an index of the directory.
- Now using the index, go to the directory and find bucket address where the record might be.

How to insert a new record

- Firstly, you have to follow the same procedure for retrieval, ending up in some bucket.
- If there is still space in that bucket, then place the record in it.
- If the bucket is full, then we will split the bucket and redistribute the records.

For example:

Consider the following grouping of keys into buckets, depending on the prefix of their hash address:



Key	Hash address
1	11010
2	00000
3	11110
4	00000
5	01001
6	10101
7	10111

Figure 6.20

The last two bits of 2 and 4 are 00. So it will go into bucket B0. The last two bits of 5 and 6 are 01, so it will go into bucket B1. The last two bits of 1 and 3 are 10, so it will go into bucket B2. The last two bits of 7 are 11, so it will go into B3.

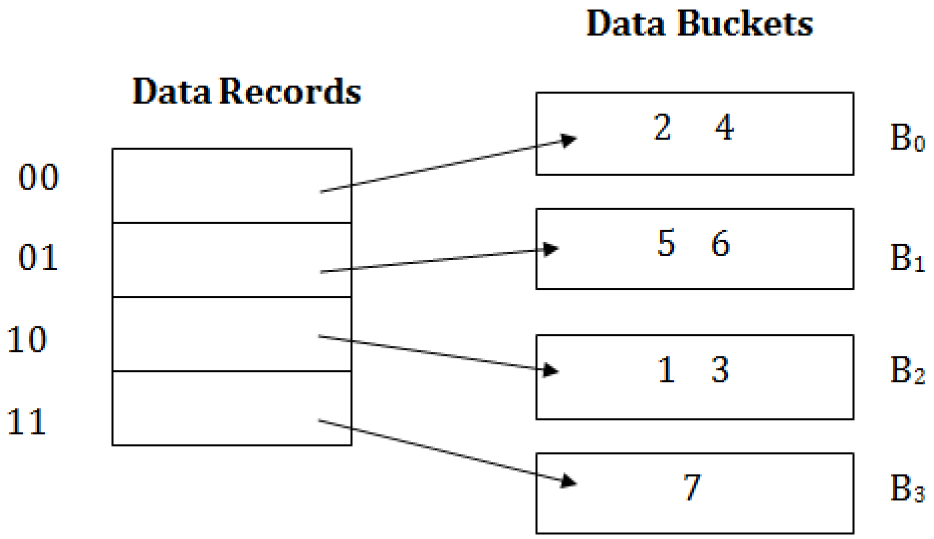


Figure 6.21

Insert key 9 with hash address 10001 into the above structure:

- Since key 9 has hash address 10001, it must go into the first bucket. But bucket B1 is full, so it will get split.
- The splitting will separate 5, 9 from 6 since last three bits of 5, 9 are 001, so it will go into bucket B1, and the last three bits of 6 are 101, so it will go into bucket B5.
- Keys 2 and 4 are still in B0. The record in B0 pointed by the 000 and 100 entry because last two bits of both the entry are 00.
- Keys 1 and 3 are still in B2. The record in B2 pointed by the 010 and 110 entry because last two bits of both the entry are 10.
- Key 7 are still in B3. The record in B3 pointed by the 111 and 011 entry because last two bits of both the entry are 11.

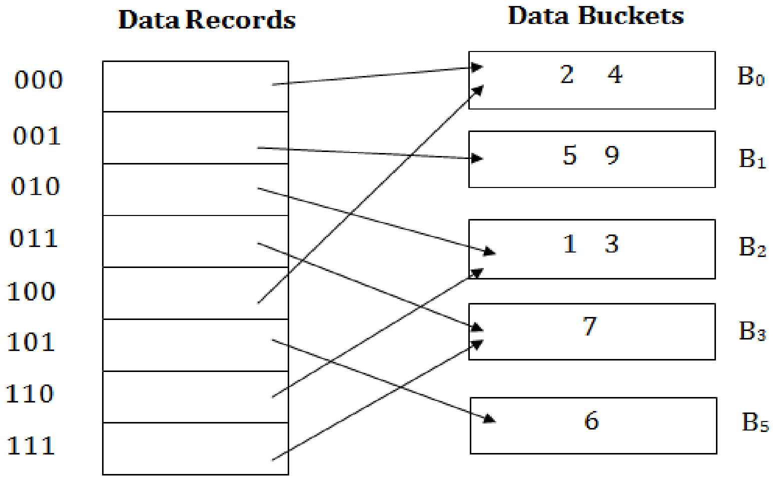


Figure 6.22

Advantages of dynamic hashing:

- In this method, the performance does not decrease as the data grows in the system. It simply increases the size of memory to accommodate the data.
- In this method, memory is well utilized as it grows and shrinks with the data. There will not be any unused memory lying.
- This method is good for the dynamic database where data grows and shrinks frequently.

6.3.3 Linear Hashing:

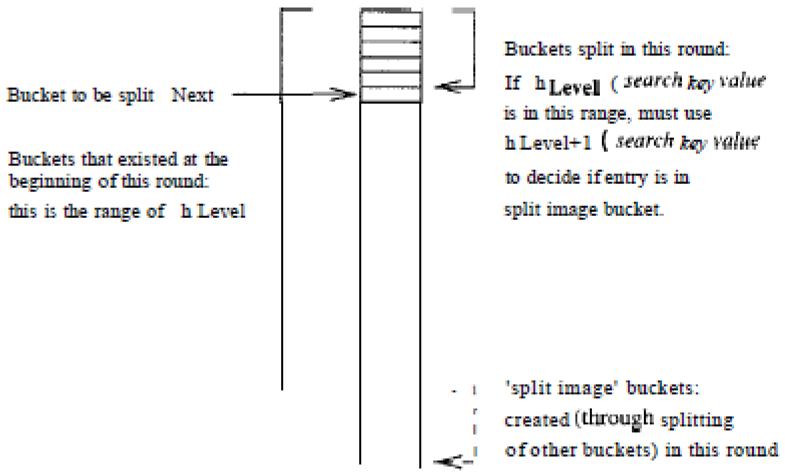
Linear Hashing is a dynamic hashing technique, like Extendible Hashing, adjusting gracefully to inserts and deletes. In contrast to Extendible Hashing, it does not require a directory, deals naturally with collisions, and offers a lot of flexibility with respect to the timing of bucket splits.

If the data distribution is very skewed, however, overflow chains could cause Linear Hashing performance to be worse than that of Extendible Hashing.

The scheme utilizes a family of hash functions h0, h1, h2, ... , with the property that each function's range is twice that of its predecessor.

That is, if h_i maps a data entry into one of M buckets, h_{i+1} maps a data entry into one of 2M buckets.

The idea is best understood in terms of **rounds** of splitting. During round number *Level*, only hash functions *hLevel* and *hLevel+1* are in use. The buckets in the file at the beginning of the round are split, one by one from the first to the last bucket, thereby doubling the number of buckets.


Bucket to be split Next
Buckets that existed at the beginning of this round: this is the range of h Level
Buckets split in this round: If h Level (search key value is in this range, must use h Level+1 (search key value to decide if entry is in split image bucket.
'split image' buckets: created (through splitting of other buckets) in this round

Buckets during a Round in Linear Hashing

Figure 6.23

Unlike Extendible Hashing, when an insert triggers a split, the bucket into which the data entry is inserted is not necessarily the bucket that is split. An overflow page is added to store the newly inserted data entry as in Static Hashing.

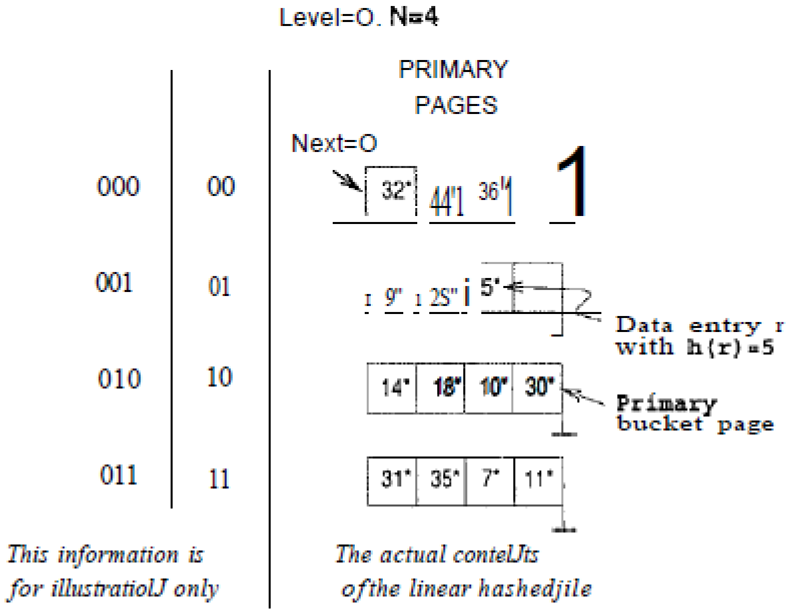
We now describe Linear Hashing in more detail.

A counter *Level* is used to indicate the current round number and is initialized to 0. The bucket to split is denoted by *Next* and is initially bucket. We denote the number of buckets in the file at the beginning of round *Level* by *NLevel*.

We can easily verify that N*Level* = N * *2Level*. Let the number of buckets at the beginning of round 0, denoted by No, be N.

Whenever a split is triggered the *Next* bucket is split, and hash function *hLevel+l* redistributes entries between this bucket (say bucket number b) and its split image the split image is therefore bucket number *b+ NLevel*.

After splitting a bucket, the value of Next is incremented by 1. In the example file,



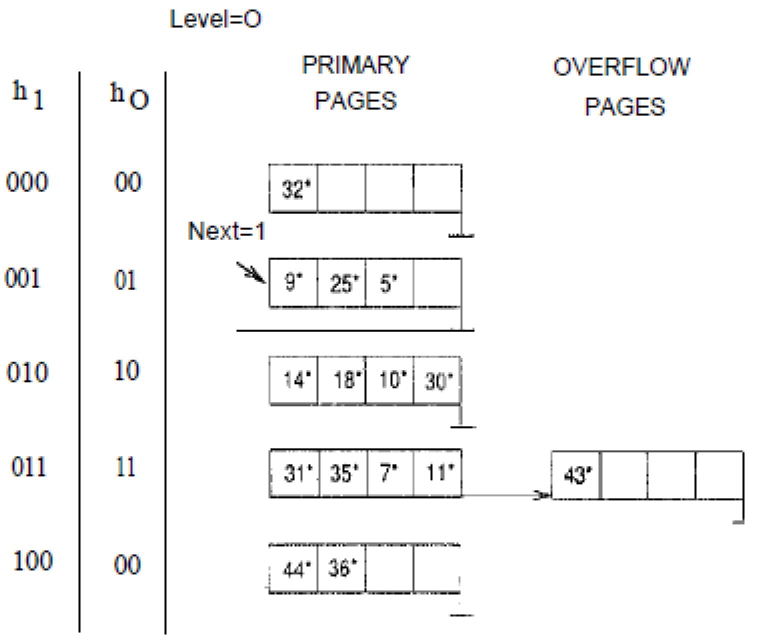
Example of a Linear Hashed File

Figure 6.24

data entry 43* triggers a split. The file after completing the insertion is shown in the figure.

At any time in .the middle of a round *Level*, all buckets above bucket Next have been split, and the file contains buckets that are their split images, as illustrated.

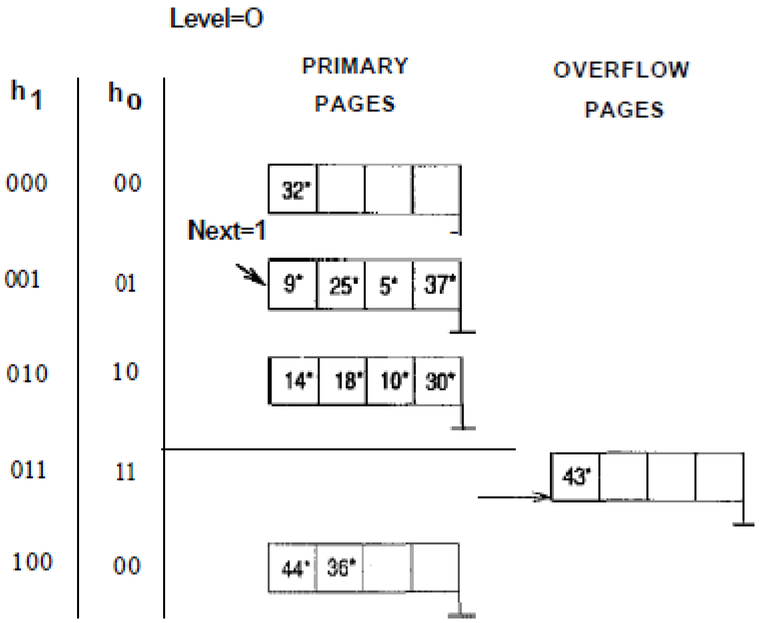
Buckets *Next* through *NLevel* have not yet been split. If we use *hLevel* on a data entry and obtain a number b in the range Next through *NLevel*, the data entry belongs to bucket b.



After Inserting Record r with h(T) = 43

Figure 6.25

Not all insertions trigger a split, of course. If we insert 37* into the file the appropriate bucket has space for the new data entry. The file after the insertion is shown in Figure



After Inserting Record r with h(r) = 37

Figure 6.26

Sometimes the bucket pointed to by Next (the current candidate for splitting) is full, and a new data entry should be inserted in this bucket. In this case, a split is triggered, of course, but we do not need a new overflow bucket. This situation is illustrated by inserting 29* into the file.

When Next is equal to NLevel - 1 and a split is triggered, we split the last of the buckets present in the file at the beginning of round Level.

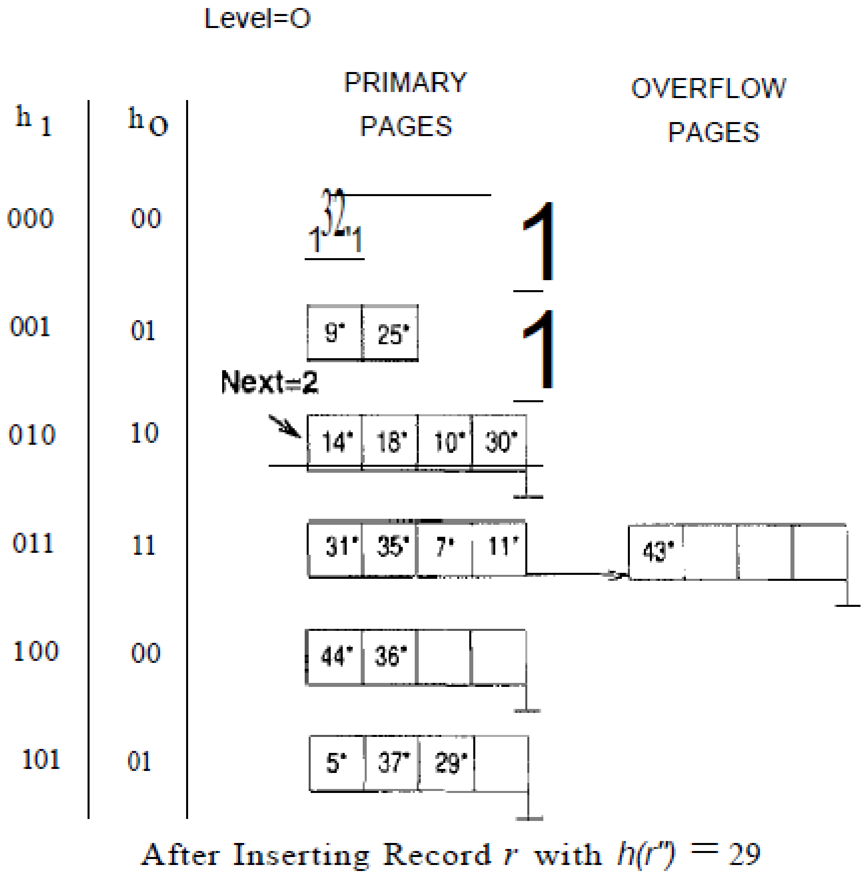


Figure 6.27

The number of buckets after the split is twice the number at the beginning of the round, and we start a new round with Level incremented by 1 and Next reset to O. Incrementing Level amounts to doubling the effective range into which keys are hashed.

'We not discuss deletion in detail, but it is essentially the inverse of insertion. If the last bucket in the file is empty, it can be removed and Next can be decremented.

If we wish, we can combine the last bucket with its split image even when it is not empty, using some criterion to trigger this merging in essentially the same way.

6.3.4 Extendable vs. Linear hashing:

To understand the relationship between Linear Hashing and Extendible Hashing, imagine that we also have a directory in Linear Hashing with elements 0 to N - 1. The first split is at bucket 0, and so we add directory element N.

we may imagine that the entire directory has been doubled at this point; however, because element 1 is the same as element N + 1, element 2 is the same as element N+2, and so on, we can avoid the actual copying for the rest of the directory.

The second split occurs at bucket 1; now directory element N + 1 becomes significant and is added. At the end of the round, all the original N buckets are split, and the directory is doubled in size.

The directory analogy is useful for understanding the ideas behind Extendible and Linear Hashing.

However, the directory structure can be avoided for Linear Hashing by allocating primary bucket pages consecutively, which would allow us to locate the page for bucket i by a simple offset calculation.

For uniform distributions, this implementation of Linear Hashing has a lower average cost for equality selections.

A different implementation of Linear Hashing, in which a directory is actually maintained, offers the flexibility of not allocating one page per bucket; null directory elements can be used as in Extendible Hashing. However, this implementation introduces the overhead of a directory level and could prove costly for large, uniformly distributed files.