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

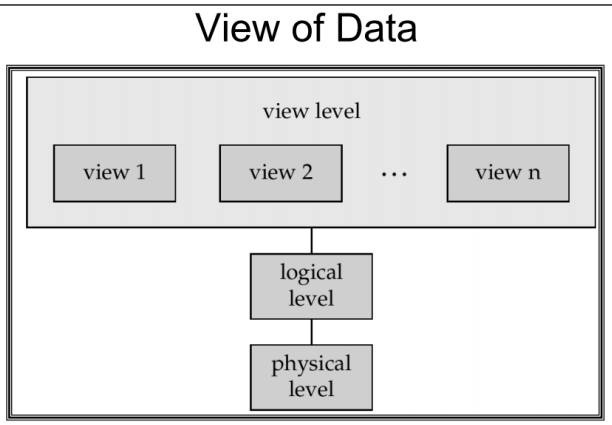
VIEWS OF DATA
It refers that how database is actually stored in database, what data and structure of data used by database for data. So describe all this database provides user with views and these are

- **Data abstraction**
- **Instances and schemas**

Dataabstraction
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.



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.

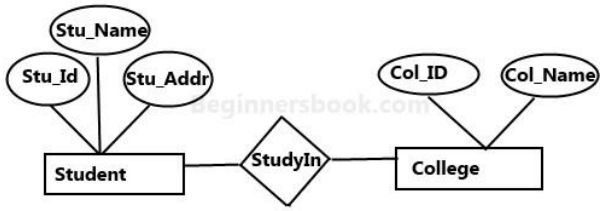
Types of Data Models:

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.

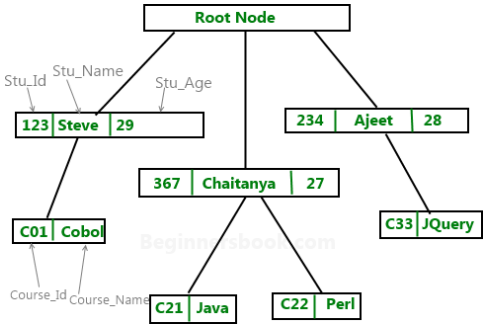
Sample relationship Model: Student table with 3 columns and three records.

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

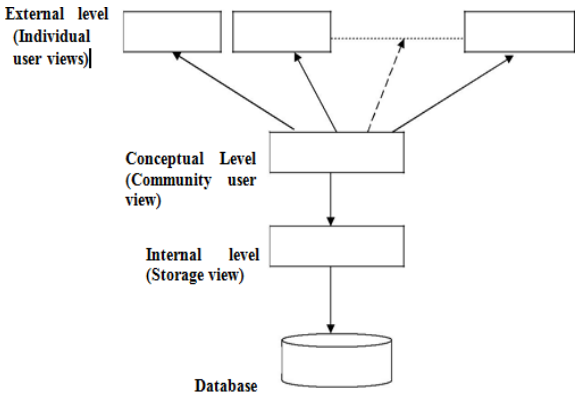
The goal of the three schema architecture is to separate the user applications and the physical database. The schemas can be defined at the following levels:

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.

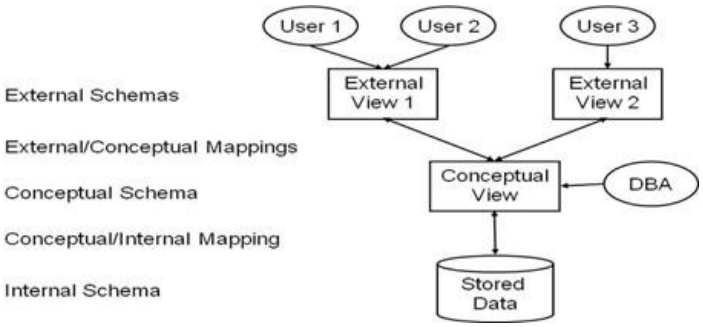
2. **The conceptual level** – has a conceptual schema which describes the structure of the database for users. It hides the details of the physical storage structures, and concentrates on describing entities, data types, relationships, user operations and constraints. Usually a representational data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

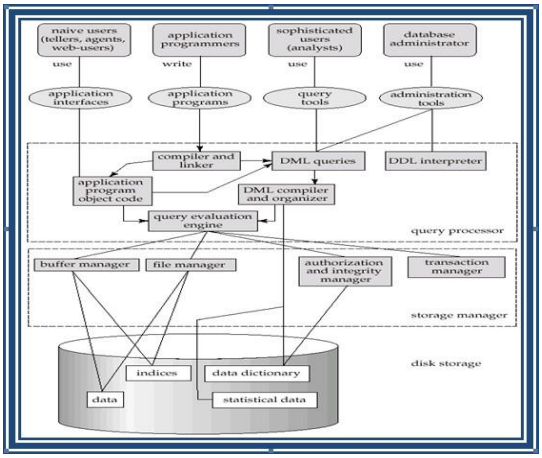
- Storage manager is a program module that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system.
- **The storage manager is responsible to the following tasks:**
  - interaction with the file manager
  - efficient storing, 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 − =, ≠, ≥, < , >, ≤.

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.

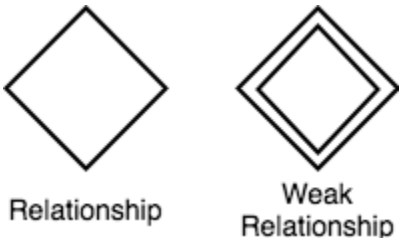
Let's see how we can represent these in our ER Diagram.

Entity

Simple rectangular box represents an Entity.

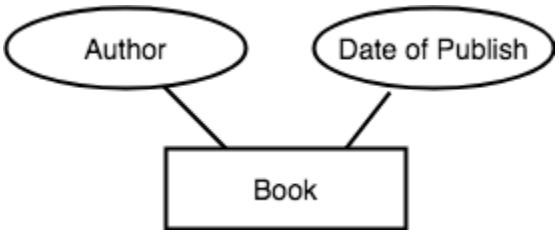
Relationships between Entities - Weak and Strong

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.

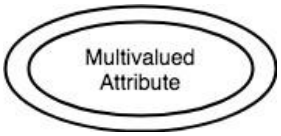
Key Attribute for any Entity

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

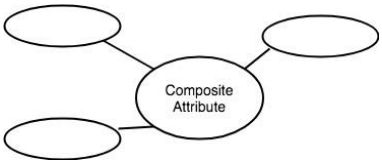
Derived Attribute for any Entity
Derived attributes are those which are derived based on other attributes, for example, age can be derived from date of birth.

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.



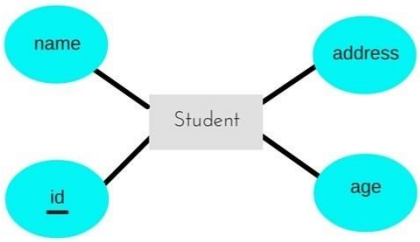
Composite Attribute for any Entity
A composite attribute is the attribute, which also has attributes.



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.

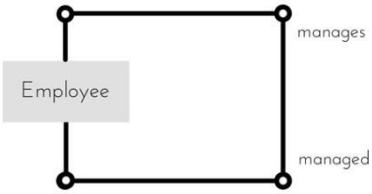
One to Many Relationship
The below example showcases this relationship, which means that 1 student can opt for many courses, but a course can only have 1 student. Sounds weird! This is how it is.

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

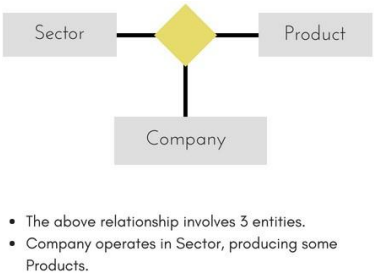
The above diagram represents that one student can enroll for more than one courses. And a course can have more than 1 student enrolled in it.

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.



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:

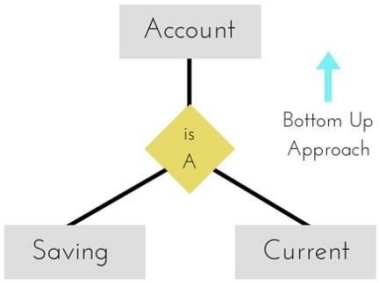
1. Generalization
2. Specialization
3. Aggregration

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-

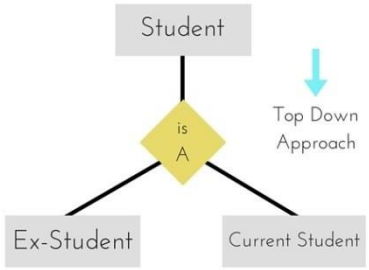
class.



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

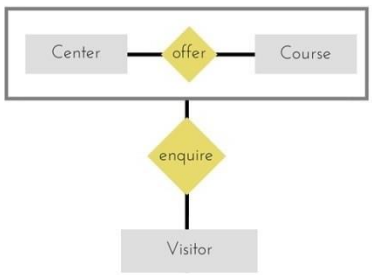
Specialization

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.

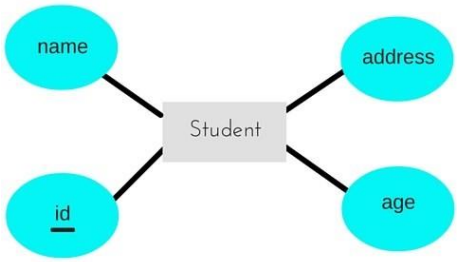
ER Model to Relational Model

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

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.

$X \rightarrow Y$

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

For example:

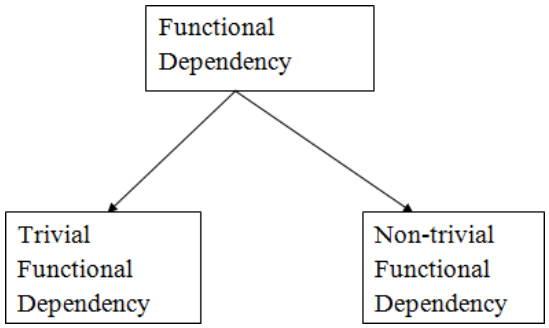
Assume we have an employee table with attributes: Emp_Id, Emp_Name, Emp_Address.

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

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

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

$A \rightarrow B$ has a non-trivial functional dependency if B is not a subset of A.

When A intersection B is NULL, then $A \rightarrow B$ is called as complete non-trivial.

Example:

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

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.

1	Java
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.

1	10	1	70	Java 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 ($X \rightarrow Y$), X should be a super Key.In simple words, it means, that for a dependency $A \rightarrow B$, 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
Mathematics	Alex
Mathematics	Rose
Physics	Rose
Physics	Joseph
Chemistry	Adam

CLASS	LECTURER
SEMESTER 1	Alex
SEMESTER 1	Rose
SEMESTER 1	Rose
SEMESTER 2	Joseph
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

TRANSACTION CONCEPTS.

A **transaction** is a collection of operations that forms single logical unit of work.

Simple Transaction Example

1. Read your account balance
2. Deduct the amount from your balance
3. Write the remaining balance to your account
4. Read your friend's account balanace
5. Add the amount to his account balance
6. Write the new updated balance to his account

This whole set of operations can be called a transaction

Transaction processing system

- The system with large database and hundreds of concurrent users that are executing database transaction.
- Eg :reservation system , banking system etc

Concurrent access

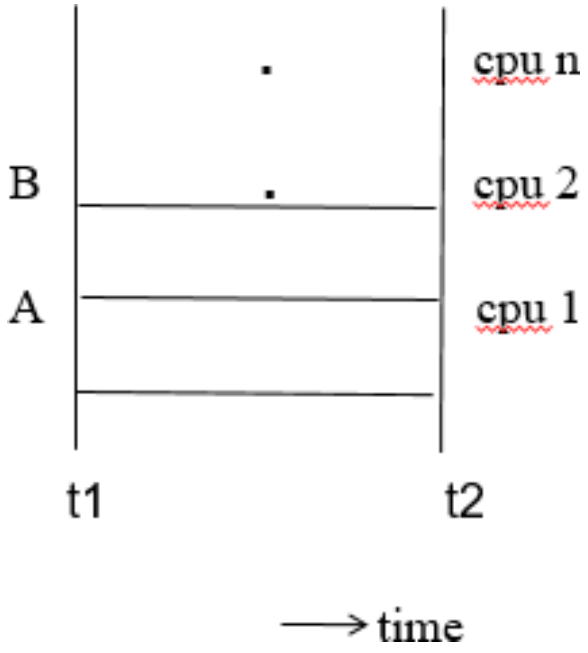
- mutiple user accessing a system at the same time

Single user-one user at a time can use a system

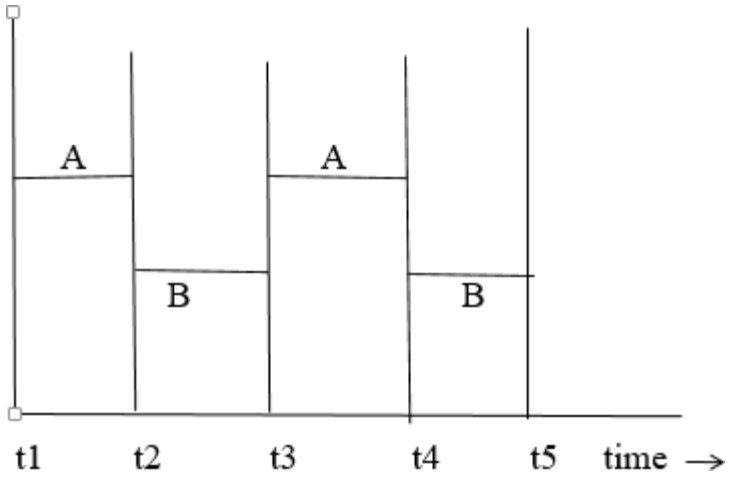
Multi user-many user use the system at a time.

It can be achieved by multiprogramming:

Parallel- multi-users access different resources at the same time.



Interleaved- Multiple users access a single resource based on time.



Transaction access data using two operations

- Read(x)

It transfer the data item **x** from the database to a local buffer belonging to the transaction that executed the read operation.

- Write(x)

It transfer the data item **x** from the local buffer of the transaction to the database i.e. it write back to the database.

ACID Properties

To ensure the integrity of data during a transaction, the database system maintains the following properties. These properties are widely known as ACID properties:

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

E.g. transaction to transfer $50 from account A to account B:

1. **read**(*A*)
2. *A* := *A* – 50
3. **write**(*A*)
4. **read**(*B*)

5. $B := B + 50$
6. **write**(B)

Example of Fund Transfer

- **Atomicity requirement**
  - if the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state
  - the system should ensure that updates of a partially executed transaction are not reflected in the database
- **Durability requirement**
  - once the user has been notified that the transaction has completed, the updates to the database by the transaction must persist even if there are software or hardware failures.
- **Isolation requirement** — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database **T1** **T2**

1. **read**(A)
2. $A := A - 50$
3. **write**(A)

read(A), read(B), print(A+B)

4. **read**(B)
5. $B := B + 50$
6. **write**(B)

- Isolation can be ensured trivially by running transactions **serially**
  - that is, one after the other.

SCHEDULES

- Schedule – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.
- **Serial Schedule**

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

  Schedule 1
  - Let T_1 transfer 50 from A to B, and T_2 transfer 10% of the balance from A to B.
  - A serial schedule in which T_1 is followed by T_2 :

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Schedule 2

- A serial schedule where T_2 is followed by T_1

T_1	T_2
	read(A) $temp := A * 0.1$ $A := A - temp$ write(A) read(B) $B := B + temp$ write(B)
read(A) $A := A - 50$ write(A) read(B) $B := B + 50$ write(B)	

Schedule 3

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1.

T_1	T_2
read(A) $A := A - 50$ write(A)	
	read(A) $temp := A * 0.1$ $A := A - temp$ write(A)
read(B) $B := B + 50$ write(B)	
	read(B) $B := B + temp$ write(B)

Schedule 4

The following concurrent schedule does not preserve the value of $(A + B)$.

T_1	T_2
read(A) $A := A - 50$	
	read(A) $temp := A * 0.1$ $A := A - temp$ write(A) read(B)
write(A) read(B) $B := B + 50$ write(B)	
	$B := B + temp$ write(B)

SERIALIZABILITY

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

- Serializability is the classical concurrency scheme.

- It ensures that a schedule for executing concurrent transactions is equivalent to one that executes the transactions serially in some order.

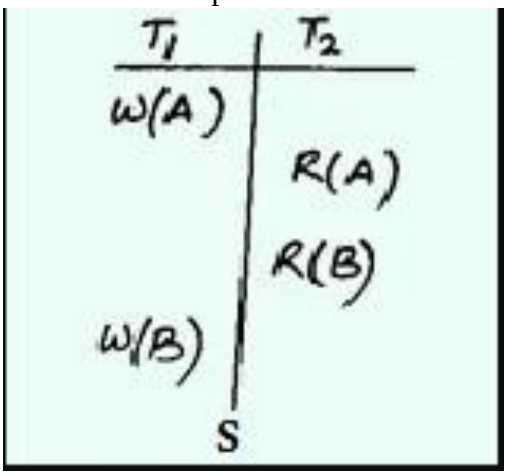
erializable schedule

If a schedule is equivalent to some serial schedule then that schedule is called Serializable schedule

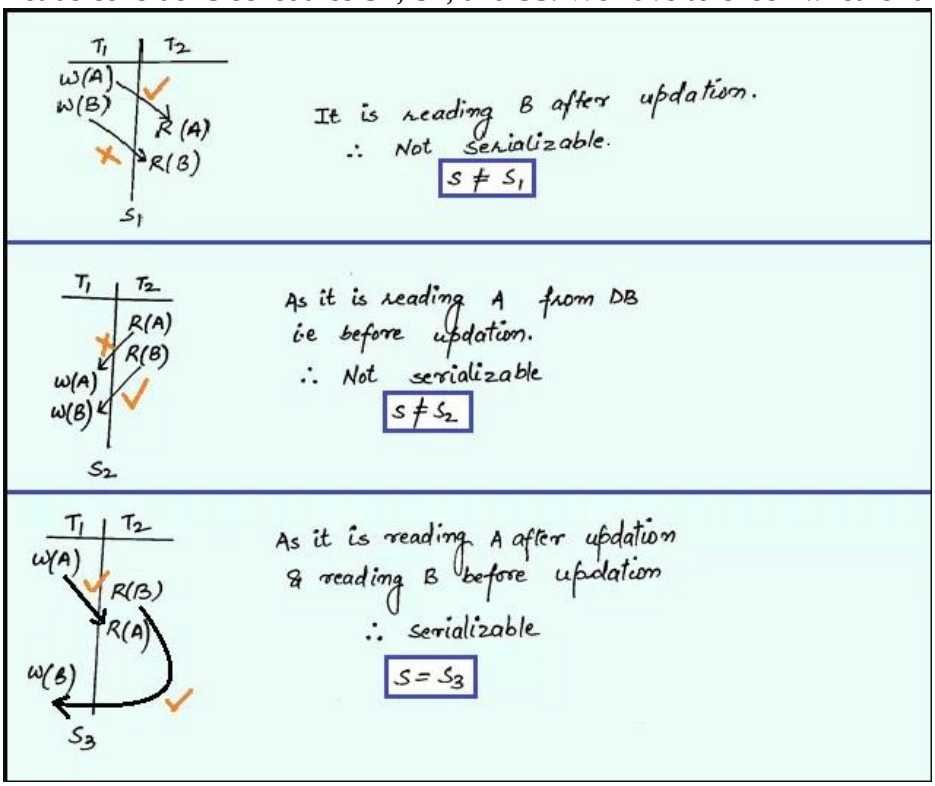
Let us consider a schedule S. What the schedule S says ?

Read A after updation.

Read B before updation.



Let us consider 3 schedules S1, S2, and S3. We have to check whether they are serializable with S or not ?



Types of Serializability

-Conflict Serializability

-View Serializability

Conflict Serializable

Any given concurrent schedule is said to be Conflict Serializable if and only if it is Conflict equivalent to one of the possible serial schedule.

Two schedules would be conflicting if they have the following properties

- Both belong to separate transactions.
- Both accesses the same data item.
- At least one of them is "write" operation.

Conflicting Instructions

Instructions l_i and l_j of transactions T_i and T_j respectively, **conflict** if they are operations by different transaction on the same data item, and at least one of these instruction is **write** operation.

1. l_i = **read**(Q), l_j = **read**(Q). l_i and l_j don't conflict.
2. l_i = **read**(Q), l_j = **write**(Q). They conflict.
3. l_i = **write**(Q), l_j = **read**(Q). They conflict
4. l_i = **write**(Q), l_j = **write**(Q). They conflict

Two schedules having multiple transactions with conflicting operations are said to be conflict equivalent if and only if

- Both the schedules contain the same set of Transactions.
- The order of conflicting pairs of operation is maintained in both the schedules.

- If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are **conflict equivalent**.
- We say that a schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule

Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable.

Schedule 3

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Schedule 6

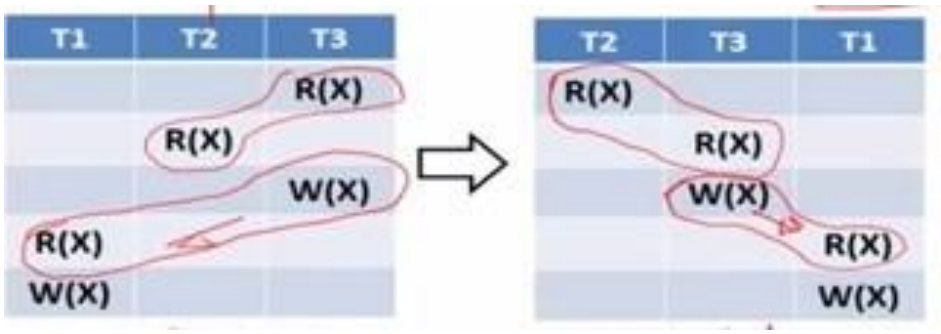
T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

View Serializable

Any given concurrent schedule is said to be View Serializable if and only if it is View equivalent to one of the possible serial schedule.

Let *S* and *S´* be two schedules with the same set of transactions. *S* and *S´* are **view equivalent** if the following three conditions are met, for each data item *Q*,

1. If in schedule S, transaction T_i reads the initial value of *Q*, then in schedule *S'* also transaction T_i must read the initial value of *Q*.
2. If in schedule S, transaction T_i executes **read**(*Q*), and that value was produced by transaction T_j (if any), then in schedule *S'* also transaction T_i must read the value of *Q* that was produced by the same **write**(Q) operation of transaction T_j .
3. The transaction (if any) that performs the final **write**(*Q*) operation in schedule *S* must also perform the final **write**(*Q*) operation in schedule *S'*.



CONCURRENCY CONTROL

Process of managing simultaneous execution of transactions in a shared database, to ensure the serializability of transactions, is known as concurrency control.

- Process of managing simultaneous operations on the database without having them interfere with one another.
- Prevents interference when two or more users are accessing database simultaneously and at least

one is updating data.

- Although two transactions may be correct in themselves, interleaving of operations may produce an incorrect result.

Why we need Concurrency Control

- Simultaneous execution of transactions over a shared database can create several data integrity and consistency problems.
- lost updated problem
- Temporary updated problem
- Incorrect summery problem

Lost updated problem

- This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.
- Successfully completed update is overridden by another user.

Example:

- T1 withdraws £10 from an account with balx, initially £100.
- T2 deposits £100 into same account.
- Serially, final balance would be £190.
- *Loss of T2's update!!*
- This can be avoided by preventing T1 from reading balx until after update.

T1	T2
Read(X)	
X:=X-N	
	Read(X)
	X:=X+M
Write(X)	
Read(Y)	
	Write(X)
Y:=Y+N	
Write(Y)	

Temporary updated problem

- This problem occurs when one transaction updates a database item and then the transaction fails for some reason. The updated item is accessed by another transaction before it is changed back to its original value.
- Occurs when one transaction can see intermediate results of another transaction before it has committed.

Example:

- T1 updates balx to £200 but it aborts, so balx should be back at original value of £100.
- T2 has read new value of balx (£200) and uses value as basis of £10 reduction, giving a new balance of £190, instead of £90.
- Problem avoided by preventing T2 from reading balx until after T1 commits or aborts.

T1	T2
Read(X) X:=X-N Write(X) |	
	Read(X) X:=X-M Write(X)
Rollback Read(Y)	

Incorrect summary problem

- If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated.
- Occurs when transaction reads several values but second transaction updates some of them during execution of first.

T1	T2
|	Sum:=0 Read(A) Sum:=sum+A Read(B) Sum:=sum+B . . . Read(X) Sum:=sum+X
Read(B) B:=B-N Write(B)	
Read(X) X:=X-N Write(X)	

Example:

- T6 is totaling balances of account x (£100), account y (£50), and account z (£25).
- Meantime, T5 has transferred £10 from balx to balz, so T6 now has wrong result (£10 too high).
- Problem avoided by preventing T6 from reading balx and balz until after T5 completed updates.

Concurrency control techniques

Some of the main techniques used to control the concurrent execution of transaction are based on the concept of locking the data items

LOCKING PROTOCOLS

A lock is a variable associated with a data item that describe the statues of the item with respect to possible operations that can be applied to it.

Locking is an operation which secures

(a) permission to Read

(b) permission to Write a data item for a transaction.
Example: Lock (X). Data item X is locked in behalf of the requesting transaction.
Unlocking is an operation which removes these permissions from the data item.
Example: Unlock (X): Data item X is made available to all other transactions.
Lock and Unlock are Atomic operations.
Conflict matrix

	Read	Write
Read	Y	N
Write	N	N

Lock Manager:
- Managing locks on data items.

Lock table:
- Lock manager uses it to store the identify of transaction locking a data item, the data item, lock mode and pointer to the next data item locked. One simple way to implement a lock table is through linked list

Types of lock
- Binary lock
- Read/write(shared / Exclusive) lock

Binary lock
- It can have two states (or) values 0 and 1.
  0 – unlocked
  1 - locked
- Lock value is 0 then the data item can accessed when requested.
- When the lock value is 1,the item cannot be accessed when requested.

- Lock_item(x)

```
B : if lock(x) = 0 ( * item is unlocked * )
  then lock(x)      1
          else begin
    wait ( until lock(x) = 0 )
    goto B;
  end;
```

- Unlock_item(x)

```
B : if lock(x)=1 ( * item is locked * )
    then lock(x)      0
    else
    printf (‘ already is unlocked ‘)
    goto B;
    end;
```

Read / write(shared/exclusive) lock

Read_lock

- its also called shared-mode lock
- If a transaction T_i has obtain a shared-mode lock on item X, then T_i can read, but cannot write ,X.
- Outer transactions are also allowed to read the data item but cannot write.

Read_lock(x)

```
B : if lock(x) = "unlocked"  then
                    begin
                    lock(x)        " read_locked"
                    no_of_read(x)       1
                    else if
                    lock(x) = "read_locked"
                    then
                    no_of_read(x)        no_of_read(x) +1
                    else begin
                    wait (until lock(x) = "unlocked"
                    goto B;
                    end;
```

Write_lock(x)

```
B : if lock(x) = "unlocked" then
          begin
          lock(x)       "write_locked"
          else if
          lock(x) = "write_locked"
          wait ( until lock(x) = "unlocked" )
          else begin
          lock(x)="read_locked" then
          wait ( until lock(x) = "unlocked" )
    end;
```

Unlock(x)

```
If lock(x) = "write_locked" then
Begin
Lock(x)       "unlocked"
Else if
lock(x) = "read_locked" then
Begin
No_of_read(x)        no_of_read(x) - 1
If ( no_of_read(x) = 0 ) then
Begin
Lock(x)       "unlocked"
End
```

TWO PHASE LOCKING PROTOCOL

This protocol requires that each transaction issue lock and unlock request in two phases

- Growing phase
- Shrinking phase

Growing phase

- During this phase new locks can be occurred but none can be released

Shrinking phase

- During which existing locks can be released and no new locks can be occurred

Types

- Strict two phase locking protocol
- Rigorous two phase locking protocol

Strict two phase locking protocol

This protocol requires not only that locking be two phase, but also all exclusive locks taken by a transaction be held until that transaction commits.

Rigorous two phase locking protocol

This protocol requires that all locks be held until all transaction commits.

Consider the two transaction T_1 and T_2

T_1 : read(a_1);
read(a_2);
…….
read(a_n);
write(a_1);

T_2: read(a_1);
read(a_2);
display(a_1+a_1);

Lock conversion

- Lock Upgrade
- Lock Downgrade

Lock upgrade:

- Conversion of existing read lock to write lock
- Take place in only the growing phase

if Ti has a read-lock (X) and Tj has no read-lock (X) ($i \neq j$) then
convert read-lock (X) to write-lock (X)
else
force Ti to wait until Tj unlocks X

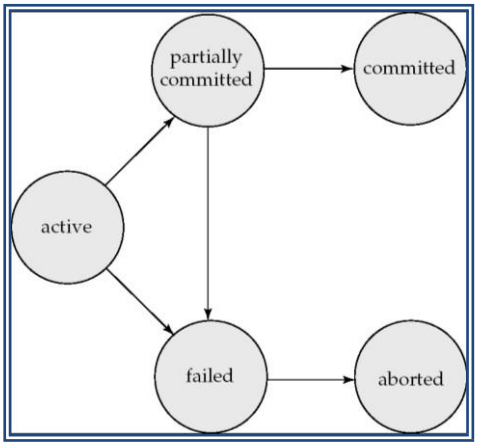
Lock downgrade:

- conversion of existing write lock to read lock
- Take place in only the shrinking phase

Ti has a write-lock (X) (*no transaction can have any lock on X*)
convert write-lock (X) to read-lock (X)

T_1	T_2
Lock-S(a_1)	
	Lock-S(a_1)
Lock-S(a_2)	
	Lock-S(a_1)
Lock-S(a_3)	
Lock-S(a_4)	
	Unlock(a_1)
	Unlock(a_2)
Lock-S(a_1)	
Upgrade(a_1)	

Transaction State



- Active – the initial state; the transaction stays in this state while it is executing
- Partially committed – after the final statement has been executed.
- Failed -- after the discovery that normal execution can no longer proceed.
- Aborted – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - restart the transaction
  - kill the transaction
- Committed – after successful completion

Log

- Log is a history of actions executed by a database management system to guarantee ACID properties over crashes or hardware failures.
- Physically, a log is a file of updates done to the database, stored in stable storage.

Log rule

- A log records for a given database update must be physically written to the log, before the update physically written to the database.
- All other log record for a given transaction must be physically written to the log, before the commit log record for the transaction is physically written to the log.
- Commit processing for a given transaction must not complete until the commit log record for the transaction is physically written to the log.

System log

- **[Begin transaction ,T]**
- **[write_item , T, X , oldvalue,newvalue]**
- **[read_item,T,X]**
- **[commit,T]**
- **[abort,T]**

TWO - PHASE COMMIT

- Assumes fail-stop model – failed sites simply stop working, and do not cause any other harm, such as sending incorrect messages to other sites.
- Execution of the protocol is initiated by the coordinator after the last step of the transaction has been reached.
- The protocol involves all the local sites at which the transaction executed
- Let T be a transaction initiated at site S_i, and let the transaction coordinator at S_i be C_i

Phase 1: Obtaining a Decision (prepare)

- Coordinator asks all participants to *prepare* to commit transaction T_i.

  - C_i adds the records <prepare *T*> to the log and forces log to stable storage
  - sends prepare *T* messages to all sites at which *T* executed
- Upon receiving message, transaction manager at site determines if it can commit the transaction
  - if not, add a record <no *T*> to the log and send abort *T* message to C_i
  - if the transaction can be committed, then:
  - add the record <ready *T*> to the log
  - force *all records* for *T* to stable storage
  - send ready *T* message to C_i

Phase 2: Recording the Decision (commit)

- *T* can be committed of C_i received a ready *T* message from all the participating sites: otherwise *T* must be aborted.
- Coordinator adds a decision record, <commit *T*> or <abort *T*>, to the log and forces record onto stable storage. Once the record stable storage it is irrevocable (even if failures occur)
- Coordinator sends a message to each participant informing it of the decision (commit or abort)
- Participants take appropriate action locally.

Handling of Failures - Site Failure

When site S_i recovers, it examines its log to determine the fate of transactions active at the time of the failure.

- Log contain <commit *T*> record: site executes redo (*T*)
- Log contains <abort *T*> record: site executes undo (*T*)
- Log contains <ready *T*> record: site must consult C_i to determine the fate of *T*.
  - If *T* committed, redo (*T*)
  - If *T* aborted, undo (*T*)
- The log contains no control records concerning *T* replies that S_k failed before responding to the prepare *T* message from C_i
  - since the failure of S_k precludes the sending of such a response C_1 must abort *T*
  - S_k must execute undo (*T*)

Handling of Failures- Coordinator Failure

- If coordinator fails while the commit protocol for *T* is executing then participating sites must decide on *T*'s fate:
  1. If an active site contains a <commit *T*> record in its log, then *T* must be committed.
  2. If an active site contains an <abort *T*> record in its log, then *T* must be aborted.
  3. If some active participating site does not contain a <ready *T*> record in its log, then the failed coordinator C_i cannot have decided to commit *T*. Can therefore abort *T*.
  4. If none of the above cases holds, then all active sites must have a <ready *T*> record in their logs, but no additional control records (such as <abort *T*> of <commit *T*>). In this case active sites must wait for C_i to recover, to find decision.
- Blocking problem : active sites may have to wait for failed coordinator to recover.

Handling of Failures - Network Partition

- If the coordinator and all its participants remain in one partition, the failure has no effect on the commit protocol.
- If the coordinator and its participants belong to several partitions:
  - Sites that are not in the partition containing the coordinator think the coordinator has failed, and execute the protocol to deal with failure of the coordinator.

- No harm results, but sites may still have to wait for decision from coordinator.

- The coordinator and the sites are in the same partition as the coordinator think that the sites in the other partition have failed, and follow the usual commit protocol.
  - Again, no harm results

DEADLOCK

System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.

Consider the following two transactions:

T_1: write (*A*) T_2: write(*A*)
write(*B*) write(*B*)

Schedule with deadlock

T_1	T_2
lock-X on *A* write (*A*) wait for **lock-X** on *B*	 **lock-X** on *B* write (*B*) wait for **lock-X** on *A*

Deadlock Handling

Deadlock prevention protocol

Ensure that the system will *never* enter into a deadlock state.

Some prevention strategies :

Approach1

- Require that each transaction locks all its data items before it begins execution either all are locked in one step or none are locked.
- Disadvantages
  - Hard to predict ,before transaction begins, what data item need to be locked.
  - Data item utilization may be very low.

Approach2

- Assign a unique timestamp to each transaction.
- These timestamps only to decide whether a transaction should wait or rollback.

schemes:

- wait-die scheme
- wound-wait scheme

wait-die scheme

- non preemptive technique

When transaction Ti request a data item currently held by Tj, Ti is allowed to wait only if it has a timestamp smaller than that of Tj. otherwise ,Ti rolled back(dies)

- **older transaction may wait for younger one** to release data item. Younger transactions

never wait for older ones; they are rolled back instead.
  - A transaction may die several times before acquiring needed data item

Example.

- Transaction T_1,T_2,T_3 have time stamps 5,10,15,respectively.
- if T_1 requests a data item held by T_2,then T_1 will wait.
- If T_3 request a data item held by T_2,then T_3 will be rolled back.

.wound-wait scheme

- Preemptive technique

- When transaction T_i requests a data item currently held by T_j,T_i is allowed to wait only if it has a timestamp larger than that of T_j. Otherwise T_j is rolled back
  - Older transaction wounds (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.

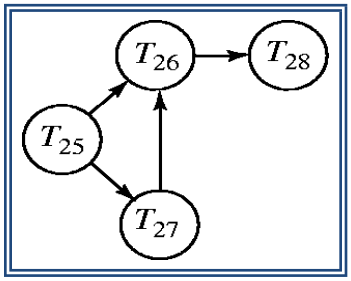
Example

- Transaction T_1,T_2,T_3 have time stamps 5,10,15,respectively.
- if T_1 requests a data item held by T_2,then the data item will be preempted from T_2,and T_2 will be rolled back.
- If T_3 requests a data item held by T_2,then T_3 will wait.

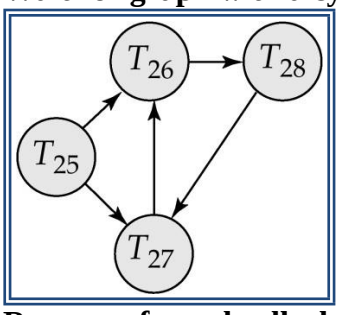
Deadlock Detection

- Deadlocks can be described as a *wait-for graph*, which consists of a pair $G = (V,E)$,
  - V is a set of vertices
  - E is a set of edges
- If $T_i \rightarrow T_j$ is in E, then there is a directed edge from T_i to T_j, implying that T_i is waiting for T_j to release a data item.
- The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.

Wait-for graph without a cycle



Wait-for graph with a cycle



Recovery from deadlock

- The common solution is to roll back one or more transactions to break the deadlock.
- Three action need to be taken
  - Selection of victim

- Rollback
- Starvation

Selection of victim

- Set of deadlocked transations,must determine which transaction to roll back to break the deadlock.
- Consider the factor minimum cost

Rollback

- once we decided that a particular transaction must be rolled back, must determine how far this transaction should be rolled back
- Total rollback
- Partial rollback

Starvation

Ensure that a transaction can be picked as victim only a finite number of times.

Intent locking

- Intent locks are put on all the ancestors of a node before that node is locked explicitly.
- If a node is locked in an intention mode, explicit locking is being done at a lower level of the tree.

Types of Intent Locking

- Intent shared lock(IS)
- Intent exclusive lock(IX)
- Shared lock (S)
- Shared Intent exclusive lock (SIX)
- Exclusive lock (X)

Intent shared lock(IS)

- If a node is locked in indent shared mode, explicit locking is being done at a lower level of the tree, but with only shared-mode lock
- Suppose the transaction T_1 reads record r_{a2}in file $F_{a.}$ Then,T_1 needs to lock the database, area $A_{1,and}$ F_a in IS mode, and finally lock r_{a2} in S mode.

Intent exclusive lock(IX)

If a node is locked in intent locking is being done at a lower level of the tree, but with exclusive mode or shared-mode locks.

- Suppose the transaction T_2 modifies record r_{a9} in file $F_{a.}$ Then,T_2 needs to lock the database, area $A_{1,and}$ F_a in IX mode, and finally to lock r_{a9} in X mode.

Shared Intent exclusive lock (SIX)

If the node is locked in Shared Intent exclusive mode, the subtree rooted by that node is locked explicitly in shared mode, and that explicit locking is being done at lower level with exclusive mode.

Shared lock (S)

-T can tolerate concurrent readers but not concurrent updaters in R.

Exclusive lock (X)

-T cannot tolerate any concurrent access to R at all.

Lock compatibility

Tran 2

Tran 1	NL	IS	IX	S	SIX	X
NL	Yes	Yes	Yes	Yes	Yes	Yes
IS	Yes	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	Yes	No	No	No
S	Yes	Yes	No	Yes	No	No
SIX	Yes	Yes	No	No	No	No
X	Yes	No	No	No	No	

If Tran 1 holds a lock of the given type and Tran 2 requests another lock of the given type will that request be granted?

TRANSACTION RECOVERY

Recovery Algorithms

- Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures
- Recovery algorithms have two parts
    1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures
    2. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

Example

```
Begin transaction
Update Acc 1001{balance:=Balance-100};
If any error occurred then
Goto Undo;
End if;
Update Acc 1002{balance:=balance+100};
If any error occurred then
Goto undo;
End if;
Commit;
Goto finish;
Undo: rollback;
Finish:  return;
```

Requirement for recovery

- Implicit rollback
- Message handling
- Recovery log
- Statement atomicity
- No nested transaction

Transaction recovery

Database updates are kept in buffer in main memory and not physically written to disk until commit**.**

System recovery

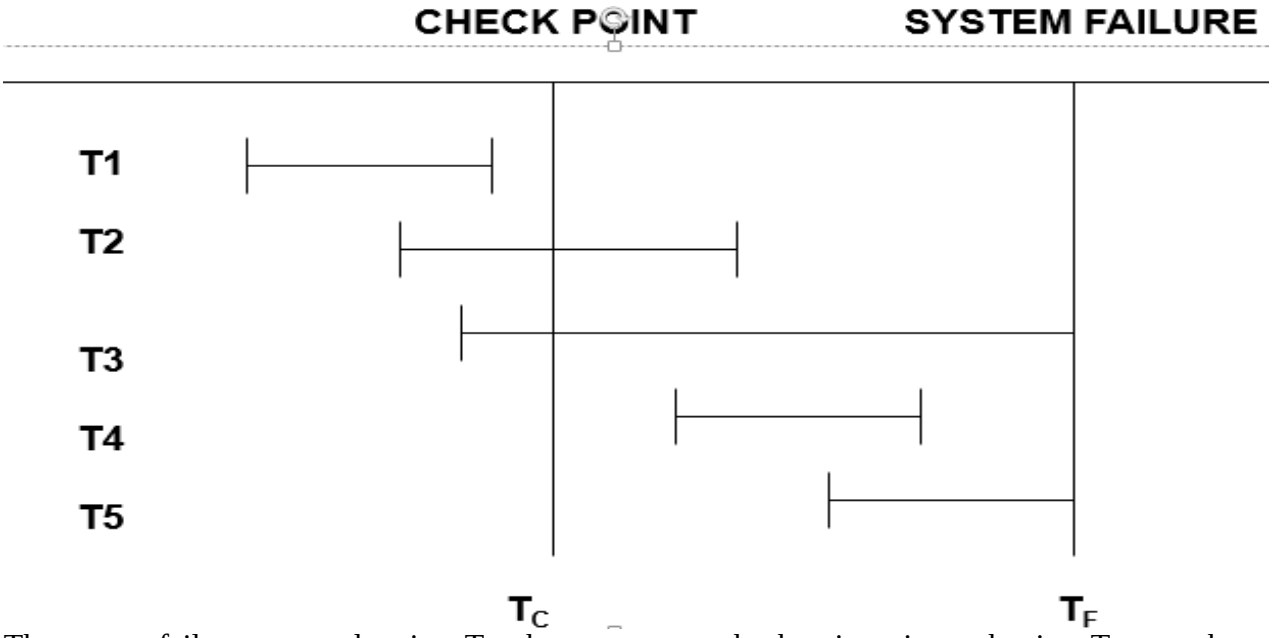
Local failures –affect only the transaction which the failure has actually occurred.

Global failures- affect all the transaction in progress at the time of failure.

System failure – do not physically damage the DB Eg: power shut down
Media failure-cause damage to the DB. Eg: head crash

ARIES Recovery Algorithm

- ARIES-Algorithm for Recovery and Isolation Exploiting Semantics
- ARIES recovery involves three passes
  Analysis pass: Determines the REDO and UNDO lists.
  Redo pass: Repeats history, redoing all actions from REDO List
  Undo pass: Rolls back all incomplete transactions



- The system failure occurred at time T_f , the most recent check point prior to the time T_f was taken at a time T_f
- Start with two list of transaction the UNDO and REDO list
- search forward through the log starting from check point.
- if begin transaction log record is found for transaction(T) add T to UNDO list.
- if commit log record is found for transaction(T),add T to REDO list
- when the end of log record is reached the UNDO and REDO list is identified

UNDO	REDO
T_3	T_2
T_5	T_4

SAVE POINTS

- It is possible for a transaction to create a savepoint.
- It is used to store intermediate results

So that it will rollback to a previously established savepoint whenever any recovery process starts.
Create: Savepoint <savepoint_name>;
Rollback: Rollback to <savepoint_name>;
Drop: Release <savepoint_name>;

SQL

COMMIT: Used to made the changes permanently in the Database.
SAVEPOINT: Used to create a savepoint or a reference point.

ROLLBACK: Similar to the undo operation.

Example:

```
SQL> select * from customer;
   CUSTID    PID      QUANTITY
    ---------- ----------  ----------
      100      1234        10
      101      1235        15
      102      1236        15
      103      1237        10
SQL> savepoint s1;
Savepoint created.
SQL> Delete from customer where custid=103;
   CUSTID   PID      QUANTITY
    ---------- ----------  ----------
      100      1234        10
      101      1235        15
      102      1236        15
SQL> rollback to s1;
Rollback complete.
SQL> select * from customer;
   CUSTID   PID      QUANTITY
   ---------- ---------- ----------
      100      1234        10
      101      1235        15
      102      1236        15
      103      1237        10
SQL> commit;
```

ISOLATION LEVEL

- Degree of interference
- An isolation levels mechanism is used to isolate each transaction in a multi-user environment
- **Dirty Reads:** This situation occurs when transactions read data that has not been committed.
- **Nonrepeatable Reads:** This situation occurs when a transaction reads the same query multiple times and results are not the same each time
- **Phantoms:** This situation occurs when a row of data matches the first time but does not match subsequent times

Types

Higher isolation level (Repeatable read)

- Less interference
- Lower concurrency
- All schedules are serializable

Lower isolation level(cursor stability)

- More interference
- Higher concurrency
- Not a serializable

One special problem that can occur if transaction operates at less than the maximum isolation level (i.e) less then repeatable read level is called phantom problem.

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

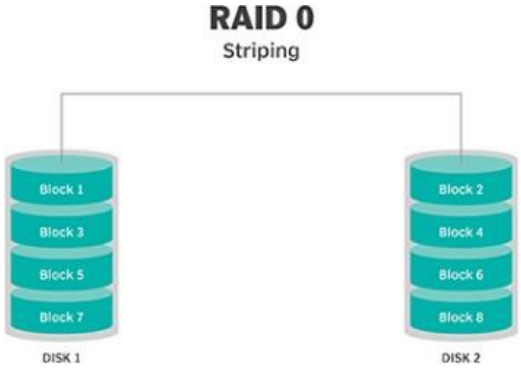
- Just as additional memory in form of cache, can improve the system performance, in the same way additional disks can also improve system performance.
- In RAID we can use an array of disks which operates independently since there are many disks, multiple I/O requests can be handled in parallel if the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

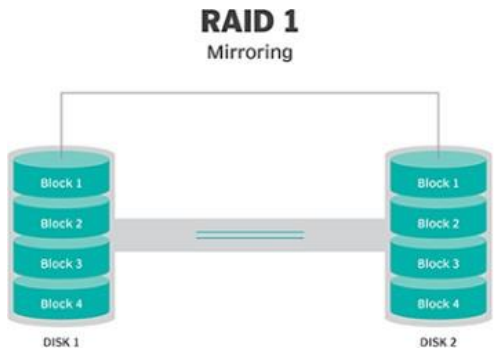
- It is not fault-tolerant, failure of one drive will result in all data in an array being lost

RAID Level 1: **Mirroring** (or **shadowing**)

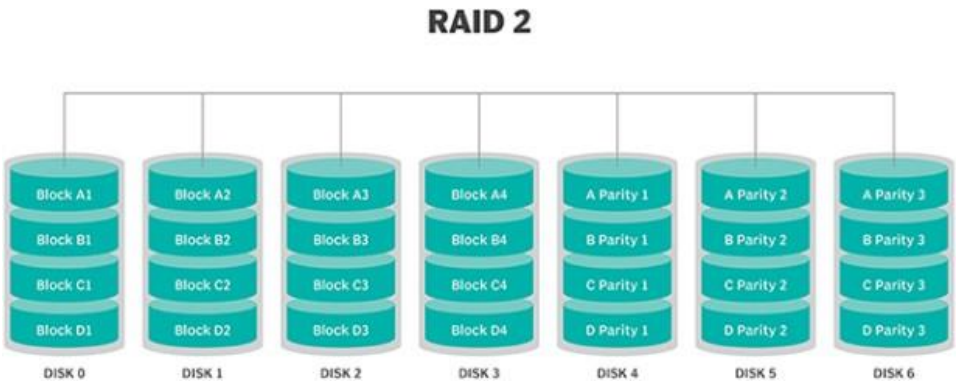
- Also known as *disk mirroring*, this configuration consists of at least two drives that duplicate the storage of

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.

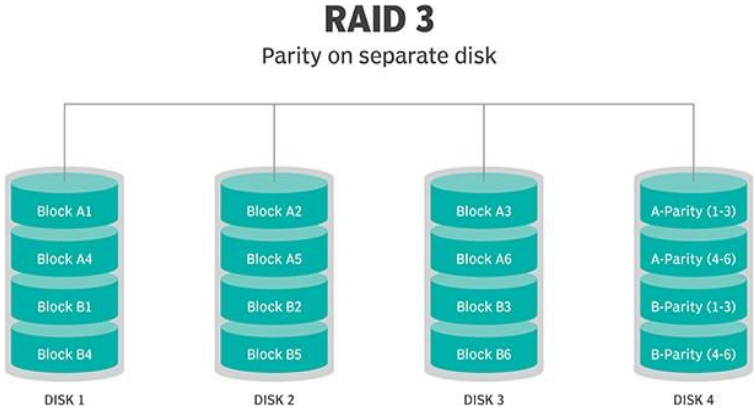


RAID Level 2:
This configuration uses striping across disks, with some disks storing error checking and correcting (ECC) information. It has no advantage over RAID 3 and is no longer used.



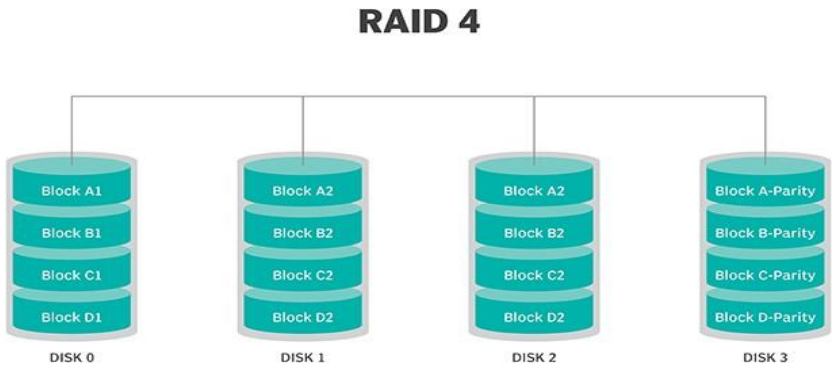
RAID Level 3: Bit-Interleaved Parity

- 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.



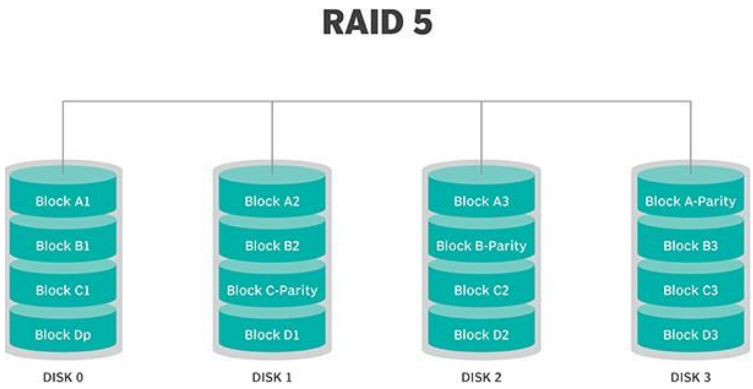
RAID Level 4: Block-Interleaved Parity

- 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.



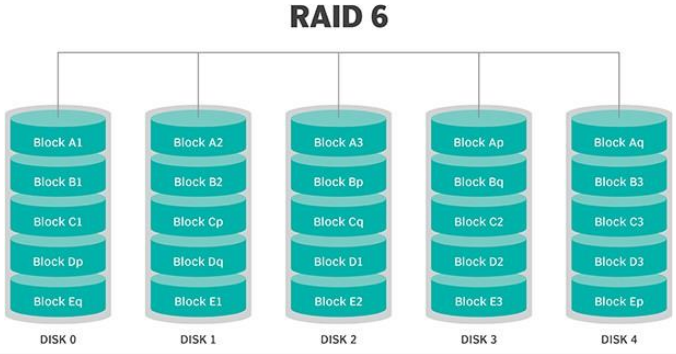
RAID Level 5:

- 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 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.



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

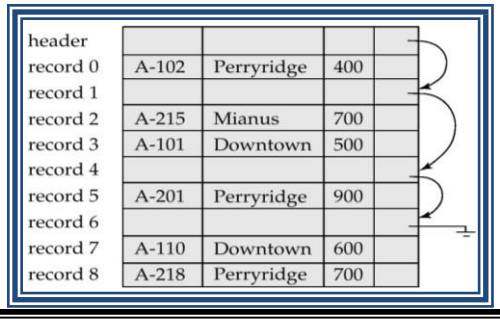
- Difficult to delete record from this structure.
- Some record will cross block boundaries, that is part of the record will be stored in one block and part in another. It would require two block accesses to read or write

Reuse the free space alternatives*:*

- move records i + 1, . . ., n to n i, . . . , 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



Variable-Length Records

Byte string representation

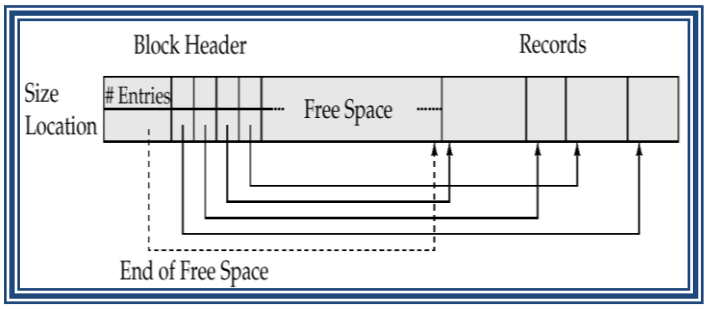
- Attach an *end-of-record* (⊥) control character to the end of each record
- Difficulty with deletion

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

Disadvantage

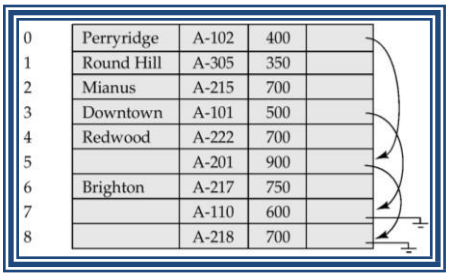
- It is not easy to reuse space occupied formerly by deleted record.
- There is no space in general for records grows longer

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

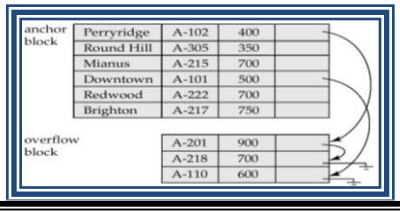


- A variable-length record is represented by a list of fixed-length records, chained together via pointers.
- Can be used even if the maximum record length is not known.

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.

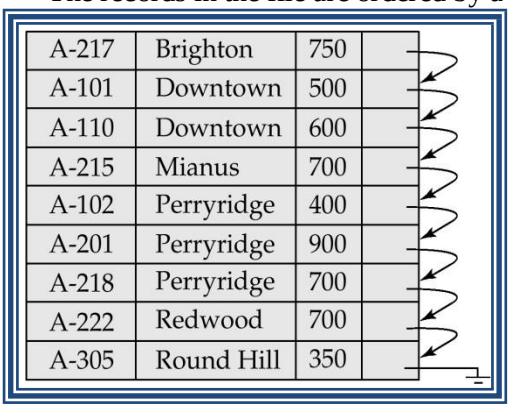


Organization of Records in Files

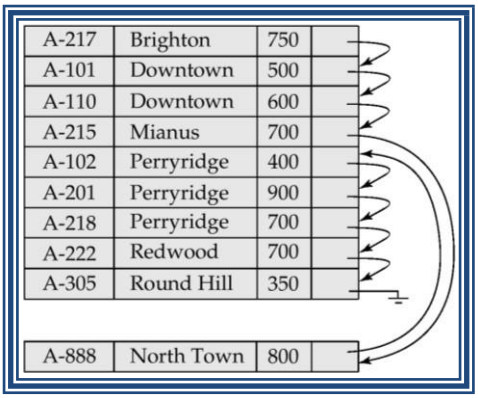
- Sequential – store records in sequential order, based on the value of the search key of each record
- Heap – a record can be placed anywhere in the file where there is space
- Hashing – a hash function computed on some attribute of each record; the result specifies in which block of the file the record 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

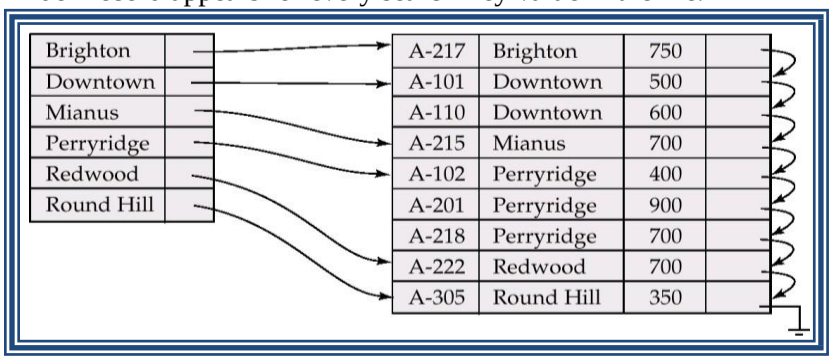
- In an **ordered index,** index entries are stored sorted on the search key value.
- **Primary index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
- **Secondary index**: an index whose search key specifies an order different from the sequential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

Types of Ordered Indices

- Dense index
- Sparse index

Dense Index Files

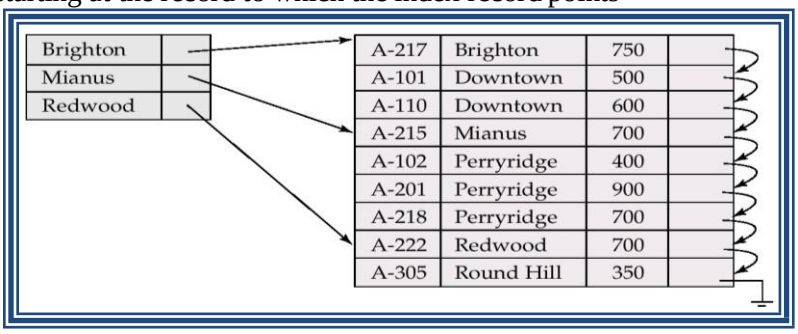
- Dense index — Index record appears for every search-key value in the file.



Sparse Index Files

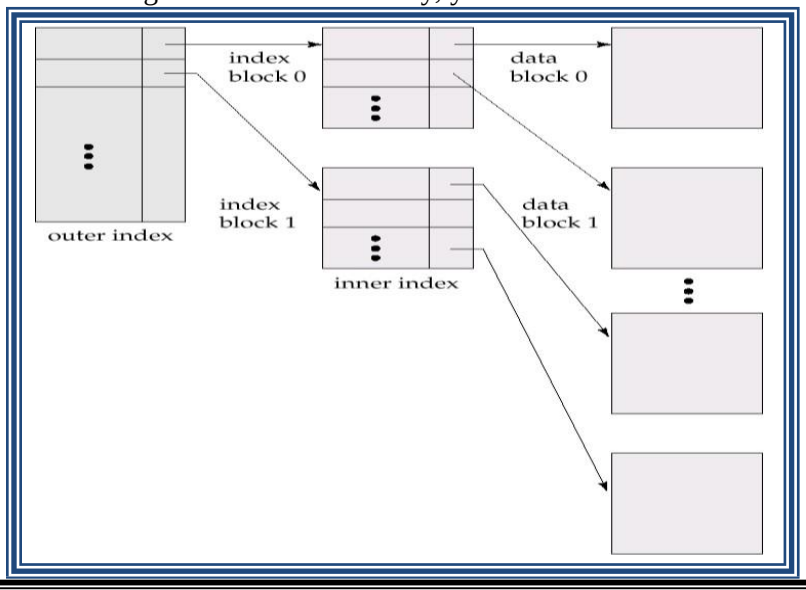
- Sparse Index
  contains index records for only some search-key values.
- To locate a record with search-key value *K* we:
  - Find index record with largest search-key value that is less than or equal to

Search file sequentially starting at the record to which the index record points



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

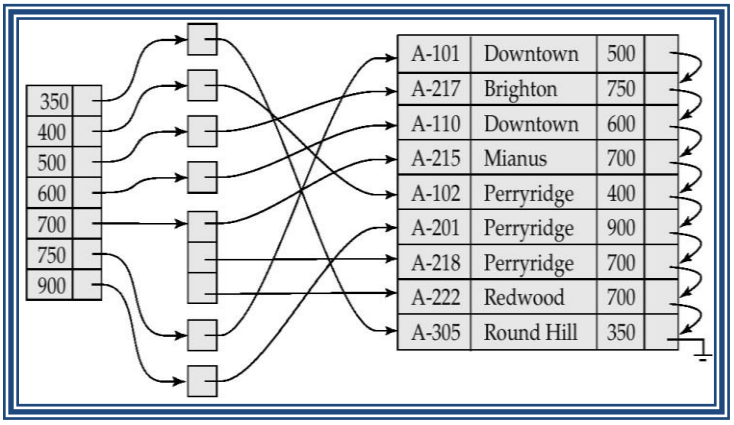
If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.

- **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***



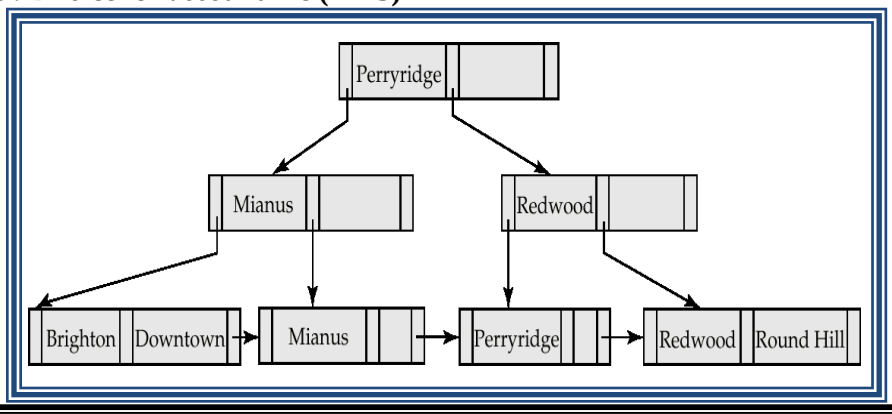
Primary and Secondary Indices

- 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)**



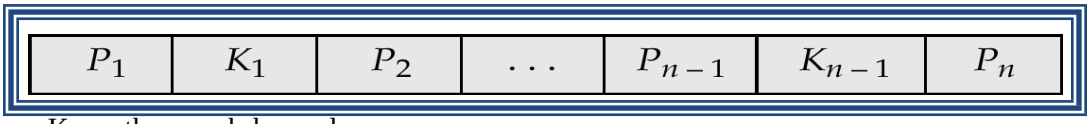
- **Disadvantage of indexed-sequential files**: performance degrades as file grows, since many overflow blocks get created. Periodic reorganization of entire file is required.
- **Advantage of B$^+$-tree index files**: automatically reorganizes itself with small, local, changes, in the face of insertions 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**

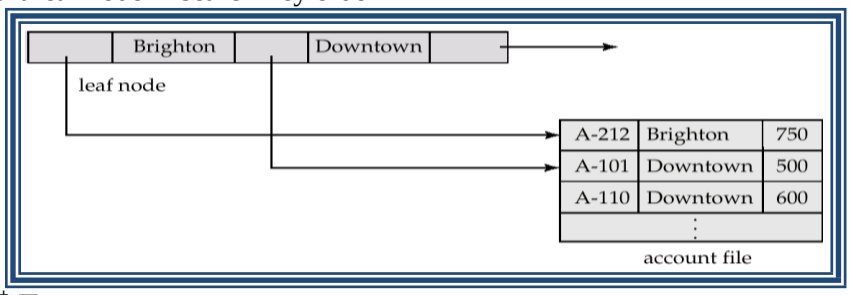


  - K_i are the search-key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

  $K_1 < K_2 < K_3 < \ldots < K_{n-1}$

Properties of Leaf Nodes

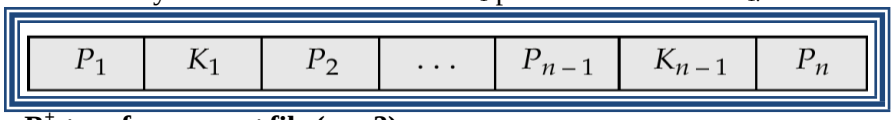
- For i = 1, 2, . . ., n–1, pointer P_i either points to a file record with search-key value K_i, or to a bucket of pointers to file records, each record having search-key value K_i.
- P_n points to next leaf node in search-key order



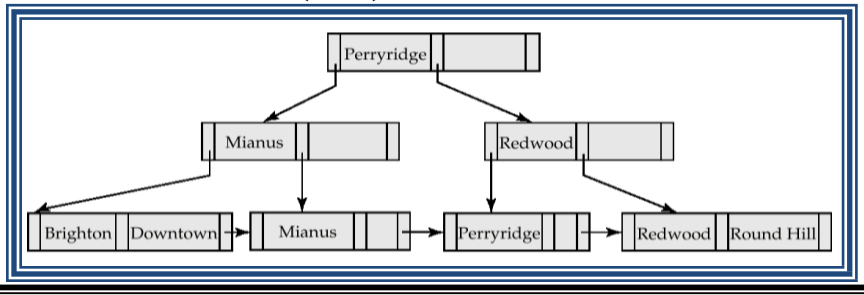
Non-Leaf Nodes in B$^+$-Trees

Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with m pointers:

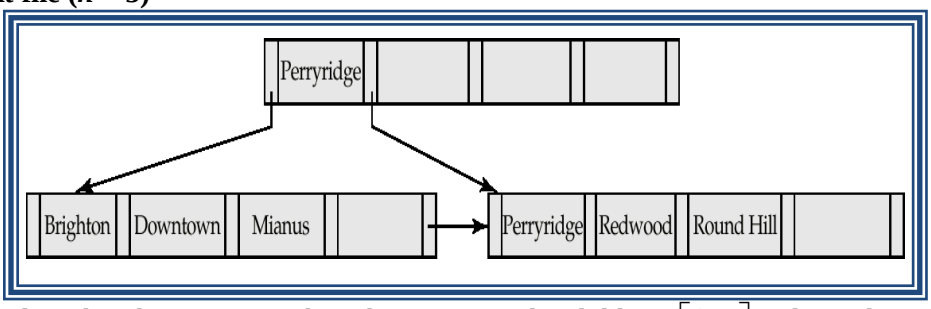
- All the search-keys in the subtree to which P_1 points are less than K_1.



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.

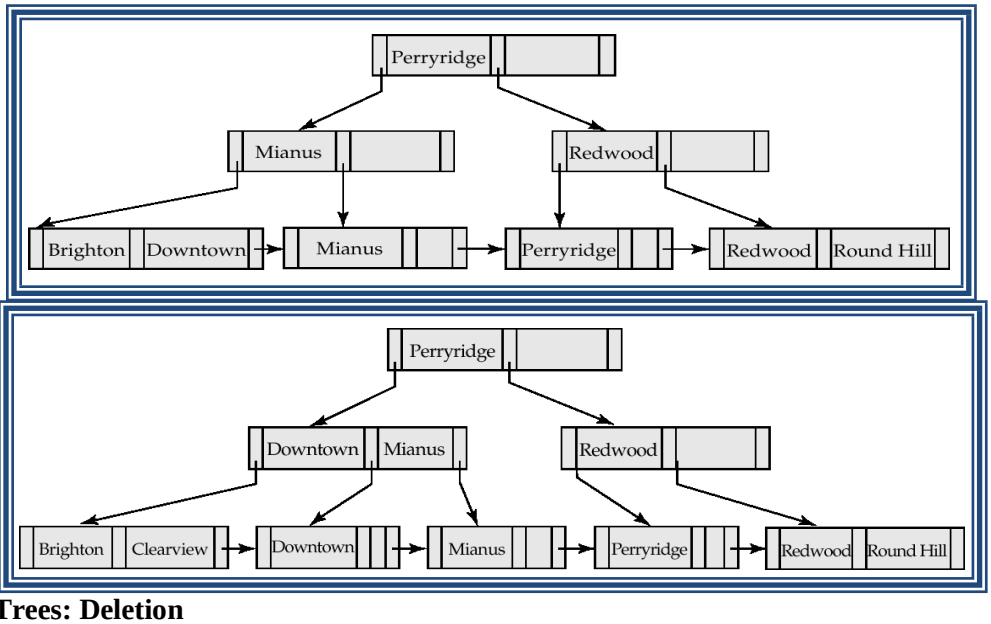
Observations about B^+-trees

- 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"

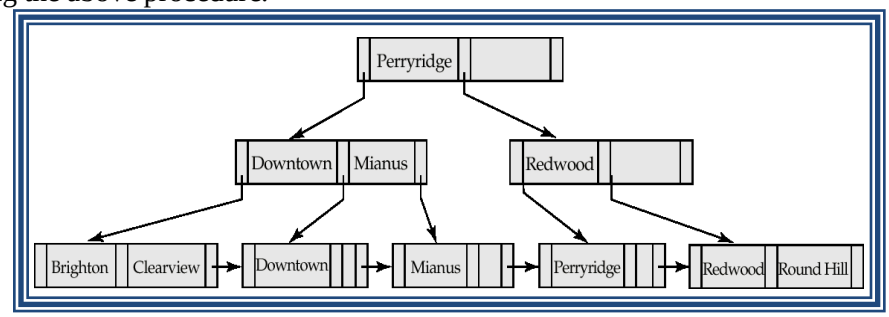


Updates on B^+-Trees: Deletion

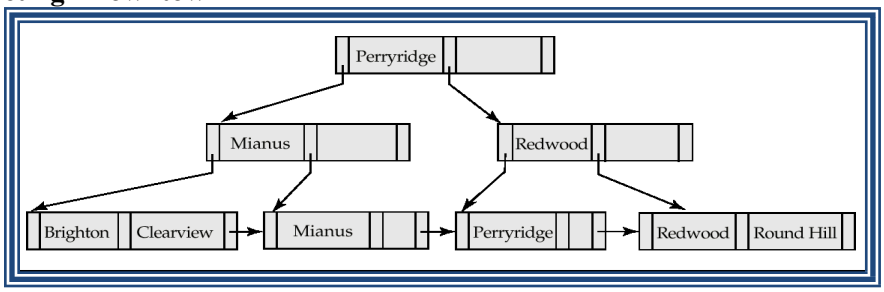
- Find the record to be deleted, and remove it from the main file and from the bucket (if present)
- Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty
- If the node has too few entries due to the removal, and the entries in the node and a sibling 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.

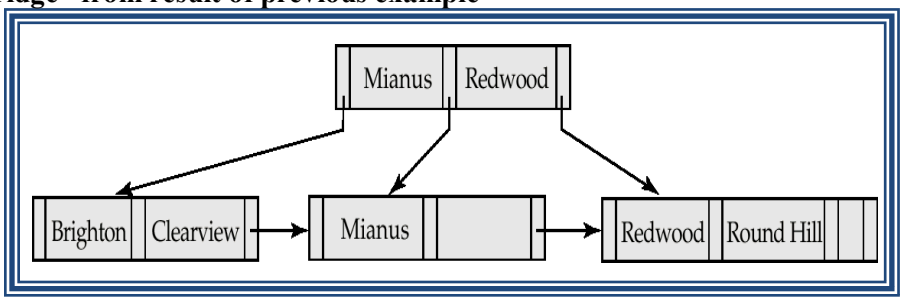


Before and after deleting "Downtown"



- 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.

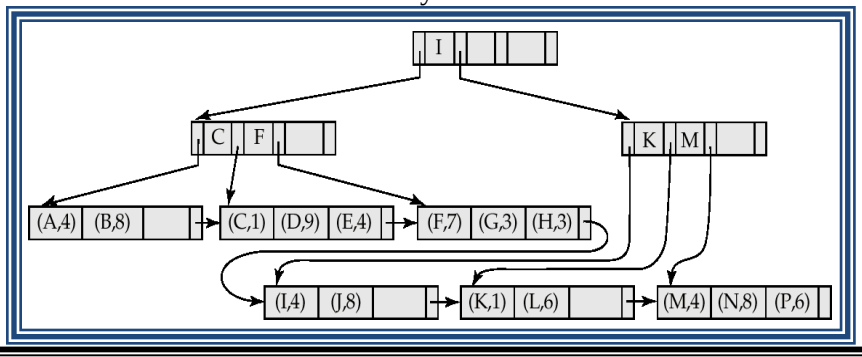
Deletion of "Perryridge" from result of previous example



- 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

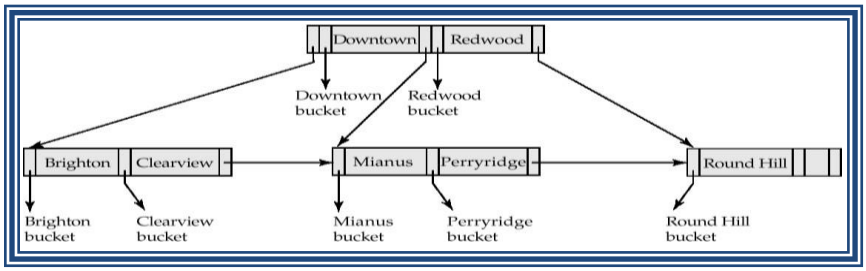
B^+-Tree File Organization

- 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.

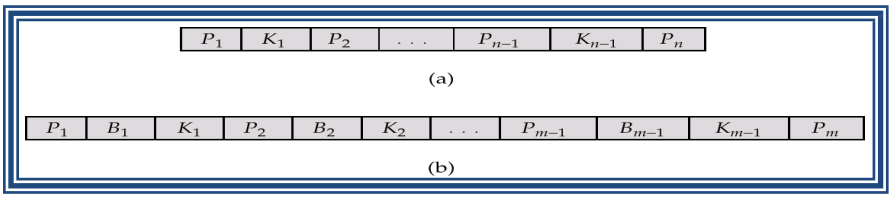


B-Tree Index Files

- 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.

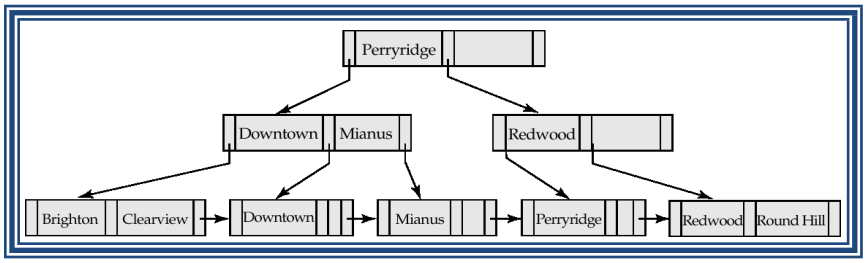


Generalized B-tree leaf node



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

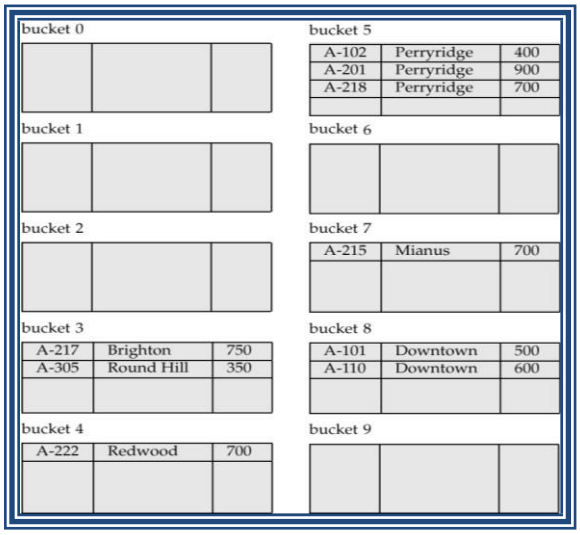
- Static Hashing
- Dynamic Hashing

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



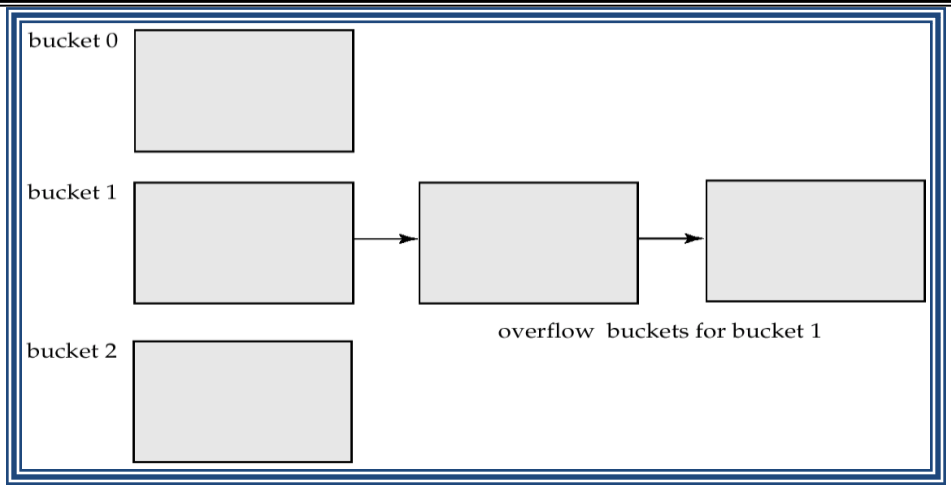
Operation

- **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.

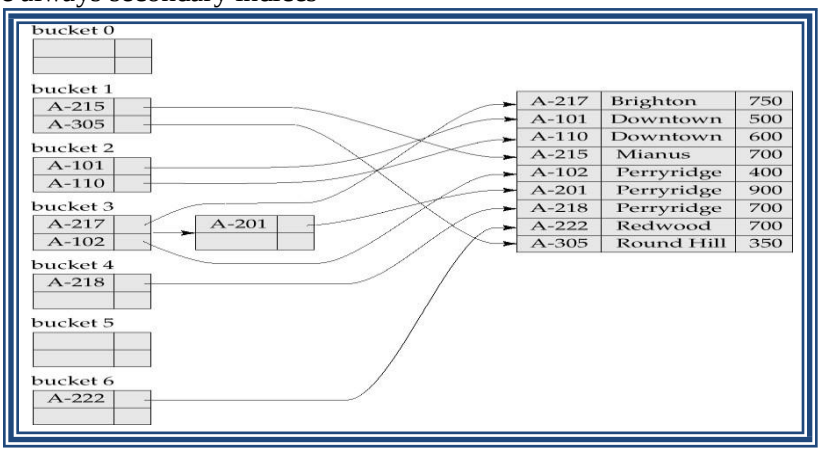
Handling of Bucket Overflows

- 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**.

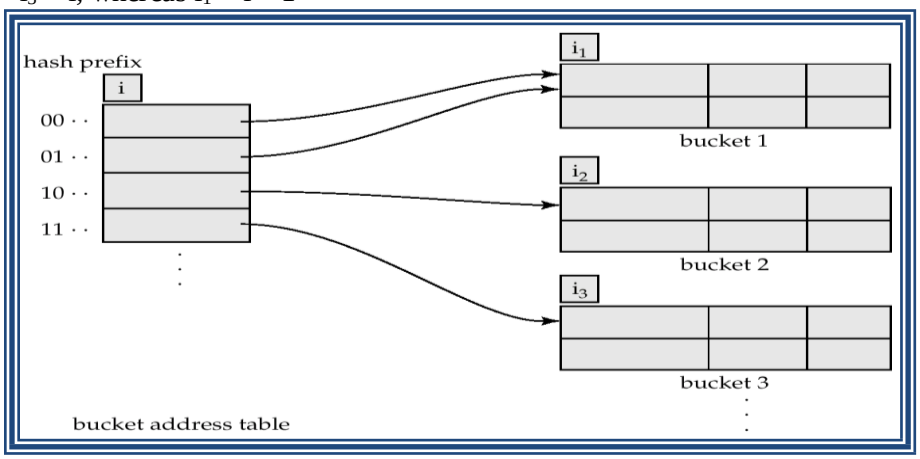
Dynamic Hashing

- 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:

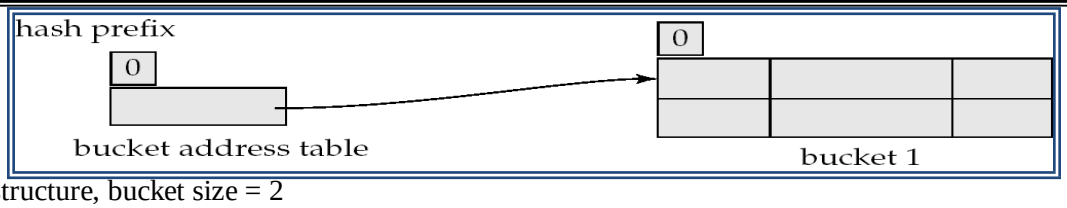
- If $i > i_j$ (more than one pointer to bucket j)
  - allocate a new bucket z, and set $i_j = i_z = (i_j + 1)$
  - Update the second half of the bucket address table entries originally pointing to j, to point to z
  - remove each record in bucket j and reinsert (in j or z)
  - recompute new bucket for K_j 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

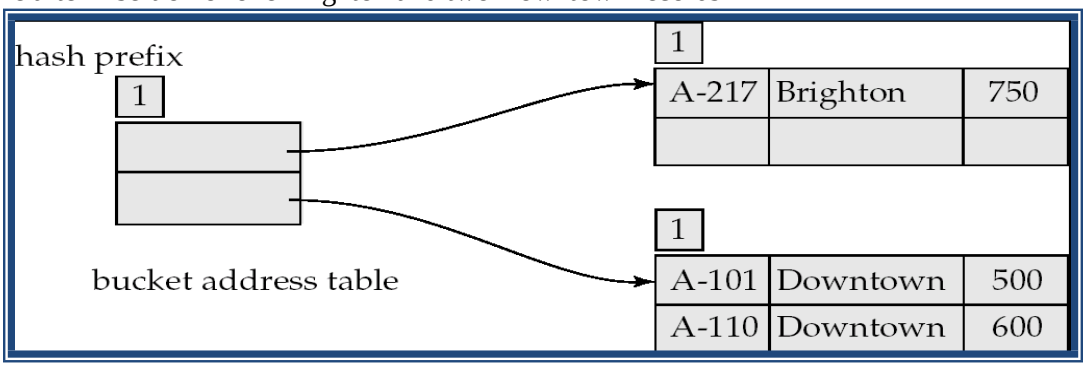
Example

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

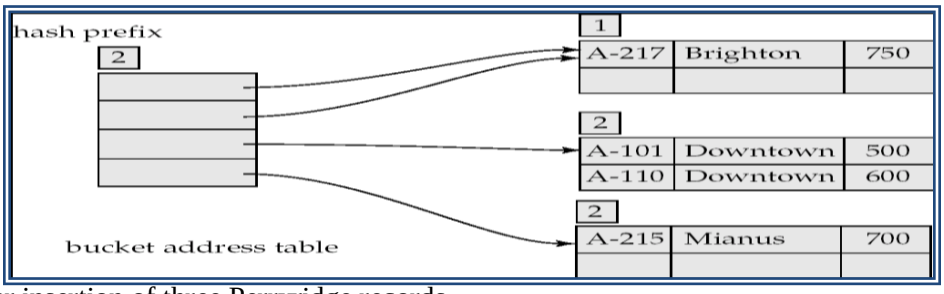


Initial Hash structure, bucket size = 2

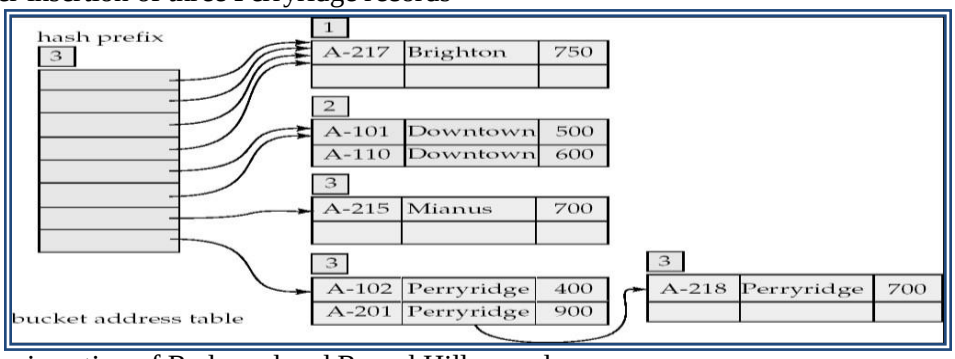
Hash structure after insertion of one Brighton and two Downtown records



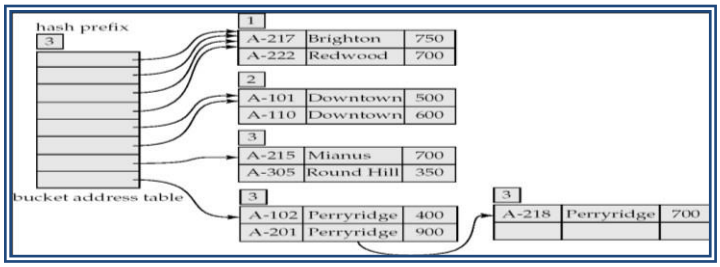
Hash structure after insertion of Mianus record



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

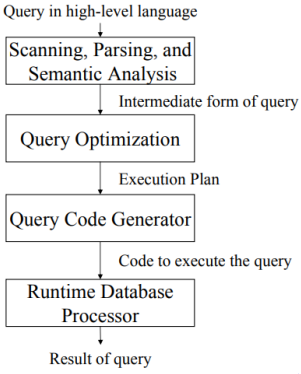
Updates in Extendable Hash Structure

- **To insert a record with search-key value K_j**
  - 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.



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);
```

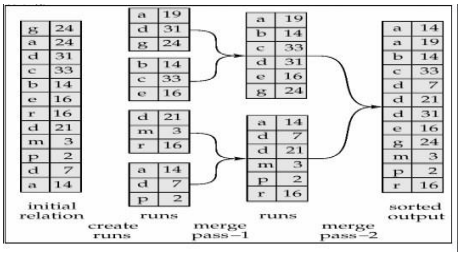
- The inner block
  - (SELECT MAX (SALARY) FROM EMPLOYEE WHERE DNO=5)
  - Translated in:
  - ℑ MAX SALARY(σDNO=5(EMPLOYEE))
- The Outer block
  - SELECT LNAME, FNAME FROM EMPLOYEE WHERE SALARY > C
  - Translated in:
  - ∏ LNAZME, FNAME (σSALARY>C(EMPLOYEE))
    (C represents the result returned from 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..

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

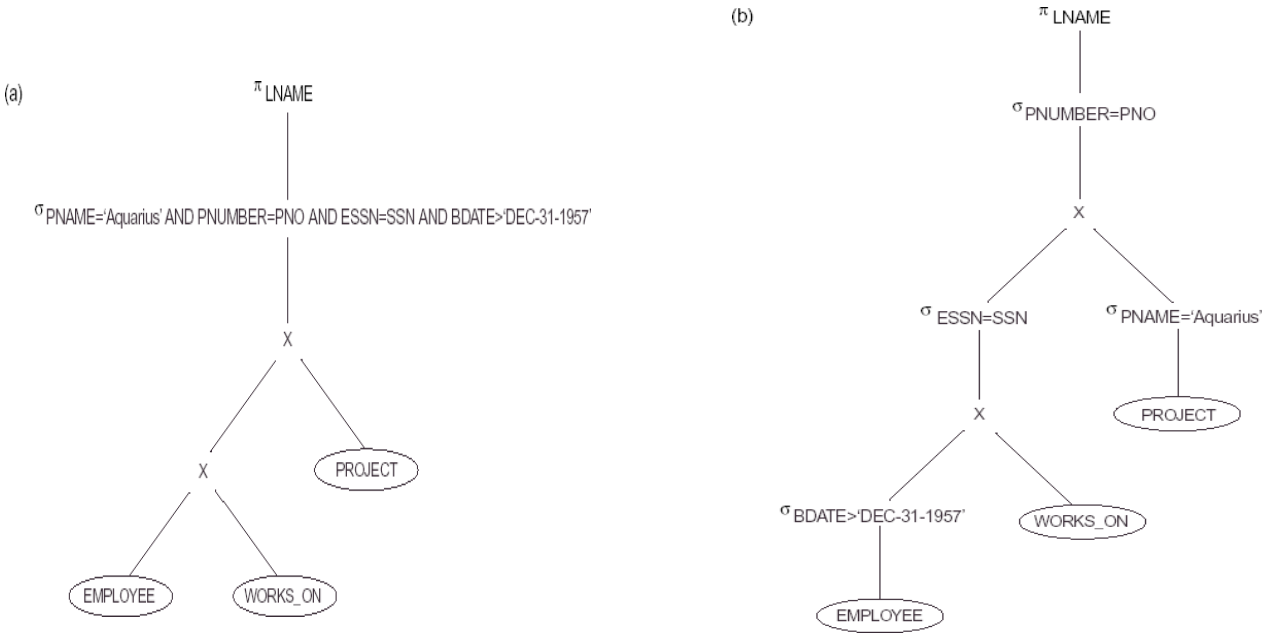
1. Break up SELECT operations with conjunctive conditions into a cascade of SELECT operations
2. Using the commutativity of SELECT with other operations, move each SELECT operation as far down the query tree as is permitted by the attributes involved in the select condition

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.

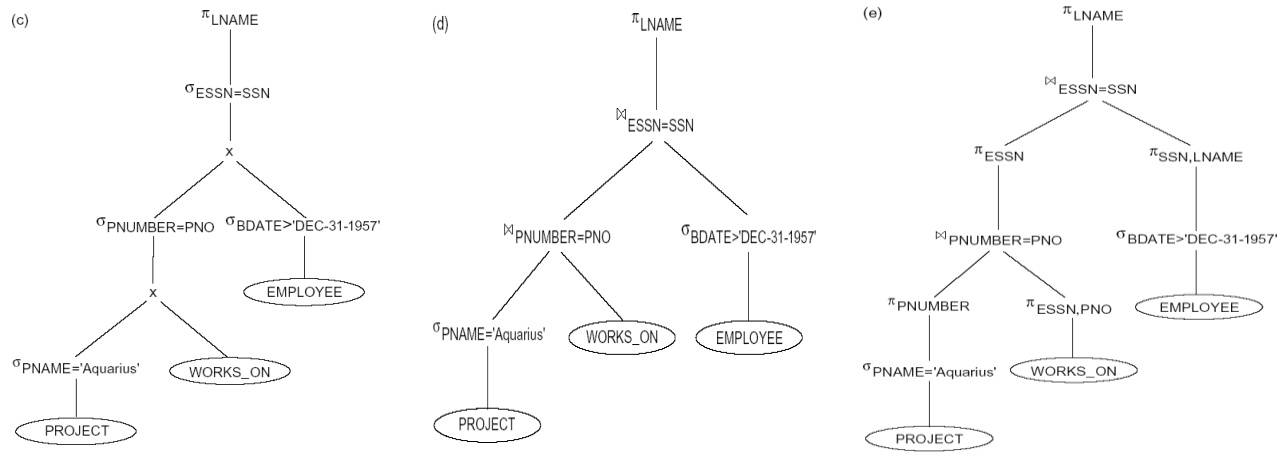
- The query optimizer should not depend solely on heuristic rules, but, it should also estimate the cost of executing the different strategies and find out the strategy with the minimum cost estimate.

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.

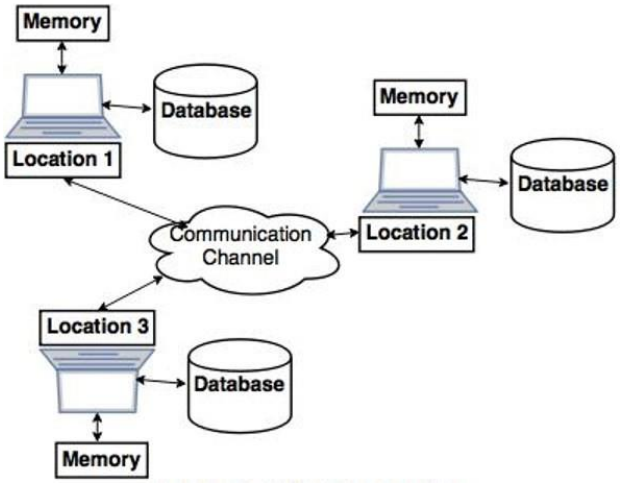
UNIT V ADVANCED TOPICS

Distributed Databases: Architecture, Data Storage, Transaction Processing – Object-based Databases: Object Database Concepts, Object-Relational features, ODMG Object Model, ODL, OQL – XML Databases: XML Hierarchical Model, DTD, XML Schema, XQuery – Information Retrieval: IR Concepts, Retrieval Models, Queries in IR systems.

DISTRIBUTED DATABASES

A distributed database is a database in which not all storage devices are attached to a common processor. It may be stored in multiple computers, located in the same physical location; or may be dispersed over a network of interconnected computers.

- Distributed database is a system in which storage devices are not connected to a common processing unit.
- Database is controlled by Distributed Database Management System and data may be stored at the same location or spread over the interconnected network. It is a loosely coupled system.
- Shared nothing architecture is used in distributed databases.



Distributed Database System

- Communication channel is used to communicate with the different locations and every system has its own memory and database.

Goals of Distributed Database system

Reliability: In distributed database system, if one system fails down or stops working for some time another system can complete the task.

Availability: In distributed database system reliability can be achieved even if sever fails down. Another system is available to serve the client request.

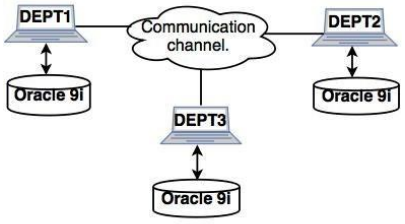
Performance: Performance can be achieved by distributing database over different locations. So the databases are available to every location which is easy to maintain.

TYPES OF DISTRIBUTED DATABASES

1. Homogeneous distributed databases system:

- Homogeneous distributed database system is a network of two or more databases (With same type of DBMS software) which can be stored on one or more machines.
- So, in this system data can be accessed and modified simultaneously on several databases in the network. Homogeneous distributed system are easy to handle.

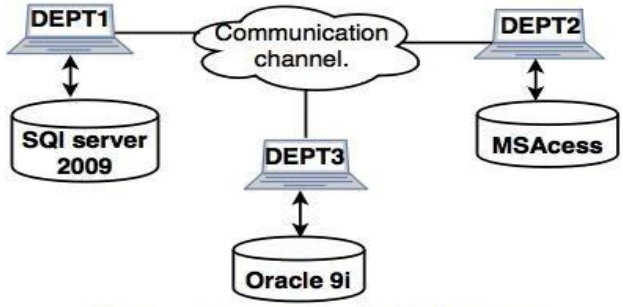
Example: Consider that we have three departments using Oracle-9i for DBMS. If some changes are made in one department then, it would update the other department also.



Homogeneous distributed system

2. Heterogeneous distributed database system.

- Heterogeneous distributed database system is a network of two or more databases with different types of DBMS software, which can be stored on one or more machines.
- In this system data can be accessible to several databases in the network with the help of generic connectivity (ODBC and JDBC).
- **Example:** In the following diagram, different DBMS software are accessible to each other using ODBC and JDBC.



Heterogeneous distributed system

Distributed DBMS Architectures

DDBMS architectures are generally developed depending on three parameters −

Distribution − It states the physical distribution of data across the different sites.

Autonomy − It indicates the distribution of control of the database system and the degree to which each constituent DBMS can operate independently.

Heterogeneity − It refers to the uniformity or dissimilarity of the data models, system components and databases.

Architectural Models

Some of the common architectural models are –

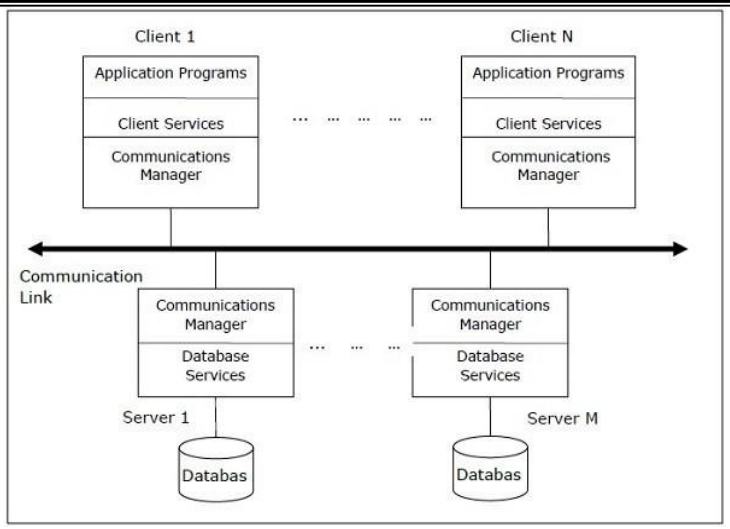
- Client - Server Architecture for DDBMS
- Peer - to - Peer Architecture for DDBMS
- Multi - DBMS Architecture

Client - Server Architecture for DDBMS

This is a two-level architecture where the functionality is divided into servers and clients. The server functions primarily encompass data management, query processing, optimization and transaction management. Client functions include mainly user interface. However, they have some functions like consistency checking and transaction management.

The two different client - server architecture are −

- Single Server Multiple Client
- Multiple Server Multiple Client (shown in the following diagram)



Peer- to-Peer Architecture for DDBMS

In these systems, each peer acts both as a client and a server for imparting database services. The peers share their resource with other peers and co-ordinate their activities.

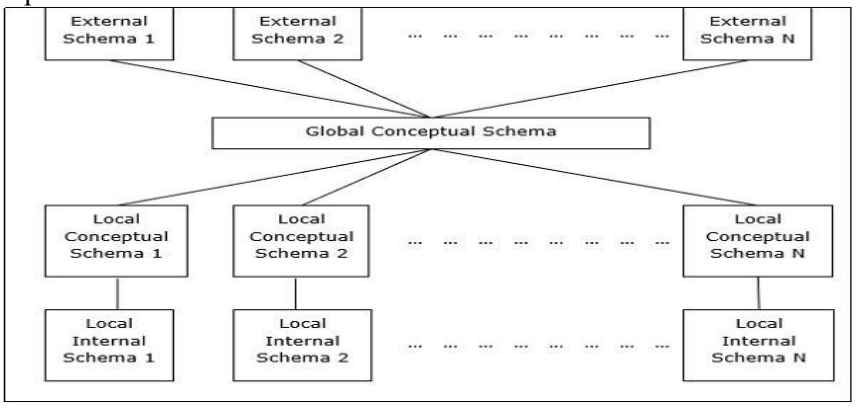
This architecture generally has four levels of schemas −

Global Conceptual Schema − Depicts the global logical view of data.

Local Conceptual Schema − Depicts logical data organization at each site.

Local Internal Schema − Depicts physical data organization at each site.

External Schema − Depicts user view of data.



Multi - DBMS Architectures

This is an integrated database system formed by a collection of two or more autonomous database systems. Multi-DBMS can be expressed through six levels of schemas −

Multi-database View Level − Depicts multiple user views comprising of subsets of the integrated distributed database.

Multi-database Conceptual Level − Depicts integrated multi-database that comprises of global logical multi-database structure definitions.

Multi-database Internal Level − Depicts the data distribution across different sites and multi-database to local data mapping.

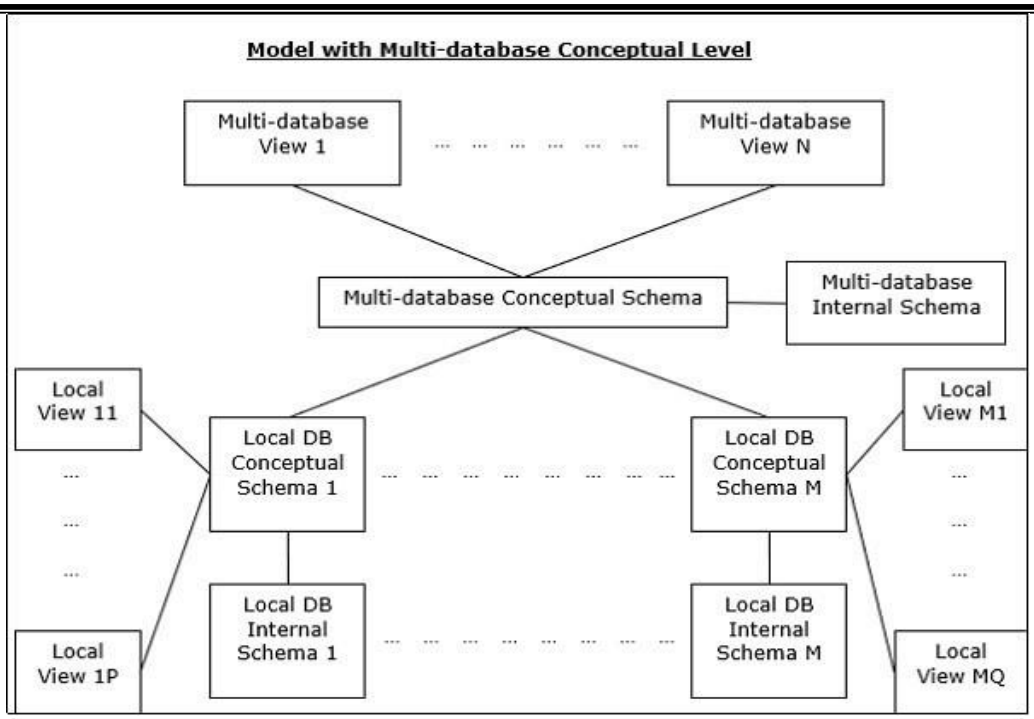
Local database View Level − Depicts public view of local data.

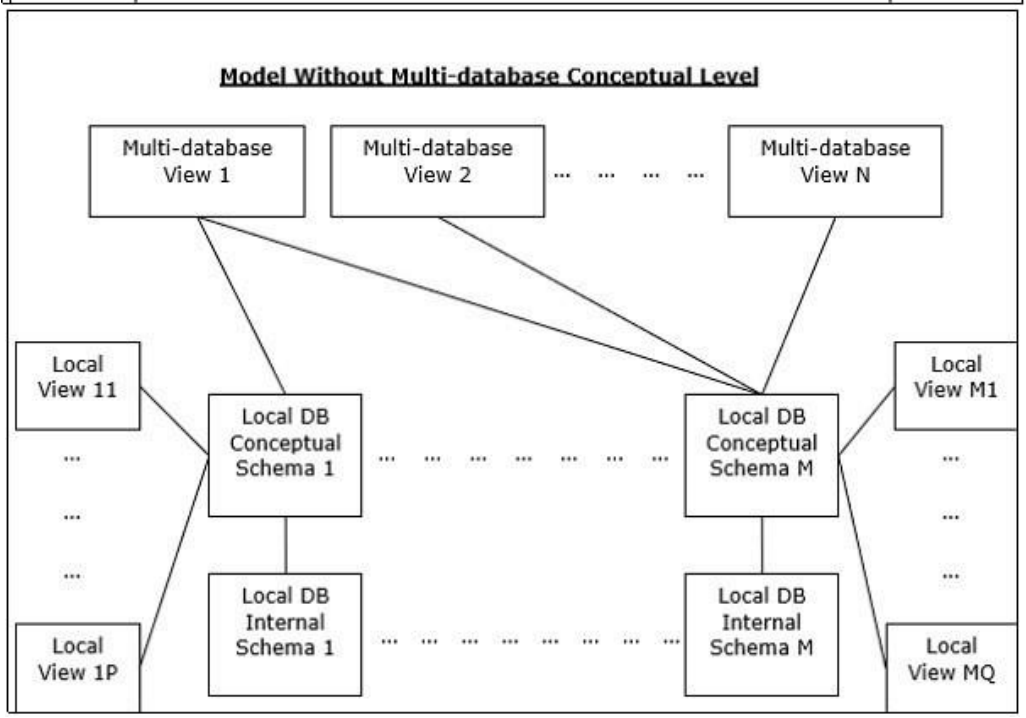
Local database Conceptual Level − Depicts local data organization at each site.

Local database Internal Level − Depicts physical data organization at each site.

There are two design alternatives for multi-DBMS −

- Model with multi-database conceptual level.
- Model without multi-database conceptual level.





DISTRIBUTED DATA STORAGE

Consider a relation r that is to bestored in the database. There are two approaches to storing this relation in the distributed database:

Replication. The system maintains several identical replicas of the relation, and stores each replica at a different site. The alternative to replication is to store only one copy of relation r.

Fragmentation. The system partitions the relation into several fragments, and stores each fragment at a different site.

Data Replication

If relation r is replicated, a copy of relation r is stored in two or more sites. In the most extreme case, we have **full replication**, in which a copy is stored in every site in the system.

There are a number of advantages and disadvantages to replication.

Availability If one of the sites containing relation r fails, then the relation r can be found in another site. Thus, the system can continue to process queries involving r, despite the failure of one site.

Increased parallelism. In the case where the majority of accesses to the relation r result in only the reading of the relation, then several sites can process queries involving r in parallel. The more replicas of r there are, the greater the chance that the needed data will be found in the site where the transaction is executing. Hence, data replication minimizes movement of data between sites. **Increased overhead on update.** The system must ensure that all replicas of a relation r are consistent; otherwise, erroneous computations may result. Thus, whenever is updated, the update must be propagated to all sites containing replicas. The result is increased overhead. For example, in a banking system, where account information is replicated in various sites, it is necessary to ensure that the balance in a particular account agrees in all sites.

Data Fragmentation

If relation r is fragmented, r is divided into a number of fragments r1, r2,...,rn. These fragments contain sufficient information to allow reconstruction of the original relation r.

There are two different schemes for fragmenting a relation: horizontal fragmentation and vertical fragmentation.

- Horizontal fragmentation splits the relation by assigning each tuple of r to one or more fragments.
- Vertical fragmentation splits the relation by decomposing the scheme R of relation r.

In **horizontal fragmentation**, a relation r is partitioned into a number of subsets, r1, r2,...,rn. Each tuple of relation r must belong to at least one of the fragments, so that the original relation can be reconstructed, if needed.

account1 = branch name = “Hillside” (account)

account2 = branch name = “Valleyview” (account)

Horizontal fragmentation is usually used to keep tuples at the sites where they are used the most, to minimize data transfer.

In general, a horizontal fragment can be defined as a selection on the global relation r. That is, we use a predicate Pi to construct fragment ri:

$$r_i = \sigma_{P_i}(r)$$

We reconstruct the relation r by taking the union of all fragments; that is:

$$r = r1 \cup r2 \cup \cdots \cup rn$$

Transparency

The user of a distributed database system should not be required to know where the data are physically located nor how the data can be accessed at the specific local site. This characteristic, called **data transparency**, can take several forms:

Fragmentation transparency. Users are not required to know how a relation has been fragmented.

Replication transparency. Users view each data object as logically unique. The distributed system may replicate an object to increase either system performance or data availability. Users do not have to be concerned with what data objects have been replicated, or where replicas have been placed.

Location transparency. Users are not required to know the physical location of the data. The distributed database system should be able to find any data as long as the data identifier is supplied by the user transaction.

DISTRIBUTED TRANSACTIONS

There are two types of transaction that we need to consider.

- **Local transactions** are those that access and update data in only one local database;
- **Global transactions** are those that access and update data in several local databases.

System Structure

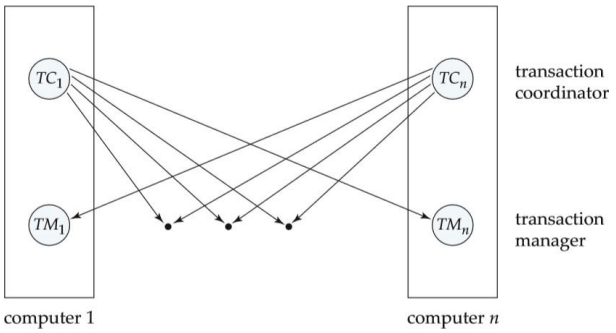
Each site has its own local transaction manager, whose function is to ensure the ACID properties of those transactions that execute at that site. The various transaction managers cooperate to execute global transactions. To

understand how such a manager can be implemented, consider an abstract model of a transaction system, in which each site contains two subsystems:

- The **transaction manager** manages the execution of those transactions (or sub transactions) that access data stored in a local site.
- The **transaction coordinator** coordinates the execution of the various transactions (both local and global) initiated at that site.

Each transaction manager is responsible for:

- Maintaining a log for recovery purposes.
- Participating in an appropriate concurrency-control scheme to coordinate the concurrent execution of the transactions executing at that site.



The transaction coordinator subsystem is not needed in the centralized environment, since a transaction accesses data at only a single site. A transaction coordinator, as its name implies, is responsible for coordinating the execution of all the transactions initiated at that site. For each such transaction, the coordinator is responsible for:

• Starting the execution of the transaction.

• Breaking the transaction into a number of sub transactions and distributing these sub transactions to the appropriate sites for execution.

• Coordinating the termination of the transaction, which may result in the transaction being committed at all sites or aborted at all sites.

System Failure Modes

- Failure of a site.
- Loss of messages.
- Failure of a communication link.
- Network partition.

OBJECT-BASED DATABASES

An object-oriented database system is a database system that natively supports an object-oriented type system, and allows direct access to data from an object-oriented programming language using the native type system of the language.

Complex Data Types

Traditional database applications have conceptually simple datatypes. The basic data items are records that are fairly small and whose fields are atomic.

In recent years, demand has grown for ways to deal with more complex data types. Consider, for example, addresses. While an entire address could be viewed as an atomic data item of type string, this view would hide details such as the street address, city, state, and postal code, which could be of interest to queries. On the other hand, if an address were represented by breaking it into the components (street address, city, state, and postal code), writing queries would be more complicated since they would have to mention each field. A better alternative is to allow structured datatypes that allow a type address with subparts street address, city, state, and postal code.

Structured Types

Structured types allow composite attributes of E-R designs to be represented directly. For instance, we can define the following structured type to represent a composite attribute name with component attribute *firstname* and *lastname:*

```
create type Name as
(firstname varchar(20),
lastname varchar(20))
final;
```

Such types are called **user-defined** types in SQL. The **final** and **not final** specifications are related to subtyping.

The components of a composite attribute can be accessed using a **"dot"** notation; for instance, name. firstname returns the firstname component of the name attribute. An access to attribute name would return a value of the structured type Name.

We can also create a table whose rows are of a user-defined type. For example, we could define a type Person Type and create the table person as follows

```
create type PersonType as (
name Name,
address  Address,
dateOfBirth date)
not final
create table person of PersonType;
```

Type Inheritance

Suppose that we have the following type definition for people:

```
create type Person
(name varchar(20),
address varchar(20));
```

We may want to store extra information in the database about people who are students, and about people who are teachers. Since students and teachers are also people, we can use inheritance to define the student and teacher types in SQL:

```
create type Student
under Person
(degree varchar(20),
 department varchar(20));
create type Teacher
under Person
(salary integer,
department varchar(20));
```

Both Student and Teacher inherit the attributes of Person—namely, name and address. Student and Teacher are said to be subtypes of Person, and Person is a supertype of Student, as well as of Teacher.

Table Inheritance

Sub tables in SQL correspond to the E-R notion of specialization/generalization. For instance, suppose we define the people table as follows:

```
create table people of Person;
```

We can then define tables students and teachers as sub tables of people, as follows:

```
create table students of Student
under people;
create table teachers of Teacher
under people;
```

The types of the sub tables (Student and Teacher in the above example) are subtypes of the type of the parent table (Person in the above example). As a result, every attribute present in the table people is also present in the sub tables students and teachers.

Array and Multiset Types in SQL

SQL supports two collection types: arrays and multisets

A multiset is an unordered collection, where an element may occur multiple times.

Multisets are like sets, except that a set allows each element to occur at most once.

Suppose we wish to record information about books, including a set of keywords for each book. Suppose also that we wished to store the names of authors of a book as an array; unlike elements in a multiset, the elements of an array are ordered, so we can distinguish the first author from the second author, and so on. The following example illustrates how these array and multiset-valued attributes can be defined in SQL:

```
create type Publisher as
(name varchar(20),
branch varchar(20));
create type Book as
(title varchar(20),
Autho_arrray varchar(20) array [10],
Pub_date date, publisher Publisher, keyword_set varchar(20) multiset);
create table books of Book;
```

The first statement defines a type called Publisher with two components: a name and a branch. The second statement defines a structured type Book that contains a title, an author array, which is an array of up to 10 author names, a publication date, a publisher (of type Publisher), and a multiset of keywords. Finally, a table books containing tuples of type Book is created.

Object-Identity and Reference Types in SQL

Object-oriented languages provide the ability to refer to objects. An attribute of a type can be a reference to an object of a specified type. For example, in SQL we can define a type *Department* with a field *name* and a field *head* that is a reference to the type *Person,* and a table *departments* of type *Department*, as follows:

```
create type Department (
 name varchar(20),
head ref(Person) scope people);
create table departments of Department;
```

Here, the reference is restricted to tuples of the table people. The restriction of the scope of a reference to tuples of a table is mandatory in SQL, and it makes references behave like foreign keys.

Object-relational Features

Object-relational database systems are basically extensions of existing relational database systems. Changes are clearly required at many levels of the database system. However, to minimize changes to the storage-system code (relation storage, indices, etc.), the complex datatypes supported by object-relational systems can be translated to the simpler type system of relational databases.

Sub tables can be stored in an efficient manner, without replication of all inherited fields, in one of two ways:

- Each table stores the primary key (which may be inherited from a parent table) and the attributes that are defined locally. Inherited attributes (other than the primary key) do not need to be stored, and can be derived by means of a join with the super table, based on the primary key.
- Each table stores all inherited and locally defined attributes. When a tuple is inserted, it is stored only in the table in which it is inserted, and its presence is inferred in each of the super tables. Access to all attributes of a tuple is faster, since a join is not required.

The ODMG· Object Model

The **ODMG object model** is the data model upon which the object definition language (ODL) and object query language (OQL) are based. It is meant to provide a standard data model for object databases, just as SQL describes a standard data model for relational databases. It also provides a standard terminology in a field where the same terms were sometimes used to describe different concepts.

Objects and Literals

Objects and literals are the basic building blocks of the object model. The main difference between the two is that an object has both an **object identifier** and a **state** (or current value), whereas a literal has a value (state) but **no object identifier**. In either case, the value can have a complex **structure. The object** state can change over time by modifying the object value. A literal is basically a constant value, possibly having a complex structure, but it does not change.

An object has five aspects: identifier, name, lifetime, structure, and creation.

1. The **object identifier** is a unique system-wide identifier (or Object_id). Every object must have an object identifier.
2. Some objects may optionally be given a unique **name** within a particular ODMS—this name can be used to locate the object, and the system should return the object given that name. Obviously, not all individual objects will have unique names. Typically, a few objects, mainly those that hold collections of objects of a particular object type—such as ***extents***—will have a name. These names are used as ***entry points*** to the database; that is, by locating these objects by their unique name, the user can then locate other objects that are referenced from these objects. Other important objects in the application may also have unique names, and it is possible to give *more than one name* to an object. All names within a particular ODMS must be unique.
3. The ***lifetime*** of an object specifies whether it is a *persistent object* (that is, a database object) or *transient object* (that is, an object in an executing pro-gram that disappears after the program terminates). Lifetimes are independent of types—that is, some objects of a particular type may be transient whereas others may be persistent.
4. The ***structure*** of an object specifies how the object is constructed by using the type constructors. The structure specifies whether an object is ***atomic*** or not. An ***atomic object*** refers to a single object that follows a user-defined type, such as *Employee* or *Department*. If an object is not atomic, then it will be composed of other objects. For example, a *collection object* is not an atomic object, since its state will be a collection of other objects. In the ODMG model, an atomic object is any *individual user-defined object*. All values of the basic built-in data types are considered to be *literals*.
5. ***Object creation*** refers to the manner in which an object can be created. This is typically accomplished via an operation new for a special ***Object_Factory*** interface.

In the object model, a ***literal*** is a value that *does not have* an object identifier. However, the value may have a simple or complex structure.

There are three types of literals: atomic, structured, and collection.

1. **Atomic literals** correspond to the values of basic data types and are predefined. The basic data types of the object model include long, short, and unsigned integer numbers (these are specified by the keywords *long, short, unsigned long, and unsigned short* in ODL), regular and double precision floating point numbers *(float, double)*, Boolean values *(boolean*), single characters *(char),* character strings *(string),* and enumeration types *(enum),* among others.
2. ***Structured literals*** correspond roughly to values that are constructed using the tuple constructor. The built-in structured lit-erals include *Date, Interval,Time, and Timestamp.*
3. ***Collection literals*** specify a literal value that is a collection of objects or values but the collection itself does not have an Object_id. The collections in the object model can be defined by the type generators set<T>, bag<T>, list<T>, and array<T>, where T is the type of objects or values in the collection.28 Another collection type is dictionary<K, V>, which is a collection of associations <K, V>, where each K is a key (a unique search value) associated with a value V; this can be used to create an index on a collection of values V.

The notation of ODMG uses three concepts: ***interface, literal, and class.*** Following the ODMG terminology, we use the word **behavior** to refer to *operations* and **state** to refer to *properties* (attributes and relationships).

An **interface** specifies only behavior of an object type and is typically ***noninstantiable*** (that is, no objects are created corresponding to an interface). Although an interface may have state properties (attributes and relationships) as part of its specifications, these cannot be inherited from the interface. Hence, an interface serves to define operations that can be inherited by other interfaces, as well as by classes that define the user-defined objects for a particular application.

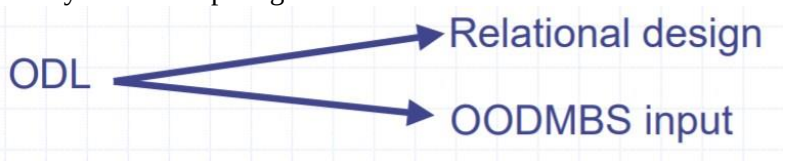
A **class** specifies both state (attributes) and behavior (operations) of an object type, and is instantiable. Hence, database and application objects are typically created based on the user-specified class declarations that form a database schema.

Finally, a **literal** declaration specifies state but no behavior. Thus, a literal instance holds a simple or

complex structured value but has neither an object identifier nor encapsulated operations.

ODL: OBJECT DEFINITION LANGUAGE

Object Definition Language (ODL) is the specification language defining the interface to object types conforming to the ODMG Object Model. Often abbreviated by the acronym ODL.This language's purpose is to define the structure of an Entity-relationship diagram.



Class Declarations

- interface < name > {elements = attributes, relationships, methods }

Element Declarations

- attribute < type > < name > ;
- **relationship < rangetype > < name > ;**

Method Example

- float gpa(in: Student) raises(noGrades) float = return type.
- in: indicates Student argument is read-only.
  Other options: out, inout.

Relationships

- use inverse to specify inverse relationships
- at most one' semantics remain
- multiplicity
  - if many-many between C and D, then use Set<D> and Set<C>, respectively
  - if many-one from C to D, then use D in C and Set<C> in D
  - if many-one from D to C, then use C in D and Set<D> in C
  - if one-one between C and D, then use D and C, respectively

Datatypes

- basis
  - atomic: integer, float, character, character string, boolean, and enumeration
  - classes
- type constructors (can be composed to create complex types)
  - set: Set<T>
  - bag: Bag<T>
  - list: List<T> (sequential access)
  - array: Array<T,i> (random access)
  - dictionary: Dictionary<T,S>
  - structures
- difference between sets, bags, and lists
- rules for types and relationships
  - type of a relationship is either a class type or a (single use of a) collection type constructor applied to a class type' [FCDB]
  - type of an attribute is built starting with atomic type(s)' [FCDB]
- relationship types cannot involve
  - atomic types (e.g., Set<integer>),
  - structures (e.g., Struct N {Movie field1, Star field2}, or
  - two applications of collection types (e.g., Set<Array<Star, 10>>)

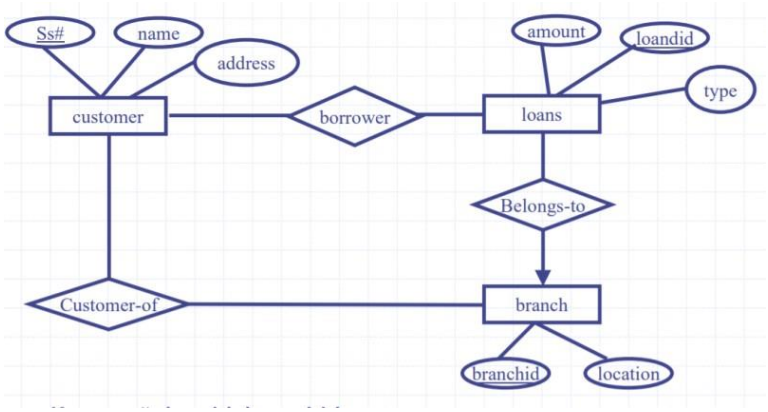
Similarities between E/R and ODL

- both support all multiplicities of relationships
- both support inheritance

Differences between E/R and ODL

- attributes are single-valued in E/R; can be multivalued in ODL
- keys required in E/R, optional in ODL
- can only model one key in E/R; can model all (or any subset) in ODL
- multiway relationships supported in E/R; only binary relationships supported in ODL
- no multiple inheritance in E/R; ODL supports multiple inheritance

Banking Example 1



- **Keys:** ss#, loanid, branchid
- **Cardinality constraint**: each loan belongs to a single branch.

OQL - OBJECT QUERY LANGUAGE

OQL is the way to access data in an O2 database. OQL is a powerful and easy-to-use SQL-like query language with special features dealing with complex objects, values and methods.

SELECT, FROM, WHERE

SELECT <list of values>
FROM <list of collections and variable assignments>
WHERE <condition>

The SELECT clause extracts those elements of a collection meeting a specific condition. By using the keyword DISTINCT duplicated elements in the resulting collection get eliminated. Collections in FROM can be either extents (persistent names - sets) or expressions that evaluate to a collection (a set). Strings are enclosed in double-quotes in OQL. We can rename a field by if we prefix the path with the desired name and a colon.

Example Query 1

Give the names of people who are older than 26 years old:

```
SELECT SName: p.name
FROM p in People
WHERE p.age > 26
(hit Ctrl-D)
```

Dot Notation & Path Expressions

We use the dot notation and path expressions to access components of complex values.

Let variables t and ta range over objects in extents (persistent names) of Tutors and TAs (i.e., range over objects in sets Tutors and TAs).

ta.salary -> real

t.students -> set of tuples of type tuple(name: string, fee: real) representing students

alary -> real

Cascade of dots can be used if all names represent objects and not a collection.

Example Query 2

Find the names of the students of all tutors:

```
SELECT s.name
```

FROM Tutors t, t.students s

Here we notice that the variable t that binds to the first collection of FROM is used to help us define the second collection s. Because students is a collection, we use it in the FROM list, like t.students above, if we want to access attributes of students.

Subqueries in FROM Clause

Example Query 3

Give the names of the Tutors which have a salary greater than $300 and have a student paying more than $30:

```
SELECT t.name
FROM ( SELECT t FROM Tutors t WHERE t.salary > 300 ) r, r.students s
WHERE s.fee > 30
```

Subqueries in WHERE Clause

Example Query 4

Give the names of people who aren't TAs:

```
SELECT p.name
FROM p in People
WHERE not ( p.name in SELECT t.name FROM t in TAs )
```

Set Operations and Aggregation

The standard O2C operators for sets are + (union), * (intersection), and - (difference). In OQL, the operators are written as UNION, INTERSECT and EXCEPT , respectively.

Example Query 5

Give the names of TAs with the highest salary:

```
SELECT t.name
FROM t in TAs
WHERE t.salary = max ( select ta.salary from ta in TAs )
```

GROUP BY

The GROUP BY operator creates a set of tuples with two fields. The first has the type of the specified GROUP BY attribute. The second field is the set of tuples that match that attribute. By default, the second field is called PARTITION.

Example Query 6

Give the names of the students and the average fee they pay their Tutors:

```
SELECT sname, avgFee: AVG(SELECT p.s.fee FROM partition p)
FROM t in Tutors, t.students s
GROUP BY sname: s.name
```

Embedded OQL

Instead of using query mode, you can incorporate these queries in your O2 programs using the "o2query" command:

```
run body {
 o2 real total_salaries;
o2query( total_salaries, "sum ( SELECT ta->get_salary \
FROM ta in TAs )" );
printf("TAs combined salary: %.2f\n", total_salaries);
};
```

The first argument for o2query is the variable in which you want to store the query results. The second argument is a string that contains the query to be performed. If your query string takes up several lines, be sure to backslash (\) the carriage returns.

XML DATABASES

XML Hierarchical (Tree) Data Model

The basic object in XML is the XML document. Two main structuring concepts are used to construct an XML document: elements and attributes.

An example of an XML element called . As in HTML, elements are identified in a document by their start tag and end tag. The tag names are enclosed between angled brackets < ... >, and end tags are further identified by a

slash.</…..>.

Complex elements are constructed from other elements hierarchically, whereas **simple elements** contain data values. A major difference between XML and HTML is that XML tag names are defined to describe the meaning of the data elements in the document, rather than to describe how the text is to be displayed. This makes it possible to process the data elements in the XML document automatically by computer programs. Also, the XML tag (element) names can be defined in another document, known as the schema document, to give a semantic meaning to the tag names that can be exchanged among multiple users. In HTML, all tag names are predefined and fixed; that is why they are not extendible.

It is possible to characterize three main types of XML documents:

- **Data-centric XML documents.** These documents have many small data items that follow a specific structure and hence may be extracted from a structured database. They are formatted as XML documents in order to exchange them over or display them on the Web. These usually follow a predefined schema that defines the tag names.
- **Document-centric XML documents**. These are documents with large amounts of text, such as news articles or books. There are few or no structured data elements in these documents.
- **Hybrid XML documents.** These documents may have parts that contain structured data and other parts that are predominantly textual or unstructured. They may or may not have a predefined schema.

```
<Projects>
    <Project>
        <Name>ProductX</Name>
        <Number>1</Number>
        <Location>Bellaire</Location>
        <Dept_no>5</Dept_no>
        <Worker>
            <Ssn>123456789</Ssn>
            <Last_name>Smith</Last_name>
            <Hours>32.5</Hours>
        </Worker>
        <Worker>
            <Ssn>453453453</Ssn>
            <First_name>Joyce</First_name>
            <Hours>20.0</Hours>
        </Worker>
    </Project>
    <Project>
        <Name>ProductY</Name>
        <Number>2</Number>
        <Location>Sugarland</Location>
        <Dept_no>5</Dept_no>
        <Worker>
            <Ssn>123456789</Ssn>
            <Hours>7.5</Hours>
        </Worker>
        <Worker>
            <Ssn>453453453</Ssn>
            <Hours>20.0</Hours>
        </Worker>
        <Worker>
            <Ssn>333445555</Ssn>
            <Hours>10.0</Hours>
        </Worker>
    </Project>
...
</Projects>
```

XML documents that do not follow a predefined schema of element names and corresponding tree structure are known as schemaless XML documents. It is important to note that data-centric XML documents can be considered either as semistructured data or as structured data

DOCUMENT TYPE DEFINITION (DTD)

The document type definition (DTD) is an optional part of an XML document. The main purpose of a DTD is much like that of a schema: to constrain and type the information present in the document.

```
<!DOCTYPE university [
    <!ELEMENT university ( (department|course|instructor|teaches)+)>
    <!ELEMENT department ( dept_name, building, budget)>
    <!ELEMENT course ( course_id, title, dept_name, credits)>
    <!ELEMENT instructor (IID, name, dept_name, salary)>
    <!ELEMENT teaches (IID, course_id)>
    <!ELEMENT dept_name( #PCDATA )>
    <!ELEMENT building( #PCDATA )>
    <!ELEMENT budget( #PCDATA )>
    <!ELEMENT course_id ( #PCDATA )>
    <!ELEMENT title ( #PCDATA )>
    <!ELEMENT credits( #PCDATA )>
    <!ELEMENT IID( #PCDATA )>
    <!ELEMENT name( #PCDATA )>
    <!ELEMENT salary( #PCDATA )>
] >
```

However, the DTD does not in fact constrain types in the sense of basic types like integer or string. Instead, it constrains only the appearance of sub elements and attributes within an element. The DTD is primarily a list of rules for what pattern of subelements may appear within an element.

Example of a DTD

Thus, in the DTD, a university element consists of one or more course, department, or instructor elements; the operator specifies “or” while the + operator specifies “one or more.” Although not shown here, the ∗ operator is used to specify “zero or more,” while the? operator is used to specify an optional element (that is, “zero or one”). The course element contains sub elements course id, title, dept name, and credits (in that order).

Similarly, department and instructor have the attributes of their relational schema defined as sub elements in the DTD. Finally, the elements course id, title, dept name, credits, building, budget, IID, name, and salary are all declared to be of type #PCDATA. The keyword #PCDATA indicates text data; it derives its name, historically, from “parsed character data.” Two other special type declarations are empty, which says that the element has no contents, and any, which says that there is no constraint on the sub elements of the element; that is, any elements, even those not mentioned in the DTD, can occur as sub elements of the element. The absence of a declaration foran element is equivalent to explicitly declaring the type as any.

XML SCHEMA

XML Schema defines a number of built-in types such as string, integer, decimal date, and boolean. In addition, it allows user-defined types; these may be simple types with added restrictions, or complex types constructed using constructors such as complex Type and sequence.

Note that any namespace prefix could be used in place of xs; thus we could replace all occurrences of “xs:” in the schema definition with “xsd:” without changing the meaning of the schema definition. All types defined by XML Schema must be prefixed by this namespace prefix. The first element is the root element university, whose type is specified to be University Type, which is declared later. The example then defines the types of elements department, course, instructor, and teaches. Note that each of these is specified by an element with tag xs:element, whose body contains the type definition.

The type of department is defined to be a complex type, which is further specified to consist of a sequence of elements dept name, building, and budget. Any type that has either attributes or nested sub elements must be specified to be a complex type. Alternatively, the type of an element can be specified to be a predefined type by the attribute type; observe how the XML Schema types xs: string and xs: decimal are used to constrain the types of data elements such as dept name and credits. Finally, the example defines the type University Type as containing zero or more occurrences of each of department, course, instructor, and teaches. Note the use of ref to specify theoccurrence

of an element defined earlier.

XML Schema can define the minimum and maximum number of occurrences of sub elements by using minOccurs and max Occurs. The default for both minimum and maximum occurrences is 1, so these have to be specified explicitly to allow zero or more department, course, instructor, and teaches elements

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="university" type="universityType" />
<xs:element name="department">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="dept_name" type="xs:string"/>
            <xs:element name="building" type="xs:string"/>
            <xs:element name="budget" type="xs:decimal"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="course">
            <xs:element name="course_id" type="xs:string"/>
            <xs:element name="title" type="xs:string"/>
            <xs:element name="dept_name" type="xs:string"/>
            <xs:element name="credits" type="xs:decimal"/>
</xs:element>
<xs:element name="instructor">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="IID" type="xs:string"/>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="dept_name" type="xs:string"/>
            <xs:element name="salary" type="xs:decimal"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

.

```
<xs:element name="teaches">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="IID" type="xs:string"/>
            <xs:element name="course_id" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:complexType name="UniversityType">
    <xs:sequence>
        <xs:element ref="department" minOccurs="0"
                maxOccurs="unbounded"/>
        <xs:element ref="course" minOccurs="0"
                maxOccurs="unbounded"/>
        <xs:element ref="instructor" minOccurs="0"
                maxOccurs="unbounded"/>
        <xs:element ref="teaches" minOccurs="0"
                maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
</xs:schema>
```

Attributes are specified using the xs:attribute tag. For example, we could have defined dept name as an attribute by adding:
<xs:attribute name = "dept name"/>
within the declaration of the department element. Adding the attribute use = "required" to the above attribute specification declares that the attribute must be specified, whereas the default value of use is optional. Attribute specifications would appear directly under the enclosing complex Type specification, even if elements are nested within a sequence specification.

In addition to defining types, a relational schema also allows the specification of constraints. XML Schema allows the specification of keys and key references, corresponding to the primary-key and foreign-key definition in SQL. In SQL, a primary-key constraint or unique constraint ensures that the attribute values do not recur within the relation. In the context of XML, we need to specify a scope within which values are unique and form a key. The selector is a path expression that defines the scope for the constraint, and field declarations specify the elements or attributes that form the key. To specify that dept name forms a key for department elements under the root university element, we add the following constraint specification to the schema definition:

```
<xs:key name = "deptKey">
    <xs:selector xpath = "/university/department"/>
    <xs:field xpath = "dept_name"/>
</xs:key>
```

Correspondingly a foreign-key constraint from course to department may be defined as follows:

```
<xs: name = "courseDeptFKey" refer="deptKey">
    <xs:selector xpath = "/university/course"/>
    <xs:field xpath = "dept_name"/>
</xs:keyref>
```

XML Schema offers several benefits over DTDs, and is widely used today. Among the benefits that we have seen in the examples above are these:

- It allows the text that appears in elements to be constrained to specific types, such as numeric types in specific formats or complex types such as sequences of elements of other types.
- It allows user-defined types to be created.
- It allows uniqueness and foreign-key constraints.
- It is integrated with namespaces to allow different parts of a document to conform to different schemas.

In addition to the features we have seen, XML Schema supports several other features that DTDs do not, such as these:

- It allows types to be restricted to create specialized types, for instance by specifying minimum and maximum values.
- It allows complex types to be extended by using a form of inheritance.

XQUERY

XPath allows us to write expressions that select items from a tree-structured XML document. XQuery permits the specification of more general queries on one or more XML documents. The typical form of a query in XQuery is known as a FLWR expression, which stands for the four main clauses of XQuery and has the following form:

FOR<variable bindings to individual nodes (elements)>
LET <variable bindings to collections of nodes (elements)>
WHERE <qualifier conditions>
RETURN<query result specification>

There can be zero or more instances of the FOR clause, as well as of the LET clause in a single XQuery. The WHERE clause is optional, but can appear at most once, and the RETURN clause must appear exactly once. Let us

illustrate these clauses with the following simple example of a XQuery.

```
LET $d := doc(www.company.com/info.xml)
FOR $x IN $d/company/project[projectNumber = 5]/projectWorker,
    $y IN $d/company/employee
WHERE $x/hours gt 20.0 AND $y.ssn = $x.ssn
RETURN <res> $y/employeeName/firstName, $y/employeeName/lastName,
           $x/hours </res>
```

1. Variables are prefixed with the $ sign. In the above example, $d, $x, and $y are variables.
2. The LET clause assigns a variable to a particular expression for the rest of the query. In this example, $d is assigned to the document file name. It is possible to have a query that refers to multiple documents by assigning multiple variables in this way.
3. The FOR clause assigns a variable to range over each of the individual items in a sequence. In our example, the sequences are specified by path expressions. The $x variable ranges over elements that satisfy the path expression $d/company/project[projectNumber = 5]/projectWorker. The $y variable ranges over elements that satisfy the path expression $d/company/employee. Hence, $x ranges over projectWorker elements, whereas $y ranges over employee elements.
4. The WHERE clause specifies additional conditions on the selection of items. In this example, the first condition selects only those projectWorker elements that satisfy the condition (hours gt 20.0). The second condition specifies a join condition that combines an employee with a projectWorker only if they have the same ssn value.
5. Finally, the RETURN clause specifies which elements or attributes should be retrieved from the items that satisfy the query conditions. In this example, it will return a sequence of elements each containing for employees who work more that 20 hours per week on project number 5.

XQuery has very powerful constructs to specify complex queries. In particular, it can specify universal and existential quantifiers in the conditions of a query, aggregate functions, ordering of query results, selection based on position in a sequence, and even conditional branching. Hence, in some ways, it qualifies as a full-fledged programming language.

INFORMATION RETRIEVAL

IR CONCEPTS

Information retrieval is the process of retrieving documents from a collection in response to a query (or a search request) by a user. Information retrieval is "the discipline that deals with the structure, analysis, organization, storage, searching, and retrieval of information" as defined by Gerald Salton, an IR pioneer.

Information in the context of IR does not require machine-understandable structures, such as in relational database systems. Examples of such information include written texts, abstracts, documents, books, Web pages, e-mails, instant messages, and collections from digital libraries. Therefore, all loosely represented (unstructured) or semi structured information is also part of the IR discipline.

IR systems go beyond database systems in that they do not limit the user to a specific query language, nor do they expect the user to know the structure (schema) or content of a particular database. IR systems use a user's information need expressed as a **free-form search request** (sometimes called a **keyword search query**, or just query) for interpretation by the system.

An IR system can be characterized at different levels: by *types of users*, *types of data*, *and the types of the information need*, along with the size and scale of the information repository it addresses. Different IR systems are designed to address specific problems that require a combination of different characteristics. These characteristics can be briefly described as follows:

Types of Users

The user may be an *expert user* (for example, a curator or a librarian), who is searching for specific information that is clear in his/her mind and forms relevant queries for the task, or a *layperson user* with a generic information need.

Types of Data

Search systems can be tailored to specific types of data. For example, the problem of retrieving information

about a specific topic may be handled more efficiently by customized search systems that are built to collect and retrieve only information related to that specific topic. The information repository could be hierarchically organized based on a concept or topic hierarchy. These topical domain-specific or vertical IR systems are not as large as or as diverse as the generic World Wide Web, which contains information on all kinds of topics.

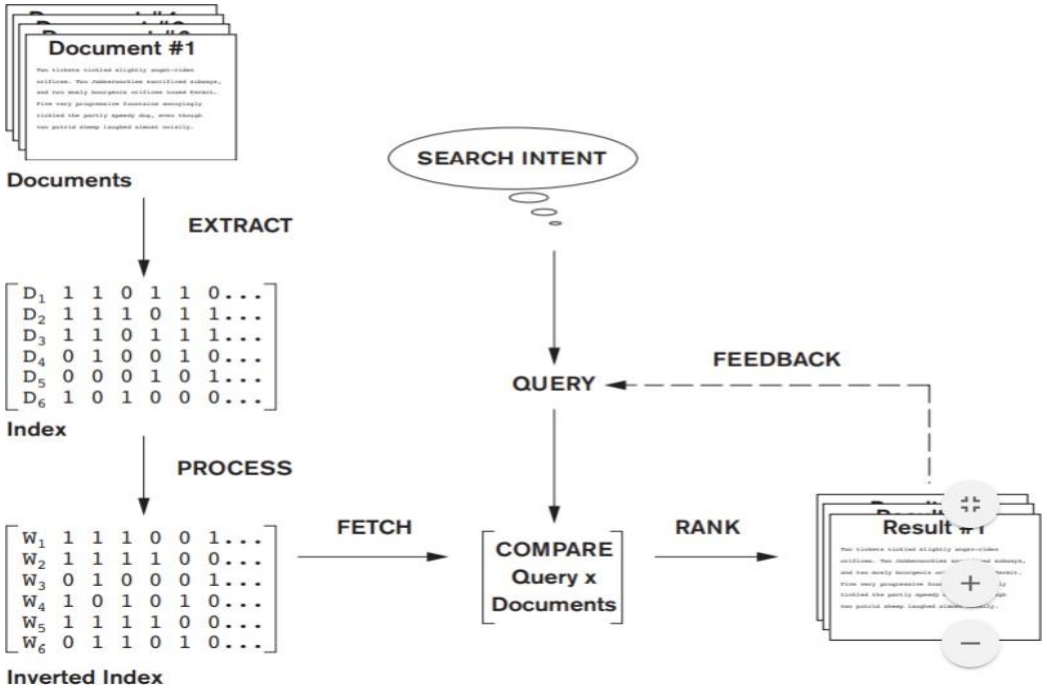
Types of Information Need

In the context of Web search, users' information needs may be defined as navigational, informational, or transactional.

Navigational search refers to finding a particular piece of information (such as the Georgia Tech University Website) that a user needs quickly. The purpose of **informational search** is to find current information about a topic (such as research activities in the college of computing at Georgia Tech—this is the classic IR system task). The goal of **transactional search** is to reach a site where further interaction happens (such as joining a social network, product shopping, online reservations, accessing databases, and so on).

RETRIEVAL MODELS

There are the three main statistical models—Boolean, vector space, and probabilistic—and the semantic model.



Simplified IR process Pipeline

Boolean Model

In this model, documents are represented as a set of terms. Queries are formulated as a combination of terms using the standard Boolean logic set-theoretic operators such as AND, OR and NOT. Retrieval and relevance are considered as binary concepts in this model, so the retrieved elements are an "exact match" retrieval of relevant documents.

Boolean retrieval models lack sophisticated ranking algorithms and are among the earliest and simplest information retrieval models. These models make it easy to associate metadata information and write queries that match the contents of the documents as well as other properties of documents, such as date of creation, author, and type of document.

Vector Space Model

The vector space model provides a framework in which term weighting, ranking of retrieved documents, and relevance feedback are possible. Documents are represented as features and weights of term features in an n-dimensional vector space of terms. Features are a subset of the terms in a set of documents that are deemed most

relevant to an IR search for this particular set of documents. The process of selecting these important terms (features) and their properties as a sparse (limited) list out of the very large number of available terms (the vocabulary can contain hundreds of thousands of terms) is independent of the model specification. The query is also specified as a terms vector (vector of features), and this is compared to the document vectors for similarity/relevance assessment.

In the vector model, the document term weight wij (for term i in document j) is represented based on some variation of the TF (term frequency) or TF-IDF (term frequency-inverse document frequency) scheme (as we will describe below). TF-IDF is a statistical weight measure that is used to evaluate the importance of a document word in a collection of documents. The following formula is typically used:

$$\text{cosine}(d_j, q) = \frac{\langle d_j \times q \rangle}{\|d_j\| \times \|q\|} = \frac{\sum_{i=1}^{|V|} w_{ij} \times w_{iq}}{\sqrt{\sum_{i=1}^{|V|} w_{ij}^2} \times \sqrt{\sum_{i=1}^{|V|} w_{iq}^2}}$$

In the formula given above, we use the following symbols:

- dj is the document vector.
- q is the query vector.
- wij is the weight of term i in document j.
- wiq is the weight of term i in query vector q.
- |V| is the number of dimensions in the vector that is the total number of important keywords (or features).

Probabilistic Model

In the probabilistic framework, the IR system has to decide whether the documents belong to the **relevant set** or the **nonrelevant** set for a query. To make this decision, it is assumed that a predefined relevant set and nonrelevant set exist for the query, and the task is to calculate the probability that the document belongs to the relevant set and compare that with the probability that the document belongs to the nonrelevant set.

Given the document representation D of a document, estimating the relevance R and nonrelevance NR of that document involves computation of conditional probability P(R|D) and P(NR|D). These conditional probabilities can be calculated using Bayes' Rule

$$P(R|D) = P(D|R) \times P(R)/P(D)$$
$$P(NR|D) = P(D|NR) \times P(NR)/P(D)$$

A document D is classified as relevant if P(R|D) > P(NR|D). Discarding the constant P(D), this is equivalent to saying that a document is relevant if:

$$P(D|R) \times P(R) > P(D|NR) \times P(NR)$$

The likelihood ratio P(D|R)/P(D|NR) is used as a score to determine the likelihood of thedocument with representation D belonging to the relevant set.

Semantic Model

Semantic approaches include different levels of analysis, such as morphological, syntactic, and semantic analysis, to retrieve documents more effectively. In morphological analysis, roots and affixes are analyzed to determine the parts of speech (nouns, verbs, adjectives, and so on) of the words.

The development of a sophisticated semantic system requires complex knowledge bases of semantic information as well as retrieval heuristics. These systems often require techniques from artificial intelligence and expert systems. Knowledge bases like Cyc15 and WordNet16 have been developed for use in knowledge-based IR systems based on semantic models.

QUERIES IN IR SYSTEMS

The queries formulated by users are compared to the set of index keywords. Most IR systems also allow the use of Boolean and other operators to build a complex query. The query language with these operators enriches the expressiveness of a user's information need.

Keyword Queries

Keyword-based queries are the simplest and most commonly used forms of IR queries: the user just enters keyword combinations to retrieve documents. The query keyword terms are implicitly connected by a logical AND operator. A query such as 'database concepts' retrieves documents that contain both the words 'database' and 'concepts' at the top of the retrieved results. In addition, most systems also retrieve documents that contain only 'database' or only 'concepts' in their text. Some systems remove most commonly occurring words (such as a, the, of, and so on, called stop words) as a preprocessing step before sending the filtered query keywords to the IR engine.

Boolean Queries

Some IR systems allow using the AND, OR, NOT, ( ), + , and – Boolean operators in combinations of keyword formulations. AND requires that both terms be found. OR lets either term be found. NOT means any record containing the second term will be excluded. '( )' means the Boolean operators can be nested using parentheses. '+' is equivalent to AND, requiring the term; the '+' should be placed directly in front of the search term.'–' is equivalent to AND NOT and means to exclude the term; the '–' should be placed directly in front of the search term not wanted. Complex Boolean queries can be built out of these operators and their combinations, and they are evaluated according to the classical rules of Boolean algebra.

Phrase Queries

When documents are represented using an inverted keyword index for searching, the relative order of the terms in the document is lost. In order to perform exact phrase retrieval, these phrases should be encoded in the inverted index or implemented differently (with relative positions of word occurrences in documents). A phrase query consists of a sequence of words that makes up a phrase. The phrase is generally enclosed within double quotes. Each retrieved document must contain at least one instance of the exact phrase. Phrase searching is a more restricted and specific version of proximity searching.

Proximity Queries

Proximity search refers to a search that accounts for how close within a record multiple terms should be to each other. The most commonly used proximity search option is a phrase search that requires terms to be in the exact order.

Wildcard Queries

Wildcard searching is generally meant to support regular expressions and pattern matching-based searching in text. In IR systems, certain kinds of wildcard search support may be implemented—usually words with any trailing characters (for example, 'data*' would retrieve data, database, datapoint, dataset, and so on).

Natural Language Queries

There are a few natural language search engines that aim to understand the structure and meaning of queries written in natural language text, generally as a question or narrative. This is an active area of research that employs techniques like shallow semantic parsing of text, or query reformulations based on natural language understanding. The system tries to formulate answers for such queries from retrieved results. Some search systems are starting to provide natural language interfaces to provide answers to specific types of questions, such as definition and factoid questions, which ask for definitions of technical terms or common facts that can be retrieved from specialized databases.