UNIT I   RELATIONAL DATABASES
Purpose of Database System – Views of data – Data Models – Database System Architecture – Introduction to relational databases – Relational Model – Keys – Relational Algebra – SQL fundamentals – Advanced SQL features – Embedded SQL– Dynamic SQL

INTRODUCTION
DATABASE

Database is collection of data which is related by some aspect. Data is collection of facts and figures which can be processed to produce information. Mostly data represents recordable facts. Data aids in producing information which is based on facts. A database management system stores data, in such a way which is easier to retrieve, manipulate and helps to produce information.

So a database is a collection of related data that we can use for

- Defining - specifying types of data
- Constructing - storing & populating
- Manipulating - querying, updating, reporting

DISADVANTAGES OF FILE SYSTEM OVER DB

In the early days, File-Processing system is used to store records. It uses various files for storing the records.
 Drawbacks of using file systems to store data:

- Data redundancy and inconsistency
  -Multiple file formats, duplication of information in different files
- Difficulty in accessing data
  Need to write a new program to carry out each new task
- Data isolation — multiple files and formats
- Integrity problems
  - Hard to add new constraints or change existing ones
- Atomicity problem
  -Failures may leave database in an inconsistent state with partial updates carried
  Out. E.g. transfer of funds from one account to another should either complete or not
  happen at all
- Concurrent access anomalies
  - Concurrent accessed needed for performance
- Security problems

 Database systems offer solutions to all the above problems

PURPOSE OF DATABASE SYSTEM

The typical file processing system is supported by a conventional operating system. The system stores permanent records in various files, and it needs different application programs to extract records from, and add records to, the appropriate files. A file processing system has a number of major disadvantages.

- Data redundancy and inconsistency
- Difficulty in accessing data
- Data isolation – multiple files and formats
- Integrity problems
- Atomicity of updates
- Concurrent access by multiple users
- Security problems

1.Data redundancy and inconsistency:

In file processing, every user group maintains its own files for handling its data processing applications.
Example:
Consider the UNIVERSITY database. Here, two groups of users might be the course registration personnel and the accounting office. The accounting office also keeps data on registration and
related billing information, whereas the registration office keeps track of student courses and grades.Storing the same data multiple times is called data redundancy.This redundancy leads to several problems.
•Need to perform a single logical update multiple times.
•Storage space is wasted.
•Files that represent the same data may become inconsistent.
Data inconsistency is the various copies of the same data may no larger Agree. **Example:**
One user group may enter a student's birth date erroneously as JAN-19-1984, whereas the
other user groups may enter the correct value of JAN-29-1984.

2. Difficulty in accessing data
File processing environments do not allow needed data to be retrieved in a convenient and efficient manner.

3.Data isolation
Because data are scattered in various files, and files may be in different formats, writing new application programs to retrieve the appropriate data is difficult.

4.Integrity problems
The data values stored in the database must satisfy certain types of consistency constraints. **Example:**
The balance of certain types of bank accounts may never fall below a prescribed amount . Developers enforce these constraints in the system by addition appropriate code in the various application programs

5.Atomicity problems
Atomic means the transaction must happen in its entirety or not at all. It is difficult to ensure atomicity in a conventional file processing system.

Example:
Consider a program to transfer $50 from account A to account B. If a system failure occurs during the execution of the program, it is possible that the $50 was removed from account A but was not credited to account B, resulting in an inconsistent database state.

6.Concurrent access anomalies
For the sake of overall performance of the system and faster response, many systems allow multiple users to update the data simultaneously. In such an environment, interaction of concurrent updates is possible and may result in inconsistent data. To guard against this possibility, the system must maintain some form of supervision. But supervision is difficult to
provide because data may be accessed by many different application programs that have not been coordinated previously.

Example: When several reservation clerks try to assign a seat on an airline flight, the system should ensure that each seat can be accessed by only one clerk at a time for assignment to a passenger.

7. Security problems
Enforcing security constraints to the file processing system is difficult.

APPLICATION OF DATABASE
Database Applications
- Banking: all transactions
- Airlines: reservations, schedules
- Universities: registration, grades
- Sales: customers, products, purchases
- Manufacturing: production, inventory, orders, supply chain
- Human resources: employee records, salaries, tax deductions
- Telecommunication: Call History, Billing
- Credit card transactions: Purchase details,Statements

VIEWS OF DATA
It refers that how database is actually stored in database, what data and structure of data used by database for data. So describe all this database provides user with views and these are
- **Data abstraction**
- **Instances and schemas**

Data abstraction
As a data in database are stored with very complex data structure so when user come and want to access any data, he will not be able to access data if he has go through this data structure. So to simplify the interaction of user and database, DBMS hides some information which is not of user interest, a this is called data abstraction:- **So developer hides complexity from user and store abstract view of data.**

Data abstraction has three level of abstractions
- **level / internal level**
- **Logical level / conceptual level**
- **view level / external level**

Physical level:- this is the lowest level of data abstraction which describe How data is actual stored in database. This level basically describe the data structure and access path /indexing use for accessing file.

Logical level:- The next level of abstraction describe what data are stored in the database and what are the relationship existed among those of data.

View level:- In this level user only interact with database and the complexity remain unview . user see data and there may be many views of one data like chart and graph.

View of Data
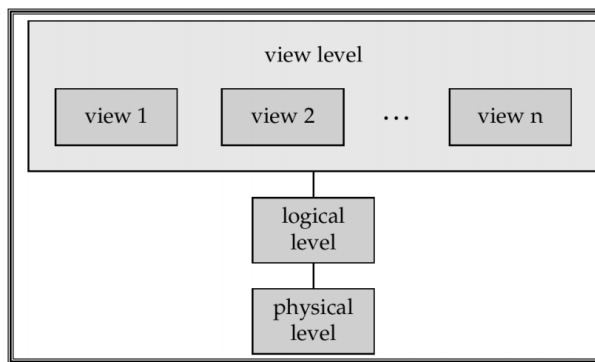
Levels of Abstraction

DATA MODELS IN DBMS

A **Data Model** is a logical structure of Database. It describes the design of database to reflect entities, attributes, relationship among data, constrains etc.
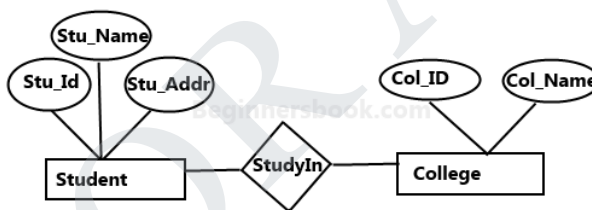
Types of Data Models:

Object based logical Models – Describe data at the conceptual and view levels.

   1.   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.

A sample E-R Diagram:

Sample E-R Diagram

   2.   Object oriented Model

An object data model is a data model based on object-oriented programming, associating methods (procedures) with objects that can benefit from class hierarchies. Thus, "objects" are levels of abstraction that include attributes and behavior

Record based logical Models – Like Object based model, they also describe data at the conceptual and view levels. These models specify logical structure of database with records, fields and attributes.

   1.   Relational Model

In relational model, the data and relationships are represented by collection of inter-related tables. Each table is a group of column and rows, where column represents attribute of an entity and rows represents records.

Sample relationship Model: Student table with 3 columns and three records.

Stu_Id	Stu_Name	Stu_Age
111	Ashish	23
123	Saurav	22
169	Lester	24

   2.   Hierarchical Model

In hierarchical model, data is organized into a tree like structure with each record is having one parent record and many children. The main drawback of this model is that, it can have only one to many relationships between nodes.

Sample Hierarchical Model Diagram:
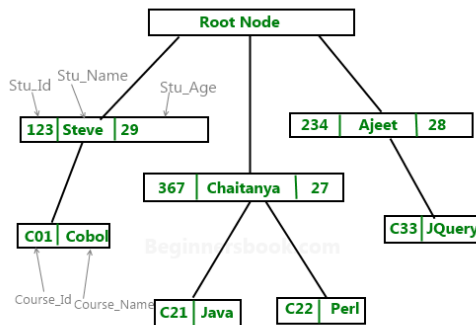
3. **Network Model** – Network Model is same as hierarchical model except that it has graph-like structure rather than a tree-based structure. Unlike hierarchical model, this model allows each record to have more than one parent record.

Physical Data Models – These models describe data at the lowest level of abstraction.

Three Schema Architecture

The goal of the three schema architecture is to separate the user applications and the physical database. The schemas can be defined at the following levels:

1. **The internal level** – has an internal schema which describes the physical storage structure of the database. Uses a physical data model and describes the complete details of data storage and access paths for the database.

2. **The conceptual level** – has a conceptual schema which describes the structure of the database for users. It hides the details of the physical storage structures, and concentrates on describing entities, data types, relationships, user operations and constraints. Usually a representational data model is used to describe the conceptual schema.

3. **The External or View level** – includes external schemas or user vies. Each external schema describes the part of the database that a particular user group is interested in and hides the rest of the database from that user group. Represented using the representational data model.

The three schema architecture is used to visualize the schema levels in a database. The three schemas are only descriptions of data, the data only actually exists is at the physical level.

Database Architecture
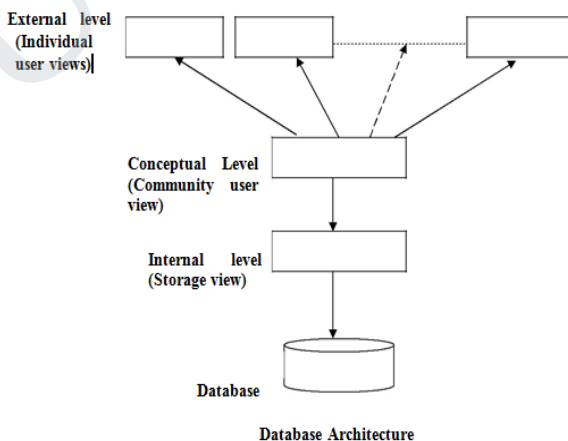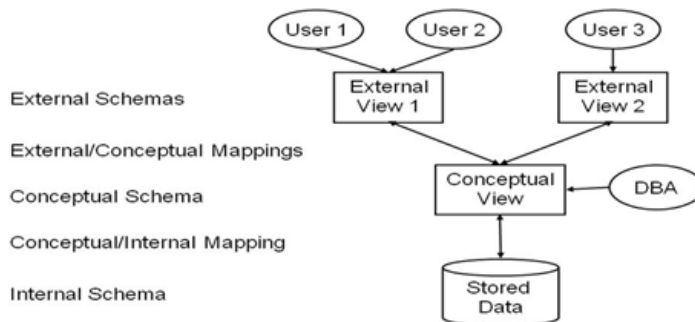
COMPONENTS OF DBMS
Database Users
Users are differentiated by the way they expect to interact with the system
- Application programmers
- Sophisticated users
- Naïve users
- Database Administrator
- Specialized users etc,.

Application programmers:
   Professionals who write application programs and using these application programs they interact with the database system

Sophisticated users :
   These user interact with the database system without writing programs, But they submit queries to retrieve the information

Specialized users:
   Who write specialized database applications to interact with the database system.

Naïve users:
   Interacts with the database system by invoking some application programs that have been written previously by application programmers
 Eg : people accessing database over the web

Database Administrator:
  Coordinates all the activities of the database system; the database administrator has a good understanding of the enterprise's information resources and needs.
> - Schema definition
> - Access method definition
> - Schema and physical organization modification
> - Granting user authority to access the database
> - Monitoring performance

Storage Manager
The Storage Manager include these following components/modules
> - Authorization Manager
> - Transaction Manager
> - File Manager
> - Buffer Manager

❖ Storage manager is a program module that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system.

❖ **The storage manager is responsible to the following tasks:**
> - interaction with the file manager
> - efficient storing, retrieving and updating of data

       Authorization Manager
> - Checks whether the user is an authorized person or not
> - Test the satisfaction of integrity constraints

      Transaction Manager
Responsible for concurrent transaction execution It ensures that the database remains in a consistent state despite of the system failure
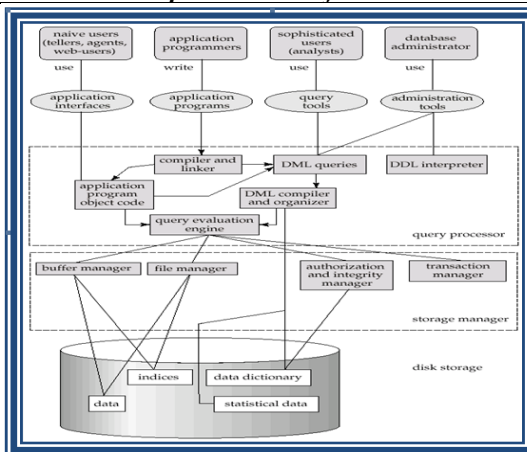
EVOLUTION OF RDBMS

Before the acceptance of Codd's Relational Model, database management systems was just an ad hoc collection of data designed to solve a particular type of problem, later extended to solve more basic purposes. This led to complex systems, which were difficult to understand, install, maintain and use. These database systems were plagued with the following problems:

- They required large budgets and staffs of people with special skills that were in short supply.
- Database administrators' staff and application developers required prior preparation to access these database systems.
- End-user access to the data was rarely provided.
- These database systems did not support the implementation of business logic as a DBMS responsibility.

Hence, the objective of developing a relational model was to address each and every one of the shortcomings that plagued those systems that existed at the end of the 1960s decade, and make DBMS products more widely appealing to all kinds of users.

The existing relational database management systems offer powerful, yet simple solutions for a wide variety of commercial and scientific application problems. Almost every industry uses relational systems to store, update and retrieve data for operational, transaction, as well as decision support systems.

RELATIONAL DATABASE

A relational database is a database system in which the database is organized and accessed according to the relationships between data items without the need for any consideration of physical orientation and relationship. Relationships between data items are expressed by means of **tables**.

It is a tool, which can help you store, manage and disseminate information of various kinds. It is a collection of objects, tables, queries, forms, reports, and macros, all stored in a computer program all of which are inter-related.

It is a method of structuring data in the form of records, so that relations between different entities and attributes can be used for data access and transformation.

RELATIONAL DATABASE MANAGEMENT SYSTEM

A Relational Database Management System (RDBMS) is a system, which allows us to perceive data as tables (and nothing but tables), and *operators* necessary to manipulate that data are at the user's disposal.

Features of an RDBMS

The features of a relational database are as follows:

- ➢ The ability to create multiple relations (tables) and enter data into them
- ➢ An interactive query language
- ➢ Retrieval of information stored in more than one table
- ➢ Provides a *Catalog* or *Dictionary*, which itself consists of tables (called *system* tables)

Basic Relational Database Terminology

Catalog:

A catalog consists of all the information of the various schemas (external, conceptual and internal) and also all of the corresponding mappings (external/conceptual, conceptual/internal).

It contains detailed information regarding the various objects that are of interest to the system itself; e.g., tables, views, indexes, users, integrity rules, security rules, etc.

In a relational database, the entities of the ERD are represented as *tables* and their attributes as the *columns* of their respective tables in a database schema.

It includes some important terms, such as:

- *Table*: Tables are the basic storage structures of a database where data about something in the real world is

stored. It is also called a *relation or an entity.*

- *Row:* Rows represent collection of data required for a particular entity. In order to identify each row as unique there should be a *unique identifier* called the *primary key*, which allows no duplicate rows. For example in a library every member is unique and hence is given a membership number, which uniquely identifies each member. A row is also called a *record* or a *tuple.*
- *Column:* Columns represent characteristics or attributes of an entity. Each attribute maps onto a column of a table. Hence, a column is also known as an *attribute.*
- *Relationship:* Relationships represent a logical link between two tables. A relationship is depicted by a *foreign key* column.
- Degree: number of attributes
- Cardinality: number of tuples
- An attribute of an entity has a particular value. The set of possible values That a given attribute can have is called its *domain.*

KEYS AND THEIR USE

Key: An attribute or set of attributes whose values uniquely identify each entity in an entity set is called a key for that entity set.

Super Key: If we add additional attributes to a key, the resulting combination would still uniquely identify an instance of the entity set. Such augmented keys are called super keys.

Primary Key: It is a minimum super key.

It is *a unique **identifier** for the table* (a column or a column combination with the property that at any given time no two rows of the table contain the same value in that column or column combination).

Foreign Key: A foreign key is a field (or collection of fields) in one table that uniquely identifies a row of another table. In simpler words, the foreign key is defined in a second table, but it refers to the primary key in the first table.

Candidate Key: There may be two or more attributes or combinations of attributes that uniquely identify an instance of an entity set. These attributes or combinations of attributes are called candidate keys.

Secondary Key: A secondary key is an attribute or combination of attributes that may not be a candidate key, but that classifies the entity set on a particular characteristic. Any key consisting of a single attribute is called a **simple key,** while that consisting of a combination of attributes is called a **composite key**.

Referential Integrity

Referential Integrity can be defined as an integrity constraint that specifies that the value (or existence) of an attribute in one relation depend on the value (or existence) of an attribute in the same or another relation. Referential integrity in a relational database is consistency between coupled tables. It is usually enforced by the combination of a primary key and a foreign key. For referential integrity to hold, any field in a table that is declared a foreign key can contain only values from a parent table's primary key field. For instance, deleting a record that contains a value referred to by a foreign key in another table would break referential integrity.

Relational Model

Relational data model is the primary data model, which is used widely around the world for data storage and processing. This model is simple and it has all the properties and capabilities required to process data with storage efficiency.

Concepts

Tables − In relational data model, relations are saved in the format of Tables. This format stores the relation among entities. A table has rows and columns, where rows represents records and columns represent the attributes.

Tuple − A single row of a table, which contains a single record for that relation is called a tuple.

Relation instance − A finite set of tuples in the relational database system represents relation instance. Relation instances do not have duplicate tuples.

Relation schema − A relation schema describes the relation name (table name), attributes, and their names.

Relation key − Each row has one or more attributes, known as relation key, which can identify the row in the relation (table) uniquely.

Attribute domain − Every attribute has some pre-defined value scope, known as attribute domain.

Constraints

Every relation has some conditions that must hold for it to be a valid relation. These conditions are called **Relational Integrity Constraints**. There are three main integrity constraints −

- Key constraints
- Domain constraints
- Referential integrity constraints

Key Constraints

There must be at least one minimal subset of attributes in the relation, which can identify a tuple uniquely. This minimal subset of attributes is called **key**for that relation. If there are more than one such minimal subsets, these are called *candidate keys*.

Key constraints force that −

- in a relation with a key attribute, no two tuples can have identical values for key attributes.
- a key attribute can not have NULL values.

Key constraints are also referred to as Entity Constraints.

Domain Constraints

Attributes have specific values in real-world scenario. For example, age can only be a positive integer. The same constraints have been tried to employ on the attributes of a relation. Every attribute is bound to have a specific range of values. For example, age cannot be less than zero and telephone numbers cannot contain a digit outside 0-

Referential integrity Constraints

Referential integrity constraints work on the concept of Foreign Keys. A foreign key is a key attribute of a relation that can be referred in other relation.

Referential integrity constraint states that if a relation refers to a key attribute of a different or same relation, then that key element must exist.

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.

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.

The fundamental operations of relational algebra are as follows −

- Select
- Project
- Union
- Set different
- Cartesian product
- Rename

We will discuss all these operations in the following sections.

Select 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 − $=, \neq, \geq, <, >, \leq$.

For example −

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

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

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

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

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

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 −

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

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

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 U 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.

$\prod_{author}$ (Books) $\cup$ $\prod_{author}$ (Articles)

Output − Projects the names of the authors who have either written a book or an article or both.

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**.

$\prod_{author}$ (Books) − $\prod_{author}$ (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}

$\sigma_{author = 'tutorialspoint'}$(Books X Articles)

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

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**.

Additional operations are −

- Set intersection
- Assignment
- Natural join

SQL FUNDAMENTALS:

SQL is a standard computer language for accessing and manipulating databases.

What is SQL?

- SQL stands for **S**tructured **Q**uery **L**anguage
- SQL allows you to access a database
- SQL is an ANSI standard computer language
- SQL can execute queries against a database
- SQL can retrieve data from a database
- SQL can insert new records in a database
- SQL can delete records from a database
- SQL can update records in a database
- SQL is easy to learn

SQL is a Standard - BUT....

SQL is an ANSI (American National Standards Institute) standard computer language for accessing and manipulating database systems. SQL statements are used to retrieve and update data in a database. SQL works with database programs like MS Access, DB2, Informix, MS SQL Server, Oracle, Sybase, etc.Unfortunately, there are many different versions of the SQL language, but to be in compliance with the ANSI standard, they must support the same major keywords in a similar manner (such as SELECT, UPDATE, DELETE, INSERT, WHERE, and others).

Note: Most of the SQL database programs also have their own proprietary extensions in addition to the SQL standard!

SQL Database Tables

A database most often contains one or more tables. Each table is identified by a name (e.g. "Customers" or "Orders"). Tables contain records (rows) with data.Below is an example of a table called "Persons":

LastName	FirstName	Address	City
Hansen	Ola	Timoteivn 10	Sandnes
Svendson	Tove	Borgvn 23	Sandnes
Pettersen	Kari	Storgt 20	Stavanger

The table above contains three records (one for each person) and four columns (LastName, FirstName, Address, and City).

SQL Queries

With SQL, we can query a database and have a result set returned.

A query like this:

```
SELECT LastName FROM Persons
```

Gives a result set like this:

LastName
Hansen
Svendson
Pettersen

Note: Some database systems require a semicolon at the end of the SQL statement. We don't use the semicolon in our tutorials.

SQL Data Manipulation Language (DML)

SQL (Structured Query Language) is a syntax for executing queries. But the SQL language also includes a syntax to update, insert, and delete records.

These query and update commands together form the Data Manipulation Language (DML) part of SQL:

- **SELECT** - extracts data from a database table
- **UPDATE** - updates data in a database table
- **DELETE** - deletes data from a database table
- **INSERT INTO** - inserts new data into a database table

SQL Data Definition Language (DDL)

The Data Definition Language (DDL) part of SQL permits database tables to be created or deleted. We can also define indexes (keys), specify links between tables, and impose constraints between database tables.

The most important DDL statements in SQL are:

- **CREATE TABLE** - creates a new database table
- **ALTER TABLE** - alters (changes) a database table
- **DROP TABLE** - deletes a database table
- **CREATE INDEX** - creates an index (search key)
- **DROP INDEX** - deletes an index

The SQL SELECT Statement

The SELECT statement is used to select data from a table. The tabular result is stored in a result table (called the result-set).

Syntax

```
SELECT column_name(s)
FROM table_name
```

Note: SQL statements are not case sensitive. SELECT is the same as select.

SQL SELECT Example

To select the content of columns named "LastName" and "FirstName", from the database table called "Persons", use a SELECT statement like this:

```
SELECT LastName,FirstName FROM Persons
```

The database table "Persons":

LastName	FirstName	Address	City
Hansen	Ola	Timoteivn 10	Sandnes
Svendson	Tove	Borgvn 23	Sandnes
Pettersen	Kari	Storgt 20	Stavanger

The result

LastName	FirstName
Hansen	Ola
Svendson	Tove
Pettersen	Kari

Select All Columns

To select all columns from the "Persons" table, use a * symbol instead of column names, like this:

SELECT * FROM Persons

Result

LastName	FirstName	Address	City
Hansen	Ola	Timoteivn 10	Sandnes
Svendson	Tove	Borgvn 23	Sandnes
Pettersen	Kari	Storgt 20	Stavanger

The Result Set

The result from a SQL query is stored in a result-set. Most database software systems allow navigation of the result set with programming functions, like: Move-To-First-Record, Get-Record-Content, Move-To-Next-Record, etc. Programming functions like these are not a part of this tutorial. To learn about accessing data with function calls,

Semicolon after SQL Statements?

Semicolon is the standard way to separate each SQL statement in database systems that allow more than one SQL statement to be executed in the same call to the server.

Some SQL tutorials end each SQL statement with a semicolon. Is this necessary? We are using MS Access and SQL Server 2000 and we do not have to put a semicolon after each SQL statement, but some database programs force you to use it.

The SELECT DISTINCT Statement

The DISTINCT keyword is used to return only distinct (different) values.

The SELECT statement returns information from table columns. But what if we only want to select distinct elements?

With SQL, all we need to do is to add a DISTINCT keyword to the SELECT statement:

Syntax

SELECT DISTINCT column_name(s) FROM table_name

Using the DISTINCT keyword

To select ALL values from the column named "Company" we use a SELECT statement like this:

SELECT Company FROM Orders

"Orders" table

Company	OrderNumber
Sega	3412
W3Schools	2312
Trio	4678
W3Schools	6798

Result

Company
Sega
W3Schools
Trio
W3Schools

Note that "W3Schools" is listed twice in the result-set.

To select only DIFFERENT values from the column named "Company" we use a SELECT DISTINCT statement like this:

SELECT DISTINCT Company FROM Orders

Result:

Company
Sega
W3Schools
Trio

Now "W3Schools" is listed only once in the result-set.

The WHERE clause is used to specify a selection criterion.

The WHERE Clause

To conditionally select data from a table, a WHERE clause can be added to the SELECT statement.

Syntax

SELECT column FROM table
WHERE column operator value

With the WHERE clause, the following operators can be used:

Operator	Description
=	Equal
<>	Not equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
BETWEEN	Between an inclusive range
LIKE	Search for a pattern

Note: In some versions of SQL the <> operator may be written as !=

Using the WHERE Clause

To select only the persons living in the city "Sandnes", we add a WHERE clause to the SELECT statement:

SELECT * FROM Persons
WHERE City='Sandnes'

"Persons" table

LastName	FirstName	Address	City	Year
Hansen	Ola	Timoteivn 10	Sandnes	1951
Svendson	Tove	Borgvn 23	Sandnes	1978
Svendson	Stale	Kaivn 18	Sandnes	1980
Pettersen	Kari	Storgt 20	Stavanger	1960

Result

LastName	FirstName	Address	City	Year
Hansen	Ola	Timoteivn 10	Sandnes	1951
Svendson	Tove	Borgvn 23	Sandnes	1978
Svendson	Stale	Kaivn 18	Sandnes	1980

Using Quotes

Note that we have used single quotes around the conditional values in the examples.

SQL uses single quotes around text values (most database systems will also accept double quotes). Numeric values should not be enclosed in quotes.

For text values:

This is correct:
SELECT * FROM Persons WHERE FirstName='Tove'
This is wrong:
SELECT * FROM Persons WHERE FirstName=Tove

For numeric values:

This is correct:
SELECT * FROM Persons WHERE Year>1965
This is wrong:
SELECT * FROM Persons WHERE Year>'1965'

The LIKE Condition

The LIKE condition is used to specify a search for a pattern in a column.

Syntax

```
SELECT column FROM table
WHERE column LIKE pattern
```

A "%" sign can be used to define wildcards (missing letters in the pattern) both before and after the pattern.

Using LIKE

The following SQL statement will return persons with first names that start with an 'O':

```
SELECT * FROM Persons
WHERE FirstName LIKE 'O%'
```

The following SQL statement will return persons with first names that end with an 'a':

```
SELECT * FROM Persons
WHERE FirstName LIKE '%a'
```

The following SQL statement will return persons with first names that contain the pattern 'la':

```
SELECT * FROM Persons
WHERE FirstName LIKE '%la%'
```

The INSERT INTO Statement

The INSERT INTO statement is used to insert new rows into a table.

Syntax

```
INSERT INTO table_name
VALUES (value1, value2,....)
```

You can also specify the columns for which you want to insert data:

```
INSERT INTO table_name (column1, column2,...)
VALUES (value1, value2,....)
```

Insert a New Row

This "Persons" table:

LastName	FirstName	Address	City
Pettersen	Kari	Storgt 20	Stavanger

And this SQL statement:

```
INSERT INTO Persons
VALUES ('Hetland', 'Camilla', 'Hagabakka 24', 'Sandnes')
```

Will give this result:

LastName	FirstName	Address	City
Pettersen	Kari	Storgt 20	Stavanger
Hetland	Camilla	Hagabakka 24	Sandnes

Insert Data in Specified Columns

This "Persons" table:

LastName	FirstName	Address	City
Pettersen	Kari	Storgt 20	Stavanger
Hetland	Camilla	Hagabakka 24	Sandnes

And This SQL statement:
```
INSERT INTO Persons (LastName, Address)
VALUES ('Rasmussen', 'Storgt 67')
```

Will give this result:

LastName	FirstName	Address	City
Pettersen	Kari	Storgt 20	Stavanger
Hetland	Camilla	Hagabakka 24	Sandnes
Rasmussen		Storgt 67	

 Null (no value …not space not empty)

The UPDATE statement is used to modify the data in a table.

Syntax

```
UPDATE table_name
SET column_name = new_value
WHERE column_name = some_value
```

Person:

LastName	FirstName	Address	City
Nilsen	Fred	Kirkegt 56	Stavanger
Rasmussen		Storgt 67	

Update one Column in a Row

We want to add a first name to the person with a last name of "Rasmussen":

```
UPDATE Person SET FirstName = 'Nina'
WHERE LastName = 'Rasmussen'
```

Result:

LastName	FirstName	Address	City
Nilsen	Fred	Kirkegt 56	Stavanger
Rasmussen	Nina	Storgt 67	

Update several Columns in a Row

We want to change the address and add the name of the city:

```
UPDATE Person
SET Address = 'Stien 12', City = 'Stavanger'
WHERE LastName = 'Rasmussen'
```

Result:

LastName	FirstName	Address	City
Nilsen	Fred	Kirkegt 56	Stavanger
Rasmussen	Nina	Stien 12	Stavanger

The DELETE Statement

The DELETE statement is used to delete rows in a table.

Syntax

```
DELETE FROM table_name
WHERE column_name = some_value
```

Person:

LastName	FirstName	Address	City
Nilsen	Fred	Kirkegt 56	Stavanger
Rasmussen	Nina	Stien 12	Stavanger

Delete
Drop

Delete a Row

"Nina Rasmussen" is going to be deleted:

```
DELETE FROM Person WHERE LastName = 'Rasmussen'
```

Result

LastName	FirstName	Address	City
Nilsen	Fred	Kirkegt 56	Stavanger

Delete All Rows

It is possible to delete all rows in a table without deleting the table. This means that the table structure, attributes, and indexes will be intact:

```
DELETE FROM table_name
or
DELETE * FROM table_name
```

The ORDER BY keyword is used to sort the result.

Sort the Rows

The ORDER BY clause is used to sort the rows.

Orders:

Company	OrderNumber
Sega	3412
ABC Shop	5678
W3Schools	2312
W3Schools	6798

Example

To display the company names in alphabetical order:

```
SELECT Company, OrderNumber FROM Orders
ORDER BY Company  ASC (asending)
```

Result:

Company	OrderNumber
ABC Shop	5678
Sega	3412
W3Schools	6798
W3Schools	2312

Example

To display the company names in alphabetical order AND the OrderNumber in numerical order:

```
SELECT Company, OrderNumber FROM Orders
ORDER BY Company, OrderNumber
```

Result:

Company	OrderNumber
ABC Shop	5678
Sega	3412
W3Schools	2312
W3Schools	6798

Aggregate functions

Aggregate functions operate against a collection of values, but return a single value.

Note: If used among many other expressions in the item list of a SELECT statement, the SELECT must have a GROUP BY clause!!

"Persons" table (used in most examples)

Name	Age
Hansen, Ola	34
Svendson, Tove	45
Pettersen, Kari	19

Aggregate functions in MS Access

Function	Description
AVG(column)	Returns the average value of a column
COUNT(column)	Returns the number of rows (without a NULL value) of a column
COUNT(*)	Returns the number of selected rows
FIRST(column)	Returns the value of the first record in a specified field
LAST(column)	Returns the value of the last record in a specified field
MAX(column)	Returns the highest value of a column
MIN(column)	Returns the lowest value of a column
STDEV(column)	
STDEVP(column)	
SUM(column)	Returns the total sum of a column
VAR(column)	

VARP(column)	

Aggregate functions in SQL Server

Function	Description
AVG(column)	Returns the average value of a column
BINARY_CHECKSUM	
CHECKSUM	
CHECKSUM_AGG	
COUNT(column)	Returns the number of rows (without a NULL value) of a column
COUNT(*)	Returns the number of selected rows
COUNT(DISTINCT column)	Returns the number of distinct results
FIRST(column)	Returns the value of the first record in a specified field (not supported in SQLServer2K)
LAST(column)	Returns the value of the last record in a specified field (not supported in SQLServer2K)
MAX(column)	Returns the highest value of a column
MIN(column)	Returns the lowest value of a column
STDEV(column)	
STDEVP(column)	
SUM(column)	Returns the total sum of a column
VAR(column)	
VARP(column)	

Scalar functions

Scalar functions operate against a single value, and return a single value based on the input value.

Useful Scalar Functions in MS Access

Function	Description
UCASE(c)	Converts a field to upper case
LCASE(c)	Converts a field to lower case
MID(c,start[,end])	Extract characters from a text field
LEN(c)	Returns the length of a text field
INSTR(c,char)	Returns the numeric position of a named character within a text field
LEFT(c,number_of_char)	Return the left part of a text field requested
RIGHT(c,number_of_char)	Return the right part of a text field requested
ROUND(c,decimals)	Rounds a numeric field to the number of decimals specified
MOD(x,y)	Returns the remainder of a division operation

Aggregate functions (like SUM) often need an added GROUP BY functionality.

GROUP BY

GROUP BY... was added to SQL because aggregate functions (like SUM) return the aggregate of all column values every time they are called, and without the GROUP BY function it was impossible to find the sum for each individual group of column values.

The syntax for the GROUP BY function is:

SELECT column,SUM(column) FROM table GROUP BY column

GROUP BY Example

This "Sales" Table:

Company	Amount
W3Schools	5500
IBM	4500
W3Schools	7100

And This SQL:

SELECT Company, SUM(Amount) FROM Sales

Returns this result:

Company	SUM(Amount)
W3Schools	17100
IBM	17100
W3Schools	17100

The above code is invalid because the column returned is not part of an aggregate. A GROUP BY clause will solve this problem:

SELECT Company,SUM(Amount) FROM Sales
GROUP BY Company

Returns this result:

Company	SUM(Amount)
W3Schools	12600
IBM	4500

HAVING…

HAVING... was added to SQL because the WHERE keyword could not be used against aggregate functions (like SUM), and without HAVING... it would be impossible to test for result conditions.

The syntax for the HAVING function is:

SELECT column,SUM(column) FROM table
GROUP BY column
HAVING SUM(column) condition value

This "Sales" Table:

Company	Amount
W3Schools	5500
IBM	4500
W3Schools	7100

This SQL:

SELECT Company,SUM(Amount) FROM Sales
GROUP BY Company
HAVING SUM(Amount)>10000

Returns this result

Company	SUM(Amount)
W3Schools	12600

EMBEDDED SQL

Embedded SQL is a method of inserting inline SQL statements or queries into the code of a programming language, which is known as a host language. Because the host language cannot parse SQL, the inserted SQL is parsed by an embedded SQL preprocessor.

Embedded SQL is a robust and convenient method of combining the computing power of a programming language with SQL's specialized data management and manipulation capabilities.

Structure of embedded SQL

Structure of embedded SQL defines step by step process of establishing a connection with DB and executing the code in the DB within the high level language.

Connection to DB

This is the first step while writing a query in high level languages. First connection to the DB that we are accessing needs to be established. This can be done using the keyword CONNECT. But it has to precede with 'EXEC SQL' to

indicate that it is a SQL statement.

EXEC SQL CONNECT db_name;

EXEC SQL CONNECT HR_USER; //connects to DB HR_USER

Once connection is established with DB, we can perform DB transactions. Since these DB transactions are dependent on the values and variables of the host language. Depending on their values, query will be written and executed. Similarly, results of DB query will be returned to the host language which will be captured by the variables of host language. Hence we need to declare the variables to pass the value to the query and get the values from query. There are two types of variables used in the host language.

- **Host variable :** These are the variables of host language used to pass the value to the query as well as to capture the values returned by the query. Since SQL is dependent on host language we have to use variables of host language and such variables are known as host variable. But these host variables should be declared within the SQL area or within SQL code. That means compiler should be able to differentiate it from normal C variables. Hence we have to declare host variables within BEGIN DECLARE and END DECLARE section. Again, these declare block should be enclosed within EXEC SQL and ';'.

EXEC SQL **BEGIN** DECLARE SECTION;

    int STD_ID;

    char STD_NAME [15];

    char ADDRESS[20];

EXEC SQL **END** DECLARE SECTION;

We can note here that variables are written inside begin and end block of the SQL, but they are declared using C code. It does not use SQL code to declare the variables. Why? This is because they are host variables – variables of C language. Hence we cannot use SQL syntax to declare them. Host language supports almost all the datatypes from int, char, long, float, double, pointer, array, string, structures etc.

When host variables are used in a SQL query, it should be preceded by colon – ':' to indicate that it is a host variable. Hence when pre-compiler compiles SQL code, it substitutes the value of host variable and compiles.

EXEC SQL SELECT * FROM STUDENT WHERE STUDENT_ID =:STD_ID;

The following code is a simple embedded SQL program, written in C. The program illustrates many, but not all, of the embedded SQL techniques. The program prompts the user for an order number, retrieves the customer number, salesperson, and status of the order, and displays the retrieved information on the screen.

```
int main() {
  EXEC SQL INCLUDE SQLCA;
  EXEC SQL BEGIN DECLARE SECTION;
    int OrderID;      /* Employee ID (from user)      */
    int CustID;       /* Retrieved customer ID        */
    char SalesPerson[10]   /* Retrieved salesperson name     */
    char Status[6]        /* Retrieved order status       */
  EXEC SQL END DECLARE SECTION;

  /* Set up error processing */
  EXEC SQL WHENEVER SQLERROR GOTO query_error;
  EXEC SQL WHENEVER NOT FOUND GOTO bad_number;

  /* Prompt the user for order number */
  printf ("Enter order number: ");
  scanf_s("%d", &OrderID);

  /* Execute the SQL query */
  EXEC SQL SELECT CustID, SalesPerson, Status
    FROM Orders
    WHERE OrderID = :OrderID
    INTO :CustID, :SalesPerson, :Status;
```

```
            /* Display the results */
            printf ("Customer number:  %d\n", CustID);
            printf ("Salesperson: %s\n", SalesPerson);
            printf ("Status: %s\n", Status);
            exit();

        query_error:
            printf ("SQL error: %ld\n", sqlca->sqlcode);
            exit();

        bad_number:
            printf ("Invalid order number.\n");
            exit();
        }
```

DYNAMIC SQL

The main disadvantage of embedded SQL is that it supports only static SQLs. If we need to build up queries at run time, then we can use dynamic sql. That means if query changes according to user input, then it always better to use dynamic SQL. Like we said above, the query when user enters student name alone and when user enters both student name and address, is different. If we use embedded SQL, one cannot implement this requirement in the code. In such case dynamic SQL helps the user to develop query depending on the values entered by him, without making him know which query is being executed. It can also be used when we do not know which SQL statements like Insert, Delete update or select needs to be used, when number of host variables is unknown, or when datatypes of host variables are unknown or when there is direct reference to DB objects like tables, views, indexes are required.

However this will make user requirement simple and easy but it may make query lengthier and complex. That means depending upon user inputs, the query may grow or shrink making the code flexible enough to handle all the possibilities. In embedded SQL, compiler knows the query in advance and pre-compiler compiles the SQL code much before C compiles the code for execution. Hence embedded SQLs will be faster in execution. But in the case of dynamic SQL, queries are created, compiled and executed only at the run time. This makes the dynamic SQL little complex, and time consuming.

Since query needs to be prepared at run time, in addition to the structures discussed in embedded SQL, we have three more clauses in dynamic SQL. These are mainly used to build the query and execute them at run time.

PREPARE

Since dynamic SQL builds a query at run time, as a first step we need to capture all the inputs from the user. It will be stored in a string variable. Depending on the inputs received from the user, string variable is appended with inputs and SQL keywords. These SQL like string statements are then converted into SQL query. This is done by using PREPARE statement.

For example, below is the small snippet from dynamic SQL. Here sql_stmt is a character variable, which holds inputs from the users along with SQL commands. But is cannot be considered as SQL query as it is still a sting value. It needs to be converted into a proper SQL query which is done at the last line using PREPARE statement. Here sql_query is also a string variable, but it holds the string as a SQL query.

EXECUTE

This statement is used to compile and execute the SQL statements prepared in DB.

EXEC SQL EXECUTE sql_query;

EXECUTE IMMEDIATE

This statement is used to prepare SQL statement as well as execute the SQL statements in DB. It performs the task of PREPARE and EXECUTE in a single line.

EXEC SQL EXECUTE IMMEDIATE :sql_stmt;

Dynamic SQL will not have any SELECT queries and host variables. But it can be any other SQL statements like insert, delete, update, grant etc. But when we use insert/ delete/ updates in this type, we cannot use host variables. All the input values will be hardcoded. Hence the SQL statements can be directly executed using EXECUTE IMMEDIATE rather than using PREPARE and then EXECUTE.

```
EXEC SQL EXECUTE IMMEDIATE  'GRANT SELECT ON STUDENT TO Faculty';
EXEC SQL EXECUTE IMMEDIATE  'DELETE FROM STUDENT WHERE STD_ID = 100';
EXEC SQL EXECUTE IMMEDIATE  'UPDATE STUDENT SET ADDRESS = 'Troy' WHERE STD_ID =100';
```

UNIT II	DATABASE DESIGN

Entity-Relationship model – E-R Diagrams – Enhanced-ER Model – ER-to-Relational Mapping – Functional Dependencies – Non-loss Decomposition – First, Second, Third Normal Forms, Dependency Preservation – Boyce/Codd Normal Form – Multi-valued Dependencies and Fourth Normal Form – Join Dependencies and Fifth Normal Form

DATABASE DESIGN

A well-designed database shall:

- Eliminate Data Redundancy: the same piece of data shall not be stored in more than one place. This is because duplicate data not only waste storage spaces but also easily lead to inconsistencies.
- Ensure Data Integrity and Accuracy

Entity-Relationship Data Model

- Classical, popular conceptual data model
- First introduced (mid 70's) as a (relatively minor) improvement to the relational model: pictorial diagrams are easier to read than relational database schemas
- Then evolved as a popular model for the first conceptual representation of data structures in the process of database design

ER Model: Entity and Entity Set

Considering the above example, **Student** is an entity, **Teacher** is an entity, similarly, **Class**, **Subject**etc are also entities.

An Entity is generally a real-world object which has characteristics and holds relationships in a DBMS.

If a Student is an Entity, then the complete dataset of all the students will be the **Entity Set**

ER Model: Attributes

If a Student is an Entity, then student's roll no., student's name, student's age, student's gender etc will be its attributes.

An attribute can be of many types, here are different types of attributes defined in ER database model:

1. **Simple attribute**: The attributes with values that are atomic and cannot be broken down further are simple attributes. For example, student's age.
2. **Composite attribute**: A composite attribute is made up of more than one simple attribute. For example, student's address will contain, house no., street name, pincode etc.
3. **Derived attribute**: These are the attributes which are not present in the whole database management system, but are derived using other attributes. For example, average age of students in a class.
4. **Single-valued attribute**: As the name suggests, they have a single value.
5. **Multi-valued attribute**: And, they can have multiple values.

ER Model: Relationships

When an Entity is related to another Entity, they are said to have a relationship. For example, A ClassEntity is related to Student entity, because students study in classes, hence this is a relationship.

Depending upon the number of entities involved, a degree is assigned to relationships.

For example, if 2 entities are involved, it is said to be Binary relationship, if 3 entities are involved, it is said to be Ternary relationship, and so on.

Working with ER Diagrams

ER Diagram is a visual representation of data that describes how data is related to each other. In ER Model, we disintegrate data into entities, attributes and setup relationships between entities, all this can be represented visually using the ER diagram.

Components of ER Diagram

Entitiy, Attributes, Relationships etc form the components of ER Diagram and there are defined symbols and shapes to represent each one of them.
Let's see how we can represent these in our ER Diagram.

Entity

Simple rectangular box represents an Entity.

Relationships between Entities - Weak and Strong

Rhombus is used to setup relationships between two or more entities.
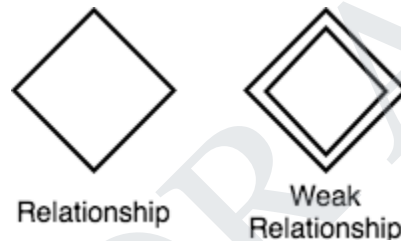
Attributes for any Entity

Ellipse is used to represent attributes of any entity. It is connected to the entity.
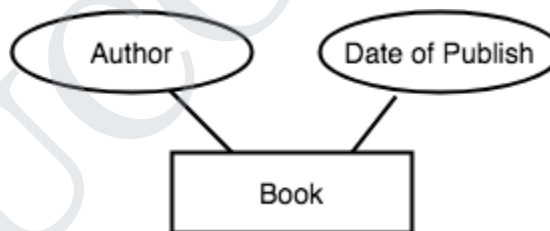
Weak Entity

A weak Entity is represented using double rectangular boxes. It is generally connected to another entity.

Key Attribute for any Entity

To represent a Key attribute, the attribute name inside the Ellipse is underlined.

Derived Attribute for any Entity

Derived attributes are those which are derived based on other attributes, for example, age can be derived from date of birth.

To represent a derived attribute, another dotted ellipse is created inside the main ellipse.

Multivalued Attribute for any Entity

Double Ellipse, one inside another, represents the attribute which can have multiple values.

Composite Attribute for any Entity

A composite attribute is the attribute, which also has attributes.
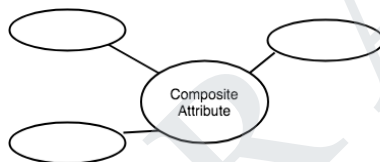
ER Diagram: Entity

An Entity can be any object, place, person or class. In ER Diagram, an entity is represented using rectangles. Consider an example of an Organisation- Employee, Manager, Department, Product and many more can be taken as entities in an Organisation.

The yellow rhombus in between represents a relationship.

ER Diagram: Key Attribute

Key attribute represents the main characteristic of an Entity. It is used to represent a Primary key. Ellipse with the text underlined, represents Key Attribute.
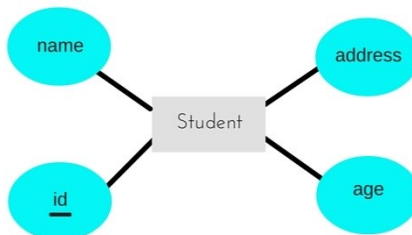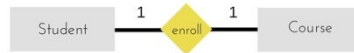
ER Diagram: Binary Relationship
Binary Relationship means relation between two Entities. This is further divided into three types.

One to One Relationship
This type of relationship is rarely seen in real world.

The above example describes that one student can enroll only for one course and a course will also have only one Student. This is not what you will usually see in real-world relationships.

One to Many Relationship
The below example showcases this relationship, which means that 1 student can opt for many courses, but a course can only have 1 student. Sounds weird! This is how it is.

Many to One Relationship
It reflects business rule that many entities can be associated with just one entity. For example, Student enrolls for only one Course but a Course can have many Students.

Many to Many Relationship
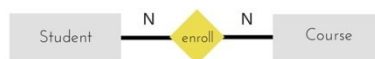
The above diagram represents that one student can enroll for more than one courses. And a course can have more than 1 student enrolled in it.
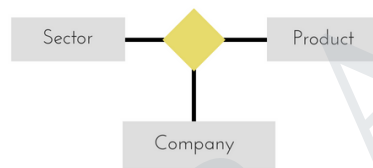
ER Diagram: Recursive Relationship

When an Entity is related with itself it is known as Recursive Relationship.
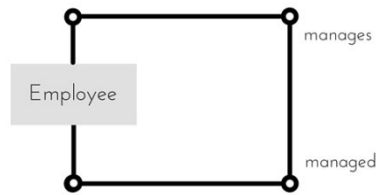
ER Diagram: Ternary Relationship

Relationship of degree three is called Ternary relationship.

A Ternary relationship involves three entities. In such relationships we always consider two entites together and then look upon the third.

- The above relationship involves 3 entities.
- Company operates in Sector, producing some Products.

For example, in the diagram above, we have three related entities, Company, Product and Sector. To understand the relationship better or to define rules around the model, we should relate two entities and then derive the third one.

A Company produces many Products/ each product is produced by exactly one company.

A Company operates in only one Sector / each sector has many companies operating in it.

Considering the above two rules or relationships, we see that although the complete relationship involves three entities, but we are looking at two entities at a time.

The Enhanced ER Model

As the complexity of data increased in the late 1980s, it became more and more difficult to use the traditional ER Model for database modelling. Hence some improvements or enhancements were made to the existing ER Model to make it able to handle the complex applications better.

Hence, as part of the Enhanced ER Model, along with other improvements, three new concepts were added to the existing ER Model, they were:
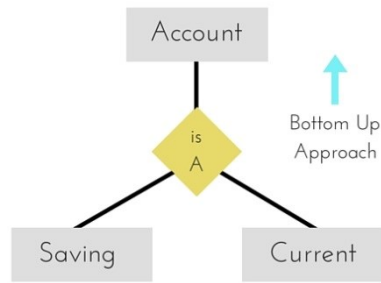
1. Generalization
2. Specialization
3. Aggregration

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 entities to make further higher level entity.

It's more like Superclass and Subclass system, but the only difference is the approach, which is bottom-up. Hence, entities are combined to form a more generalised entity, in other words, sub-classes are combined to form a super-

STUCOR APP

class.

For example, Saving and Current account types entities can be generalised and an entity with name Account can be created, which covers both.

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, a higher level entity may not have any lower-level entity sets, it's possible.
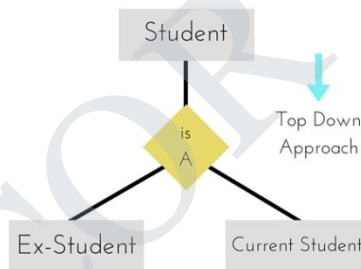
Aggregation

Aggregation is a process when relation between two entities is treated as a single entity.
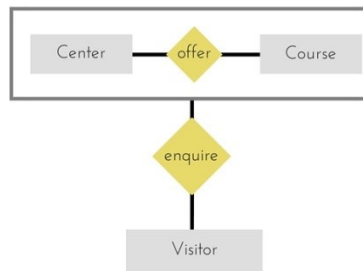
In the diagram above, the relationship between Center and Course together, is acting as an Entity, which is in relationship with another entity Visitor. Now in real world, if a Visitor or a Student visits a Coaching Center, he/she will never enquire about the center only or just about the course, rather he/she will ask enquire about both.

ER Model to Relational Model

ER Model can be represented using ER Diagrams which is a great way of designing and representing the database design in more of a flow chart form.

It is very convenient to design the database using the ER Model by creating an ER diagram and later on converting it into relational model to design your tables.

Not all the ER Model constraints and components can be directly transformed into relational model, but an approximate schema can be derived.

Few examples of ER diagrams and convert it into relational model schema, hence creating tables in RDBMS.
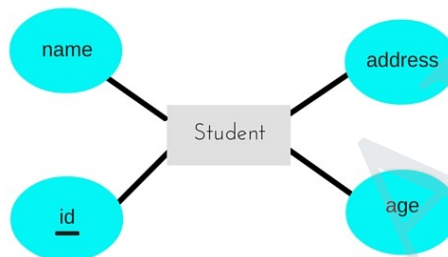
Entity becomes Table

Entity in ER Model is changed into tables, or we can say for every Entity in ER model, a table is created in Relational Model.

And the attributes of the Entity gets converted to columns of the table.

And the primary key specified for the entity in the ER model, will become the primary key for the table in relational model.

For example, for the below ER Diagram in ER Model,

A   table   with   name Student will   be   created   in   relational   model,   which   will   have   4 columns, id, name, age, address and id will be the primary key for this table.

Table:Student

id	name	age	address

Relationship becomes a Relationship Table

In ER diagram, we use diamond/rhombus to represent a relationship between two entities. In Relational model we create a relationship table for ER Model relationships too.

In the ER diagram below, we have two entities Teacher and Student with a relationship between them.

As discussd above, entity gets mapped to table, hence we will create table for Teacher and a table for Student with all the attributes converted into columns.

Now, an additional table will be created for the relationship, for example StudentTeacher or give it any name you like. This table will hold the primary key for both Student and Teacher, in a tuple to describe the relationship, which teacher teaches which student.

If there are additional attributes related to this relationship, then they become the columns for this table, like subject

St. Joseph's College of Engineering/St. Joseph's Institute of

name.
Also proper foreign key constraints must be set for all the tables.

Functional Dependency

The functional dependency is a relationship that exists between two attributes. It typically exists between the primary key and non-key attribute within a table.

X → Y

The left side of FD is known as a determinant, the right side of the production is known as a dependent.
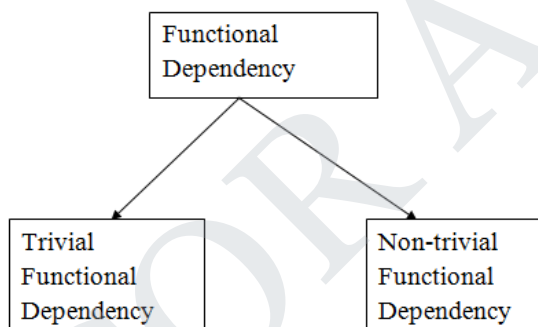
For example:

Assume we have an employee table with attributes: Emp_Id, Emp_Name, Emp_Address.

Here Emp_Id attribute can uniquely identify the Emp_Name attribute of employee table because if we know the Emp_Id, we can tell that employee name associated with it.

Functional dependency can be written as:

$$Emp\_Id \rightarrow Emp\_Name$$

Types of Functional dependency

Trivial functional dependency

- o A → B has trivial functional dependency if B is a subset of A.
- o The following dependencies are also trivial like: A → A, B → B

Example:

Consider a table with two columns Employee_Id and Employee_Name.

{Employee_id, Employee_Name} → Employee_Id is a trivial functional dependency as Employee_Id is a subset of {Employee_Id, Employee_Name}.

3. Also, Employee_Id → Employee_Id and Employee_Name → Employee_Name are trivial dependencies too.

Non-trivial functional dependency

A → B has a non-trivial functional dependency if B is not a subset of A.

When A intersection B is NULL, then A → B is called as complete non-trivial.

Example:

$$ID \quad \rightarrow \quad Name,$$
$$Name \quad \rightarrow \quad DOB$$

Normalization of Database

Database Normalization is a technique of organizing the data in the database. Normalization is a systematic

approach of decomposing tables to eliminate data redundancy(repetition) and undesirable characteristics like Insertion, Update and Deletion anomalies. It is a multi-step process that puts data into tabular form, removing duplicated data from the relation tables.

Normalization is used for mainly two purposes,

- Eliminating reduntant(useless) data.
- Ensuring data dependencies make sense i.e data is logically stored.

Problems Without Normalization

If a table is not properly normalized and have data redundancy then it will not only eat up extra memory space but will also make it difficult to handle and update the database, without facing data loss. Insertion, Updation and Deletion Anomalies are very frequent if database is not normalized. To understand these anomalies let us take an example of a Student table.

rollno	name	branch	hod	office_tel
401	Akon	CSE	Mr. X	53337
402	Bkon	CSE	Mr. X	53337
403	Ckon	CSE	Mr. X	53337
404	Dkon	CSE	Mr. X	53337

In the table above, we have data of 4 Computer Sci. students. As we can see, data for the fields branch, hod(Head of Department) and office_tel is repeated for the students who are in the same branch in the college, this is Data Redundancy.

Insertion Anomaly

Suppose for a new admission, until and unless a student opts for a branch, data of the student cannot be inserted, or else we will have to set the branch information as NULL.

Also, if we have to insert data of 100 students of same branch, then the branch information will be repeated for all those 100 students.

These scenarios are nothing but Insertion anomalies.

Updation Anomaly

What if Mr. X leaves the college? or is no longer the HOD of computer science department? In that case all the student records will have to be updated, and if by mistake we miss any record, it will lead to data inconsistency. This is Updation anomaly.

Deletion Anomaly

In our Student table, two different informations are kept together, Student information and Branch information. Hence, at the end of the academic year, if student records are deleted, we will also lose the branch information. This is Deletion anomaly.

Normalization Rule
Normalization rules are divided into the following normal forms:
1. First Normal Form
2. Second Normal Form
3. Third Normal Form
4. BCNF
5. Fourth Normal Form
6. Fifth Normal Form

First Normal Form (1NF)
For a table to be in the First Normal Form, it should follow the following 4 rules:
1. It should only have single(atomic) valued attributes/columns.
2. Values stored in a column should be of the same domain
3. All the columns in a table should have unique names.
4. And the order in which data is stored, does not matter.

Rules for First Normal Form
The first normal form expects you to follow a few simple rules while designing your database, and they are:

Rule 1: Single Valued Attributes
Each column of your table should be single valued which means they should not contain multiple values. We will explain this with help of an example later, let's see the other rules for now.

Rule 2: Attribute Domain should not change
This is more of a "Common Sense" rule. In each column the values stored must be of the same kind or type.
For example: If you have a column dob to save date of births of a set of people, then you cannot or you must not save 'names' of some of them in that column along with 'date of birth' of others in that column. It should hold only 'date of birth' for all the records/rows.

Rule 3: Unique name for Attributes/Columns
This rule expects that each column in a table should have a unique name. This is to avoid confusion at the time of retrieving data or performing any other operation on the stored data.
If one or more columns have same name, then the DBMS system will be left confused.

Rule 4: Order doesn't matters
This rule says that the order in which you store the data in your table doesn't matter.

EXAMPLE
Create a table to store student data which will have student's roll no., their name and the name of subjects they have opted for.
Here is the table, with some sample data added to it.

roll_no	name	subject
101	Akon	OS, CN
103	Ckon	Java
102	Bkon	C, C++

The table already satisfies 3 rules out of the 4 rules, as all our column names are unique, we have stored data in the order we wanted to and we have not inter-mixed different type of data in columns.

But out of the 3 different students in our table, 2 have opted for more than 1 subject. And we have stored the subject names in a single column. But as per the 1st Normal form each column must contain atomic value.
It's very simple, because all we have to do is break the values into atomic values.
Here is our updated table and it now satisfies the First Normal Form.

roll_no	name	subject
101	Akon	OS
101	Akon	CN
103	Ckon	Java
102	Bkon	C
102	Bkon	C++

By doing so, although a few values are getting repeated but values for the subject column are now atomic for each record/row. Using the First Normal Form, data redundancy increases, as there will be many columns with same data in multiple rows but each row as a whole will be unique.

Second Normal Form (2NF)

For a table to be in the Second Normal Form,
1. It should be in the First Normal form.
2. And, it should not have Partial Dependency.

Dependency

Let's take an example of a Student table with columns student_id, name, reg_no(registration number), branch and address(student's home address).

student_id	name	reg_no	branch	address

In this table, student_id is the primary key and will be unique for every row, hence we can use student_id to fetch any row of data from this table
Even for a case, where student names are same, if we know the student_id we can easily fetch the correct record.

student_id	name	reg_no	branch	address
10	Akon	07-WY	CSE	Kerala
11	Akon	08-WY	IT	Gujarat

Hence we can say a Primary Key for a table is the column or a group of columns(composite key) which can uniquely identify each record in the table.

I can ask from branch name of student with student_id 10, and I can get it. Similarly, if I ask for name of student with student_id 10 or 11, I will get it. So all I need is student_id and every other column depends on it, or can be fetched using it. This is Dependency and we also call it Functional Dependency.

Partial Dependency

Now that we know what dependency is, we are in a better state to understand what partial dependency is.
For a simple table like Student, a single column like student_id can uniquely identfy all the records in a table.
But this is not true all the time. So now let's extend our example to see if more than 1 column together can act

as a primary key.

Let's create another table for Subject, which will have subject_id and subject_name fields and subject_id will be the primary key.

subject_id	subject_name
1	Java
2	C++
3	Php

Now we have a Student table with student information and another table Subject for storing subject information.

Let's create another table Score, to store the marks obtained by students in the respective subjects. We will also be saving name of the teacher who teaches that subject along with marks.

score_id	student_id	subject_id	marks	teacher
1	10	1	70	Java Teacher
2	10	2	75	C++ Teacher
3	11	1	80	Java Teacher

In the score table we are saving the **student_id** to know which student's marks are these and **subject_id** to know for which subject the marks are for.

Together, student_id + subject_id forms a **Candidate Key** which can be the **Primary key**.

To get me marks of student with student_id 10, can you get it from this table? No, because you don't know for which subject. And if I give you subject_id, you would not know for which student. Hence we need student_id + subject_id to uniquely identify any row.

But where is Partial Dependency?

Now if you look at the Score table, we have a column names teacher which is only dependent on the subject, for Java it's Java Teacher and for C++ it's C++ Teacher & so on.

Now as we just discussed that the primary key for this table is a composition of two columns which is student_id & subject_id but the teacher's name only depends on subject, hence the subject_id, and has nothing to do with student_id.

This is Partial Dependency, where an attribute in a table depends on only a part of the primary key and not on the whole key.

How to remove Partial Dependency?

There can be many different solutions for this, but out objective is to remove teacher's name from Score table. The simplest solution is to remove columns teacher from Score table and add it to the Subject table. Hence, the Subject table will become:

subject_id	subject_name	teacher
1	Java	Java Teacher
2	C++	C++ Teacher
3	Php	Php Teacher

And our Score table is now in the second normal form, with no partial dependency.

score_id	student_id	subject_id	marks
1	10	1	70
2	10	2	75
3	11		

Third Normal Form (3NF)
A table is said to be in the Third Normal Form when,
1. It is in the Second Normal form.
2. And, it doesn't have Transitive Dependency.

So let's use the same example, where we have 3 tables, **Student**, **Subject** and **Score**.

Student Table

student_id	name	reg_no	branch	address
10	Akon	07-WY	CSE	Kerala
11	Akon	08-WY	IT	Gujarat
12	Bkon	09-WY	IT	Rajasthan

Subject Table

subject_id	subject_name	teacher
1	Java	Java Teacher
2	C++	C++ Teacher
3	Php	Php Teacher

Score Table
In the Score table, we need to store some more information, which is the exam name and total marks, so let's add 2 more columns to the Score table.

score_id	student_id	subject_id	marks
1	10	1	70
2	10	2	75
3	11	1	80

Transitive Dependency
With exam_name and total_marks added to our Score table, it saves more data now. Primary key for the Score table is a composite key, which means it's made up of two attributes or columns → student_id + subject_id.

The new column exam_name depends on both student and subject. For example, a mechanical engineering student will have Workshop exam but a computer science student won't. And for some subjects you have Practical exams and for some you don't. So we can say that exam_name is dependent on both student_id and subject_id.

And what about our second new column total_marks? Does it depend on our Score table's primary key?

Well, the column total_marks depends on exam_name as with exam type the total score changes. For example, practicals are of less marks while theory exams are of more marks.

But, exam_name is just another column in the score table. It is not a primary key or even a part of the primary key, and total_marks depends on it.

This is Transitive Dependency. When a non-prime attribute depends on other non-prime attributes rather than depending upon the prime attributes or primary key.

How to remove Transitive Dependency

Again the solution is very simple. Take out the columns exam_name and total_marks from Score table and put them in an Exam table and use the exam_id wherever required.

Score Table: In 3rd Normal Form

score_id	student_id	subject_id	marks	exam_id

The new Exam table

exam_id	exam_name	total_marks
1	Workshop	200
2	Mains	70
3	Practicals	30

Advantage of removing Transitive Dependency

The advantage of removing transitive dependency is,

- Amount of data duplication is reduced.
- Data integrity achieved.

Boyce and Codd Normal Form (BCNF)

Boyce and Codd Normal Form is a higher version of the Third Normal form. This form deals with certain type of anomaly that is not handled by 3NF. A 3NF table which does not have multiple overlapping candidate keys is said to be in BCNF. For a table to be in BCNF, following conditions must be satisfied:

- R must be in 3rd Normal Form
- and, for each functional dependency (X → Y), X should be a super Key.In simple words, it means, that for a dependency A → B, A cannot be a non-prime attribute, if B is a prime attribute.

Example

College enrolment table with columns student_id, subject and professor.

student_id	subject	professor
101	Java	P.Java
101	C++	P.Cpp
102	Java	P.Java2
103	C#	P.Chash
104	Java	P.Java

In the table above:

One student can enroll for multiple subjects. For example, student with student_id 101, has opted for subjects - Java & C++

- For each subject, a professor is assigned to the student.
- And, there can be multiple professors teaching one subject like Java.

What do you think should be the Primary Key?

Well, in the table above student_id, subject together form the primary key, because using student_id and subject, we can find all the columns of the table.

One more important point to note here is, one professor teaches only one subject, but one subject may have two different professors.

Hence, there is a dependency between subject and professor here, where subject depends on the professor name.

This table satisfies the 1st Normal form because all the values are atomic, column names are unique and all the values stored in a particular column are of same domain.

This table also satisfies the 2nd Normal Form as there is no Partial Dependency.

And, there is no Transitive Dependency, hence the table also satisfies the 3rd Normal Form.

But this table is not in Boyce-Codd Normal Form.

Why this table is not in BCNF?

In the table above, student_id, subject form primary key, which means subject column is a prime attribute.

But, there is one more dependency, professor → subject.

And while subject is a prime attribute, professor is a non-prime attribute, which is not allowed by BCNF.

How to satisfy BCNF?

To make this relation(table) satisfy BCNF, we will decompose this table into two tables, student table and professor table.

Below we have the structure for both the tables.

Student Table

student_id	p_id
101	1
101	2

Professor Table

p_id	professor	subject
1	P.Java	Java
2	P.Cpp	C++

And now, this relation satisfy Boyce-Codd Normal Form.

Fourth Normal Form (4NF)

A table is said to be in the Fourth Normal Form when,

1. It is in the Boyce-Codd Normal Form.
2. And, it doesn't have Multi-Valued Dependency.

Multi-valued Dependency

A table is said to have multi-valued dependency, if the following conditions are true,

1. For a dependency A → B, if for a single value of A, multiple value of B exists, then the table may have multi-valued dependency.
2. Also, a table should have at-least 3 columns for it to have a multi-valued dependency.
3. And, for a relation R(A,B,C), if there is a multi-valued dependency between, A and B, then B and C should be independent of each other.

If all these conditions are true for any relation(table), it is said to have multi-valued dependency.

Example

Below we have a college enrolment table with columns s_id, course and hobby.

s_id	course	hobby
1	Science	Cricket
1	Maths	Hockey
2	C#	Cricket
2	Php	Hockey

From the table above, student with s_id 1 has opted for two courses, Science and Maths, and has two hobbies, Cricket and Hockey.

You must be thinking what problem this can lead to, right?

Well the two records for student with s_id 1, will give rise to two more records, as shown below, because for one student, two hobbies exists, hence along with both the courses, these hobbies should be specified.

s_id	course	hobby
1	Science	Cricket
1	Maths	Hockey
1	Science	Hockey
1	Maths	Cricket

And, in the table above, there is no relationship between the columns course and hobby. They are independent of each other.

So there is multi-value dependency, which leads to un-necessary repetition of data and other anomalies as well.

How to satisfy 4th Normal Form?

To make the above relation satify the 4th normal form, we can decompose the table into 2 tables.

CourseOpted Table

s_id	course
1	Science
1	Maths
2	C#
2	Php

Hobbies Table,

s_id	hobby
1	Cricket
1	Hockey
2	Cricket
2	Hockey

Now this relation satisfies the fourth normal form.

A table can also have functional dependency along with multi-valued dependency. In that case, the functionally dependent columns are moved in a separate table and the multi-valued dependent columns are moved to separate tables.

Fifth Normal Form (5NF)

A database is said to be in 5NF, if and only if,

1. It's in 4NF
2. If we can decompose table further to eliminate redundancy and anomaly, and when we re-join the decomposed tables by means of candidate keys, we should not be losing the original data or any new record set should not arise. In simple words, joining two or more decomposed table should not lose records nor create new records.

What is Join Dependency

If a table can be recreated by joining multiple tables and each of this table have a subset of the attributes of the table, then the table is in Join Dependency. It is a generalization of Multivalued DependencyJoin Dependency can be related to 5NF, wherein a relation is in 5NF, only if it is already in 4NF and it cannot be decomposed further.

Example

<Employee>

EmpName	EmpSkills	EmpJob (Assigned Work)
Tom	Networking	EJ001
Harry	Web Development	EJ002
Katie	Programming	EJ002

The above table can be decomposed into the following three tables; therefore it is not in 5NF:

<EmployeeSkills>

EmpName	EmpSkills
Tom	Networking
Harry	Web Development
Katie	Programming

<EmployeeJob>

EmpName	EmpJob
Tom	EJ001
Harry	EJ002
Katie	EJ002

<JobSkills>

EmpSkills	EmpJob
Networking	EJ001
Web Development	EJ002
Programming	EJ002

Our Join Dependency:

{(EmpName, EmpSkills), (EmpName, EmpJob), (EmpSkills, EmpJob)}

The above relations have join dependency, so they are not in 5NF. That would mean that a join relation of the above three relations is equal to our original relation **<Employee>**.

FIFTH NORMAL FORM EXAMPLE

Consider an example of different Subjects taught by different lecturers and the lecturers taking classes for different semesters.

Note: Please consider that Semester 1 has Mathematics, Physics and Chemistry and Semester 2 has only Mathematics in its academic year!!

COURSE
SUBJECT
LECTURER
CLASS

SUBJECT	LECTURER	CLASS
Mathematics	Alex	SEMESTER 1
Mathematics	Rose	SEMESTER 1
Physics	Rose	SEMESTER 1
Physics	Joseph	SEMESTER 2
Chemistry	Adam	SEMESTER 1

In above table, Rose takes both Mathematics and Physics class for Semester 1, but she does not take Physics class for Semester 2. In this case, combination of all these 3 fields is required to identify a valid data. Imagine we want to add a new class - Semester3 but do not know which Subject and who will be taking that subject. We would be simply inserting a new entry with Class as Semester3 and leaving Lecturer and subject as NULL. As we discussed above, it's not a good to have such entries. Moreover, all the three columns together act as a primary key, we cannot leave other two columns blank!

Hence we have to decompose the table in such a way that it satisfies all the rules till 4NF and when join them by using keys, it should yield correct record. Here, we can represent each lecturer's Subject area and their classes in a better way. We can divide above table into three - (SUBJECT, LECTURER), (LECTURER, CLASS), (SUBJECT, CLASS)

				5NF

SUBJECT	LECTURER		CLASS	LECTURER
Mathematics	Alex		SEMESTER 1	Alex
Mathematics	Rose		SEMESTER 1	Rose
Physics	Rose		SEMESTER 1	Rose
Physics	Joseph		SEMESTER 2	Joseph
Chemistry	Adam		SEMESTER 1	Adam

CLASS	SUBJECT
SEMESTER 1	Mathematics
SEMESTER 1	Physics
SEMESTER 1	Chemistry
SEMESTER 2	Physics

Now, each of combinations is in three different tables. If we need to identify who is teaching which subject to which semester, we need join the keys of each table and get the result.

For example, who teaches Physics to Semester 1, we would be selecting Physics and Semester1 from table 3 above, join with table1 using Subject to filter out the lecturer names. Then join with table2 using Lecturer to get correct lecturer name. That is we joined key columns of each table to get the correct data. Hence there is no lose or new data - satisfying 5NF condition.

UNIT III - TRANSACTIONS

Transaction Concepts – ACID Properties – Schedules – Serializability – Concurrency Control – Need for Concurrency – Locking Protocols – Two Phase Locking – Deadlock – Transaction Recovery - Save Points – Isolation Levels – SQL Facilities for Concurrency and Recovery

1. TRANSACTION CONCEPTS

What is Transaction?

A set of logically related operations is known as transaction. The main operations of a transaction are:

Read(A): Read operations Read(A) or R(A) reads the value of A from the database and stores it in a buffer in main memory.

Write (A): Write operation Write(A) or W(A) writes the value back to the database from buffer.

Let us take a debit transaction from an account which consists of following operations:

1.R(A);
2.A=A-1000;
3.W(A);

Assume A's value before starting of transaction is 5000.

- The first operation reads the value of A from database and stores it in a buffer.
- Second operation will decrease its value by 1000. So buffer will contain 4000.
- Third operation will write the value from buffer to database. So A's final value will be 4000.

But it may also be possible that transaction may fail after executing some of its operations. The failure can be because of **hardware, software or power** etc. For example, if debit transaction discussed above fails after executing operation 2, the value of A will remain 5000 in the database which is not acceptable by the bank.

To avoid this, Database has two important operations:

Commit: After all instructions of a transaction are successfully executed, the changes made by transaction are made permanent in the database.

Rollback: If a transaction is not able to execute all operations successfully, all the changes made by transaction are undone.

Why do you need concurrency in Transactions?

A database is a shared resource accessed. It is used by many users and processes concurrently. For example, the banking system, railway, and air reservations systems, stock market monitoring, supermarket inventory, and checkouts, etc.

Not managing concurrent access may create issues like:

- Hardware failure and system crashes
- Concurrent execution of the same transaction, deadlock, or slow performance

States of Transactions

The various states of a Database Transaction are listed below

Prepared By: Mrs. E. Ajitha (AP/CSE)

State	Transaction types
Active State	A transaction enters into an active state when the execution process begins. During this state read or write operations can be performed.
Partially Committed	A transaction goes into the partially committed state after the end of a transaction.
Committed State	When the transaction is committed to state, it has already completed its execution successfully. Moreover, all of its changes are recorded to the database permanently.
Failed State	A transaction considers failed when any one of the checks fails or if the transaction is aborted while it is in the active state.
Terminated State	State of transaction reaches terminated state when certain transactions which are leaving the system can't be restarted.
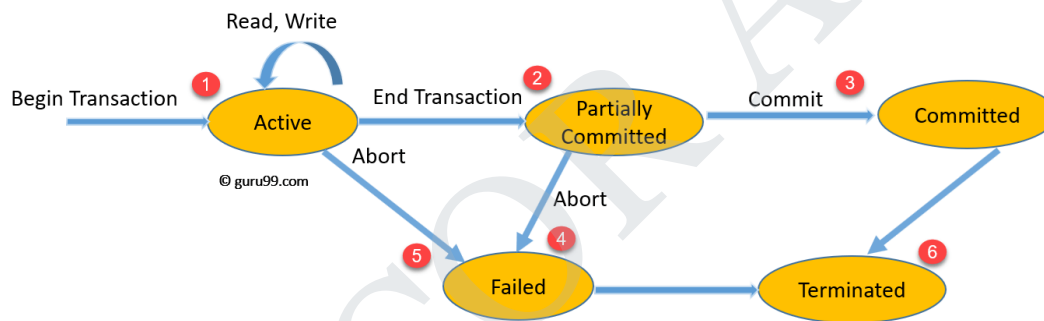
Transition Diagram for a Database Transaction

Let's study a state transition diagram that highlights how a transaction moves between these various states.

1. Once a transaction states execution, it becomes active. It can issue READ or WRITE operation.
2. Once the READ and WRITE operations complete, the transactions becomes partially committed state.
3. Next, some recovery protocols need to ensure that a system failure will not result in an inability to record changes in the transaction permanently. If this check is a success, the transaction commits and enters into the committed state.
4. If the check is a fail, the transaction goes to the Failed state.
5. If the transaction is aborted while it's in the active state, it goes to the failed state. The transaction should be rolled back to undo the effect of its write operations on the database.
6. The terminated state refers to the transaction leaving the system.

2. 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 transaction, certain properties are followed. These are called **ACID properties.**

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 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'.

Consider the following transaction T consisting of T1 and T2: Transfer of 100 from account X to account Y.

Before: X : 500	Y: 200
Transaction T	
T1	T2
Read (X)	Read (Y)
X: = X − 100	Y: = Y + 100
Write (X)	Write (Y)
After: X : 400	Y : 300

If the transaction fails after completion of T1 but before completion of T2.(say, after write(X) but before write(Y)), then amount has been deducted from X but not added to Y. This results in an inconsistent database state. Therefore, the transaction must be executed in entirety in order to ensure correctness of database state.

Consistency

This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to **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.

Isolation

This property ensures that multiple transactions can occur concurrently without leading to 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.

Let X= 500, Y = 500.

Consider two transactions T and T".

Prepared By: Mrs. E. Ajitha (AP/CSE)

T	T''
Read (X)	Read (X)
X: = X*100	Read (Y)
Write (X)	Z: = X + Y
Read (Y)	Write (Z)
Y: = Y − 50	
Write	

Suppose T has been executed till Read (Y) and then T'' starts. As a result , interleaving of operations takes place due to which T'' reads correct value of X but incorrect value of Y and sum computed by
T'': (X+Y = 50, 000+500=50, 500)
is thus not consistent with the sum at end of transaction:
T: (X+Y = 50, 000 + 450 = 50, 450).

This results in database inconsistency, due to a loss of 50 units. Hence, transactions must take place in isolation and changes should be visible only after a they have been made to the main memory.

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 system failure occurs. These updates now become permanent and are stored in a non-volatile memory. The effects of the transaction, thus, are never lost.

3. SCHEDULES

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 T1 and T2.

T1	T2
R(A)	
W(A)	
R(B)	
	W(B)
	R(A)
	R(B)

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 T1 —> T2

2. Complete Schedules

Schedules in which the last operation of each transaction is either abort (or) commit are called complete schedules.

Prepared By: Mrs. E. Ajitha (AP/CSE)

Example: Consider the following schedule involving three transactions T1, T2 and T3.

T1	T2	T3
R(A)		
	W(A)	
R(B)		
		W(B)
commit		
	commit	
		abort

This is a complete schedule since the last operation performed under every transaction is either "commit" or "abort".

3. 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 Tj is reading value updated or written by some other transaction Ti, then the commit of Tj must occur after the commit of Ti.

Example – Consider the following schedule involving two transactions T1 and T2.

T1	T2
R(A)	
W(A)	
	W(A)
	R(A)
commit	
	commit

This is a recoverable schedule since T1 commits before T2, that makes the value read by T2 correct.

4. Cascadeless Schedules –

Also called Avoids **cascading aborts/rollbacks (ACA).** Schedules in which transactions read values only after all transactions whose changes they are going to read commit are called **cascadeless schedules.** Avoids that a single transaction abort leads to a series of **transaction rollbacks.** A strategy to prevent cascading aborts is to disallow a transaction from reading uncommitted changes from another transaction in the same schedule.

In other words, if some transaction Tj wants to read value updated or written by some other transaction Ti, then the commit of Tj must read it after the commit of Ti.

Example: Consider the following schedule involving two transactions T1 and T2.

Prepared By: Mrs. E. Ajitha (AP/CSE)

T1	T2
R(A)	
W(A)	
	W(A)
commit	
	R(A)
	commit

This schedule is cascadeless. Since the updated value of A is read by T2 only after the updating transaction i.e. T1 commits.

5. Strict Schedules

A schedule is strict if for any two transactions Ti, Tj, if a write operation of Ti precedes a conflicting operation of Tj (either read or write), then the commit or abort event of Ti also precedes that conflicting operation of Tj.

In other words, Tj can read or write updated or written value of Ti only after Ti commits/aborts.

Example: Consider the following schedule involving two transactions T1 and T2.

T1	T2
R(A)	
	R(A)
W(A)	
commit	
	W(A)
	R(A)
	commit

This is a strict schedule since T2 reads and writes A which is written by T1 only after the commit of T1.

Note – It can be seen that:

1. Cascadeless schedules are stricter than recoverable schedules or are a subset of recoverable schedules.
2. Strict schedules are stricter than cascadeless schedules or are a subset of cascadeless schedules.
3. Serial schedules satisfy constraints of all recoverable, cascadeless and strict schedules and hence is a subset of strict schedules.

4. SERIALIZABILITY

When multiple transactions are running concurrently then there is a possibility that the database may be left in an inconsistent state. **Serializability** is a concept that helps us to check which schedules are serializable. A serializable schedule is the one that always leaves the database in consistent state.

Prepared By: Mrs. E. Ajitha (AP/CSE)

What is a serializable schedule?

A serializable schedule always leaves the database in consistent state. A serial schedule is always a serializable schedule because in serial schedule, a transaction only starts when the other transaction finished execution. However, a non-serial schedule needs to be checked for Serializability.

Types of Serializability

There are two types of Serializability.

     I.   Conflict Serializability

    II.   View Serializability

1. Conflict Serializability

Conflict Serializability is one of the type of Serializability, which can be used to check whether a non-serial schedule is conflict serializable or not.

What is Conflict Serializability?

A schedule is called conflict serializable if we can convert it into a serial schedule after swapping its non-conflicting operations.

Conflicting operations

Two operations are said to be in conflict, if they satisfy all the following three conditions:

1. Both the operations should belong to different transactions.

2. Both the operations are working on same data item.

3. At least one of the operations is a write operation.

Let's see some examples to understand this:

Example 1: Operation W(X) of transaction T1 and operation R(X) of transaction T2 are conflicting operations, because they satisfy all the three conditions mentioned above. They belong to different transactions, they are working on same data item X, one of the operations in write operation.

Example 2: Similarly, Operations W(X) of T1 and W(X) of T2 are conflicting operations.

Example 3: Operations W(X) of T1 and W(Y) of T2 are non-conflicting operations because both the write operations are not working on same data item so these operations don't satisfy the second condition.

Example 4: Similarly, R(X) of T1 and R(X) of T2 are non-conflicting operations because none of them is write operation.

Example 5: Similarly, W(X) of T1 and R(X) of T1 are non-conflicting operations because both the operations belong to same transaction T1.

Conflict Equivalent Schedules

Two schedules are said to be conflict Equivalent if one schedule can be converted into other schedule after swapping non-conflicting operations.

Conflict Serializable check

Let's check whether a schedule is conflict serializable or not. If a schedule is conflict Equivalent to its serial schedule then it is called **Conflict Serializable schedule**. Let's take few examples of schedules.

Example of Conflict Serializability

Let's consider this schedule:

```
T1      T2
-----   ------
R(A)
R(B)
```

7

```
      R(A)
      R(B)
      W(B)
W(A)
```

To convert this schedule into a serial schedule we must have to swap the R(A) operation of transaction T2 with the W(A) operation of transaction T1. However, we cannot swap these two operations because they are conflicting operations, thus we can say that this given schedule is **not Conflict Serializable.**

Let's take another example:

```
T1       T2
-----    ------
R(A)
         R(A)
         R(B)
         W(B)
R(B)
W(A)
```

Let's **swap non-conflicting operations:**

After swapping R(A) of T1 and R(A) of T2 we get:

```
T1       T2
-----    ------
         R(A)
R(A)
         R(B)
         W(B)
R(B)
W(A)
```

After swapping R(A) of T1 and R(B) of T2 we get:

```
T1       T2
-----    ------
         R(A)
         R(B)
R(A)
         W(B)
R(B)
W(A)
```

After swapping R(A) of T1 and W(B) of T2 we get:

```
T1       T2
-----    ------
         R(A)
         R(B)
         W(B)
R(A)
R(B)
W(A)
```

We finally got a serial schedule after swapping all the non-conflicting operations so we can say that the given schedule is **Conflict Serializable.**

8   Prepared By: Mrs. E. Ajitha (AP/CSE)

2. View Serializability

View Serializability is a process to find out that a given schedule is view serializable or not.
To check whether a given schedule is view serializable, we need to check whether the given schedule is **View Equivalent** to its serial schedule. Lets take an example to understand what I mean by that.
Given Schedule:

T1	T2
R(X)	
W(X)	
	R(X)
	W(X)
R(Y)	
W(Y)	
	R(Y)
	W(Y)

Serial Schedule of the above given schedule:

As we know that in Serial schedule a transaction only starts when the current running transaction is finished. So the serial schedule of the above given schedule would look like this:

T1	T2
R(X)	
W(X)	
R(Y)	
W(Y)	
	R(X)
	W(X)
	R(Y)
	W(Y)

If we can prove that the given schedule is **View Equivalent** to its serial schedule then the given schedule is called **view Serializable**.

Testing for Serializability

Example 3.5.6 Consider the following schedules. The actions are listed in the order they are scheduled, and prefixed with the transaction name.

S1 : T1 : R(X),   T2 : R(X),   T1 : W(Y),   T2 : W(Y) T1 : R(Y),   T2 : R(Y)

S2 : T3 : W(X),   T1 : R(X),   T1 : W(Y),   T2 : R(Z), T2 : W(Z)   T3 : R(Z)

For each of the schedules, answer the following questions :

i) What is the precedence graph for the schedule ?

ii) Is the schedule conflict-serializable ? If so, what are all the conflict equivalent serial schedules ?

iii) Is the schedule view-serializable ? If so, what are all the view equivalent serial schedules ?

AU : May-15, Marks 2 + 7 + 7

Prepared By: Mrs. E. Ajitha (AP/CSE)

Solution : i) We will find **conflicting operations.** Two operations are called as conflicting operations if all the following conditions hold true for them-

Both the **operations belong to different transactions.**

Both the operations are on **same data item.**

At least one of the two operations is a write operation

For S1: From above given example in the top to bottom scanning we find the conflict as

$T1 : W(Y), T2 : W(Y)$ and

$T2 : W(Y), T1 : R(Y)$

Hence we will build the precedence graph. Draw the edge between conflicting transactions. For example in above given scenario, the conflict occurs while moving from $T_1:W(Y)$ to $T_2:W(Y)$. Hence edge must be from T_1 to T_2. Similarly for second conflict, there will be the edge from T2 to T1
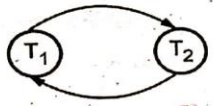
Fig. 3.5.6 Precedence graph for S1

For S2: The conflicts are

o $T3 : W(X), \quad T1 : R(X)$

o $T2 : W(Z) \quad T3 : R(Z)$

Hence the precedence graph is as follows –
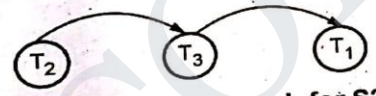
Fig. 3.5.7 Precedence graph for S2

(ii)

o S1 is **not conflict-serializable** since the dependency graph has a cycle.

o S2 is conflict-serializable as the dependency graph is acyclic. The order T2-T3-T1 is the only equivalent serial order.

(iii)

o S1 is **not view serializable.**

o S2 is trivially view-serializable as it is conflict serializable. The **only serial order allowed is**

T2-T3-T1.

5. CONCURRENCY CONTROL

In the concurrency control, the multiple transactions can be executed simultaneously. It may affect the transaction result. It is highly important to maintain the order of execution of those transactions.

6. NEED FOR CONCURRENCY

Problems of concurrency control

Several problems can occur when concurrent transactions are executed in an uncontrolled manner. **Following are the three problems in concurrency control.**

10   Prepared By: Mrs. E. Ajitha (AP/CSE)

- **Lost updates**
- **Dirty read**
- **Unrepeatable read**

1. Lost update problem

When two transactions that access the same database items contain their operations in a way that makes the value of some database item incorrect, then the lost update problem occurs.

If two transactions T1 and T2 read a record and then update it, then the effect of updating of the first record will be overwritten by the second update.

Example:

Transaction-X	Time	Transaction-Y
——	t1	——
Read A	t2	——
——	t3	Read A
Update A	t4	——
——	t5	Update A
——	t6	——

Here,

- o   At time t2, transaction-X reads A's value.

- o   At time t3, Transaction-Y reads A's value.

- o   At time t4, Transactions-X writes A's value on the basis of the value seen at time t2.

- o   At time t5, Transactions-Y writes A's value on the basis of the value seen at time t3.

- o   So at time T5, the update of Transaction-X is lost because Transaction y overwrites it without looking at its current value.

- o   Such type of problem is known as Lost Update Problem as update made by one transaction is lost here.

2. Dirty Read

- o   The dirty read occurs in the case when one transaction updates an item of the database, and then the transaction fails for some reason. The updated database item is accessed by another transaction before it is changed back to the original value.

- o   A transaction T1 updates a record which is read by T2. If T1 aborts then T2 now has values which have never formed part of the stable database.

Example:

Transaction-X	Time	Transaction-Y
——	t1	——
——	t2	Update A
Read A	t3	——
——	t4	Rollback
——	t5	——

- o   At time t2, transaction-Y writes A's value.

- o   At time t3, Transaction-X reads A's value.

11   Prepared By: Mrs. E. Ajitha (AP/CSE)

o   At time t4, Transactions-Y rollbacks. So, it changes A's value back to that of prior to t1.

o   So, Transaction-X now contains a value which has never become part of the stable database.

o   Such type of problem is known as Dirty Read Problem, as one transaction reads a dirty value which has not been committed.

3. Inconsistent Retrievals Problem

o   Inconsistent Retrievals Problem is also known as unrepeatable read. When a transaction calculates some summary function over a set of data while the other transactions are updating the data, then the Inconsistent Retrievals Problem occurs.

o   A transaction T1 reads a record and then does some other processing during which the transaction T2 updates the record. Now when the transaction T1 reads the record, then the new value will be inconsistent with the previous value.

Example:

Suppose two transactions operate on three accounts.

Account-1	Account-2	Account-3
Balance = 200	Balance = 250	Balance = 150

Transaction-X	Time	Transaction-Y
───	t1	───
Read Balance of Acc-1 sum <-- 200 Read Balance of Acc-2	t2	───
Sum <-- Sum + 250 = 450	t3	───
───	t4	Read Balance of Acc-3
───	t5	Update Balance of Acc-3 150 --> 150 - 50 --> 100
───	t6	Read Balance of Acc-1
───	t7	Update Balance of Acc-1 200 --> 200 + 50 --> 250
Read Balance of Acc-3	t8	COMMIT
Sum <-- Sum + 250 = 550	t9	───

o   Transaction-X is doing the sum of all balance while transaction-Y is transferring an amount 50 from Account-1 to Account-3.

o   Here, transaction-X produces the result of 550 which is incorrect. If we write this produced result in the database, the database will become an inconsistent state because the actual sum is 600.

o   Here, transaction-X has seen an inconsistent state of the database.

Prepared By: Mrs. E. Ajitha (AP/CSE)

Concurrency Control Protocol

Concurrency control protocols ensure atomicity, isolation, and serializability of concurrent transactions. The concurrency control protocol can be divided into three categories:

1.   Lock based protocol
2.   Time-stamp protocol

7. LOCKING PROTOCOLS

Lock-Based Protocol

In this type of protocol, any transaction cannot read or write data until it acquires an appropriate lock on it. There are two types of lock:

1. Shared lock:

o   It is also known as a Read-only lock. In a shared lock, the data item can only read by the transaction.

o   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.

2. Exclusive lock:

o   In the exclusive lock, the data item can be both reads as well as written by the transaction.

o   This lock is exclusive, and in this lock, multiple transactions do not modify the same data simultaneously.

8. TWO PHASE LOCKING (2PL)

The two-phase locking protocol divides the execution phase of the transaction into three parts.

o   In the first part, when the execution of the transaction starts, it seeks permission for the lock it requires.

o   In the second part, the transaction acquires all the locks. The third phase is started as soon as the transaction releases its first lock.

o   In the third phase, the transaction cannot demand any new locks. It only releases the acquired locks.

Prepared By: Mrs. E. Ajitha (AP/CSE)
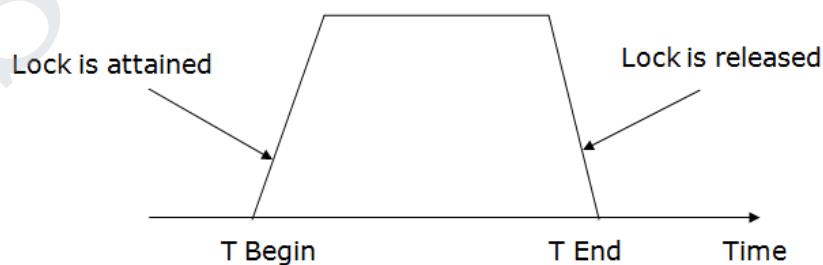
There are two phases of 2PL:

- **Growing phase:** In the growing phase, a new lock on the data item may be acquired by the transaction, but none can be released.

- **Shrinking phase:** In the shrinking phase, existing lock held by the transaction may be released, but no new locks can be acquired.

In the below example, if lock conversion is allowed then the following phase can happen:

1. Upgrading of lock (from S(a) to X (a)) is allowed in growing phase.

2. Downgrading of lock (from X(a) to S(a)) must be done in shrinking phase.

Example:

	T1	T2
0	LOCK-S(A)	
1		LOCK-S(A)
2	LOCK-X(B)	
3	—	—
4	UNLOCK(A)	
5		LOCK-X(C)
6	UNLOCK(B)	
7		UNLOCK(A)
8		UNLOCK(C)
9	—	—

The following way shows how unlocking and locking work with 2-PL.

Transaction T1:
- o **Growing phase:** from step 1-3
- o **Shrinking phase:** from step 5-7
- o **Lock point**: at 3

Transaction T2:
- o **Growing phase:** from step 2-6
- o **Shrinking phase:** from step 8-9
- o **Lock point:** at 6

Types of Two Phase Locking (2PL)

1. **Strict Two-phase locking (Strict-2PL)**

   o The first phase of Strict-2PL is similar to 2PL. In the first phase, after acquiring all the locks, the transaction continues to execute normally.

   o The only difference between 2PL and strict 2PL is that Strict-2PL does not release a lock after using it.

14   Prepared By: Mrs. E. Ajitha (AP/CSE)

o   Strict-2PL waits until the whole transaction to commit, and then it releases all the locks at a time.

o   Strict-2PL protocol does not have shrinking phase of lock release.
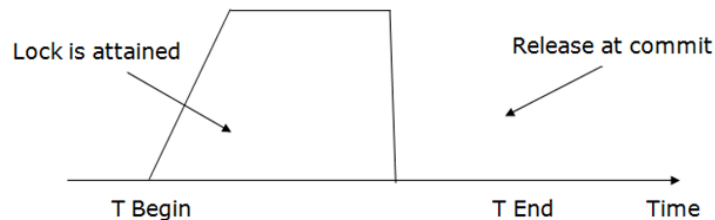
It does not have cascading abort as 2PL does.

2. **Rigorous Two-Phase Locking**

- Rigorous Two – Phase Locking Protocol avoids cascading rollbacks.
- This protocol requires that all the share and exclusive locks to be held until the transaction commits.

Timestamp Ordering Protocol

o   The Timestamp Ordering Protocol is used to order the transactions based on their Timestamps. The order of transaction is nothing but the ascending order of the transaction creation.

o   The priority of the older transaction is higher that's why it executes first. To determine the timestamp of the transaction, this protocol uses system time or logical counter.

o   The lock-based protocol is used to manage the order between conflicting pairs among transactions at the execution time. But Timestamp based protocols start working as soon as a transaction is created.

o   Let's assume there are two transactions T1 and T2. Suppose the transaction T1 has entered the system at 007 times and transaction T2 has entered the system at 009 times. T1 has the higher priority, so it executes first as it is entered the system first.

o   The timestamp ordering protocol also maintains the timestamp of last 'read' and 'write' operation on a data.

Basic Timestamp ordering protocol works as follows:

1. Check the following condition whenever a transaction Ti issues a **Read (X)** operation:

o   If $W\_TS(X) > TS(Ti)$ then the operation is rejected.

o   If $W\_TS(X) <= TS(Ti)$ then the operation is executed.

o   Timestamps of all the data items are updated.

2. Check the following condition whenever a transaction Ti issues a **Write(X)** operation:

o   If $TS(Ti) < R\_TS(X)$ then the operation is rejected.

o   If $TS(Ti) < W\_TS(X)$ then the operation is rejected and Ti is rolled back otherwise the operation is executed.

Prepared By: Mrs. E. Ajitha (AP/CSE)

Where,

TS(TI) denotes the timestamp of the transaction Ti.

R_TS(X) denotes the Read time-stamp of data-item X.

W_TS(X) denotes the Write time-stamp of data-item X.

Thomas write Rule

Thomas Write Rule provides the guarantee of serializability order for the protocol. It improves the Basic Timestamp Ordering Algorithm.

The basic Thomas write rules are as follows:

o   If $TS(T) < R\_TS(X)$ then transaction T is aborted and rolled back, and operation is rejected.

o   If $TS(T) < W\_TS(X)$ then don't execute the W_item(X) operation of the transaction and continue processing.

o   If neither condition 1 nor condition 2 occurs, then allowed to execute the WRITE operation by transaction Ti and set $W\_TS(X)$ to $TS(T)$.

Problems

Example 3.10.2 *Consider the following two transactions :*

T1:read(A)

Read(B);

If A=0 then B=B+1;

Write(B)

T2:read(B); read(A)

If B=0 then A=A+1

Write(A)

Add lock and unlock instructions to transactions T1 and T2, so that they observe two phase locking protocol. Can the execution of these transactions result in deadlock ?

AU : Dec.-16, Marks 6

Prepared By: Mrs. E. Ajitha (AP/CSE)

Solution :

T1	T2
Lock-S(A)	Lock-S(B)
Read(A)	Read(B)
Lock-X(B)	Lock-X(A)
Read(B)	Read(A)
if A=0 then B=B+1	if B=0 then A=A+1
Write(B)	Write(A)
Unlock(A)	Unlock(B)
Commit	Commit
Unlock(B)	Unlock(A)

This is lock-unlock instruction sequence help to satisfy the requirements for strict two phase locking for the given transactions.

The execution of these transactions result in deadlock. Consider following partial execution scenario which leads to deadlock.

T1	T2
Lock-S(A)	Lock-S(B)
Read(A)	Read(B)
Lock-X(B)	Lock-X(A)
Now it will wait for T2 to release exclusive lock on A	Now it will wait for T1 to release exclusive lock on B

9. DEADLOCK

A **deadlock** is a condition wherein two or more tasks are waiting for each other in order to be finished but none of the task is willing to give up the resources that other task needs. In this situation no task ever gets finished and is in waiting state forever.

Coffman conditions

Coffman stated four conditions for a deadlock occurrence. A deadlock may occur if all the following conditions holds true.

- **Mutual exclusion condition**: There must be at least one resource that cannot be used by more than one process at a time.
- **Hold and wait condition**: A process that is holding a resource can request for additional resources that are being held by other processes in the system.
- **No preemption condition**: A resource cannot be forcibly taken from a process. Only the process can release a resource that is being held by it.
- **Circular wait condition**: A condition where one process is waiting for a resource that is being held by second process and second process is waiting for third process ….so on and the last process is waiting for the first process. Thus, making a circular chain of waiting.

For example: In the student table, transaction T1 holds a lock on some rows and needs to update some rows in the grade table. Simultaneously, transaction T2 holds locks on some rows in the grade table and needs to update the rows in the Student table held by Transaction T1.

Now, the main problem arises. Now Transaction T1 is waiting for T2 to release its lock and similarly, transaction T2 is waiting for T1 to release its lock. All activities come to a halt state and remain at a standstill. It will remain in a standstill until the DBMS detects the deadlock and aborts one of the transactions.

17   Prepared By: Mrs. E. Ajitha (AP/CSE)
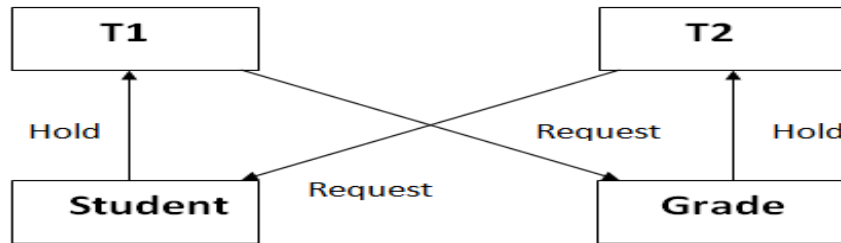
Figure: Deadlock in DBMS

Deadlock Avoidance

- o When a database is stuck in a deadlock state, then it is better to avoid the database rather than aborting or restating the database. This is a waste of time and resource.

- o Deadlock avoidance mechanism is used to detect any deadlock situation in advance. A method like "wait for graph" is used for detecting the deadlock situation but this method is suitable only for the smaller database. For the larger database, deadlock prevention method can be used.

Deadlock Detection

In a database, when a transaction waits indefinitely to obtain a lock, then the DBMS should detect whether the transaction is involved in a deadlock or not. The lock manager maintains a Wait for the graph to detect the deadlock cycle in the database.

Wait for Graph

- o This is the suitable method for deadlock detection. In this method, a graph is created based on the transaction and their lock. If the created graph has a cycle or closed loop, then there is a deadlock.

- o The wait for the graph is maintained by the system for every transaction which is waiting for some data held by the others. The system keeps checking the graph if there is any cycle in the graph.

The wait for a graph for the above scenario is shown below:

Prepared By: Mrs. E. Ajitha (AP/CSE)
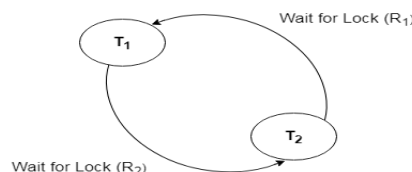
Deadlock Prevention

- o Deadlock prevention method is suitable for a large database. If the resources are allocated in such a way that deadlock never occurs, then the deadlock can be prevented.

- o The Database management system analyzes the operations of the transaction whether they can create a deadlock situation or not. If they do, then the DBMS never allowed that transaction to be executed.

I. Wait-Die scheme

In this scheme, if a transaction requests for a resource which is already held with a conflicting lock by another transaction then the DBMS simply checks the timestamp of both transactions. It allows the older transaction to wait until the resource is available for execution.

Let's assume there are two transactions Ti and Tj and let TS(T) is a timestamp of any transaction T. If T2 holds a lock by some other transaction and T1 is requesting for resources held by T2 then the following actions are performed by DBMS:

1. Check **if TS(Ti) < TS(Tj)** - If Ti is the older transaction and Tj has held some resource, then Ti is allowed to wait until the data-item is available for execution. That means if the older transaction is waiting for a resource which is locked by the younger transaction, then the older transaction is allowed to wait for resource until it is available.

2. **Check if TS(Ti) < TS(Tj)** - If Ti is older transaction and has held some resource and if Tj is waiting for it, then Tj is killed and restarted later with the random delay but with the same timestamp.

II. Wound wait scheme

- o In wound wait scheme, if the older transaction requests for a resource which is held by the younger transaction, then older transaction forces younger one to kill the transaction and release the resource. After the minute delay, the younger transaction is restarted but with the same timestamp.

- o If the older transaction has held a resource which is requested by the Younger transaction, then the younger transaction is asked to wait until older releases it.

Here is the table representation of resource allocation for each algorithm. Both of these algorithms take process age into consideration while determining the best possible way of resource allocation for deadlock avoidance. One of the famous deadlock avoidance algorithm is **Banker's algorithm**

　　　　Prepared By: Mrs. E. Ajitha (AP/CSE)

10. TRANSACTION RECOVERY

3.13 Transaction Recovery

- An integral part of a database system is a recovery scheme that can **restore the database** to the consistent state that existed before the failure.
- The recovery scheme must also provide **high availability**; that means, it must minimize the time for which the database is not usable after a failure.

3.13.1 Failure Classification

Various types of failures are -

1. **Transaction Failure :** Following are two types of errors due to which the transaction gets failed.

   - **Logical Error :**
     i) This error is caused due to internal conditions such as bad input, data not found, overflow of resource limit and so on.

     ii) Due to logical error the transaction can not be continued.

   - **System Error :**
     i) When the system enters in an undesired state and then the transaction can not be continued then this type of error is called as system error.

2) **System Crash :** The situation in which there is a hardware malfunction, or a bug in the database software or the operating system, and because of which there is a loss of the content of volatile storage, and finally the transaction processing come to a halt is called system crash.

3) **Disk Failure :** A disk block loses its content as a result of either a head crash or failure during a data-transfer operation. The backup of data is maintained on the secondary disks or DVD to recover from such failure.

3.13.2 Storage

A DBMS stores the data on external storage because the amount of data is very huge and must persist across program executions.

The storage structure is a memory structure in the system. It has following categories -

1) **Volatile :**

- Volatile memory is a primary memory in the system and is placed along with the CPU.

- These memories can store only small amount of data, but they are very fast. For example - main memory, cache memory.

- A volatile storage cannot survive system crashes.

- That means data in these memories will be lost on failure.

2) Non Volatile :

Non volatile memory is a secondary memory and is huge in size. For example : Hard disk, Flash memory, magnetic tapes.

These memories are designed to withstand system crashes.

3) Stable :

- Information residing in stable storage is never lost.
- To implement stable storage, we replicate the information in several nonvolatile storage media (usually disk) with independent failure modes.

Stable Storage Implementation

- Stable storage is a kind of storage on which the information residing on it is never lost.
- Although stable storage is theoretically impossible to obtain it can be approximately built by applying a technique in which **data loss is almost impossible.**
- That means the information is **replicated** in several nonvolatile storage media with **independent failure modes**.
- Updates must be done with care to ensure that a failure during an update to stable storage **does** not cause a loss of information.

3.13.3 Recovery with Concurrent Transactions

There are four ways for recovery with concurrent transactions :

1) **Interaction with concurrency control :** In this scheme recovery depends upon the concurrency control scheme which is used for the transaction. If a transaction gets failed, then rollback and undo all the updates performed by transaction.

2) **Transaction Rollback :** In this scheme, if the transaction gets failed, then the failed transaction can be rolled back with the help of **log.** The system scans the log backward, and with the help of log entries the system can restore the data items.

3) **Checkpoints :** The checkpoints are used to reduce the number of log records.

4) **Restart recovery :** When the system recovers from the crash it constructs two lists : Undo-list and Redo-list. The undo list consists of transactions to be undone. The redo list consists of transactions to be redone. These two lists are handled as follows

Step 1 : Initially both the lists are empty.

Step 2 : The system scans the log entries from backwards. It scans each entry until it finds first checkpoint record.

21

3.13.4 Shadow Copy Technique

- In the **shadow-copy scheme**, a transaction that wants to update the database first creates a complete **copy of the database.**

  - All updates are done on the new database copy, leaving the original copy, the shadow copy, untouched.

- If at any point the transaction has to be aborted, the system merely deletes the new copy. The old copy of the database has not been affected.

- The current copy of the database is identified by a pointer, called **db-pointer**, which is stored on disk.

- If the transaction **partially commits,** it is committed as follows :

  - First, the operating system is asked to make sure that all pages of the new copy of the database have been written out to disk.

  - After the operating system has written all the pages to disk, the database system updates the pointer **db-pointer** to point to the new copy of the database

  - The new copy then becomes the current copy of the database.

  - The old copy of the database is then deleted.

    - The transaction is said to have been **committed** at the point where the **updated db-pointer** is written to disk.

    - The disk system guarantees that it will update **db-pointer** atomically, as long as we make sure that **db-pointer** lies entirely in a single sector.

Where is shadow copy technique used ?

  - Shadow copy schemes are commonly used by text editors.

  - Shadow copying can be used for small databases.

3.13.5 Log Based Recovery Approach

- Log is the most commonly used structure for recording the modifications that as to be made in the actual database. Hence during the recovery procedure a **log file** is maintained.

- A log record maintains four types of operations. Depending upon the type of operations there are four types of log records-

  1. **<Start>** Log record: It is represented as $<T_i, Start>$

  2. **<Update>** Log record

22

3. **<Commit>** Log record: It is represented as

$<T_i,$ Commit$>$

4. **<Abort>** Log record: It is represented as

$<T_i,$ Abort$>$

- The log contains various fields as shown in following Fig. 3.13.1. This structure is for <update> operation

Transaction ID(T_i)	Data Item Name	Old Value of Data Item	New Value of Data Item

- For example : The sample log file is

$<T_1,$ Start$>$
$< T_1, a, 10, 20>$
$<T_1,$ Commit$>$

→ Here 10 represents the old value before commit operation and 20 is the new value that needs to be updated in the database after commit operation
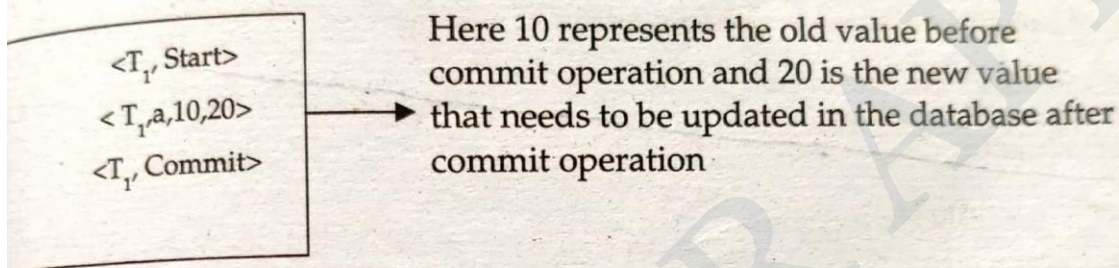
Fig. 3.13.1 Sample Log File

- The log must be maintained on the stable storage and the entries in the log file are maintained before actually updating the physical database.

- There are two approaches used for log based recovery technique - Deferred Database Modification and Immediate Database Modification.

I. Deferred Database Modification :

- In this technique, the database is not updated immediately.

- Only log file is updated on each transaction.

- When the transaction reaches to its commit point, then only the database is physically updated from the log file.

- In this technique, if a transaction fails before reaching to its commit point, it will not have changed database anyway. Hence there is no need for the UNDO operation. The REDO operation is required to record the operations from log file to physical database. Hence deferred database modification technique is also called as **NO UNDO/REDO algorithm.**

- **For example :**

Consider two transactions T_1 and T_2 as follows :

T_1	T_2
Read (A, a)	Read (C, c)
a = a − 10	c = c − 20
Write (A, a)	Write (C, c)
Read (B, b)	
b = b + 10	
Write (B, b)	

If T_1 and T_2 are executed serially with initial values of A = 100, B = 200 and C = 300, then the state of log and database if crash occurs

  a) Just after write (B, b)

  b) Just after write (C, c)

  c) Just after <T_2, commit>

  The result of above 3 scenarios is as follows :

  Initially the log and database will be

Log	Database
<T_1, Start>	
<T_1, A, 90>	
<T_1, B, 210>	
<T_1, Commit>	
	A = 90
	B = 210
<T_2, Start>	
<T_2, C, 280>	
<T_2, Commit>	
	C = 280

a) Just after write (B, b)

Just after write operation, no commit record appears in log. Hence no write operation is performed on database. So database retains only old values. Hence A = 100 and B = 200 respectively.

24   Thus the system comes back to original position and no redo operation take place.

The incomplete transaction of T_1 can be deleted from log.

b) Just after write (C, c)

The state of log records is as follows

Note that crash occurs before T_2 commits. At this point T_1 is completed successfully, so new values of A and B are written from log to database. But as T_2 is not committed, there is no redo (T_2) and the incomplete transaction T_2 can be deleted from log.

The redo (T_1) is done as < T_1, commit> gets executed. Therefore A = 90, B = 210 and C = 300 are the values for database.

c) Just after < T_2, commit>

The log records are as follows :

< T_1, Start>

< T_1, A, 90>

< T_1, B, 210>

< T_1, Commit>

< T_2, Start>

< T_2, 6, 280>

< T_2, Commit>

←——— Crash occurs here

Clearly both T_1 and T_2 reached at commit point and then crash occurs. So both redo (T_1) and redo (T_2) are done and updated values will be A = 90, B = 210, C = 280.

2. Immediate Database Modification :

In this technique, the database is updated during the execution of transaction even before it reaches to its commit point.

If the transaction gets failed before it reaches to its commit point, then the a ROLLBACK Operation needs to be done to bring the database to its earlier consistent state. That means the effect of operations need to be undone on the database. For that purpose both Redo and Undo operations are both required during the recovery. This technique is known as UNDO/ REDO technique.

25

For example : Consider Two transaction T_1 and T_2 as follows :

T_1	T_2
Read(A,a)	Read(C,c)
a=a-10	c=c-20
Write(A,a)	Write(C,c)
Read(B,b)	
b=b+10	
Write(B,b)	

Here T_1 and T_2 are executed serially. Initially A=100, B=200 and C=300

If the crash occurs after

i) Just after Write(B,b)   ii) Just after Write(C,c)   iii) Just after <T_2,Commit>

Then using the immediate Database modification approach the result of above three scenarios can be elaborated as follows :

The contents of **log** and **database** is as follows :

Log	Database
<T_1,Start>	
<T_1,A,100,90>	
	A=90
<T_1,B,200,210>	
	B=210
<T_1,Commit>	
<T_2,Start>	
<T_2,C,300,280>	
	C=280
<T_2,Commit>	

The recovery scheme uses two recovery techniques -

i) **UNDO (T_i)** :The transaction T_i needs to be undone if the log contains <T_i,Start> but does not contain <T_i,Commit>. In this phase, it restores the values of all data items updated by T_i to the old values.

ii) **REDO (T_i)** : The transaction T_i needs to be redone if the log contains both <T_i,Start> and <T_i,Commit>. In this phase, the data item values are set to the new values as per the transaction. After a failure has occurred log record is consulted to determine which transaction need to be redone.

26

a) **Just after Write (B,b)** : When system comes back from this crash, it sees that there is $<T_1,$ Start> but no $<T_1,$ Commit>. Hence T_1 must be undone. That means old values of A and B are restored. Thus old values of A and B are taken from log and both the transaction T_1 and T_2 are re-executed.

b) **Just after Write (C,c):** Here both the redo and undo operations will occur

c) **Undo** : When system comes back from this crash, it sees that there is $<T_2,$ Start> but no $<T_2,$ Commit>. Hence T_2 must be undone. That means old values of C is restored. Thus old value of C is taken from log and the transaction T_2 is re-executed.

d) **Redo** : The transaction T_1 must be done as log contains both the $<T_1,$ Start> and $<T_1,$ Commit>

So A=90, B=210 and C=300

e) **Just after $<T_2,$ Commit>** : When the system comes back from this crash, it sees that there are two transaction T_1 and T_2 with both start and commit points. That means T_1 and T_2 need to be redone. So A=90, B=210 and C=280

3.14 Save Points

The COMMIT, ROLLBACK, and SAVEPOINT are collectively considered as Transaction Commands

(1) **COMMIT** : The COMMIT command is used to save permanently any transaction to database.

When we perform, Read or Write operations to the database then those changes can be undone by rollback operations. To make these changes permanent, we should make use of **commit**

(2) **ROLLBACK** : The ROLLBACK command is used to undo transactions that have not already saved to database. For example

Consider the database table as

RollNo	Name
1	AAA
2	BBB
3	CCC
4	DDD
5	EEE

Fig.3.14.1 Student Table

27

Following command will delete the record from the database, but if we immediately performs ROLLBACK, then this deletion is undone.

For instance –

DELETE FROM Student

WHERE RollNo =2;

ROLLBACK;

Then the resultant table will be

RollNo	Name
1	AAA
2	BBB
3	CCC
4	DDD
5	EEE

(3) SAVEPOINT : A SAVEPOINT is a point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction. The SAVEPOINT can be created as

SAVEPOINT savepoint_name;

Then we can ROLLBACK to SAVEPOIT as

ROLLBACK TO savepoint_name;

For example – Consider Student table as follows –

RollNo	Name
1	AAA
2	BBB
3	CCC
4	DDD
5	EEE

Fig.3.14.2 Student Table

Consider Following commands

SQL> SAVEPOINT S1

28

SQL>DELETE FROM Student

Where RollNo=2;

SQL> SAVEPOINT S2

SQL>DELETE FROM Student

Where RollNo=3;

SQL> SAVEPOINT S3

SQL>DELETE FROM Student

Where RollNo=4

SQL> SAVEPOINT S4

SQL>DELETE FROM Student

Where RollNo=5

SQL> ROLLBACK TO S3;

Then the resultant table will be

RollNo	Name
1	AAA
2	BBB
3	CCC

Thus the effect of deleting the record having RollNo 2, and RollNo3 is undone.

3.15 Isolation Levels

- The consistency of the database is maintained with the help of isolation property(one of the property from ACID properties) of transaction.
- The transaction should take place in a system in such a way that it is the only transaction that is accessing the resources in a database system at particular instance.
- Isolation levels defines the degree to which a transaction must be isolated from the data modifications made by any other transaction in the database system.
- There are four levels of transaction isolation defined by SQL -

  o **Serializable :**
    - This is the **Highest isolation level.**
    - **Serializable execution is defined** to be an execution of operations in which concurrently executing transactions **appears to be serially executing.**

29

- o **Repeatable Read :**
  - ▪ This is the **most restrictive isolation level.**
  - ▪ The transaction **holds read locks on all rows** it references.
  - ▪ It holds **write locks on all rows** it inserts, updates, or deletes.
  - ▪ Since other transaction cannot read, update or delete these rows, it avoids non repeatable read.

- o **Read Committed :**
  - ▪ This isolation level allows only **committed data to be read.**
  - ▪ Thus it does **not allows dirty read** (i.e. one transaction reading of data immediately after written by another transaction).
  - ▪ The transaction hold a **read or write lock** on the current row, and thus prevent other rows from reading, updating or deleting it.

- o **Read Uncommitted :**
  - ▪ It is **lowest isolation level.**
  - ▪ In this level, one transaction may read not yet committed changes made by other transaction.
  - ▪ This level **allows dirty reads**.

  In this level **transactions are not isolated** from each other.

3.16 SQL Facilities for Concurrency and Recovery   AU : May-14, Marks 8

It is possible to achieve concurrency control and recovery using SQL statements. Following is an illustration of implementation of isolation level

The different isolation levels can be set in SQL as follows

(1) READ UNCOMMITTED : It permits dirty reads, nonrepeatable reads, phantoms.

In SQL, we can write

SET TRANSACTION

ISOLATION LEVEL READ UNCOMMITTED

(2) READ COMMITTED : It permits nonrepeatable reads and phantoms, but prohibits dirty reads

In SQL, we can write

SET TRANSACTION

ISOLATION LEVEL READ COMMITTED

(3) REPEATABLE READ : It permits phantoms, but not dirty reads, nor nonrepeatable reads.

30

In SQL, we can write

SET TRANSACTION

ISOLATION LEVEL REPEATABLE READ

(4) **SERIALIZABLE** : In this isolation level there is true two phase locking, and it keeps all transactions serializable.

In SQL, we can write

SET TRANSACTION

ISOLATION LEVEL SERIALIZABLE

For example : Let us discuss the Implementation of READ COMMITTED isolation level using SQL facilities.

Step 1 : Create Table

CREATE TABLE Emp(ID int,Name char(50),Salary int)

Step 2 : Insert values into the table

INSERT INTO Emp(ID,Name,Salary)

VALUES (1,'Sharda',1000)

INSERT INTO Emp(ID,Name,Salary)

VALUES(2,'Ashwini',2000)

INSERT INTO Emp(ID,Name,Salary)

VALUES(3,'Madhura',3000)

The Table will be

ID	Name	Salary
1	Sharda	1000
2	Ashwini	2000
3	Madhura	3000

Step 3 : In select query it will take only commited values of table. If any transaction is opened and incompleted on table in others sessions then select query will wait till no transactions are pending on same table. Read Committed is the default transaction isolation level.

Session 1:

BEGIN TRAN

UPDATE emp SET Salary=999 WHERE ID=1

31

```
WAITFOR DELAY '00:00:15'
COMMIT
```

Session 2 :

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
SELECT Salary FROM emp WHERE ID=1
```

Step 4 : If we run above mentioned two sessions concurrently then

The output will be 999

Step 5 : Following SQL facilities demonstrate the **REPEATABLE READ** isolation level. In this technique, select query data of table that is used under transaction of isolation level "Repeatable Read" can not be modified from any other sessions till transaction is completed.

Assume that we have already created database and inserted some values in it.

Session 1 :

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRAN
SELECT * FROM emp WHERE ID IN(1,2)
WAITFOR DELAY '00:00:15'
SELECT * FROM Emp WHERE ID IN (1,2)
ROLLBACK
```

Session 2 :

```
UPDATE emp SET Salary=999 WHERE ID=1
```

If we run above two sessions concurrently, then Update command in session 2 will wait till session 1 transaction is completed because emp table row with ID=1 has locked in session1 transaction.

Prepared By: Mrs. E. Ajitha (AP/CSE)

UNIT IV	IMPLEMENTATION TECHNIQUES	
RAID – File Organization – Organization of Records in Files – Indexing and Hashing –Ordered Indices – B+ tree Index Files – B tree Index Files – Static Hashing – Dynamic Hashing – Query Processing Overview – Algorithms for SELECT and JOIN operations – Query optimization using Heuristics and Cost Estimation		

RAID

RAID (redundant array of independent disks) originally redundant array of inexpensive disks) is a way of storing the same data in different places on multiple hard disks to protect data in the case of a drive failure.

RAID: Redundant Arrays of Independent Disks

Disk organization techniques that manage a large numbers of disks, providing a view of a single disk of high capacity and high speed by using multiple disks in parallel, and high reliability by storing data redundantly, so that data can be recovered even if a disk fails

Motivation for RAID

- Just as additional memory in form of cache, can improve the system performance, in the same way additional disks can also improve system performance.
- In RAID we can use an array of disks which operates independently since there are many disks, multiple I/O requests can be handled in parallel if the data required is on separate disks
- A single I/O operation can be handled in parallel if the data required is distributed across multiple disks.

Benefits of RAID

- Data loss can be very dangerous for an organization
- RAID technology prevents data loss due to disk failure
- RAID technology can be implemented in hardware or software
- Servers make use of RAID Technology

RAID Level 0- Stripping and non-redundant

- RAID level 0 divides data into block units and writes them across a number of disks. As data is placed across multiple disks it is also called "data Striping".
- The advantage of distributing data over disks is that if different I/O requests are pending for two different blocks of data, then there is a possibility that the requested blocks are on different disks

There is no parity checking of data. So if data in one drive gets corrupted then all the data would be lost. Thus RAID 0 does not support data recovery Spanning is another term that is used with RAID level 0 because the logical disk will span all the physical drives. RAID 0 implementation requires minimum 2 disks.

Advantages

- I/O performance is greatly improved by spreading the I/O load across many channels & drives.
- Best performance is achieved when data is striped across multiple controllers with only one driver per controller

Disadvantages

- It is not fault-tolerant, failure of one drive will result in all data in an array being lost

RAID Level 1: Mirroring (or shadowing)

- Also known as *disk mirroring,* this configuration consists of at least two drives that duplicate the storage of

---

data. There is no striping.
- Read performance is improved since either disk can be read at the same time. Write performance is the same as for single disk storage.
- Every write is carried out on both disks. If one disk in a pair fails, data still available in the other.
- Data loss would occur only if a disk fails, and its mirror disk also fails before the system is repaired Probability of combined event is very small.

RAID 1
Mirroring
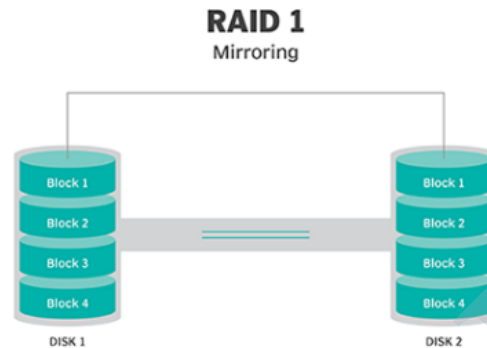
RAID Level 2:

This configuration uses striping across disks, with some disks storing error checking and correcting (ECC) information. It has no advantage over RAID 3 and is no longer used.

RAID 2
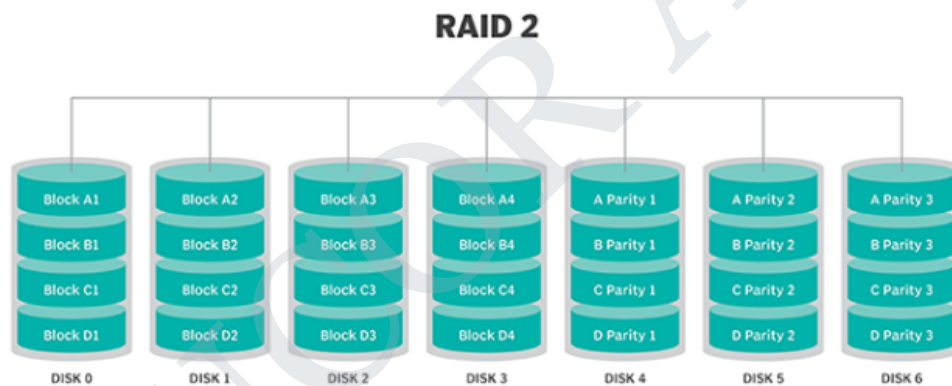
RAID Level 3: Bit-Interleaved Parity
- A single parity bit is enough for error correction, not just detection, since we know which disk has failed
    - When writing data, corresponding parity bits must also be computed and written to a parity bit disk
    - To recover data in a damaged disk, compute XOR of bits from other disks (including parity bit disk)
- I/O operation addresses all the drives at the same time, RAID 3 cannot overlap I/O. For this reason, RAID 3 is best for single-user systems with long record applications.

RAID 3
Parity on separate disk
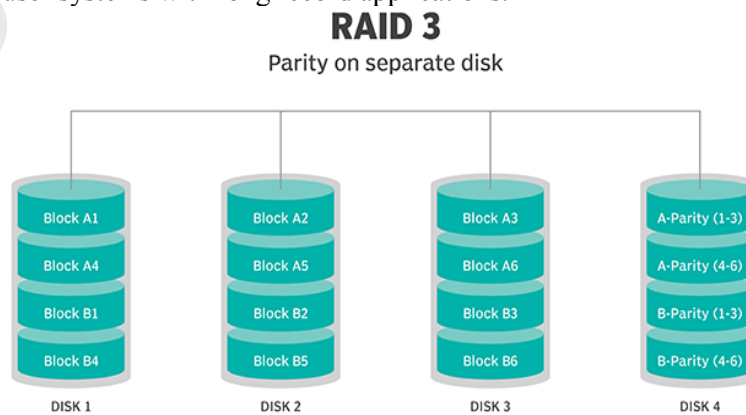
RAID Level 4: Block-Interleaved Parity
- When writing data block, corresponding block of parity bits must also be computed and written to parity disk
- To find value of a damaged block, compute XOR of bits from corresponding blocks (including parity block) from other disks.

RAID 4

DISK 0	DISK 1	DISK 2	DISK 3
Block A1	Block A2	Block A2	Block A-Parity
Block B1	Block B2	Block B2	Block B-Parity
Block C1	Block C2	Block C2	Block C-Parity
Block D1	Block D2	Block D2	Block D-Parity
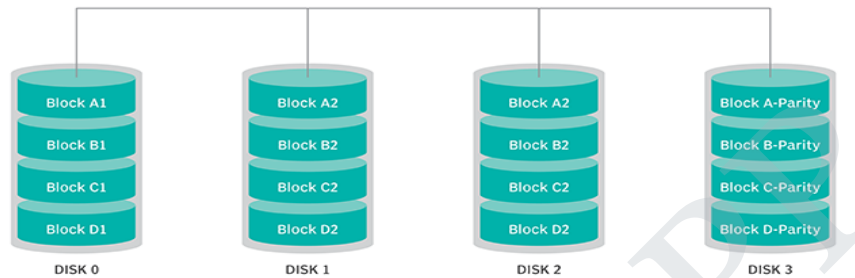
RAID Level 5:
- RAID 5 uses striping as well as parity for redundancy. It is well suited for heavy read and low write operations.
- Block-Interleaved Distributed Parity; partitions data and parity among all $N + 1$ disks, rather than storing data in N disks and parity in 1 disk.

RAID 5

DISK 0	DISK 1	DISK 2	DISK 3
Block A1	Block A2	Block A3	Block A-Parity
Block B1	Block B2	Block B-Parity	Block B3
Block C1	Block C-Parity	Block C2	Block C3
Block Dp	Block D1	Block D2	Block D3
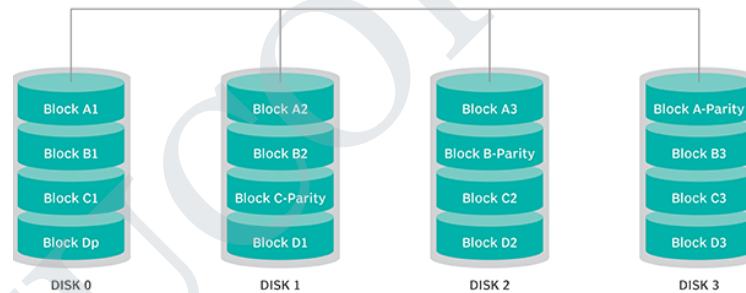
RAID Level 6:
- This technique is similar to RAID 5, but includes a second parity scheme that is distributed across the drives in the array. The use of additional parity allows the array to continue to function even if two disks fail simultaneously. However, this extra protection comes at a cost.
- P+Q Redundancy scheme; similar to Level 5, but stores extra redundant information to guard against multiple disk failures.
  - Better reliability than Level 5 at a higher cost; not used as widely.

RAID 6

DISK 0	DISK 1	DISK 2	DISK 3	DISK 4
Block A1	Block A2	Block A3	Block Ap	Block Aq
Block B1	Block B2	Block Bp	Block Bq	Block B3
Block C1	Block Cp	Block Cq	Block C2	Block C3
Block Dp	Block Dq	Block D1	Block D2	Block D3
Block Eq	Block E1	Block E2	Block E3	Block Ep
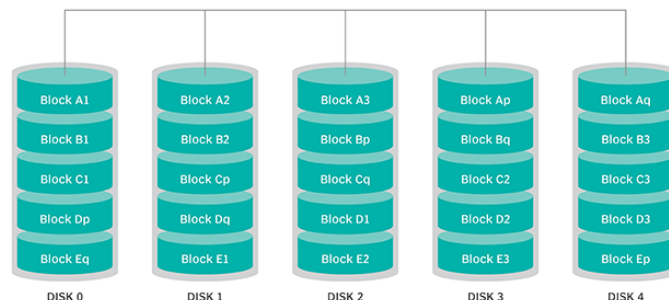
File Organization
- The database is stored as a collection of *files*.
- Each file is a sequence of *records*.
- A record is a sequence of fields.
- Classifications of records
    - Fixed length record
    - Variable length record
- Fixed length record approach:
    Assume record size is fixed each file has records of one particular type only different files are used for different relations

Simple approach
- Record access is simple

Example pseudo code
  type account = record
                account_number char(10);
              branch_name char(22);
               balance numeric(8);
            end
Total bytes 40 for a record

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 2	A-215	Mianus	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

Two problems
    - Difficult to delete record from this structure.
    - Some record will cross block boundaries, that is part of the record will be stored in one block and part in another. It would require two block accesses to read or write

Reuse the free space alternatives:
    - move records $i + 1, \ldots, n$ to n $i, \ldots, n - 1$
    - do not move records, but link all free records on a free list
    - Move the final record to deleted record place.

Free Lists
- Store the address of the first deleted record in the file header.
- Use this first record to store the address of the second deleted record, and so on

header				
record 0	A-102	Perryridge	400	
record 1				
record 2	A-215	Mianus	700	
record 3	A-101	Downtown	500	
record 4				
record 5	A-201	Perryridge	900	
record 6				
record 7	A-110	Downtown	600	
record 8	A-218	Perryridge	700	
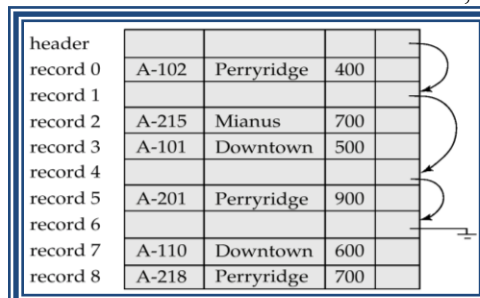
Variable-Length Records
Byte string representation
- Attach an *end-of-record* (⊥) control character to the end of each record
- Difficulty with deletion

0	perryridge	A-102	400	A-201	900	⊥
1	roundhill	A-305	350	⊥		
2	mianus	A-215	700	⊥		

Disadvantage
- It is not easy to reuse space occupied formerly by deleted record.
- There is no space in general for records grows longer

Slotted Page Structure
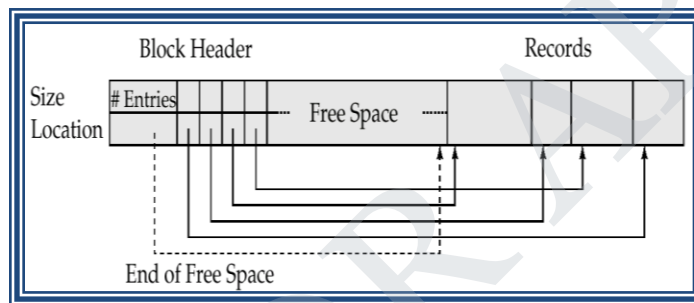
- Slotted page header contains:
  – number of record entries
  – end of free space in the block
  – location and size of each record

Pointer Method
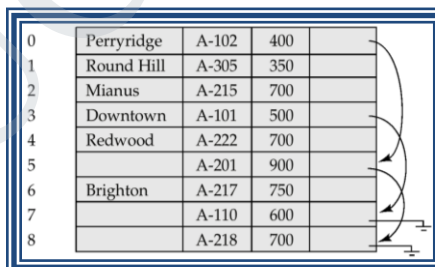
- A variable-length record is represented by a list of fixed-length records, chained together via pointers.
- Can be used even if the maximum record length is not known.

Disadvantage to pointer structure; space is wasted in all records except the first in a a chain.
Solution is to allow two kinds of block in file:
- Anchor block – contains the first records of chain
- Overflow block – contains records other than those that are the first records of chains.
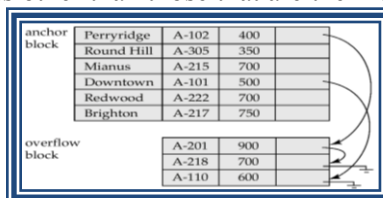
Organization of Records in Files
- Sequential – store records in sequential order, based on the value of the search key of each record
- Heap – a record can be placed anywhere in the file where there is space
- Hashing – a hash function computed on some attribute of each record; the result specifies in which block of the file the record should be placed

Sequential File Organization
- Suitable for applications that require sequential processing of the entire file
- The records in the file are ordered by a search-key

A-217	Brighton	750
A-101	Downtown	500
A-110	Downtown	600
A-215	Mianus	700
A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700
A-222	Redwood	700
A-305	Round Hill	350

A-217	Brighton	750
A-101	Downtown	500
A-110	Downtown	600
A-215	Mianus	700
A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700
A-222	Redwood	700
A-305	Round Hill	350
A-888	North Town	800
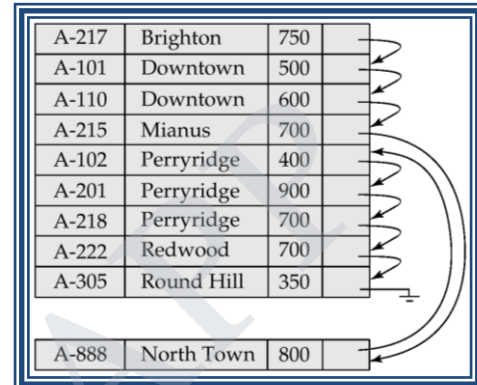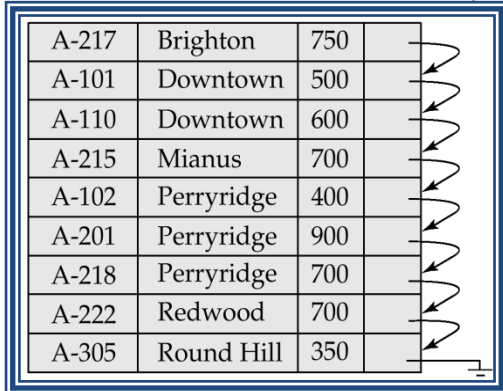
Deletion – use pointer chains

Insertion – locate the position where the record is to be inserted
- if there is free space insert there
- if no free space, insert the record in an overflow block
- In either case, pointer chain must be updated

Indexing and Hashing

Basic Concepts
- Indexing mechanisms used to speed up access to desired data.
    - E.g., author catalog in library
- **Search Key** - attribute to set of attributes used to look up records in a file.

An **index file** consists of records (called **index entries**) of the form

Search-key	pointer

- Index files are typically much smaller than the original file
- Two basic kinds of indices:
    - **Ordered indices:** search keys are stored in sorted order
    - **Hash indices:** search keys are distributed uniformly across "buckets" and by using a "hash function" the values are determined.

Ordered Indices
- In an **ordered index,** index entries are stored sorted on the search key value.
- **Primary index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file.

Types of Ordered Indices
- Dense index
- Sparse index

Dense Index Files

- Dense index — Index record appears for every search-key value in the file.

Brighton			A-217	Brighton	750	
Downtown			A-101	Downtown	500	
Mianus			A-110	Downtown	600	
Perryridge			A-215	Mianus	700	
Redwood			A-102	Perryridge	400	
Round Hill			A-201	Perryridge	900	
			A-218	Perryridge	700	
			A-222	Redwood	700	
			A-305	Round Hill	350	
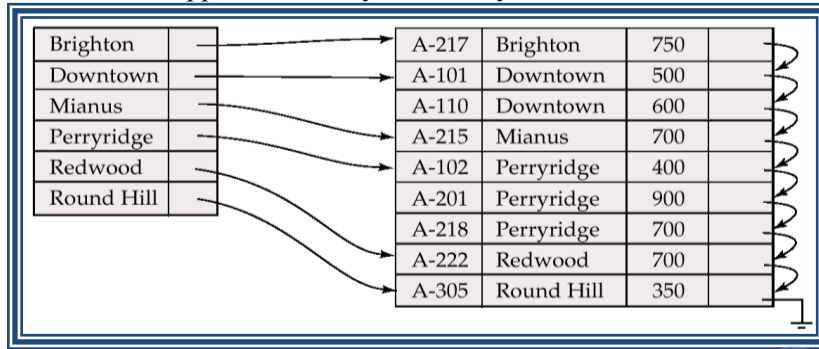
Sparse Index Files
- Sparse Index
       contains index records for only some search-key values.
- To locate a record with search-key value *K* we:
    – Find index record with largest search-key value that is less than or equal to
Search file sequentially starting at the record to which the index record points

Brighton			A-217	Brighton	750	
Mianus			A-101	Downtown	500	
Redwood			A-110	Downtown	600	
			A-215	Mianus	700	
			A-102	Perryridge	400	
			A-201	Perryridge	900	
			A-218	Perryridge	700	
			A-222	Redwood	700	
			A-305	Round Hill	350	
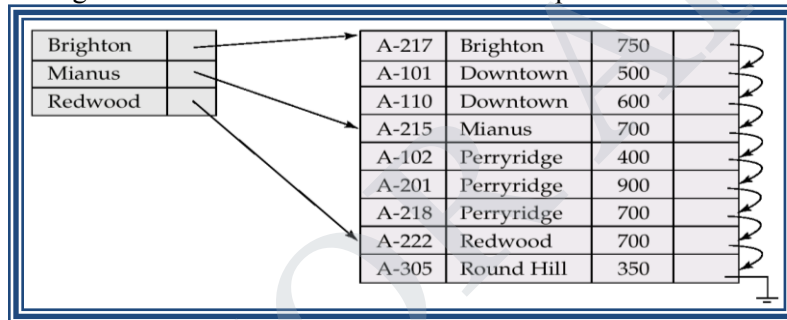
Multilevel Index
- If primary index does not fit in memory, access becomes expensive.
- To reduce number of disk accesses to index records, treat primary index kept on disk as a sequential file and construct a sparse index on it.
    – outer index – a sparse index of primary index
    – inner index – the primary index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
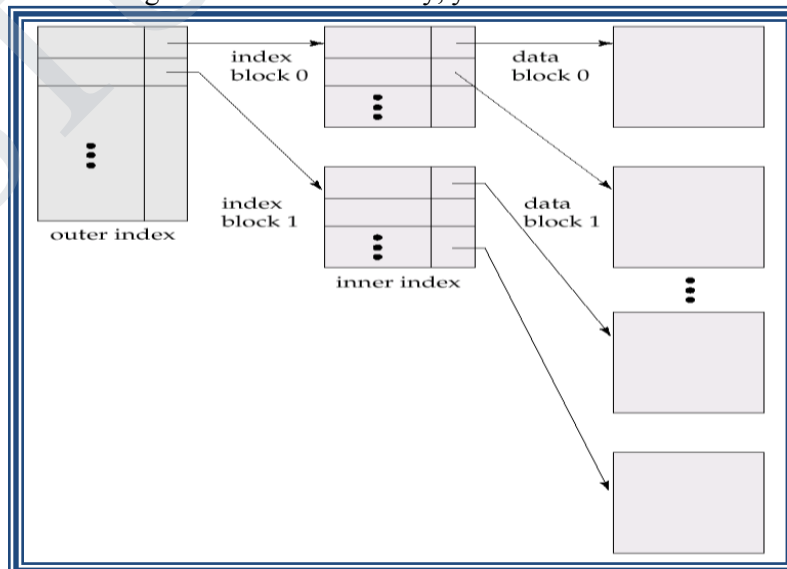
Index Update: Deletion
If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.
- **Single-level index deletion:**
  - Dense indices – deletion of search-key is similar to file record deletion.
  - Sparse indices – if an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order). If the next search-key value already has an index entry, the entry is deleted instead of being replaced.

Index Update: Insertion
- **Single-level index insertion:**
  - Perform a lookup using the search-key value appearing in the record to be inserted.
  - **Dense indices** – if the search-key value does not appear in the index, insert it.
  - **Sparse indices** – if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created. In this case, the first search-key value appearing in the new block is inserted into the index.

Secondary Index on *balance* field of *account*
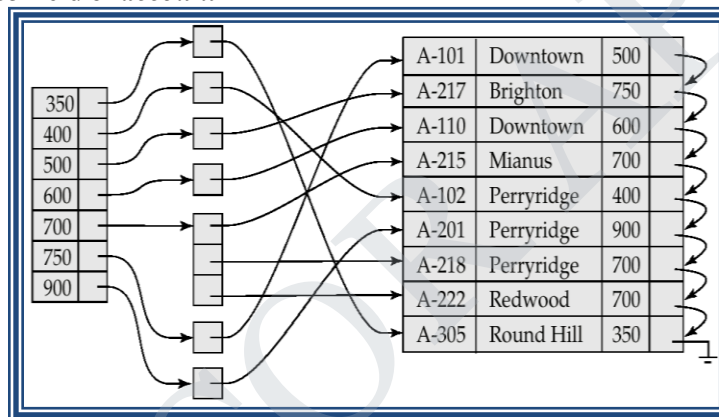
Primary and Secondary Indices
- Secondary indices have to be dense.
- Indices offer substantial benefits when searching for records.
- When a file is modified, every index on the file must be updated, Updating indices imposes overhead on database modification.
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive
  - each record access may fetch a new block from disk

`
B^+**-Tree Index Files**

Example of a B^+-tree : B^+**-tree for *account* file ($n = 3$)**

St. Joseph's College of Engineering/St. Joseph's Institute of Technology   *Page 8*
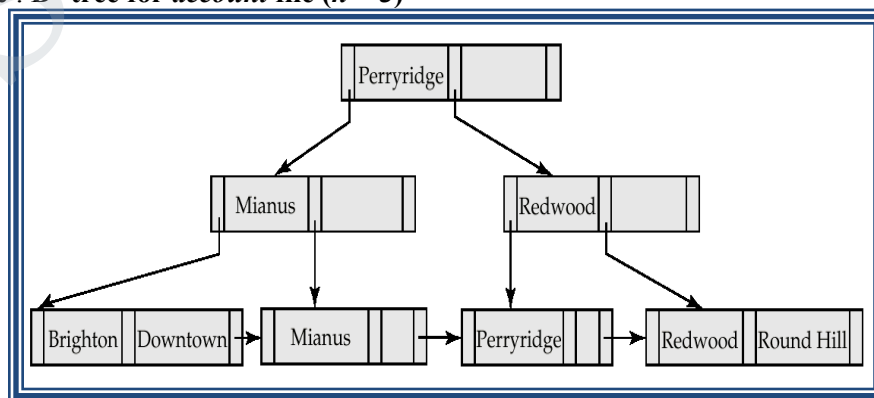
- **Disadvantage of indexed-sequential files**: performance degrades as file grows, since many overflow blocks get created. Periodic reorganization of entire file is required.
- **Advantage of B$^+$-tree index files**: automatically reorganizes itself with small, local, changes, in the face of insertions and deletions. Reorganization of entire file is not required to maintain performance.
- **Disadvantage of B$^+$-trees:** extra insertion and deletion overhead, space overhead.

A B$^+$-tree is a rooted tree satisfying the following properties:
- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf has between [$n/2$] and n children.
- Special cases:
  - If the root is not a leaf, it has at least 2 children.
  - If the root is a leaf, it can have between 0 and (n–1) values.

B$^+$-Tree Node Structure
- **Typical node**

P_1	K_1	P_2	. . .	P_{n-1}	K_{n-1}	P_n

  - K_i are the search-key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered
        $K_1 < K_2 < K_3 < . . . < K_{n-1}$

Properties of Leaf Nodes
- For $i = 1, 2, . . ., n$–1, pointer P_i either points to a file record with search-key value K_i, or to a bucket of pointers to file records, each record having search-key value K_i.
- P_n points to next leaf node in search-key order
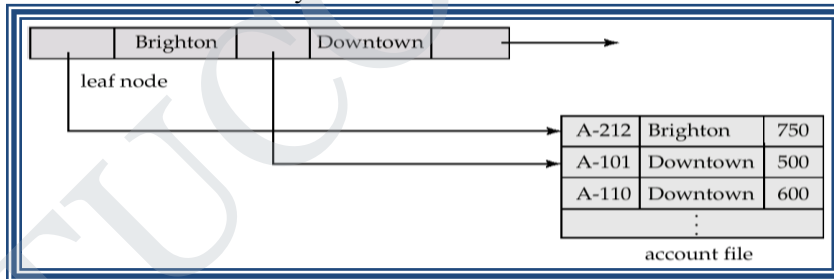
Non-Leaf Nodes in B$^+$-Trees
Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with m pointers:
  - All the search-keys in the subtree to which P_1 points are less than K_1.

P_1	K_1	P_2	. . .	P_{n-1}	K_{n-1}	P_n

Example of a B$^+$-tree: **B$^+$-tree for *account* file ($n = 3$)**
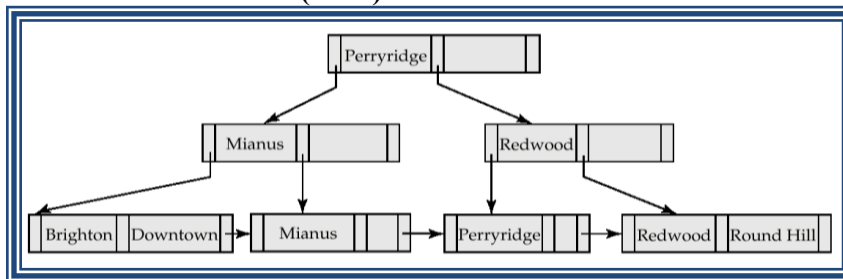
B⁺-tree for *account* file (*n* = 5)
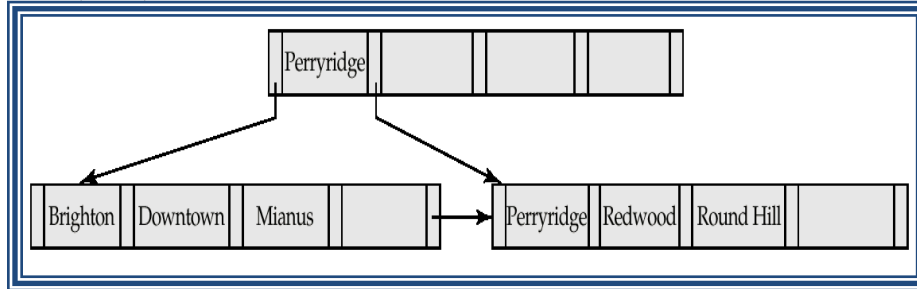
- Non-leaf nodes other than root must have between 3 and 5 children ($\lceil (n/2) \rceil$ and *n* with *n* =5).
- Root must have at least 2 children.

Observations about B⁺-trees
- Since the inter-node connections are done by pointers, "logically" close blocks need not be "physically" close.
- The B⁺-tree contains a relatively small number of levels thus searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently.

Updates on B⁺-Trees: Insertion
- Find the leaf node in which the search-key value would appear
- If the search-key value is already there in the leaf node, record is added to file and if necessary a pointer is inserted into the bucket.
- If the search-key value is not there, then add the record to the main file and create a bucket if necessary. Then:
    - If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node otherwise, split the node.

Example: B⁺-Tree before and after insertion of "Clearview"
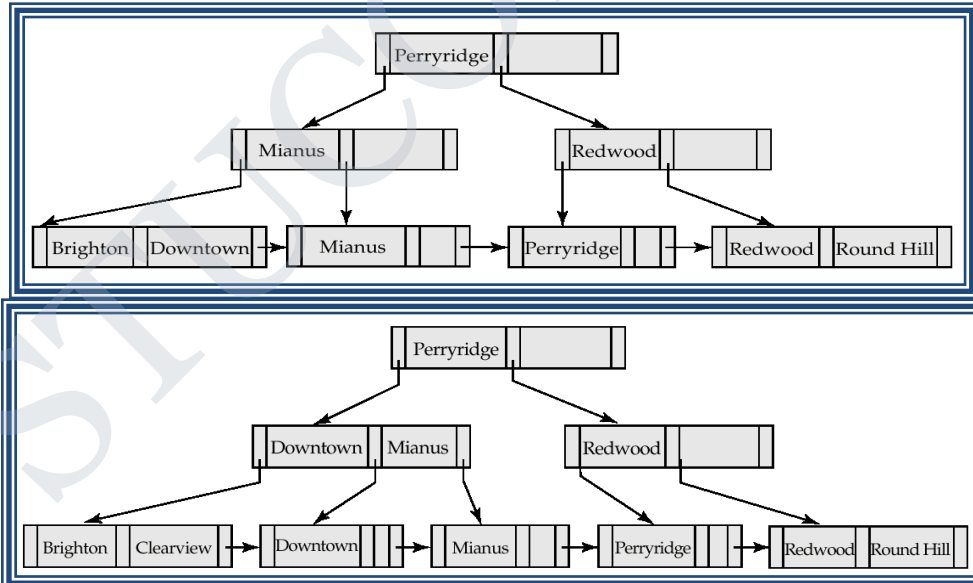
Updates on B⁺-Trees: Deletion
- Find the record to be deleted, and remove it from the main file and from the bucket (if present)
- Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then
    - Insert all the search-key values in the two nodes into a single node (the one on the left), and delete

---

the other node.
- Delete the pair (K_{i-1}, P_i), where P_i is the pointer to the deleted node, from its parent, recursively using the above procedure.
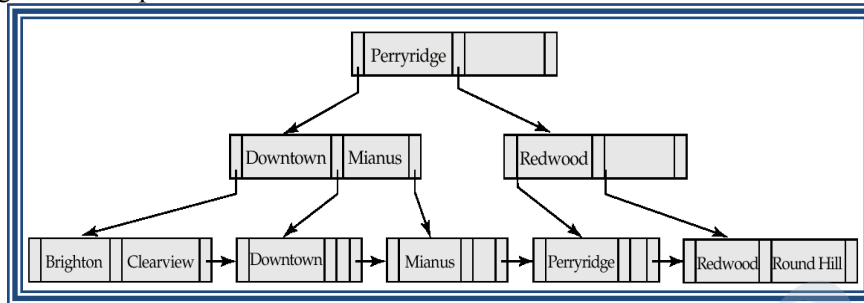
Before and after deleting "Downtown"
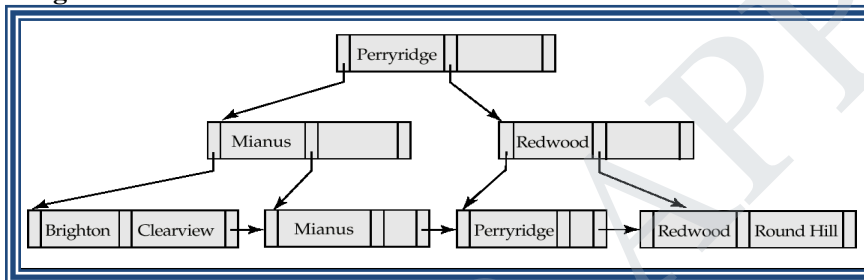
- The removal of the leaf node containing "Downtown" did not result in its parent having too little pointers. So the cascaded deletions stopped with the deleted leaf node's parent.

Deletion of "Perryridge" from result of previous example
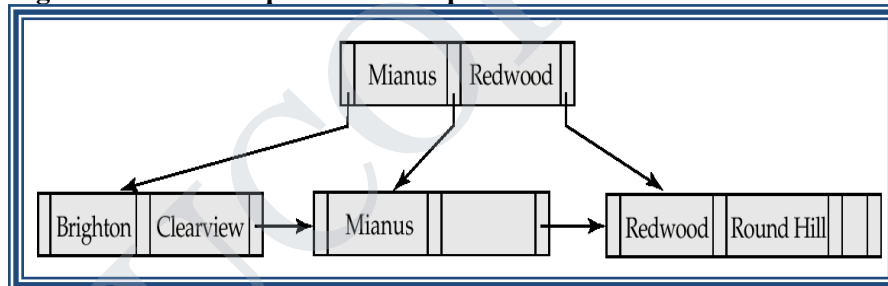
- Node with "Perryridge" becomes empty and merged with its sibling.
- Root node then had only one child, and was deleted and its child became the new root node

B⁺-Tree File Organization
- The leaf nodes in a B⁺-tree file organization store records, instead of pointers.
- Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node.
- Leaf nodes are still required to be half full.
- Insertion and deletion are handled in the same way as insertion and deletion of entries in a B⁺-tree index.
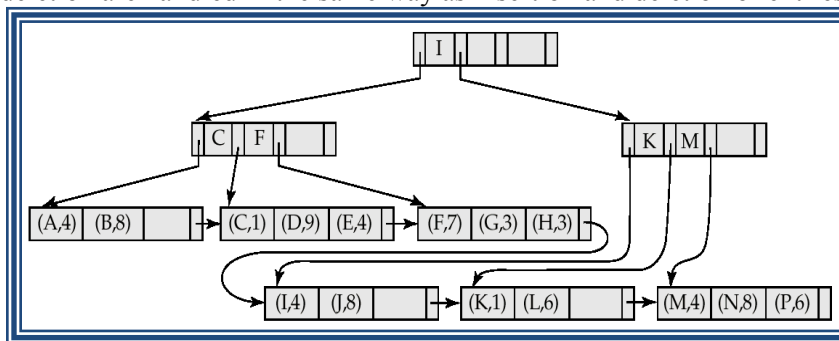
B-Tree Index Files

- Similar to B+-tree, but B-tree allows search-key values to appear only once; eliminates redundant storage of search keys.
- Search keys in nonleaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a nonleaf node must be included.
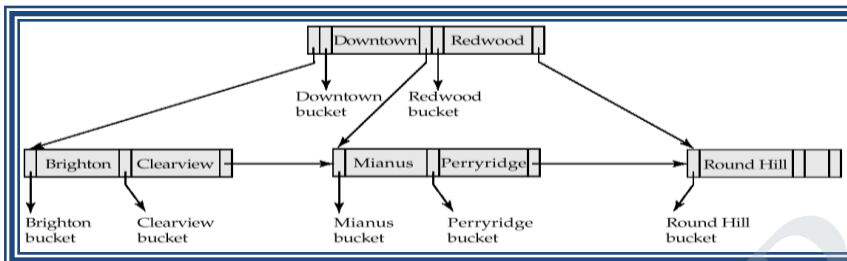
Generalized B-tree leaf node
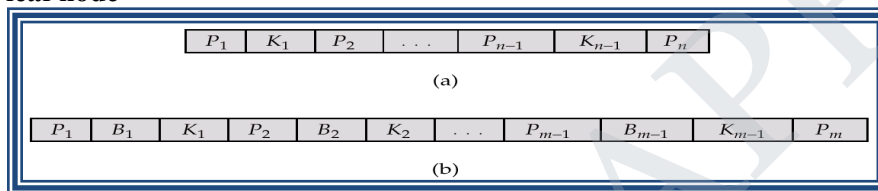
Nonleaf node – pointers B_i are the bucket or file record pointers.

B+-tree on same data
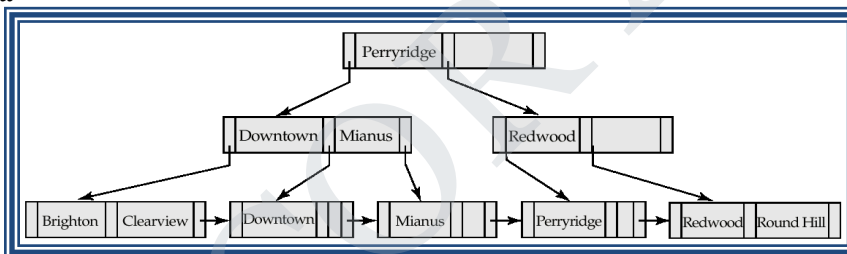
Advantages of B-Tree indices:

- May use less tree nodes than a corresponding B^+-Tree.
- Sometimes possible to find search-key value before reaching leaf node.

Disadvantages of B-Tree indices:

- Only small fraction of all search-key values are found early
- Non-leaf nodes are larger, so fan-out is reduced (no. of pointers). Thus, B-Trees typically have greater depth than corresponding B^+-Tree
- Insertion and deletion more complicated than in B^+-Trees
- Implementation is harder than B^+-Trees.

HASHING

- Hashing is an effective technique to calculate the direct location of a data record on the disk without using index structure.
- Hashing uses hash functions with search keys as parameters to generate the address of a data record.

Hash Organization

Bucket

A hash file stores data in bucket format. Bucket is considered a unit of storage. A bucket typically stores one complete disk block, which in turn can store one or more records.

Hash Function

A hash function, **h,** is a mapping function that maps all the set of search-keys **K** to the address where actual records are placed. It is a function from search keys to bucket addresses.

- Worst hash function maps all search-key values to the same bucket.

- An ideal hash function is uniform, i.e., each bucket is assigned the same number of search-key values from the set of all possible values.
- Ideal hash function is random, so each bucket will have the same number of records.

Types
- Static Hashing
- Dynamic Hashing

Static Hashing
- In static hashing, when a search-key value is provided, the hash function always computes the same address.
- For example, if mod-4 hash function is used, then it shall generate only 5 values. The output address shall always be same for that function.
- The number of buckets provided remains unchanged at all times.

Example of Hash File Organization
- There are 10 buckets,
- The hash function returns the sum of the binary representations of the characters modulo 10
  - E.g. h(Perryridge) = 5   h(Round Hill) = 3   h(Brighton) = 3

Operation
- **Insertion** − When a record is required to be entered using static hash, the hash function **h** computes the bucket address for search key **K**, where the record will be stored.
  Bucket address = h(K)
- **Search** − When a record needs to be retrieved, the same hash function can be used to retrieve the address of the bucket where the data is stored.
- **Delete** − This is simply a search followed by a deletion operation.

Handling of Bucket Overflows
- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records. This can occur due to :
    - multiple records have same search-key value
- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using overflow buckets.
- Overflow chaining – the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called **closed hashing**.
  - An alternative, called **open hashing**, which does not use overflow buckets, is not suitable for database applications.
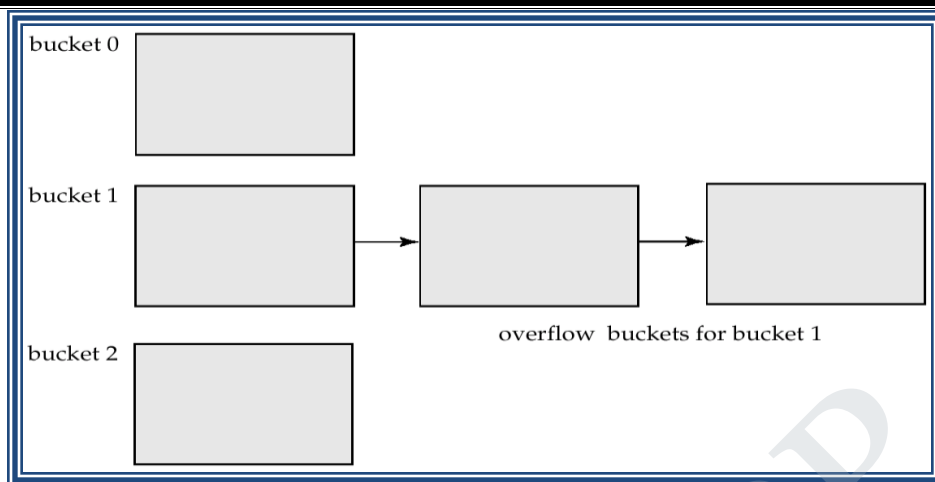
Hash Indices

- Hashing can be used not only for file organization, but also for index-structure creation.
- A hash index organizes the search keys, with their associated record pointers, into a hash file structure.
- Hash indices are always secondary indices
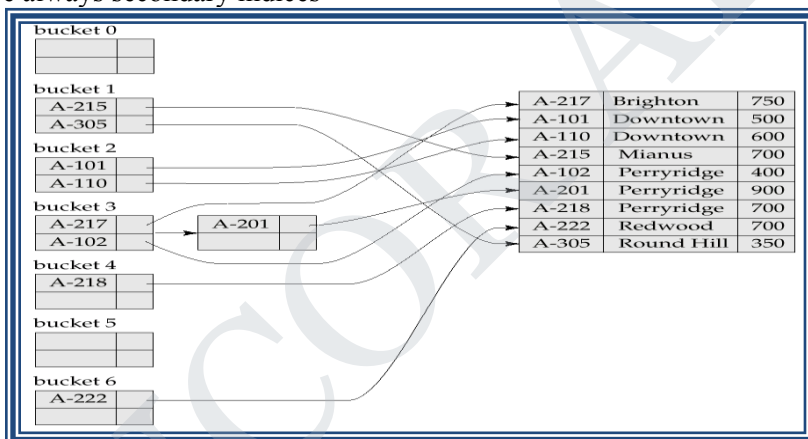
Deficiencies of Static Hashing

- In static hashing, function h maps search-key values to a fixed set of B of bucket addresses.
  - Databases grow with time. If initial number of buckets is too small, performance will degrade due to too much overflows.
  - If file size at some point in the future is anticipated and number of buckets allocated accordingly, significant amount of space will be wasted initially.
  - If database shrinks, again space will be wasted.
- These problems can be avoided by using techniques that allow the number of buckets to be **modified dynamically**.

Dynamic Hashing

- Good for database that grows and shrinks in size
- Allows the hash function to be modified dynamically
- Extendable hashing – one form of dynamic hashing
  - Hash function generates values over a large range — typically b-bit integers, with b = 32.
  - At any time use only a prefix of the hash function to index into a table of bucket addresses.
  - Let the length of the prefix be i bits, $0 \le i \le 32$.
  - Bucket address table size = 2^i. Initially i = 0
  - Value of i grows and shrinks as the size of the database grows and shrinks.
  - Multiple entries in the bucket address table may point to a bucket.
  - Thus, actual number of buckets is $< 2^i$

– The number of buckets also changes dynamically due to coalescing and splitting of buckets.

General Extendable Hash

In this structure, $i_2 = i_3 = i$, whereas $i_1 = i - 1$
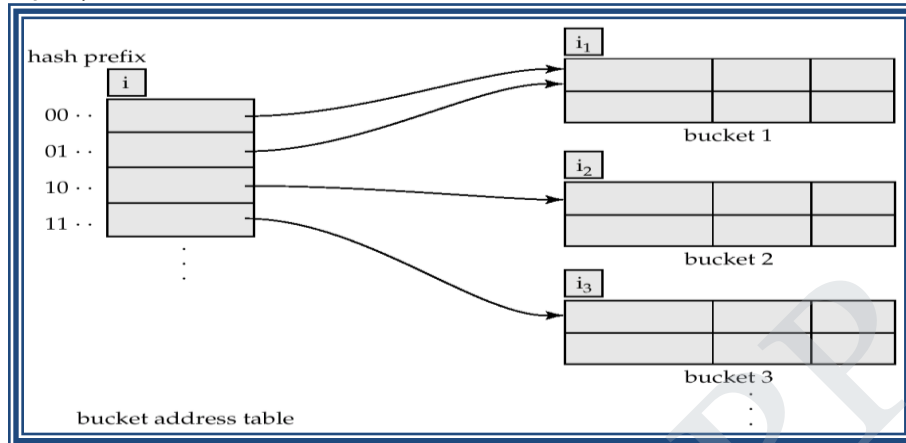
Insertion in Extendable Hash Structure

To split a bucket j when inserting record with search-key value K_j:

- If $i > i_j$ (more than one pointer to bucket j)
    - allocate a new bucket z, and set $i_j = i_z = (i_j + 1)$
    - Update the second half of the bucket address table entries originally pointing to j, to point to z
    - remove each record in bucket j and reinsert (in j or z)
    - recompute new bucket for K_i and insert record in the bucket (further splitting is required if the bucket is still full)
- If $i = i_j$ (only one pointer to bucket j)
    - If i reaches some limit b, or too many splits have happened in this insertion, create an overflow bucket
- Else
    - increment i and double the size of the bucket address table.
    - replace each entry in the table by two entries that point to the same bucket.
   - recompute new bucket address table entry for K_j
    - Now $i > i_j$ so use the first case above.

Deletion in Extendable Hash Structure

- To delete a key value,
    - locate it in its bucket and remove it.
    - The bucket itself can be removed if it becomes empty (with appropriate updates to the bucket address table).
    - Coalescing of buckets can be done (can coalesce only with a "*buddy*" bucket having same value of i_j and same $i_j - 1$ prefix, if it is present)
    - Decreasing bucket address table size is also possible
        - Note: decreasing bucket address table size is an expensive operation and should be done only if number of buckets becomes much smaller than the size of the table

Example

| branch_name	h(branch_name)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001
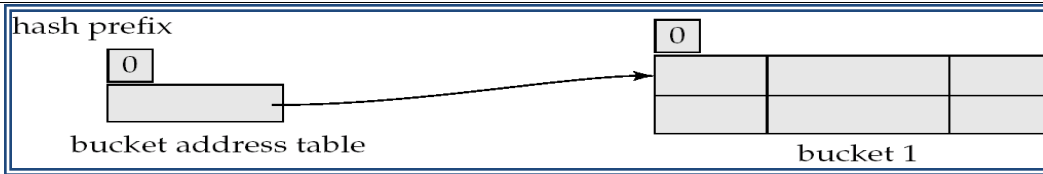
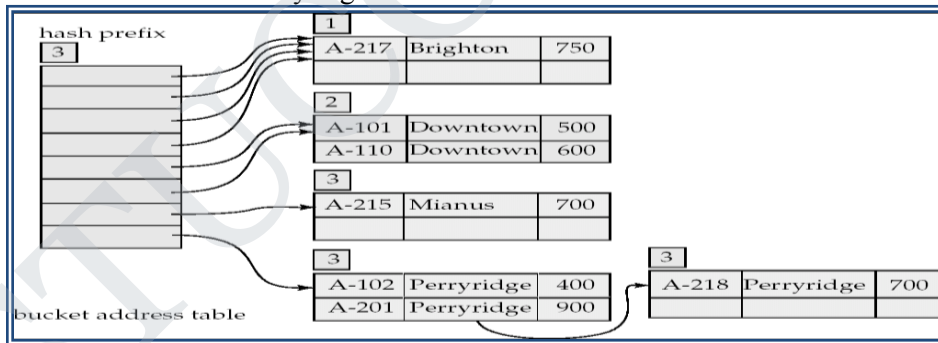Initial Hash structure, bucket size = 2
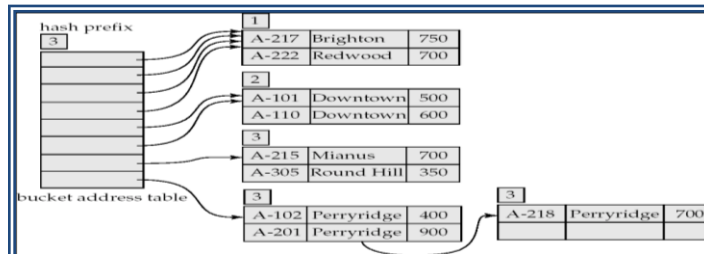Hash structure after insertion of one Brighton and two Downtown records
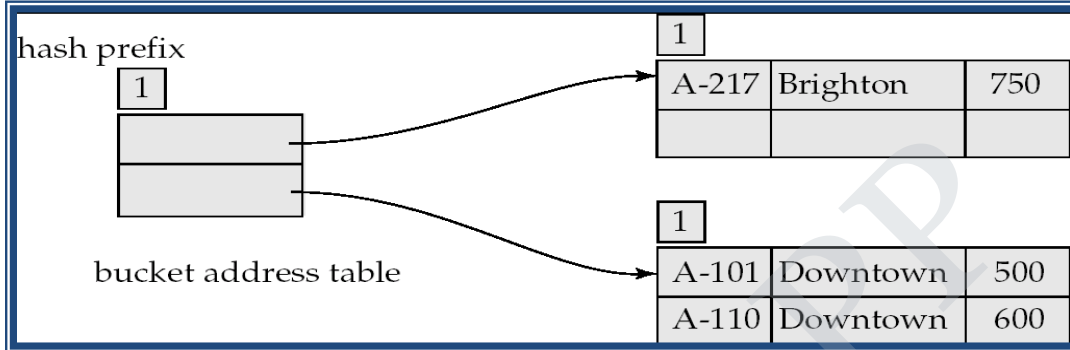
Hash structure after insertion of Mianus record
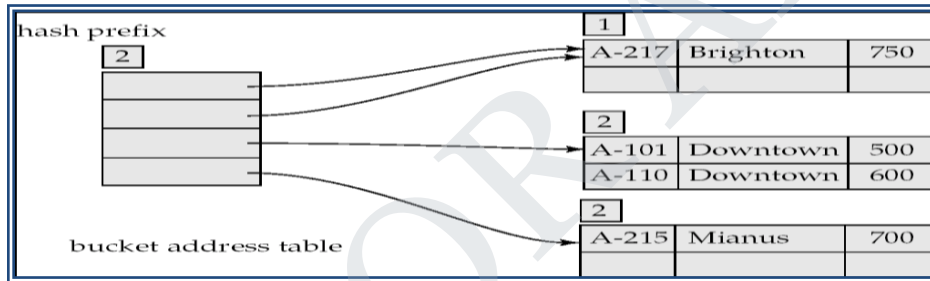
Hash structure after insertion of three Perryridge records

Hash structure after insertion of Redwood and Round Hill records

Use of Extendable Hash Structure

- To locate the bucket containing search-key K_j:
  1. Compute $h(K_j) = X$
  2. Use the first i high order bits of X as a displacement into bucket address table, and follow the pointer to
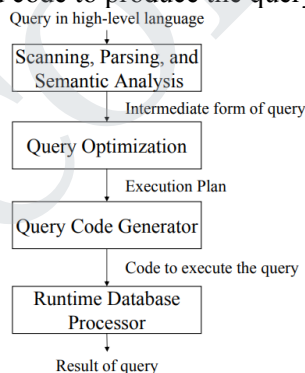
---

appropriate bucket

Updates in Extendable Hash Structure
- **To insert a record with search-key value K$_i$**
  – follow same procedure as look-up and locate the bucket, say j.
  – If there is room in the bucket j insert record in the bucket.
  – Overflow buckets used instead in some cases.
- **To delete a key value,**
  – locate it in its bucket and remove it.
  – The bucket itself can be removed if it becomes empty
  – Coalescing of buckets can be done
  – Decreasing bucket address table size is also possible
- **Benefits of extendable hashing:**
  – Hash performance does not degrade with growth of file
  – Minimal space overhead
- **Disadvantages of extendable hashing**
  – Extra level of indirection to find desired record

Bucket address table may itself become very big.

QUERY PROCESSING OVERVIEW

1. The scanning, parsing, and validating module produces an internal representation of the query.
2. The query optimizer module devises an execution plan which is the execution strategy to retrieve the result of the query from the database files. A query typically has many possible execution strategies differing in performance, and the process of choosing a reasonably efficient one is known as query optimization. Query optimization is beyond this course. The code generator generates the code to execute the plan.The runtime database processor runs the generated code to produce the query result.

Query in high-level language
↓
Scanning, Parsing, and Semantic Analysis
↓ Intermediate form of query
Query Optimization
↓ Execution Plan
Query Code Generator
↓ Code to execute the query
Runtime Database Processor
↓ Result of query

Evaluation of SQL Statement

The query is evaluated in a different order.
- The tables in the from clause are combined using Cartesian products.The where predicate is then applied.
- The resulting tuples are grouped according to the group by clause.The having predicate is applied to each group, possibly eliminating some groups.The aggregates are applied to each remaining group.The select clause is performed last.

Translating SQL Queries into Relational Algebra
- SQL query is first translated into an equivalent extended relational algebra expression.
- SQL queries are decomposed into query blocks, which form the basic units that can be translated into the algebraic operators and optimized.
- Query block contains a single SELECT-FROM-WHERE expression, as well as GROUP BY and HAVING clauses.
- Nested queries within a query are identified as separate query blocks.

Example:

SELECT LNAME, FNAME
FROM EMPLOYEE
WHERE SALARY > (SELECT MAX(SALARY)
                FROM EMPLOYEE
   WHERE DNO=5);

- The inner block
  - (SELECT MAX (SALARY) FROM EMPLOYEE WHERE DNO=5)
  - – Translated in:
  - ℑ MAX SALARY(σDNO=5(EMPLOYEE))
- The Outer block
  - SELECT LNAME, FNAME FROM EMPLOYEE WHERE SALARY > C
  - – Translated in:
  - ∏ LNAZME, FNAME (σSALARY>C(EMPLOYEE))
          (C represents the result returned from the inner block.)
- The query optimizer would then choose an execution plan for each block.
- The inner block needs to be evaluated only once. (Uncorrelated nested query).
- It is much harder to optimize the more complex correlated nested queries.

External Sorting

It refers to sorting algorithms that are suitable for large files of records on disk that do not fit entirely in main memory, such as most database files..

- ORDER BY.
- Sort-merge algorithms for JOIN and other operations (UNION, INTERSECTION). Duplicate elimination algorithms for the PROJECT operation (DISTINCT). Typical external sorting algorithm uses a sort-merge strategy: Sort phase: Create sort small sub-files (sorted sub-files are called runs).
- Merge phase: Then merges the sorted runs. N-way merge uses N memory buffers to buffer input runs, and 1 block to buffer output. Select the 1st record (in the sort order) among input buffers, write it to the output buffer and delete it from the input buffer. If output buffer full, write it to disk. If input buffer empty, read next block from the corresponding run..

E.g. 2-way Sort-Merge
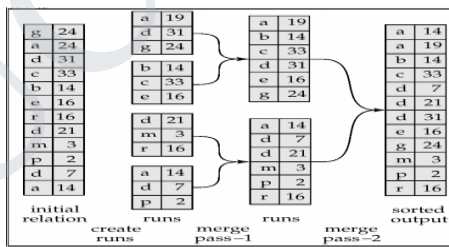
Basic Algorithms for Executing Relational Query Operations

- An RDBMS must include one or more alternative algorithms that implement each relational algebra operation (SELECT, JOIN,…) and, in many cases, that implement each combination of these operations.
- Each algorithm may apply only to particular storage structures and access paths (such index,…).
- Only execution strategies that can be implemented by the RDBMS algorithms and that apply to the particular query and particular database design can be considered by the query optimization module.

Algorithms for implementing SELECT operation

- These algorithms depend on the file having specific access paths and may apply only to certain types of selection conditions.
- We will use the following examples of SELECT operations:
  - (OP1):σSSN='123456789' (EMPLOYEE)
  - (OP2):σ DNUMBER > 5 (DEPARTMENT)

- (OP3):σDNO=5 (EMPLOYEE)
    - (OP4):σ DNO=5 AND SALARY>30000 AND SEX = 'F' (EMPLOYEE)
    - (OP5):σESSN='123456789' AND PNO=10 (WORKS_ON)
- Many search methods can be used for simple selection: S1 through S6
- **S1: Linear Search (brute force) –full scan in Oracle's terminology-**
    - Retrieves every record in the file, and test whether its attribute values satisfy the selection condition: an expensive approach.
    - Cost: b/2 if key and b if no key
- **S2: Binary Search**
    - If the selection condition involves an equality comparison on a key attribute on which the file is ordered.
    - σSSN='1234567' (EMPLOYEE), SSN is the ordering attribute.
    - Cost: log2b if key.
- **S3: Using a Primary Index (hash key)**
    - An equality comparison on a key attribute with a primary index (or hash key).
    - This condition retrieves a single record (at most).
    - Cost :primary index : bind/2 + 1(hash key: 1bucket if no collision).
- **S4: Using a primary index to retrieve multiple records**
    - Comparison condition is >, >=, <, or <= on a key field with a primary index
    - σDNUMBER >5(DEPARTMENT)
    - Use the index to find the record satisfying the corresponding equality condition (DNUMBER=5), then retrieve all subsequent records in the (ordered) file.
    - For the condition (DNUMBER <5), retrieve all the preceding records.
    - Method used for range queries too(i.e. queries to retrieve records in certain range)
    - Cost: bind/2 + ?. '?' could be known if the number of duplicates known.
- **S5: Using a clustering index to retrieve multiple records**
    - If the selection condition involves an equality comparison on a non-key attribute with a clustering index.
    - σDNO=5(EMPLOYEE)
    - Use the index to retrieve all the records satisfying the condition.
    - Cost: log2bind + ?. '?' could be known if the number of duplicates known.
- **S6: Using a secondary (B+-tree) index on an equality comparison**
    - The method can be used to retrieve a single record if the indexing field is a key or to retrieve multiple records if the indexing field is not a key.
    - This can also be used for comparisons involving >, >=, <, or <=.
    - Method used for range queries too.
    - Cost to retrieve: a key= height + 1; a non key= height+1(extra-level)+?, comparison=(height-1)+?+?
- Many search methods can be used for complex selection which involve a Conjunctive Condition: S7 through as S9.
    - Conjunctive condition: several simple conditions connected with the AND logical connective.
    - (OP4): s DNO=5 AND SALARY>30000 AND SEX = 'F' (EMPLOYEE).
- **S7:Conjunctive selection using an individual index.**
    - If an attribute involved in any single simple condition in the conjunctive condition has an access path that permits the use of one of the Methods S2 to S6, use that condition to retrieve the records.
    - Then check whether each retrieved record satisfies the remaining simple conditions in the conjunctive condition
- **S8:Conjunctive selection using a composite index:**
    - If two or more attributes are involved in equality conditions in the conjunctive condition and a composite index (or hash structure) exists on the combined fields.
    - Example: If an index has been created on the composite key (ESSN, PNO) of the WORKS_ON file,

we can use the index directly.
- (OP5):$\sigma_{ESSN='123456789' \text{ AND } PNO=10}$ (WORKS_ON).
- **S9: Conjunctive selection by intersection of record pointers**
  - If the secondary indexes are available on more than one of the fields involved in simple conditions in the conjunctive condition, and if the indexes include record pointers (rather than block pointers), then each index can be used to retrieve the set of record pointers that satisfy the individual condition.
  - The intersection of these sets of record pointers gives the record pointers that satisfy the conjunctive condition.
  - If only some of the conditions have secondary indexes, each retrieval record is further tested to determine whether it satisfies the remaining conditions.

Algorithms for implementing JOIN Operation
- **Join: time-consuming operation. We will consider only natural join operation**
  - Two-way join: join on two files.
  - Multiway join: involving more than two files.
- **The following examples of two-way JOIN operation (RΘ A=BS) will be used:**
  - OP6: EMPLOYEE Θ DNO=DNUMBER DEPARTMENT
  - OP7: DEPARTMENT Θ MGRSSN=SSN EMPLOYEE
- **J1: Nested-loop join (brute force)**
  - For each record t in R (outer loop), retrieve every record s from S (inner loop) and test whether the two records satisfy the join condition t[A] = s[B].
- **J2: Single-loop join (using an access structure to retrieve the matching records)**
  - If an index (or hash key) exists for one of the two join attributes (e.g B of S), retrieve each record t in R, one at a time (single loop), and then use the access structure to retrieve directly all matching records s from S that satisfy s[B] = t[A]
- **J3: Sort-merge join:**
  - If the records of R and S are physically sorted (ordered) by value of the join attributes A and B, respectively, we can implement the join in the most efficient way.
  - Both files are scanned concurrently in order of the join attributes, matching the records that have the same values for A and B.
  - If the files are not sorted, they may be sorted first by using external sorting.
  - Pairs of file blocks are copied into memory buffers in order and records of each file are scanned only once each for matching with the other file if A & B are key attributes.
  - The method is slightly modified in case where A and B are not key attributes.
- **J4: Hash-join**
  - The records of files R and S are both hashed to the same hash file using the same hashing function on the join attributes A of R and B of S as hash keys.
- **Partitioning Phase**
  - First, a single pass through the file with fewer records (say, R) hashes its records to the hash file buckets.
  - Assumption: The smaller file fits entirely into memory buckets after the first phase.
  - (If the above assumption is not satisfied, the method is a more complex one and number of variations have been proposed to improve efficiency: partition has join and hybrid hash join.)
- **Probing Phase**
  - A single pass through the other file (S) then hashes each of its records to probe appropriate bucket, and that record is combined with all matching records from R in that bucket.

Heuristic-Based Query Optimization
1. Break up SELECT operations with conjunctive conditions into a cascade of SELECT operations
2. Using the commutativity of SELECT with other operations, move each SELECT operation as far down the query tree as is permitted by the attributes involved in the select condition

3. Using commutativity and associativity of binary operations, rearrange the leaf nodes of the tree
4. Combine a CARTESIAN PRODUCT operation with a subsequent SELECT operation in the tree into a JOIN operation, if the condition represents a join condition
5. Using the cascading of PROJECT and the commuting of PROJECT with other operations, break down and move lists of projection attributes down the tree as far as possible by creating new PROJECT operations as needed
6. Identify sub-trees that represent groups of operations that can be executed by a single algorithm

Example

- Query
  "Find the last names of employees born after 1957 who work on a project named 'Aquarius'."
- SQL

SELECT LNAME
FROM EMPLOYEE, WORKS_ON, PROJECT
WHERE PNAME='Aquarius' **AND** PNUMBER=PNO **AND** ESSN=SSN **AND** BDATE.'1957-12-31';
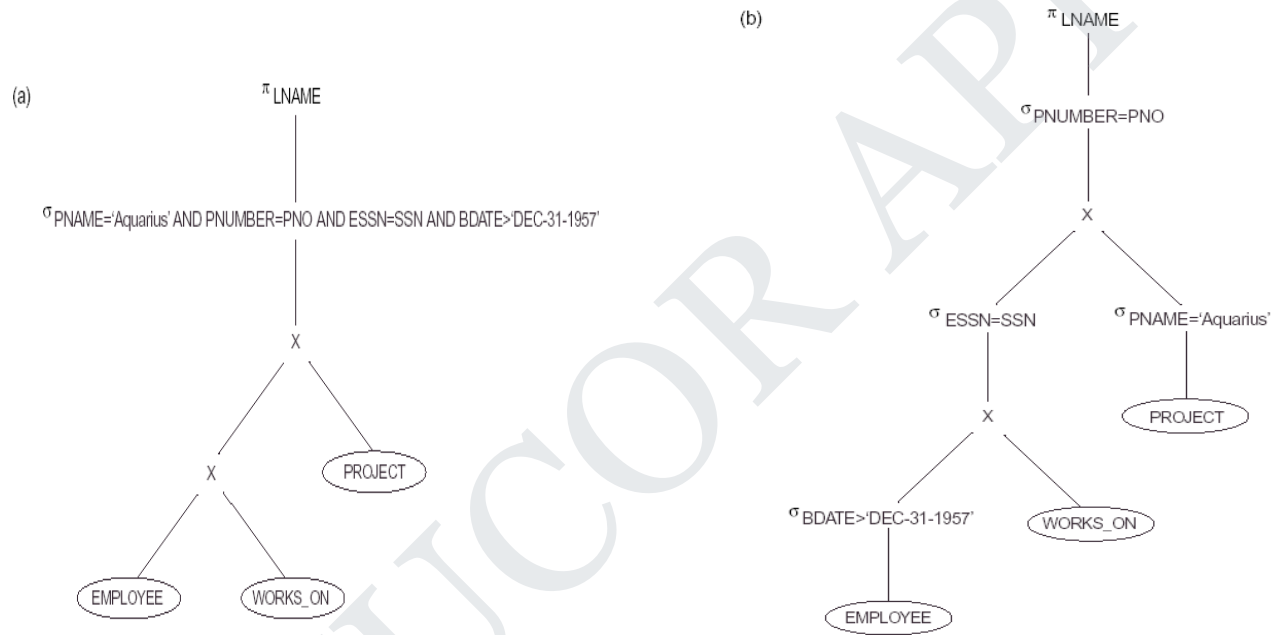
Cost Estimation in Query Optimization

The main aim of query optimization is to choose the most efficient way of implementing the relational algebra operations at the lowest possible cost.

- The query optimizer should not depend solely on heuristic rules, but, it should also estimate the cost of executing the different strategies and find out the strategy with the minimum cost estimate.

The cost functions used in query optimization are estimates and not exact cost functions.

- The cost of an operation is heavily dependent on its selectivity, that is, the proportion of select operation(s) that forms the output.
- In general the different algorithms are suitable for low or high selectivity queries.
- In order for query optimizer to choose suitable algorithm for an operation an estimate of the cost of executing that algorithm must be provided

The cost of an algorithm depends on cardinality of its input.

- To estimate the cost of different query execution strategies, the query tree is viewed as containing a series of basic operations which are linked in order to perform the query.
- It is also important to know the expected cardinality of an operation's output because this forms the input to the next operation.

Cost Components of Query Execution
The cost of executing the query includes the following components:
– Access cost to secondary storage.
– Storage cost.
– Computation cost.
– Memory uses cost.
– Communication cost.

Importance of Access cost
Out of the above five cost components, the most important is the secondary storage access cost.
- The emphasis of the cost minimization depends on the size and type of database applications.
- For example in smaller database the emphasis is on the minimizing computing cost as because most of the data in the files involve in the query can be completely store in the main memory.
- For large database, the main emphasis is on minimizing the access cost to secondary device.
- For distributed database, the communication cost is minimized as because many sites are involved
- for the data transfer.

Cost functions for SELECT Operation
Linear Search:
– [nBlocks(R)/2], if the record is found.
– [nBlocks(R)], if no record satisfied the condition.

Binary Search :
o [log2(nBlocks(R))], if equality condition is on key attribute, because SCA(R) = 1 in this case.
o[log2(nBlocks(R))] + [SCA(R)/bFactor(R)] – 1, otherwise.

Equity condition on Primary key
– [nLevelA(I) + 1]

Equity condition on Non-Primary key
– [nLevelA(I) + 1] + [nBlocks(R)/2]

Cost functions for JOIN Operation
Join operation is the most time consuming operation to process.
- An estimate for the size (number of tuples) of the file that results after the JOIN operation is required to develop reasonably accurate cost functions for JOIN operations.
- The JOIN operations define the relation containing tuples that satisfy a specific predicate F from the Cartesian product of two relations R and S.