UNIT-1

INTRODUCTION

Data

- Data is Raw, unorganized facts that need to be processed.
- Data itself is meaningless and worthless.
- A meaningful information is only possible after processing such a raw data.

Information

- When data is processed, organized, structured or presented in a given context so as to make useful, is called information.
- It is output of data processing operation.
- For example, marks obtained by students and their roll numbers form data where their mark sheet is the information.

Database

- A Database is a collection of inter-related data organized in a way that data can be easily accessed, managed and updated.
- For example: The college Database organizes the data about the admin, staff, students and faculty etc.
- Using the database, you can easily retrieve, insert, and delete the information.

Database Management system

- Database management system (DBMS) is a collection of programs that allows to create and maintain databases.
- DBMS provides an interface to the user so that the operations on database can be performed using the interface.

Some examples of DBMS are:

- MySQL
- Microsoft SQL-Server
- Oracle
- IBM DB2
- PostgreSQL etc.

Operations that allowed by DBMS

- **Data Definition:** Creating, modifying, and deleting the structure of the database, including tables, fields, and relationships.
- **Data Modification:** It is used for the insertion, modification, and deletion of the actual data in the database.
- **Data Retrieval:** DBMS allows users to fetch data from the database.
- **Data Security**: Ensuring the security of the database and its contents, such as controlling access to the data, protecting against unauthorized access, and ensuring the integrity and consistency of the data.
- **Data Administration:** Managing the overall performance and efficiency of the database system, including monitoring, and tuning the system, optimizing queries, and managing resources such as disk space and memory.

Objective of DBMS

- Providing mass storage of relevant data.
- Make easy access to data for authorized users.
- Eliminating data redundancy.
- Providing data integrity.
- Providing security with user access privilege.
- Allow multiple users to be active at one time.
- Serving different types of users.
- Provide prompt response to user requests for data.
- Combining interrelated data to generate reports.
- Protect the data from physical harm and un-authorized systems.
- Allowing the growth of the database system.
- Make the latest modifications visible to all the database available immediately.

Application of DBMS

- **Telecom:** For keeping records of calls made, generating monthly bills, maintaining balances on prepaid calling cards, and storing information about the communication networks.
- **Manufacturing**: For management of product manufactured by companies such as supply chain and tracking production , inventories of items in stores, and orders for items.
- **Banking System**: For storing customer info, tracking day to day credit and debit transactions, generating bank statements etc.
- **Sales:** To store customer information, production information and invoice as well to track, manage and generate historical data to analyze the sales data.
- **Airlines:** For reservations, and schedule Information.
- **Education sector:** Used in schools and colleges to store and retrieve the data regarding student details, staff details, course details, exam details, attendance details, fees details etc.
- **Human Resource Management:** For information of employees such as salaries, performance reviews, job history etc.

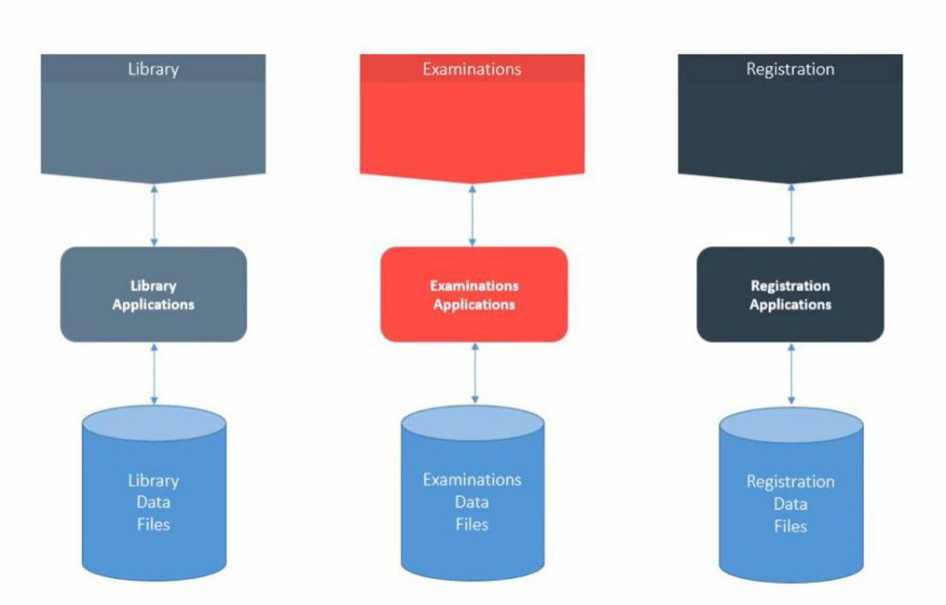
File processing system

In earlier days, data was stored manually, using pen and paper.

But after the computer was discovered, the same task could be done by using files. There are various formats in which data can be stored. e.g. Text files can be stored in .txt format.

Files are used to store various documents. All files are grouped based on their categories. The file names are very related to each other and arranged properly to easily access the files. In file processing system, if one needs to insert, delete, modify, store or update data, one must know the entire hierarchy of the files.

Let us consider the following example

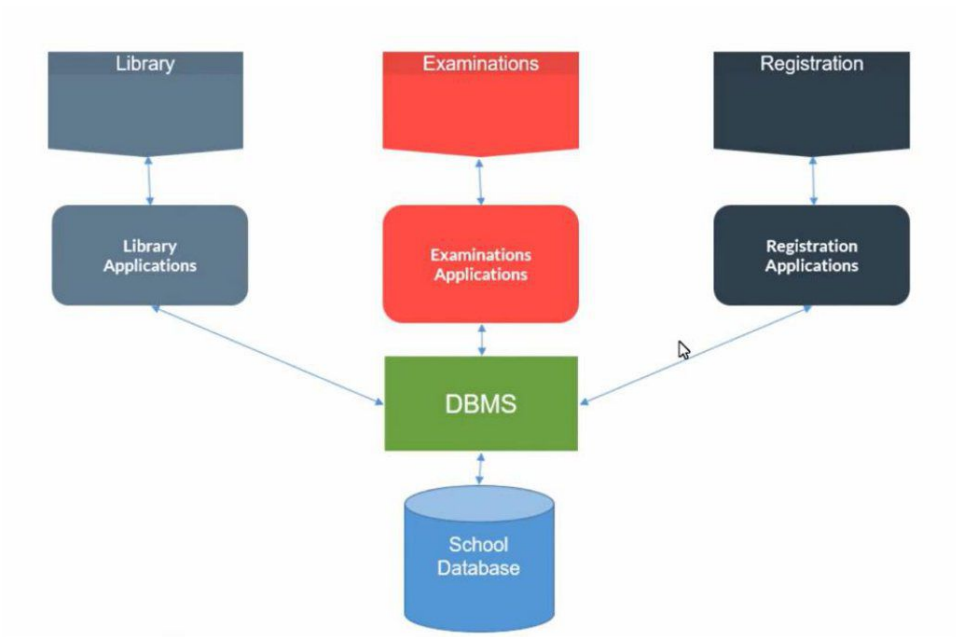


Examination File
1 Nadeem Ahmad city B+ totalMarks, 3 Sara khan city B+ totalMarks, 2 Muhammad Saim city B+ totalMarks,

Registration File
1 Nadeem Ahmad city B+ Admission date, 3 Sara khan city B+ Admission date, 2 Muhammad Saim city B+ Admission date,

- In the above figure we can see that each application contains data files.
- The Examination file will contain information regarding the student and their marks. Similarly, we have a registration file containing information regarding the student and the admission date. Student information is duplicated here, which leads to data redundancy.
- Another major drawback of traditional file processing is the non-sharing of data. It means, if different systems of organization are using some common data rather than storing it once and sharing it, each system stores separate files. This creates problems of redundancy and storage.

DBMS System



Database is shared collection of logically interrelated data designed to meet multiple users of an organization.

The typical database environment is as shown in the above figure. The figure shows different subsystems over application in an educational institution like library system, examination system and registration system. There are separate different application programs for every application over subsystem. However, data for all applications is stored in the same place in the database and all application program relevant data and users are managed by Database Management System (DBMS).

This introduces the major benefit of data sharing. That is data that is common among different applications need not to be stored repeatedly as was the case in the file processing environment.

Now from the above discussion we can analyze and conclude problem associated with file processing system and advantages of DBMS over file processing system.

Problems associated with file processing system

1) Data Redundancy

- Data redundancy refers to the duplication of data.
- Lets say we are managing the data of new admitted student. The same student details in such case will be stored in account, library and exam section etc. which will take more storage than needed.

2) Data inconsistency

- Data redundancy leads to data inconsistency.
- lets same example that we have taken above, where we store the information of student in multiple places.
- Now lets say student requests to change his city. If the city is changed at one place and not on all the records then this can lead to data inconsistency.

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) **Security problem**

- Not every user of the database system should be able to access all the data. for example a student in a college should not be able to see the payroll details of the teachers, such kind of security constraints are difficult to apply in file processing systems.

5) **Integrity problem**

- The data values stored in the data must satisfy certain types of consistency constraints. For example, CGPA of students cannot be greater than 4 in Pokhara university grading system.

6) **Lack of atomicity**

- Operations performed in the database must be atomic i.e. either the operation takes place as a whole or does not take place at all.

7) **Problem in concurrent Access**

- When a number of users operates on a common data in database at the same time then anomalies arise, due to lack of concurrency control.

8**) Slow access time**

- Direct access of files is very difficult and one needs to know the entire hierarchy of folders to get to a specific file. This involves a lot of time.

Advantages of DBMS over File Processing system/Advantages of DBMS

1) **Remove data redundancy:** Redundancy removed by data normalization. Removal of duplication of data saves storage and improves access time.

2) **Data security:** It is easier to apply access constraints in database systems so that only an authorized user can access the data. Each user has a different set of access thus data is secured from issues such as identity theft, data leaks and misuse of data.

3) **Data Integrity:** DBMS ensures data integrity by enforcing data validation rules, such as data type, range, and format, which helps to prevent data entry errors and ensure data accuracy.

4) **Data sharing:** In DBMS, the authorized users of an organization can share the data among multiple users.

5) **Scalability and flexibility:** DBMS systems are scalable, the database size can be increased and decreased based on the amount of storage required. It also allows addition of additional tables as well as removal of existing tables without disturbing the consistency of data.

6) **Data Backup and Recovery:** DBMS provides backup and recovery mechanisms to protect against data loss or corruption. This helps to ensure the availability and reliability of the data, even in the event of hardware or software failures.

7) **Fast response:** Database systems manages data in such a way so that the data is easily accessible with fast response times.

Disadvantages of DBMS

- It occupies a large space of disks and large memory to run them efficiently.
- Except MYSQL(which is open source), licensed DBMS are costly.
- DBMS can be complex and difficult to learn, especially for non-technical users.
- Can be vulnerable to security breaches if not properly configured and managed. This can lead to data loss or theft.

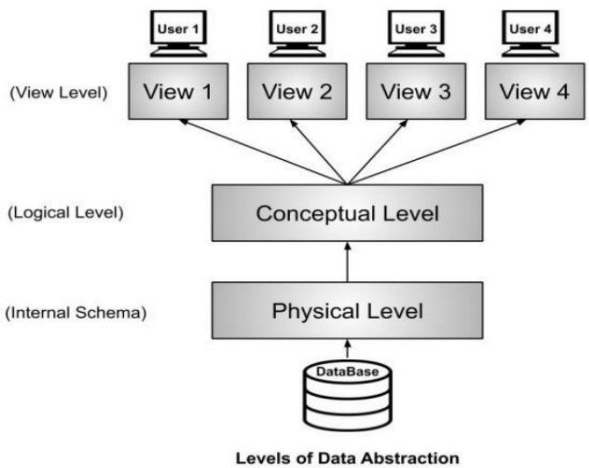
DBMS vs File processing system

DBMS	File processing system
A database management system is collection of programs that enables to create and maintain a database.	A file processing system is a collection of programs that store and manage files in computer hard-disk.
Data redundancy problem is less.	Data redundancy problem exist.
DBMS gives an abstract view of data that hides the details.	The file system provides the details of data representation and storage of data.
DBMS systems offers high security.	File systems provide less security in comparison to DBMS.
There is more data consistency because of process of normalization.	There is less data consistency in file processing system.

The problem of data isolation is not found in it.	Data are scattered in various files in different formats. Writing new program to retrieve appropriate data is difficult.
DBMS is more complex.	File processing system is less complex.
DBMS system provides backup and recovery of data even if it is lost.	It does not offer data recovery processes.
The data independence feature of DBMS allows separating data definition from program.	The data used for any file is dependent on program because file structure is defined in program code.

Data abstraction

- Data abstraction is a technique to hide the implementation details of a database that how a database is structured and how the data is stored physically.
- It allows database systems to provide abstract views to database users.
- It is a mechanism to hide the complexity of databases.
- Data abstraction simplifies users' interactions with the system.



Levels of Data Abstraction

There are three levels of abstraction.

1. Physical level
2. Logical level
3. View level

1. Physical Level

- It is the lowest level of abstraction, and it describes how a record are actually stored.
- At the physical level, complex level data structures are described in detail. (e.g. index, B-tree, hashing)

2. Logical Level

- This is the next highest level of abstraction.
- It describes what data are stored in databases and what relationship exists among them.

3. **View level**

- It is the highest level of abstraction. It describes only part of the entire database. It simplifies interaction with the system.
- It allows database system to provide many views for the same database. That is it allows each user/application to get a different perspective of the database.

Let's consider an example of a database system for managing employee information in a company.

Physical level: At the physical level, the database system might store the employee information in a specific file format optimized for fast access and efficient storage. It might also use disk blocks and indexing to manage the storage and retrieval of data.

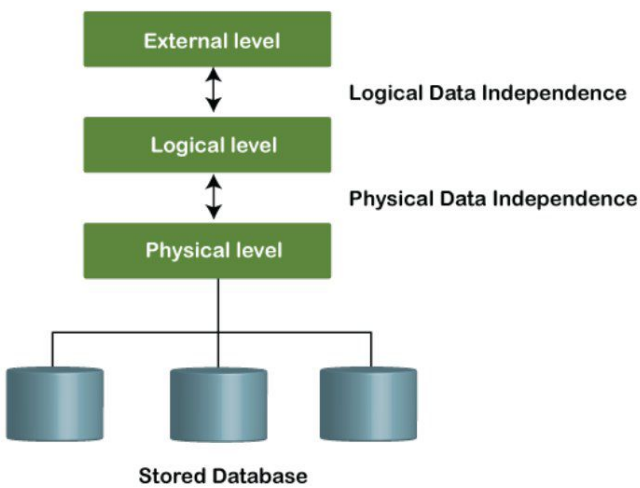
Logical level: At the logical level, the database system might define tables for storing employee information, with columns for employee ID, name, address, job title, salary, and other relevant data elements. Relationships might be defined between tables, such as a relationship between the employee table and a department table to track which department an employee belongs to.

View level: At the view level, different views of the data might be created for different users or applications. For example, HR department might have a view that shows employee names, job titles, and salaries for managing payroll, while the management team might have a view that shows employee performance metrics and career development plans for strategic planning.

Data independence

- Data independence can be explained using the three-schema architecture.
- Data independence is an ability to modify a schema definition in one level without affecting the schema definition in higher level.
- There are two types of data independence.

1. Logical data independence
2. Physical data independence



1. **Logical Data Independence**

- Logical data independence is the capacity to change the conceptual schema without having to change external schemas or application programs.
- Logical data independence is used to separate the external level from the conceptual view.
- If we do any changes in the conceptual view of the data, then the user view of the data would not be affected.
- E.g. Add/Modify/Delete a new attribute, entity or relationship is possible without a rewrite of existing application programs.

2. **Physical data independence**

- Physical data independence is the capacity to change the internal schema without having to change the conceptual schemas.
- With Physical independence, we can easily change the physical storage structures or devices with an effect on the conceptual schema.
- Due to Physical independence, any of the below changes will not affect the conceptual layer.
  - ✓ Using a new storage device
  - ✓ Switching to different data structures
  - ✓ Changing the location of database

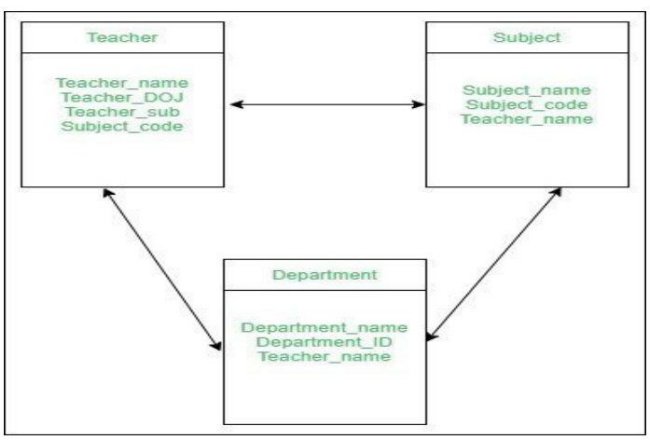
Assignment:

- ❖ *Differentiate between physical data independence and logical data independence.*

Schema and instances

Schema

- Schema is overall design of database.
- In other words, the basic structure of how the data will be stored in the database is called schema.
- In the following diagram, we have a schema that shows the relationship between three tables: Teacher, Subject and Department.
- The diagram only shows the design of the database, it doesn't show the data present in those tables.



There are three types of schema.

1. **Physical schema**: The design of a database at physical level is called physical schema, how the data stored in blocks of storage is described at this level and database administrator works at this level.
2. **Conceptual/Logical schema:** Design of database at logical level is called logical schema, programmers and database designers work at this level, at this level data can be described as certain types of data records gets stored in data structures, however the internal details such as implementation of data structure is hidden at this level (available at physical level).
3. **View schema**: Design of database at view level is called view schema. This generally describes end user interaction with database systems.

Instances

- The data stored in database at a particular moment of time is called instance of database.
- Database schema defines the attributes in tables that belong to a particular database.
- The value of these attributes at a moment of time is called the instance of that database.
- The instances may be changed by certain operations such as addition, deletion etc.
- It may be noted that any search query will not make any kind of change in the instances.

For better understanding of schema and instances Let us consider simple example

An employee table in database exists with the following attributes:

Eid	Name	Position	Salary

- This is the schema of the employee table.
- Schema defines the attributes of tables in the database.
- It shows the design of the database, it doesn't show the data present in those tables.

Instances

- we have seen the schema of table "employee".
- Let's look at the table with the data now.

Eid	Name	Position	Salary
1	Hari	Manager	55625
2	Ramesh	Helper	12550
3	Gita	CEO	95450
4	Sita	HOD	68325
5	Gopal	Secretary	25450

- At this moment the table contains 5 rows (records). This is the current instance of the table "employee" because this is the data that is stored in this table at this moment of time.
- Instances can be changed if we perform certain operations such as addition, deletion etc.

Concept of DDL, DML, DCL and TCL

DDL (Data Definition Language)

- It consists of SQL commands that can be used to define the database schema.
- It is used to create and modify the structure of database objects in database but not data.
- DDL is used by database designers and Database Administrators (DBA) but not user.

CREATE- is used to create the database or its objects (like table, function, views, store procedure etc).

Example:

1. CREATE DATABASE db_eemc; *//creates database named db_eemc*
2. CREATE TABLE employee_info(eid int, name varchar(50),position varchar(50),department varchar(50);

DROP-is used to delete objects from the database.

Example:

1. DROP DATABASE db_eemc; *//deletes database named db_eemc*
2. DROP TABLE employee_info; *//deletes table named employee_info from database*

ALTER- is used to Alter the structure of the database.

Eg.Add/Rename/Delete/Change columns in tables of database.

1. ALTER TABLE employee_info
   ADD salary decimal (10,2);
   //column named salary will be added in table named employee_info
2. ALTER TABLE employee_info
   DROP COLUMN salary;
   //column named salary will be deleted from table named employee_info

TRUNCATE–is used to remove all records from a table, including all spaces allocated for the records are removed.

Example:

TRUNCATE TABLE employee_info; *//All records will be deleted but not table*

Data Manipulation Language (DML)

It deals with the manipulation of data present in database.

INSERT – is used to insert data into a table.

Example:

INSERT INTO employee_info

VALUES(1,'Hari','manager', 'sales');

UPDATE – is used to update existing data within a table.

Example:

UPDATE employee_info

SET department='marketing'

WHERE eid=1;

//updates the department to marketing in employee_info table whose eid is 1

DELETE – is used to delete records from a database table.

Example:

DELETE FROM employee_info

WHERE eid=1;

// delete a row from table named employee_info which has eid=1

SELECT - is used to select (retrieve) data from a database table.

Example:

SELECT name,position

FROM employee_info

WHERE deparment='marketing';

//selects the name and position of employees whose department is marketing from employee_info table

Data Control Language (DCL)

- ✓ DCL includes commands such as GRANT and REVOKE which mainly deals with the rights, permissions, and other controls of the database system.
- ✓ It is used to give and withdraw specific privileges (as defined by query) to the user in a multi-user database.
- ✓ By setting up the permission, user can be prevented from unauthorized access to the database.

DCL commands are:

GRANT

- ✓ This is a SQL command which is used to provide privileges/permissions to modify and retrieve database objects like tables, views etc.
- ✓ It can be used to grant SELECT, INSERT, UPDATE, DELETE etc. privileges to a user.

Syntax:

GRANT <privilege list> on <relation or view> to <user>;

Example:

GRANT INSERT, SELECT on employee_info to ram;
GRANT SELECT, UPDATE,DELETE on employee_info to sita;

REVOKE

This command is used to withdraw user's access privileges given by using the GRANT command.

Syntax:

REVOKE <privilege list> on <relation or view> from <user>;

Example:

REVOKE INSERT on employee_info from ram;
REVOKE DELETE on employee_info from sita;

Transaction control Language (TCL)

- ✓ Transaction Control Language (TCL) is a set of special commands that deal with the transactions within the database.
- ✓ The changes made by DML commands are either committed or rolled back by TCL commands.
- ✓ There is another TCL command that can place a save point in the transactions which makes it possible to rollback all the transaction till the last save point.

COMMIT

Commit command make the changes made to the database permanent.

Syntax:

```
COMMIT;
```

Here's the syntax demonstrating the use of the COMMIT command with a transaction in MySQL:

```
START TRANSACTION;
{a set of SQL statements};
COMMIT;
```

The parameters used in the syntax are:

- ✓ START TRANSACTION: It is used for marking the beginning of changes or operations in a transaction.
- ✓ {a set of SQL statements}: It is used for mentioning the task that is supposed to be completed.
- ✓ COMMIT: It is used to save transactional changes made by SQL statement

Example:

```
START TRANSACTION;
DELETE FROM student_info
WHERE sid = 11;
COMMIT ;
```

ROLLBACK

- ✓ Rollback command is used to undo the changes that have been made to the database temporarily.
- ✓ The important point to note here is that the changes saved using COMMIT command cannot be undone using ROLLBACK command.

Example:

```
UPDATE student_info SET location='Dharan'
WHERE name='ram';
ROLLBACK;
```

SAVEPOINT

It's used to roll back a transaction to a specific point rather than the complete transaction.

Syntax:

SAVEPOINT SavepointName;

- ✓ Among all transactions, this command is exclusively used to create SAVEPOINT.
- ✓ ROLLBACK is a command that is used to undo a set of transactions.

The syntax for rollback to savepoint command:

ROLLBACK TO SavepointName;

Example:

```
UPDATE student_info
SET program = 'BBA'
WHERE sid = 5;
SAVEPOINT A;

UPDATE student_info
SET name = 'ram'
WHERE location = 'pokhara;
SAVEPOINT B;
```

Now if we want to roll back the certain DML commands, we can do so by using Rollback like this:
This will rollback the transaction till savepoint A;
Rollback to A;

Database Administrator (DBA)

- Database administrator is a person who has central control of both the data and program accessing that data.
- Coordinates all the activities of the database system; the database administrator has a good understanding of the enterprise's information resources and needs.
- The responsibility of a database administrator varies and depends on the job description, corporate and Information Technology (IT) policies and the technical features and capabilities of the DBMS being administrated.

The functions of DBA

1. **Schema definition:** The DBA creates the original database schema by executing a set of data definition statements in the DDL.

2. **Storage structure and access method modification**

The DBA is responsible for selecting the storage device, defining storage structure and access methods of database.

3. **Schema and physical-organization modification**: The DBA carries out changes to the schema and physical organization of database to reflect the changing needs of the organization, or to alter the physical organization to improve performance.

4. **Granting the authorization for data access:** By granting different types of authorization, the database administrator can regulate which parts of the database system various users can access. The authorization information is kept in special system structure that the database system consults whenever someone attempts to access the data in the system.

5. **Routine maintenance**

Examples of the database administrator's routine maintenance activities are:

- Periodically backing up the database, either onto tapes or remote servers, to percent loss of data in case of disasters such as flooding.
- Ensuring that enough free disk space is available for normal operations and upgrading disk space as required.
- Monitoring jobs running on the database and ensuring that performance is not degraded by very expensive tasks submitted by some users.

Database system architecture

Database Management System (DBMS) system architecture refers to the overall structure and organization of components that make up a DBMS. The architecture of a DBMS plays a crucial role in managing and accessing data efficiently.

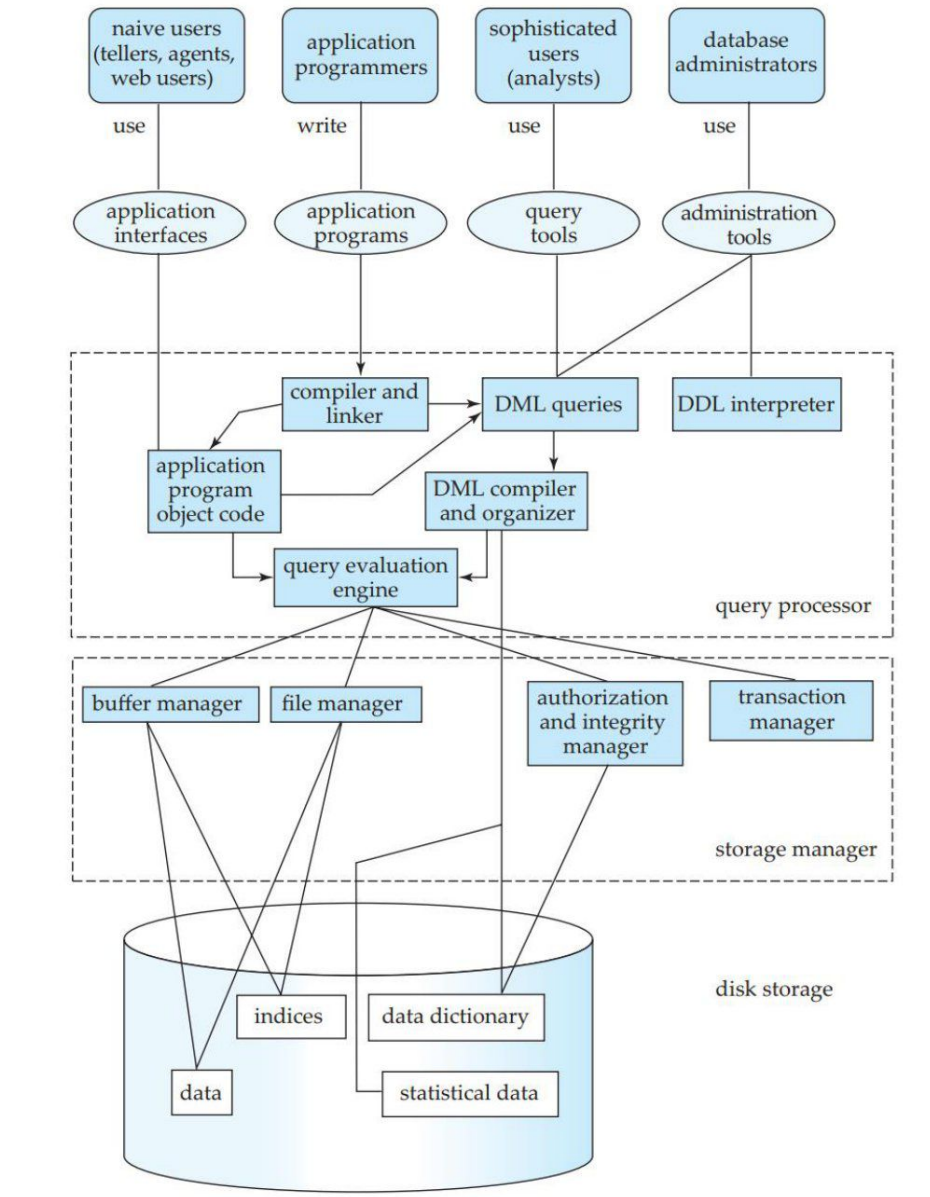


Figure: Database system Architecture

From the above figure we can see that four different categories of users are interacting with the database.

- ✓ **Naïve users**: They are not interacting with database directly. They use application programs to interact with the database. (Eg. Tellers, Agent, Web users)
- ✓ **Application programmers:** They write application programs to interact with the database.
- ✓ **Sophisticated users:** They directly access with database using database query language. (Eg. Analyst)
- ✓ **Database Administrators**: They issue certain system administration related commands and functions such as schema design, schema alternations, granting and revoking privileges.

Query processor

- ✓ As we know application programmers write application programs then **compiler** make the **application program object code**, **linker** combines with necessary components and query evaluation engine will evaluate and execute.
- ✓ **DML compiler** translates the **DML queries** in the query language into an evaluation plan and query evaluation plan consists of low-level instructions (relational algebra) that query evaluation engine understood.
- ✓ Among various query evaluation plans generated by the **query evaluation engine** pick the best plan to execute the query which is of least cost that will execute.
- ✓ **DDL interpreter** interprets the DDL statements that are provided by the database administrator and records the definition to the data dictionary.

Storage Manager

- ✓ Here, Buffer means temporary data keeping region. So, **Buffer manger is** responsible for managing data is kept on the buffer fetching from disk storage and it manages who is using buffer at particular moment or particular time.
- ✓ **File manager** is responsible for allocating and deallocating space on storage device to store data. When data needs to be retrieved, it also is responsible for locating the appropriate file and retrieving the necessary records.
- ✓ **Authorization and integrity manager** handles who can access which data. It is used to test all the integrity constraints that we enforced to the database. Eg. Balance should not be less than zero.
- ✓ **The Transaction manager ensures** that the database remains consistent despite system failures or any hardware or software failure. (i.e. ACID properties of transaction is satisfied).It also ensures concurrent operations carried out on this database are actually not conflicting with each other.

Disk Storage

- ✓ **Data** is stored in an organized manner using some data structures.
- ✓ **Indices** are data structures that help us for faster retrieval of data.
- ✓ **Data dictionary** stores metadata. Which means it stores the information about the databases such as tables, columns, datatype, constraints etc.
- ✓ **Statical data** refers to information about the distribution and characteristics of the data within the database. Statistical data is valuable for query optimization.

Database Application Architecture (Two Tier and Three Tier)

Two tier architecture

- ✓ The 2-tier architecture is similar to a basic client-server model.
- ✓ The application at the client end directly communicates with the database on the server side.
- ✓ The application resides at the client machine, where it invokes database system functionality at the server machine through query language statements.
- ✓ Application program interface standards like ODBC and JDBC are used for interaction between the client and the server.
- ✓ The server side is responsible for providing query processing and transaction management functionalities.
- ✓ On the client side, the user interfaces and application programs are run.

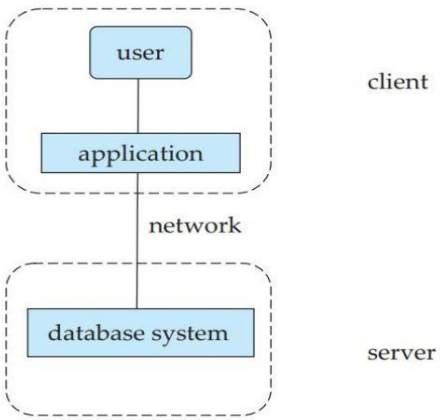


Figure: Two tier architecture

Advantages

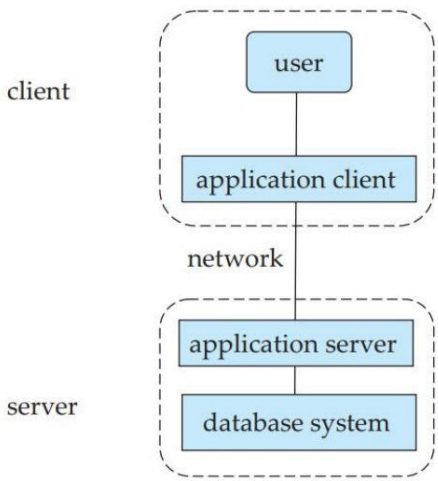
- ✓ **Simple:** Two-Tier Architecture is easily understandable as well as simple to design and implement.
- ✓ **Performance:** Direct communication between the client and the database server can result in good performance for certain types of applications.
- ✓ **Cost-Effectiveness:** Two-tier architectures can be cost-effective, especially for small to medium-sized applications.

Disadvantages

- ✓ **Scalability:** Scaling may require scaling both the client and the server, which can be less efficient than scaling just the server in a three-tier architecture.
- ✓ **Limited Security:** Security is less because the client often has direct access to the database server.

Three tier architecture

- ✓ In this architecture, the client can't directly communicate with the server.
- ✓ The three-Tier architecture contains another layer between the client and server.
- ✓ The application on the client-end interacts with an application server which further communicates with the database system. The business logic of the application, which says what actions to carry out under what conditions, is embedded in the application server, instead of being distributed across multiple clients.
- ✓ The end user has no idea about the existence of the database beyond the application server.
- ✓ The database also has no idea about any other user beyond the application server.
- ✓ The 3-Tier architecture is used in the case of large web applications and for applications that runs on World Wide Web.



Advantages:

- ✓ **Scalability:** Scalability is improved compared to two-tier architectures. The middle tier (application server) can be scaled independently of the client and database server.
- ✓ **Improved Security:** Security is enhanced as the presentation layer does not have direct access to the database. All database interactions are managed through the application server.

Disadvantages:

- ✓ **Complexity and Development Time:** Implementing a three-tier architecture can be more complex and time-consuming compared to simpler architectures.
- ✓ **Increased Resource Consumption:** With additional layers, there is a potential increase in resource consumption and overhead due to communication between the client, application server, and database server.

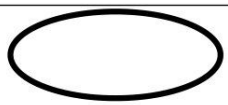
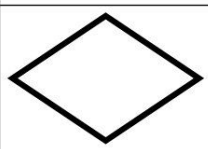
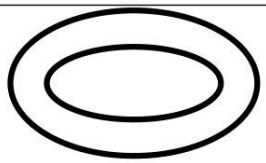
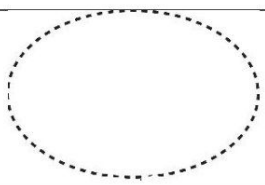
Unit 2

ER and Relational Model

Introduction to ER Model

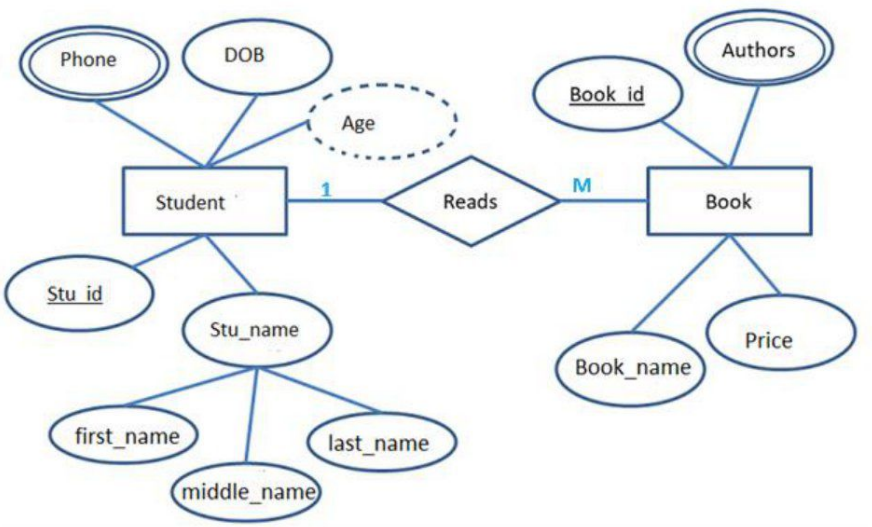
- In 1976, Peter Chen developed the Entity-Relationship (ER) model.
- It develops a conceptual design for the database. It also develops a very simple and easy to design view of data.
- An Entity-Relationship Model represents the structure of the database with the help of a diagram containing entities, relationships among them, and attributes of the entities.
- Diagrams created by this process are called entity-relationship diagrams, ER diagrams, or ERDs.
- Creation of an ER diagram, which is one of the first steps in designing a database, helps the designer(s) to understand and to specify the desired components of the database and the relationships among those components.

Symbols used in ER-diagram

Rectangle:-		to represent entity set
Ellipse:-		to represent attributes
Diamonds :-		to represent Relationship among entity set
Lines:-		to Link attributes to entity sets and entity sets to relationship.
Double ellipse:-		To represent multivalued attributes
Dashed Ellipse:-		To represent derived attribute

Double Rectangle:		To represent weak entity set

Example



- ✓ Here ER-diagram shows the relationship named reads in between two entities Student and Book.
- ✓ This is a one-to-many relationship.
- ✓ Student entity have attributes such as key attribute of stu_id, a composite attribute of stud_name with the components such as first_name,middle_name and last_name multivalued attribute of phone and derived attribute of age with its base attribute of DOB.
- ✓ Similarly, the book entity has a key attribute of book _id, a multivalued attribute of authors and other two attributes such as book_name and price.

Basic concept

This model is based on three basic concepts:

1. Entity and Entity sets
2. Relationships and Relationship sets
3. Attribute

1. Entity and Entity set

Entity

- An entity is a "thing" or "object" in the real world that is distinguished from all other objects. An Entity may be
  - ✓ **concrete** such as person, book, driver, product, employee etc.
  - ✓ it may be **abstract** such as course

- An entity has a set of properties, and the values for some set of properties may uniquely identify an entity.
- For example, employee may have employee_id property whose value uniquely identifies that employee. Thus, the value 677-89-9011 for employee_id would uniquely identify one particular employee in the organization.

Entity Set

- An entity set is a set of entities of the same type that share the same properties.
- **Example:**
  - ✓ The set of all employees of an organization can be defined as the entity set employees.
  - ✓ Similarly, the entity set projects represents the set of all projects undertaken by the organization.
- An entity set is represented by a set of attributes. Possible attributes of the employee's entity set are emp_id, emp_name, address, sex, date_of_birth. Possible attributes of the project entity set are project_id, Project_name.
- For each attribute there is a set of permitted values, called the domain of that attribute, which can be assigned to the attribute.

  One instance of the entity set **employee** and **project** are shown as:

Employee

Eid	Name	Sex	Address
E1	Ram	Male	Pokhara
E2	Shyam	Male	Butwal
E3	Sita	Female	Chitwan
E4	Krishna	Male	Kathmandu

Project

Project_id	Project _Name
P1	MIS
P2	Image processing
P3	Linux
P4	OS

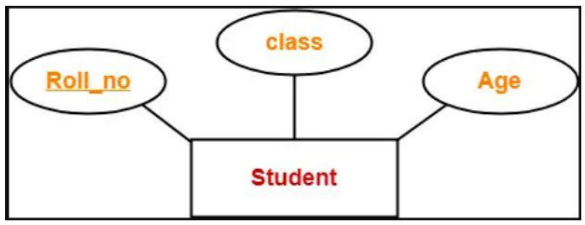
2. Attributes

- Attributes are descriptive properties possessed by each member of entity set.
- There exists a specific domain or set of values for each attribute from where the attribute can take its values.
- Eg. The student entity might have attributes like id, name, age program, semester etc.
- In ER diagram attributes are represented by an ellipse.

Attribute types

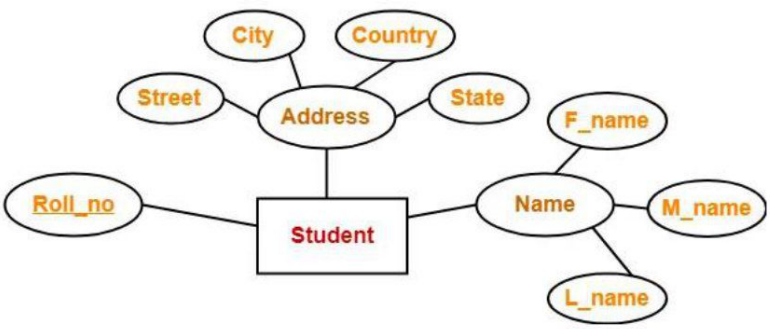
❖ Simple Attribute:

- ✓ Simple attributes are those attributes which cannot be divided further.
- ✓ Here, all the attributes are simple attributes as they cannot be divided further.



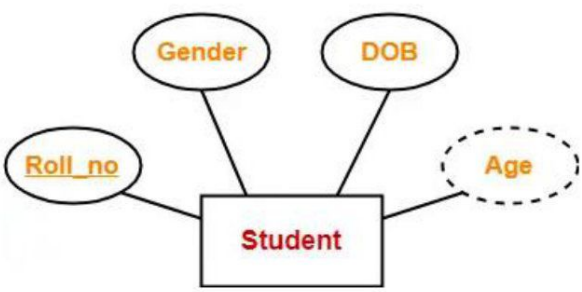
❖ Composite Attribute:

- ✓ Composite attributes are those attributes which are composed of many other simple attributes.
- ✓ Here, the attributes "Name" and "Address" are composite attributes as they are composed of many other simple attributes.
- ✓ In ER diagram, composite attribute is represented by an ellipse comprising of ellipse.



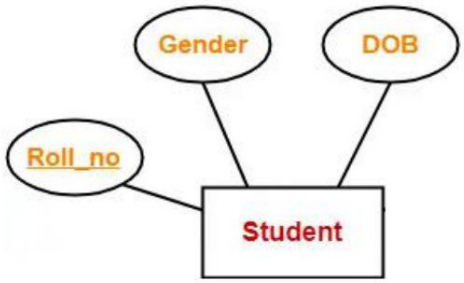
❖ Derived Attribute:

- ✓ Derived attributes are those attributes which can be derived from other attribute(s).
- ✓ Here, the attribute "Age" is a derived attribute as it can be derived from the attribute "DOB".
- ✓ In ER diagram, the derived attribute is represented by dashed ellipse.



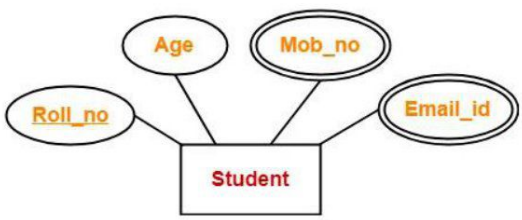
❖ Single valued Attribute:

- ✓ Single valued attributes are those attributes which can take only one value for a given entity from an entity set.
- ✓ Here, all the attributes are single valued attributes as they can take only one specific value for each entity.



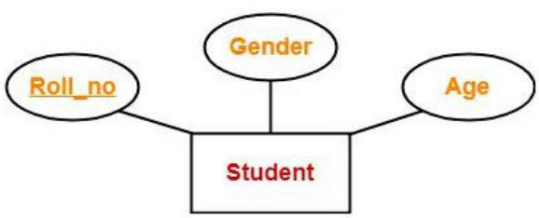
❖ Multivalued attribute

- ✓ Multi valued attributes are those attributes which can take more than one value for a given entity from an entity set.
- ✓ Here, the attributes "Mob_no" and "Email_id" are multi valued attributes as they can take more than one values for a given entity.
- ✓ In ER diagram, multivalued attribute is represented by double ellipse.



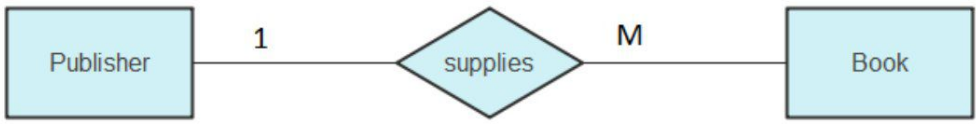
❖ Key attributes

- ✓ Key attributes are those attributes which can identify an entity uniquely in an entity set.
- ✓ Here, the attribute "Roll_no" is a key attribute as it can identify any student uniquely.
- ✓ In ER diagram, key attribute is represented by an ellipse with underlying lines.



Relationship and relationship sets

- A relationship is an association among several entities.
- It is represented using a diamond shape in the entity-relationship diagram. Example: Publisher supplies a book.



Here In the above example, the publisher and the book are entities whereas the word supplies is a relationship between those entities. Moreover, the attributes of the publisher entity can be that is a pub_id,Pub_name,address and the attributes of book entity can be book_id,book_name.

- A relationship set is a set of relationships of same type. It is a mathematical relation of n>=2 (possibly non distinct) entity sets.
- If E1, E2, E3,...., En are entity sets, then a relationship set R is a subset of {(e1, e2, e3, ..., en) | e1 ∈ E1, e2 ∈ E2, ..., en ∈ En} Where (e1, e2, e3, ..., en) is a relationship.
- The association between entity sets is referred to as participation; that is the entity sets E1,E2,....En participate in a relationship set R.
- A relationship instance in an E-R schema represents an association between entities in the real world that is being modeled.

Consider the two-entity set publisher and book. The relationship set supplies to denote the association between publisher and book that publisher supplies.

Publisher entity set

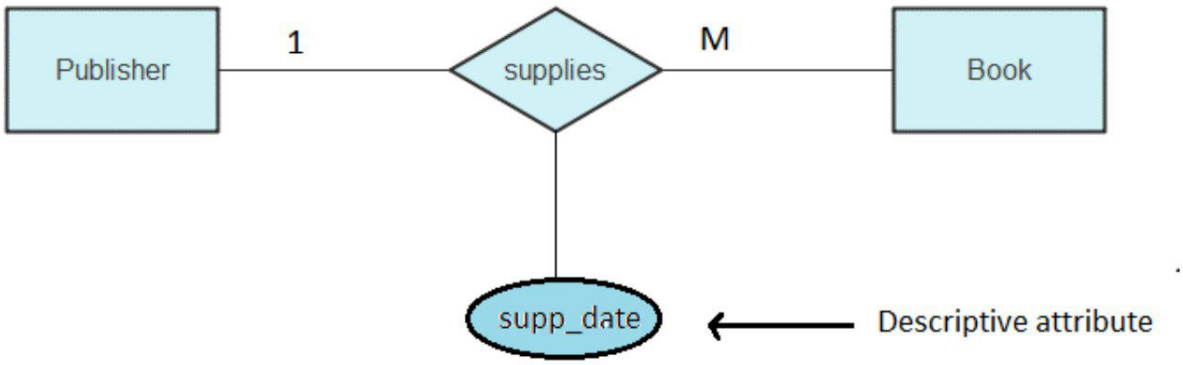
Pub_id	Pub_name	Address
P1	Asmita book suppliers	Newroad
P2	Goodwill	Kalanki
P3	Pariabi prakasan	Biratnagar

Book entity set

Book_ id	Book name
B1	Dbms
B2	Operating system
B3	C++
B4	Math

As an illustration, an individual Publisher entity Asmita book suppliers who has pub_id and the Book entity who has Book_id B1 and B2, participate in a relationship instance of supplies. This relationship instance represents that Asmita book suppliers supplies two books i.e DBMS and Operating System.

A relationship may also have attributes called descriptive attributes. Consider a relationship set supplies with entity sets Publisher and Book. We could associate the attribute date with that relationship to specify date when supplier supplies book.



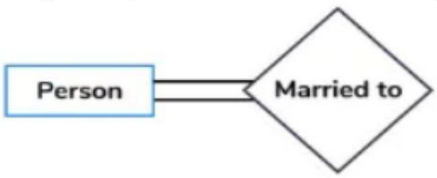
Degree of Relationship Set

- It refers to the number of entities sets that participate in a relationship.
- Unary relationship is of degree 1.
- Binary relationship is of degree 2.
- Ternary relationship is of degree 3.
- Relationships are often binary.

On the basis of degree of a relationship set, a relationship set can be classified into the following types:

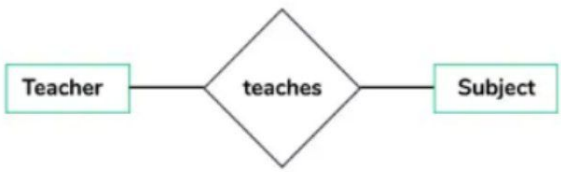
Unary Relationship set

- Unary relationship set is a relationship set where only one entity set participates in a relationship set.
- Eg. One person is married to only one person.



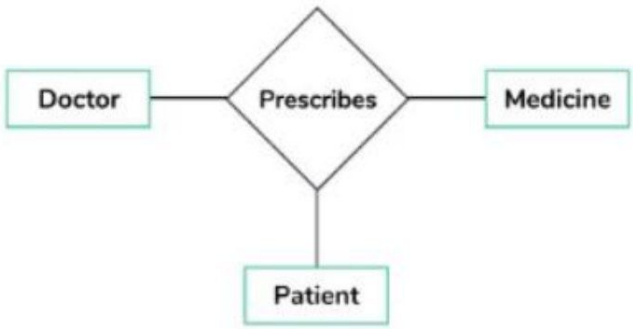
Binary Relationship set

- Binary relationship set is a relationship set where two entity sets participate in a relationship set.
- Eg. Teacher teaches a subject here 2 entities are teacher and subject for the relationship teacher teaches subject



Ternary Relationship set

- Ternary relationship set is a relationship set where three entity sets participate in a relationship set.
- Eg. In the real world, a patient goes to a doctor and doctor prescribes the medicine to the patient, three entities Doctor, patient and medicine are involved in the relationship "prescribes".



N-ary relationship set

- Ternary relationship set is a relationship set where n entity sets participate in a relationship set.

Constraints in E-R models

An ER enterprise schema may define certain constraints to which the contents of a database must conform. We have

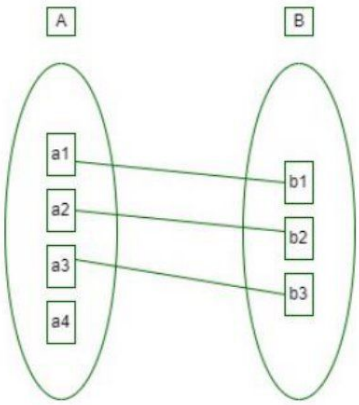
i. Mapping cardinalities
ii. Participation constraints

Mapping cardinalities

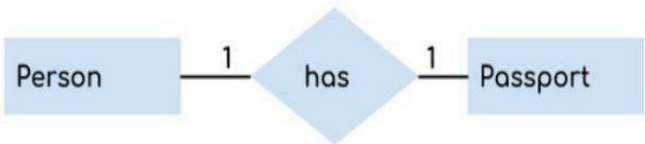
- Mapping cardinalities, or cardinality ratios, express the number of entities with which another entity can be associated via a relationship set.
- Mapping cardinalities are most useful in describing binary relationship sets, although they can contribute to the description of relationship sets that involve more than two entity sets.

For a binary relationship set R between entity sets A and B, the mapping cardinality must be one of the following:

One to One: An entity in A is associated with at most one entity in B, and an entity in B is associated with at most one entity in A.

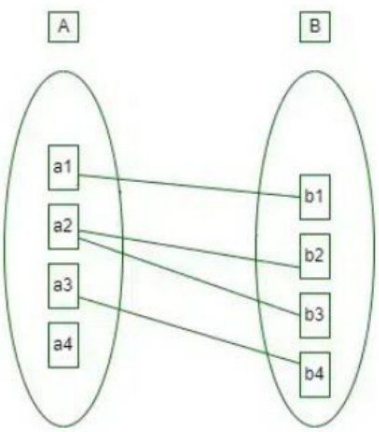


For example, a person has only one passport and a passport is given to one person.



One to many:

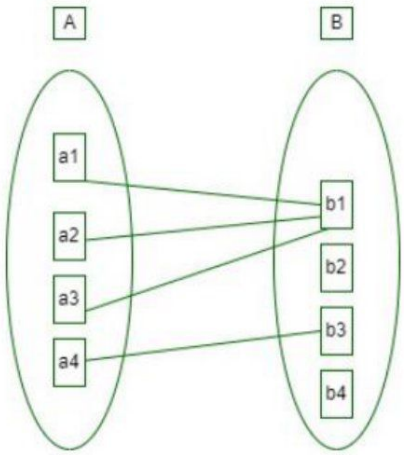
An entity in A is associated with any number (zero or more) of entities in B, and an entity in B, however, B can be associated with at most one entity in A.



For example, a customer can place many orders but a order cannot be placed by many customers.

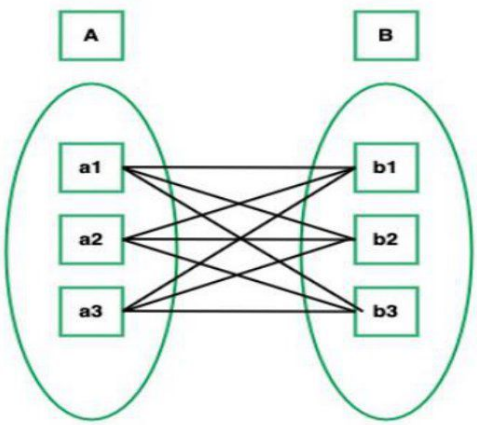
Many to One:

An entity in A is associated with at most one entity in B, and an entity in B, however, can be associated with any number (zero or more) of entities in A.



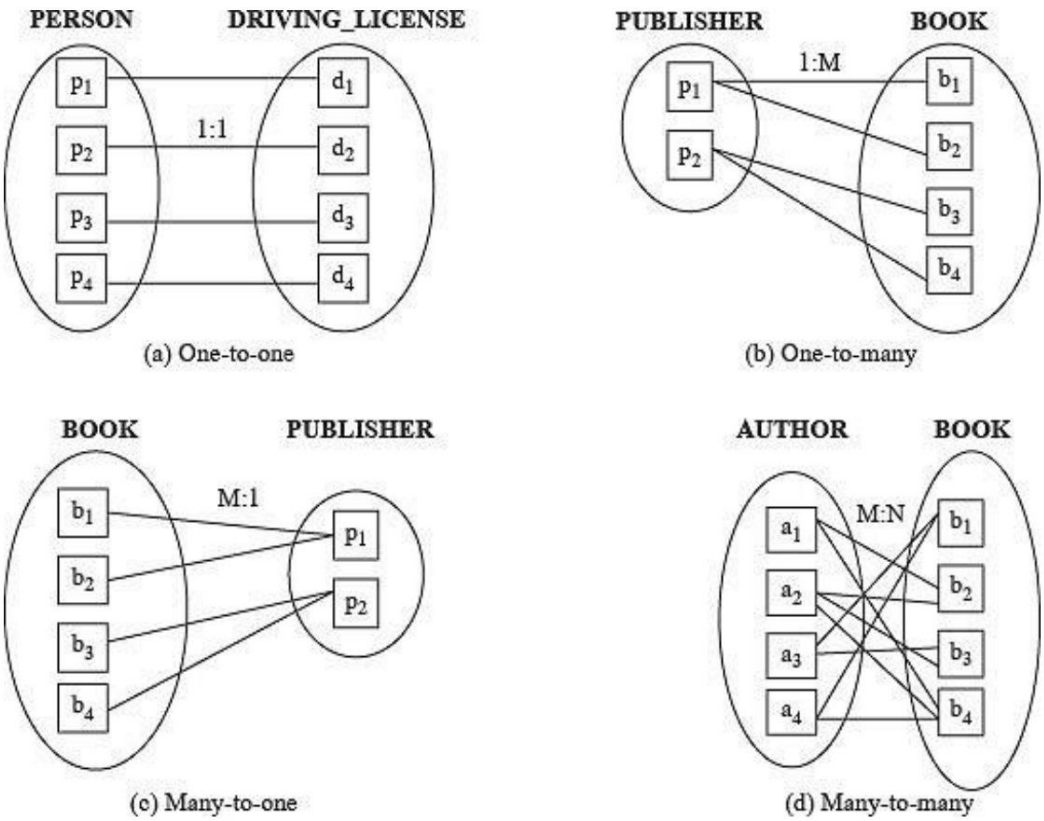
For example, many students can study in a single college, but a student cannot study in many colleges at the same time.

Many to Many: An entity in A is associated with any number (zero or more) of entities in B, and an entity in B is associated with any number (zero or more) of entities in A.



Eg. Student can be assigned to many projects and a project can be assigned to many students.

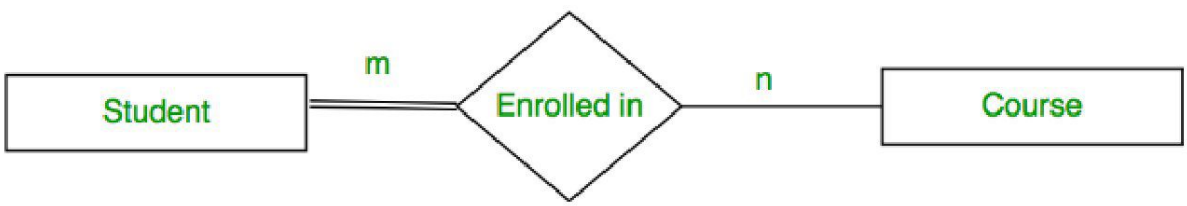
Alternative Example



Participation Constraints

Participation Constraint is applied to the entity participating in the relationship set.
1. Total Participation – Each entity in the entity set must participate in the relationship. If each student must enroll in a course, the participation of students will be total. Total participation is shown by a double line in the ER diagram.

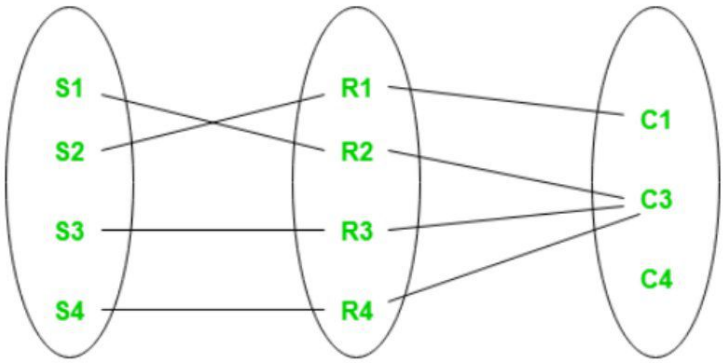
2. Partial Participation – The entity in the entity set may or may NOT participate in the relationship. If some courses are not enrolled by any of the students, the participation in the course will be partial.



Total Participation and Partial Participation

The diagram depicts the 'Enrolled in' relationship set with Student Entity set having total participation and Course Entity set having partial participation.

Using Set, it can be represented as



Set representation of Total Participation and Partial Participation

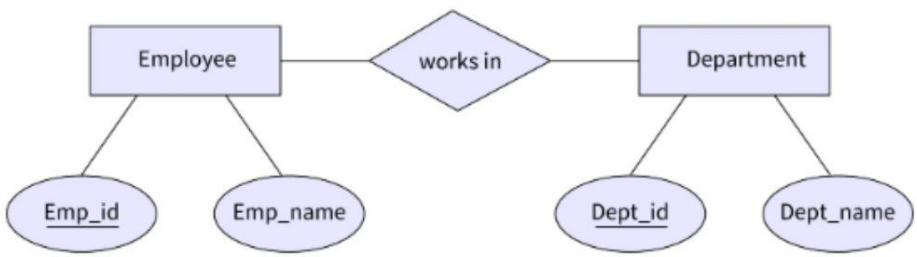
Every student in the Student Entity set participates in a relationship but there exists a course C4 that is not taking part in the relationship.

Strong and weak entity

Strong Entity

- A strong entity is not dependent on any other entity in the schema.
- A strong entity will always have a primary key.
- Strong entities are represented by a single rectangle.
- The relationship of two strong entities is represented by a single diamond.
- A strong entity set is a set that is made up of many strong entities.
- It may or may not participate in a relationship between entities.

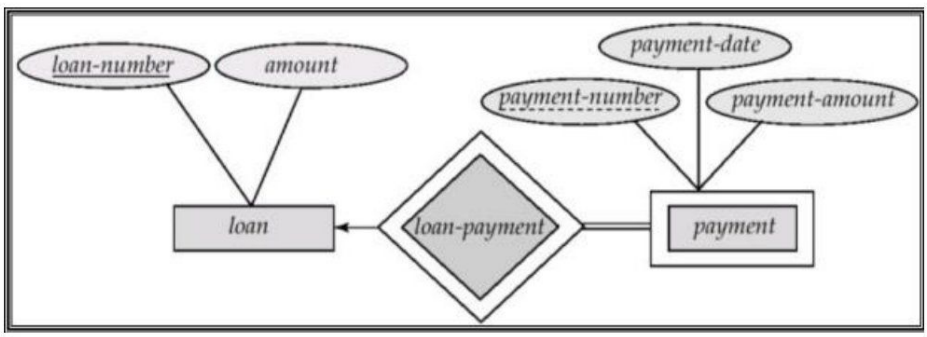
Example:



- In the given image, we have two strong entities namely Employee and Department hence they are represented using a single rectangle.
- The relationship between them is works in i.e it gives information about an employee working in a particular department hence it is represented using a single diamond.
- In the above image, if we remove the relationship between the two entities then also the two entities will exist i.e Employee as well as Department will exist since they both are independent of each other, this explains **the independent nature of strong entities.**

Weak entity

- A weak entity is dependent on a strong entity to ensure its existence.
- Unlike a strong entity, a weak entity does not have any primary key. It instead has a partial discriminator key.
- A weak entity is represented by a double rectangle.
- The discriminator (or partial key) of a weak entity set is the set of attributes that distinguishes among all the entities of a weak entity set.
- The relation between one strong and one weak entity is represented by a double diamond. This relationship is also known as identifying relationship.
- It always participates in the relationship between entities since its existence is dependent on other strong entities and it always has total participation.



- We underline the discriminator of a weak entity set with a dashed line.
- payment-number – discriminator (partial key)of the payment entity set
- Primary key for payment – (loan-number, payment-number)

Keys

- ✓ Key is a way to specify how entities within a given entity set are distinguished.
- ✓ Conceptually, individual entities are distinct; from a database perspective, however, the difference among them must be expressed in terms of their attributes.
- ✓ Therefore, the values of the attribute values of an entity must be such that they can uniquely identify the entity. In other words, no two entities in an entity set are allowed to have exactly the same value for all attributes.
- ✓ A key allows us to identify a set of attributes that is enough to distinguish entities from each other.
- ✓ Keys also help uniquely identify relationships, and thus distinguish relationships from each other.

Role of keys in relational database

- KEYS in DBMS is an attribute or set of attributes which helps us to identify a row(tuple) in a relation(table).
- In a real-world application, a table could contain thousands of records. Moreover, the records could be duplicated. Keys in RDBMS ensure that you can uniquely identify a table record despite these challenges.
- They allow us to find and establish the relation between two tables.
- Help us to enforce identity and integrity in the relationship**.**

Let us consider the table named **Employee_info**

Employee_id	Name	Address	Pan_no	Age
1	Pradip	Kathmandu	55821	43
2	Hari	Pokhara	55837	32
3	Nikita	Chitwan	55237	24
4	Pradip	Butwal	55345	43
5	Sita	Lalitpur	55432	48
6	Ramesh	Chitwan	55755	25
7	Hari	Pokhara	55896	32

- Now to fetch any particular record from such dataset, you will have to apply some conditions, but what if there is duplicate data present and every time you try to fetch some data by applying certain condition, you get the wrong data. How many trials before you get the right data?
- To avoid all this, Keys are defined to easily identify any row of data in a table.

Various types of keys are:

1. Super key
2. Candidate key
3. Primary key
4. Alternate key
5. Foreign key
6. Composite key

1. Super key

✓ Super Key is defined as a set of attributes within a table that can uniquely identify each record within a table. Super Key is a superset of Candidate key.

✓ In above example following are the examples of super keys

{Employee_id}
{Pan_no}
{Employee_id,Name}
{Pan_no,Name}
{Employee_id,Address} etc.

2. Candidate key

- ✓ A set of minimal attributes that can identify each tuple uniquely in the given relation is called a candidate key.
- ✓ The value of candidate key must always be unique.
- ✓ It is possible to have multiple candidate keys in a relation.
- ✓ In above example following are the examples of candidate keys

  {Employee_id}

  {Pan_no}

3. Primary key

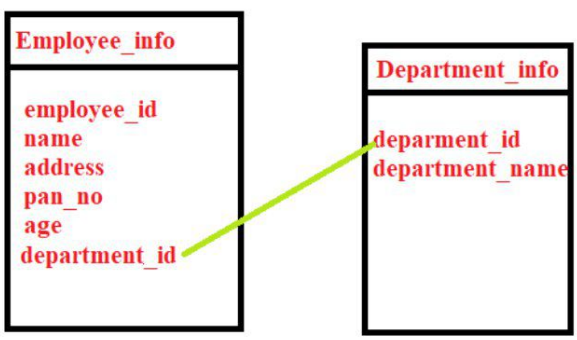
- ✓ The primary key is a candidate key that is most appropriate to become the main key for any table. It is a key that can uniquely identify each record in a table.
- ✓ In the above example {Employee_id} can be chosen are primary key.

4.Alternate key

- ✓ Candidate keys that are left unimplemented or unused after implementing the primary key are called as alternate keys
- ✓ In this example {Pan_no} act as alternate keys.

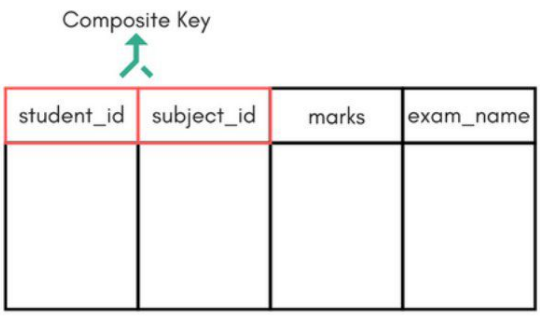
5. Foreign key

- ✓ Foreign keys are the column of the table used to point to the primary key of another table.
- ✓ Every employee works in a specific department in a company, and employee and department are two different entities. So we can't store the department's information in the employee_info table. That's why we link these two tables through the primary key of one table.
- ✓ We add the primary key of the DEPARTMENT table, Department_id, as a new attribute in the employee_info table.
- ✓ In the EMPLOYEE table, Department_id is the foreign key, and both the tables are related.



6.Composite key

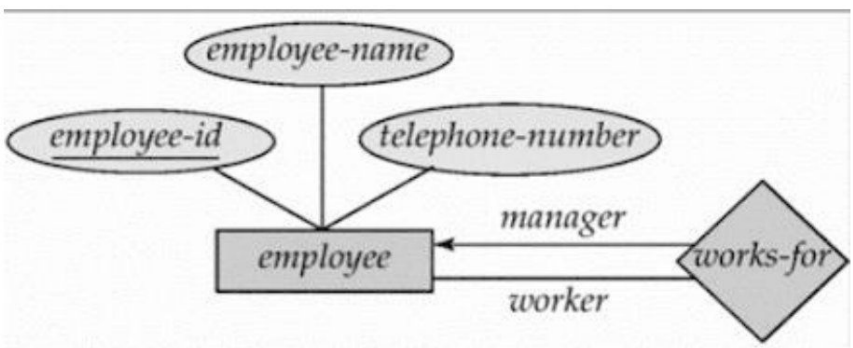
- ✓ Key that consists of two or more attributes that uniquely identify any record in a table is called Composite key. But the attributes which together form the Composite key are not a key independently or individually.
- ✓ In the above picture we have a Score table which stores the marks scored by a student in a particular subject.
- ✓ In this table student_id and subject_id together will form the composite key



Score Table - To save scores of the student for various subjects.

Roles in E-R diagram

- It is a function that plays(relationship) in an entity called its role.
- Roles are normally explicit and not specified.
- Useful when the relationship meaning needs to be clarified.
- Example, the relationship works-for define in employees as manager or worker. The labels “manager” and “worker” are called roles; they specify how employee entities interact via the works-for relationship set.
- Roles are indicated in E-R diagrams by labeling the lines that connect diamonds to rectangles.
- Role labels are optional and are used to clarify the semantics of the relationship.

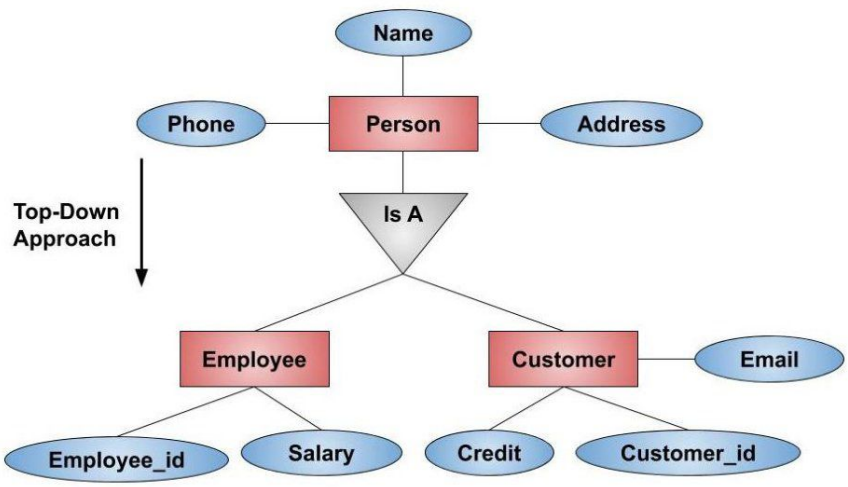


EER(Extended / Enhanced ER model)

- Today the complexity of the data is increasing so it becomes more and more difficult to use the traditional ER model for database modeling.
- To reduce this complexity of modeling we must make improvements or enhancements to the existing ER model to make it able to handle the complex application in a better way.
- Enhanced entity-relationship diagrams are advanced database diagrams very similar to regular ER diagrams which represent the requirements and complexities of complex databases.
- It is a diagrammatic technique for displaying the Sub Class and Super Class; Specialization and Generalization; Aggregation etc.

Specialization

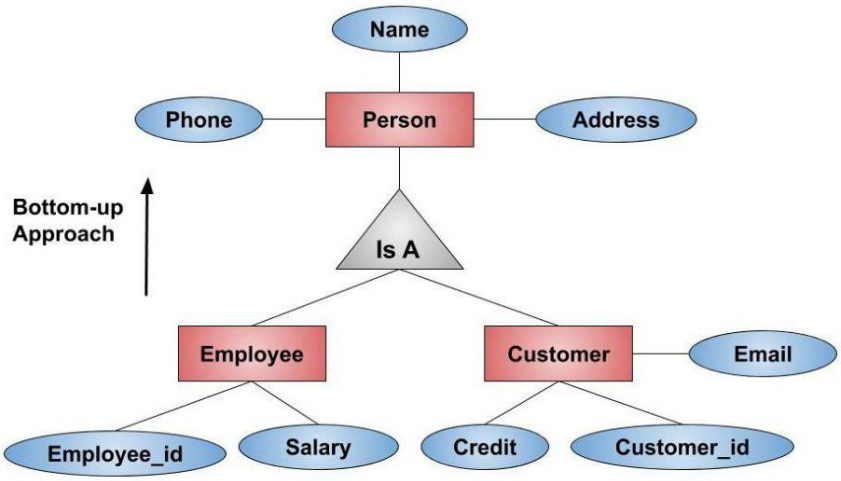
- The process of designating sub grouping within an entity set is called specialization.
- Top-down design process; we designate subgroupings within an entity set that are distinctive from other entities in the set.
- These subgroupings become lower-level entity sets that have attributes or participate in relationships that do not apply to the higher-level entity set.
- Depicted by a triangle component labeled ISA (E.g. customer "is a" person).
- Attribute inheritance – a lower-level entity set inherits all the attributes and relationship participation of the higher-level entity set to which it is linked.



Specialization

Generalization

- A bottom-up design process – combine a number of entities sets that share the same features into a higher-level entity set.
- Generalization is a containment relationship that exists between a higher level entity set and one or more lower level entity sets.
- Specialization and generalization are simple inversions of each other; they are represented in an E-R diagram in the same way.



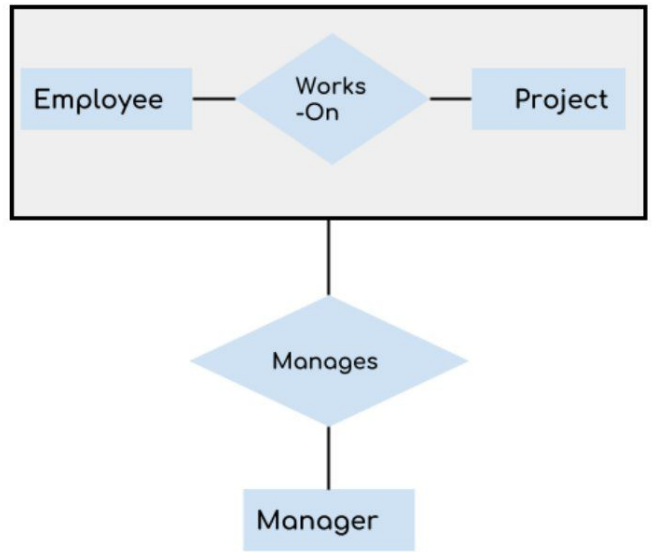
Generalization

Generalization vs specialization

Generalization	Specialization
Generalization works in Bottom-Up approach.	Specialization works in top-down approach.
Generalization can be defined as a process of creating groupings from various entity sets	Specialization can be defined as process of creating subgrouping within an entity set
In Generalization process, what actually happens is that it takes the union of two or more lower-level entity sets to produce a higher-level entity sets.	Specialization is the reverse of Generalization. Specialization is a process of taking a subset of a higher-level entity set to form a lower-level entity set
Generalization process starts with the number of entity sets and it creates high-level entity with the help of some common features	Specialization process starts from a single entity set and it creates a different entity set by using some different features.
Generalization is normally applied to group of entities.	We can apply Specialization to a single entity.
In Generalization, size of schema gets reduced.	In Specialization, size of schema gets increased.

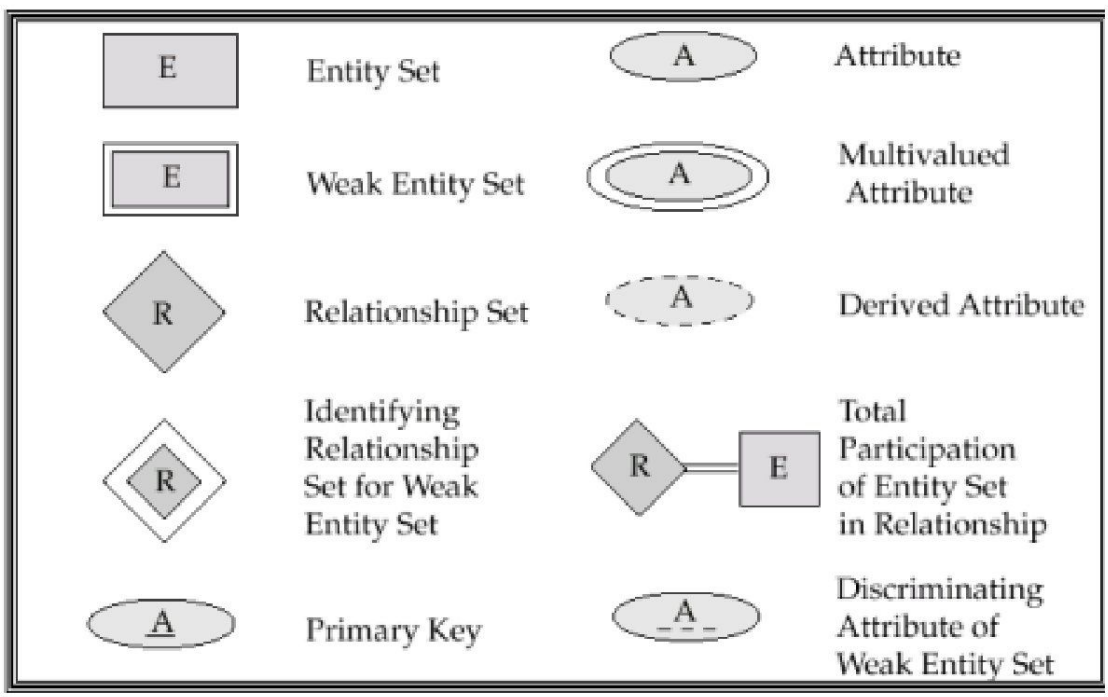
Aggregation

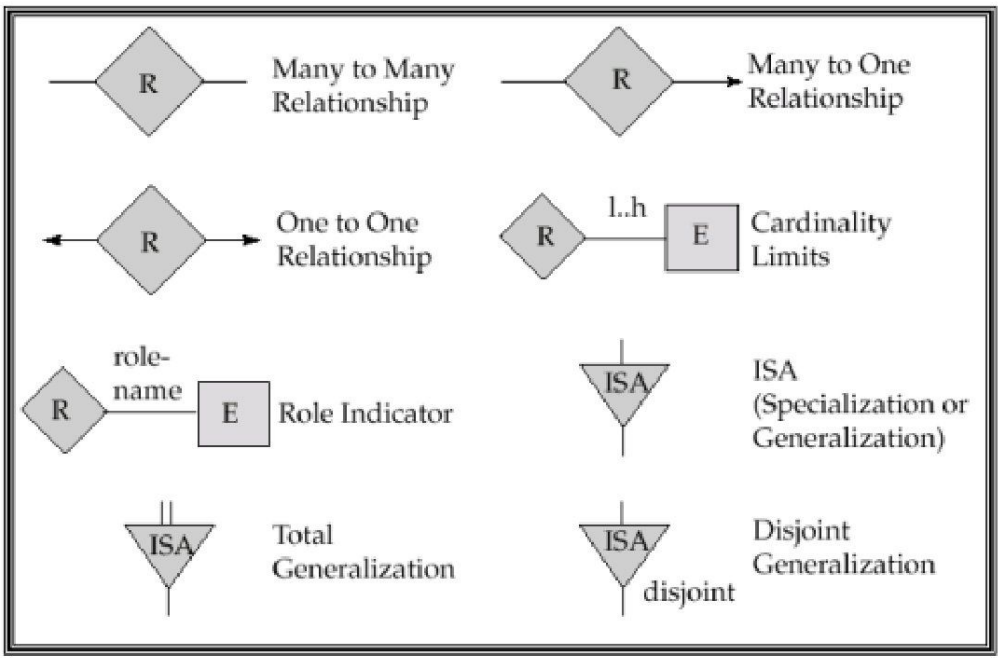
Aggregation in DBMS (Database Management System) is a process of combining one or more entities into a single one that is the more meaningful entity. Also, it is a process in which a single entity does not make sense in a relationship, and one cannot infer results from that relationship so the relationship of one or more entities acts as one entity.



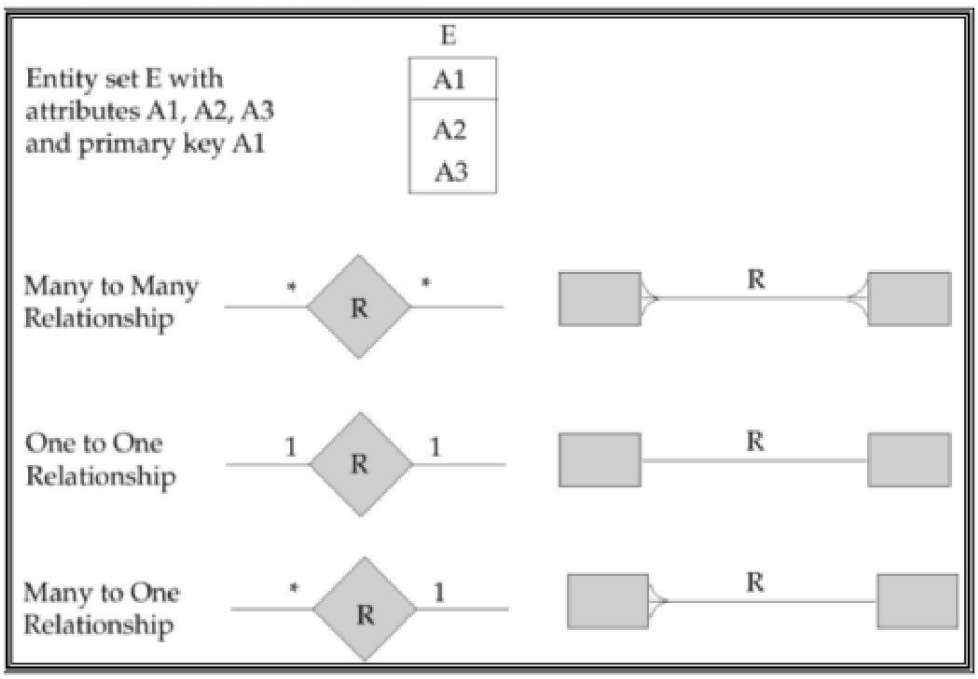
In the real world, we know that a manager not only manages the employee working under them, but he has to manage the project as well. In such scenario if entity "Manager" makes a "manages" relationship with either "Employee" or "Project" entity alone then it will not make any sense because he has to manage both. In these cases, the relationship of two entities acts as one entity. In our example, the relationship "Works-On" between "Employee" & "Project" acts as one entity that has a relationship "Manages" with the entity "Manager".

Summary of Symbols Used in E-R Notation

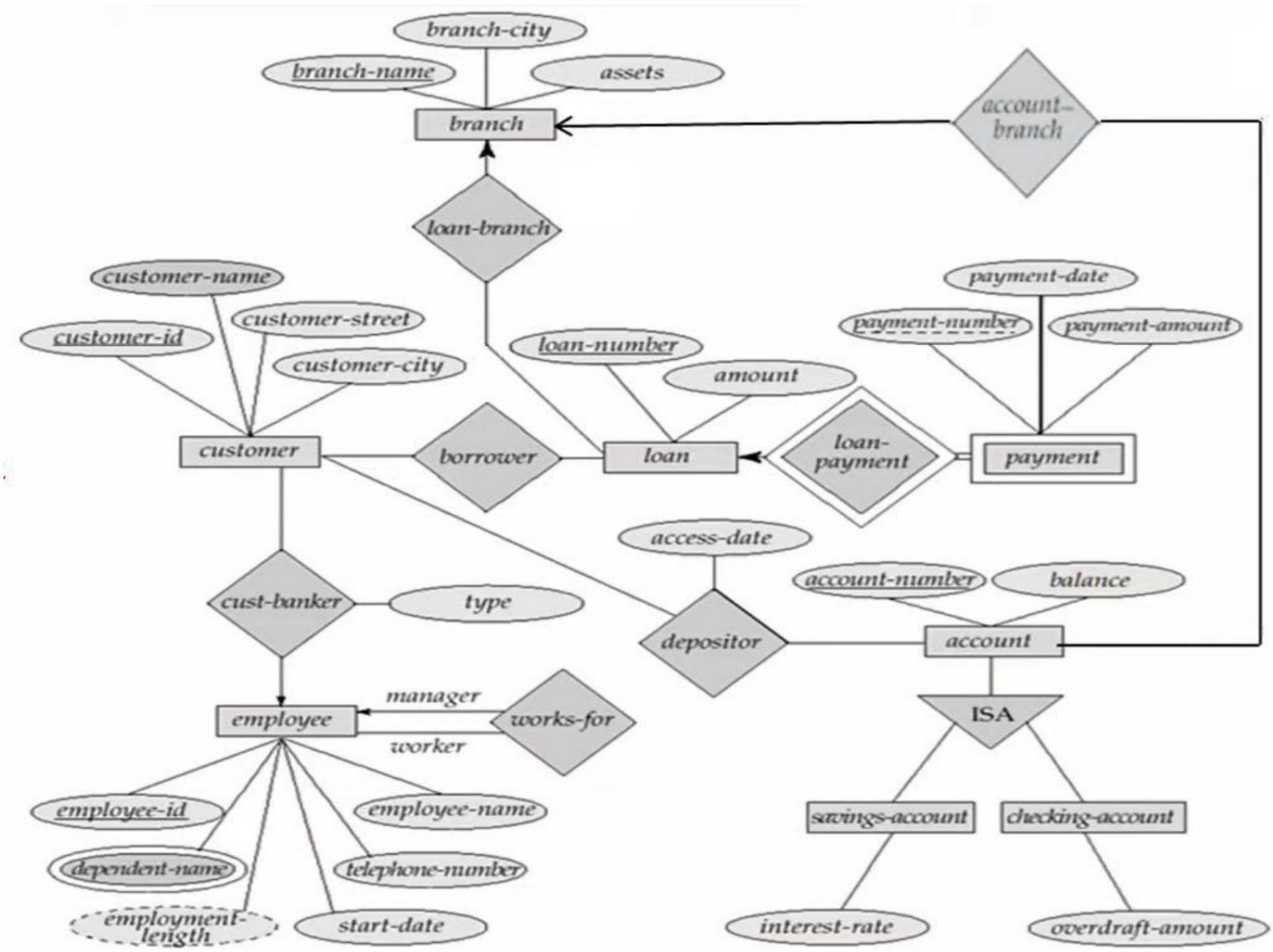




Alternative ER notations



Draw an ER-diagram for Banking enterprise that keeps the information about the employee, customer, loan, account and payment.



Introduction to Relational Model

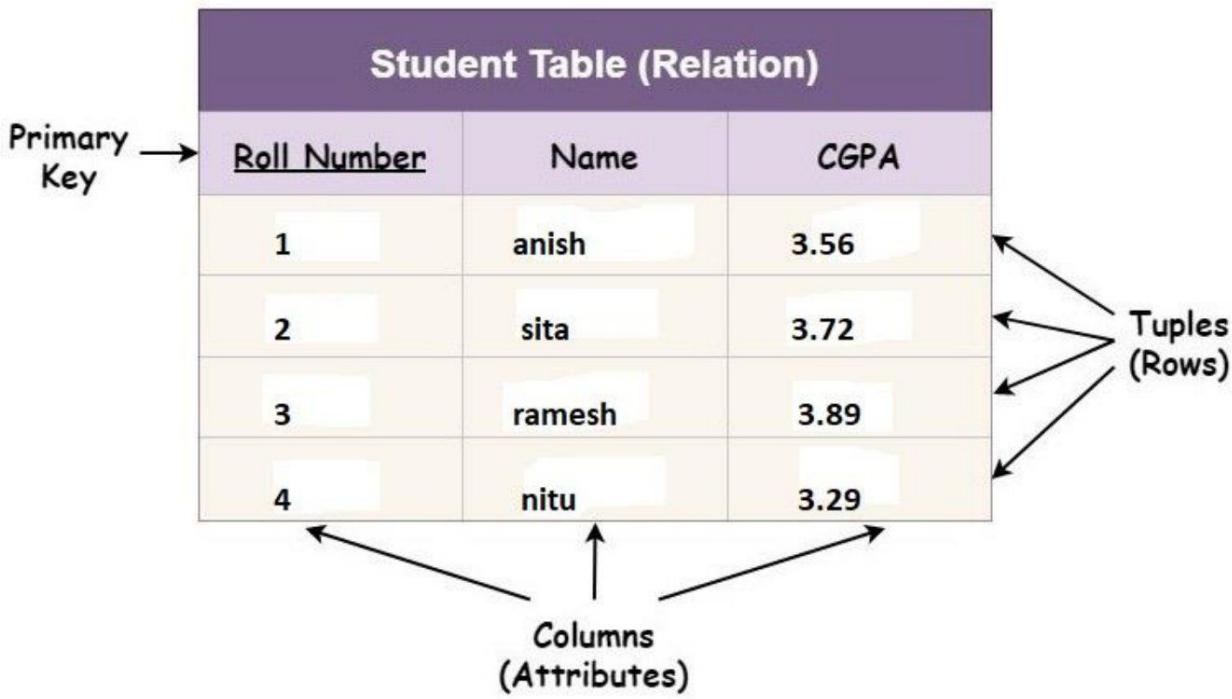
Definitions and Terminology

- Relational model is today a primary data model in which database is represented as a collection of Relation where each relation is represented by a two dimensional table.
- The relational model was first proposed by E.F. Codd in 1970.
- Because of its simplicity, the relational model is now the dominant model for commercial data processing operation.

Structure of relational model

- In this model, the data is organized into a collection of two-dimensional inter-related tables, also known as relations.
- Each relation is a collection of columns and rows.
    - ✓ column represents the attributes of an entity
    - ✓ the rows (or tuples) represents the records.

Consider a case we wish to store the name, the CGPA attained, and the roll number of all the students of a particular class. This structured data can be easily stored in a table as described below.



As we can notice from the above relation:

- Any given row of the relation indicates a student i.e. the row of the table describes a real-world entity.
- The columns of the table indicate the attributes related to the entity. In this case, the roll number, name and CGPA of student.

- ✓ Relational database is a collection of organized set of tables related to each other, and from which data can be accessed easily.
- ✓ **A Relational Database management System (RDBMS) is a database management system based on the relational model .It is used to manage Relational database.**
- ✓ Examples of RDBMS are Oracle, MySQL, Microsoft SQL Server, PostgreSQL etc.

As discussed earlier, a relational database is based on the relational model. This database consists of various components based on the relational model. These include:

Relation: Two-dimensional table used to store a collection of data elements.

Tuple: Each row of a table is known as record. It is also known as tuple. For example, the following row is a record that we have taken from the above table.

1	anish	3.56

Attribute: Column of the relation, depicting properties that define the relation.
Eg. Roll_Number, Name, CGPA

Domain: A domain is a set of permitted values for an attribute in table. For example, a domain of CGPA must be in the range of 0 to 4.

Relation Schema: A relation schema defines the structure of the relation and represents the name of the relation with its attributes. e.g. student (Roll_Number, Name, CGPA) is the relation schema for **student**. If a schema has more than 1 relation, it is called Relational Schema.

Relational Instance: It is the collection of records present in the relation at a given time. Above table shows the relation instance of **student** at a particular time. It can change whenever there is an insertion, deletion, or update in the database.

Degree: It is the total number of attributes present in the relation.eg. The **Student** relation defined above has degree 3.

Cardinality: It specifies the number of entities involved in the relation i.e., it is the total number of rows present in the relation. The **student** relation defined above has cardinality 4.

Properties of Relational model

- Each relation (or table) in a database has a unique name
- An entry at the intersection of each row and column is atomic (Each relation cell contains exactly one atomic (single) value)
- Each row is unique; no two rows in a relation are identical
- Each attribute (or column) within a table has a unique name
- Tuples in a relation do not have to follow a significant order as the relation is not order-sensitive.
- Similarly, the attributes of a relation also do not have to follow certain ordering, it's up to the developer to decide the ordering of attributes.

Advantages of Relational model

- ✓ A relational database model is much simpler compared to other data models because data is stored in the form of rows and columns.
- ✓ Since there are several tables in a relational database, certain tables can be made to be confidential. These tables are protected with username and password such that only authorized users will be able to access them. The users are only allowed to work on that specific table.
- ✓ Relational database uses primary keys and foreign keys to make the tables interrelated to each other. Thus, all the data which is stored is non-repetitive. Which means that the data does not duplicate. Therefore, the data stored can be guaranteed to be accurate.
- ✓ It is flexible, so one can get the data in the form which he/she wants. He/she can extract the information very easily and information can also be manipulated by using various operators such as project, join, etc.
- ✓ The Structure of Relational database can be changed without having to change any application.

Disadvantages of relational model

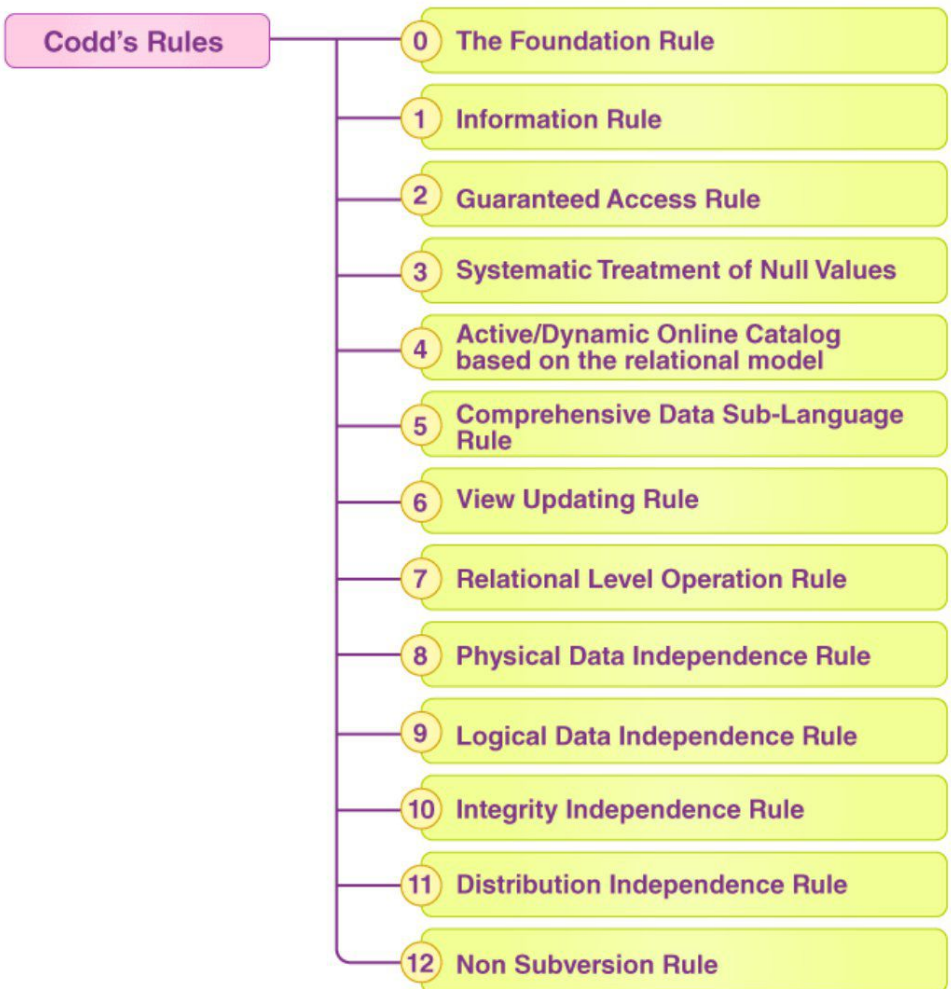
- ✓ The performance of the relational model depends upon the number of relations present in the database.
- ✓ Hence, as the number of tables increases, the requirement of physical memory increases.
- ✓ The structure becomes complex and there is a decrease in the response time for the queries.
- ✓ Because of all these factors, the cost of implementing a relational database increase.

Difference between DBMS and RDBMS

DBMS	RDBMS
DBMS applications store data as file.	RDBMS applications store data in a tabular form.
Data elements need to access individually.	Multiple data elements can be accessed at the same time.
No relationship between data.	Data is stored in the form of tables which are related to each other.
Normalization is not present in DBMS.	Normalization is present in RDBMS
DBMS does not support distributed database.	RDBMS supports distributed database.
DBMS is meant to be for small organization and deal with small data.	RDBMS is designed to handle large amount of data.
The software and hardware requirements are low.	The software and hardware requirements are higher.
The data in a DBMS is subject to low security levels with regards to data manipulation.	It features multiple layers of security while handling data.
Examples: Window Registry, Forxpro, dbase III plus etc.	Examples: MySQL, PostgreSQL, SQL Server, Oracle, Microsoft Access etc.

E.F. Codd's 12 Rules for RDBMS

- ✓ E.F Codd was a Computer Scientist who invented the Relational model for Database management. Based on relational model, the Relational database was created.
- ✓ Codd proposed 13 rules popularly known as Codd's 12 rules to test DBMS's concept against his relational model.
- ✓ Codd's rule actualy define what quality a DBMS requires in order to become a Relational Database Management System(RDBMS).



Rule 0: The Foundation Rule

The database must be in relational form. So that the system can manage the database through its relational capabilities.

Rule 1: The Information Rule

A database contains various information, and this information must be stored in each cell of a table in the form of rows and columns.

Rule 2: The Guaranteed Access Rule

Every single or precise data (atomic value) may be accessed logically from a relational database using the combination of primary key value, table name, and column name.

Rule 3: The Systematic Treatment of Null Values

This rule defines the systematic treatment of Null values in database records. The null value has various meanings in the database, like missing the data, no value in a cell, inappropriate information, unknown data and the primary key should not be null.

Rule 4: The Dynamic/Active Online Catalog on the basis of the Relational Model

Database dictionary(catalog) is the structure description of the complete Database and it must be stored online. The Catalog must be governed by same rules as rest of the database. The same query language should be used on catalog as used to query database.

Rule 5: The Comprehensive Data SubLanguage Rule

The relational database supports a variety of languages, and in order to access the database, the language has to be linear, explicit, or a well-defined syntax, character strings. It must support the following operations: view definition, integrity constraints, data manipulation, data definition, as well as limit transaction management. It is considered a DB violation if the DB permits access to the data and information without the use of any language.

Rule 6: The View Updating Rule

A view table can theoretically be updated, and DB systems must update them in practice.

Rule 7: The Relational Level Operation (or High-Level Insert, Delete, and Update) Rule

A database system should follow high-level relational operations such as insert, update, and delete in each level or a single row. It also supports union, intersection and minus operation in the database system.

Rule 8: The Physical Data Independence Rule

The working of a database system should be independent of the physical storage of its data. If a file is modified (renamed or moved to another location), it should not interfere with the working of the system.

Rule 9: The Logical Data Independence Rule

It indicates that any modifications made at the logical level (or the table structures) should not have an impact on the user's experience (application). For example, if a table is split into two separate tables or into two table joins in order to produce a single table, the application at the user view should not be affected.

Rule 10: The Integrity Independence Rule

A database must maintain integrity independence when inserting data into table's cells using the SQL query language. All entered values should not be changed or rely on any external factor or application to maintain integrity. It is also helpful in making the database-independent for each front-end application

Rule 11: The Distribution Independence Rule

This rule denotes that a database must function properly even if it's stored in multiple locations and used by various end-users. Let's say a person uses an application to access the database. In such a case, they must not be aware that another user is using the same data, and thus, the data they always obtain is only available on one site. The database can be accessed by end-users, and each user's access data must be independent in order for them to run SQL queries.

Rule 12: The Non-Subversion Rule

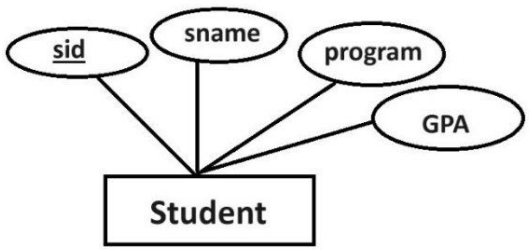
The non-subversion rule defines RDBMS as a SQL language to store and manipulate the data in the database. If a system has a low-level or separate language other than SQL to access the database system, it should not subvert or bypass integrity to transform data**.**

Reducing ER diagram to relational schema

Rule 1: Strong Entity Set with only simple attributes

- ✓ Attributes of the schema will be attributes of the entity set.
- ✓ The primary key of the schema will be the key attribute of the entity set.

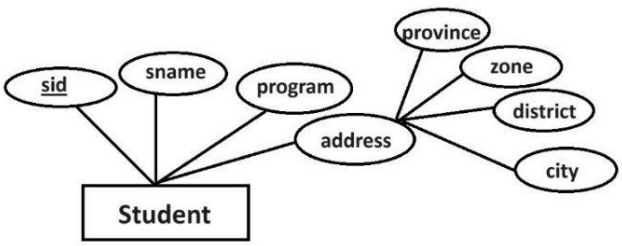
Example:



Schema:

Student(sid, sname, program, GPA)

Rule 2: Strong Entity Set with composite attributes



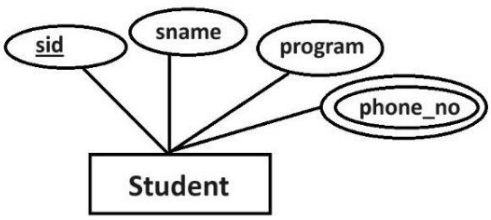
- ✓ A strong entity set with any number of composite attributes requires only one schema in relational model.
- ✓ While conversion, simple attributes of the composite attributes are taken into account and not the composite attribute itself.

Schema:

Student(sid, sname, program, province, zone, district, city)

Rule 3: Strong Entity set with multivalued attributes.

- ✓ A strong Entity set with any number of multivalued attributes will require two schemas in relational model.
- ➢ One schema will contain all the simple attributes with the primary key.
- ➢ Other schema will contain the primary key and all the multivalued attributes.



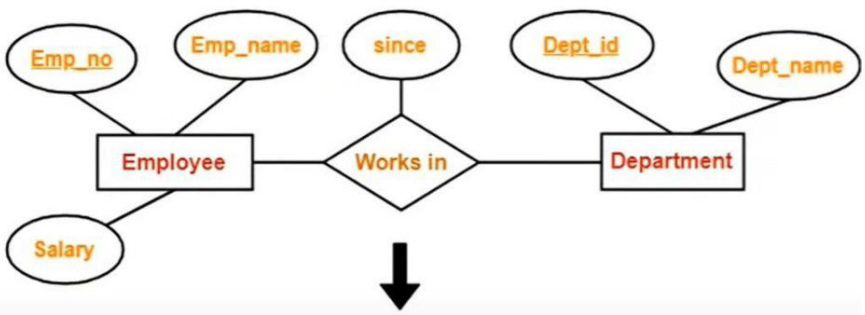
Schema:

Student(sid, sname, program)

Student_phone_no(sid, phone_no)

Rule 4: Translating Relationship set into a schema.

- ➢ A relationship set will require one schema in relational model.
- ➢ Attributes of a schemas are:
  - ✓ Primary key attributes of the participating entity set.
  - ✓ Its own descriptive attribute if any.
- ➢ A set of non-descriptive attributes will be the primary key.



Schemas:

Employee(Emp_no, Emp_name, Salary)

Workin(Emp_no, Dept_id, since)

Department(Dept_id, Dept_name)

Rule 5: For Binary relationships with cardinality ratios

the following four cases are possible:

Case-1: Binary relationship with cardinality ratio m:n

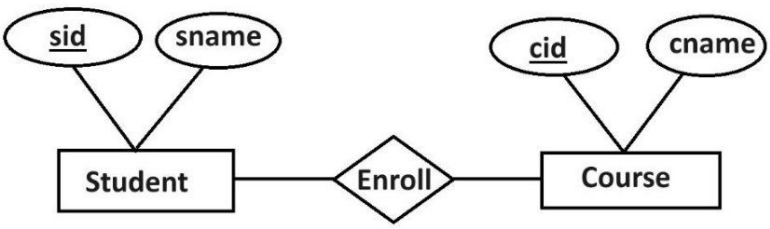
Case-2: Binary relationship with cardinality ratio 1:n

Case-3: Binary relationship with cardinality ratio m:1

Case-4: Binary relationship with cardinality ratio 1:1

Case-1: Binary relationship with cardinality ratio m:n

Union of the primary key attributes from the participating entity sets become the primary key of the relationship.



Schemas:

Student(sid, sname)

Enroll(sid, cid)

Course(cid, cname)

If enroll date is mentioned as descriptive attribute, then

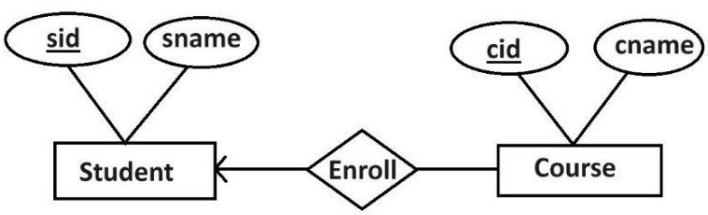
Enroll(sid, cid, enroll_date)

Case-2: Binary relationship with cardinality ratio m:1/1:m

- ✓ Construct two schemas, one for entity set for 1 side and another for entity set at M side.
- ✓ Add the descriptive attributes and a reference of the primary key of 1 side to the entity set at M side.

For Binary relationship with cardinality ratio 1:m

Example:



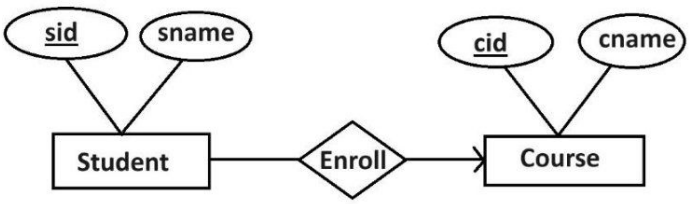
Schemas:

Student(sid, sname)

Course_enroll(sid, cid, cname)

For Binary relationship with cardinality ratio m:1

Example:



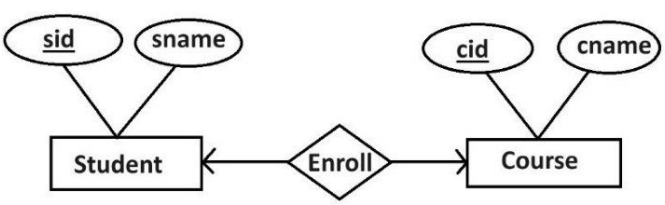
Schemas:

Student_enroll(sid, sname, cid)

Course(cid, cname)

Case-3: Binary relationship with cardinality ratio 1:1

Construct two schemas. In this case, either side can be chosen to act as the many side. That is, extra attributes can be added to either side of table corresponding to the two entity sets, but not at the same time.



- ✓ **If student entity set is considered many side**

  Schemas:

  Student_enroll(sid, sname, cid)
  Course(cid, cname)

- ✓ **If course entity set is considered many side**

  Schemas:

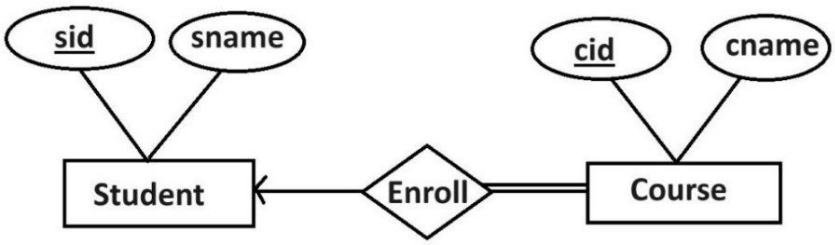
  Student(sid, sname)
  Course_enroll(sid, cid, cname)

Rule 6: For Binary Relationship With Both Cardinality Constraints and Participation Constraints

- ✓ Cardinality constraints will be implemented as discussed in Rule-05.
- ✓ Because of the total participation constraint, foreign key acquires NOT NULL constraint i.e. now foreign key can not be null.

Case-01: For Binary Relationship with Cardinality Constraint and Total Participation Constraint From One Side

Because cardinality ratio = 1 : n , so we will combine the entity set B and relationship set R.



Then, two schemas will be required-

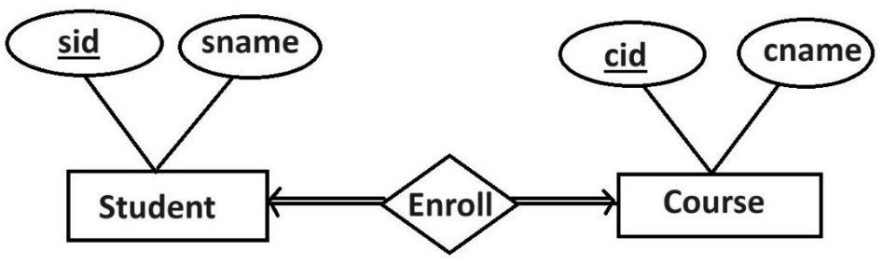
Student(sid, sname)

Student_Enroll(sid, cid, cname)

Because of total participation, foreign key sid has acquired NOT NULL constraint, so it can't be null now.

Case-02: For Binary Relationship with Cardinality Constraint and Total Participation Constraint from Both Sides

If there is a key constraint from both the sides of an entity set with total participation, then that binary relationship is represented using only single schema.

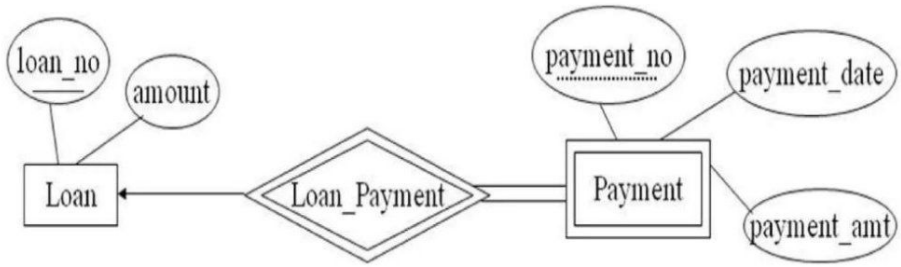


Here only one schema is required,

Student_enroll_course(sid, sname, cid, cname)

Rule 7: Representation of weak entity sets

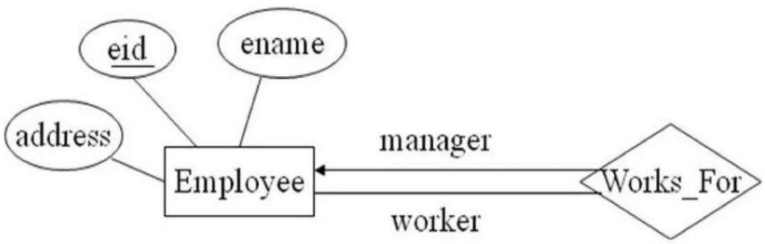
- ✓ A weak entity set becomes a schema that includes a column for the primary key of the identifying strong entity set.
- ✓ The primary key is constructed by the collection of foreign key and partial key.



Loan(loan_no, amount)

Payment(loan_no, payment_no, payment_date, payment_amt)

Rule 8: Representation of Recursive relationship sets



Two schemas will be constructed; one for entity set and one for relationship set.

Employee(eid, ename, address)

Works_for(mgrid,workerid)

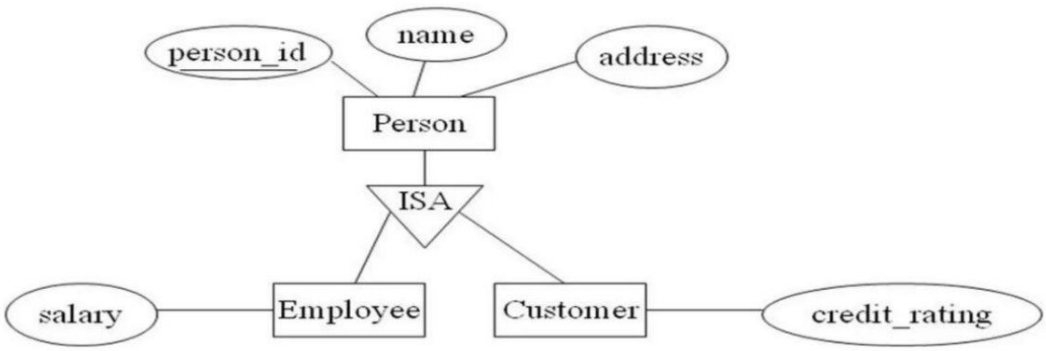
This ER diagram can also be represented by using a single relation schema. In such cases, the schema contains a foreign key for each tuple in the original entity set.

Employee(eid,ename,address,manager_id)

Rule 9:

Representation of Generalization/Specialization.

In case of generalization/specialization related ER diagram, one schema will be constructed for the generalized entity set and the schemas for each of the specialized entity sets.



Person(person_id,name,address)
Employee(person_id,salary)
Customer(person_id,credit_rating)

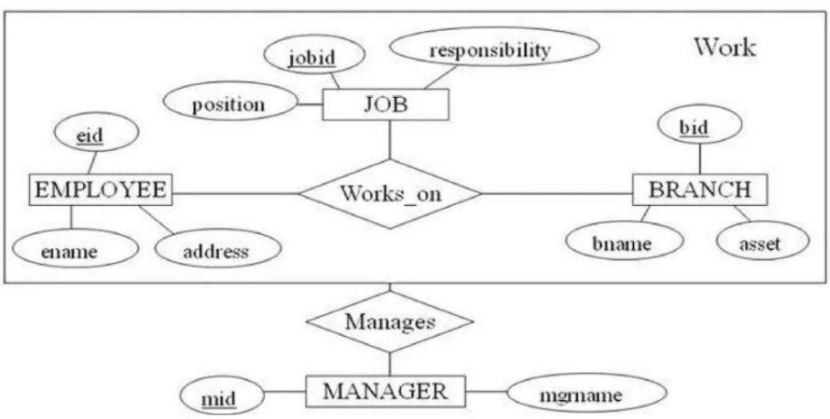
When the generalization/specialization is a disjointness case, the schemas are constructed only for the specialized entity sets.

Employee(Employee_id, name, address,salary)
Customer(customer_id, name, address, credit_rating)

Rule 10:

Representation of Aggregation

To represent aggregation, create a schema containing the primary key of the aggregated relationship ,primary key of the associated entity set and descriptive attributes(if any).



Employee (eid, name, address)

Branch(bid, bname, asset)

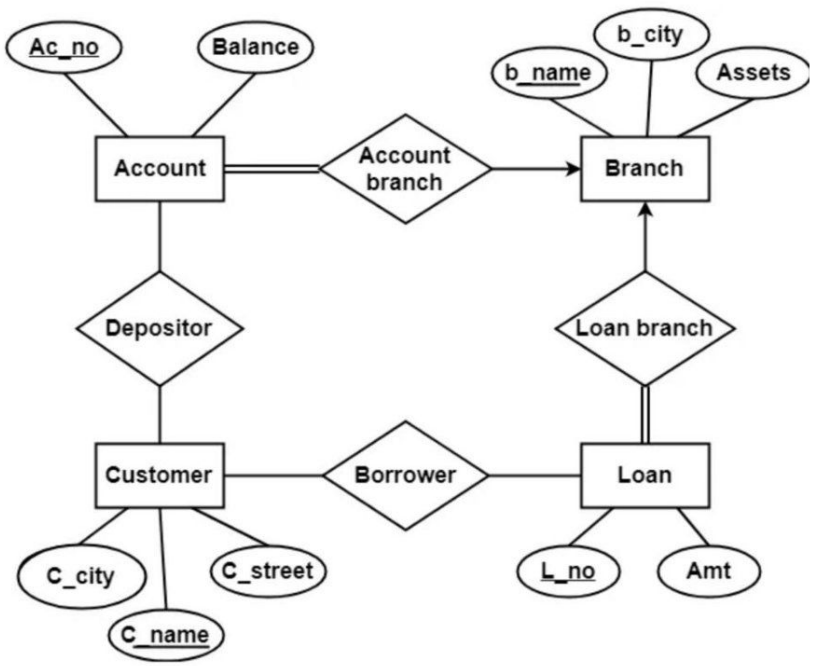
Job(jobid, position, responsibility)

Works_on(eid, bid, jobid)

Manager(mid, mgrname)

Manages(eid, bid, job_id, mid)

Example:



Applying the rules that we have learnt, minimum 6 tables will be required-

Account (Ac_no , Balance , b_name)

Branch (b_name , b_city , Assets)

Loan (L_no , Amt , b_name)

Borrower (C_name , L_no)

Customer (C_name , C_street , C_city)

Depositor (C_name , Ac_no)

Things to be understand

Many to Many cardinality

For example, a student can be enrolled many courses and a course can be enrolled by many students.

Student(sid,sname)

Enroll(sid,cid)

Course(cid,cname)

Table Student

sid	sname
1	Ram
2	Sita
3	Anish
4	Gita

Table Course

cid	cname
C1	C++
C2	DBMS
C3	Microprocessor

Table Enroll

sid	cid
1	C1
1	C2
2	C1
2	C2
3	C3
4	C2

Many to one cardinality

For example, a student can be enrolled only in one course, but a course can be enrolled by many students.

Student_enroll(sid,sname,cid)

Course(cid,cname)

Student_enroll

sid	sname	cid
1	Ram	c1
2	Sita	c1
3	Anish	c2
4	Gita	c3

Course

cid	cname
c1	C++
c2	DBMS
c3	Microprocessor

One to many cardinality

For example, a student can be enrolled many course, but a course can be enrolled by only one student.

Student_enroll(sid,sname)

Course(sid,cid,cname)

Student_enroll

sid	sname
1	Ram
2	Sita
3	Anish
4	Gita

Course

sid	cid	cname
1	c1	C++
2	c2	DBMS
1	c3	Microprocessor
2	c4	Instrumentation

Relational Algebra

- ✓ The relational algebra is a procedural query language.
- ✓ It consist set of operation that takes one or more relations as inputs and produce a new relation as output.
- ✓ The fundamental operations in relational algebra are selection, projection, union, set difference, Cartesian product, and rename.
- ✓ Set intersection, natural join, division and assignments other operations of relational algebra which can be define in terms of fundamental operations.

1. Fundamental Operations

- ✓ The fundamental operations **selection, projection and rename** on one relation so they called unary operations.
- ✓ Others operations **union, set difference and Cartesian product** operates on pairs of relations and so called binary operations.

1.1 Selection Operation

- ✓ The Select Operation selects tuples that satisfy a given predicate.
- ✓ Select is denoted by a lowercase Greek letter sigma (σ), with the predicate appearing as a subscript.
- ✓ The relation is specifying within parentheses after σ. That is, general structure of selection is $\sigma_{p(r)}$ where p is selection predicate.
- ✓ Formally, selection operation define as
  $\sigma_{p(r)} = \{t | t \in r \text{ and } p(t)\}$
  where p is formula in propositional calculus consisting terms connected by connectives: ^(and), ∨ (or), ¬ (not).
- ✓ Each term is in the format <attribute>op<attribute>or <constant>
  where op is one of the comparison operators: =,≠ ,<,≤,>,≥

Let us consider the following employee relation

emp_id	name	department	salary
1	ramesh	civil	68000
2	krishna	computer	75000
3	rita	software	65000
4	sita	mechanical	32000
5	juna	mechanical	85000
6	nikita	computer	60000
7	anish	civil	55000

Find all the employees working on computer department

$\sigma_{department="computer"}(employee)$

output

emp_id	name	department	Salary
2	krishna	computer	75000
6	nikita	computer	60000

Find all the employees whose salary is greater than 70000

$\sigma_{salary>70000}(employee)$

output

emp_id	name	department	salary
2	krishna	computer	75000
5	juna	mechanical	85000

Find all the employees with name sita and department is mechanical

$\sigma_{name= "sita" \wedge department="mechanical"}(employee)$

emp_id	name	department	salary
4	sita	mechanical	32000

Find all the employees whose department is either civil or computer

$\sigma_{department= "civil" \vee department="computer"}(employee)$

emp_id	name	department	salary
1	ramesh	civil	68000
2	krishna	computer	75000
6	nikita	computer	60000
7	anish	civil	55000

Find all the employees whose department is either civil or computer and name is not nikita

$\sigma_{((department="civil" \vee department="computer") \wedge name \neq nikita)}(employee)$

emp_id	name	department	salary
1	ramesh	civil	68000
2	krishna	computer	75000
7	anish	civil	55000

1.2 Projection operation

- ✓ The projection operation retrieves tuples for specified attributes of relation.
- ✓ It eliminates duplicate tuples in relation.
- ✓ The projection is denoted by uppercase Greek letter pi (∏).
- ✓ We need to specify attributes that we wish to appear in the result as a subscript to ∏.
- ✓ The general structure of projection is

$\prod_{A1,A2,\ldots,Ak}(r)$
where A1, A2, . .Ak are attributes of relation r.

Example:

Find name and their salary from employee relation

$\prod_{name,salary}(employee)$

output

name	salary
Ramesh	68000
krishna	75000
rita	65000
sita	32000
juna	85000
nikita	60000
anish	55000

Composition of relational operations

Relational algebra operations can be composed together into relational-algebra expression. This required for complicated query.

Example:

Find name and salary of employees of civil department

$\prod_{name,\ salary}(\sigma_{department="civil"}(employee))$

output

name	salary
ramesh	68000
anish	55000

Find name and department of employees whose salary is less than or equals to 60000

$\prod_{name, department}(\sigma_{salary \leq 60000}(employee))$

output

name	department
sita	mechanical
nikita	computer
anish	civil

1.3 Union operation

Suppose r and s are two relations, then union operation contains all tuples that appear in r, s, or both.

The union of two relations r and s are defines as

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

For r U s to be valid, it must hold

- ✓ r,s must have same arity (same number of attributes)
- ✓ The attribute domain must be compatible (e.g. domain of ith column of r must deals with same type of domain of ith column of s)
- ✓ Duplicate rows are eliminated by this operations

Let us consider two relations

depositer

customer_name	account_no
ram	A-101
sita	A-105
hari	A-205
nishant	A-405
gita	A-505

borrower

customer_name	loan_no
Aditya	L-224
nishant	L-185
mahesh	L-150
gita	L-174
ronish	L-145

Example:

Find name of all customers who have either account or loan

$\prod_{customer\_name}$(depositor) U $\prod_{customer\_name}$(borrower)

Output

customer_name
ram
sita
hari
nishant
gita
Aditya
Mahesh
ronish

1.4 Set difference Operation

- ✓ The set difference allows us to find tuples that are in one relation but not in another relation. The expression r-s produces a relation containing those tuples in r but not in s.
- ✓ Formally, let r and s are two relations then their difference r-s define as

  $r-s= \{t \mid t \in r \text{ and } t \notin s\}$

- ✓ The set difference must be taken between compatible relations. For r-s to be valid, it must hold

  - r and s must have the same arity (same number of attributes)
  - attribute domains of r and s must be compatible

Example:

Find name of all customer of the bank who have account but not loan

$\prod_{customer\_name}$(depositor) - $\prod_{customer\_name}$(borrower)

customer_name
ram
sita
hari

1.5 Cartesian product

- ✓ It allows us to combine information from any two relations.
- ✓ The Cartesian product operation denoted by cross (X).
- ✓ Cartesian product of two relations r and s, denoted by r X s returns a relation instance whose schema contains all the fields of r (in same order as they appear in r(followed all field of s (in the same order as they appear in s).
- ✓ The result of r×s contains one tuples <r,s> (concatenation of tuples of r and s) for each pair tuples t∈r, q∈s.

Formally, r X s={<t,q>|t∈r and q∈s}

Note:

If r and s are two relation having n and m number of attributes and p and q number of records respectively, then the Cartesian product of these two relation denoted by r X s results a new relation having (n+m) numberof attributes and (p*q) number of records

Example:

Relation Employee

emp_id	name
1	anish
2	sita
3	nitesh

Relation Department

dept_id	dept_name
1	computer
2	civil
3	electrical

Employee X Department

emp_id	name	dept_id	dept_name
1	anish	1	computer
1	anish	2	civil
1	anish	3	electrical
2	nitesh	1	computer
2	nitesh	2	civil
2	nitesh	3	electrical
3	sita	1	computer
3	sita	2	civil
3	sita	3	electrical

Example 2:

Relation borrower

customer_name	loan_number
X	L01
Y	L02

Relation loan

loan_number	branch_name	amount
L01	B1	5000
L02	B2	6000

Query: Find all customer who taken loan from branch "B1".

$\prod_{customer\_name}(\sigma_{borrower.loan\_number=loan.loan\_number}(\sigma_{branch\_name="B1"}(borrower \times loan)))$

Process:

borrower×loan

customer_name	borrower.loan_number	loan.loan_number	branch_name	amount
X	L01	L01	B1	5000
X	L01	L02	B2	6000
Y	L02	L01	B1	5000
Y	L02	L02	B2	6000

$\sigma_{branch\_name="B1"}(borrower \times loan)$

customer_name	borrower.loan_number	loan.loan_number	branch_name	amount
X	L01	L01	B1	5000
Y	L02	L01	B1	5000

$\sigma_{borrower.loan\_number=loan.loan\_number}(\sigma_{branch\_name="B1"}(borrower \times loan))$

customer_name	borrower.loan_number	loan.loan_number	branch_name	amount
X	L01	L01	B1	5000

$\prod_{customer\_name}(\sigma_{borrower.loan\_number=loan.loan\_number}(\sigma_{branch\_name="B1"}(borrower \times loan)))$

customer_name
X

1.6 Rename operation

- ✓ The result of relational-algebra expression does not have a name to refer it. It is better to give name to result relation.
- ✓ The rename operator is denoted by lower case Greek letter rho (ρ).
- ✓ Rename operation in relation-algebra expressed as $\rho_x(E)$ where E is a relational algebra expression and x is name for result relation. It returns the result of expression E under the name x.
- ✓ Since a relation r is itself a relational-algebra expression thus, the rename operation can also apply to rename the relation r (i.e. to get same relation under a new name).
- ✓ Rename operation can also used to rename attributes of relation. Assume a relational algebra expression E has arity n. Then expression $\rho_{x(A1,A2,...An)}(E)$ returns the result of expression E under the name x and it renames attributes to A1,A2,..,An.

Let us consider the relation

Customer(customer_id,customer_name,customer_city)

Example1:

To find all the customer_name from this relation algebra expression can be written as

$\prod_{customer\_name}(customer)$

This result of the expression can be renamed as

$\rho_{cname}(\prod_{customer\_name}(customer))$

Example 2:

Rename the relation named customer

$\rho_{newcustomer}(customer)$

Here relation named customer can be renamed as newcustomer.

Example 3:

Relation named can also be renamed as well their attributes also

$\rho_{c(cid,cname,ccity)}$ (Customer)

Her relation name customer is renamed as c and their attributes customer_id,customer_name,customer_city are renamed as cid,cname and ccity respectively.

2. Additional relational operations

2.1 Set intersection operation

- ✓ Suppose r and s are two relations, then set intersection operation contains all tuples that are in both r and s.
- ✓ Let r and s are two relation having same arity and attributes of r and s are compatible then their intersection r∩ s define as r∩ s={t|t∈r and t∈s}

Example

Find all customer who have both loan and account

$\prod_{customer\_name}$(borrower) ∩ $\prod_{customer\_name}$ (depositor)

customer_name
nishant
gita

2.2 Division

- ✓ The division operator takes two relations and builds another relation consisting of values of an attribute of one relation that matches all the values in another relation
- ✓ Division operator r÷s or r/s can be applied if and only if, Attributes of s is proper subset of Attributes of r.
- ✓ The relation returned by division operator will have attributes = (All attributes of r – All Attributes of s)

Example 1:

K	X	Y
1	A	2
1	B	4
2	A	2
3	B	4
4	B	4
3	A	2

Relation r

X	Y
A	2
B	4

Relation s

r÷ s:

K
1
3

Example 2:

Consider the following relation

subject

subject_name	course_name
DBMS	CMP
C++	ELX
C++	CMP
OS	CMP

course

course_name
CMP
ELX

Find the name of subject taught in all course

subject÷ course

subject_name
C++

2.3 Assignment Operation

- ✓ The assignment operation provides convenient way to express complex query.
- ✓ The assignment operation denoted by ←, works like assignment in programming language.
- ✓ The evaluation of an assignment does not result any relation being displayed to the user. But the result of the expression to the right of the ← is assigned to the relation variable.
- ✓ This relation variable may used in subsequent expressions.
- ✓ With the assignment expression, a query can be written as a sequential program consisting a series of assignments followed by an expression whose value is displayed as the result of the query.

Example: Find all customer who taken loan from bank as well as has bank account.

temp1$\leftarrow \prod_{customer_name}$(borrower)

temp2$\leftarrow \prod_{customer_name}$(depositor)

result$\leftarrow$ temp1 $\cap$ temp2

2.4 Join Operation

- ✓ A Join operation combines related tuples from different relations, if and only if a given join condition is satisfied.
- ✓ It is denoted by ⋈.
- ✓ Join operation is essentially a Cartesian product followed by a selection criterion.
- ✓ In its simplest form, the join operation is just the cross product of two relations which produce large result size but using this operation, one record from relation **R** and on record from Relation **S** can be combined together to form the result if the combination satisfies the join condition.
- ✓ The join condition can be =, , ≤, ≥, ≠

Various forms of join operation are:

1. Equi Join
2. Natural Join
3. Outer Join
    i. Left Outer Join
    ii. Right Outer Join
    iii. Full Outer Join

Before discussing various joins let us consider the following relations

Student

stu_id	sname	address
1	Anish	Kathmandu
2	Rita	Lalitpur
3	Krishna	Bhaktapur
4	Nitu	Butwal

Course

stu_id	cname	fee
1	Java	25000
3	Python	22000
4	PHP	18000
5	MERN stack	30000

1. Equi join

An equijoin is an operation that combines two relations based on the equality of values in specified attributes.

It is denoted by the symbol ⋈ and is defined as follows:

R ⋈ <condition> S

Where R and S are the relations to be joined, and <condition> represents the equality condition on the common attribute(**S**).

student ⋈ Student.stu_id = Course.stu_id course

stu_id	sname	address	cname	fee
1	Anish	Kathmandu	Java	25000
3	Krishna	Bhaktapur	Python	22000
4	Nitu	Butwal	PHP	18000

The resulting relation includes only the tuples that have matching "stu_id" values in both the "Student" and "Course" relations.

2. Natural Join

- ✓ Natural join automatically matches and combines tuples from two relations based on the common attribute(s) with the same name.
- ✓ In example given below, the common attribute is "stu_id".
- ✓ Natural join does not use any comparison operator
- ✓ The name and type of the attribute must be same.
- ✓ It eliminates duplicate attributes in the result.
- ✓ It is denoted by ⋈

Student ⋈ course

stu_id	sname	address	cname	fee
1	Anish	Kathmandu	Java	25000
3	Krishna	Bhaktapur	Python	22000
4	Nitu	Butwal	PHP	18000

The resulting relation includes only the tuples that have matching attribute values with the same name ("stu_id" in this case).

3. Outer join

- ✓ It is an extension of natural join to deal with missing values of relation.
- ✓ It is used to retrieve all records from relation, even for those tuples with no matching value in the other relation based on the join condition.
- ✓ In such cases, it returns NULL as the value for the missing attributes.

It is further classified as:

i. Left Outer Join
ii. Right Outer Join
iii. Full Outer Join

i) Left outer join (⟕)

- ✓ Left outer join returns all tuples from the left relation and the matching tuples from the right relation.
- ✓ If there is no match in the right relation, NULL values are used for the attributes of the right relation in the resulting relation

Student ⟕ course

stu_id	sname	address	cname	fee
1	Anish	Kathmandu	Java	25000
2	Rita	Lalitpur	NULL	NULL
3	Krishna	Bhaktapur	python	22000
4	Nitu	Butwal	PHP	18000

The resulting relation includes all tuples from the left relation ("Student") and the matching tuples from the right relation ("Course"). The NULL values in the "cname" and "fee" columns indicate that there is no match for the corresponding tuples in the "Course" relation.

ii) Right Outer Join(⟖)

- ✓ Right outer join returns all tuples from the right relation and the matching tuples from the left relation.
- ✓ If there is no match in the left relation, NULL values are used for the attributes of the left relation in the resulting relation.

Student ⟖ Course

stu_id	sname	address	cname	fee
1	Anish	Kathmandu	Java	25000
3	Krishna	Bhaktapur	Python	22000
4	Nitu	Butwal	PHP	18000
5	NULL	NULL	MERN stack	30000

The resulting relation includes all tuples from the right relation ("Course") and the matching tuples from the left relation ("Student"). The NULL values in the "sname" and "address" columns indicate that there is no match for the corresponding tuples in the "Student" relation.

iii) Full outer join(⟗)

- ✓ Full outer join returns all tuples from both relations.
- ✓ If there is no match for a tuple in either relation, NULL values are used for the attributes of the relation that does not have a match.

Student ⟗ Course

stu_id	sname	address	cname	fee
1	Anish	Kathmandu	Java	25000
2	Rita	Lalitpur	NULL	NULL
3	Krishna	Bhaktapur	python	22000
4	Nitu	Butwal	PHP	18000
5	NULL	NULL	MERN stack	30000

The resulting relation includes all tuples from both the left relation ("Student") and the right relation ("Course"). The NULL values indicate that there is no match for the corresponding tuples in either relation.

3. Extended Relational- Algebra Operations

3.1 Generalized Projection

Generalized projection operation allows arithmetic and string functions in the projection list.

The generalized projection has the form

$\prod_{F1,F2,\ldots Fn}(E)$

where E is any relational algebra expression. Each F1,F2, . .Fn are arithmetic expression involving constants and attributes in the schema of E.

Example 1:

Suppose a relation

credit_info(customer_name, credit_limit,credit_balance)

Find how much more each person can spend.

$\prod_{customer\_name,\ credit\_limit - credit\_balance}(credit\_info)$

Example 2:

Suppose relation

employee(employee_id,ename,salary)

Find employee and their corresponding bonus, assume that bonus for each employee is 10% of his/her salary.

$\prod_{ename,salary*1.10}(employee)$

3.2 Aggregation

- ✓ The aggregate operation permits the use of aggregate functions such as min ,average etc. on set of values.
- ✓ Aggregation function takes a collection of values and returns a single value as a result.

Some aggregate functions are

avg: average value
min: minimum value
max: maximum value
sum: sum of values
count: number of value

- ✓ Aggregate operation in relational algebra denoted by the symbol g (i.e. $\mathcal{G}$ is the letter G in calligraphic font)

$$_{G1, G2, \ldots, Gn}\mathcal{G}_{F1(A1), F2(A2), \ldots, Fn(An)}(E)$$

- E is any relational-algebra expression
- $G_1, G_2 \ldots, G_n$ is a list of attributes on which to group (can be empty)
- Each F_i is an aggregate function
- Each A_i is an attribute name

Let us consider the following relation named Employee

emp_id	name	department	salary
1	Ramesh	Civil	55000
2	Prizma	Computer	65000
3	Riya	IT	52000
4	Narayan	Civil	25000
5	Nimesh	computer	5000

Find the average salary of employee

$$\mathcal{G}_{avg(salary)}(\text{Employee})$$

Find the minimum salary of employee

$$\mathcal{G}_{min(salary)}(\text{Employee})$$

Find the total salary paid by employee in each department

$$_{department}\mathcal{G}_{sum(salary)}(\text{Employee})$$

department	salary
Civil	80000
Computer	70000
IT	52000

Modification of database

Insertion, deletion and updating operations are responsible for database modification

Deletion

- ✓ A delete request is expressed similarly to a query, except instead of displaying tuples to the user, the selected tuples are removed from the database.
- ✓ Can delete only whole tuples; cannot delete values on only particular attributes
- ✓ A deletion is expressed in relational algebra by:

$$r \leftarrow r - E$$

where r is a relation and E is a relational algebra query.

Let us consider the following relation

Employee(employee_id,name,department,salary)

Example 1: Delete information of all employee from civil department

Employee ← Employee- $\sigma_{department="civil"}$ (Employee)

Example 2:

Delete all records of employee with salary in the range 25000 to 60000

$Employee \leftarrow Employee - \sigma_{(salary \geq 25000) \wedge (salary \leq 60000)}$ (Employee)

Insertion

To insert data into relation we can either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted.

In relational-algebra, an insertion is express by $r \leftarrow r \cup E$
where r is a relation and E is a relational algebra expression

Example:

Insert the information of employee whose employee id is 10, named "rikesh" working in civil department and having salary 50000

Employee ← Employee $\cup$ {(10,"rikesh", "civil",50000)}

Updating

Updating allow to change a value in a tuple without changing all values in tuple. In relational algebra, updating express by
$r \leftarrow \prod_{F1,F2,\ldots,Fn}(r)$

where each F_i is either

- ✓ the *i*th attribute of *r*, if the *i*th attribute is not updated, or,
- ✓ expression involving only constant and attributes of r, if the attribute is to be updated. It gives the new value for the attribute.

Example:

Increase salary of all employees by 5 %

$Employee \leftarrow \prod_{employee\_id,name,deparment,salary*1.05}(Employee)$

Increase the salary of employee by 20% if salary is less than 50000 and increase salary of remaining employees by 10%

$$Employee \leftarrow \prod_{employee\_id,\ name,,department,\ salary\ *\ 1.2}(\sigma_{\ salary\ <50000}\ (Employee)) \cup \prod_{employee\_id,name,department,salary*1.1}(\sigma_{salary \geq 50000}\ (Employee))$$

Increase salary of employees of civil department by 15%

$Employee \leftarrow \prod_{employee\_id,name,department,salary*1.15}(\sigma_{department=\ "civil}\ (Employee))$

$U\ (Employee - \sigma_{department=\ "civil}\ (Employee))$

Update an employee so that ram now shifted to computer department

$Employee \leftarrow \prod_{employee\_id,name,"computer",salary}(\sigma_{name=\ "ram"}(Employee))\ U$

$(Employee - \sigma_{name="ram"}(Employee))$

Note:

update operation will be in another form as well

$r \leftarrow \prod_{F1,F2,F3,\ldots Fn}(\ \sigma_p\ (r)\ )\ U\ (r - \sigma_p(r)\ )$

Database schema

Database schema is a logical design of a database and the database instance is a snapshot of the data in the database at a given instant in time.

The concept of a relation corresponds to the programming language notation of a variable While the concept of a relation schema corresponds to the programming language notation of type definition. In general a relation schema consists of list of attributes and their corresponding domains.

The concept of a **relation instance** corresponds to the programming –language notation of a value of a variable. The value of a given variable may change with time; similarly the contents of a relation instance may change with time as the relation is updated. In contrast, the schema of a relation does not generally change.

For example, the relation schema for relation customer is express as

Customer (customer_id, customer_name, customer_city)

We may also specify domains of attributes as

Customer (customer_id: integer, customer_name: string,customer_city:string)

The below figure shows the instance of relation

customer_id	customer_name	customer_city
1	ronit	Kathmandu
2	sita	Pokhara
3	nitu	Lalitpur

Let us consider university database example

Each course in a university may be offered multiple times, across different semesters, or even within a semester. We need a relation to describe each individual offering, or section, of the class.

The schema is section (course id, sec id, semester, year, building, room number, time slot id)

We need a relation to describe the association between instructors and the class sections that they teach. The relation schema to describe this association is

teaches (ID, course id, sec id, semester, year)

As we can imagine, there are many more relations maintained in a real university database.

Now, all the database schemas for university database can be listed as

instructor(id,name,dept_name,salary)
course(course_id,title,dept_name,credits)
department (dept_name, building, budget)
section (course_id, sec_id, semester, year, building, room_number, time_slot_id)
teaches (ID, course_id, sec_id, semester, year)
student (ID, name, dept_name, tot_cred)
advisor (s_id, i_id)
prereq(course_id,prereq_id)
takes (ID, course_id, sec_id, semester, year, grade)
classroom (building, room_number, capacity)
time_slot (time_slot_id, day, start_time, end_time)

Schema diagrams

A database schema, along with primary key and foreign key dependencies, can be depicted by schema diagrams. Figure given below shows the schema diagram for university organization. Each relation appears as a box, with the relation name at the top in blue, and the attributes listed inside the box. Primary key attributes are shown underlined. Foreign key dependencies appear as arrows from the foreign key attributes of the referencing relation to the primary key of the referenced relation.

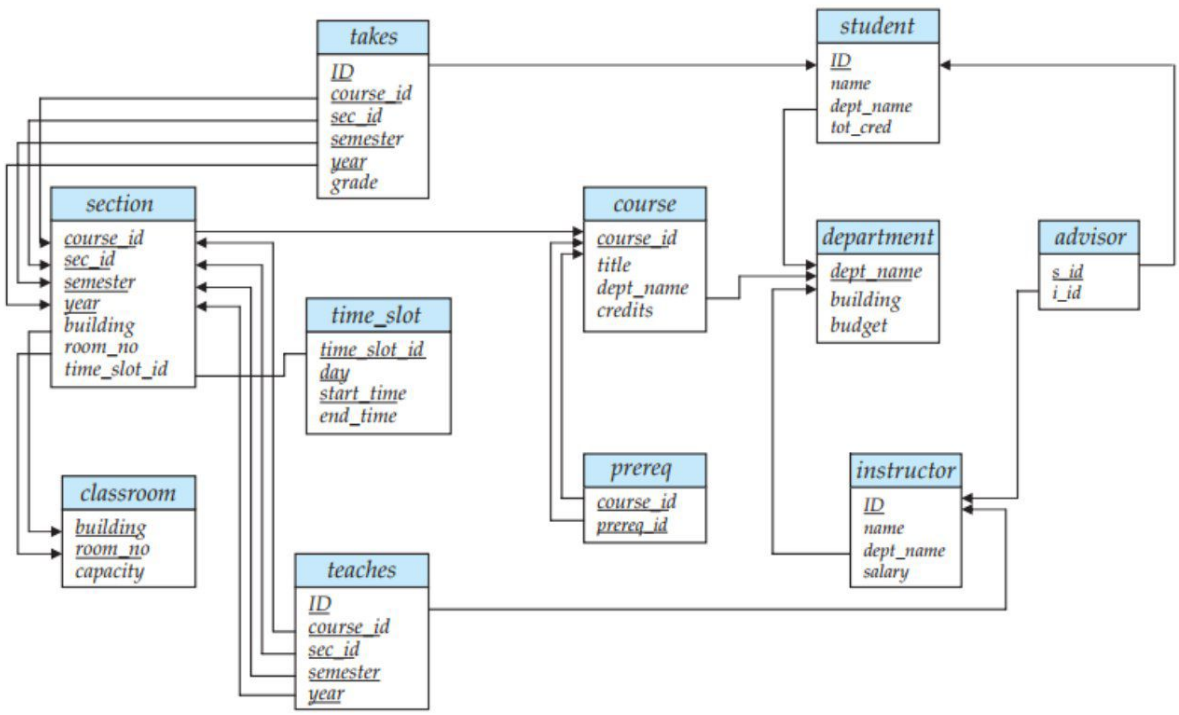


Figure:Schema diagram for the university database.

Assignment

Consider the following relations for a database that keeps track of student enrollment in courses and the books adopted for each course: **[PU:2017 spring]**

STUDENT(SSN,Name,Major,Bdate)
COURSE(Course#,Cname,Dept)
ENROLL(SSN,Course#,Quarter,Grade)
BOOK_ADOPTION(Course#,Quarter,Book_ISBN)
TEXT(Book_ISBN,Book_Title,Publisher,Author)

Draw a relational schema diagram specifying the foreign keys for this schema.

Data Dictionary storage

A relational database system needs to maintain data about the relations, such as the schema of the relations. In general, such "data about data" is referred to as metadata. Relational schemas and other metadata about relations are stored in a structure called the **data dictionary** or system catalog.

Among the types of information that the system must store are these:

- ✓ Names of the relations.
- ✓ Names of the attributes of each relation.
- ✓ Domains and lengths of attributes.
- ✓ Names of views defined on the database, and definitions of those views.
- ✓ Integrity constraints (for example, key constraints).

In addition, many systems keep the following data on users of the system:

- ✓ Names of authorized users.
- ✓ Authorization and accounting information about users.
- ✓ Passwords or other information used to authenticate users.

Further, the database may store statistical and descriptive data about the relations, such as:

- ✓ Number of tuples in each relation.
- ✓ Method of storage for each relation

The data dictionary may also note the storage organization (sequential, hash, or heap) of relations, and the location where each relation is stored:

- ✓ If relations are stored in operating system files, the dictionary would note the names of the file (or files) containing each relation.
- ✓ If the database stores all relations in a single file, the dictionary may note the blocks containing records of each relation in a data structure such as a linked list.

Assignment:

How relational algebra is different from relational calculus? Define Tuple Relational Calculus and Domain Relational Calculus.

(Refer: Text Book "Database system concept" by Abraham Silberschatz)

RELATIONAL ALGREBRA OLD QUESTIONS SOLUTION

1. Consider the following relational database, where primary keys are underlined.

   employee(person_name,street,city)

   works(person_name,company_name,salary)

   company(company_name,city)

   manages(person_name,manager_name)

   Given an expression in the relational algebra to express each of the following queries.

   i)Find the name of all employees who work for first bank corporation.

   $\prod_{person\_name} (\sigma_{company\_name="first\ bank\ corporation"}(works))$

   ii)Find the name and cities of residence of all employees who work for first bank corporation.

   $\prod_{person\_name,city} (\sigma_{company\_name="first\ bank\ corporation"}(employee \bowtie works))$

   iii)Find the name, street address and city of residence of all employees who work for first bank corporation and earn more than $10000 per annum.

   $\prod_{person\_name,street,city} (\sigma_{company\_name="first\ bank\ corporation" \wedge salary>10000}(employee \bowtie works))$

   (Here we assume attribute named salary represent Annual Salary)

   iv)Find the name of all employees in this database who lives in same city as the company for which they work.

   $\prod_{person\_name}(employee \bowtie works \bowtie company)$

   v)Find the name of all employees who live in the same city and on the same street as do their managers.

   $\prod_{person\_name}((employee \bowtie manages) \bowtie_{(manages.manager\_name=employee2.person\_name \wedge employee.street=employee2.street \wedge employee.city=employee2.city)} (\rho_{employee2}(employee)))$

   vi)Find the name of all employees in this database who do not work for first bank corporation

   $\prod_{person\_name} (\sigma_{company\_name \neq "first\ bank\ corporation"}(works))$

   vii) Assume that companies may located in several cities. Find all companies located in every city in which small bank corporation is located.

   $\prod_{company\_name,city}(company) \div \prod_{city} (\sigma_{company\_name ="small\ bank\ corporation"}(company))$

2. Consider the following relational database.

Students(RollNo, StudentName, Address, Semester)

Teachers(TeacherID, TeacherName,CourseID,Salary,Department)

Courses(CourseID,RollNo,CourseTitle,Semester)

Write relational Algebra Expressions for the following requests.

i) Find the name of students of 4th semester and studying "Operating System"

$\prod_{StudentName}(\sigma_{Semester= "4th" \wedge CourseTitle="Operating System"}(Students \bowtie Courses))$

ii) Find the name of teacher who teaches subject "DBMS" to "Arten Khadka"

$\prod_{TeacherName}(\sigma_{CourseTitle="DBMS" \wedge StudentName="Arten Khadka"}((Students \bowtie Courses) \bowtie Teachers))$

iii) Delete Record of 2nd semester students of Account Department

$Students \leftarrow \prod_{RollNo, StudentName, Address, Semester}(Students)$

$- \prod_{RollNo, StudentName, Address, Semester}(\sigma_{Department="account" \wedge Semester=2nd}((Students \bowtie Courses) \bowtie Teachers))$

iv) Increase salary of "Bhaskar Bhatta" by 6%

$Teachers \leftarrow \prod_{TeacherID, TeacherName,CourseID,Salary*1.06,Department}(\sigma_{TeacherName= "Bhaskar Bhatta"}(Teachers)) \cup (Teachers - \sigma_{TeacherName= "Bhaskar Bhatta"}(Teachers))$

3. Consider the following schema

SUPPLIER(Sid,S_name,S_addr)

PARTS(Pid,p_name,color)

CATALOG(sid,pid,cost)

Now answer the following queries in Relational Algebra.

i) Find the name of all supplier who supply yellow parts

$\prod_{S\_name}(\sigma_{color='yellow'}(SUPPLIER \bowtie CATALOG \bowtie PARTS))$

ii) Find the name of suppliers who supply Both blue and black parts.

$\prod_{S\_name,color}(SUPPLIER \bowtie (CATALOG \bowtie PARTS)) \div \prod_{color}(\sigma_{color='blue' \vee color='black'}(PARTS))$

iii) Find the name of suppliers who supply all parts.

$\prod_{S\_name,Pid}(SUPPLIER \bowtie CATALOG) \div \prod_{Pid}(PARTS)$

4. **Consider the relational database** **[PU:2010 spring]**

Employee(Empname,street,city)

Works(Empname,post,cmpname,salary)

Company(cmpname,location)

Write relational algebraic expression for

i) An employee named John is promoted from Assistant manager to manager.

$Works \leftarrow \prod_{Empname,"Manager",cmpname,salary}(\sigma_{Empname="John" \wedge post="Assistant\ Manager"}(Works))$
$\cup\ (Works - \sigma_{Empname="John" \wedge post="Assistant\ Manager"}(Works))$

ii) Update the relation company so that all companies located in Biratnagar is shifted to Kathmandu.

$Company \leftarrow \prod_{cmpname,"Kathmandu"}(\sigma_{location="Biratnagar"}(Company))$
$\cup\ (Company - \sigma_{location="Biratnagar"}(Company))$

iii) Remove all the records of employee who lives in pokhara

$Employee \leftarrow Employee - \sigma_{city="Pokhara"}(Employee)$

5. Consider the following schema

customer(cus_id,cus_name,cus_phno)
employee(cus_id,emp_id,emp_name,emp_add)
works(branch_id,salary,cus_id)
branch(branch_id,branch_name)
Write relational algebra notations for the following queries for the given schema.
[PU:2011 spring]

i) select name of all employees.

$\prod_{emp\_name}(employee)$

ii) Give salary rise to 5% to all the employee

$works \leftarrow \prod_{branch\_id,salary*1.05,cust\_id}(works)$

iii) List all branch names

$\prod_{branch\_name}(branch)$

iv) select name of all employees working for "manang" branch

$\prod_{emp\_name}(\sigma_{branch\_name="manang"}(branch \bowtie (works \bowtie employee)))$

v) Delete any record from work table

$works \leftarrow works - \sigma_{branch\_id=101}(works)$

This operation can be customized according to requirement.

vi) List the name and phno of all customers

$\prod_{cust\_name,cus\_phno}(customer)$

vii) select name of all employees deal with customer having id "201"

$\prod_{emp\_name}(\sigma_{cus\_id=201}(employee))$

viii) Delete all records from works table whose salary is less than 10000

$works \leftarrow works - \sigma_{salary<10000}(works)$

ix) Delete all records from works table

$works \leftarrow works - \prod_{branch\_id,salary,cus\_id}(works)$

6. By using the following schemas write relational algebraic expression and SQL statements. (underlined attributes represent primary key attributes)

EMPLOYEE(EMPNO,NAME,ADDRESS)
PROJECT(PNO,PNAME)
WORKON(EMPNO,PNO)
PART(PARTNO,PARTNAME,QTY_ON_HAND)
USE(EMP_NO,PNO,PARTNO,NUMBER) **[PU:2011 fall]**

i)Listing all employees details who are not working yet

$\prod_{EMPNO,NAME,ADDRESS}(Employee) - \prod_{EMPNO,NAME,ADDRESS}(Employee \bowtie WORKON)$

ii) Listing Part Name and Quantity on hand those were used in DBMS project

$\prod_{PARTNAME,QTY\_ON\_HAND}(PART \bowtie (USE \bowtie \sigma_{PNAME="DBMS"}(PROJECT)))$

OR

$\prod_{PARTNAME,QTY\_ON\_HAND}(\sigma_{PNAME="DBMS"}(PART \bowtie (USE \bowtie PROJECT)))$

iii) List the name of projects that are used by employee from Kathmandu

$\prod_{PNAME}(\sigma_{ADDRESS="Kathmandu"}((EMPLOYEE \bowtie WORKON) \bowtie PROJECT))$

7. Consider the following relations for order processing database application in a company.

CUSTOMER(Cust#,Cname,City)

ORDER(Order#,Odate,Cust#,ord_Amt)

ORDER_ITEM(Order#,Item#,Qty)

ITEM(Item#,Unit_price)

SHIPMENT(Order#,Warehouse#,Ship_date)

WAREHOUSE(Warehouse#,City)

Answer the following queries in relational algebra. **[PU:2012 spring]**

i) List the order# and ship_date for all orders shipped from Warehouse number "W2".

$\prod_{Order\#,Ship\_date}(\sigma_{warehouse\#="w2"}(SHIPMENT))$

ii) List the warehouse information for which the customer named 'JOSE Copez' was supplied his orders.

$\prod_{warehouse\#,city}(\sigma_{Cname="JOSE\ Copez"}(CUSTOMER \bowtie (ORDER \bowtie (SHIPMENT \bowtie WAREHOUSE))))$

iii) List the orders that were not shipped within 30 days of ordering.

$TIMELY\_SHIPPED \leftarrow \sigma_{Ship\_date \le Odate+30}(ORDER \bowtie SHIPMENT)$

$RESULT \leftarrow \prod_{Order\#}(ORDER) - \prod_{Order\#}(TIMELY\_SHIPPED)$

iv) List the order# for orders that were shipped from all warehouses in the network.

$\prod_{Order\#,\ Warehouse\#}(Shipment) \div \prod_{Warehouse\#}(\sigma_{City = "New\ York"}(Warehouse))$

8. consider the following database: [**PU:2012 fall]**

Student(sid,name,age)
Has(sid,cid)
College(cid,cname)
Write relational algebra expression to perform the following.

i) find the average age of student.

$\mathcal{G}_{avg(age)}$ (Student)

ii) Display the name of student who studies in "QWERT" college.

$\prod_{name}(\sigma_{cname="QWERT"} (Student \bowtie (Has \bowtie College)))$

iii) Insert a new student.

Student ←Student U {104,"Roshan",29}

iv) Delete record of "ASDFG" college from college relation

College←College-($\sigma_{cname="ASDFG"}$ (College))

v) Display name of students whose name begin from 'S.'

$\prod_{name}(\sigma_{name\ LIKE\ 'S\%'}$ (Student)

9. Consider the following relation R and S **[PU:2013 spring]**

R

Sid	SName	Marks(%)
S001	Hari	85
S002	Sita	78
S003	Bidur	85
S005	Vinod	68

S

Sid	SName	Marks(%)
S004	Sarita	76
S003	Bidur	85
S006	Shyam	75
S005	Vinod	68

i) Show the id and name of those students whose marks is less than 80 from relation schema R.(Write only relational schema)

$\prod_{Sid,SName}(\sigma_{Marks<80} (R))$

ii) Write the results.

RUS

Sid	SName	Marks(%)
S001	Hari	85
S002	Sita	78
S003	Bidur	85
S005	Vinod	68
S004	Sarita	76
S006	Shyam	75

R-S

Sid	SName	Marks(%)
S001	Hari	85
S002	Sita	78

$\Pi_{SName}(\sigma_{Marks=85}(S))$

SName
Bidur

10. Consider the relational database of figure below, where primary keys are underlined. Given an expression in the relational algebra to express each of the following queries. [PU:2014 spring]

Employee(person_name,street,city)
Works(person_name,bank_name,salary)
Bank(bank_name,city)
Manages(person_name,manager_name)

i) Find the total salary sum of all the banks.

$\mathcal{G}_{sum(salary)}$ (works)

ii) Modify the database so that Ram now lives in Kathmandu.

$Employee \leftarrow \Pi_{person\_name,street,"Kathmandu"}(\sigma_{person\_name="Ram"}(Employee))$
$U\ (Employee - \sigma_{person\_name="Ram"}(Employee))$

iii) Find the name, street address and cities of residence of all employees who work for Nepal world Bank corporation and earn more than $10,000 per annum.

$\prod_{person\_name,street,city}(\sigma_{bank\_name="Nepal\ World\ Bank\ Corporation"\wedge salary>10000}(Employee \bowtie Works))$

Here, we assume attribute named salary represent annual salary.

iv) Delete all tuples in work relation for employee of small bank corporation.

$Works \leftarrow Works - \sigma_{bank\_name="Small\ Bank\ Corporation"}(Works))$

11. Consider the following relational database of figure below, where the primary keys are underlined. Give an expression in the relational algebra to express the following queries

employee(person_name,street,city)

works(person_name,bank_name,salary)

company(bank_name,city)

manages(person_name,manager_name)

Given an expression in the relational algebra to express each of the following queries.

i) Find the name of all employees in this database who lives in same city as the company for which they work.

$\prod_{person\_name}(employee \bowtie works \bowtie company)$

ii) Give all employees of First Bank Corporation a 10 percent salary rise

$Works \leftarrow \prod_{person\_name,bank\_name,salary*1.1}(\sigma_{bank\_name="First\ Bank\ corporation"}(Works))$
$\cup\ (Works - \sigma_{bank\_name="First\ Bank\ Corporation"}(Works))$

iii) Modify the database so that Hari now lives in Biratnagar

$Employee \leftarrow \prod_{person\_name,street,"Biratnagar"}(\sigma_{person\_name="Hari"}(Employee))$
$\cup\ (Employee - \sigma_{person\_name="Hari"}(Employee))$

iv) Delete all tuples in work relation for employee of First Bank Corporation.

$Works \leftarrow Works - (\sigma_{bank\_name="First\ Bank\ Corporation"}(Works))$

12. Consider the following relational database of figure below, Where primary keys are underlined. Given an expression in the relational algebra to express each of the following queries. [PU:2015 fall]

employee(person_name,street,city)
works(person_name,bank_name,salary)
bank(bank_name,city)
manages(person_name,manager_name)

i) Find the name of all employees who work for Nepal Rastra Bank and Salary greater than $10000.

$\Pi_{person\_name}(\sigma_{bank\_name="Nepal\ Rastra\ Bank" \wedge salary > 10000}(employee \bowtie works))$

ii) Find the name and cities of residence of all employees who work for Nepal Rastra Bank

$\Pi_{person\_name,city}(\sigma_{bank\_name = "Nepal\ Rastra\ Bank"}(employee \bowtie works))$

iii) Find name, street address, and cities of residence of all employees who work for Nepal Rastra Bank Corporation and earn more than $10000 per annum.

$\Pi_{person\_name,street,city}(\sigma_{bank\_name = "Nepal\ Rastra\ Bank\ corporation" \wedge salary > 10000*12}(employee \bowtie works))$

(Here we assume that attributed named salary represents monthly salary)

iv) Delete all tuples in work relation for employee of Nepal Rastra Bank

$works \leftarrow works - \sigma_{bank\_name = "Nepal\ Rastra\ Bank"}(works))$

13. Consider the student registration database comprising of the schema below. **[PU:2016 fall]**

Student(CRN,Name,Gender,Address,Telephone)
Course(CourseID,CourseName,Hour,TeacherID)
Teacher(TeacherID,TeacherName,Office)
Registration(CRN,CourseID,Date)

i) Count the number of student registered subject in year 2015 gender wise.

$_{Gender}\mathcal{G}_{count(CRN)}(\sigma_{EXTRACT(YEAR\ FROM\ Date)=2015}(Student))$

ii) Show student details taught by teacher Ronit Shreshta.

$\prod_{CRN,Name,Gender,Address,Telephone}(\sigma_{TeacherName="Ronit\ Shreshta)}(((Student \bowtie Registration) \bowtie course) \bowtie Teacher))$

iii) Delete student information taught by teacher N.Mathema

$Student \leftarrow Student - \prod_{CRN,Name,Gender,Address,Telephone}(\sigma_{TeacherName="N.Mathema"}(((Student \bowtie Registration) \bowtie course) \bowtie Teacher))$

14. Consider the following schema of a relational database.
Branch(branch_name, branch_city, assests)
Account(account_number,branch_name,balance)
Customer(customer_id,customer_name,customer_street,customer_city)
Depositor(customer_id,account_number)
Loan(loan_number,branch_name,amount)
Borrower(customer_id,loan_number)

Write relational algebra for the following queries: **[PU:2016 spring]**

i) Find all customer either account or loan

$\prod_{customer\_name}(Customer \bowtie Depostior) \cup \prod_{customer\_name}(Customer \bowtie Borrower)$

ii) List the name and city of customer who have their account at the branch location "Butwal"

$\prod_{customer\_name,customer\_city}(\sigma_{branch\_city="Butwal"}(((Customer \bowtie Depositor) \bowtie Account) \bowtie Branch))$

iii) Delete all account in the branch "B1"

$Account \leftarrow Account - \sigma_{branch\text{-}name = "B1"}(Account)$

iv) Increase balance by 5% to all branches

$Account \leftarrow \prod_{account\_number,\ branch\_name,\ balance * 1.05}(Account)$

15. Consider the following schema: **[PU:2017 fall]**

employee(person_name,street,city)

works(person_name,company_name,salary)

company(company_name,city)

manages(person_name,manager_name)

Given an expression in relational algebra to express each of the following queries.

i) Find the name of all employees who earn more than their managers.

$\prod_{person\_name}$ ((works ⋈ manages) $\bowtie_{manages.manager\_name=e2.person\_name \wedge works.salary>e2.salary}$ (ρ_{e2} (works)))

In SQL this will works

```
SELECT works.person_name
FROM works NATURAL JOIN manages
JOIN works AS e2 ON manages.manager_name = e2.person_name
AND works.salary > e2.salary;
```

ii) Find the name of all employees who live in the same city and on the same street as their managers.

$\prod_{person\_name}$ ((employee ⋈ manages) $\bowtie_{(manages.manager\_name=employee2.person\_name \wedge employee.street=employee2.street \wedge employee.city=employee2.city)}$ ($\rho_{employee2}$ (employee)))

iii) Find the name of all employees with database that do not work for "NBL company".

$\prod_{person\_name}$($\sigma_{company\_name\neq"NBL\ company"}$ (works))

iv) Find the name of all employees in the database who earn more than top earner at "NBL company in the database".

topearner← $\mathcal{G}_{max(salary)}$ ($\sigma_{company\_name=" NBL\ company"}$(works))
result← $\prod_{personname}$($\sigma_{salary>topearner}$ (works))

16. Consider the following relational schema **[PU:2018 Fall]**

Department(**DepartmentID**,DepartmentName)
Designation(**DesignationID,**DesignationName,Salary)
Employee(**EmpID**,EmpName,Gender,DesignationID,DepartmentID)
Allowance(**AllowanceID**,AllowanceName)
Allowance_Details(DetailID,EmpID,AllowanceID,Amount)

Write the relational algebraic expression for the following task:

i) Find the number of employees department-wise.

$_{\text{DepartmentName}}\mathcal{G}_{\text{count(eid)}}$ (Employee⋈ Department)

ii) List the employee details whose salary is above 50000.

$\prod_{\text{EmpID,EmpName,Gender,DesignationID,DepartmentID}}(\sigma_{\text{(Salary+amount)>50000}}((\text{Employee}\bowtie \text{Designation}) \bowtie \text{Allowance}))$

iii) List the employee those who are getting house allowance.

$\prod_{\text{EmpName}}(\sigma_{\text{AllowanceName="houseallowance"}}((\text{Employee}\bowtie \text{Allowance\_Details}) \bowtie \text{Allowance}))$

17. Consider the relational database model
Users(<u>uid</u>,cname,city)
Items(<u>itemid</u>,itemname,city,quantity,price)
Manager(<u>mid</u>,aname,city)
Query(<u>queryno</u>,uid,mid,itemid,query_details,hitratio)
[PU:2018 spring]
Write the relational algebraic expressions for the following tasks.

i) Find all (queryno,uid) pairs for query with a hit ratio value greater than 500.

$\prod_{\text{queryno,uid}}(\sigma_{\text{hitratio>500}}(\text{Query}))$

ii) Find all item names of items in Pokhara ordered with query details as Pokhara details.

$\prod_{\text{itemname}}(\sigma_{\text{querydetails="Pokhara details"^city="Pokhara"}}(\text{Items}\bowtie \text{Query}))$

iii) Find item ids of items ordered through manager 35 but not through manager 27.

$\prod_{\text{itemid}}(\sigma_{\text{mid=35}}(\text{Query})) - \prod_{\text{itemid}}(\sigma_{\text{mid=27}}(\text{Query}))$

18. Using the following schema represent the following queries using Relational algebra.

PROJECT(Projectnum,ProjectName,ProjectType,ProjectManager)
EMPLOYEE(Empnum,Empname)
ASSIGNED_TO(Projectnum,Empnum)

[PU:2019 spring]

i) Find employee details working on project name starts with 'L'.

$\prod_{Empnum,Empname}(\sigma_{ProjectName\ Like\ 'L\%'}(\ (EMPLOYEE \bowtie ASSIGNED\_TO) \bowtie PROJECT)\ )$

ii) List all the employee details who are working under project manager "Roshan".

$\prod_{Empnum,Empname}(\sigma_{ProjectManager="Rohan"}((EMPLOYEE \bowtie ASSIGNED\_TO) \bowtie PROJECT)\ )$

iii) List the employes who are still not assigned with any project.

$\prod_{Empnum,Empname}(EMPLOYEE) - \prod_{Empnum,Empname}(EMPLOYEE \bowtie ASSIGNED\_TO)$

iv) List the employees who are working in more than one project.

$temp \leftarrow {}_{Empname}\mathcal{G}_{count\ (Projectnum)}(Employee \bowtie ASSIGNED\_TO)$

$result \leftarrow \prod_{Empname}((\sigma_{Projectnum>1}(temp))$

19. Write relational algebra for the following schemas. (Underlined indicates Primary) [PU:2020 spring]

Employee(**Emp_No**,Name,Address)
Project(**PNO**,Pname)
Workon(**Emp_No**,PNo)
Part(**Partno**,Part_name,Qty_on_hand)
Use(**Emp_No**,**PNO**,**Partno**,Number)

i)Listing all employees details who are not working yet.

$\prod_{Emp\_No,Name,Address}$ (Employee) - $\prod_{Emp\_No,Name,Address}$ (Employee⋈Workon)

ii) Listing Part Name and Quantity on hand those were used in DBMS project

$\prod_{Part\_name,Qty\_On\_hand}$ (Part⋈ (Use⋈$\sigma_{Pname="DBMS"}$(Project)))

iii) List the name of projects that are used by employee from London

$\prod_{Pname}$($\sigma_{Address="London"}$((Employee⋈Workon)⋈ Project))

iv) Modify the database so that Jones now live in USA.

Employee← $\prod_{Emp\_No,Name,"USA"}$($\sigma_{Name="Jones"}$(Employee)) U (Employee-$\sigma_{Name="Jones"}$(Employee))

v) Update address of an employee 'Japan' to 'USA'

Employee← $\prod_{Emp\_No,Name,"USA"}$($\sigma_{Address="Japan"}$(Employee)) U (Employee-$\sigma_{Address="Japan"}$(Employee))

20. Consider the following schemas: **[PU:2021 spring]**

Sailors(sid,sname,rating,age)

Boats(bid,bname,color)

Reserves(sid,bid,day)

Write relational algebra expression for the following queries:

i) Find record of sailors who have reserved boat number 103(bid=103)

$\prod_{sid,sname,rating,age}$ (Sailors⋈ (Reserves⋈($\sigma_{bid=103}$(Boat)))

OR

$\prod_{sid,sname,rating,age}$ ($\sigma_{bid=103}$ (Sailors⋈ (Reserves ⋈ Boat)))

ii) Update the color of the boat, where bid is 104,into green.

Boat→$\prod_{bid,bname,"green"}$ ($\sigma_{bid=104}$(Boat)) U (Boat-$\sigma_{bid=104}$(Boat))

iii) Find the name of sailors who have reserved a red or green boat.

$\prod_{sname}$ (Sailors⋈ (Reserves⋈($\sigma_{color="red" \vee color="green"}$(Boat)))

iv) Find the name of sailors who have reserved boat number 103 on day 5.

$\prod_{sname}$ ($\sigma_{bid=103 \wedge day=5}$ (Sailors⋈ Reserves))

v) Find the name of sailors whose name is not 'Ram'.

$\prod_{sname}$ ($\sigma_{sname \neq "ram"}$(Sailors))

vi) Find the name of all boats.

$\prod_{bname}$ (Boats)

21. Suppose we have the following relation. **[PU:2022 fall]**

Employee(person_name,street,city)
Works(person_name,company_name,salary)
Company(company_name,city)

Write relational algebra for the following queries.

i) Find the name of all employees who live in 'Butwal' and whose salary is less than Rs.50,000

$\prod_{person\_name}$($\sigma_{city="Butwal" \wedge salary<50000}$ (Employee ⋈ Works))

ii) Find the name of all employees who work for "Nepal Bank Limited".

$\prod_{person\_name}$ ($\sigma_{company\_name="Nepal\ Bank\ Limited"}$(works))

iii) Find the name and cities of residence of all employees who work for "Global Bank"

$\prod_{person\_name,city}$ ($\sigma_{company\_name="Global\ Bank"}$(employee⋈works))

iv) Update the salary of all employees by 10%

Works←$\prod_{person\_name,company\_name,salary*1.1}$(Works)

22. Suppose we have the following relation
Employee(person_name,street,city)
Works(person_name,company_name,salary)
Company(company_name,city)
Write relational algebraic expressions for the following queries:

i)List the name and city of employee who work in "Pokhara" and have salary greater than Rs.50,000.

$\prod_{Employee.person\_name,Employee.city}$ ($\sigma_{Company.city="Pokhara"\wedge salary>50000}$ (Employee $\bowtie_{Employee.person\_name=works.person\_name}$ (works$\bowtie$company)))

ii)Find the names of all employees who work for "ABC bank".

$\prod_{person\_name}$ ($\sigma_{company\_name="ABC\ bank"}$(works))

iii)Delete all employee who come from "Chitwan".

Employee$\leftarrow$Employee-$\sigma_{city="Chitwan"}$(Employee)

iv)Increase salary of all employee by 15%

Works$\leftarrow\prod_{person_name,company_name,salary*1.15}$(Works)

UNIT 3

Structured Query Language

Introduction to SQL

- ✓ SQL is a standard database language used to access and manipulate data in databases.
- ✓ The language, Structured English Query Language (SEQUEL) was developed by IBM Corporation, Inc., to use Codd's model. SEQUEL later became SQL (still pronounced "sequel").
- ✓ SQL stands for Structured Query Language.
- ✓ In 1979, Relational Software, Inc. (now Oracle) introduced the first commercially available implementation of SQL. Today, SQL is accepted as the standard RDBMS language.
- ✓ By executing queries SQL can create, update, delete, and retrieve data in within a database management system like MySQL, Oracle, PostgreSQL, etc.
- ✓ Overall SQL is a query language that communicates with databases.

The SQL language has several parts,

Data Definition Language (DDL): SQL provides a set of commands to define and modify the structure of a database, including creating tables, modifying table structure, and dropping tables.

Data Manipulation Language (DML): SQL provides a set of commands to manipulate data within a database, including adding, modifying, and deleting data.

SQL provides a rich set of commands for querying a database to retrieve data, including the ability to filter, sort, group, and join data from multiple tables.

Transaction Control: SQL supports transaction processing, which allows users to group a set of database operations into a single transaction that can be rolled back in case of failure.

Data Integrity: SQL includes features to enforce data integrity, such as the ability to specify constraints on the values that can be inserted or updated in a table, and to enforce referential integrity between tables.

User Access Control: SQL provides mechanisms to control user access to a database, including the ability to grant and revoke privileges to perform certain operations on the database.

Data types in SQL

char(n) : Fixed length character string, with user-specified length n.

varchar(n) : Variable length character strings, with user-specified maximum length n.

int : Integer (a finite subset of the integers that is machine-dependent).

smallint: Small integer (a machine-dependent subset of the integer domain type).

numeric(p,d): Fixed point number, with user-specified precision of p digits, with d digits to the right of decimal point. (ex., numeric(3,1), allows 44.5 to be stores exactly, but not 444.5 or 0.32)

real, double precision: Floating point and double-precision floating point numbers, with machine-dependent precision.

float(n): Floating point number, with user-specified precision of at least n digits.

Date and Time types in SQL

date: A calendar date containing a (four-digit) year, month, and day of the month.

time: The time of day, in hours, minutes and seconds.

timestamp: A combination of date and time

Date and time values can be specified like this:

date: '2001-04-25'
time:'09:30:00'
timestamp '2001-04-25 10:29:01.45'

DDL and DML statements in SQL

Data Definition Language (DDL)

- ✓ The Data Definition Language is made up of SQL commands that can be used to design the database structure.
- ✓ DDL refers to a set of SQL instructions for creating, modifying, and deleting database structures, but not data
- ✓ Popular DDL commands are: CREATE, DROP, ALTER and TRUNCATE.

CREATE: The database or its objects are created with this command (like table, views, store procedure, and triggers).

- ✓ A database is a systematic collection of data. To store data in a well-structured manner, the first step with SQL is to establish a database. To build a new database in SQL, use the CREATE DATABASE statement.
  Syntax: CREATE DATABASE db_name;

  Example:
  CREATE DATABASE student_db;
  The above example will create a database named student_db;

- ✓ We've already learned how to create databases. To save the information, we'll need a table. In SQL, the CREATE TABLE statement is used to make a table. A table is made up of rows and columns, as we all know. As a result, while constructing tables, we must give SQL all relevant information, such as the names of the columns, the type of data to be stored in the columns, the data size, and so on.

  Syntax:

  CREATE TABLE table_name(

  column1 data_type1,
  column2 data_type2,
  column3 data_type3,
  column4 data_type4,
  …..
  );
  Example:

```
CREATE TABLE student_info(
sid int,
name  varchar(30),
program varchar(30),
roll int);
```

The above command will create the table schema that look like:

sid	name	program	roll

DROP

- ✓ The DROP statement deletes existing database objects such as tables and views.

  For dropping table

  Syntax: DROP TABLE table_name;
  Example: DROP TABLE student_info;

  For dropping database

  Syntax: DROP DATABASE db_name;
  Example: DROP DATABASE student_db;

ALTER

- ✓ The ALTER statement in SQL is used to make changes to the structure of existing database objects. It allows you to modify tables, views, and other database elements.
  Example:
  - In an existing table, this command is used to add, delete or edit columns.
  - It can also be used to create and remove constraints from a table that already exists.

To add Column in table

Syntax:

ALTER TABLE table_name

ADD column_name datatype;

Example:

ALTER TABLE student_info;

ADD address varchar(30);

To remove existing column from table

Syntax:

ALTER TABLE table_name
DROP COLUMN column_name;

Example:
ALTER TABLE student_info
DROP COLUMN roll;

To rename column of table

Syntax:

ALTER TABLE *table_name*
CHANGE COLUMN *old_name new_name datatype*;

Example:
ALTER TABLE student_info
CHANGE COLUMN address location varchar(30);

(Note: This syntax is for MariaDB and may vary upon different DBMS)

To modify data type of column

Syntax:

ALTER TABLE table_name
MODIFY column_name datatype;

Example:

ALTER TABLE student_info

MODIFY program char(20);

TRUNCATE

- ✓ This statement deletes all the rows from the table.
- ✓ This is different from the DROP command, the DROP command deletes the entire table along with the table schema, however TRUNCATE just deletes all the rows and leaves an empty table.

Syntax:

TRUNCATE TABLE table_name;

Example:

TRUNCATE TABLE student_info;

//Deletes all the rows from the table student_info.

By performing all the above operations, finally our table named student_info becomes

Sid	Name	program	location
int	varchar(30)	char(20)	varchar(30)

Data Manipulation Language (DML)

- ✓ The SQL commands that deal with manipulating data in a database are classified as DML (Data Manipulation Language)
- ✓ The popular commands that come under DML are INSERT,UPDATE,DELETE,SELECT

INSERT

- ✓ This command is used to insert records in a table.
- ✓ When you are not inserting the data for all the columns and leaving some columns empty. In that case specify the column name and corresponding value. The non selected field will have NULL value inserted upon execution of the given query.

Syntax:

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

Example:

```
INSERT INTO student_info(sid,name,location)
VALUES(1,'Hari','Pokhara');
```

- ✓ When inserting the data for all the columns. No need to specify column name.

Syntax:

```
INSERT INTO table_name

VALUES (value1, value2, value3, ...);
```

Example:

```
INSERT INTO student_info

VALUES(2,'Rita','Computer','Butwal');
```

UPDATE

- ✓ In SQL, the UPDATE statement is used to update data in an existing database table.

Syntax:

```
UPDATE table_name

SET column1 = value1, column2 = value2,...

WHERE condition;
```

Example:

```
UPDATE student_info
SET location='kathmandu'
WHERE sid=2;
```

DELETE

- ✓ DELETE statement is used to delete records from a table.
- ✓ Depending on the condition we set in the WHERE clause, we can delete a single record or numerous records.

Syntax:

DELETE FROM table_name

WHERE condition;

Example1:

```
DELETE FROM student_info
WHERE location='kathmandu';
```

//Delete records of student from table named student_info whose location is Kathmandu

Example 2:

```
DELETE FROM  student_info;
```

//Delete all records from table named student_info;

SELECT

- ✓ SELECT command fetches the records from the specified table that matches the given condition, if no condition is provided, it fetches all the records from the table.

Syntax:

SELECT column1, column2, ...

FROM table_name;

Here, column1, column2, ... are the column names of the table we want to select data from.

- ✓ If we want to apply conditions while selecting the data then syntax becomes

SELECT column1, column2, ...

FROM table_name

WHERE condition;

- ✓ If we want to select all the columns and rows available in the table, use the following syntax:

  SELECT * FROM table_name;

  Example:

  SELECT * FROM student_info;

  //Displays all the information of students from table named student_info

  SELECT name,program

  FROM student_info

  WHERE location='pokhara';

  //Displays name and program of students from table named student_info whose location is 'pokhara'

Note: Some authors grouped SELECT command as DQL(Data Query Language)

SQL constraints

- SQL constraints are used to specify rules for the data in a table.
- Constraints are used to limit the type of data that can go into a table.
- This ensures the accuracy and reliability of the data in the table. If there is any violation between the constraint and the data action, the action is aborted.

The following constraints are commonly used in SQL:

NOT NULL - Ensures that a column cannot have a NULL value

UNIQUE - Ensures that all values in a column are different

PRIMARY KEY - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table

FOREIGN KEY - Prevents actions that would destroy links between tables

CHECK - Ensures that the values in a column satisfies a specific condition

DEFAULT - Sets a default value for a column if no value is specified

NOT NULL

- ✓ By default, a column can hold NULL values.
- ✓ The NOT NULL constraint enforces a column to NOT accept NULL values.
- ✓ This enforces a field to always contain a value, which means that you cannot insert a new record without adding a value to this field.

❖ **create table with NOT NULL constraint**

Example:

```
CREATE TABLE Colleges (
college_id INT NOT NULL,
college_code VARCHAR(20),
college_name VARCHAR(50)
);
```

❖ **Add the NOT NULL constraint to a column in an existing table**

Example:

```
ALTER TABLE Colleges
MODIFY COLUMN college_id INT NOT NULL;
```

❖ **Remove NOT NULL Constraint**

Example:

```
ALTER TABLE Colleges
MODIFY college_id INT;
```

UNIQUE

- ✓ The UNIQUE constraint ensures that all values in a column are different.

❖ **Create a table with unique constraint**

Example

```
CREATE TABLE Colleges (
college_id INT NOT NULL UNIQUE,
college_code VARCHAR(20) UNIQUE,
college_name VARCHAR(50)
);
```

❖ **Add the UNIQUE constraint to an existing column**

For single column

Example

```
ALTER TABLE Colleges
ADD UNIQUE (college_id);
```

For multiple columns

Example

```
ALTER TABLE Colleges
ADD UNIQUE Unique_College (college_id, college_code);
```

- ✓ Here, the SQL command adds the UNIQUE constraint to college_id and college_code columns in the existing Colleges table.
- ✓ Also, Unique_College is a name given to the UNIQUE constraint defined for college_id and college_code columns.

❖ **DROP a UNIQUE Constraint**

Example

```
ALTER TABLE Colleges
DROP INDEX Unique_College;
```

PRIMARY KEY

- ✓ The PRIMARY KEY constraint uniquely identifies each record in a table.
- ✓ Primary keys must contain UNIQUE values, and cannot contain NULL values.

❖ **Create table with PRIMARY KEY constraint**

Syntax:

```
CREATE TABLE table_name (
 column1 data_type,
  ..............................,
  [CONSTRAINT constraint_name] PRIMARY KEY (column1)
);
```

Example

```
CREATE TABLE Colleges (
  college_id INT,
  college_code VARCHAR(20) ,
  college_name VARCHAR(50),
CONSTRAINT CollegePK  PRIMARY KEY (college_id)
);
```

//Create Colleges table with primary key college_id

❖ **Add the PRIMARY KEY constraint to a column in an existing table**

Example

```
ALTER TABLE Colleges
ADD CONSTRAINT CollegePK PRIMARY KEY (college_id);
```

❖ **DROP a PRIMARY KEY Constraint**

Example

```
ALTER TABLE Colleges
DROP PRIMARY KEY;
```

DEFAULT

- ✓ the DEFAULT constraint is used to set a default value if we try to insert an empty value into a column.
- ✓ However if the user provides value then the particular value will be stored.

❖ **Default constraint while creating table**

The following example set default value of college_country column to 'Nepal'

Example:

```
CREATE TABLE Colleges (
  college_id INT PRIMARY KEY,
  college_code VARCHAR(20),
  college_country VARCHAR(20) DEFAULT 'Nepal'
);
```

❖ **Add the DEFAULT constraint to an existing column**

Example:

```
ALTER TABLE Colleges
ALTER college_country SET DEFAULT 'Nepal';
```

❖ **Remove DEFAULT Constraint**

Example:

```
ALTER TABLE Colleges
ALTER college_country DROP DEFAULT;
```

CHECK

- ✓ The CHECK constraint is used to limit the value range that can be placed in a column.
- ✓ If you define a CHECK constraint on a column it will allow only certain values for this column.

❖ **CHECK constraint while creating table**

Example:

Here we are Applying the CHECK constraint named amountCK the constraint makes sure that amount is greater than 0.

```
CREATE TABLE Orders (
order_id INT PRIMARY KEY,
amount INT,
CONSTRAINT amountCK CHECK (amount > 0)
);
```

❖ **Add CHECK Constraint in Existing Table**

Here we add CHECK constraint named amountCK the constraint makes sure that amount is greater than 0.

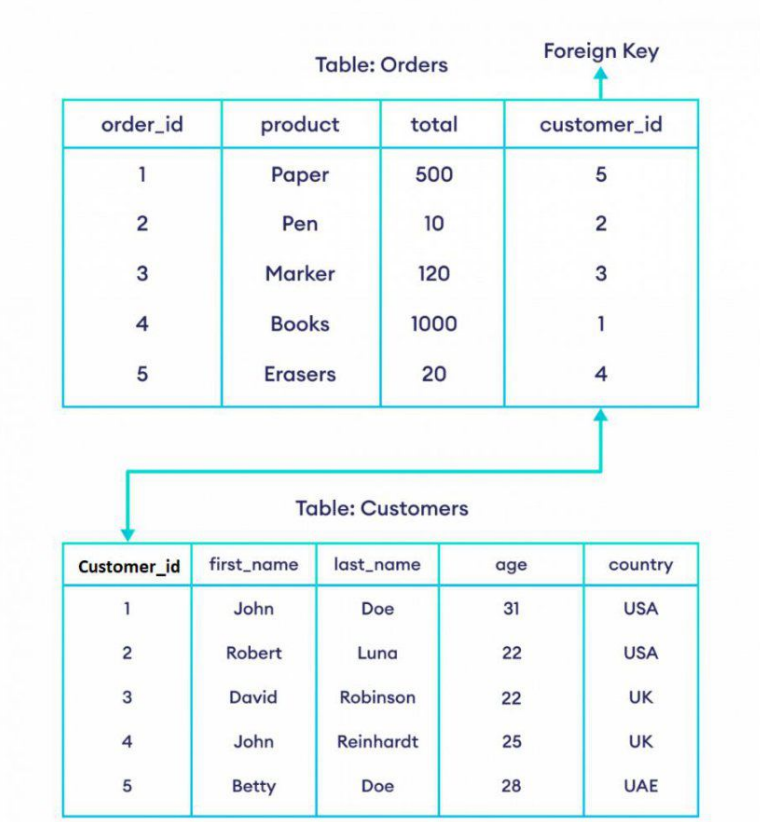
```
ALTER TABLE Orders
ADD CONSTRAINT amountCK CHECK (amount > 0);
```

❖ **Remove CHECK Constraint**

```
ALTER TABLE Orders
DROP CONSTRAINT amountCK;
```

FOREIGN KEY

The FOREIGN KEY constraint in SQL establishes a relationship between two tables by linking columns in one table to those in another.



- ✓ Here, the **customer_id** field in the Orders table is a FOREIGN KEY that references the **customer_id** field in the Customers table.
- ✓ This means that the value of the **customer_id** (of the Orders table) must be a value from the **customer_id** column (of the Customers table).

The syntax of the SQL FOREIGN KEY constraint is:

```
CREATE TABLE table_name (
   column1 data_type,
   column2 data_type,
   ……..,
 [CONSTRAINT CONSTRAINT_NAME] FOREIGN KEY (column_name)
   REFERENCES referenced_table_name (referenced_column_name)
);
```

Here,

- ✓ table_name is the name of the table where the FOREIGN KEY constraint is to be defined
- ✓ column_name is the name of the column where the FOREIGN KEY constraint is to be defined
- ✓ referenced_table_name and referenced_column_name are the names of the table and the column that the FOREIGN KEY constraint references
- ✓ [CONSTRAINT CONSTRAINT_NAME] is optional

Let us see with following example

- ✓ This table doesn't have a foreign key
- ✓ add foreign key to the customer_id field
- ✓ the foreign key references the id field of the Customers table

```
    -- this table doesn't have a foreign key

CREATE TABLE Customers (
 customer_id INT,
  first_name VARCHAR(40),
  last_name VARCHAR(40),
  age INT,
  country VARCHAR(10),
  CONSTRAINT CustomersPK PRIMARY KEY (customer_id)
);
-- add foreign key to the customer_id field
-- the foreign key references the id field of the Customers table
CREATE TABLE Orders (
  order_id INT,
  product VARCHAR(40),
  total INT,
  customer_id INT,
  CONSTRAINT OrdersPK PRIMARY KEY (order_id),
  CONSTRAINT CustomerOrdersFK FOREIGN KEY (customer_id) REFERENCES
Customers(customer_id)
);
```

Add the FOREIGN KEY constraint to an existing table

- ✓ add foreign key to the **customer_id** field of Orders the foreign key references the **customer_id** field of Customers

```
ALTER TABLE Orders
ADD FOREIGN KEY (customer_id) REFERENCES Customers(customer_id);
```

Remove a FOREIGN KEY Constraint

```
ALTER TABLE Orders
DROP FOREIGN KEY CustomerOrdersFK;
```

Operators in SQL

- An operator is a reserved word or a character that is used to query our database in a SQL expression.
- To query a database using operators, we use a WHERE clause.
- The operator manipulates the data and gives the result based on the operator's functionality.

Before starting with operators let us consider the following relation that we use to illustrate the examples of operators
Customers(customer_id,first_name,last_name,age,country);
Orders(order_id,product,total,customer_id);

Some operators available in SQL are:

Arithmetic Operators

- ✓ These operators are used to perform operations such as addition, multiplication, subtraction etc.
- ✓ Example. + (Addition) ,- (substraction), * (multiplication), / (division), % (modulus) etc.

```
UPDATE Orders
SET total=total+15;
```

This query increase the total amount of all records by 15.

Comparison Operators

- ✓ We can compare two values using comparison operators in SQL.
- ✓ These operators return either 1 (means true) or 0 (means false).
  Example:

Operator	Description
=	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<> , !=	Not equal to

```
SELECT *
FROM customers
WHERE age>20;
```

This query display the information of customers whose age is greater than 20

Logical operators

We can use logical operators to compare multiple SQL commands. These operators return either 1 (means true) or 0 (means false).
Some of the Logical operators available in SQL are,

AND
OR
NOT
BETWEEN
IN
LIKE

AND

- Returns the records if all the conditions separated by AND are TRUE

Example:

- ✓ *Display the first_name and last_name of all customers who live in 'Nepal' and have the last_name 'Paudel'*

```
SELECT first_name, last_name
FROM Customers
WHERE country = 'Nepal' AND last_name = 'Paudel';
```

OR

- Returns the records for which any of the conditions separated by OR is true

Example:

- ✓ *Display the first_name and last_name of all customers who either live in the 'Nepal' or have the last name 'Paudel'*

```
SELECT first_name, last_name
FROM Customers
WHERE country = 'Nepal' OR last_name = 'Paudel';
```

NOT

- Used to reverse the output of any logical operator

Example:

Display customers who don't live in the USA

```
SELECT first_name, last_name
FROM Customers
WHERE NOT country = 'USA';
```

Combining Multiple Operators

- ✓ It is also possible to combine multiple AND, OR and NOT operators in an SQL statement.

Display customers who live in either USA or UK and whose age is less than 26

```
SELECT  *
FROM Customers
WHERE (country = 'USA' OR country = 'UK') AND age < 26;
```

BETWEEN

- ❖ Returns the rows for which the value lies between the mentioned range.

Example:

Displays customers first_name,last_name, age from customers table whose age lies in the range 20-30

```
SELECT first_name,last_name,age
FROM Customers
WHERE age BETWEEN 20 AND 30;
```

Note: The **NOT BETWEEN** operator is used to exclude the rows that match the values in the range. It returns all the rows except the excluded rows.

IN

- ❖ Used to compare a value to a specified value in a list
- ❖ The IN operator selects values that match any one values given in the list

Example:

Select rows if the country lies in following list USA,UK,Nepal ,India,Pakistan

```
SELECT  *
FROM Customers
WHERE country IN ('USA', 'UK','Nepal','India','Pakistan');
```

Note: The **NOT IN** operator is used to exclude the rows that match values in the list. It returns all the rows except the excluded rows

LIKE

- ✓ The SQL LIKE operator is used with the WHERE clause to get a result set that matches the given string pattern.
- ✓ The pattern includes combination of wildcard characters and regular characters

Example :

```
SELECT  *
FROM Customers
WHERE last_name LIKE 'r%';
```

- ✓ Here, % (means zero or more characters) is a wildcard character.
- ✓ Hence, the SQL command selects customers whose last_name starts with r followed by zero or more characters after it.

Wildcard Characters

Symbol	Description	Example
%	Represents zero or more characters	bl% finds bl, black, blue,
_	Represents a single character	h_t finds hot, hat, and hit
[]	Represents any single character within the brackets	h[oa]t finds hot and hat, but not hit
^	Represents any character not in the brackets	h[^oa]t finds hit but not hot and hat
-	Represents any single character within the specified range	c[a-b]t finds cat and cbt

Here are some examples showing different LIKE operators with '%' and '_' wildcards:

LIKE Operator	Description
WHERE first_name LIKE 'a%'	Finds first_name that starts with "a"
WHERE first_name LIKE '%a'	Finds first_name that ends with "a"
WHERE first_name LIKE '%or%'	Finds first_name that have "or" in any position
WHERE first_name LIKE '_r%'	Finds first_name that have "r" in the second position
WHERE first_name LIKE 'a_ _%'	Finds first_name that starts with "a" and are at least 3 characters in length
WHERE first_name LIKE 'a%h'	Finds first_name that starts with "a" and ends with "h"

NULL values

- ✓ The term NULL in SQL is used to specify that a data value does not exist in the database.
- ✓ If a field in a table is optional, it is possible to insert a new record or update a record without adding a value to this field. Then, the field will be saved with a NULL value.

Some common reasons why a value may be NULL

- ❖ The value may not be provided during the data entry.
- ❖ The value is not yet known.

- ✓ It is not possible to test for NULL values with comparison operators, such as =, <, or <>.
- ✓ We will have to use the IS NULL and IS NOT NULL operators instead.

IS NULL

The IS NULL operator is used to test for empty values (NULL values).

Syntax:

```
SELECT column_names
FROM table_name
WHERE column_name IS NULL;
```

Example:

The following SQL lists first_name and last_name of all customers with a NULL value in the "country" field

```
SELECT first_name,last_name
FROM Customers
WHERE country IS NULL;
```

IS NOT NULL

The IS NOT NULL operator is used to test for non-empty values (NOT NULL values).

Syntax

```
SELECT column_names
FROM table_name
WHERE column_name IS NOT NULL;
```

The following SQL lists first_name,last_name and country of all customers with a value in the "country " field:

```
SELECT first_name,last_name,country
FROM Customers
WHERE country IS NOT NULL;
```

SQL SELECT DISTINCT

- ✓ The SQL SELECT DISTINCT statement retrieves distinct values from a database table.

Example 1:

Select the unique ages from the Customers table

```
SELECT DISTINCT age
FROM Customers;
```

Example 2:

- ✓ select the unique countries from the customers table

```
SELECT DISTINCT country
FROM Customers;
```

SQL DISTINCT With Multiple Columns

- ✓ We can also use SELECT DISTINCT with multiple columns.

Select rows if the first name and country of a customer is unique

```
SELECT DISTINCT country, first_name
FROM Customers;
```

Rename operation

- ✓ The AS command is used to rename a column or table with an alias.
- ✓ An alias only exists for the duration of the query.
- ✓ We can also use aliases with more than one column.

Example1:

```
SELECT first_name AS name
FROM Customers;
```

Here, the SQL command selects the first_name column of Customers. However, the column name will change to name in the result set.

Example2:

```
SELECT customer_id AS cid, first_name AS name
FROM Customers;
```

Here, the SQL command selects customer_id as cid and first_name as name

Sorting Results

- ✓ The ORDER BY keyword is used to sort the result-set in ascending or descending order.
- ✓ The ORDER BY keyword sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword.

ORDER BY Syntax

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1, column2, ... ASC|DESC;
```

Example:

The following SQL statement selects all customers from the "Customers" table, sorted by the "Country" column:

```
SELECT *
FROM Customers
ORDER BY country;
```

The following SQL statement selects all customers from the "Customers" table, sorted DESCENDING by the "Country" column.

```
SELECT * FROM
Customers
ORDER BY country DESC;
```

ORDER BY Several Columns

The following SQL statement selects all customers from the "Customers" table, sorted by the "country" and the "first_name" column. This means that it orders by Country, but if some rows have the same Country, it orders them by first_name:

Example

```
SELECT * FROM Customers

ORDER BY country,first_name;
```

The following SQL statement selects all customers from the "Customers" table, sorted ascending by the "Country" and descending by the "first_name" column:

Example

```
SELECT * FROM Customers
ORDER BY country ASC, first_name DESC;
```

Aggregate functions

An aggregate function in SQL returns one value after calculating multiple values of a column

Let us consider the following relation

Employee(employee_id,name,deparment,position,salary);

COUNT()

- ✓ The COUNT() function returns the number of rows that matches a specified criterion.

Syntax:

```
SELECT COUNT(column_name)
FROM table_name
WHERE condition;
```

Example:

```
SELECT COUNT( DISTINCT employee_id)
FROM Employee;
```

AVG()

- ✓ The AVG() function returns the average value of a numeric column.

Syntax:

```
SELECT AVG(column_name)
FROM table_name
WHERE condition;
```

Example:

```
SELECT AVG(salary)
FROM Employee;
```

SUM()

- ✓ The SUM() function returns the total sum of a numeric column.

Syntax:

```
SELECT SUM(column_name)
FROM table_name
WHERE condition;
```

Example:

```
SELECT SUM(salary)
FROM Employee;
```

MIN()

- ✓ The MIN() function returns the smallest value of the selected column

Syntax:

```
SELECT MIN(column_name)
FROM table_name
WHERE condition;
```

Example:

```
SELECT MIN(salary)
FROM Employee;
```

MAX()

- ✓ The MAX() function returns the largest value of the selected column

Syntax:

```
SELECT MAX(column_name)
FROM table_name
WHERE condition;
```

Example:

```
SELECT MAX(salary)
FROM Employee;
```

GROUP BY and HAVING clause

GROUP BY

- ✓ The GROUP BY statement groups rows that have the same values into summary rows, like "find the number of Employees in each department".
- ✓ The GROUP BY statement is often used with aggregate functions (COUNT(), MAX(), MIN(), SUM(), AVG()) to group the result-set by one or more columns.

Let us consider the following table

Table: Employee

eid	name	address	dept_name	salary
1	Hari	Butwal	Civil	62000
2	Shyam	Kathmandu	Computer	80000
3	Sita	Pokhara	Civil	90000
4	Ramesh	Kathmandu	IT	32000
5	Riya	Pokhara	Computer	76000
6	Dinesh	Kathmandu	Civil	94000
7	Srijana	Butwal	IT	68000

As an illustration consider the query "find the average salary of employee in each department"

```
SELECT dept_name, avg(salary) as average_salary
FROM Employee
GROUP BY dept_name;
```

dept_name	average_salary
Civil	82000
Computer	78000
IT	50000

HAVING clause

- ✓ SQL HAVING clause is similar to the WHERE clause; they are both used to filter rows in a table based on conditions.
- ✓ However, the HAVING clause was included in SQL to filter grouped rows instead of single rows.
- ✓ These rows are grouped together by the GROUP BY clause, so, the HAVING clause must always be followed by the GROUP BY clause.
- ✓ It can be used with aggregate functions, whereas the WHERE clause cannot.

As an illustration consider the query "find the name department where the average salary is greater than 60000"

```
SELECT dept_name, avg(salary) as average_salary
FROM employee
GROUP BY dept_name
HAVING avg(salary)>60000;
```

dept_name	average_salary
Civil	82000
Computer	78000

HAVING clause vs WHERE clause

HAVING clause	WHERE clause
HAVING Clause is used to filter record from the groups based on the specified condition.	WHERE Clause is used to filter the records from the table based on the specified condition.
HAVING Clause cannot be used without GROUP BY Clause	WHERE Clause can be used without GROUP BY Clause
HAVING Clause can contain aggregate function	WHERE Clause cannot contain aggregate function
HAVING Clause can only be used with SELECT statement	WHERE Clause can be used with SELECT, UPDATE, DELETE statement.
HAVING Clause implements in column operation	WHERE Clause implements in row operations

Let's take a look at another example, for following relations

```
Customers(customer_id,first_name,last_name,country)
Orders(order_id,product,amount,customer_id)
```

We can write a WHERE clause to filter out rows where the value of amount in the Orders table is less than 500:

```
SELECT customer_id, amount
FROM Orders
WHERE amount < 500;
```

But with the HAVING clause, we can use an aggregate function like SUM to calculate the sum of amounts in the Orders table and get the total order value of less than 500 for each customer:

```
SELECT customer_id, SUM(amount) AS total
FROM Orders
GROUP BY customer_id
HAVING SUM(amount) < 500;
```

Sub Query (Inner Query/Nested Query)

- ✓ A Subquery or Inner query or a Nested query is a query within another SQL query and embedded within clauses, most commonly in the WHERE clause.
- ✓ It is used to return data from a table, and this data will be used in the main query as a condition to further restrict the data to be retrieved.
- ✓ Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN etc.

Consider the following relation

```
employee(emp_id,name,age,department,salary)
```

Subqueries with SELECT statement

Display information of employee whose salary is greater than average salary of all employees

```
select *
from employee
where salary>(select avg(salary) from employee);
```

Display the information of employees whose salary is greater than 26000

```
select *
from employee
where emp_id in (select emp_id from employee where salary>26000);
```

Display information of employee whose salary is greater than at least one employee of IT department.

```
select *
from employee
where salary>some(select salary from employee where department ='IT ');
```

Display information of employee whose salary is greater than that of all employee of IT department.

```
select *
from employee
where salary>all(select salary from employee where department ='IT ');
```

Subqueries with UPDATE statement

Increase salary of employees by 10% whose salary is greater than the average salary of all employees.

```
update employee
 set salary=salary*1.1
 where salary> (select avg(salary) from employee);
```

Subqueries with DELETE statement

Delete the information of employees whose salary is less than average salary of all employees

```
delete from employee where salary < (select avg(salary) from employee);
```

Subqueries with INSERT statement

The INSERT statement uses the data returned from the subquery to insert into another table.

Suppose we want to make each employee board member of company whose department is 'finance' and age>55

Consider a table Boardmember with similar structure as Employee table. Now to copy the records of employee table whose department is 'finance' and age>55 into the Boardmember table, we can use the following syntax.

```
insert into Boardmember
select *
 from employee
where emp_id in (select emp_id from employee  where department='finance' and age>55);
```

Set operations

- ✓ In SQL, set operation is used to combine the two or more SQL SELECT statements.
- ✓ They allow the results of multiple SELECT queries to be combined into single result set.
- ✓ SQL set operators enable the comparison of rows from multiple tables or a combination of results from multiple queries
- ✓ There are certain rules which must be followed to perform operations using SET operators in SQL. Rules are as follows:
  - ☞ The number of columns in the SELECT statement on which you want to apply the SQL set operators must be the same.
  - ☞ The order of columns must be in the same order.
  - ☞ The selected columns must have the same data type.

UNION

- ✓ The SQL Union operation is used to combine the result of two or more SQL SELECT queries.
- ✓ The union operation eliminates the duplicate rows from its result set.

Syntax:

```
SELECT column_name(s) FROM table_1
UNION
SELECT column_name(s) FROM table_2;
```

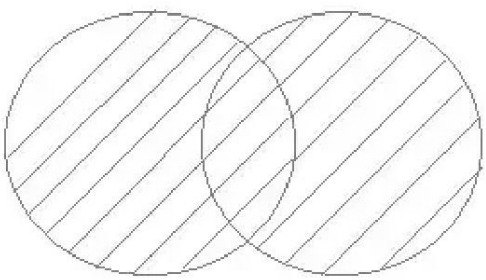


figure: pictorial representation of UNION operation

Let us consider the following two relations.

tbl_first

id	name
1	riya
2	durga
3	anish

tbl_second

Id	name
3	anish
4	roshan
5	rojina

Example:

```
SELECT * FROM tbl_first
UNION
SELECT * FROM tbl_second;
```

Output:

Id	name
1	riya
2	durga
3	anish
4	roshan
5	rojina

UNION ALL

- ✓ It is similar to Union but it also shows the duplicate rows.

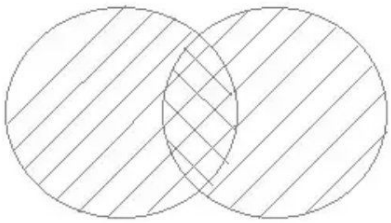


figure: Pictorial representation of UNION ALL operation

Let us consider the following two relations.

tbl_first

Id	name
1	riya
2	durga
3	anish

tbl_second

id	name
3	anish
4	roshan
5	rojina

Example:

```
SELECT * FROM tbl_first
UNION ALL
SELECT * FROM tbl_second;
```

Output

Id	name
1	riya
2	durga
3	anish
3	anish
4	roshan
5	rojina

INTERSECT

- ✓ The Intersect operation returns the common rows from both the SELECT statements.
- ✓ It has no duplicates and it arranges the data in ascending order by default.

Syntax:

```
SELECT column_name(s) FROM table_1
INTERSECT
SELECT column_name(s) FROM table_2;
```

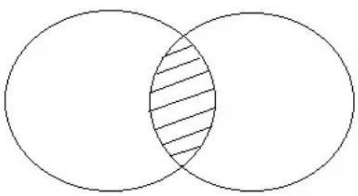


figure: pictorial representation of INTERSECT operation

Let us consider the following two relations.

tbl_first

id	name
1	riya
2	durga
3	anish

tbl_second

id	name
3	anish
4	roshan
5	rojina

Example:

```
SELECT * FROM tbl_first
INTERSECT
SELECT * FROM tbl_second;
```

Output

id	name
3	anish

EXCEPT

- ✓ EXCEPT operator is used to display the rows which are present in the first query but absent in the second query.
- ✓ It has no duplicates and data arranged in ascending order by default.

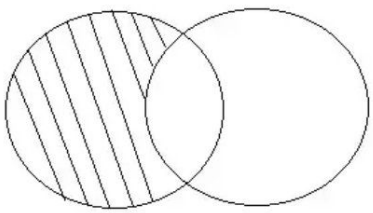


figure: pictorial representation of EXCEPT operation

Syntax:

```
SELECT column_name(s) FROM table_1
EXCEPT
SELECT column_name(s) FROM table_2;
```

Let us consider the following two relations.

tbl_first

id	name
1	riya
2	durga
3	anish

tbl_second

id	name
3	anish
4	roshan
5	rojina

Example:

```
SELECT * FROM tbl_first
EXCEPT
SELECT * FROM tbl_second;
```

Output

id	name
1	riya
2	durga

Join

- ✓ In SQL, a join is an operation that combines rows from two or more tables based on a related column between them.
- ✓ It allows you to retrieve data from multiple tables simultaneously by establishing a relationship between them.
- ✓ Joins are typically performed using the JOIN keyword in an SQL query.

There are different types of joins that can be used:

Before performing join operations, let us consider the following table

Employee

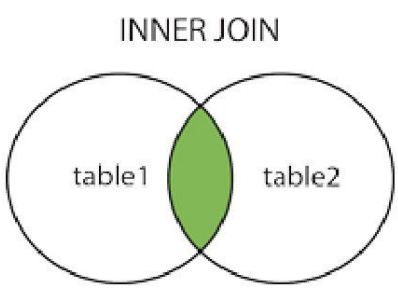
emp_id	emp_name	salary	dept_id
1	anish	25000	1
2	sita	55000	2
3	ronit	40000	4
4	riya	50000	5

Department

dept_id	dept_name
1	sales
2	marketing
3	finance
4	operations

Inner Join

- ✓ Returns only the rows that have matching values in both tables. It combines rows from the tables based on the specified join condition.
- ✓ It is the simple and most popular form of join and assumes as a default join.
- ✓ If we omit the INNER keyword with the JOIN query, we will get the same output.



Syntax:

```
SELECT column_name(s)
FROM table1 INNER JOIN table2
ON
table1.column_name = table2.column_name;
```

Example 1:

```
SELECT  *
FROM Employee INNER JOIN Department
ON
Employee.dept_id=Department.dept_id;
```

Output:

emp_id	emp_name	salary	dept_id	dept_id	dept_name
1	anish	25000	1	1	sales
2	sita	55000	2	2	marketing
3	ronit	40000	4	4	operations

Example 2:

```
SELECT  Employee.emp_name,Department.dept_name
FROM Employee INNER JOIN Department
ON
Employee.dept_id=Department.dept_id;
```

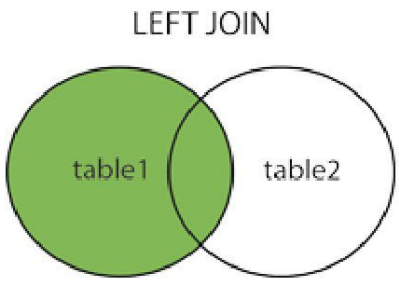
Output:

emp_name	dept_name
Anish	sales
Sita	marketing
ronit	operations

Outer JOIN

- ✓ An outer join is a type of join operation in SQL that includes unmatched rows from one or both tables in the join result.
- ✓ Unlike an inner join, which only returns matching rows, an outer join ensures that all rows from one table (or both tables) are included in the result set, even if there is no corresponding match in the other table.
  There are three types of outer joins:

1) LEFT JOIN (LEFT OUTER JOIN): Returns all rows from the left table and the matching rows from the right table. If there are no matching rows in the right table, NULL values are returned for the columns of the right table.



Syntax:

```
SELECT column_name(s)
FROM table1 LEFT JOIN table2
ON
table1.column_name = table2.column_name;
```

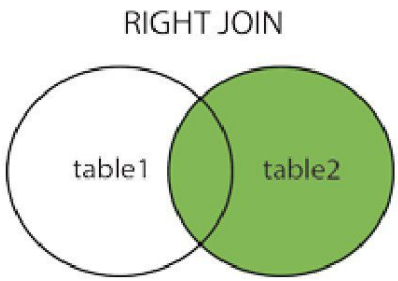
Example:

```
SELECT *
FROM Employee LEFT JOIN Department
ON
Employee.dept_id=Department.dept_id;
```

Output:

emp_id	emp_name	salary	dept_id	dept_id	dept_name
1	anish	25000	1	1	sales
2	sita	55000	2	2	marketing
3	ronit	40000	4	4	operations
4	riya	50000	5	NULL	NULL

2) RIGHT JOIN (RIGHT OUTER JOIN): Returns all rows from the right table and the matching rows from the left table. If there are no matching rows in the left table, NULL values are returned for the columns of the left table.



Syntax:

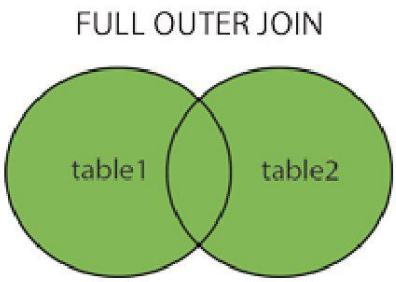
```
SELECT column_name(s)
FROM table1 RIGHT JOIN table2
ON
table1.column_name = table2.column_name;
```

Example:

```
SELECT *
FROM Employee RIGHT JOIN Department
ON
Employee.dept_id=Department.dept_id;
```

emp_id	emp_name	salary	dept_id	dept_id	dept_name
1	anish	25000	1	1	sales
2	sita	55000	2	2	marketing
NULL	NULL	NULL	NULL	3	finance
3	ronit	40000	4	4	operations

3)FULL JOIN (FULL OUTER JOIN): Returns all rows from both tables, regardless of whether they have a match or not. If there is no match, NULL values are returned for the columns of the table that does not have a match.



Syntax:

```
SELECT column_name(s)
FROM table1 FULL JOIN table2
ON
table1.column_name = table2.column_name;
```

Example:

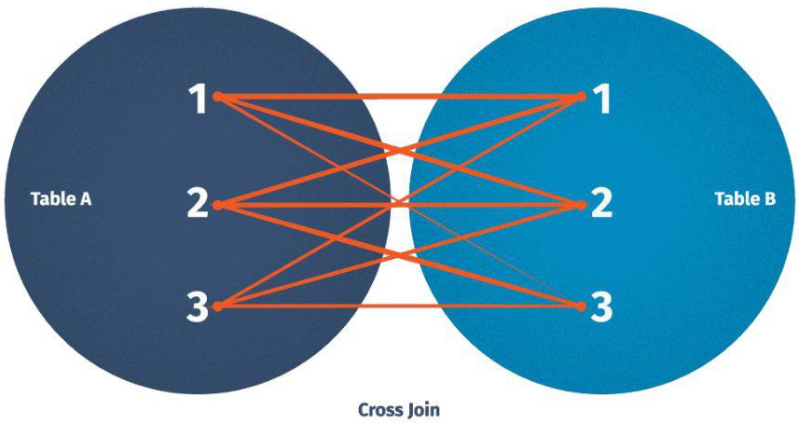
```
SELECT *
FROM Employee FULL JOIN DEPARTMENT
ON
Employee.dept_id=Department.dept_id;
```

Output

emp_id	emp_name	salary	dept_id	dept_id	dept_name
1	anish	25000	1	1	sales
2	sita	55000	2	2	marketing
3	ronit	40000	4	4	operations
4	riya	50000	5	NULL	NULL
Null	Null	Null	Null	3	finance

CROSS JOIN:

- ✓ Returns the Cartesian product of the two tables, which means it combines every row from the first table with every row from the second table.
- ✓ It does not require a join condition.
- ✓ Here each row is the combination of rows of both tables:



Syntax:

```
SELECT column(s) name
FROM table1 CROSS JOIN table2;
```

Example:

SELECT *
FROM Employee CROSS JOIN Department;

emp_id	emp_name	salary	dept_id	dept_id	dept_name
1	anish	25000	1	1	sales
2	sita	55000	2	1	sales
3	ronit	40000	4	1	sales
4	riya	50000	5	1	sales
1	anish	25000	1	2	marketing
2	sita	55000	2	2	marketing
3	ronit	40000	4	2	marketing
4	riya	50000	5	2	marketing
1	anish	25000	1	3	finance
2	sita	55000	2	3	finance
3	ronit	40000	4	3	finance
4	riya	50000	5	3	finance
1	anish	25000	1	4	operations
2	sita	55000	2	4	operations

3	ronit	40000	4	4	operations
4	riya	50000	5	4	operations

NATURAL JOIN

- ✓ The NATURAL JOIN is a type of join in SQL that automatically matches columns with the same name in the joined tables.
- ✓ It eliminates the need to specify the join condition explicitly.
- ✓ The resulting join will include only one instance of columns with the same name. It automatically eliminates duplicate columns from the join result.

Syntax:

```
SELECT column(s) name
FROM table1 NATURAL JOIN table2;
```

Example

```
SELECT *
 FROM Employee NATURAL JOIN Department;
```

Output

dept_id	emp_id	emp_name	salary	dept_name
1	1	anish	25000	sales
2	2	sita	55000	marketing
4	3	ronit	40000	operations

It's important to note that the NATURAL JOIN relies on columns having the same name and data types in both tables.

Stored procedure

- ✓ A stored procedure is a named collection of SQL statements that are precompiled and stored in a database.
- ✓ If the user has an SQL query that you write over and over again, keep it as a stored procedure and execute it.
- ✓ Users can also pass parameters to a stored procedure so that the stored procedure can act based on the parameter value that is given.
- ✓ Based on the statements in the procedure and the parameters we pass, it can perform one or multiple DML operations on the database, and return value, if any.
- ✓ Thus, it allows us to pass the same statements multiple times, thereby, enabling reusability.

Advantages

- ✓ **Reusability:** Once a stored procedure is created, it can be called multiple times from different parts of an application or by multiple users.
- ✓ **Improved performance**: Since stored procedures are precompiled and stored in the database, they can execute faster than sending individual SQL statements from an application to the database server. This is because the database server doesn't have to re-parse and optimize the code each time it is executed.
- ✓ **Reduced network traffic:** The server only passes the procedure name and parameter instead of the whole query, reducing network traffic.
- ✓ **Easy to modify:** WE can easily change the statements in a stored procedure as per necessary.
- ✓ **Security:** Stored procedures can provide an additional layer of security by allowing access to the underlying data through the procedure while restricting direct access to the tables.
- ✓ **Modularity and encapsulation:** Stored procedures enable the modularization and encapsulation of database logic, making it easier to maintain and update the database code.
- ✓ **Parameterization**: Stored procedures can accept input parameters, allowing you to pass values into the procedure at runtime. These parameters can be used within the SQL statements to make the procedure more flexible and reusable

Creating stored procedure

To create a stored procedure in **MySQL**, we can use the following syntax:

```
DELIMITER //
CREATE PROCEDURE procedure_name(parameter1 datatype, parameter2 datatype, ...)
BEGIN
  ----Statements using the input parametes
END //
DELIMITER ;
```

Note:

- ✓ **DELIMITER //** is used to change the delimiter temporarily so that you can use the semicolon ; within the procedure body without ending the entire statement prematurely.
- ✓ **DELIMITER ;** sets the delimiter back to the default semicolon ;

Executing stored procedure

To execute a stored procedure in MySQL, you can use the CALL statement followed by the name of the procedure and list of parameters if any. The syntax is as follows:

CALL procedure_name(parameter_list);

Creating stored procedure without parameters

Example:

```
DELIMITER //
CREATE PROCEDURE  getallEmployee ()
BEGIN
SELECT * FROM employee;
END //
DELIMITER ;
```

Now, we can execute the above stored procedure as follows

```
CALL getallEmployee();
```

Creating parameterized stored procedure

In SQL, a parameterized procedure is a type of stored procedure that can accept input parameters. These parameters can be used to customize the behavior of the procedure and perform operations based on the input values provided.

To create a parameterized stored procedure in MySQL, we can define input parameters within the procedure definition.

Example:

```
DELIMITER //
CREATE PROCEDURE getdepartmentEmployee (dept varchar(30))
BEGIN
SELECT *
FROM employee
WHERE department=dept;
END //
DELIMITER ;
```

To call this stored procedure, you can use the CALL statement and pass the parameter value:

```
CALL getdepartmentEmployee('civil');
```

Drop procedure

We can use the DROP PROCEDURE statement followed by the name of the procedure.

Here's the syntax:

```
DROP PROCEDURE procedure_name;
```

Example:

DROP PROCEDURE getdepartmentEmployee;

Views

- ✓ In SQL, a view is a virtual table based on the result-set of an SQL statement.
- ✓ A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.
- ✓ Views are defined based on queries, and they can be used to simplify complex queries, restrict access to certain data, or present a customized perspective of the data to different users or applications.

Syntax

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

Types of views:

1. **simple view:** creating views from single table
2. **complex view:** creating views from multiple table

For dropping view

Syntax: DROP VIEW view_name;

Need of views

In some cases, it is not desirable for all users to see all the actual relations stored in the database.

For example consider the following relation

Employee(emp_id,emp_name,postion,salary,dept_id)
Department(dep_id,dept_name,location,budget)

Consider a person who needs to know information of employees with name, position and department name but not salary as well as other information, then this person should see a relation described. In such case views are created.

Example

```
CREATE VIEW employee_info as
SELECT Employee.emp_name, Employee.position, Department.dept_name
FROM Employee,Department
WHERE Employee.dept_id=Department.dept_id;
```

Now we can deny direct access to the Employee and Department relation but grant access to the view employee_info which consists of only attributes emp_name, position from relation Employee and another attribute dept_name from Department .

SQL OLD QUESTIONS SOLUTION

1. **Write SQL statements for the following queries in reference to relation Emp_time provided. [PU:2010 Spring,PU:2011 fall]**

Eid#	Name	Start_time	End_time
E101	Magale	10:30	18:30
E102	Malati	8:30	14:30
E103	Fulmaya	9:00	18:00

i)create the table Eid# as primary key and insert the values provided

```
CREATE TABLE Emp_time (
 `Eid#` VARCHAR(10) PRIMARY KEY,
 Name VARCHAR(30),
 Start_time TIME,
 End_time TIME
);

INSERT INTO Emp_time  VALUES ('E101','Mangale', '10:30:00', '18:30:00');
INSERT INTO Emp_time  VALUES  ('E102','Malati', '8:30:00', '14:30:00');
INSERT INTO Emp_time  VALUES    ('E103', 'Fulmaya', '9:00:00', '18:00:00');
```

Note: *using backticks around the column name will allow us to use special characters. In above example we are using backtick in column name Eid#. Here special character # is used. So column name becomes `Eid#`*

ii)**Display the name of the employee whose name start from letter 'M' and who work more than seven hours**

```
SELECT Name
FROM Emp_time
WHERE Name LIKE 'M%' AND TIMEDIFF(End_time, Start_time) >  '07:00:00';
```

iii)**Delete the entire content of the table so that new records can be inserted.**

```
TRUNCATE TABLE Emp_time;
```

2. Consider the following relational database **[PU:2010 fall]**

Employee(Empno,Name,Address)
Project(Pno,Pname)
Workon(Empno,Pno)
Parts(Partno,Partname,Qty_on_Hand)
Use(Empno,Pno,Partno)
The primary key are underlined

Now write SQL command for the following:

i) Modify the database so that Jones now lives in Pokhara

```
UPDATE Employee
SET Address = 'Pokhara'
WHERE Name = 'Jones';
```

ii) Give and SQL DDL definition for Employee table(assume your own data types for attributes)

```
CREATE TABLE Employee (
   Empno INT PRIMARY KEY,
   Name VARCHAR(30),
   Address VARCHAR(30)
);
```

iii) Insert a new record into the employee table

```
INSERT INTO Employee
VALUES (11, 'Roshan Pokhrel', 'Kathamandu');
```

iv) To retrieve the name of the employee who are working on a project named “DBMS”

```
SELECT Employee.Name
FROM Employee, Workon, Project
WHERE Employee.Empno = Workon.Empno
 AND Workon.Pno = Project.Pno
 AND Project.Pname = 'DBMS';
```

3. Consider the following relation: **[PU:2012 Spring]**

Employee(empID,FirstName,LastName,address,DOB,sex,position,deptNo)
Department(deptNo,deptName,mgr,empID)
Project(projNo,projName,deptNo)
Work on(empID,projNo,hour worked)

Write SQL statements for the following

i) List the name and addresses of all employees who works for IT department

```
SELECT CONCAT(Employee.FirstName, ' ', Employee.LastName) AS name, Employee.address
FROM Employee, Department
WHERE Employee.deptNo = Department.deptNo
AND Department.deptName = 'IT';
```

ii) List the total hours worked by each employee, arranged in order of department number and within department alphabetically by employee surname

```
SELECT Employee.empID,  Employee.FirstName,Employee.LastName,Employee.deptNo,
SUM(`hour worked`) AS Totalhoursworked
FROM Employee, `Work on`
WHERE Employee.empID = `Work on`.empID
GROUP BY Employee.empID, Employee.FirstName, Employee.LastName, Employee.deptNo
ORDER BY Employee.deptNo, Employee.LastName;
```

iii) List the total number of employees in each department for those departments with more than 10 employees .

```
SELECT Department.deptName, COUNT(Employee.empID) AS TotalEmployees
FROM Employee, Department
WHERE Department.deptNo = Employee.deptNo
GROUP BY Department.deptName
HAVING COUNT(Employee.empID) > 10;
```

iv) List the project number ,project name and the number of employees who work in that project

```
 SELECT Project.projNo, Project.projName, COUNT(`Work on`.empID) AS num_employees
FROM Project ,`Work on`
WHERE Project.projNo = `Work on`.projNo
GROUP BY Project.projNo, Project.projName;
```

Note: *using backticks around the column name will allow us to use characters with blank space. Here `Work on` can be used.*

4. Consider the relational database **[PU:2012 fall]**
   Employee(emp_name,street,city)
   Works(empname,cmpname,salary)
   Company(cmpname,city)
   Manages(empname,cmpname)

   Write SQL statement to:

i. Modify the database so that Amrit now lives in Naxal

```
UPDATE Employee
SET city = 'Naxal'
WHERE emp_name = 'Amrit';
```

ii. Delete all tuples in the works relation for employee of xyz corporation

```
DELETE FROM Works
WHERE cmpname = 'xyz corporation';
```

iii. Increase salary of all employees of ABC company by 10%

```
UPDATE Works
SET salary = salary * 1.1
WHERE cmpname = 'ABC';
```

iv. Display all company name located at city pokhara or kathmandu from company tables

```
SELECT cmpname
FROM Company
WHERE city = 'Pokhara' OR city= 'Kathmandu';
```

v. Display all empname who have salary greater than 5000 from works table

```
SELECT empname
FROM Works
WHERE salary > 5000;
```

5. Consider the table tbl_emp as follow **[PU:2013 spring]**

EmpId*	EmpName	Salary(Nrs.)	Date_of_join	Phone	Department
E001	Ram	20000	2060-02-01	#1234	Packing
E002	Hari	18000	2065-04-01	#5647	Cleaning
E004	Sita	15000	2068-04-01	#2564	Polishing

Write the SQL statements for the following

i) Insert a new record

```
INSERT INTO Employee
VALUES ('E004', 'Gita', 22000, '2070-08-15', '#7890', 'Sales');
```

ii) Delete the record of sita

```
DELETE FROM Employee
WHERE EmpName='Sita';
```

iii) Change the Department of Hari to marketing

```
UPDATE Employee
SET Department = 'Marketing'
WHERE EmpName= 'Hari';
```

iv) Add a new column Address to the above table

```
ALTER TABLE Employee
ADD COLUMN Address VARCHAR(30);
```

v) Increase the salary of all employee by 5000

```
UPDATE Employee
 SET Salary = Salary + 5000;
```

vi) Select the row having salary greater than 16000

```
SELECT *
FROM Employee
WHERE Salary > 16000;
```

vii) Delete the entire table

```
DROP TABLE Employee;
```

6. Consider the relational database of figure given below, where primay keys are underlined. Given an expression in SQL for each of the following queries. **[PU:2014 spring]**

Employee(employee_name,street,city)
Works(employee_name,company_name,salary)
Company(company_name,ciy)
Manages(employee_name,manager_name)

i. modify the databases so that Ram now lives in kathmandu

```
UPDATE Employee
SET city = 'Kathmandu'
WHERE employee_name = 'Ram';
```

ii. Give all employees of First Bank Corporation a 10 percent raise

```
UPDATE Works
SET salary = salary * 1.1
WHERE company_name = 'First Bank Corporation';
```

iii. Give all managers of First Bank Corporation a 10 percent raise

```
UPDATE WORKS
SET salary = salary * 1.1
WHERE employee_name IN (SELECT manager_name  FROM Manages)
AND company_name= 'First Bank Corporation';
```

iv. Delete all in the work relation for employees of small bank corporation

```
DELETE FROM Works
WHERE company_name = 'Small Bank Corporation';
```

v. Find all employees who earn more than the average salary of all employees of their company

```
SELECT employee_name
FROM Works a
WHERE salary > (
   SELECT avg(salary)
   FROM Works b
   WHERE a.company_name = b.company_name
);
```

7. Consider the following relations: **[PU: 2014 fall]**

Employee(emp_name,street,city)
Works(emp_name,company,salary)
Company(comp_name,city)
Manages(emp_name,manager_name)
Write SQL statements for:

i. Find employee names that lives in the city same as the company city

```
SELECT Employee.emp_name
FROM Employee, Works, Company
WHERE Employee.emp_name = Works.emp_name
AND Works.company = Company.comp_name
AND Employee.city = Company.city;
```

ii. List all the employee details who earn more than 25000

```
SELECT  Employee.emp_name,Employee.street,Employee.city
FROM Employee, Works
WHERE Employee.emp_name = Works.emp_name
AND Works.salary > 25000;
```

iii. Update address of an employee 'Sriyash' to 'Pokhara'

```
UPDATE Employee
SET city = 'Pokhara'
WHERE emp_name = 'Sriyash';
```

iv. Create view for employee earns RS. 20,000 or more

```
CREATE VIEW HighEarningEmployee AS
SELECT  Employee.emp_name,Employee.street,Employee.city,Works.salary
FROM Employee, Works
WHERE Employee.emp_name = Works.emp_name
AND Works.salary >= 20000;
```

v. Delete all the employees from the table employee

```
DELETE FROM Employee;
```

8. Suppose we are given the following table definitions with certain records in each table. **[PU: 2015 Spring]**

EMPLOYEE(EID,NAME,POST,AGE)
POST(POST_TITLE,SALARY)
PROJECT (PID,PNAME,DURATION,BUDGET)
WORK-IN (PID,EID,JOIN_DATE)

Write SQL statements for

i) **List the name of Employees whose age is greater than average age of all the employees**

```
SELECT NAME
FROM EMPLOYEE
WHERE AGE > (
  SELECT AVG(AGE)
  FROM EMPLOYEE
);
```

ii) Display all the employee numbers of those employees who are not working in any project

```
SELECT EID
FROM EMPLOYEE
WHERE EID NOT IN (
  SELECT EID
  FROM WORK_IN
);
```

iii**) List the name of employee and their salary who are working in the project 'DBMS'**

```
SELECT EMPLOYEE.NAME, POST.SALARY
FROM EMPLOYEE, POST,PROJECT,WORK_IN
WHERE PROJECT.PID=WORK_IN.PID AND
WORK_IN.EID=EMPLOYEE.EID  AND
EMPLOYEE.POST=POST.POST_TITLE  AND
PROJECT.PNAME='DBMS';
```

iv) update the database so that "Rishab now lives in "Butwal"

(This question cannot be solved from above relation. It seems question is incorrect)

9. Write SQL statements for the following queries in reference to relation Emp_time provided. **[2015 Fall]**

Eid*	Name	Start_time	End_time
E101	Hari	10:15	18:00
E102	Malati	8:00	15:30
E103	Kalyan	9:30	17:00

i)create the table Eid* as primary key and insert the values provided.

ii)Display the name of the employee whose name starts from letter 'M' and who work more than seven hours.

iii)Delete the entire content of the table so that new records can be inserted.

Solution:

(similar to Question of PU:2010 Spring and solution is already done in Question No.1)

10. Consider a simple relational database of Hospital management system.(Underlined attributes represent primary key attributes)

    Doctors (Doctor_ID,DoctorName,Department,Address,Salary)
    Patients(PatientID,Patient_Name,Address,Age,Gender)
    Hospitals (PatientID,DoctorID,HospitalName,Location)
    Write down the SQL statements for the following
    [PU:2016 spring][PU:2019 fall]

i. **Display ID of patient admitted in hospital at pokhara and whose name ends with 'a'**

```
SELECT Patients.PatientID
FROM Patients,Hospitals
WHERE Hospitals.PatientID = Patients.PatientID
AND Hospitals.Location= 'Pokhara' AND Patients.Patient_Name LIKE '%a';
```

ii. **Delete the records of Doctors whose salary is greater than average salary of doctors**

```
DELETE FROM Doctors
WHERE Salary > (SELECT AVG(Salary) FROM Doctors);
```

iii. **Increase salary of doctors by 18.5% who works in OPD department**

```
UPDATE Doctors
SET Salary = Salary * 1.185
WHERE Department = 'OPD';
```

iv. **Find the average salary of Doctors for each address who have average salary more than 55k.**

```
SELECT Address, AVG(Salary) AS AverageSalary
FROM Doctors
GROUP BY Address
HAVING AVG(Salary) > 55000;
```

11. Consider the relational schema: **[PU:2016 fall]**

Teacher (**TeacherID**,TeacherName,Offfice)

Write SQL statements for the following task:

i. To create a table from a table

```
CREATE TABLE Teacher
(
 TeacherID INT PRIMARY KEY,
 TeacherName VARCHAR(30),
 Office VARCHAR(30)
);
```

ii. To eliminate duplicate rows

```
SELECT DISTINCT *
FROM Teacher;
```

iii. To add new column 'Gender' in the table

```
ALTER TABLE Teacher ADD Gender VARCHAR(10);
```

(Note: This query is for mariadb)

iv. To sort data in a table

```
SELECT *
FROM Teacher
ORDER BY TeacherName ASC;
```

v. To delete rows

```
DELETE FROM Teacher
WHERE TeacherID = 123;
```

In this example, rows with TeacherID equal to 123 is deleted. we can modify the condition in the WHERE clause to match your specific deletion criteria.

vi. Count the number of rows based in office

```
SELECT Office, COUNT(*) AS RowCount
FROM Teacher
GROUP BY Office;
```

12. Write the SQL statements for the following Queries by reference to Liquors_info relation:

[PU:2017 fall]

Serial_No	Liquors	Start_year	Bottles	Ready_Year
1	Gorkha	1997	10	1998
2	Divine Wine	1998	5	2000
3	Old Durbar	1997	12	2001
4	Khukhuri Rum	1991	10	1992
5	Xing	1994	5	1995

i) **creates the Liquors_info relation**

```
CREATE TABLE Liquors_info
 (
   Serial_No INT,
   Liquors VARCHAR(50),
   Start_year YEAR,
   Bottles INT,
   Ready_Year YEAR
);
```

ii)insert the records in Liquor_info as above

```
INSERT INTO Liquors_info
VALUES (1, 'Gorkha', '1997', 10, '1998');

INSERT INTO Liquors_info
VALUES (2, 'Divine Wine', '1998', 5, '2000');

INSERT INTO Liquors_info
VALUES (3, 'Old Durbar', '1997', 12, '2001');

INSERT INTO Liquors_info
VALUES (4, 'Khukhuri Rum', '1991', 10, '1992');

INSERT INTO Liquors_info
VALUES (5, 'Xing', '1994', 5, '1995');
```

iii)**List all the records which were ready by 2000**

```
SELECT *
FROM Liquors_info
WHERE Ready_Year <= '2000';
```

iv)**Remove all records from database that required more than 2 years to get ready**

```
DELETE FROM Liquors_info

WHERE (Ready_Year -Start_year) > 2;
```

13. Write SQL statements for the following: [PU:2018 spring]

create a table named Vehicle with veh_number as primary key and following attributes:

veh_type,veh_brand,veh_year,veh_mileage,veh_owner,veh_photo,veh_price

```
CREATE TABLE Vehicle (
  veh_number VARCHAR(50)  PRIMARY KEY,
  veh_type VARCHAR(50),
  veh_brand VARCHAR(50),
  veh_year YEAR,
  veh_mileage int,
  veh_owner VARCHAR(50),
  veh_photo LONGBLOB,
  veh_price DECIMAL(12, 2)
);
```

ii)Enter a full detailed information of a vehicle

```
INSERT INTO Vehicle
VALUES ('B DE 5425', 'Sedan', 'Toyota', 2022, 55, 'Hari Pokhrel',
LOAD_FILE('C:\\Users\\Downloads\\veh1.jpg'), 18500.75);
```

iii)Increment a vehicle price by 10,000

```
UPDATE Vehicle
SET veh_price = veh_price + 10000
WHERE veh_number = 'B AA 8422'
```

This SQL UPDATE statement will find the vehicle with veh_number **B AA 8422** *in the "Vehicle" table and increase its veh_price by 10,000.*

iv)**Remove all vehicle's records whose brand contains character 'o' second position**

```
DELETE FROM Vehicle
WHERE veh_brand LIKE '_o%';
```

v)Display the total price of all vehicles

```
SELECT SUM(veh_price) AS total_price
FROM Vehicle;
```

vi)create a view a from above table

```
CREATE VIEW Low_price_vehicle AS
SELECT veh_number, veh_type, veh_brand, veh_year, veh_price
FROM Vehicle
WHERE veh_price < 150000;
```

vii)Display details of vehicles ordering on descending manner in brand and by mileage when brand matches

```
SELECT *
FROM Vehicle
ORDER BY veh_brand DESC, veh_mileage DESC;
```

viii) change data type of year to datetime

```
ALTER TABLE Vehicle
MODIFY COLUMN veh_year DATETIME;
```

14. Consider the following three relations **[PU:2018 fall]**

Doctor(Name,age,address)
Works(Name,Depart_no,salary)
Department(Depart_no,dept_name,floor,room)
Write down the SQL statement for the following

i)**Display the name of doctor who do not work in any department**

```
SELECT Name
FROM Doctor
WHERE Name NOT IN (
  SELECT Name
  FROM Works
);
```

ii) **Modify the database so that Dr.Hari Lives in pokhara**

```
UPDATE Doctor
SET address = 'Pokhara'
WHERE Name = 'Dr. Hari';
```

iii) **Delete all records of Doctor working OPD department**

```
DELETE FROM Doctor
WHERE Name IN
( SELECT Works.Name
FROM Works ,Department
WHERE Works.Depart_no = Department.Depart_no
 AND Department.dept_name = 'OPD'
);
```

iv) **Display the name of doctors who works in at least two department**

```
SELECT Doctors.Name
FROM Doctor,Works
WHERE Doctors. Name = Works.Name
GROUP BY Doctors.Name
HAVING COUNT(DISTINCT Works.Depart_no) >= 2;
```

15. Write the SQL statements for the following queries by reference of Hotel_details relation. [PU:2019 spring]

Hotel_id	Hotel_name	Estb_year	Hotel_star	Hotel_worth
1	Hyatt	2047	Five	15M
2	Hotel ktm	2043	Three	5M
3	Fullbari	2058	Five	20M
4	Yak and Yeti	2052	Four	11M
5	Hotel chitwan	2055	Three	7M

i) **create a database named hotel and table relation**

```
-- Create the database

CREATE DATABASE hotel;

-- Switch to the hotel database

USE hotel;

-- Create the table Hotel_details

CREATE TABLE Hotel_details (
  Hotel_id INT,
  Hotel_name VARCHAR(50),
  Estb_year INT,
  Hotel_star VARCHAR(10),
  Hotel_worth BIGINT
);
```

ii)create a view named price which shows hotel name and its worth

```
CREATE VIEW price AS
SELECT Hotel_name, Hotel_worth
FROM Hotel_details;
```

iii) modify the data so that hotel chitwan is now four star level

```
UPDATE Hotel_details
SET Hotel_star = 'Four'
WHERE Hotel_name = 'Hotel_chitwan';
```

iv)delete the records of all hotels having worth more than 9M

```
DELETE FROM Hotel_details
WHERE Hotel_worth > 9000000;
```

16. Write SQL statement for the following schemas (underline indicates primary key) **[PU:2020 spring]**

Employee(**Emp_No**,Name,Address)
Project(**PNo**,Pname)
Workon(**Emp_No**,PNo)
Part(**Partno**,Part_name,Qty_on_hand)
Use(**Emp_No**,**PNo**,**Partno**,Number)

a. Listing all the employee details who are not working yet

```
SELECT Emp_No, Name, Address
FROM Employee
WHERE Emp_No NOT IN (SELECT DISTINCT Empno FROM Workon);
```

b. Listing Part_name and Qty_on_hand those were used in DBMS project

```
SELECT Part.Part_name, Part.Qty_on_hand
FROM Part,Use,Project
WHERE Part.Partno = Use.Partno
AND Use.PNo=Project.PNo
AND Project.pname='DBMS';
```

c. List the name of the projects that are used by employee from London

```
SELECT DISTINCT Project.Pname
FROM Project,Use,Employee
WHERE Project.PNo = Use.PNo AND
Use.Emp_No = Employee.Emp_No AND
Employee.Address = 'London';
```

d. Modify the database so that Jones now lives in 'USA'

```
UPDATE Employee
SET Address = 'USA'
WHERE Name = 'Jones';
```

e. Update address of an employee 'Japan' to 'USA'

```
UPDATE Employee
SET Address = 'USA'
WHERE Address = 'Japan';
```

17. Write a SQL statements for the following **[PU:2020 fall]**

i)Create a table named Automotor with chasis_number as primary key and following attributes.

veh_brand,veh_name,veh_model,veh_year,veh_cost,veh_color,veh_weight

```
CREATE TABLE Automotor (
  chasis_number INT PRIMARY KEY,
  veh_brand VARCHAR(50),
  veh_name VARCHAR(50),
  veh_model VARCHAR(50),
  veh_year YEAR,
  veh_cost DECIMAL(10,2),
  veh_color VARCHAR(10),
  veh_weight DECIMAL(10,2)
);
```

ii)**Enter a full detailed information of automotor**

```
INSERT INTO Automotor

VALUES (1654, 'Toyota', 'Corolla', 'XE', 2020, 25000.00, 'Red', 1200.50);
```

iii)Change any Automotor's year to 2019

```
UPDATE Automotor
SET veh_year = 2019
WHERE chasis_number=125;
```

iv)Remove all Automotor's records whose model contains 'i' in last position

```
DELETE FROM Automotor

WHERE veh_model LIKE '%i';
```

v)Display the total cost of all vehicles of the table Automotor

```
SELECT SUM(veh_cost) AS total_cost
FROM Automotor;
```

vi) Create a view from above table having vehicle only red color

```
CREATE VIEW RedVehicles AS
SELECT * FROM Automotor
 WHERE veh_color = 'red';
```

vii) Display details of Automotor ordering on descending manner by brand name and ascending order on model when brand matches

```
SELECT *
FROM Automotor
ORDER BY veh_brand DESC, veh_model ASC;
```

viii) change data type of veh_color so that it only takes one character

```
ALTER TABLE Automotor
MODIFY COLUMN veh_color CHAR(1);
```

(Note: This query is for mariaDB. Syntax may vary in different DBMS)

18. Let us consider the following relation

Sailors (sid,sname,rating,age)
Boats(bid, bname,color)
Reserves(sid,bid,day)

Write a SQL statements for the following **[PU:2021 spring]**

i)Find the records of sailors who have reserved boat number 103(bid=103)

```
SELECT Sailors.sid,Sailors.sname,Sailors.rating,Sailors.age
FROM Sailors , Reserves
WHERE Sailors.sid = Reserves.sid
AND Reserves.bid = 103;
```

ii)Update the color of the boat ,where bid is 104,into green

```
UPDATE Boats
SET color = 'green'
WHERE bid = 104;
```

iii) find the name of sailors who have reserved a red or green boat

```
SELECT  Sailors.sname
FROM Sailors, Reserves, Boats
WHERE Sailors.sid = Reserves.sid
AND Reserves.bid = Boats.bid
AND Boats.color IN ('red', 'green');
```

iv)find the name of sailors who have reserved boat number 103 on day 5

```
SELECT Sailors.sname
FROM Sailors , Reserves
WHERE Sailors.sid = Reserves.sid
 AND Reserves.bid = 103
 AND Reserves.day = 5;
```

v)find the name of sailors whose name is not 'Ram'

```
SELECT  sname
FROM Sailors
WHERE sname != 'Ram';
```

vi) find the name of all boats

```
SELECT  bname

FROM Boats;
```

18.Consider the relation Actress_Details and Write SQL statements for the following queries. **[PU:2022 fall]**

Players_id	Actress_name	Debut_year	Recent_release	Actress_fee
1	Renu	2010	Samay	400000
2	Sita	2022	Radha	300000
3	Geeta	2001	Mato	600000
4	Amita	1990	Man	700000
5	Karishma	1989	Prem	100000

i) Create the table Actress_details relation

```
CREATE TABLE Actress_details (
   Players_id INT PRIMARY KEY,
   Actress_name VARCHAR(50),
   Debut_year YEAR,
   Recent_release VARCHAR(50),
   Actress_fee INT
);
```

ii) Delete the data of actress whose recent release is prem

```
DELETE FROM Actress_details
WHERE Recent_release = 'Prem';
```

iii) Modify the database so that Renu's new release is "Win the race" film

```
UPDATE Actress_details
SET Recent_release = 'Win the race'
WHERE Actress_name = 'Renu';
```

iv) Insert a new record in the above table

```
INSERT INTO Actress_details
VALUES (6, 'Priya', '2015', 'Sunset', 500000);
```

19. Write SQL statements for the following queries using the given Employees relation: **[PU:2023 spring]**

E_id	Fname	Lname	Department	Salary	Hire_Date
01	Ramu	Bashyal	Sales	20000	2023-08-08
02	Damu	Pandey	IT	50000	2022-01-01
03	Biru	B.k.	Sales	40000	2021-02-10
04	Hiru	Dhamala	HR	35000	2023-12-18
05	Biren	Khadka	IT	60000	2012-10-22

i)Create a database named Company and Employees relation.

```
CREATE DATABASE Company;

CREATE TABLE Employees(
E_id INT PRIMARY KEY,
Fname varchar(30),
Lname,varchar(30),
Department varchar(30),
Salary INT,
Hire_Date DATE
);
```

ii)Create a view that shows the E_id ,Department and Hire_Date of all employees

```
CREATE VIEW emp_view
SELECT E_id,Department,Hire_Date
FROM employees;
```

iii)Modify the table such that the Department of Biren is HR now.

```
Update Employees
SET Department='HR'
WHERE Fname='Biren';
```

iv) Delete the record of employees whose Lname is "Pandey"

```
DELETE FROM Employees
WHERE Lname='Pandey';
```

20. Consider the following relation

Orders(order_id,product_name,price,quantity,order_date,delivery_date)

1) Create table orders

```
CREATE TABLE Orders (
   order_id INT PRIMARY KEY,
   product_name VARCHAR(50) ,
   price DECIMAL(10, 2) ,
   quantity INT ,
   order_date DATE ,
   delivery_date DATE
);
```

2) Now insert any 8 records

```
INSERT INTO Orders
VALUES (1, 'T-shirt', 25.99, 2, '2023-07-15', '2023-07-25');

INSERT INTO Orders
VALUES (2, 'Jeans', 49.95, 1, '2023-07-17', '2023-07-20');

INSERT INTO Orders
VALUES (3, 'Shoes', 69.50, 1, '2023-07-20', '2023-07-30');

INSERT INTO Orders
VALUES (4, 'Sunglasses', 12.75, 3, '2023-07-22', '2023-07-28');

INSERT INTO Orders
VALUES (5, 'Backpack', 34.99, 2, '2023-07-25', '2023-07-29');

INSERT INTO Orders
VALUES (6, 'Headphones', 59.99, 1, '2023-07-29', '2023-08-05');

INSERT INTO Orders
VALUES (7, 'Smartphone', 299.99, 2, '2023-07-29', '2023-11-01');

INSERT INTO Orders
VALUES (8, 'Laptop', 799.95, 1, '2023-07-29', '2025-08-01');
```

3) Retrieve all orders placed on a 2023-07-15

```
SELECT *
 FROM Orders
WHERE order_date = '2023-07-15';
```

4) Find the number of days that required to delivered shoes

```
SELECT DATEDIFF(delivery_date, order_date) AS delivery_time
FROM Orders
where product_name='shoes';
```

5) Find all the orders that is received from '2023-07-15' to '2023-07-25'

```
SELECT *
FROM Orders
WHERE order_date BETWEEN '2023-07-15' AND '2023-07-25';
```

6) find all the orders that is received today

```
SELECT *
FROM Orders
WHERE order_date = CURDATE();
```

7) Calculate the average number of days it takes to deliver a orders

```
SELECT AVG(DATEDIFF(delivery_date, order_date)) AS avg_delivery_time
FROM Orders;
```

Here, in DATDIFF() function we have passed two parameters that is **DATEDIFF(date1, date2)**

This will returns date difference in terms of number of days (date1-date2) and this result occurs if we run this query on **MySQL DBMS.**

But sometimes it is necessary to find out date difference in terms of number of month, week, year, quarter etc. In such case three parameters need to passed three parameters.

Syntax

```
DATEDIFF(interval, date1, date2)
```

Parameter Values

Parameter	Description
interval	Required. The part to return. Can be one of the following values: • year, yyyy, yy = Year • quarter, qq, q = Quarter • month, mm, m = month • dayofyear = Day of the year • day, dy, y = Day • week, ww, wk = Week • weekday, dw, w = Weekday • hour, hh = hour • minute, mi, n = Minute • second, ss, s = Second • millisecond, ms = Millisecond
date1, date2	Required. The two dates to calculate the difference between

Note:

- DATEDIFF() function with two parameters are supported in MYSQL DBMS.
- DATEDIFF() function with three parameters are supported in MS SQL Server DBMS

If you want test query in different DBMS ,you can follow this link

http://sqlfiddle.com

8) Find the number of months required to deliver smartphone

```
SELECT DATEDIFF(delivery_date, order_date) AS delivery_time
FROM Orders
where product_name='smartphone';
```

If we run this query in MySQL DBMS.

DATEDIFF() function will returns 95 for this query by considering above relations.

delivery_time
95

DATEDIFF() with three parameters are not supported in MySQL DBMS.

This query can be re-written as follows by passing three parameters in MS SQL server DBMS.

```
SELECT DATEDIFF(month, order_date,delivery_date) AS delivery_time
FROM Orders
where product_name='smartphone';
```

Note: you can write DATEDIFF() function with three parameters in exam.

9) Find the number of weeks required to deliver smartphone

```
SELECT DATEDIFF(week, order_date,delivery_date) AS delivery_time
FROM Orders
where product_name='smartphone';
```

10) Find the products that required more than 2 month to delivered

```
SELECT product_name
FROM orders
WHERE DATEDIFF(month,order_date,delivery_date)>2;
```

11) Find the products that required more than 3 weeks to delivered

```
SELECT product_name
FROM orders
WHERE DATEDIFF(week,order_date,delivery_date)>3;
```

12. **Find the products that required more than 1 years to delivered.**

```
SELECT product_name
FROM orders
WHERE DATEDIFF(year,order_date,delivery_date)>1;
```

The SQL SELECT TOP Clause

- ✓ The SELECT TOP clause is used to specify the number of records to return.
- ✓ The SELECT TOP clause is useful on large tables with thousands of records. Returning a large number of records can impact performance.

Note: Not all database systems support the SELECT TOP clause. MySQL supports the LIMIT clause to select a limited number of records, while Oracle uses FETCH FIRST n ROWS ONLY and ROWNUM.

My SQL syntax

```
SELECT column_name(s)
FROM table_name
WHERE condition
LIMIT number;
```

Find the top 3 records from orders from orders

```
SELECT *
FROM orders
LIMIT 3;
```

Create relational database for the Department of computer Engineering (DOCE) of pokhara university. Your database should have at least three relations Describe referential integrity constraint based on above database of DOCE.[PU:2017 spring]

Based on the Department of Computer Engineering (DOCE) of Pokhara University, we can create a relational database with following relations: "Student," "Faculty ,"Course" "Enroll"

Student(student_id,student_name,email,address)
Faculty(faculty_id,faculty_name,qualification)
Course(course_id ,course_name,course_description,faculty_id)
Enroll(enroll_id,student_id,course_id,enrollment_date)

Now creating tables

```
CREATE TABLE Student (
   student_id INT PRIMARY KEY,
   student_name VARCHAR(50),
   email VARCHAR(50),
   address VARCHAR(100)
);
CREATE TABLE Faculty (
   faculty_id INT PRIMARY KEY,
   faculty_name VARCHAR(50),
   qualification VARCHAR(50)
);
CREATE TABLE Course (
   course_id INT PRIMARY KEY,
   course_name VARCHAR(50),
   course_description LONGTEXT,
   faculty_id INT,
   FOREIGN KEY (faculty_id) REFERENCES Faculty (faculty_id)
);
CREATE TABLE Enroll (
   enroll_id INT PRIMARY KEY,
   student_id INT,
   course_id INT,
   enrollment_date DATE,
   FOREIGN KEY (student_id) REFERENCES Student(student_id),
   FOREIGN KEY (course_id) REFERENCES Course(course_id)
);
```

Based on the provided relations, the following are the foreign key integrity constraints that can be applied to maintain referential integrity:

Course table:
The faculty_id column in the Course table is a foreign key referencing the faculty_id column in the Faculty table. This ensures that the faculty_id value in the Course table must exist in the Faculty table.

Enroll table:

The student_id column in the Enroll table is a foreign key referencing the student_id column in the Student table. This ensures that the student_id value in the Enroll table must exist in the Student table.
The course_id column in the Enroll table is a foreign key referencing the course_id column in the Course table. This ensures that the course_id value in the Enroll table must exist in the Course table.

These foreign key constraints help maintain data integrity by enforcing the relationships between the tables. They prevent the insertion of invalid values that do not exist in the referenced tables, ensuring the consistency of the data across the relations.

Note: You can draw schema diagram for above relations as well.

Unit 4

Relational database design

- ✓ A relational database is a database based on the relational model of data, as proposed by E. F. Codd in 1970.
- ✓ A relational database organizes data in tables (or relations). A table is made up of rows and columns. A row is also called a record (or tuple). A column is also called a field (or attribute).
- ✓ The relationships that can be created among the tables enable a relational database to efficiently store huge amount of data, and effectively retrieve selected data.
- ✓ A language called SQL (Structured Query Language) was developed to work with relational databases.
- ✓ Today, there are many commercial Relational Database Management System (RDBMS), such as Oracle, IBM DB2 and Microsoft SQL Server.
- ✓ There are also many free and open-source RDBMS, such as MySQL.

Features of good relational database design

- ✓ **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.
- ✓ **Scalability:** The design should accommodate future growth and evolving data requirements
- ✓ **Accuracy and Integrity:** the data stored in the database is correct, complete, and consistent. This is achieved by enforcing data validation rules, such as data type, length, and format.
- ✓ **Performance:** database can process data efficiently and quickly, even under high load conditions
- ✓ **Privacy and Security:** The objective of privacy and security in a database is to protect the confidentiality, integrity, and availability of the data stored in the database. This is achieved by implementing access controls, such as user authentication and authorization.

Integrity constraints

- ✓ In Database Management Systems, integrity constraints are pre-defined set of rules that are applied on the table fields(columns) or relations to ensure that the overall validity, integrity, and consistency of the data present in the database table is maintained.
- ✓ Evaluation of all the conditions or rules mentioned in the integrity constraint is done every time a table insert, update, delete, or alter operation is performed.
- ✓ The data can be inserted, updated, deleted, or altered only if the result of the constraint comes out to be True. Thus, integrity constraints are useful in preventing any accidental damage to the database by an authorized user.

Types of Integrity constraints in DBMS

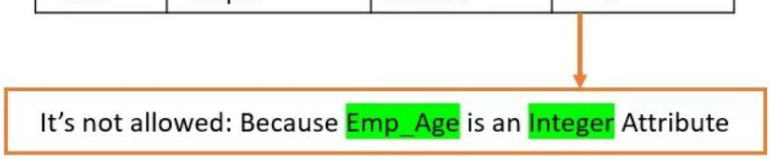
1. Domain Integrity Constraint
2. Entity Integrity Constraint
3. Referential Integrity Constraint

1. Domain integrity constraints

- ✓ A domain constraint is a set of rules that restricts the kind of attributes or values a column or relation can hold in the database table. i.e. we can specify if a specific column can hold null values or not, furthermore, if the values have to be unique or not, the size of values or the data type that can be entered in the column, the default values for the column, etc.

Let us consider the following table

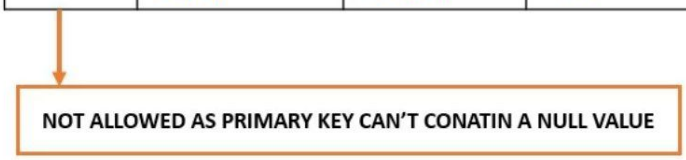
Emp_ID	Emp_Name	Emp_City	Emp_Age
101	Arun	Kathmandu	32
102	Sita	Pokhara	36
103	Rita	Chitwan	41
104	Gopal	Butwal	Z



2. Entity Integrity Constraints

- ✓ Entity Integrity Constraint is used to ensure the uniqueness of each record or row in the data table.
- ✓ In other words, the entity integrity constraint states that primary key value can't be null. This is because the primary key value is used to identify individual rows in relation and if the primary key has a null value, then we can't identify those rows.

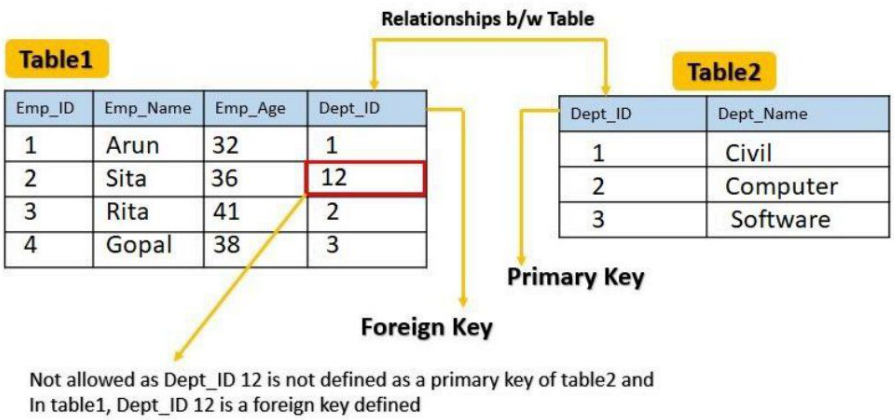
Emp_ID	Emp_Name	Emp_City	Emp_Age
101	Arun	Kathmandu	32
102	Sita	Pokhara	36
103	Rita	Chitwan	41
	Gopal	Butwal	38



3. Referential integrity constraints

- ✓ Referential integrity constraint is specified between two tables.
- ✓ Referential Integrity Constraint is defined as it ensures that there always exists a valid relationship between two tables.
- ✓ Furthermore, this makes confirms that if an foreign key exists in a table relationship then it should always reference a corresponding value in the 2nd table or it should be null.

Let us consider the following example



Assertions

- ✓ An assertion is a statement in SQL that ensures a certain condition will always exist in the database.
- ✓ Same as domain or other constraints, assertions differ in the way that they are defined separately from table definitions.
- ✓ An assertion in SQL takes the form

  create assertion <assertion-name> check <predicate condition>
- ✓ When an assertion is made, the system tests it for validity, and tests it again on every update that may violate the assertion. This testing may introduce a significant amount of overhead; hence assertions should be used with great care.
- ✓ The assertion is said to be violated if the query result is not empty and hence the user is denied from modifying the database.

Example : Let us consider the following relation

sailors(sid,sname,rating,age)
Boats(bid,bname,color)
Reserves(sid,bid,day)

Create an Assertion to For number of boats plus number of sailors is < 100

```
CREATE ASSERTION check_number
 CHECK  ((SELECT  COUNT (sid) FROM SAILORS )  +
(SELECT COUNT (bid)  FROM BOATS  )<100);
```

Triggers

A trigger is a statement that the system executes automatically as a side effect of modification of database.

To design a trigger mechanism, we must meet two requirements.

- ✓ Specify when a trigger is to be executed. This is broken up into an event that causes the trigger to be checked and a condition that must be satisfied for trigger execution to proceed.
- ✓ Specify the actions to be taken when the trigger executes

Once we enter a trigger into a database, the database system takes on the responsibility of executing it whenever the specified event occurs and the corresponding condition satisfied.

Syntax:

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER } {INSERT | UPDATE | DELETE}
ON table_name
[FOR EACH ROW]
[WHEN (condition)]
BEGIN
  -- Trigger body: SQL statements or code
END;
```

CREATE [OR REPLACE] TRIGGER: Specifies the creation of a new trigger or replaces an existing trigger with the same name.

BEFORE | AFTER: Specifies when the trigger should be executed relative to the triggering event. A "BEFORE" trigger fires before the event, and an "AFTER" trigger fires after the event.

INSERT | UPDATE | DELETE: Indicates the event that triggers the execution of the trigger. You can specify multiple events using commas.

ON table_name: Specifies the name of the table on which the trigger is defined.

FOR EACH ROW: Indicates that the trigger should be executed once for each affected row (used for row-level triggers).

BEGIN and END: Enclose the trigger body, which contains the SQL statements or code to be executed when the trigger is fired.

Need of triggers

Triggers can be used to implement certain integrity constraints that cannot be specified using the constraint mechanism of SQL. Triggers are useful mechanisms for alerting humans or for starting certain tasks automatically when certain conditions are met.

Suppose a warehouse wishes to maintain the a minimum inventory of each item. When inventory level of an items falls below the minimum level, an order can by placed automatically. On update of the inventory level of each item, the trigger compares the current inventory level with minimum inventory level for the item, and if the level is at below new order is created.

Implementation

First, we need to create a table called "Items" that will store the item details, including the inventory level and the minimum level. The table schema might look like this:

Items (ItemID ,ItemName ,InventoryLevel , MinimumLevel)

We need a table to store the generated orders when the inventory level falls below the minimum level. Create a table called "Orders" with the necessary columns to store the order details.

Orders (OrderID , ItemID ,Quantity ,OrderDate)

Now, we'll create a trigger that will automatically place an order when the inventory level falls below the minimum level for any item. The trigger will be executed after an update operation on the "Items" table.

```
DELIMITER //
CREATE TRIGGER check_inventory_level
AFTER UPDATE ON Items
FOR EACH ROW
BEGIN
   DECLARE current_inventory INT;
   DECLARE minimum_level INT;
   SET current_inventory = NEW.InventoryLevel;
   SET minimum_level = NEW.MinimumLevel;

   IF current_inventory < minimum_level THEN
      INSERT INTO Orders (ItemID, Quantity, OrderDate)
      VALUES (NEW.ItemID, (minimum_level - current_inventory), CURDATE());
   END IF;
END //
DELIMITER ;
```

Here in orders table we have assumed that it we defined the "OrderID" column to be an auto-increment primary key.

Note that auto-increment allows a unique number to be generated automatically when a new record is inserted into a table. Often primary key field that we would like to be created automatically every time a new record is inserted.

In this trigger:

- The trigger is defined as an "AFTER UPDATE" trigger on the "Items" table.
- For each updated row, the trigger checks the current inventory level (NEW.InventoryLevel) and the minimum inventory level (NEW.MinimumLevel).
- If the current inventory level is less than minimum level, a new order is inserted into the "Orders" table. The order includes the ItemID, the quantity needed to reach the minimum level (minimum_level - current_inventory), and the current date (CURDATE()).

Functional dependency

- ✓ A functional dependency is a relationship between two attributes, typically between the Primary key and other non-key attributes within a table.
- ✓ For a given relation R with attribute X and Y, Y is said to be functionally dependent on X, if given value for each X uniquely determines the value of the attribute in Y.
- ✓ X is called determinant and Y is the object of the determinant and denoted by X→Y.

Example:

emp_id	emp_name	address	salary
101	ram	kathmandu	25000
102	shyam	pokhara	47000
103	hari	Chitwan	65000

Assume we have a employee table with attributes emp_id,emp_name,address,salary.

Here emp_id attribute can uniquely identify the emp_name attribute of employee table because if we know the emp_id we can tell the employee's name associated with it.

Functional dependency can be written as:

emp_id→emp_name

we can say that emp_name is functionally dependent on emp_id.

Similarly,

emp_id→address
emp_id→salary

compound determinant

- ✓ If more than one attribute is necessary to determine another attribute, then such determinant is called as compound Determinant

Let us consider the Student relation

Student(student_id,student_name,salutation,address,course_id,course_name,marks)

{student_id,course_id} → marks

- ✓ In this example, the compound determinant is (student_id, course_id).
- ✓ It indicates that the attributes marks is determined by the combination of student_id and course_id.

Types of functional dependency

❖ Full functional dependency

- ✓ A full functional dependency is dependency in which a non-key attributes is dependent on all the attributes of composite key.
- ✓ This is the situation in which all the attributes of the composite key is used to uniquely identify its object.
- ✓ A Functional dependency X→Y is said to be full functional dependency if removal of any attribute A from X removes the dependency, which means set of attributes on the left side cannot be reduced further.

❖ Partial functional dependency

- ✓ A partial dependency is dependency in which a non-key attribute is dependent on only a part of composite key.
- ✓ This is the situation in which only a subset of the attributes of the composite key is used to uniquely identify its object.
- ✓ A Functional dependency X→Y is partial dependency if some attributes A ∈ X can be removed from X and dependency still holds.

Example:

student_id	student_name	salutation	course_id	course_name	marks
1	Ram	Mr.	1	DBMS	92
2	Shyam	Mr.	2	C++	85
3	Gita	Ms.	1	DBMS	78
4	Hari	Mr.	3	Java	75
5	Ritu	Ms.	2	C++	68

Here, {student_id,course_id}→marks

is an example of **full functional dependency**. Removal of any attributes breaks the dependency.

Here,student_id or course_id alone cannot determine marks.

Again,

{student_id,course_id}→{course_name}

is an example of **partial dependency.**

{course_id}→{course_name} still holds true. Here, removal of student_id do not have any effect.

❖ Trival functional dependency

X →Y is trivial functional dependency if Y is a subset of X.

The following dependencies are also trivial: X→X and Y→Y

Example

In above student relation

{student_id,student_name}→student_name

{course_id,course_name}→course_name

student_id→student_id

are examples of trival functional dependency.

❖ Non-trivial functional dependency

- ✓ X → Y has a non-trivial functional dependency if Y is not a subset of X.
- ✓ When X intersection Y is NULL, then X → Y is called as complete non-trivial.

Example:

In above student relation

{student_id,course_id} →marks

Student_id → student_name

Course_id → course_name

❖ Transitive dependency

- ✓ A Transitive Dependency is a type of functional dependency which happens when it is indirectly formed by two functional dependencies.
- ✓ If X→ Y and Y→Z then X→Z is a transitive dependency.

Example:In above student relation, we have the following functional dependencies:

student_id →student_name(Each student_id uniquely determines the student_name)

student_name →salutation (Each student_name uniquely determines the salutation)

Therefore, by transitive dependency, we can conclude that student_id indirectly determines the salutation. i.e. student_id→salutation

❖ Multivalued dependency

- ✓ A multivalued dependency between two attributes X and Y in relation R is represented as X→→Y which means for each value of attribute X there can be multiple corresponding value of Y.
- ✓ multivalued dependency occurs when there are more than one independent multivalued attributes in a relation.
- ✓ A multivalued dependency consist of at least two attributes that are dependent on third attribute that's why it requires at least three attributes.

Example:

course	teacher	book
DBMS	Ramesh	Database concept
DBMS	Nikita	Fundamental on DBMS
DBMS	Ramesh	Fundamentals on DBMS
DBMS	Nikita	Database concept

Here attributes teacher and book are dependent on course and independent to each other.

As well course is associated with set of teachers and set of books .The representation of these dependencies is shown below.

course→→teacher

course→→ book

Closure set of functional dependencies

- ✓ For a given set of functional dependencies F, there are certain other functional dependencies that are logically implies by F.
- ✓ For example if A→B and B→C, then we can infer that A→C.
- ✓ The set of all functional dependencies logically implied by F is the closure of F.
- ✓ Closure of F is denoted by F^+.
- ✓ F^+ is superset of F.

We can use the following three rules to find logically implied functional dependencies. By applying this rules repeatedly ,we can find all of F^+ for given F. This collection of rules is called **Armstrong's axioms** in honor of the person who purposed it.

Reflexivity rule: If α is a set of attributes and β ⊆ α,then α → β holds.

Augmentation rule: If α → β holds and γ is a set of attributes ,then γ α →γ β holds.

Transitivity rule: if α → β holds and β →γ holds then α →γ holds.

We can further simplify the the computation of F^+ by using the following addition rule

union rule: if α → β holds and α → γ holds, then α → β γ holds.

Decomposition rule: if α → β γ holds then, α → β holds and α → γ holds

Pseudotransitivity rule: if α → β holds and γ β →δ holds then α γ →δ holds
Let us apply our rules to the example of schema R = (A, B, C, G, H, I) and the set F of functional dependencies {A → B, A → C, CG → H, CG → I, B → H}.

We list several members of F^+ here:

A→BC

- Since A→B and A→C , the union rule implies that A→BC

A→H

- Since *A → B* and *B → H* hold, we apply the transitivity rule.

CG → HI

- Since *CG → H* and *CG → I* , the union rule implies that *CG →HI* .

AG → I

- Since A → C and CG → I, the pseudo transitivity rule implies that AG → I holds

Another way of finding that AG → I holds is as follows: We use the augmentation rule on A → C to infer AG → CG. Applying the transitivity rule to this dependency and CG → I, we infer AG → I.

F^+={A → B, A → C, CG → H, CG → I, B → H, A→BC, A→H , *CG → HI,* AG → I}

Closure of Attribute Sets

Closure of attribute set is the set of attributes which are functionally dependent on the attribute set.Let α be a set of attributes. We call the set of all attributes functionally determined by α under a set *F* of functional dependencies the **closure** of under *F*; we denote it by α^+.

Algorithm

result := α;
repeat
for each functional dependency β → γ **in** *F* **do**
begin
if β ⊆ *result* **then** *result* := *result* ∪ γ ;
end
until (*result* does not change)

Here, an algorithm, written in pseudocode to compute α^+. The input is a set *F* of functional dependencies and the set of attributes. The output is stored in the variable *result.*

Let R = (A, B, C, G, H, I) and the
F= {A → B, A → C, CG → H, CG → I, B → H}

Now compute $(AG)^+$

Let result= {AG}

1st Iteration

For A → B

causes us to include B in result. To see this fact, we observe that A → B is in F, A ⊆ result (which is AG), so result := result ∪ B.

∴result={ABG}

For A → C

A ⊆ {ABG} is true

∴result={ABCG}

For CG → H

CG ⊆ {ABCG} is true

∴result={ABCGH}

For CG → I

CG ⊆ {ABCGH} is true

∴result={ABCGHI}

For B → H

B ⊆ {ABCGHI} is true

∴result={ABCGHI}

2nd Iteration

For A → B

A ⊆ {ABCGHI} is true

∴result={ABCGHI}

For A → C

A ⊆ {ABCGHI} is true

∴result={ABCGHI}

For CG → H

CG ⊆ {ABCGHI} is true

∴result={ABCGHI}

For CG → I

CG ⊆ {ABCGHI} is true

∴result={ABCGHI}

For B → H

B ⊆ {ABCGHI} is true

∴result={ABCGHI}

We get the final result ={ABCGHI}

no new attributes are added to *result*, and the algorithm terminates.

∴$(AG)^+$={ABCGHI}

Assignment:

Let R={A,B,C,D,E,F}

F= {A → BC, E → CF, B→ E, CD → EF}

Now compute $(AB)^+$

Procedure for determining the candidate key

Candidate Key:

An attribute or the combination of attributes is called candidate key if and only if:

- They derive all the attributes of the relation.
- They are the minimal subset of the super key.

Note:

- ✓ Candidate keys can be either simple or composite.
- ✓ Minimal subset is not with respect to the no of attributes however it always refer to the minimal level of subset which does not have any proper subsets that derives all the attributes of the relation.
- ✓ A relation can have more than one candidate key.

Determining candidate key

- ✓ Compute closure set of each attributes.
- ✓ Any attribute is called candidate key of any if attribute closure is equal to the relation.
  - ➢ Attribute that are part of candidate key are called **prime attribute.**
  - ➢ Attribute that are not part of candidate key are called **non-prime attribute.**

Example:

R=(A,B,C,D)

F={AB→C, C→D, D→A}

List all the candidate keys, prime and non-prime attributes.

Solution:

For A

A^+ ={A}; A is not candidate key

For B

B^+={B}; B is not candidate key

For C

C^+={CDA}; C is not candidate key

For D

D^+={DA}; D is not candidate key

For AB

$(AB)^+$ ={ABCD}; AB is candidate key

For AC

$(AC)^+$ ={ACD}; AC is not candidate key

For AD

$(AD)^+$ ={AD}; AD is not candidate key

For BC

$(BC)^+$ ={BCDA}; BD is a candidate key

For BD

$(BD)^+$ ={BDAC}; BD is a candidate key

For CD

$(CD)^+$ ={CDA}; CD is not a candidate key

∴ Candidate key={AB,BC,BD}

∴ prime attribute ={A,B,C,D}

Here is no any non-prime attribute.

Anomaly in DBMS

- ✓ Anomaly refer to unexpected or undesirable behaviors that can occur when manipulating or accessing data within a database.
- ✓ These anomaly are typically the result of data inconsistencies, redundancies, or dependencies that can lead to incorrect or illogical results.
- ✓ Anomalies can occur due to various reasons, such as poor database design, incomplete or inconsistent data entry, or improper data manipulation operations.

These anomalies can be categorized into three types:

1. Insertion anomaly
2. Deletion anomaly
3. Updation anomaly

Example

EmployeeID	Name	Address	Department	HOD
1	Ramesh	Kathmandu	Civil	Shyam
2	Anish	Butwal	Computer	Hari
3	Ritu	Pokhara	Civil	Shyam
4	Sita	Chitwan	Computer	Hari
5	Nabin	Pokhara	Computer	Hari

1. **Insertion anomaly**

- ✓ Insertion anomalies occur when it is not possible to add data to a database without including additional, unrelated information.
- ✓ In above example, suppose we you want to add a newly hired employee who hasn't assigned a department yet, we may not be able to insert the employee information into the table due to the missing department information or we will have to set department information NULL.
- ✓ If we insert the 100 employee of same department then the department information will be repeated for all those 100 employees.

2. Updation anomaly

- ✓ Updation anomaly occur when modifying data within a database leads to inconsistent information. This happens when data is duplicated across multiple records, and updates are made to some records but not others, resulting in inconsistent values.
- ✓ In above example, if Hari is no longer HOD of computer department then all the employee records have to be updated and if by mistake we miss any records it will lead to data inconsistency.

3. Deletion anomaly

- ✓ Deletion anomalies occur when removing data from a database results in unintentional loss of other related data.
- ✓ Suppose if employee named Ramesh leaves company and if we want to delete Ramesh information then department information, i.e deparment name civil and HOD Shyam also removed.

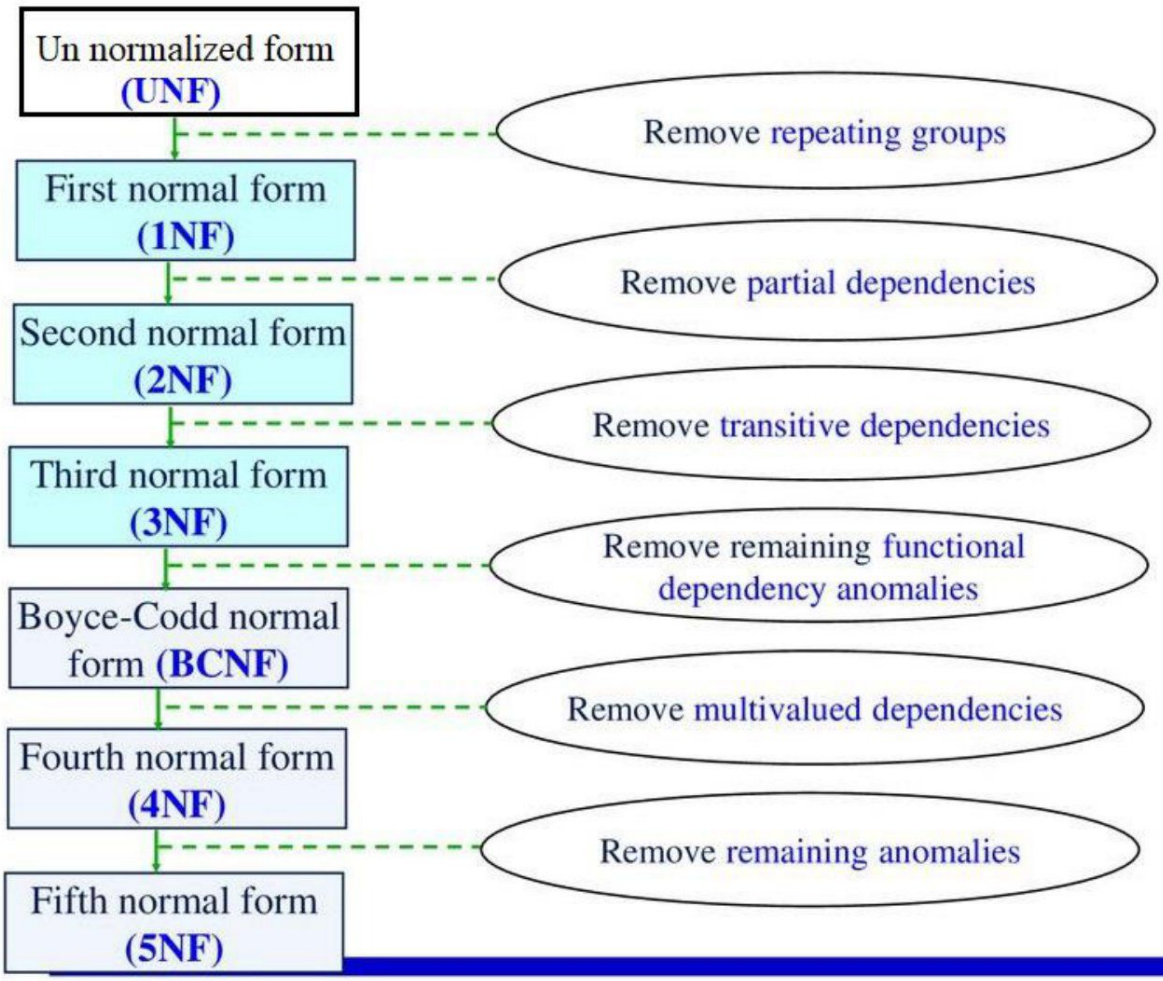
Normalization

- ✓ Database Normalization is a technique of organizing the data in the database.
- ✓ It is a systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics like Insertion, Update and Deletion Anomalies.
- ✓ It is a multi-step process that puts data into tabular form, removing duplicated data from the relation tables.

Advantages of Normalization

- ✓ Normalization ensures data integrity by minimizing data redundancies and inconsistencies in a database.
- ✓ It optimizes data storage, reducing disk space usage and storage costs.
- ✓ Normalized databases tend to have improved performance, especially when executing complex queries.
- ✓ Updates and maintenance become simpler and less prone to errors with a normalized database structure.
- ✓ Normalization provides scalability and flexibility, allowing for easier expansion and modification of the database as it evolves.

Stages of Normalization



Unnormalized form (UNF)

- ✓ A relation is unnormalized when it has not any normalization rules applied to it, and it suffers from various anomalies.
- ✓ It has repeating group.(i.e. it has more than one value for a given cell.

Example:

Emp_id	Emp_name	Project
1	Ram	A,B
2	sita	C
3	Hari	D,E

First Normal Form (1NF)

A relation is said to be in 1NF if and only if

- ✓ Each cell is single valued i.e no repeating groups
- ✓ Values stored in a column should be of the same domain
- ✓ No two rows are identical

Now transforming above relation into 1NF,we get

Emp_id	Emp_name	Project
1	Ram	A
1	Ram	B
2	sita	C
3	Hari	D
3	Hari	E

Second Normal Form (2NF)

A relation is in 2NF if and only if,

- ✓ It is in 1NF and
- ✓ Every non-prime attribute is fully dependent on Primary key.(has no partial dependency)

In other words we can say that in 2NF no non-prime attribute is dependent on the proper subset of any candidate key of the table.

Converting from 1NF to 2NF

1. Identify the primary key for the 1NF relation.
2. Identify functional dependencies in the relation.
3. If partial dependencies exits on the primary key remove them by placing them in a new relation along with copy of their determinant.

Example:

Let us consider the following relation named **Studentcourse**

sid	student_name	course_id	course_name	Grade
1	Ram	1	Java	A
2	Ritu	1	Java	B
1	Ram	2	MERN Stack	B
2	Ritu	3	C ++	C

it has a composite key (Here, the composite key means primary key on two attributes "sid" and "course_id" and non-prime attributes are not fully functionally dependent on the primary key attribute.

For example,

- ✓ The "student_name" depends on "sid" and not depends on "course_id". (referred to a partial dependency)
- ✓ The "course" depends on "course_id" and not depends on "student_id". (referred to a partial dependency)

Now decompose table as follows

student

student_id	student_name
1	Ram
2	Ritu

course

course_id	course_name
1	Java
2	MERN stack
3	C++

Score

student_id	course_id	Grade
1	1	A
2	1	B
1	2	B
2	3	C

Third Normal Form (3NF)

A relation is in 3NF if

- ✓ It is in 2NF.
- ✓ Every non-prime attribute is non-transitively dependent on the primary key. Which means there should not be case that non-prime attribute is functionally dependent on another non-prime attribute.

converting from 2NF to 3NF

1. identify the primary key in the 2NF relation
2. Identify functional dependencies in the relation
3. If transitive dependency exits on the primary key remove them by placing them in a new relation along with copy of their dominant.

Example:

Let us consider the following relation

employee_info

emp_id	emp_name	zip_code	city
1	Ram	501	Kathmandu
2	Hari	502	Pokhara
3	Ritu	501	Kathmandu
4	Gopal	503	Butwal
5	Nisha	502	Pokhara

In above relation emp_id is a primary key. All other attributes are dependent on emp_id, so it is in 2NF but city ,non key attribute is dependent on zip_code, another non-key attributes .So it is not in 3NF.Now decompose above table as below.

Employee table

emp_id	emp_name	zip_code
1	Ram	501
2	Hari	502
3	Ritu	501
4	Gopal	503
5	Nisha	502

Employee_zip table

zip_code	city
501	Kathmandu
502	Pokhara
503	Butwal

Boyce Codd Normal Form(BCNF)

BCNF (Boyce Codd Normal Form) is the advanced version of 3NF which means For BCNF relation already should be in 3NF.

The relation schema *R* is in BCNF with respect to a set *F* of functional dependencies if for all functional dependencies in F^+ of the form

$$\alpha \rightarrow \beta$$

where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$)
- α is a superkey for *R*

In other words, we can also say that a relation is in BCNF if every determinant is a candidate key.

If a relation has only one candidate key then 3NF and BCNF are equivalent.

Example:

Let us assume

- ✓ For each subject ,each student is taught by one teacher
- ✓ Each teacher teaches only one subject
- ✓ Each subject is taught by several teachers

student	course	teacher
Ram	DBMS	Hari
Ram	Microprocessor	Ramesh
Anita	Math	Gita
Bipana	DBMS	Rojan
Laxman	DBMS	Hari

In above relation candidate key can be considered as {student, course}

and following functional dependency exits in above relation

1) {student,course}→teacher

2) teacher→course

In 2) Teacher is not a candidate key but determines course so in above figure is not in BCNF. so we need to decompose it.

Now decompose it as follows

table1

student	teacher
Ram	Hari
Ram	Ramesh
Anita	Gita
Bipana	Rojan
Laxman	Hari

table 2

teacher	course
Hari	DBMS
Ramesh	Microprocessor
Gita	Math
Rojan	DBMS

(Note: As per your revised syllabus you must learn 1NF,2NF,3NF and BCNF) rest of the normal forms are added in this note is added for your knowledge only.)

Fourth Normal Form 4NF

A relation R is in 4NF if

- ✓ it is in BCNF
- ✓ It contains no multivalued dependencies.

Example:

course	teacher	book
DBMS	Ramesh	Database concept
DBMS	Nikita	Fundamental on DBMS
DBMS	Ramesh	Fundamentals on DBMS
DBMS	Nikita	Database concept

Here, course is associated with set of teachers and set of books. which can be represented as

course→→teacher

course→→ book

Now, above relation can be decomposed as

table1

course	teacher
DBMS	Ramesh
DBMS	Nikita

table2

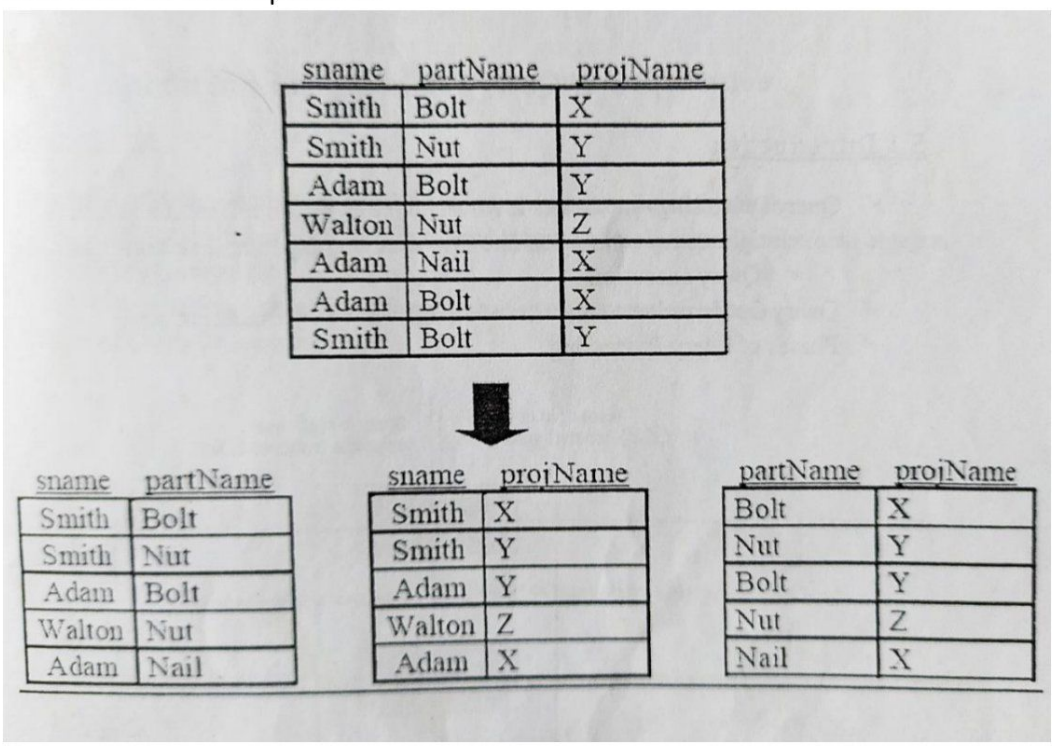
course	book
DBMS	Database concepts
DBMS	Fundamentals on DBMS

Join dependency and 5NF

- ✓ A relation R is subject to a join dependency (JD) if R can always can be recreated by joining multiple tables each having subset of attributes R.
- ✓ Let R be the relation schema and $R_1, R_2, R_3 \ldots R_N$ are projection of R ,
  a legal relation r of R satisfies the joined dependency (JD) if join of the projection of relation on R_i [i=1,2,3,.....N] is equal to
  $R = \prod_{R1}(r) \bowtie \prod_{R2}(r) \bowtie \prod_{R2}(r) \ldots\ldots\ldots\ldots \bowtie \prod_{RN}(r)$
- ✓ A Fifth normal form (5NF) is based on join dependencies.
- ✓ A relation R is in 5NF if and only if it satisfies the following conditions:
  1. R should be already in 4NF.
  2. It cannot be further non loss decomposed (join dependency).
- ✓ A join dependency describes a situation where a relation can be decomposed into two or more smaller relations, and the original relation can be reconstructed by performing a join operation on these smaller relations.

Example: consider the relation supply (sname,partName,projName)

Now it can be decomposed as



sname	partName	projName
Smith	Bolt	X
Smith	Nut	Y
Adam	Bolt	Y
Walton	Nut	Z
Adam	Nail	X
Adam	Bolt	X
Smith	Bolt	Y

sname	partName
Smith	Bolt
Smith	Nut
Adam	Bolt
Walton	Nut
Adam	Nail

sname	projName
Smith	X
Smith	Y
Adam	Y
Walton	Z
Adam	X

partName	projName
Bolt	X
Nut	Y
Bolt	Y
Nut	Z
Nail	X

Denormalization for performance

- ✓ Denormalization is a database optimization technique where we add redundant data in the database to get rid of the complex join operations.
- ✓ This is done to speed up database access speed.
- ✓ Denormalization is done after normalization for improving the performance of the database.
- ✓ The data from one table is included in another table to reduce the number of joins in the query and hence helps in speeding up the performance.

Example: Suppose after normalization we have two tables first, Employee table and second Department table. The Employee has the attributes as employee_id,employee_name,address and department_id.

Employee table

employee_id	employee_name	address	department_id
1	Ramesh	Kathmandu	1
2	Krishna	Pokhara	2
3	Ritu	Butwal	1
4	Roshan	Chitwan	3

The Department table is related to the Student table with department_id as the foreign key in the Student table.

Department table

department_id	department_name	HOD
1	Computer	Shiva
2	Civil	Gaurav
3	IT	Shankar

If we want the name of employees along with the department_name then we need to perform a join operation. The problem here is that if the table is large we need a lot of time to perform the join operations. So, we can add the data of deparment_name from department table to the Employee table and this will help in reducing the time that would have been used in join operation and thus optimize the database.

Employee table

employee_id	employee_name	address	department_id	department_name
1	Ramesh	Kathmandu	1	Computer
2	Krishna	Pokhara	2	Civil
3	Ritu	Butwal	1	Computer
4	Roshan	Chitwan	3	IT

when we use Denormalization?

- ✓ When the redundant data doesn't require to be updated frequently or doesn't update at all. In our example above, the redundant data is department_name and doesn't change frequently.
- ✓ When there is a need to join multiple tables frequently in order to get meaningful data.

Advantages of Denormalization

- ✓ Read Operations are faster as table joins are not required for most of the queries.
- ✓ Write query is easy to write to perform read, write, update operations on database.

Disadvantages of Denormalization

- ✓ Requires more storage as redundant data needs to be written in the tables.
- ✓ Data write operations are slower due to redundant data.
- ✓ Data inconsistencies are present due to redundant data.
- ✓ It requires extra effort to update the database. This is because when redundant data is present, it is important to update the data in all the places else data inconsistencies may arise.

1. **Convert the following 2NF relation into 3NF(Name as Primary key) [PU:2012 fall]**

Name	Address	Phone	Salary	Post
Gill	Ktm	456789	20000	Engineer
Van	Bkt	654321	20000	Engineer
Robert	Ktm	456789	20000	Engineer
Brown	Bkt	654322	10000	Overseer
Albert	Ktm	454545	10000	Officer

solution:

Here, table is in 2NF because it is in 1NF and all attributes fully functionally dependent on the primary key Name.

Now, converting the table to 3NF,a table is in 3NF if and only if

- ✓ It is in 2NF
- ✓ There is no transitive functional dependency. i.e There should not be the case that non-prime attribute is determined by another non-prime attribute.

Here name is a primary key, and here showing that attributes post and salary has transitive functional dependency.

Name→Post

Post→Salary

Here salary is determined by the post even they both are non-prime attribute. So the table is not in 3NF. Now breaking the table.

Table 1

Name	Address	Phone	Post
Gill	Ktm	456789	Engineer
Van	Bkt	654321	Engineer
Robert	Ktm	456789	Engineer
Brown	Bkt	654322	Overseer
Albert	Ktm	454545	Officer

Table 2

Post	Salary
Engineer	20000
Overseer	10000
Officer	10000

Here ,
Table1(Name,Address,Phone,Post)
Table2(Post,Salary)
Both tables are in 3NF because it satisfies the above 3NF criteria.

2. How will you make a given table std_master with attributes: st_id, st_name, instructor_id, inst_name, course_id1, course_name1, course_id2, course_name2, course_id3, course_name3 in 1st ,2nd and 3rd Normal forms, Write the steps. **[PU:2011 fall]**

Solution:

UNF

If the relation is in unnormalized form i.e it contains one or more repeating groups, or each row may contain multiple set of values for some columns.

1NF

To take the table into 1NF following criteria must be satisfied.

- ✓ If there are no repeating values in a column (All entries must be atomic)
- ✓ It must not have repeating columns.

Here, course_id1,course_id2,course_id3 are repeating columns, changing it to only course_id and similarly ,course_name1,course_name2,course_name3 into course_name only then,

the 1NF table relation schema will be

stdmaster(st_id,st_name,instructor_id,inst_name,course_id,course_name)

Now this is in 1NF.

2NF

A relation will be in 2NF if

- ✓ It is in 1NF and
- ✓ All non-prime attributes are fully functional dependent on the key attribute or primary key (There must not be any partial dependency)

But here partial dependency exists. All attributes are not fully functional dependent on primary key.

For example:

s_id →st_name

instructor_id →inst_name

course_id→course_name

Now, to remove these partial dependencies we can decompose the table as follows:

student(st_id,st_name)
instructor(instructor_id,inst_name)
course(course_id,course_name)
teaching_info (st_id, course_id, instructor_id)

3NF

For any relation to be in 3NF it must satisfied following properties.

- ✓ It is in 2NF.
- ✓ Every non-prime attribute is non-transitively dependent on the primary key. Which means there should not be case that non-prime attribute is functionally dependent on another non-prime attribute.

Here, all the above tables are already in 3NF.

Assignment:

Consider the following bank database
Branch_schema=(branch_name,branch_city,assets)
Loan_schema=(loan_number,branch_name,amount)

Write an assertions for bank database to ensure that assets value for koteshwor branch is equal to the sum of all the amounts lent by the koteshwor branch.

Unit 5

Security

Introduction

- ✓ Database security means the protection of data or information from accidental loss, unauthorized access, modification, destruction and unintended activities.
- ✓ Database security is also about controlling access to information i.e. some information be available to anyone, and other information be available to certain authorized people or groups.

Need of Database security

Database security is needed for a database due to the following reasons:

- ❖ Prevention from unauthorized disclosing of information.
- ❖ Prevention from unauthorized modification or destruction of valuable information.
- ❖ Prevention of Unauthorized use of service.
- ❖ Prevention of denial of service

Security and Integrity violations

The data stored in the database needs to be protected from unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency.

Misuse of the database can be categorized as being either intentional (malicious) or accidental.

Accidental loss of data consistency may result from:

- ✓ Crashes during transaction processing
- ✓ Anomalies due to concurrent access to the database
- ✓ Anomalies due to the distribution of data over several computers
- ✓ A logical error that violates the assumption that transactions preserve the database consistency constraints.

It is easier to protect accidental loss of data consistency than to protect against malicious access to the database. Among the forms of malicious access are the following:

- ✓ Unauthorized reading of data (theft of information)
- ✓ Unauthorized modification of data
- ✓ Unauthorized destruction of data

The term database **security** usually refers to the **security from malicious access**, while **integrity** refer to the **avoidance of accidental loss of consistency**. In practice dividing between the security and integrity is not always clear. We shall use the term security to refer security and integrity in cases where distinction between these concept is not essential.

To protect database we must take security measures at several levels.

1. Physical level security

Physical-level security in a database refers to the measures taken to protect the physical infrastructure and resource.

It includes:

- ✓ Protection of equipment from floods, power failure etc.
- ✓ Protection of disk from theft, erase, physical damage etc.
- ✓ Protection of network and terminal cables.

2. Human level security

- ✓ Human-level security in a database refers to the security measures and practices implemented to address the potential risks and vulnerabilities associated with human users and their actions within the database environment.
- ✓ Here are some key aspects of human-level security in a database:
  - ❖ Providing security training and awareness programs for database users
  - ❖ track and record user activities within the database
  - ❖ ensuring strong password policies (e.g., enforcing password complexity, regular password changes)

3. Operating system level security

No matter how secure the database system is, weakness in operating system security may serve as a means of unauthorized access to the database. Since almost all database systems allow remote access through terminals or networks, software-level security within the operation system is as important as physical security.

4. Network level security

- ✓ The database information must be protected from hackers and attack of viruses, leakages of data while being transferred from one computer to other in a network or internet
- ✓ Each site must be ensure that it is communicate with trusted sites (not intruders)

5. Database level security

- ✓ Database system users may be authorized to access only limited portion of the database.
- ✓ Here user may be allowed to issue queries without any modification.
- ✓ Also several views can be utilized as a form of security in the database because it can be used to hides the confidential columns from viewing and manipulation.

Access control

- ✓ Access control mechanism enforces rules who can perform what operation or who can access which data.
- ✓ Alternatively, It is a security policy specifies who is authorized to do what

This access control mechanism must concern with three basic components.

1. Accessor (Subject)

- ✓ A subject is an active element in the security mechanism that operates on the object.
- ✓ A subject is a user who is given some right to access a data object.
- ✓ A subject may be a class of users or even an application program.
- ✓ To provide security to object, identification and authentication of accessor is required.
- ✓ The process of identification may be performed with the help of password, finger print or voice pattern etc.

2. Object to be accessed

- ✓ An object is something that needs protection.
- ✓ A typical object in a database environment could be a unit of data that need to be protected.

Object can be classified as

Data: These are prime candidates for protection. Data object may be file, record, table etc.

Access Path: Access path to be followed for accessing a particular data item or service is an important object by itself in any security mechanism.

Schema: The database schema is another object for protection. Since schema declaration defines access right to different data object, anyone having access to schema declaration can eventually attain access right to different data items also. This is highest level of security.

Views: The views may involve read only facility of the data items and no modification will be permitted for one class of users while other call of user might be able to update view also.

Communication Object: In a distributed database environment, some communication protocols have to be maintained for reliable communication of environment. The communication protocol may include necessary information for the identification and authentication of the sender and receiver.

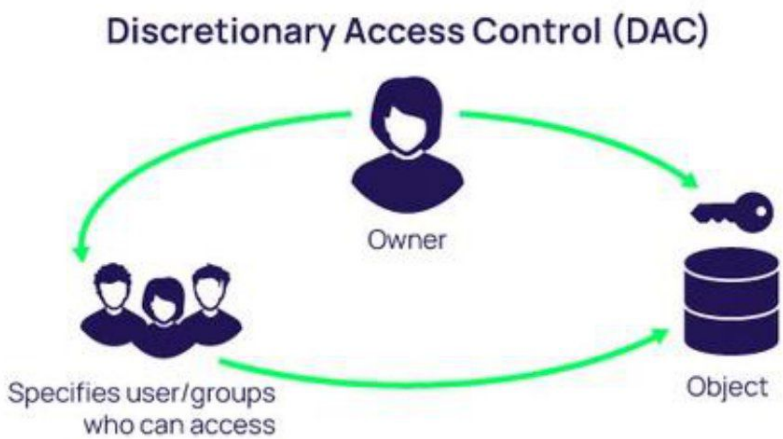
3. Types of Access Control

Once an object is created, the owner may grant the following rights to object to the other authorized users. Read, insert, delete, update, run, create and destroy.

There are following types of access control:

1. Discretionary Access Control (DAC)

- ✓ Discretionary Access Control (DAC) allows each user to control access to their own data.
- ✓ An individual user (object owner) can set an access control mechanism to allow or deny to access the object.
- ✓ This model is called Discretionary because the control of access is based on the discretion of the owner.
- ✓ Each resource object on a DAC based system has an Access Control List (ACL) associated with it. An ACL contains a list of users and groups to which the user has permitted access together with the level of access for each user or group. For example, User A may provide read-only access on one of their files to User B, read and write access on the same file to User C and full control to any user belonging to Group.



2. Mandatory Access Control(MAC)

Mandatory Access Control (MAC) is the strictest of all levels of control. All access to resource objects is strictly controlled by the operating system based on system administrator configured settings. It is not possible under MAC enforcement for users to change the access control of a resource.

Mandatory Access Control begins with security labels assigned to all resource objects on the system. These security labels contain two pieces of information - a classification (top secret, confidential, public etc) and a category (which is essentially an indication of the management level, department etc. to which the object is available).

Similarly, each user account on the system also has classification and category properties from the same set of properties applied to the resource objects.
When a user attempts to access a resource under Mandatory Access Control the operating system checks the user's classification and categories and compares them to the properties of the object's security label.

If the user's credentials match the MAC security label properties of the object access is allowed. It is important to note that both the classification and categories must match.
A user with top secret classification, for example, cannot access a resource if they are not also a member of one of the required categories for that object.

3. Role Based Access Control(RBAC)

Role Based Access Control assigns permissions to particular roles in an organization. Users are then assigned to that particular role. For example, an accountant in a company will be assigned to the Accountant role, gaining access to all the resources permitted for all accountants on the system. Similarly, a software engineer might be assigned to the developer role.

Roles differ from groups in that while users may belong to multiple groups, a user under RBAC may only be assigned a single role in an organization. Additionally, there is no way to provide individual users additional permissions over and above those available for their role. The accountant described above gets the same permissions as all other accountants, nothing more and nothing less.

Authorization

- ✓ Authorization is a security mechanism used to determine user/client privilege or access level related to system resources.
- ✓ In multiuser database system, a system administrator defines for the system which users are allowed access to the system and what privilege of use.
- ✓ During authorization system verifies authenticated users access role and either grant or revoke resource access.

Thus, Authorization includes:

- ✓ Permitting only certain users to access process or alter data.
- ✓ Applying different limitations on user access or actions .Here limitations placed on users can apply to object such as tables, rows etc.

A user may have several form of authorization on parts of database.

- ✓ Authorization to read data
- ✓ Authorization to insert new data
- ✓ Authorization to update data
- ✓ Authorization to delete data

Each of these type of authorization is called privilege. A database user may be assigned all, none, or combination of these types of privileges on specified parts of database such as relation or views. In addition to authorization data, users may be granted database schema, allowing them to create modify or drop relations. The ultimate form of authority is that given to DBA (Database Administrator) DBA may authorize new users, reconstructed the database etc.

Granting and Revoking of Privileges

- ✓ The SQL standard includes the privileges select, insert, update, and delete. The privilege **all privileges** can be used as a short form for all the allowable privileges.
- ✓ A user who creates a new relation is given all privileges on that relation automatically.
- ✓ The SQL data-definition language includes commands to grant and revoke privileges.

Grant statement

The grant statement is used to confer authorization.

The basic form of this statement is:

```
grant <privilege list>
on <relation name or view name>
to <user>;
```

The privilege list allows the granting of several privileges in one command

Let us consider the following relation

department(dept_name,building,budget)

❖ **select authorization**

✓ The select authorization on a relation is required to read tuples in the relation.

The following grant statement grants database users Amit and Satoshi select authorization on the department relation:

Example:
grant select on department to Amit, Santoshi;

This allows those users to run queries on the department relation

❖ **update authorization**

✓ The update authorization on a relation allows a user to update any tuple in the relation.
✓ The update authorization may be given either on all attributes of the relation or on only some.
✓ If update authorization is included in a grant statement, the list of attributes on which update authorization is to be granted optionally appears in parentheses immediately after the update keyword.
✓ If the list of attributes is omitted, the update privilege will be granted on all attributes of the relation.
This grant statement gives users Amit and Satoshi update authorization on the budget attribute of the department relation:

Example:
grant update (budget) on department to Amit, Santoshi;

❖ **insert authorization**

The insert authorization on a relation allows a user to insert tuples into therelation. The insert privilege may also specify a list of attributes; any inserts tothe relation must specify only these attributes, and the system either gives each of the remaining attributes default values (if a default is defined for the attribute)or sets them to null.

Example:
grant insert on department to Amit, Satoshi;

grant insert(dept_name,building) on department to Amit, Santoshi;

❖ **delete authorization**

The delete authorization on a relation allows a user to delete tuples from a relation

grant delete on department to Ram,Hari;

Revoke statement

To revoke an authorization use revoke statement.

revoke<privilege list> on <relation or view> from<user>

Example:
revoke select on department from Amit, Santoshi;

revoke update (budget) on department from Amit, Santoshi;

Authentication

- ✓ Authorization is the process to confirm what you are authorized to perform but authentication confirms who you are.
- ✓ So the primary goal of authentication system is to allow access to the legal system users and deny access to unauthorized users.

The most widely used authentication techniques are:

1) Password based authentication: A password is a secret word or string of characters used for user authentication to prove identity to a resource, which should be kept secret from those are not allowed access. It is not much reliable than other authentication system because if a weak password is chosen then it can be easily guessed.

2) Artifact based authentication

It includes machine-readable batches and electronic cards. These cards consist of magnetic strip, which represents the unique identification number. Card reader may be installed in or near the terminal and users are required to supply artifact for authentication. This form of authentication is common in ATMs in bank. Some companies also provide cards to their employee for authentication.

3) Biometric Technique

In this technique, the major groups of authentication mechanism are based on the unique characteristic of each user. This falls into two basic categories.

- ✓ **Physiological characteristics:** characteristics such as finger prints, facial characteristics, retina characteristics etc.
- ✓ **Behavioral characteristics:** characteristics such as voice pattern, signature pattern etc.

Security and views

The concept of views is a means of providing a user with a "personalized "model of a database. A view can hide data that user does not need to see. The ability of views to hide data serves both to simplify usage of system and to enhance security.

In SQL, a view is a virtual table based on the result-set of an SQL statement.

- ✓ A view contains rows and columns, just like a real table.
- ✓ The fields in a view are fields from one or more real tables in the database.
- ✓ Through a view, users can query and modify only the data they can see. The rest of the database is neither visible nor accessible.

A view is created with the CREATE VIEW statement.

Syntax

```
CREATE VIEW view_name AS

SELECT column1, column2, ...

FROM table_name

WHERE condition;
```

Benefits of using views

Data Security: Views can be used to enforce data security by limiting the access to sensitive information. By creating views that only expose certain columns or rows, we can control what data users can see and ensure that confidential or restricted information remains hidden.

borrower (customer-name, loan-number)

loan (loan-number, branch-name, amount)

Suppose a bank clerk needs to know the names of the customers of each branch but it is not authorized a specific loan information.

Approach: Deny direct access to the loan relation but grant access to the view cust_loan which consists of only the names of customers and the branches at which they have a loan.

The cust_loan view is defined in SQL as follows

```
CREATE VIEW cust_loan AS
SELECT  borrower.customer_name,loan.branch_name
FROM borrower,loan
WHERE borrower.loan_numer=loan.loan_number;
```

How views differ from relation?

Relation (table)	views
A relation is used to organize data in the form rows and columns and displayed them in structured format.	views are treated as virtual /logical table used to view or manipulate parts of the table.
It is a physical entity that means data is actually stored in the relation.	A view is virtual entity which means data is not actually stored in table.
It occupies space in the system.	A view does not occupy physical space on the system.
It is an independent data object.	It depends on the table(relation) .we cannot create a view without using table.
In the table, we can maintain relationships using a primary and foreign key.	The view contains complex multiple tables joins
It generates a fast result.	View generates a slow result because it renders the table every time we query it.
syntax: CREATE TABLE table_name (column1 datatype, column2 datatype, column3 datatype, ...);	**syntax:** CREATE VIEW view_name AS SELECT column_name_1, column_name_2,... FROM table_name WHERE condition;
Example: Let us consider the following relation **EMPLOYEE(Emp_No ,Name ,Skill ,Sal_Rate ,Address)** Now, it can be created as : **Create table EMPLOYEE** **(Emp_No int PRIMARY KEY,** **Name varchar (30),** **Skill varchar (30),** **Sal_Rate decimal (10, 2),** **Address varchar (30));**	Example: Let us consider the following relation **EMPLOYEE(Emp_No ,Name ,Skill ,Sal_Rate ,Address)** For a very personal or confidential matter, every user is not permitted to see the Sal_Rate of an EMPLOYEE. For such users, DBA can create a view, for example, EMP_VIEW defined as: **Create view EMP_VIEW as** **SELECT Emp_No, Name, Skill, Address** **From EMPLOYEE;**

Unit 6

Query Processing and Optimization

Introduction

Query processing refers to the range of activities involved in extracting data from database.

The activities includes:

- translation of queries in high-level database languages into expressions that can be used at the physical level of the file system
- a variety of query-optimizing transformations
- actual evaluation of queries

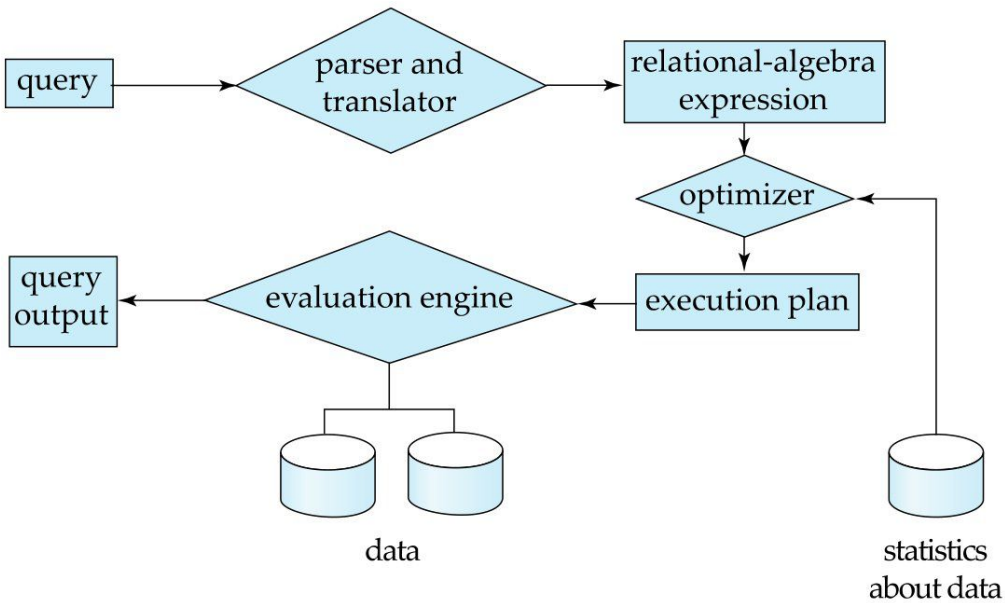
The steps involved in processing a query appear in Figure below.

The basic steps are:

1. Parsing and translation

2. Optimization

3. Evaluation



1. Parsing and translation

- ✓ In this step query is translated into its internal form. This translation process is similar to the work performed by the parser of the compiler.
- ✓ In generating the internal form of the query, the parser checks the syntax of the user's query, verifies that the relation names appearing in the query are names of the relations in the database, and so on.
- ✓ The system constructs a parse-tree representation of the query, which it then translates into a relational-algebra expression.

2. Optimization

A relational algebra expression may have many equivalent expressions.

As an illustration, consider the query:
select *salary*
from *instructor*
where *salary* < 75000;

This query can be translated into either of the following relational-algebra expressions

- $\sigma_{salary<75000}(\prod_{salary}(instructor))$

is equivalent to

- $\prod_{salary}(\sigma_{salary<75000}(instructor))$

Relational algebra expression annotate with instructions specifying how to evaluate each operation which may state the algorithm to be used for a specific operation, or the particular index or indices to use.

A relational algebra operation annotated with instructions on how to evaluate it is called an **evaluation primitive.**

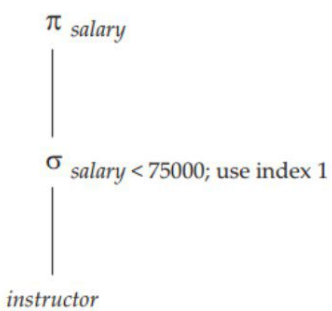


Figure: A query-evaluation plan.

A sequence of primitive operations that can be used to evaluate a query is a query-evaluation plan. The process of choosing the evaluation plan with the lowest cost amongst all equivalent evaluation plans is known as query optimization. The cost is estimated using the statistical information from the database catalog. The different statistical information is number of tuples in each relation, tuple size etc.

3. Evaluation

The **query-execution** engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.

Measures of query cost

There are multiple possible evaluation plans for a query, and it is important to be able to compare the alternatives in terms of their (estimated) cost, and choose the best plan. To do so, we must estimate the cost of individual operations, and combine them to get the cost of a query evaluation plan.
The cost of query evaluation can be measured in terms of a number of different resources, including:

- disk accesses
- CPU time to execute a query
- a distributed or parallel database system, the cost of communication

In large database systems, the cost to access data from disk is usually the most important cost, since disk accesses are slow compared to in-memory operations. Moreover, CPU speeds have been improving much faster than have disk speeds. Thus, it is likely that the time spent in disk activity will continue to dominate the total time to execute a query.

Although real-life query optimizers do take CPU costs into account, for simplicity here we ignore CPU costs and **use only disk-access costs to measure the cost of a query-evaluation plan**.

- We use the **number of block transfers** *from disk and the* **number of disk seeks** to estimate the cost of query evaluation plan.
- If a disk subsystem takes an average of t_T seconds to transfer the block of data and has average block access time of t_S seconds then an operation that transfers b blocks and performs S seeks would take b* t_T +S* t_S seconds.
- The values of t_T and t_S must be calibrated for the disk system used, but typical values for high end disks would be t_S = 4 milliseconds and t_T = 0.1 milliseconds assuming a 4 kilobyte size and a transfer rate of 40 megabytes per seconds;

Query optimization

Query optimization is the process of selecting the most efficient query-evaluation plan from among the many strategies usually possible for processing a given query, especially if the query is complex.

Let us consider the following relation:

depositor(customer_name,account_number)
account(account_number,branch_name,balance)
branch(branch_name,branch_city,assets)
Now, Consider the following relational-algebra expression, for the query "**Find the names of all customers who have an account at some branch located in kathmandu"**

$\Pi_{customer\_name}(\sigma_{branch\_city= \text{"kathmandu"}}$ (depositer ⋈ (account ⋈branch)))

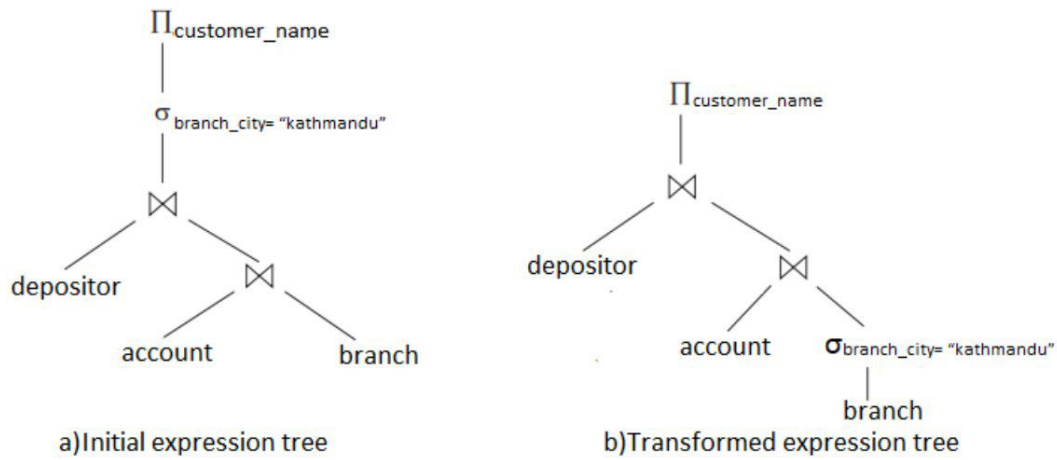
The above expression constructs a large intermediate relation. However we are interested only few records of this relation.(i.e. branch city is kathmandu)

By reducing the number of tuples in branch relation that we need access we reduce the size of intermediate result.

Our query is now represented by:

$\Pi_{customer\_name}$(depositer ⋈ (account ⋈ $\sigma_{branch\_city= \text{“kathmandu”}}$ (branch)))

which is equivalent to previous relational algebra expression, but which generates the smaller intermediate relations. Figure below depicts the initial and transformed expressions.



An evaluation plan defines exactly what algorithm should be used for each operation, and how the execution of the operations should be coordinated.

For a given a **relational-algebra expression**, it is the job of the query optimizer to come up with a query-evaluation plan that computes the same result as the given expression, and is the least-costly way of generating the result .

To find the least-costly query-evaluation plan, the optimizer needs to generate alternative plans that produce the same result as the given expression, and to choose the least-costly one.

Generation of query-evaluation plans involves three steps:

(1) generating expressions that are logically equivalent to the given expression

(2) annotating the resultant expressions in alternative ways to generate alternative query-evaluation plans

(3) estimating the cost of each evaluation plan, and choosing the one whose estimated cost is the least.

- To implement the first step, the query optimizer must generate expressions equivalent to a given expression by using equivalence rules.
- We must estimate the statistics of results of each operation in a query plan.

For example:

- **catalog information (***The database-system catalog stores the following statistical information about database relations such as the number of tuples in a relation r, number of blocks containing the relation r , the size of a tuple of relation r in bytes)*
- **size of joins estimation**
- **selection size estimation**

Using this statistics cost formulae allows us to estimate the cost of each individual operations. The individual cost is combined to determine the estimated cost of evaluating the given relational algebra expressions.

- Now we can choose query evaluation plan based on the estimated cost of the plans with is likely to be the least costly one or not more much more costly than it.

Equivalence of expressions

- ✓ Any two relational expressions are said to be equivalent if resulting relation generates the same set of tuples.
- ✓ When two expressions are equivalent we can use them interchangeably i.e. we can use either of the expressions which gives the better performance.

❖ Equivalence rule

- ✓ An equivalence rule says that expressions of two forms are equivalent.
- ✓ We can replace an expression of the first form by an expression of the second form, or vice versa i.e., we can replace an expression of the second form by an expression of the first form, since the two expressions generate the same result on any valid database.
- ✓ The optimizer uses equivalence rules to transform expressions into other logically equivalent expressions.
- ✓ We now list a number of general equivalence rules on relational-algebra expressions. We use θ1, θ2, θ3, and so on to denote predicates, L1, L2, L3, and so on to denote lists of attributes, and E1, E2, E3, and so on to denote relational-algebra expressions. A relation name r is simply a special case of a relational-algebra expression, and can be used wherever E appears.

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections

$$\sigma_{\theta 1 \wedge \theta 2}(E) \equiv \sigma_{\theta 1}(\sigma_{\theta 2}(E))$$

2. Selection operations are commutative:

$$\sigma_{\theta 1}(\sigma_{\theta 2}(E)) \equiv \sigma_{\theta 2}(\sigma_{\theta 1}(E))$$

3. Only the final operations in a sequence of projection operations is needed, the others can be omitted

$\prod_{L1}(\prod_{L2}(\ldots(\prod_{Ln}(E))\ldots)) \equiv \prod_{L1}(E)$
where $L_1 \subseteq L_2 \ldots \subseteq L_n$

4. Selections can be combined with Cartesian products and theta joins:
   a. $\sigma_\theta(E_1 \times E_2) \equiv E_1 \bowtie_\theta E_2$
   b. $\sigma_{\theta 1}(E_1 \bowtie_{\theta 2} E_2) \equiv E_1 \bowtie_{\theta 1 \wedge \theta 2} E_2$

5. Theta join operations are commutative:

$E1 \bowtie_\theta E2 \equiv E2 \bowtie_\theta E1$

6. (a) Natural join operations are associative:

$(E_1 \bowtie E_2) \bowtie E_3 \equiv E_1 \bowtie (E_2 \bowtie E_3)$

(b) Theta joins are associative in the following manner:

$(E_1 \bowtie_{\theta 1} E_2) \bowtie_{\theta 2 \wedge \theta 3} E_3 \equiv E_1 \bowtie_{\theta 1 \wedge \theta 3} (E_2 \bowtie_{\theta 2} E_3)$ where θ_2 involves attributes from only E_2 and E_3.

7. The selection operation distributes over the theta join operation under the following two conditions:

(a) When all the attributes in θ_0 involve only the attributes of one of the expressions (E_1) being joined.

$\sigma_{\theta 0}(E_1 \bowtie_\theta E_2) \equiv (\sigma_{\theta 0}(E_1)) \bowtie_\theta E_2$

(b) When θ_1 involves only the attributes of E_1 and θ_2 involves only the attributes of E_2.

$\sigma_{\theta 1 \wedge \theta 2}(E_1 \bowtie_\theta E_2) \equiv (\sigma_{\theta 1}(E_1)) \bowtie_\theta (\sigma_{\theta 2}(E_2))$

8. The projection operation distributes over the theta join operation as follows:

(a) if θ involves only attributes from $L_1 \cup L_2$:

$\prod_{L1 \cup L2}(E_1 \bowtie_\theta E_2) \equiv \prod_{L1}(E_1) \bowtie_\theta \prod_{L2}(E_2)$

(b) In general, consider a join $E_1 \bowtie_\theta E_2$.

- Let L_1 and L_2 be sets of attributes from E_1 and E_2, respectively.
- Let L_3 be attributes of E_1 that are involved in join condition θ, but are not in $L_1 \cup L_2$, and
- let L_4 be attributes of E_2 that are involved in join condition θ, but are not in $L_1 \cup L_2$.

$\prod_{L1 \cup L2}(E_1 \bowtie_\theta E_2) \equiv \prod_{L1 \cup L2}(\prod_{L1 \cup L3}(E_1) \bowtie_\theta \prod_{L2 \cup L4}(E_2))$

9. The set operations union and intersection are commutative.

$E_1 \cup E_2 \equiv E_2 \cup E_1$
$E_1 \cap E_2 \equiv E_2 \cap E_1$
(Set difference is not commutative)

10. Set union and intersection are associative.

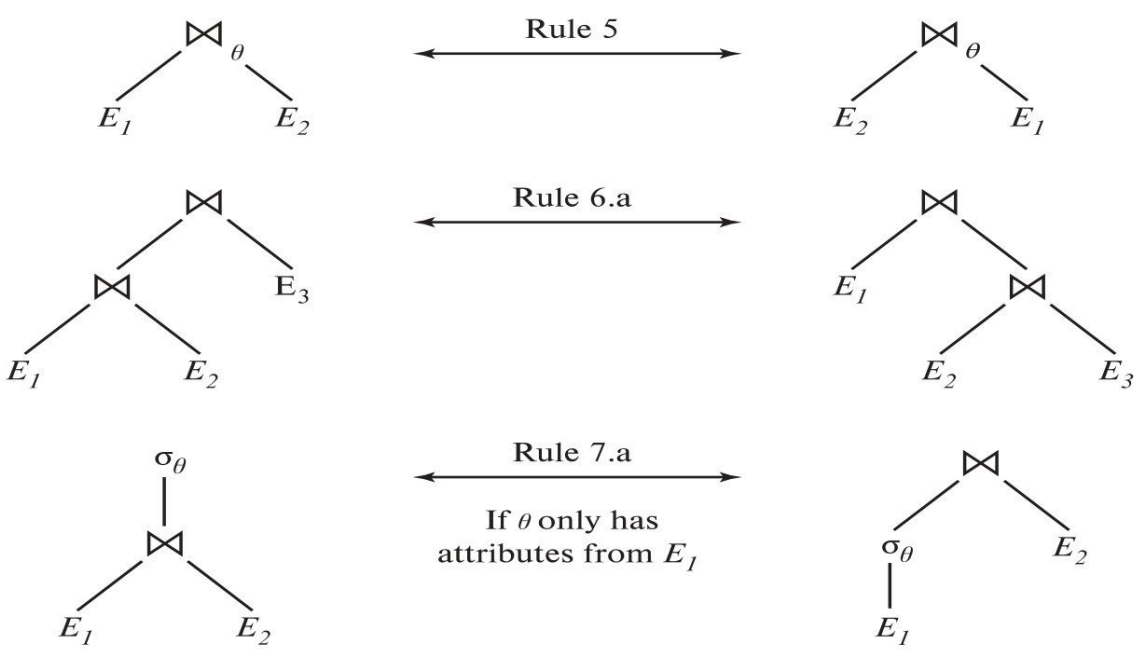
$(E_1 \cup E_2) \cup E_3 \equiv E_1 \cup (E_2 \cup E_3)$
$(E_1 \cap E_2) \cap E_3 \equiv E_1 \cap (E_2 \cap E_3)$

11. The selection operation distributes over the union, intersection, and set-difference operations.

a. $\sigma_\theta (E_1 \cup E_2) \equiv \sigma_\theta (E_1) \cup \sigma_\theta(E_2)$
b. $\sigma_\theta (E_1 \cap E_2) \equiv \sigma_\theta (E_1) \cap \sigma_\theta(E_2)$
c. $\sigma_\theta (E_1 - E_2) \equiv \sigma_\theta (E_1) - \sigma_\theta(E_2)$
d. $\sigma_\theta (E_1 \cap E_2) \equiv \sigma_\theta(E_1) \cap E_2$
e. $\sigma_\theta (E_1 - E_2) \equiv \sigma_\theta(E_1) - E_2$

12. The projection operation distributes over the union operation.
$\Pi_L(E_1 \cup E_2) \equiv (\Pi_L(E_1)) \cup (\Pi_L(E_2))$

Pictorial depiction of Equivalence rules



Example of transformations:

Let us consider the following relation:

depositor(customer_name,account_number)
account(account_number,branch_name,balance)
branch(branch_name,branch_city,assets)

Example 1:

Query:

Find the name of all customers who have an account at some branch located in Kathmandu.

$\Pi_{customer\_name}(\sigma_{branch\_city= \text{“kathmandu”}}$ (branch ⋈ (account ⋈depositor)))

Transformation using rule 7a.

$\Pi_{customer\_name}(\sigma_{branch\_city= \text{“kathmandu”}}$ (branch) ⋈ (account ⋈depositor))

Performing the selection as early as possible reduces the size of the relation to be joined.

Example 2:

Query:

Find the name of all customers with an account at a kathmandu city whose account balance is over 1000.

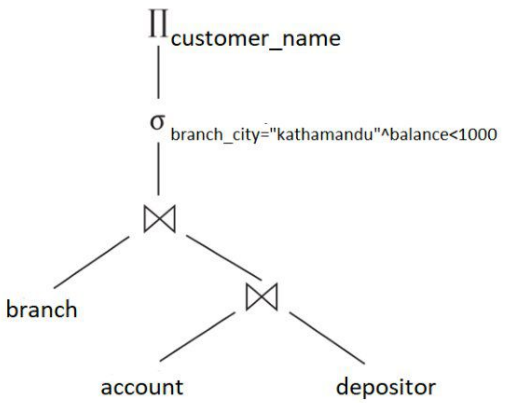
$\Pi_{customer\_name}(\sigma_{branch\_city= \text{“kathmandu”} \wedge balance>1000}$ (branch ⋈ (account ⋈depositor)))

Transformation using join associativity (Rule 6a)

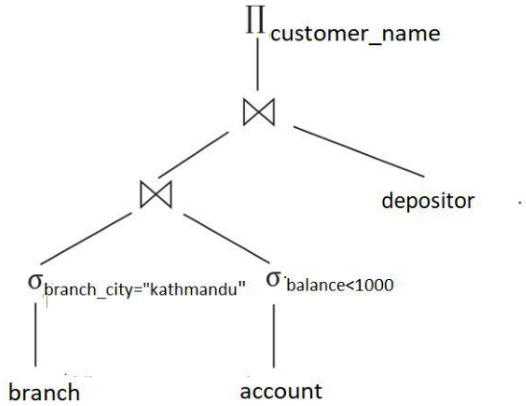
$\Pi_{customer\_name}((\sigma_{branch\_city= \text{“kathmandu”} \wedge balance>1000}$ (branch ⋈ account)) ⋈depositor)

second form provides an opportunity to apply the “perform selection early” rule, resulting in the subexpression

$\sigma_{branch\_city= \text{“kathmandu”}}$ (branch) ⋈ $\sigma_{balance>1000}$ (account)



a) Initial expression tree



b) Tree after multiple transformations

Evaluation of Expression

Expression cannot exist as a single operation instead typical query combines them.

For example:

```
select customer_name
 from account natural  join customer
 where balance <30000;
```

This example consist of selection, natural join, and a projection.

To evaluate an expression that carries multiple operations in it, computation of each operation is performed one by one.

In the query processing, two methods are used for evaluating an expression that carries multiple operations. These methods are:

1. Materialization
2. Pipelining

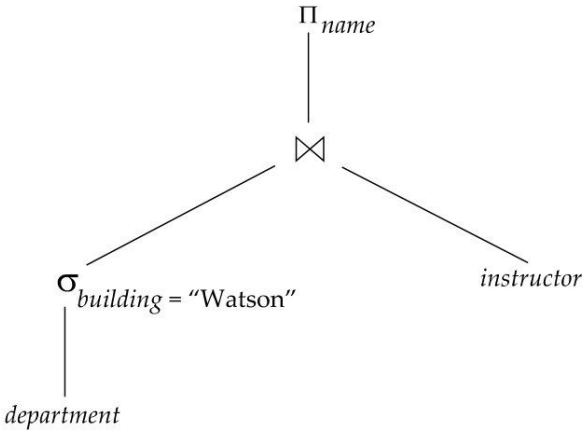
1. Materialization

In materialization, an expression is evaluated one operation at a time in an appropriate order. The result of each evaluation is **materialized** in a temporary relation for subsequent use.

A disadvantage to this approach is the need to construct the temporary relations, which (unless they are small) must be written to disk.

Consider the expression:

$\prod_{name}(\sigma_{building= \text{“watson”}}(department)) \bowtie instructor)$



If we apply the materialization approach, we start from the lowest-level operations in the expression (at the bottom of the tree). In our example, there is only one such operation: the selection operation on *department*. The inputs to the lowest-level operations are relations in the database. We execute these operations and we store the results in temporary relations. We can use these temporary relations to execute the operations at the next level up in the tree, where the inputs now are either temporary relations or relations stored in the database.

In our example, the inputs to the join are the *instructor* relation and the temporary relation created by the selection on *department*. The join can now be evaluated, creating another temporary relation.

2. Pipelining

- ✓ **Pipelined evaluation**: evaluate several operations simultaneously, passing the results of one operation on to the next.
- ✓ For example, as in previous expression tree, don't store result of instead, pass tuples directly to the join. Similarly, don't store result of join, pass tuples directly to projection.

  Creating a pipeline of operations can provide two benefits:

  1. It eliminates the cost of reading and writing temporary relations, reducing the cost of query evaluation.
  2. It can start generating query results quickly, if the root operator of a query evaluation plan is combined in a pipeline with its inputs. This can be quite useful if the results are displayed to a user as they are generated, since otherwise there may be a long delay before the user sees any query results.

Pipelines can be executed in either of two ways:

1. **In a demand-driven pipeline or lazy evaluation**, the system makes repeated requests for tuples from the operation at the top of the pipeline. Each time that an operation receives a request for tuples, it computes the next tuple (or tuples) to be returned, and then returns that tuple.

2. **In a producer-driven or eager pipelining,** operations do not wait for requests to produce tuples, but instead generate the tuples eagerly.

Choice of evaluation plan

An evaluation plan defines exactly what algorithm should be used for each operation, and how the execution of the operations should be coordinated.

For a given an evaluation plan, we can estimate its cost using statistics estimated by the techniques coupled with cost estimates for various algorithms and evaluation methods.

Approaches to choose the best evaluation plan for a query are as follows:

- Search all the plans and choose the best plan on the cost based fashions (cost based optimizer)
- Use heuristics to choose the plan

Cost based optimizer

- ✓ A cost-based optimizer explores the space of all query-evaluation plans that are equivalent to the given query, and chooses the one with the least estimated cost.
- ✓ Equivalence rules can be used to generate equivalent plans.
- ✓ Exploring the space of all possible plans may be too expensive for complex queries.

Consider the problem of choosing the optimal join order for such a query. For a complex join query, the number of different query plans that are equivalent to the query can be large.

As an illustration, consider the expression:

$$r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n$$

where the joins are expressed without any ordering. With n = 3, there are 12 different join orderings:

r1 ⋈ (r2 ⋈ r3)	r1 ⋈ (r3 ⋈ r2)	(r2 ⋈ r3) ⋈ r1	(r3 ⋈ r2) ⋈ r1
r2 ⋈ (r1 ⋈ r3)	r2 ⋈ (r3 ⋈r1)	(r1 ⋈ r3) ⋈r2	r3 ⋈ (r1 ⋈ r2)
r3 ⋈ (r2 ⋈ r1)	(r1 ⋈ r2) ⋈ r3	(r3 ⋈ r1) ⋈ r2	(r2 ⋈ r1) ⋈ r3

In general, with *n* relations, there are (2(*n* - 1))!/(*n* - 1)! different join orders. For joins involving small numbers of relations, this number is acceptable; for example, with *n* = 5, the number is 1680. However, as *n* increases, this number rises quickly.

- With *n* = 7, the number is 665,280
- with *n* = 10, the number is greater than 17.6 billion!

It is not necessary to generate all the expressions equivalent to a given expression. For example, suppose we want to find the best join order of the form:
(*r*1⋈ *r*2 ⋈ *r*3) ⋈*r*4 ⋈ *r*5
which represents all join orders where *r*1,*r*2, and *r*3 are joined first (in some order), and the result is joined (in some order) with *r*4 and *r*5. There are 12 different join

orders for computing $r1 \bowtie r2 \bowtie r3$, and 12 orders for computing the join of this result with $r4$ and $r5$. Thus, there appear to be 144 join orders to examine. However, once we have found the best join order for the subset of relations $\{r1, r2, r3\}$, we can use that order for further joins with $r4$ and $r5$, and can ignore all costlier join orders of $r1 \bowtie r2 \bowtie r3$. Thus, instead of 144 choices to examine, we need to examine only 12 + 12 choices.
Using this idea, we can develop a *dynamic-programming* algorithm for finding optimal join orders. Dynamic-programming algorithms store results of computations and reuse them, a procedure that can reduce execution time greatly.

With dynamic programming time complexity of optimization with bushy trees is $O(3^n)$.

- ✓ With $n = 10$, this number is 59000 instead of 176 billion!

Space complexity is $O(2^n)$

Heuristics in Optimization

Cost-based optimization is expensive, even with dynamic programming. Systems may use heuristics to reduce the number of choices that must be made in a cost-based fashion. Heuristic optimization transforms the query-tree by using a set of rules that typically (but not in all cases) improve execution performance:

- Perform selection early (reduces the number of tuples)
- Perform projection early (reduces the number of attributes)
- Perform most restrictive selection and join operations (i.e., with smallest result size) before other similar operations.
- Some systems use only heuristics, others combine heuristics with partial cost-based optimization.

Unit 7

Storage Management and Indexing

File

- ✓ A collection of data or information that has a name, called the filename. Almost all information stored in a computer must be in a file. There are many different types of files: data files, text files, program files, directory files, and so on.
- ✓ The physical or internal level of organization of a database system is concerned with the efficient storage of information in the secondary storage device.
- ✓ The basic problem in the physical database representation is to select suitable file system to store the desired information.

File organization

- ✓ A database is stored as collection of files. Each file is stored as a sequence of records. A record is a sequence of fields. Records are stored on disk blocks.
- ✓ A file organization essentially means organization of records in the file.
- ✓ It is a logical relationship among various records. This method defines how file records are mapped to the disk blocks.

Purpose of file organization

- ✓ File organization makes it easier & faster to perform operations (such as read, write, update, delete etc.) on data stored in files.
- ✓ **Removes data redundancy:** File organization make sure that the redundant and duplicate data gets removed. This alone saves the database from insert, update, delete operation errors which usually happen when duplicate data is present in database.
- ✓ **Save storage cost**: By organizing the data, the redundant data gets removed, which lowers the storage space required to store the data.
- ✓ **Improves accuracy:** When redundant data gets removed and the data is stored in efficient manner, the chances of data gets wrong and corrupted will be minimized.

A file can have mainly two types of records

1. Fixed Length Records
2. Variable Length Records

1. Fixed length records

Consider a records of file deposit of the form

type account=record
account_number:char(10);
branch_name:char(22);
balance:real;
end

If we assume that each character occupies one byte and a real 8 bytes, our deposit record is 40 bytes long. The simplest approach is to use the first 40 bytes for the first record, the next 40 bytes for the second, and so on.

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

However, there are two problems with this approach.

1. Unless the block size happens to be a multiple of 40 (which is unlikely), some records will cross block boundaries. That is, part of the record will be stored in one block and part in another. It would thus require two block accesses to read or write such a record.

   Solution: Allocate only as many as records to a block as would fit entirely in the block. This can be computed by
   Number of records in particular block=block size/record size
   And discarding the fractional part. Any remaining bytes of each block are left unused.

2. It is difficult to delete a record from this structure. The space occupied by the record to be deleted must be filled with some other record of the file, or we must have a way of marking deleted records so that they can be ignored.

When the record is deleted, we could move the record that came after it into the space formerly occupied by the deleted record and so on, until every record following the deleted record has been moved ahead. Such approach requires moving large number of records.

Record 2 deleted and all records moved

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

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
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

It might be easier simply to move the final record of the file into the space occupied by the deleted record.

Record 2 deleted and final record moved

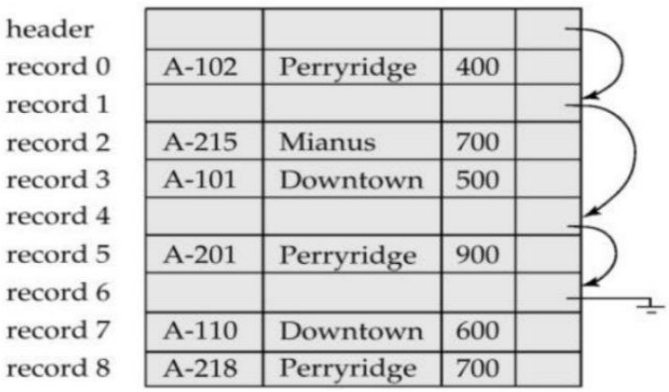
record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 2	A-215	Mianus	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 8	A-218	Perryridge	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

Thus, to avoid above problems we need to introduce an additional structure,

At the beginning of the file, we allocate a certain number of bytes as the file header. The header will contain variety of information about the file. For now, all we need to store there is the address of first record whose contents are deleted.

We use this first record to store the address of second available record and so on. We can think of these stored address as pointers, since they point to the location of a record. Thus deleted record thus form a linked list, which is often referred to as free list. The figure below the file of instructor, with the free list, after records 1,4 and 6 have been deleted.



On insertion of a new record, we use the record pointed to by the header. We change the header pointer to point to the next available record. If no space is available we add the new records to the end of the file.

Insertion and deletion for files of fixed-length records are simple because the space the space made available by a deleted record is exactly the spaced needed to insert the record.

2. Variable–Length records

Variable-length records arise in database systems in several ways:

- ✓ Storage of multiple record types in a file.
- ✓ Record types that allow variable lengths for one or more fields such as strings (varchar)
- ✓ Record types that allow repeating fields ,such as arrays.

One example with variable -length records

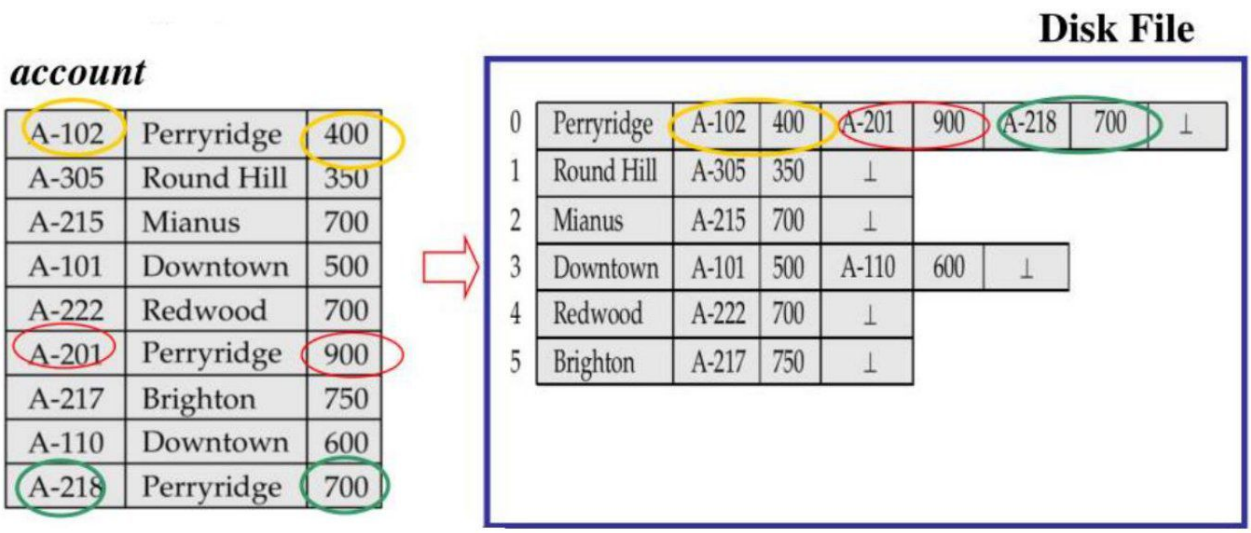
```
type account_list=record
branch_name:char(22);
account_info: array[1.....∞] of record;
                    account_number:char(10)
                    balance:real;
                    end
end
```

In this we define a account_info as an array with an arbitrary number of elements. i.e. the type definition does not limit the number of elements in the array.

Representation Methods

i) Byte String Representation

In byte string representation we attach a special end of record (⊥)



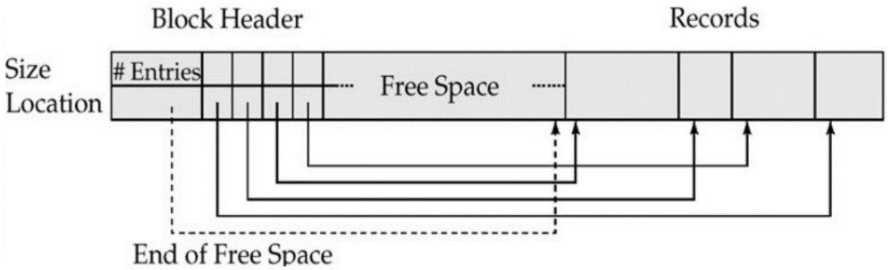
This approach has several disadvantages:

- ✓ It is not easy to reuse the space left by a deleted record
- ✓ In general there is no space for records to grow longer. If the records become longer it must be removed.

A modified form of byte–string representation called slotted page structure.

The slotted page structure there is a header at the beginning of each block, containing the following information.

- ✓ The number of record entries in the header
- ✓ The end of free space in the block
- ✓ An array whose entries contain the location and size of each record



- ✓ The actual records are allocated contiguously in the block, starting from the end of the block.
- ✓ The free space in the block is contiguous, between the final entry in the header array, and the first record.
- ✓ If the records is inserted, space is allocated for it at the end of the free space and the entry containing its size and location is added to the header.
- ✓ If the record is deleted, the space is that it occupies is freed and its entry is set to deleted(*its size is set to -1,for example)*. Further the records in the block before the deleted records are moved, so that free space created by the deletion gets occupied, and all free space is again between the final entry in the array and the first record.
- ✓ Pointers should not point directly to record-instead they should point to the entry for the record in the header.

ii) Fixed –Length representation

- ✓ Use one or more fixed length records

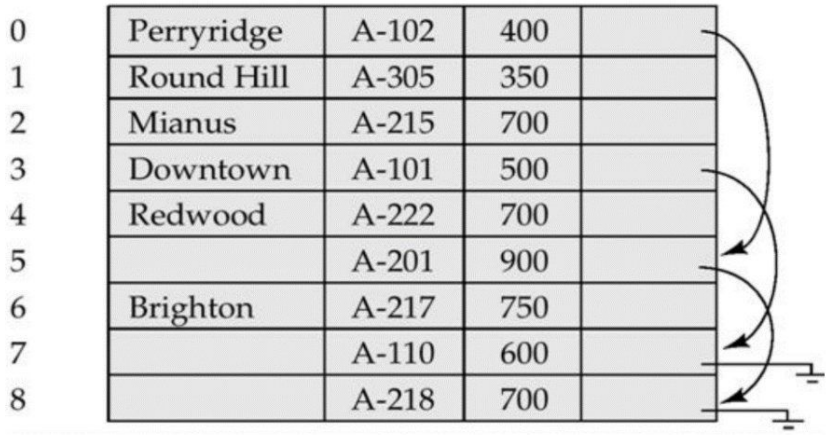
❖ **Reserved space**

- ✓ can use fixed-length records of a known maximum length
- ✓ unused space in shorter records filled with null or end of record symbol

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

❖ **List representation by pointers**

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

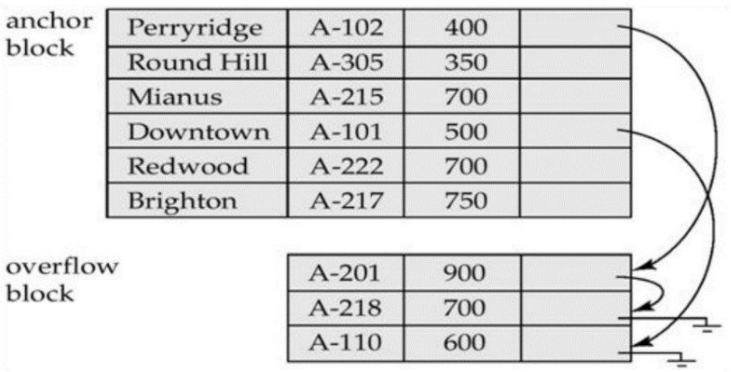


0	Perryridge	A-102	400	
1	Round Hill	A-305	350	
2	Mianus	A-215	700	
3	Downtown	A-101	500	
4	Redwood	A-222	700	
5		A-201	900	
6	Brighton	A-217	750	
7		A-110	600	
8		A-218	700	

Disadvantage to pointer structure: Space is wasted in all records except the first in chain

Solution is to allow two kinds of block in a file:

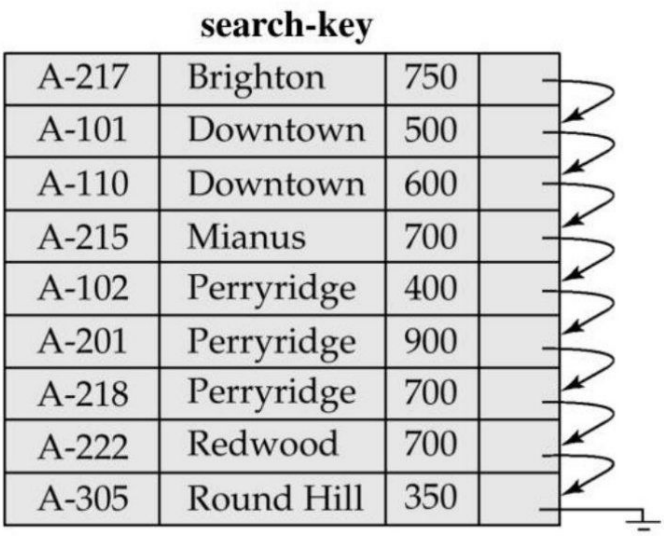
- ✓ **Anchor block-** contains the first records of chain
- ✓ **Overflow block**-contains records other than those that are the first records of chains.



Sequential file organization

- ✓ A sequential file is designed for efficient processing of records in sorted order based on some search key.
- ✓ A search key is any attribute or set of attributes; it need not be the primary key, or even a superkey.
- ✓ To permit fast retrieval of records in search-key order, we chain together records by pointers.
- ✓ The pointer in each record points to the next record in search-key order.
- ✓ Furthermore, to minimize the number of block accesses in sequential file processing, we store records physically in search-key order, or as close to search-key order as possible.

In below example, the records are stored in search-key order, using **branch name** as the search key

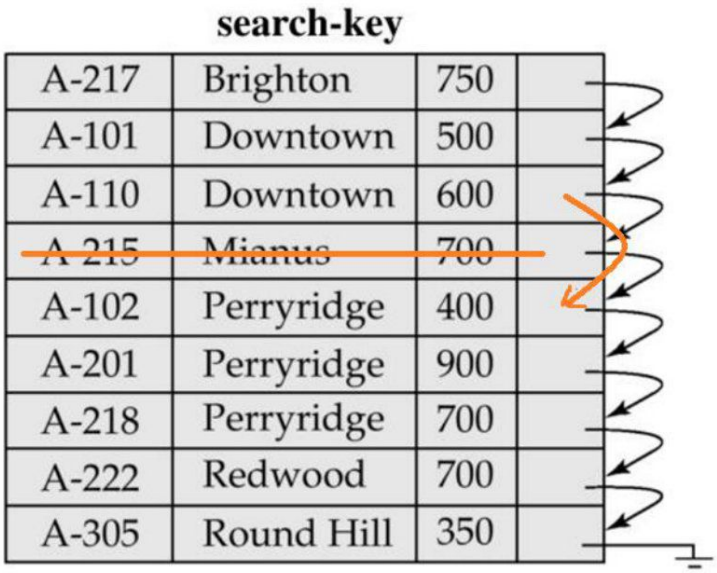


For **insertion** follow the following rule:-

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

For deletion – use pointer chains



- ✓ Need to reorganize the file from time to time to restore sequential order

Advantages of sequential file organization

- ✓ The sequential file organization is efficient and process faster for the large volume of data.
- ✓ It is simple in design. It requires no much effort to store the data
- ✓ This method can be implemented using cheaper storage devices such as magnetic tapes.
- ✓ It requires fewer efforts to store and maintain data elements.
- ✓ The sequential file organization technique is useful for report generation and statistical computation process.

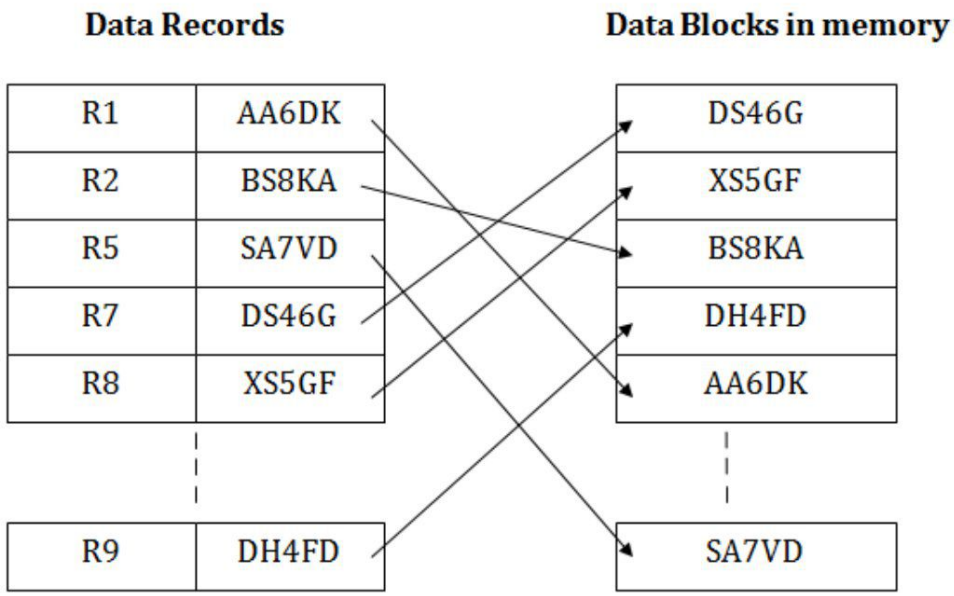
Disadvantages of sequential file organization

- ✓ It will waste time as we cannot jump on a particular record that is required but we have to move sequentially which takes our time.
- ✓ Sorted file method takes more time and space for sorting the records.

Indexed Sequential Access Method (ISAM)

ISAM (Indexed sequential access method) is an advanced sequential file organization method. In this case, records are stored in the file with the help of the primary key. For each primary key, an index value is created and mapped to the record. This index contains the address of the record in the file.

If a record has to be obtained based on its index value, the data block's address is retrieved, and the record is retrieved from memory.



Advantages of ISAM:

- ✓ In this method, each record has the address of its data block, searching a record in a huge database is quick and easy.
- ✓ This method supports range retrieval and partial retrieval of records. Since the index is based on the primary key values, we can retrieve the data for the given range of value. In the same way, the partial value can also be easily searched, i.e., the student name starting with 'JA' can be easily searched.

Disadvantages of ISAM

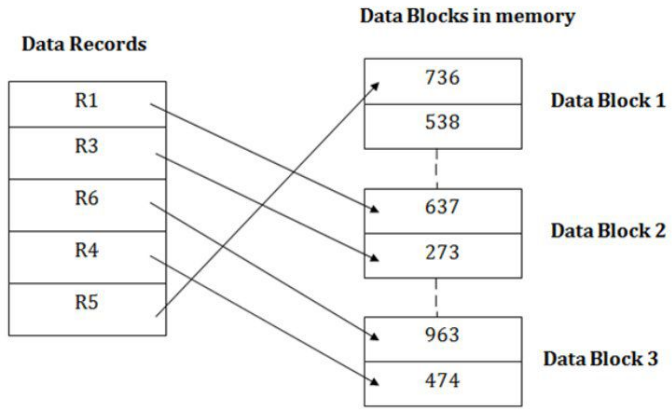
- ✓ This method requires extra space in the disk to store the index value.
- ✓ When the new records are inserted, then these files have to be reconstructed to maintain the sequence.
- ✓ When the record is deleted, then the space used by it needs to be released. Otherwise, the performance of the database will slow down.

Heap file organization

It is the simplest and most basic type of organization. It works with data blocks. In heap file organization, the records are inserted at the file's end. When the records are inserted, it doesn't require the sorting and ordering of records.

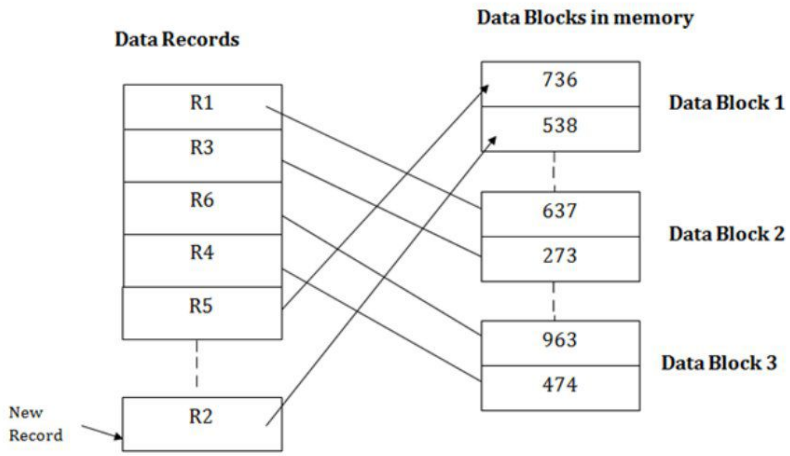
When the data block is full, the new record is stored in some other block. This new data block need not to be the very next data block, but it can select any data block in the memory to store new records. The heap file is also known as an unordered file.

In the file, every record has a unique id, and every page in a file is of the same size. It is the DBMS responsibility to store and manage the new records.



Insertion of a new record

Suppose we have five records R1, R3, R6, R4 and R5 in a heap and suppose we want to insert a new record R2 in a heap. If the data block 3 is full then it will be inserted in any of the database selected by the DBMS, let's say data block 1.



If we want to search, update or delete the data in heap file organization, then we need to traverse the data from staring of the file till we get the requested record.

If the database is very large then searching, updating or deleting of record will be time-consuming because there is no sorting or ordering of records. In the heap file organization, we need to check all the data until we get the requested record.

Advantages of Heap file organization

- ✓ It is a very good method of file organization for bulk insertion. If there is a large number of data which needs to load into the database at a time, then this method is best suited.
- ✓ In case of a small database, fetching and retrieving of records is faster than the sequential record.

Disadvantages of Heap file organization

- ✓ This method is inefficient for the large database because it takes time to search or modify the record.
- ✓ This method is inefficient for large databases.

Indexing

- ✓ Indexing mechanisms used to speed up access to desired data.(E.g., author catalog in library)
- ✓ An index takes as input a search key value and returns the address of the records hold that values.
- ✓ **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" using a "hash function"

Index evaluation metrics:

1. **Access types:** The type of access that are supported efficiently. Access types can include finding records with a specified attribute value and finding records whose attribute values fall in a specified range.
2. **Access time:** The time it takes to find a particular data item, or set of items
3. **Insertion time:** The time it takes to insert a new data item
4. **Deletion time:** The time it takes to delete a data item. This value includes the time it takes to find the item to be deleted, as well as the time it takes to update the index structure.
5. **Space overhead:** Additional space occupied by an index structure

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.

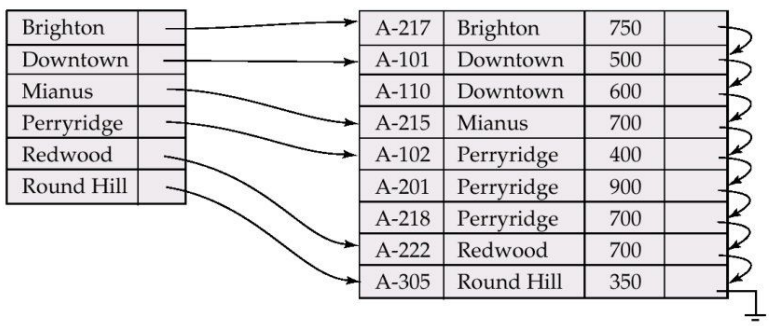
- ✓ Also called **clustering index**
- ✓ The search key of a primary index is usually but not necessarily the primary key.

Types

1. Dense index
2. Sparse index

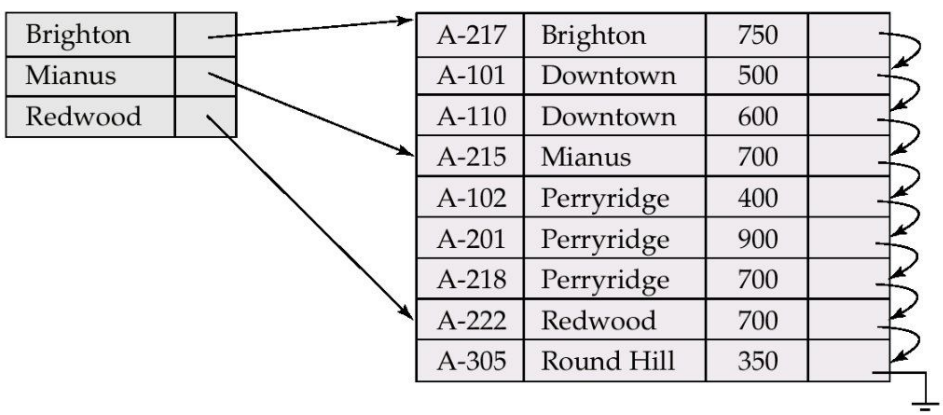
1. Dense Index

- ✓ An index record appears for every search- key value in a file.
- ✓ Index record contain the search key value and a pointer to the first data record with that search key value.
- ✓ The rest of the records with the same search-key value would be stored sequentially after the first record .
- ✓ It require more memory space for index. Records are accessed fast.



2. Sparse Index

- ✓ Contains index records for only some search-key values.
- ✓ Sparse index can be used only if the relation the relation is stored in sorted order of the search key.
- ✓ To locate a record, we find the index with the largest search key value that is less than or equal to the search key value for which we are looking.
- ✓ We start at that record pointed to by the index record, and proceed along the pointer in file (i.e. sequentially) until we find the desired record.



Comparison with dense index

- ✓ Less space and less maintenance overhead for insertions and deletions
- ✓ Generally slower than dense index for locating records.

Secondary index

- An index whose search key specifies an order different from the sequential order of the file. Also called non-clustering index**.**
- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index satisfy some condition.
  - Example 1: In the *account* database stored sequentially by account number, we may want to find all accounts in a particular branch
  - Example 2: as above, but where we want to find all accounts with a specified balance or range of balances
- We can have a secondary index with an index record for each search-key value; index record points to a bucket that contains pointers to all the actual records with that particular search-key value.

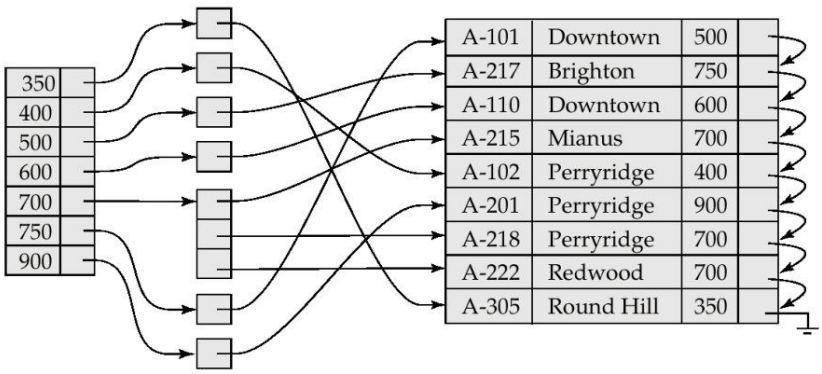


Figure: Secondary Index on balance field of account

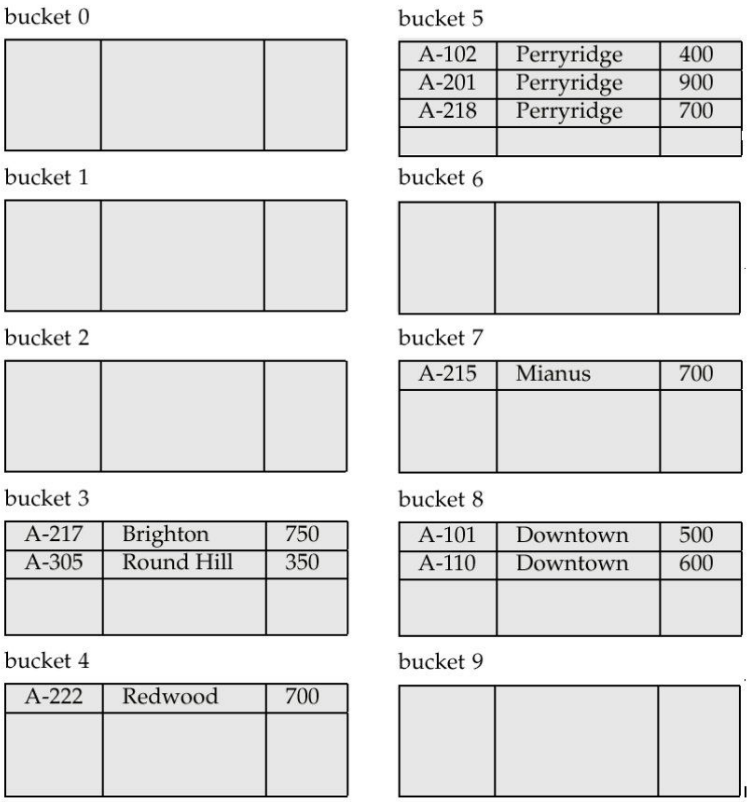
Hash file organization

- ✓ A bucket is a unit of storage containing one or more records(a bucket is typically a disk block)
- ✓ In a hash file organization we obtain the block directly from its search key value using a hash function.
- ✓ Hash function *h* is a function from the set of all search-key values *K* to the set of all bucket addresses *B.*
- ✓ Hash function is used to locate records for access, insertion as well as deletion.
- ✓ Records with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate a record

Hash file organization of *account* file, using *branch-name* as key

- ✓ There are 10 buckets,
- ✓ The binary representation of the *i*th character is assumed to be the integer *i.*
- ✓ 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



Bucket overflow

We have assumed that, when a record is inserted, the bucket to which it is mapped has space to store the record. If the bucket does not have enough space a bucket overflow is said to occur. Bucket overflow can occur for several reasons:

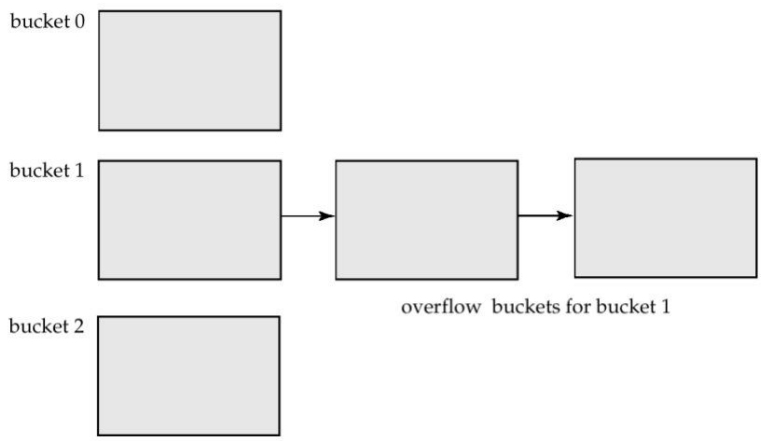
- **Insufficient buckets**
- ✓ Number of buckets (n_B) must be chosen such that $n_B > n_r/f_r$,where n_r denotes the total number of records that will be stored and f_r denotes the total number of records that will fit in a bucket.
- **Skew**
- ✓ Some buckets are assigned to more records than others, so a bucket may overflow even when other buckets still have space. This situation is called bucket skew. Skew can occur for two reasons:

1. Multiple records may have the same search key
2. The chosen hash function may result in non-uniform distribution of search keys.

The worst possible hash function maps all search key values to same bucket. Such a hash function is undesirable because all the records have to be kept in same bucket. An ideal hash function distributes the stored key uniformly across the buckets so that every bucket has the same number of records.

Handling Bucket overflow

We handle bucket overflow by using overflow buckets. If a record must be inserted in bucket b and bucket b is already full, the system provides overflow bucket for b and insert the record into the overflow bucket as shown in above figure. If the overflow bucket is also full then the system provides another overflow bucket and so on. All the overflow buckets of the given bucket are chained together in a linked list. Overflow handling using such linked list is called overflow chaining.

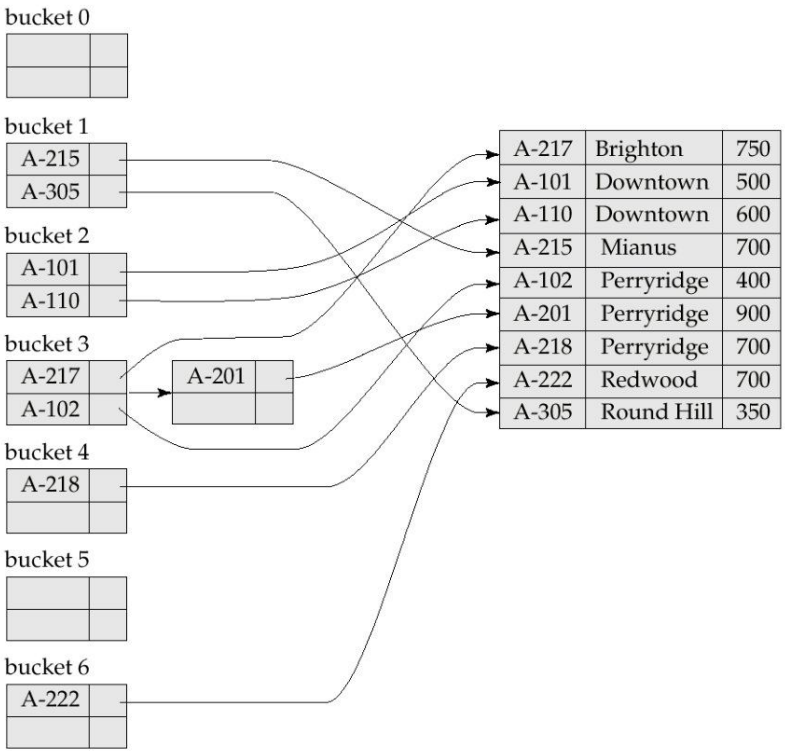


Above scheme is called closed hashing**.**

Alternative approach called open hashing. The set of buckets is fixed and there are no overflow chains. Instead, if a bucket is full, the system inserts records in some other buckets in the initial set of buckets B. One policy is to use the next bucket that has space; this policy is called linear probing.

Hash index

- ✓ A hash index organizes the search keys with their associated pointers into a hash file structure.
- ✓ We apply a hash function on a search key to identify a bucket, and store the key and its associated pointers in the bucket (or in overflow buckets).
- ✓ A hash function, h(K), is a mapping function that maps all the set of search-keys K to the bucket where keys and its associated pointers are placed.



- ✓ The hash function in the above function is H(K)= Sum of the digits of account number % 7
- ✓ The hash index has 7 buckets each size 2. One of the bucket has 3 keys mapped so it has an overflow bucket.
- ✓ Formally let K denote the set of all search keys value, B denote the set of all bucket address and h denote the hash function. To insert a record with search key ki , we compute h(Ki) which gives the address of bucket for that record. That record is then stored in that bucket.
- ✓ Deletion is also simple. If the search key value of the record to be deleted is Ki. We compute h(ki) to find the corresponding bucket for that record and delete the record from the bucket.

B+ tree Index files

A B+ Tree is simply a balanced binary search tree, in which all data is stored in the leaf nodes, while the internal nodes store just the indices.

The main disadvantage of indexed-sequential files

- ➢ Performance degrades as file grows, since many overflow blocks get created.
- ➢ Periodic reorganization of entire file is required.

B+ tree file structure maintains the efficient tree as it automatically reorganizes itself after every insertion and deletion.

Properties of B+ Trees

- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf (Internal node) has between [n/2] and n children.
- A leaf node has between [(n–1)/2] and n–1 value

Special cases:

- ✓ If the root is not a leaf, it has at least 2 children.
- ✓ If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and (n-1) values.

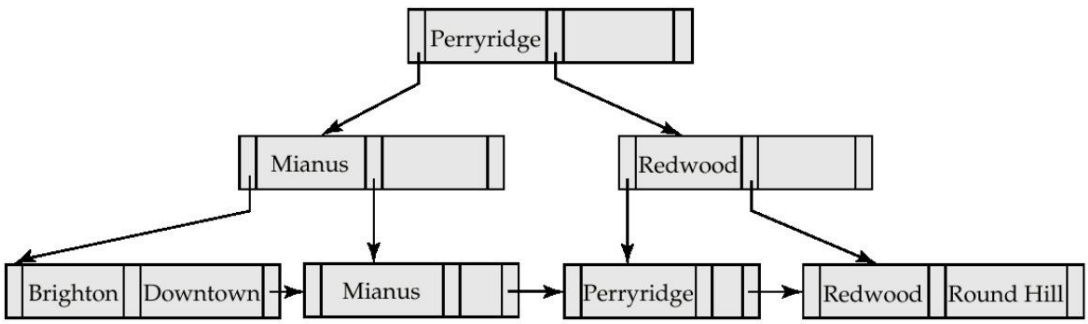
B+ tree node structure

- ➢ 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

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

Example of B+ tree of order(n=3)



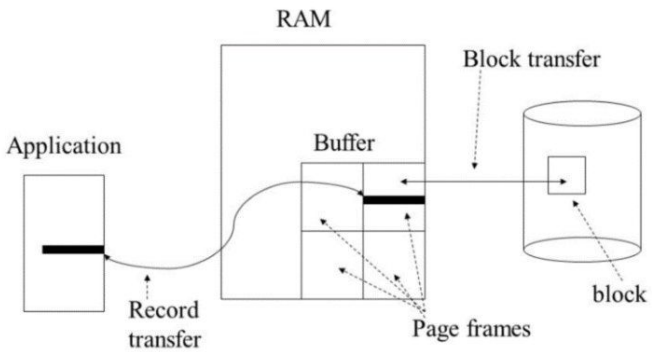
Advantages of B+ tree

- ✓ Since all records are stored only in the leaf node and are sorted sequential linked list, searching is becomes very easy.
- ✓ Using B+, we can retrieve range retrieval or partial retrieval. Traversing through the tree structure makes this easier and quicker.
- ✓ As the number of record increases/decreases, B+ tree structure grows/shrinks. There is no restriction on B+ tree size, like we have in ISAM.
- ✓ Since it is a balance tree structure, any insert/ delete/ update does not affect the performance.
- ✓ Since we have all the data stored in the leaf nodes and more branching of internal nodes makes height of the tree shorter. This reduces disk I/O. Hence it works well in secondary storage devices.

Buffer management in DBMS

Database buffer

- ✓ A database buffer is a section in the main memory that is used for the temporary storage of copies of disk block.
- ✓ Goal is to minimize the number of transfers between the disk storage and the main memory (RAM).



Buffer Manger

- ✓ A Buffer Manager is responsible for allocating space to the buffer in order to store data into the buffer.

When user request a particular block then

1. If block is already in the buffer
   - ➢ The buffer manager provides the block address in the main memory.
2. If the block is not available in the buffer
   - ➢ The buffer manager allocates the space in the buffer for a block.
   - ➢ If free space is not available, it throws out some existing blocks from the buffer to allocate the required space for the new block.
   - ➢ The blocks which are thrown are written back to the disk only if they are recently modified when writing on the disk.
   - ➢ Once the space is allocated in buffer, the buffer manager reads the requested block from the disk to the buffer and then passes the address of the requested block to the user in the main memory.

Buffer Replacement Strategy:

- ✓ If no space is left in the buffer, it is required to remove an existing block from the buffer before allocating the new one.
- ✓ The various operating system uses the LRU (least recently used) scheme.
- ✓ In LRU, the block that was least recently used is removed from the buffer and written back to the disk.
- ✓ Such type of replacement strategy is known as Buffer Replacement Strategy.
- ✓ Other buffer replacement strategy can MRU(Most Recently Used),FIFO (First In First Out) etc.

Spanned and unspanned records

- ✓ The records of a file must be allocated to disk blocks because a block is the unit of data transfer between disk and memory.
- ✓ When the block size is larger than the record size, each block will contain numerous records, although some of the files may have unusually large records that cannot fit in one block.

  Blocking factor (bfr) - the number of records that can fit into a single block

  bfr= B/R

  Where,

  - ➢ B=Block size in bytes
  - ➢ R=Record size in bytes

  Example:

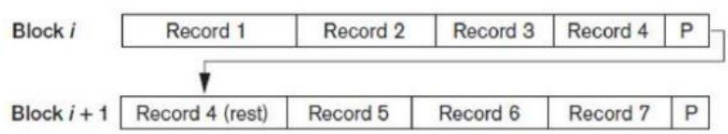
  Block size B=2000 bytes
  Record size R=100 bytes

  Now, blocking factor (bfr)= **2000/100=20**

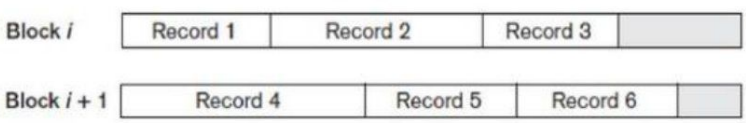
In general, R may not divide B exactly, so we have some unused space in each block equal to

B – (bfr * R) bytes.

To utilize this unused space, we can store part of a record on one block and the rest on another. A pointer at the end of the first block points to the block containing the remainder of the record. This organization is called **spanned.**



When extra space in the block is left unused the record organization is called **unspanned.**



Unit 8

Transaction and Concurrency control

Transaction Concepts

- ✓ Collections of operations that form a single logical unit of work are called transactions.
- ✓ It is a unit of program execution that accesses and possibly updates various data items
- ✓ A database system must ensure proper execution of transactions despite failures, either the entire transaction executes, or none of it does.
- ✓ During transaction execution the database may be temporarily inconsistent. When the transaction completes successfully (is committed), the database must be consistent.
- ✓ After the transaction commits the changes it has made to the database persist, even if there are system failures.
- ✓ Multiple transaction can also execute in parallel. Example: Airlines reservation system

Example: 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)
```

Two main issues to deal with:

- ✓ Failures of various kinds, such as hardware failures and system crashes
- ✓ Concurrent execution of multiple transactions

ACID Properties of Transaction

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.
- ✓ Transaction is indivisible, it completes entirely or not at all, despite failures.

Consistency

- ✓ This means that integrity constraints must be maintained so that the database is consistent before and after the transaction.
- ✓ It refers to the correctness of a database.

Isolation

- ✓ Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed.
- ✓ In simple words, if a transaction T1 is being executed and using the data item X, then data item cannot be accessed by another transaction until the transaction T1 ends.
- ✓ This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of the database state.

Durability

- ✓ The durability property of transaction indicates that the changes made to the database by a committed transaction must persist in the database.
- ✓ These changes must not be lost by any kind of failure.

Example:

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)
```

Atomicity requirement: If the transaction fails after step 3 and before step 6, after write(A) but before write(B), then the amount has been deducted from A but not added to B. In this case the system should ensure that its update are not reflected in database. Otherwise this results an inconsistent database state.

Consistency requirement: The sum of A and B is unchanged by the execution of the transaction.

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 (the sum $A + B$ will be less than it should be).

	T1	T2
1.	read(A)	
2.	$A := A - 50$	
3.	write(A)	
4.	read(B)	
5.	$B := B + 50$	read(A), read(B), print(A+B)
6.	write(B)	

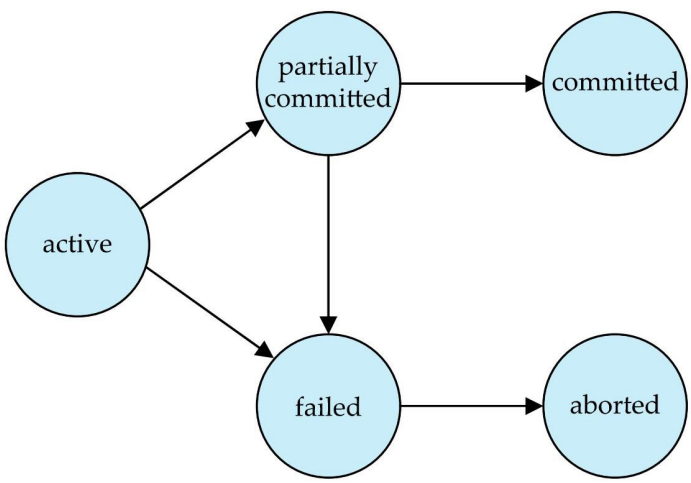
- ✓ Here, isolation can be ensured by running transactions serially, that is one after the another.
- ✓ However, executing multiple transactions concurrently has significant benefits.

Durability requirement: Once the user has been notified that the transaction has completed (i.e., the transfer of the 50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

Transaction Model and state diagram

A transaction is seen by the DBMS as a series, or list of actions. To ensure the reliability and consistency of database operations, even in the presence of system failures or errors, a transaction model is necessary.

We therefore established a simple transaction model named **transaction states.** These are the states which tell about the current state of the transaction and also tell how we will further do the processing in the transactions.



Active – This is the first state of transaction and in this state the instructions of the transaction are executing. If all the 'read and write' operations are performed without any error then it goes to the "partially committed state"; if any instruction fails, it goes to the "failed state".

Partially committed - After completion of all the read and write operation the changes are made in main memory or local buffer. If the changes are made permanent on the Database, then the state will change to "committed state" and in case of failure it will go to the "failed state".

Failed -When any instruction of the transaction fails, it goes to the "failed state" or if failure occurs in making a permanent change of data on Database.

Aborted -After having any type of failure the transaction goes from "failed state" to "aborted state" and since in previous states, the changes are only made to local buffer or main memory and hence these changes are deleted or rolled-back.

Committed – It is the state the transaction is complete and the changes are made permanent on the database

Schedules

When several transactions are executing concurrently then order of execution of various instructions is known as schedules.

Types of schedules:

1. Serial schedule
2. Non-serial schedule

1. Serial schedule

In serial schedules

- ✓ All the transactions execute serially one after the other.
- ✓ It is a type of schedule where commit or abort of one transaction initiates the execution of next transaction.
- ✓ When the first transaction completes its cycle, then the next transaction is executed and so on.
- ✓ No interleaving occurs in serial schedule

2. **Non serial schedule**

In non-serial schedules

- ✓ Multiple transactions execute concurrently.
- ✓ Operations of all the transactions are inter leaved or mixed with each other.
- ✓ It contains many possible orders in which the system can execute the individual operations of the transactions.

Schedule 1

Let T_1 transfer 50 from A to B, and T_2 transfer 10% of the balance from A to B.

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

In this schedule

- ✓ There are two transactions T1 and T2 executing serially one after the other.
- ✓ Transaction T1 executes first.
- ✓ After T1 completes its execution, transaction T2 executes.
- ✓ So, this schedule is an example of a Serial Schedule.

Schedule 2

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

In this schedule

- ✓ There are two transactions T1 and T2 executing serially one after the other.
- ✓ Transaction T2 executes first.
- ✓ After T2 completes its execution, transaction T1 executes.
- ✓ So, this schedule is also an example of a Serial Schedule.

Schedule 3

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)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

In this schedule,

- ✓ There are two transactions T1 and T2 executing concurrently.
- ✓ The operations of T1 and T2 are interleaved.
- ✓ So, this schedule is an example of a Non-Serial Schedule.

Note:

Two schedules are said to be equivalent schedules if the execution of first schedule is identical to the execution of second schedule. Here schedule 3 is equivalent to schedule 1. In schedule 1, 2 and 3, the sum A + B is preserved.

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)	
commit	
	$B := B + temp$
	write (B)
	commit

Serializability

- ✓ A schedule of S of n transaction is serializable if it is equivalent to some serial schedule of same n transactions.
- ✓ Two schedule are called result equivalent if they produce the same final state of the database.
- ✓ A schedule is **serializable** if it is equivalent to the serial schedule.

Things to be understand:

- ☞ *Serializability of schedules ensures that a non-serial schedule is equivalent to a serial schedule.*
- ☞ *It helps in maintaining the transactions to execute simultaneously without interleaving one another.*
- ☞ *In simple words, serializability is a way to check if the execution of two or more transactions are maintaining the database consistency or not.*
- ☞ *Basic assumption in each transaction preserve data consistency. Thus serial execution of a set of transaction preserves data consistency.*
- ☞ *Main objective of serializability is to find a non-serial schedule that allow transactions to execute concurrently without interference and produce a same effect on the database state that could be produced by a serial execution.*

Different forms of schedule equivalence give rise to notations of:

- ➢ Conflict serializability
- ➢ View serializability

Conflict serializability

- ✓ A non-serial schedule is a conflict serializable if, after performing some swapping on the non-conflicting operations, it can be transforms into a serial schedule.
- ✓ Actions I_i and I_j of transactions T_i and T_j respectively, conflict if and only if there exists some data item Q accessed by both I_i and I_j, and at least one of these instructions wrote Q.
- ❖ I_i=read(Q) ,I_j=read(Q) I_i and I_j don't conflict
- ❖ I_i=read(Q) ,I_j=write(Q) They conflict
- ❖ I_i=write(Q) ,I_j=read(Q) They conflict
- ❖ I_i=write(Q) ,I_j=write(Q) They conflict

- ✓ If I_i and I_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Things to be understand

Two operations inside a schedule are called conflicting if they meet these three conditions:

- ✓ They belong to two different transactions.
- ✓ They are working on the same data item.
- ✓ One of them is performing the WRITE operation.

Example:

Schedule 1:

T1	T2
Read(A)	
	Read(A)

Here, in **Schedule 1** two operations are non -conflicting so they can be swapped.

After swapping schedule becomes

T1	T2
	Read(A)
Read(A)	

Schedule 2:

T1	T2
Read(A)	
	Write(A)

Here in schedule 2 two operations are conflicting, so they cannot be swapped.

Schedule 3:

T1	T2
Read(A)	
	Write(B)

Here in schedule 3 two operations are non-conflicting, because in spite of having one write operation, but operation is performing in different data item.
So they can be swapped after swapping schedule becomes

T1	T2
	Write(B)
Read(A)	

Conflict Equivalent and Conflict serializable

- ✓ Two schedules are **conflict equivalent** if they can be turned one into another by a sequence of non-conflicting swaps of adjacent actions.
- ✓ We say that a schedule S is **conflict serializable if** it is conflict equivalent to a serial schedule.

Here, **Schedule 1** can be transformed into **Schedule 2,** a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore Schedule 1 is conflict serializable.

Schedule1

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

Schedule 2

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

Example of a schedule that is not conflict serializable:

T_3	T_4
read (Q)	
	write (Q)
write (Q)	

We are unable to swap instructions in the above schedule to obtain either the serial schedule $< T_3, T_4 >$, or the serial schedule $< T_4, T_3 >$.

Testing for conflict serializability

- ✓ Precedence graph or serialization graph is commonly used to test conflict serializability of a schedule.
- ✓ It is set of directed graph (V,E) consisting of set of nodes V={T1,T2,T3,T4,..Tn} and a set of directed edges E={e1,e2,e3….en}
- ✓ The graph contains one node for each transaction Ti.
- ✓ An edge ei is of the form Ti→Tj where Ti is starting node of ei and Tj is ending node of ei.
- ✓ For each transaction Ti participating in schedule S ,create node labeled Ti in the precedence graph.

For each case in **schedule S**

- ➢ Where Tj executes a read_item(X) after Ti executes a write_item(X) ,create an edge (Ti→Tj)
- ➢ Where Tj executes a write_item(X) after Ti executes a read_item(X) ,create an edge (Ti→Tj)
- ➢ Where Tj executes a write_item(X) after Ti executes a write_item(X) ,create an edge (Ti→Tj)

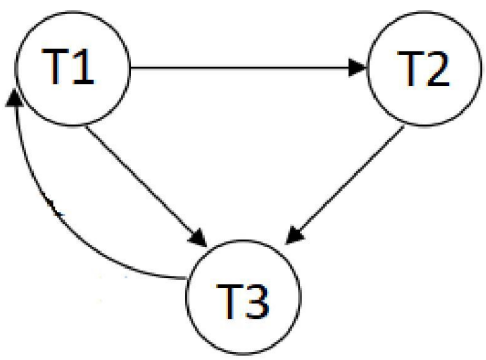
The schedule S is serializable if and only if the precedence graph has no cycles.

Example 1

Test serializability of a given schedule

T1	T2	T3
Read(X)		
		Read(X)
Write(X)		
	Read(X)	
		Write(X)

The precedence graph is as below



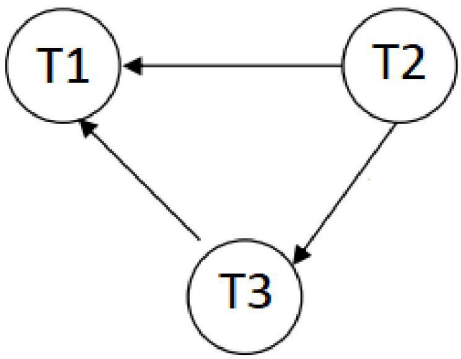
Since the graph contains a cycle, hence it is not conflict serializable.

Example 2

Test serializability of a given schedule

T1	T2	T3
		Read(X)
	Read(X)	
		Write(X)
Read(X)		
Write(X)		

The precedence graph is as below.



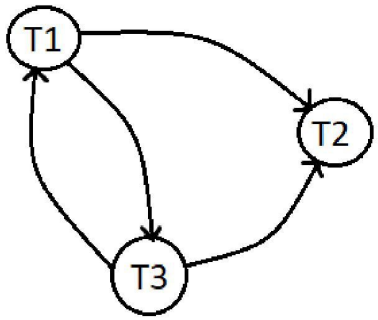
Here graph does not contain a cycle, hence it is conflict serializable.

Example 3:

Test serializability of a given schedule

T1	T2	T3
Write(A)		
	Read(A)	
		Write(B)
Write(B)		
		Write(B)
	Write(A)	
		Read(B)
	Read(B)	

The precedence graph is as below.



Since the graph contains a cycle, hence it is not conflict serializable.

Example 4:

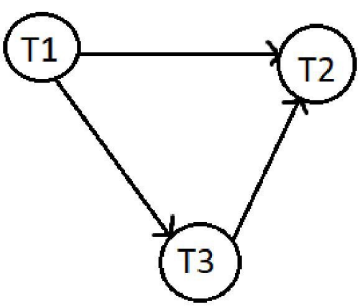
Consider the schedule S1.

S1: r1(x),r3(y) w1(x),w2(y),r3(x),w2(x)

Creating tabular representation of above schedule

T1	T2	T3
r(x)		
		r(y)
w(x)		
	w(y)	
		r(x)
	w(x)	

The precedence graph is as below.



Since the graph is acyclic ,the schedule is conflict serializable.

Performing Topological sort on this graph would give a possible serial schedule that is conflict equivalent to schedule S1.

In topological Sort, we first select the node with in-degree 0,which is T1. This would be followed by T3 and T2 so,S1 is conflict serializable since it is conflict equivalent to serial schedule T1→T3→T2.

View serializability

- ✓ A schedule is view serializable if it is view equivalent to serial schedule.
- ✓ Every conflict serializable schedule is also view serializable but reverse may not always true.
- ✓ Every view serializable schedule that is not conflict serializable has blind writes.

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.

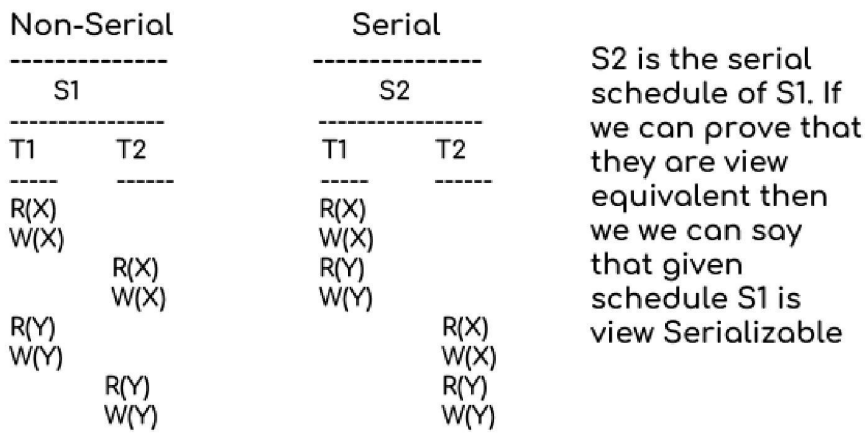
1. If in schedule S, transaction Ti reads the initial value of Q, then in schedule S' also transaction Ti must read the initial value of Q.

2. If in schedule S transaction Ti executes read(Q), and that value was produced by transaction Tj (if any), then in schedule S' also transaction Ti must read the value of Q that was produced by the same write(Q) operation of transaction Tj .

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'.

Let us discuss above conditions with examples.

Example 1:



Initial read

- ✓ In schedule S1, transaction T1 first reads the initial value of X and in also schedule S2 transaction T1 reads the initial value of X. This conditions also satisfies for Y also.
- ✓ We checked for both data items X & Y and the initial read condition is satisfied in S1 & S2.

Updated read

- ✓ In S1, transaction T2 reads the value of X, written by T1. In S2, the same transaction T2 reads the X after it is written by T1.
- ✓ In S1, transaction T2 reads the value of Y, written by T1. In S2, the same transaction T2 reads the value of Y after it is updated by T1.
- ✓ The update read condition is also satisfied for both the schedules.

Final write

- ✓ In schedule S1, the final write operation on X is done by transaction T2. In S2 also transaction T2 performs the final write on X.
- ✓ Lets check for Y. In schedule S1, the final write operation on Y is done by transaction T2. In schedule S2, final write on Y is done by T2.
- ✓ We checked for both data items X & Y and the final write condition is satisfied in S1 & S2.

Since all the three conditions that checks whether the two schedules are view equivalent are satisfied in this example, which means S1 and S2 are view equivalent. Also, as we know that the schedule S2 is the serial schedule of S1, thus we can say that the schedule S1 is view serializable schedule.

Example 2:

Schedule 1

T1	T2	T3
Read(A) Write(A)	Write(A)	 Write(A)

Schedule 2

T1	T2	T3
Read(A) Write(A)	 Write(A)	 Write(A)

Initial Read

In schedule 1, transaction T1 first reads the initial value of A and in also schedule 2 transaction T1 reads the initial value of A.

Updated read

In both schedule 1 and Schedule 2, there is no read except the initial read that's why we don't need to check that condition.

Final write

In schedule 1, the final write operation on A is done by transaction T3. In Schedule 2 also transaction T3 performs the final write on A.

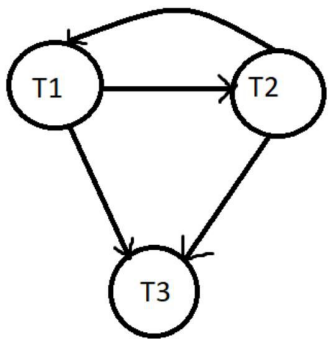
Here all the conditions are satisfied. So, schedule 1 is view equivalent to the schedule 2 which is serial schedule. As we know if any schedule is view equivalent to the serial schedule we can say that schedule is view serializable. So schedule 1 is view serializable.

Note:

Schedule that is view serializable may not be conflict serializable. In above Example 2

Schedule 1 is view serializable but not conflict serializable and also it must be noted that Every view serializable schedule that is not conflict serializable has blind writes(write without reading)

To test conflict serializability of schedule 1 in above Example 2 lets draw precedence graph



Here, precedence graph contains cycle so schedule is not conflict serializable but view serializable.

Concurrent Executions

A multiple transactions are allowed to run concurrently in the system.

Advantages of **concurrent executions** are:

- ✓ Increased processor and disk utilization: leading to better transaction throughput
  E.g. one transaction can be using the CPU while another is reading from or writing to the disk
- ✓ Reduced average response time for transactions: short transactions need not wait behind long ones.

Problems with concurrency

When multiple transactions execute concurrently in an uncontrolled or unrestricted manner then it might lead to several problems. Such problems are called concurrency problems.

Concurrency problems in transactions:

1) Dirty read problem (Temporary update problem)
2) Unrepeatable read problem
3) Lost update problem
4) Phantom read problem

1) Dirty read problem

Reading the data written by an uncommitted transaction is called as dirty read.

The dirty read problem occurs when one transaction update an data item of database, and somehow the transaction fails. But before the item gets rollback, the updated database item is accessed by another transaction.

Example:

T1	T2
Read(A) A=A+100 Write(A) . . . . . . Failure	 Read(A) A=A-200 Write(A) commit

Here, in above transaction

- ✓ T2 reads the value of A written by the uncommitted transaction T1.
- ✓ T2 writes the updated value of A.
- ✓ T2 commit
- ✓ T1 fails in later stages and rollbacks.
- ✓ Thus the value of T2 read now stands to be incorrect.
- ✓ Therefore database becomes inconsistent.

2) Unrepeatable read problem

This problem occurs when two or more read operation of the same transaction read different values of the same database item.

T1	T2
Read(A) Read(A)	 Read(A) A=A+100 Write(A)

Here,

- ☞ T1 reads the value of A(=500 say)
- ☞ T2 reads the value of A(=500)
- ☞ T2 updates the value of A from 500 to 600
- ☞ T1 again reads value of A (but=600)

Here, T1 gets reads different value of A in second reading, that was updated by T2.

3) Lost update problem

In the Lost update problem, the update done to a data item by a transaction is lost as it is overwritten by the update done by another transaction.

T1	T2
Read(A) A=A-50 Write(A) Commit	 Write(A) commit

Here,

- T1 reads the value of A(=500 say)
- T1 updates the value of A(=450)
- T2 does blind write , say A=25(write without read)
- T2 commit
- When T1 commits, it writes A=25 in the database

4) Phantom read problem

T1	T2
Read(A) delete(A)	 Read(A) Read(A)

The phantom read problem occurs when a transaction reads a data item once but when it tries to read the same data item again, an error occurs saying that the items does not exist.

Here,

- T1 reads A
- T2 reads A
- T1 deletes A
- T2 tries to reading A but does not find it

Concurrency control

- ✓ Concurrency control protocols are the set of rules which are maintained in order to solve the concurrency control problems in the database.
- ✓ It ensures that the concurrent transactions can execute properly while maintaining the database consistency.

Some of the currency control protocols are as follows:

1. **Lock- based protocol**
2. **Graph based protocol**

1. Lock –based protocol

a) Simple locking protocol

In this protocol each transaction cannot read or write the data until it acquires an appropriate lock on it.

Data items can be locked in two modes

1. **Shared (S) mode**

If a transaction T_i has a shared mode lock on data item Q then

- ✓ T_i can be read them but not update Q
- ✓ Any other transactions can also obtain shared lock on the same data item Q but no exclusive lock

S-lock is requested using **lock-S** instruction.

2. Exclusive(X) mode

If a transaction T_i has exclusive mode lock on data time Q then

- ✓ T_i can both read and update Q
- ✓ No other transaction can obtain either of shared lock or Exclusive lock on the same data item Q

X-lock is requested using **lock-X** instruction

Based on these locking modes we can define Lock-compatibility matrix

	S	X
S	True	False
X	False	False

- ✓ A transaction may granted a lock on data item if the requested lock is compatible with locks already held on data items by other transactions.
- ✓ If a lock cannot granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.
- ➢ In above compatibility matrix, shared mode is compatible with only shared mode.
- ➢ If any transactions holds an exclusive lock on item ,no other transaction may hold any lock on the item.
- ➢ A transaction can unlock a data item Q by instruction unlock(Q).
- ➢ Transaction must hold a lock on data item as long as it access item.

Problems associated with simple locking protocol

1) The serializability cannot be always assured

Let us consider a banking example. Let A and B be the two bank accounts that are accessed by transaction T1 and T2. Transaction T1 transfer transfers 50 from account B to account A. Transaction T2 displays the total amount of money in accounts A and B – that is , the sum (A+B)

```
T1: lock-X(B);
    read(B);
    B:=B-50;
    write(B);
    unlock(B)
    lock-X(A);
    read(A);
    A:=A+50;
    write (A);
    unlock(A);
```

Transaction T1

```
T2: lock-S(A)
read(A)
unlock(A)
lock-S(B);
read(B);
unlock(B);
display(A+B);
```

Transaction T2

Suppose that values of account A and B are 100 and 200 respectively. If these two transactions are executed serially, either in the order T1 ,T2 or the order T2,T1 then transaction T2 will be display the value 300.If these transaction executed concurrently, then schedule 1 as shown in figure below is possible. In this case transaction T2 displays 250 which is incorrect. **The reason for this mistake is that the transaction T1 unlocked data item B too early, as a result of which T2 saw an inconsistent state.**

Schedule 1:

T_1	T_2	concurrency-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A, T_1)
read(A)		
$A := A + 50$		
write(A)		
unlock(A)		

2) Deadlock

Consider the partial schedule

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

- ✓ Neither T3 nor T4 can make progress – executing lock-S(B) causes T4 to wait for T3 to release its lock on B, while executing lock-X(A) causes T3 to wait for T4 to release its lock on A.
- ✓ Such a situation is called a deadlock.
- ✓ To handle a deadlock one of T3 or T4 must be rolled back and its locks released.

3) Starvation

Starvation is also possible if concurrency control manager is badly designed.

For example

- ✓ Suppose a transaction T2 has a shared-mode lock on a data item, and another transaction T1 requests an exclusive-mode lock on the data item. Clearly, T1 has to wait for T2 to release the shared-mode lock.
- ✓ Meanwhile, a transaction T3 may request a shared-mode lock on the same data item. The lock request is compatible with the lock granted to T2, so T3 may be granted the shared-mode lock. At this point T2 may release the lock, but still T1 has to wait for T3 to finish.
- ✓ But again, there may be a new transaction T4 that requests a shared-mode lock on the same data item, and is granted the lock before T3 releases it.

In fact, it is possible that there is a sequence of transactions that each requests a shared-mode lock on the data item, and each transaction releases the lock a short while after it is granted, but T1 never gets the exclusive-mode lock on the data item. The transaction T1 may never make progress, and is said to be starved.

We can avoid starvation of transactions by granting locks in the following manner:

When a transaction Ti requests a lock on a data item Q in a particular mode M, the concurrency-control manager grants the lock provided that:

1. There is no other transaction holding a lock on Q in a mode that conflicts with M.

2. There is no other transaction that is waiting for a lock on Q and that made its lock request before Ti .Thus, a lock request will never get blocked by a lock request that is made later.

b) Two phase locking protocol

- ✓ Simple Lock-based protocol does not guarantee Serializability. Schedules may follow the preceding rules but a non-serializable schedule may result.
- ✓ This is where the concept of Two-Phase Locking(2-PL) comes into the picture, 2-PL **ensures serializability**.

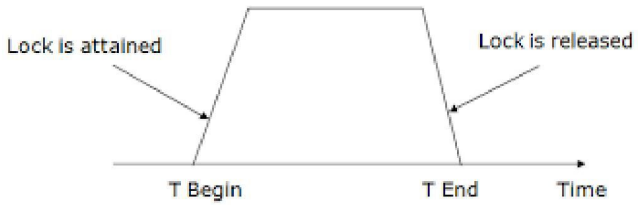
Here transaction issues lock and unlock request in two phases.

Phase 1: Growing phase

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

Phase 2: Shrinking phase

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



Note : If lock conversion is allowed, then upgrading of lock(from S(a) to X(a)) is allowed in the Growing Phase, and downgrading of lock (from X(a) to S(a)) must be done in shrinking phase.

The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e., the point where a transaction acquired its final lock).

Here, growing and shrinking phases of 2PL can be illustrated by using the following diagram

Time	T1	T2
1	Lock-S(A)	
2		Lock-S(A)
3	Lock-X(B)	
4		
5	Unlock(A)	
6		Lock-X(C)
7	Unlock(B)	
8		Unlock(A)
9		Unlock(C)

Transaction T1

Growing Phase: From Time 1 to 3.
Shrinking Phase: From Time 5 to 7
Lock Point: At time 3

Transaction T2

Growing Phase: From Time 2 to 6
Shrinking Phase: From Time 8 to 9
Lock Point: At Time 6

As we say, 2PL will generate the serial schedule then what will be the sequence of transaction execution?

Answer: If locking point of Transaction T1 comes earlier than transaction T2 then the T1 will execute first than T2.

***Note**: Locking point means the point where a transaction acquired its final lock*

Although,2-Phase Locking ensures serializability, but there are still some drawbacks of 2-PL.They are as follows:

1) **The schedule that is produced through 2PL may be irrecoverable**

   Example:

T1	T2
Lock-X(A)	
R(A)	
Write(A)	
Unlock(A)	
.	Lock-S(A)
.	Read(A)
.	Unlock(A)
.	Commit
Failure	

If a transaction does a dirty read operation from an uncommitted transaction and commits before the transaction from where it has read the value, then such a schedule is called an irrecoverable schedule.

The above schedule is a irrecoverable because of the reasons mentioned below

- ✓ The transaction T2 is also committed before the completion of transaction T1.
- ✓ The transaction T1 fails later and there are rollbacks.
- ✓ The transaction T2 reads an incorrect value.
- ✓ The transaction T2 which is performing a dirty read operation on A.

Finally, the transaction T2 cannot recover because it is already committed.

2) **The schedule that is produced through 2PL locking may still contains a deadlock problem**

T1	T2
Lock-X(A)	
	Lock-X(B)
Lock-X(B)	
	Lock-X(A)

Here,

- ✓ In T1 Exclusive lock on data A is granted.
- ✓ In T2 Exclusive lock on data B is granted
- ✓ In T1 Exclusive Lock on data B waits until T2 unlock Exclusive Lock on B
- ✓ Exclusive Lock in Data A is wait until T1 unlock the Exclusive lock on A

3) Cascading rollback problem

Time	T1	T2	T3
1	Lock-X(A)		
2	Read(A)		
3	Write(A)		
4	Unlock(A)		
5		Lock-S(A)	
6		Read(A)	
7			Lock-S(A)
8			Read(A)

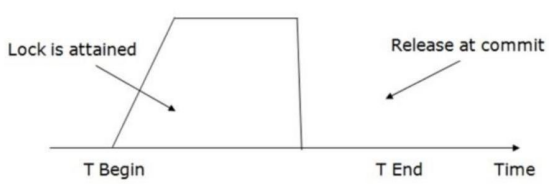
If the rollback of one transaction cause the rollback of other dependent transactions called cascading rollback.

- ✓ Here, T2 is dependent on T1,and T3 is dependent on T2
- ✓ If T1 failed,T1 must be rolled back ,similarly as T2 is dependent on T1,T2 also must be rolled back.
- ✓ Again as T3 is dependent on T2,T3 must also be rolled back.

Extensions to basic two-phase locking needed to ensure recoverability of freedom from cascading roll-back.

Strict two-phase locking

- ✓ A transaction must **hold all its exclusive locks** till it commits/aborts.
- ✓ Ensures recoverability and avoids cascading rollbacks.
- ✓ The only difference between 2PL and strict 2PL is that Strict-2PL does not release a lock after using it.
- ✓ Strict-2PL protocol does not have shrinking phase of lock release.
- ✓ Strict-2PL waits until the whole transaction to commit, and then it releases all the locks at a time.



Rigorous two-phase locking:

- ✓ A transaction must **hold all locks** till commit/abort.
- ✓ Transactions can be serialized in the order in which they commit.

Note: *The difference between Strict 2-PL and Rigorous 2-PL is that Rigorous is more restrictive, it requires both Exclusive and Shared locks to be held until after the Transaction commits and this is what makes the implementation of Rigorous 2-PL easier.*

2. Graph based protocols

Graph Based Protocols are yet another way of implementing Lock Based Protocols.

As we know the prime problems with Lock Based Protocol have been avoiding Deadlocks and ensuring a Strict Schedule. We've seen that Strict Schedules are possible with following Strict or Rigorous 2-PL. We've even seen that Deadlocks can be avoided if we follow Conservative 2-PL but the problem with this protocol is it cannot be used practically. Graph Based Protocols are used as an alternative to 2-PL.

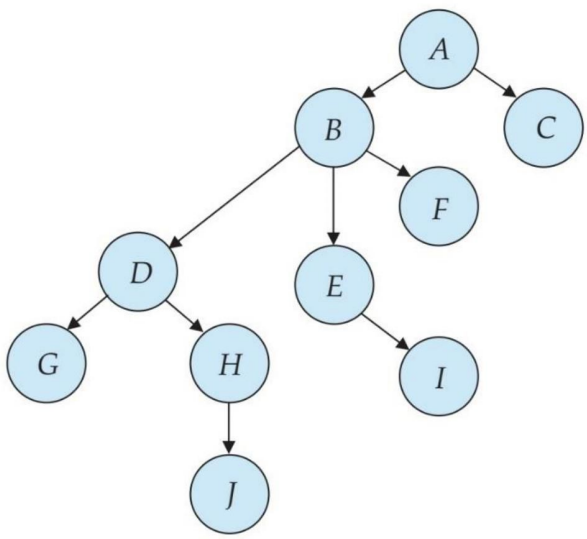
Implementation

Tree Based Protocols is a simple implementation of Graph Based Protocol. A prerequisite of this protocol is that we know the order to access a Database Item. For this we implement a Partial Ordering on a set of the Database Items (D) {d1, d2, d3, ….., dn} . The protocol following the implementation of Partial Ordering is stated as-

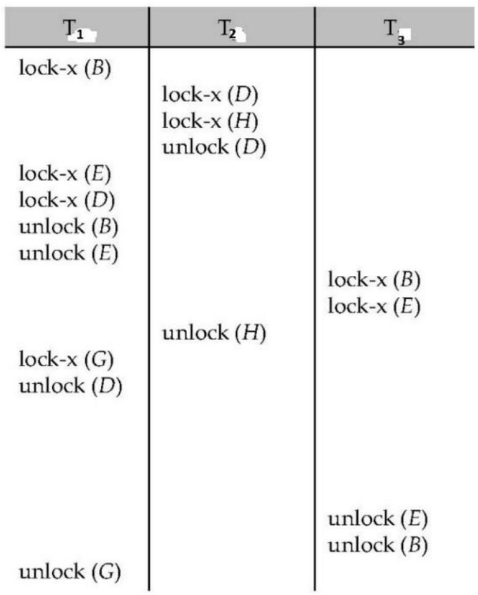
- ✓ If di –> dj then any transaction accessing both di and dj must access di before accessing dj.
- ✓ Implies that the set D may now be viewed as a directed acyclic graph (DAG), called a database graph.
- ✓ Only Exclusive Locks are allowed.
- ✓ The first lock by Ti may be on any data item. Subsequently, a data Q can be locked by Ti only if the parent of Q is currently locked by Ti.
- ✓ Data items can be unlocked at any time.
- ✓ A data item that has been locked and unlocked by T_i cannot subsequently be relocked by T_i.

Following the Tree based Protocol ensures Conflict Serializability and Deadlock Free schedule. We need not wait for unlocking a data item as we did in 2-PL protocol, thus increasing the concurrency.

Now, let us take an example, following is a Database Graph which will be used as a reference for locking the items subsequently.



Taking an example based on the above Database Graph, we have three Transactions in this schedule and we will only see how Locking and Unlocking works.



T_1	T_2	T_3
lock-x (*B*)		
	lock-x (*D*)	
	lock-x (*H*)	
	unlock (*D*)	
lock-x (*E*)		
lock-x (*D*)		
unlock (*B*)		
unlock (*E*)		
		lock-x (*B*)
		lock-x (*E*)
	unlock (*H*)	
lock-x (*G*)		
unlock (*D*)		
		unlock (*E*)
		unlock (*B*)
unlock (*G*)		

Data items Locked and Unlocked are following the same rule as given above and follow the Database Graph.

Advantages

- ✓ Ensures Conflict Serializable Schedule.
- ✓ Ensures Deadlock Free Schedule
- ✓ Unlocking can be done anytime.

Disadvantages

- ✓ Unnecessary locking overheads may happen sometimes, like if we want both D and E, then at least we have to lock B to follow the protocol.
- ✓ Cascading Rollbacks is still a problem.
- ✓ We don't follow a rule of when Unlock operation may occur so this problem persists for this protocol.

SQL Standard Isolation Levels

Isolation determines how transaction integrity is visible to other users and systems. It means that a transaction should take place in a system in such a way that it is the only transaction that is accessing the resources in a database system.

Isolation levels define the degree to which a transaction must be isolated from the data modifications made by any other transaction in the database system.

A **transaction isolation level** is defined by the following phenomena:

- ✓ Dirty Read
- ✓ Non-Repeatable read
- ✓ Phantom Read

Based on these phenomena, **The SQL standard defines four isolation levels:**

Read Uncommitted: Read Uncommitted is the lowest isolation level. In this level, one transaction may read not yet committed changes made by other transactions, thereby allowing dirty reads. At this level, transactions are not isolated from each other.

Read Committed: This isolation level guarantees that any data read is committed at the moment it is read. Thus it does not allow dirty read. The transaction holds a read or write lock on the current row, and thus prevents other transactions from reading, updating, or deleting it.

Repeatable Read: This is the most restrictive isolation level. The transaction holds read locks on all rows it references and writes locks on referenced rows for update and delete actions. Since other transactions cannot read, update or delete these rows, consequently it avoids non-repeatable read.

Serializable: This is the highest isolation level. A serializable execution is guaranteed to be serializable. Serializable execution is defined to be an execution of operations in which concurrently executing transactions appears to be serially executing.

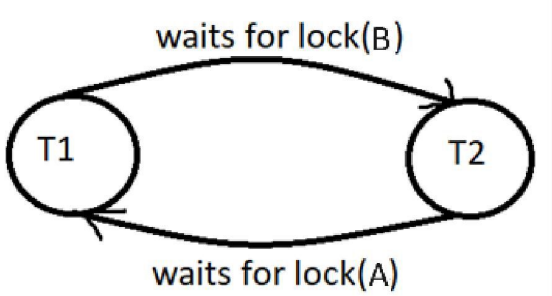
Deadlock

- ✓ A system is in a deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set.
- ✓ None of the transactions can make progress in such a situation.
  Simple Example of deadlock is given below
  - ✓ Transaction T1 request resource A and received it.
  - ✓ Transaction T2 request resource B and received it.
  - ✓ Transaction T2 request resource A and is queued up waiting the release of A by transaction T1.
  - ✓ Transaction T1 request resource B and is queued up waiting the release of B by transaction T2.

Consider a schedule

T1	**T2**
Lock-X(A)	
Read(A)	
A:=A-50	
Write(A)	
	Lock-S(B)
	Read(B)
	Lock-S(A)
Lock-X(B)	

- ✓ Neither T_1 nor T_2 can make progress
- ✓ Executing **lock-S***(A)* causes T_2 to wait for T_1 to release its lock on *A*
- ✓ while executing lock-**X***(B)* causes T_1 to wait for T_2 to release its lock on *B*.
  Such a situation is called a **deadlock**.



Conditions for deadlock

We can say a deadlock is occurred if these four conditions occur simultaneously.

1. Mutual Exclusion: Only one transaction at a time can use a resource.
2. Hold and wait: A transaction holding at least one resource is waiting to acquire additional resources held by other transactions.
3. **No preemption**: A resource can be released only voluntarily by the transaction holding it after the transaction completed its task.
4. **Circular wait:** There exist a set {T1, T2,…….TN} of waiting process such that T1 is waiting for a resource that is held by T2, T2 is waiting for a resource held by T3 and finally TN is waiting for resource held by T1.

- ✓ There are two principal methods for dealing with the deadlock problem. We can use a **deadlock prevention protocol** to ensure that the system will never enter a deadlock state.
- ✓ Alternatively, we can allow the system to enter a deadlock state, and then try to recover by using a **deadlock detection and deadlock recovery** scheme. As we shall see, both methods may result in transaction rollback.

Deadlock Prevention

Deadlock Prevention Protocols ensure that the system will never enter into a deadlock state.

Two approach to deadlock prevention

- ✓ First approach ensures that no cyclic waits can occur by ordering the requests for locks, requiring all locks to be acquired together.
- ✓ Second approach performs transaction rolled back instead of waiting for a lock, whenever the wait could be potentially result in a deadlock.

The first approach requires that each transaction locks all its data items before execution. Moreover either all are locked in one step or none are locked. There are two main disadvantages.

- ✓ It is often hard to predict, before the transaction begins, what data items need to be locked.
- ✓ Data items utilization may be very low, since many of the data items may be locked but unused for it a long time.

The second approach for preventing deadlock is use preemption and transaction rollback. Two different deadlock prevention schemes using timestamps have been proposed.

a) Wait –die scheme (non-preemptive)

- ✓ When transaction T_i requests a data item currently head by T_j . T_i is allowed to wait only if it has a timestamp smaller than that of T_j (i.e T_i older than T_j) .Otherwise T_i rolled back(dies).
- ✓ For example, suppose that transactions T1,T2 and T3 have timestamp 20,30 and 40 respectively. If T_1 request data item held by T_2, then T_1 will be wait. If T3 requests a data item held by T2, then T3 will be rolled back.

b) Wound –wait scheme(preemptive)

- ✓ 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(i.e T_i is younger than T_j) . Otherwise T_j rolled back.(T_j is wounded by T_i)
- ✓ For example, suppose that transactions T1 ,T2 and T3 have timestamp 20,30 and 40 respectively. If T1 request a data item held by T2, then data item will be preempted from T2 and T2 will be rolled back. If T3 requests a data item held by T2,then T3 will wait.

Both in wait-die and in wound-wait schemes, a rolled back transaction is restarted with its original timestamp. Older transaction thus have precedence over new ones, and starvation is hence avoided.

c) Timeout-Based schemes

- ✓ A transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back. Thus deadlock is not possible.
- ✓ Simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.

Deadlock Detection

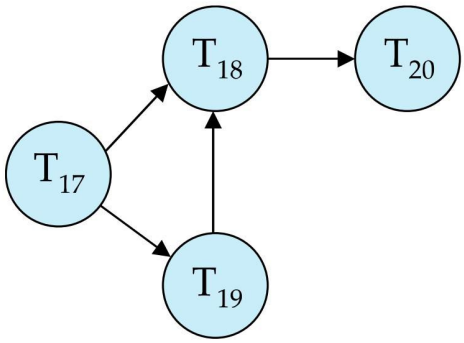
Deadlock can be described as a wait for graph, which consist of a pair G=(V,E)

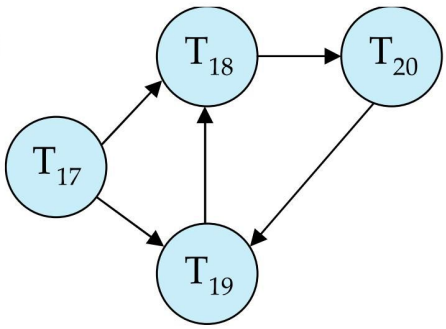
- V is a set of vertices (all the transactions in the system)
- E is a set of edges;each element is ordered pair $T_i \rightarrow T_j$.

If $T_i \rightarrow T_j$ is in E, then there is directed edge from T_i to T_j ,implying that T_i is waiting for T_j to release a data item.

When a T_i requests a data item concurrently being held by T_j , then edge $T_i \rightarrow T_j$ is inserted in the wait for graph. This edge is removed only when T_j is no longer holding a data item needed by T_i .

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**

Deadlock Recovery

When a detection algorithm determines that a deadlock exists, the system must recover from deadlock. The most common solution is to rollback one or more transactions to break the deadlock. Three actions need to be taken:

1. **Selection of victim:** Given a set of deadlocked transactions ,we must determine which transaction to rollback to break the deadlock. We should roll back the transactions that will incur minimum cost. Many factors may determine the cost of rollback, including:
   a) How long the transaction has computed, and how much longer the transaction will compute before it completes its designated task.
   b) How many data items the transaction has used.
   c) How many more data items the transactions needs for it to complete.
   d) How many transactions will involved in the rollback.

2. **Rollback:** Once we have decided that a particular solution transaction must be rolled back, we must determine how far this transaction should be rolled back.
   - ✓ Simplest solution is a total rollback. Abort the transaction and then restart it.
   - ✓ It is more effective solution is partial rollback. Rollback the transaction only as far as necessary to break the deadlock.
3. **Starvation:** In a system where the selections of victims is based primarily on cost factors, It may happen that the same transaction is always picked as a victim. As a result this transaction never completes its designated task, thus there is a starvation. We must ensure that a transaction can be picked as a victim only a small (finite number of times). The most common solution is to include the number of rollbacks in cost factor.

Unit 9

Crash Recovery

- ✓ A computer system, like any other device, is subject to failure from a variety of causes: disk crash, power outage, software error, fire etc. In any failure, information may be lost.
- ✓ Therefore, the database system must take actions in advance to ensure that the atomicity and durability properties of transactions are preserved.
- ✓ An integral part of a database system is a **recovery scheme** that can restore the database to the consistent state that existed before the failure.

Failure classification

There are various types of failure that may occur in a system, each of which needs to be dealt with in a different manner. Some major types of failure are as follows:

- ❖ **Transaction failure**
  There are two types of errors that may cause a transaction to fail:
  - ➢ **Logical error**. The transaction can no longer continue with its normal execution because of some internal condition, such as bad input, data not found, overflow, or resource limit exceeded.
  - ➢ **System error**. The system has entered an undesirable state (for example, deadlock), as a result of which a transaction cannot continue with its normal execution. The transaction, however, can be re executed at a later time.
- ❖ **System crash**
  - ➢ A power failure or other hardware or software failure causes the system to crash.
  - ➢ **Fail-stop assumption**: non-volatile storage contents are assumed to not be corrupted by system crash.
  - ➢ Database systems have numerous integrity checks to prevent corruption of disk data.
- ❖ **Disk failure**
  - ➢ A disk block loses its content as a result of either a head crash or failure during a data-transfer operation.
  - ➢ Copies of the data on other disks, or tertiary media, such as DVD or tapes, are used to recover from the failure.

Storage structure

Various data items in the database may be stored and accessed in a number of different storage media. We identified three categories of storage.

- Volatile storage
- Nonvolatile storage
- Stable storage

Volatile storage

- ✓ Data residing in volatile storage does not survive system crashes
- ✓ Examples: main memory, cache memory

Nonvolatile storage

- ✓ Data residing in non-volatile storage survives system crashes
- ✓ Examples: disk, tape, flash memory, non-volatile RAM
- ✓ But may still fail ,losing data

Stable storage

- ✓ A mythical form of storage that survives all failures
- ✓ Approximated by maintaining multiple copies on distinct nonvolatile media.

Log and log records

- ✓ The log is a sequence of log records, recording all the updated activities in the database. In stable storage, logs for each transaction are maintained.
- ✓ Any operation which is performed on the database is recorded on the log.
- ✓ When transaction T_i starts, it registers itself by writing a <T_i **start**> log record
- ✓ Prior to performing any modification to the database, an updated log record is created to reflect that modification. An update log record represented as: <Ti, X, V1, V2> has these fields:

- **Transaction identifier (**T_i): Unique Identifier of the transaction that performed the write operation.
- **Data item (X):** Unique identifier of the data item written.
- **Old value (V1)**: Value of data item prior to write.
- **New value(V2):** Value of data item after write operation.

- ✓ When T_i finishes it last statement, the log record <T_i **commi**t> is written.
- ✓ We assume for now that log records are directly to stable storage (that is, they are not buffered)

Database modification

The database can be modified using two approaches:

1. Deferred database modification
2. Immediate database modification

1. Deferred database modification

- ✓ The deferred database modification scheme records all modifications to the log but defers all the writes to after partial commit.
- ✓ If the system crashes before the transaction completes its execution, or if the transaction abort, then the information on the log is simply ignored.
- ✓ Transactions starts by writing <T_i start> record to the log.
- ✓ A write(X) operations results in a log record < T_i,,X,V> being written, where V is the new value for X.
- ✓ The write is not performed on X at this time, but is deferred.
- ✓ When T_i Partially commits, < T_i, commit> is written to the log.
- ✓ Finally ,the log records are read and used to actually execute the previously deferred writes.

During recovery after crash, a transaction needs to be redone if and only if both <T_i start> and

<T_i commit> are there is the log.

Redoing a transaction T_i,(redo T_i) sets the value of all data items updated by the transaction to the new values.

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 950>	<T_0, A, 950>	<T_0, A, 950>
<T_0, B, 2050>	<T_0, B, 2050>	<T_0, B, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 600>	<T_1, C, 600>
		<T_1 commit>

If log on stable storage at time of crash is as in case:

a) No redo actions need to be taken

b) redo (T_0) must be performed since < T_0 **commit**> is present

c) redo (T_0) must be performed followed by redo (T_1) since < T_0 **commit**> < T_1 **commit**> are present.

If the transaction fails before reaching its commit point, it will have made no changes to database in any way so no UNDO operation is necessary. It may be necessary to REDO effect of the operations of commit transactions from the log because their effect may not have been recorded in the database. Therefore it is also known as **no undo** algorithm**.**

2. Immediate database modification

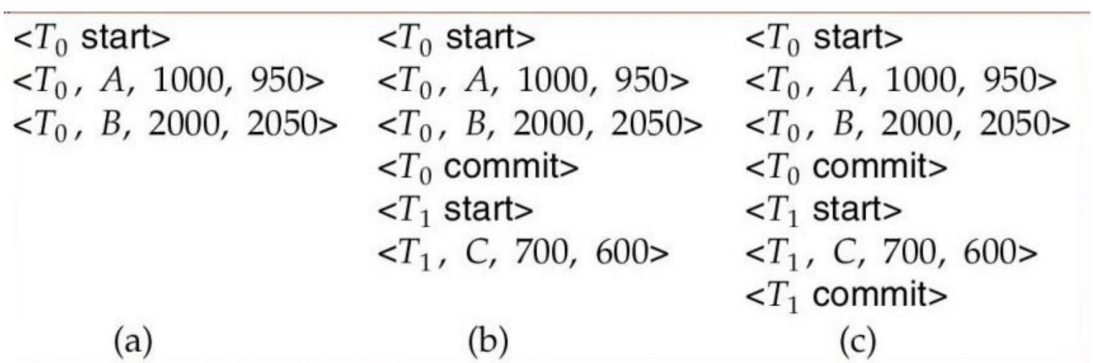
- ✓ Allows database modification while the transaction is still active.
- ✓ Which means all the modifications that is performed before the transaction reaches to commit state are updated to database.
- ✓ Database modifications written by active transactions are called uncommitted modifications.
- ✓ Update log must be written before database items is written.
- ✓ Transaction starts by writing <T_i start> record to log.
- ✓ A write(X) operations result in the log record <T_i X,V1,V2> where V1 is the old value and V2 is the new value . Since undoing may be needed, update logs must have both old value and new value.
- ✓ The write operation on X is recorded in log on disk and is output directly to stable storage without concerning transaction commits or not.
- ✓ In case of failure recovery procedure has two operations instead of one:
    1. undo (T_i) restores the value of all data items updated by T_i to their old values, going backwards from the last record for T_i.
    2. redo(T_i) sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i.

When recovering after failure:

- ✓ Transaction T_i needs to be undone if the log contains the record **<T_i start>** ,but does not contain the record **<T_i commit>**
- ✓ Transaction T_i needs to be redone if the log contains both the record **<T_i start>** and the record **<T_i commit>**
- ✓ Undo operations are performed first, then redo operations.

Example:

Below we show the log as it appears at three instances of time



Recovery actions in each case above are:

a) Undo (T_0):B is restored to 2000 and A to 1000

b) Undo (T_1) and redo (T_0) : C is restored to 700,and A and B are set to 950 and 2050 respectively.

c) Redo (T_0) and Redo (T_1): A and B is set to 950 and 2050 respectively. Then C is set to 600.

Undo and Redo transaction using log

Because all database modifications must be preceded by the creation of a log record, the system has available both the old value prior to the modification of the data item and new value that is to be written for data item. This allows system to perform redo and undo operations as appropriate:

Undo: using a log record sets the data item specified in log record to old value. (used for immediate database modification only)

Redo: using a log record sets the data item specified in log record to new value.

Checkpoint

The Checkpoint is used to declare a point before which the DBMS was in a consistent state, and all transactions were committed.

Use of Checkpoints

When a system crash occurs, user must consult the log. In principle, that need to search the entire log to determine this information. There are two major difficulties with this approach:

- ✓ The search process is time-consuming.
- ✓ Most of the transactions that, according to our algorithm, need to be redone have already written their updates into the database. Although redoing them will cause no harm, it will cause recovery to take longer.

To reduce these types of overhead, user introduce checkpoints.

A log record of the form <checkpoint L> is used to represent a checkpoint in log where L is a list of transactions active at the time of the checkpoint.

When a checkpoint log record is added to log all the transactions that have committed before this checkpoint have <Ti commit> log record before the checkpoint record.

Transactions are not allowed to perform any update actions, such as writing to a buffer block or writing a log record, while a checkpoint is in progress

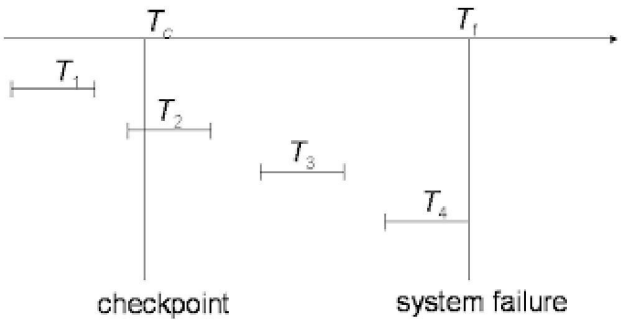
During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i.

- ✓ Scan backwards from end of log to find the most recent <**checkpoint** *L*> record
- ✓ continue scanning backwards till a record **<T_i start>** is found.
- ✓ Need to consider the part of log following above **<T_i start>** record. Earlier part of log can be ignored during recovery .

After the transaction T_i identified, the redo and undo operations to be applied to the T_i and all T_j that started execution after transaction T_i .

For all transactions (starting from T_i or later)

- with no **<T_i commit>,** execute **undo** (T_i) *(Done only in case of immediate database modification)*
- with **<T_i commit>,** execute **redo** (T_i)



- ✓ T_1 can be ignored (updates already output to disk due to checkpoint)
- ✓ T_2 and T_3 redone
- ✓ T_4 undone

Concurrency control and recovery

Concurrency control means that multiple transactions can be executed at the same time and then the interleaved logs occur. But there may be changes in transaction results so maintain the order of execution of those transactions.

During recovery, it would be very difficult for the recovery system to backtrack all the logs and then start recovering.

Recovery with concurrent transactions can be done in the following four ways.

1. Interaction with concurrency control
2. Transaction rollback
3. Checkpoints
4. Restart recovery

1. Interaction with concurrency control:

The recovery scheme depends greatly on the concurrency-control scheme that is used. To roll back a failed transaction, we must undo the updates performed by the transaction. Suppose that a transaction T_0 has to be rolled back, and a data item Q that was updated by T_0 has to be restored to its old value. Using the log-based schemes for recovery, we restore the value by using the undo information in a log record.

Suppose now that a second transaction T_1 has performed yet another update on Q before T_0 is rolled back. Then, the update performed by T_1 will be lost if T_0 is rolled back.

Therefore, we require that, if a transaction T has updated a data item Q, no other transaction may update the same data item until T has committed or been rolled back. We can ensure this requirement easily by using strict two-phase locking—that is, two-phase locking with exclusive locks held until the end of the transaction.

2. Transaction rollback:

We roll back a failed transaction, Ti, by using the log. The system scans the log backward; for every log record of the form <Ti, Xj, V1, V2> found in the log, the system restores the data item Xj to its old value V1. Scanning of the log terminates when the log record <Ti, start> is found.

Scanning the log backward is important, since a transaction may have updated a data item more than once. As an illustration, consider the pair of log records

<T_1 start>
<T_1, A, 10, 20>
<T_1, A, 20, 30>

The log records represent a modification of data item A by T_1, followed by another modification of A by T_1. Scanning the log backward sets A correctly to 10.

3. Checkpoint

The checkpoint is used to declare the point before which the DBMS was in the consistent state, and all the transactions were committed. This reduces the amount of work during recovery.

By creating checkpoints, recovery operations don't need to start from the very beginning. Instead, they can begin from the most recent checkpoint, thereby considerably speeding up the recovery process.

Since we assumed no concurrency, it was necessary to consider only the following transactions during recovery:

- ✓ Those transactions that started after the most recent checkpoint
- ✓ The one transaction, if any, that was active at the time of the most recent checkpoint

The situation is more complex when transactions can execute concurrently, since several transactions may have been active at the time of the most recent checkpoint.

In a concurrent transaction-processing system, we require that the checkpoint log record be of the form <checkpoint L>, where L is a list of transactions active at the time of the checkpoint.

Transactions are not allowed to perform any update actions, such as writing to a buffer block or writing a log record, while a checkpoint is in progress.

4. Restart Recovery

When the system recovers from a crash, it constructs two lists: The undo-list consists of transactions to be undone, and the redo-list consists of transactions to be redone.

The system constructs the two lists as follows:

1. Initially, they are both empty.
2. The system scans the log backward, examining each record, until it finds the first <checkpoint> record, then
   i. For each record found of the form $<T_i$ commit>, it adds T_i to redo-list.
   ii. For each record found of the form $<T_i$ start>, if T_i is not in redo-list, then it adds T_i to undo-list.

COMMIT Operations Performance Optimization

During COMMITs the changes of the transaction are persisted to disk and made visible for other transactions.

Factors can influence the time of COMMIT operations such as Disk I/O write performance to logs, Synchronous system replication mode, Synchronous table replication, High amount of active versions, Integrated liveCache and SAP HANA bugs.

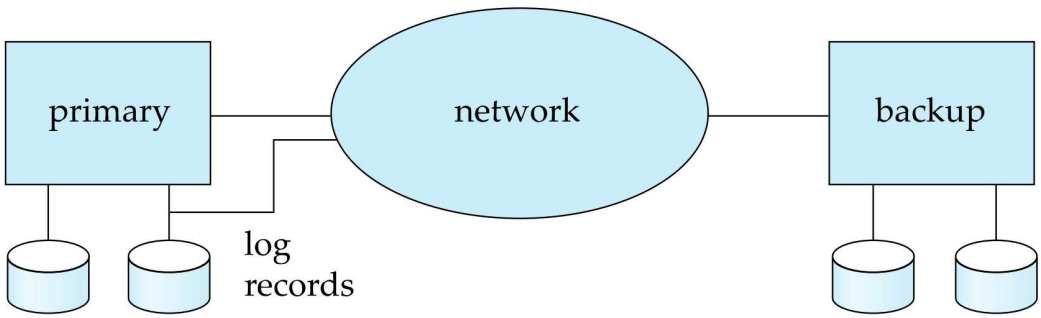
<table>
<tr><th>Reason</th><th>Details</th></tr>
<tr><td>Disk I/O write performance to logs</td><td>During COMMITs the change information is written to the log files. If writing down the data is slow, the COMMIT times can increase.</td></tr>
<tr><td>Synchronous system replication mode</td><td>If a synchronous system replication mode is activated, changes are propagated to the replication side during a COMMIT, and the local session has to wait for an acknowledgement. If you suffer from long COMMIT times in a system with activated synchronous system replication, you should at first check if a switch to a less critical system replication mode (e.g. SYNC -> SYNCMEM or SYNCMEM -> ASYNC) improve the performance. If yes, you can analyze if there are unnecessary replication delays, e.g. due to limited network bandwidth or high latency times.</td></tr>
<tr><td>Synchronous table replication</td><td>If a transaction performs changes on a table with activated optimistic synchronous table replication (OSTR), the replicas need to be refreshed during COMMIT. This can have an adverse impact on the COMMIT performance.</td></tr>
<tr><td>High amount of active versions</td><td>The COMMIT performance can be significantly impacted by a high amount of active versions.</td></tr>
<tr><td>Integrated liveCache</td><td>If a SAP HANA integrated liveCache is used, the SAP HANA COMMIT performance can be impacted by related COMMITs on liveCache side.
The time for the 'Kernel-Commit' method is part of the SAP HANA COMMIT time. Other related times like 'Flush-Cache', 'Commit-Invalidate-Callback' and 'Validate-Callback' have to be considered on top of the SAP HANA COMMIT time.
During a liveCache COMMIT a potentially large amount of data has to be flushed, so in case of high COMMIT times you should always consider the amount of flushed data to judge if it is more an I/O issue or a data volume issue.</td></tr>
<tr><td>SAP HANA bugs</td><td>The following SAP HANA bugs can be responsible for increased COMMIT times:
<table>
<tr><th>Impacted Revision</th><th>Details</th></tr>
<tr><td>122.062.00.000</td><td>Overhead in internal COMMIT processing design</td></tr>
</table>
</td></tr>
</table>

If COMMITs take very long, a termination with error **"snapshot timestamp synchronization failed"** is possible.

High Availability Using Remote Backup System

- ✓ Traditional transaction-processing systems are centralized or client–server systems. Such systems are vulnerable to environmental disasters such as fire, flooding, or earthquakes.
- ✓ So that there is a need for transaction-processing systems that can function in spite of system failures or environmental disasters and such systems must provide high availability.
- ✓ **We can achieve high availability** by performing transaction processing at one site, called the primary site, and having **a remote backup site** where all the data from the primary site are replicated. The remote backup site is sometimes also called the secondary site.
- ✓ The remote site must be kept synchronized with the primary site, as updates are performed at the primary.
- ✓ We achieve synchronization by sending all log records from primary site to the remote backup site. The remote backup site must be physically separated from the primary—for example, we can locate it in a different state—so that a disaster at the primary does not damage the remote backup site.

Figure below shows the architecture of a remote backup system.



- ✓ When the primary site fails, the remote backup site takes over processing. It performs recovery, using its copy of the data from the primary, and the log records received from the primary. In effect, the remote backup site is performing recovery actions that would have been performed at the primary site when the latter recovered.
- ✓ Once recovery has been performed, the remote backup site starts processing transactions.

Several issues must be addressed in designing a remote backup system:

Detection of failure: It is important for the remote backup system to detect when the primary has failed. Failure of communication lines can fool the remote backup into believing that the primary has failed. To avoid this problem, we maintain several communication links with independent modes of failure between the primary and the remote backup.

Transfer of control: When the primary fails, the backup site takes over processing and becomes the new primary. When the original primary site recovers, it can either play the role of remote backup, or take over the role of primary site again. In either case, the old primary must receive a log of updates carried out by the backup site while the old primary was down.

Time to recover: If the log at the remote backup grows large, recovery will take a long time. The remote backup site can periodically process the redo log records that it has received and can perform a checkpoint, so that earlier parts of the log can be deleted.

A hot-spare configuration can make takeover by the backup site almost instantaneous. In this configuration, the remote backup site continually processes redo log records as they arrive, applying the updates locally. As soon as the failure of the primary is detected, the backup site completes recovery by rolling back incomplete transactions; it is then ready to process new transactions.

Time to commit: To ensure that the updates of a committed transaction are durable, a transaction must not be declared committed until its log records have reached the backup site. This delay can result in a longer wait to commit a transaction, and some systems therefore permit lower degrees of durability.

The degrees of durability can be classified as follows:

One-safe: A transaction commits as soon as its commit log record is written to stable storage at the primary site.

Two-very-safe: A transaction commits as soon as its commit log record is written to stable storage at the primary and the backup site.

Two-safe: This scheme is the same as two-very-safe if both primary and backup sites are active. If only the primary is active, the transaction is allowed to commit as soon as its commit log record is written to stable storage at the primary site.

Unit 10

Emerging Trend in Databases

NoSQL Databases

- ✓ NoSQL stands for Not only SQL.
- ✓ NoSQL is a non-relational database that is used to store the data in the nontabular form.
- ✓ It is designed to handle and store large volumes of unstructured and semi-structured data.
- ✓ Unlike traditional relational databases that use tables with pre-defined schemas to store data, NoSQL databases use flexible data models that can adapt to changes in data structures and are capable of scaling horizontally to handle growing amounts of data.

Categories of NoSQL

NoSQL databases are generally classified into four main **categories:**

1. **Document based databases:** The document-based database is a non-relational database. Instead of storing the data in rows and columns (tables), it uses the documents to store the data in the database. A document database stores data in JSON, BSON, or XML documents.

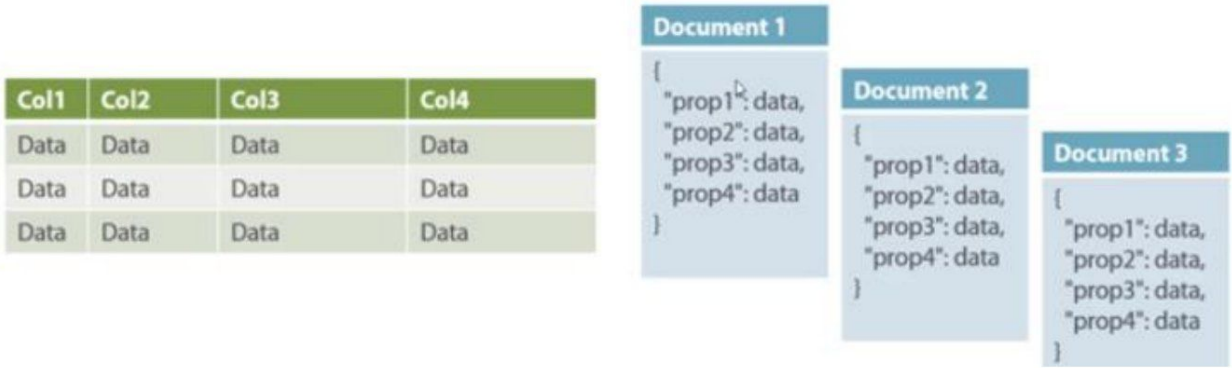


Figure: relational vs document

- ✓ In above diagram, left side we can see that we have rows and columns, and in the right, we have a document database which has a similar structure to JSON.
- ✓ Documents in the database has a flexible schema. It means the documents in the database need not be the same schema.

2. **Key-value stores:**
   - ✓ Every data element in the database is stored in key-value pairs.
   - ✓ The data can be retrieved by using a unique key allotted to each element in the database. The values can be simple data types like strings and numbers or complex objects.
   - ✓ Redis, Dynamo, Riak are some NoSQL examples of key-value store Databases.
   - ✓ For example, a key-value pair may contain a key like "Name" associated with a value like "Ramesh".

Key	Value
Name	Ramesh
Address	Kathmandu
Age	30
Occupation	Teacher

3. **Column-Oriented databases**

   These databases store data in columns instead of rows and are optimized for managing large amounts of data.

   That means when we want to run analytics on a small number of columns, you can read those columns directly without consuming memory with the unwanted data.

   They deliver high performance on aggregation queries like SUM, COUNT, AVG, MIN etc. as the data is readily available in a column.

   HBase, Cassandra, HBase, Hypertable are NoSQL query examples of column based database.

ColumnFamily			
Row Key	Column Name		
	Key	Key	Key
	Value	Value	Value
	Column Name		
	Key	Key	Key
	Value	Value	Value

Column based NoSQL database

Example:

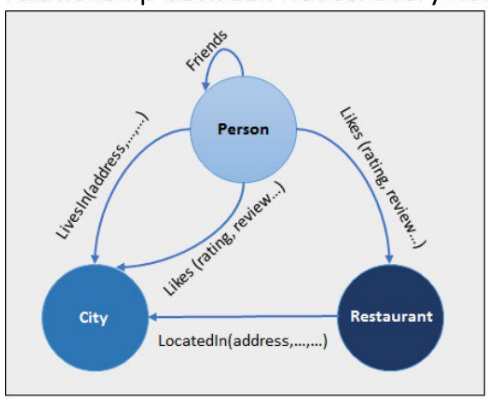
Row oriented (Relational)

Students		
ID	**First name**	**Last name**
1	Luna	Lovegood
2	Hermione	Granger
3	Ron	Weasley

Column oriented

Students		
ID	**First name**	**Last name**
1	Luna	Lovegood
2	Hermione	Granger
3	Ron	Weasley

4. **Graph databases:** A graph type database stores entities as well the relations amongst those entities. The entity is stored as a node with the relationship as edges. An edge gives a relationship between nodes. Every node and edge has a unique identifier.



- ✓ Graph based database mostly used for social networks.
- ✓ Neo4J, Infinite Graph, OrientDB, FlockDB are some popular graph-based databases.

Characteristics of No SQL

Although there are different ways that can be incorporated to understand how NoSQL databases work, we will now look at some of the most common features that define a basic NoSQL database.

- ❖ **Complex-free working:** Unlike SQL databases, NoSQL databases are not complicated. They store data in an unstructured or a semi-structured form that requires no relational or tabular arrangement. Perhaps they are easier to use and can be accomplished by all.
- ❖ **Independent of Schema:** NoSQL databases are independent of schemas which implies that they can be run over without any predetermined schemas.
- ❖ **Better Scalability:** NoSQL databases scale easily, meaning they can grow with data demands. They're also flexible in how they store data, making it easier to adapt to changing needs. This makes them ideal for managing diverse and large datasets effectively.
- ❖ **Flexible to accommodate:** Since such databases can accommodate heterogeneous data that requires no structuring, they are claimed to be flexible in terms of their usage and reliability.
- ❖ **Durable:** NoSQL databases are highly durable as they can accommodate data ranging from heterogeneous to homogeneous. They can also incorporate unstructured data that requires no query language. Undoubtedly, these databases are durable and efficient.

Advantages of NoSQL:

There are many advantages of working with NoSQL databases such as MongoDB and Cassandra. The main advantages are high scalability and high availability.

- **Scalability:** NoSQL databases are highly scalable, which means that they can handle large amounts of data and traffic with ease.
- **Flexibility**: NoSQL databases are designed to handle unstructured or semi-structured data, which means that they can accommodate dynamic changes to the data model.
- **High availability**: The auto, replication feature in NoSQL databases makes it highly available because in case of any failure data replicates itself to the previous consistent state.
- **Performance:** NoSQL databases are designed to handle large amounts of data and traffic, which means that they can offer improved performance compared to traditional relational databases.
- **Cost-effectiveness**: NoSQL databases are often more cost-effective than traditional relational databases, as they are typically less complex and do not require expensive hardware or software.

Disadvantages of NoSQL

NoSQL has the following disadvantages.

- **Lack of standardization:** There are many different types of NoSQL databases, each with its own unique strengths and weaknesses. This lack of standardization can make it difficult to choose the right database for a specific application.
- **Lack of ACID compliance:** NoSQL databases are not fully ACID-compliant, which means that they do not guarantee the consistency, integrity, and durability of data. This can be a drawback for applications that require strong data consistency guarantees.
- **Narrow focus:** NoSQL databases have a very narrow focus as it is mainly designed for storage but it provides very little functionality. Relational databases are a better choice in the field of Transaction Management than NoSQL.
- **Lack of support for complex queries**: NoSQL databases are not designed to handle complex queries, which means that they are not a good fit for applications that require complex data analysis or reporting.
- **Lack of maturity:** NoSQL databases are relatively new and lack the maturity of traditional relational databases. This can make them less reliable and less secure than traditional databases.
- **Management challenge:** The purpose of big data tools is to make the management of a large amount of data as simple as possible. But it is not so easy. Data management in NoSQL is much more complex than in a relational database.
- **Large document size:** Some database systems like MongoDB and CouchDB store data in JSON format. This means that documents are quite large (BigData, network bandwidth, speed).

Object Relational Mapping (ORM)

- ✓ Object-relational mapping is a technique that maps objects onto a relational database, converting the object into data that can be stored, retrieved and reconstructed when needed.
- ✓ In another way, we can see the ORM as the layer that connects object oriented programming (OOP) to relational databases.
- ✓ Through ORM, changes made to an object are shared with the database, which then updates the data to reflect these changes.

Why use an ORM(Object- Relational Mapping)?

- ✓ Databases and object-oriented programs use different programming languages and store data differently, which can lead to communication difficulties between languages.
- ✓ ORM maps objects onto the relational database table so that when we save an object to the database, it's broken down into smaller parts that the database can store. These parts are then saved in a logical order. When we access the object again, the program can retrieve the parts from the database to reconstruct the object.

How Does Object-Relational Mapping Work?

- ✓ ORM produces a structured map that reveals the relationships between objects and tables, or data, without getting into the details of the data structure. Objects are transformed into simpler, manageable pieces that the database can easily process and store for later retrieval.
- ✓ ORM connects object-oriented programming languages or applications with a relational database, communicating any changes made to an object to the database, which then alters the data accordingly.

Object-Relational Mapping Example

Imagine a dog model (i.e. class). Each dog object has the following attributes: name, age and breed. Here is how the dog class and dog objects look in Java.

```java
public class Dog {

  private String name;
  private String age;
  private String breed;

  public Dog(String name, String age, String breed) {

    this.name = name;
    this.age = age;
    this.breed = breed;
  }

  public String getName() {
    return name;
  }

  public String getAge() {
    return age;
  }

  public String getBreed() {
    return breed;
  }

  public void setName(String name) {
    this.name = name;
  }

  public void setAge(String age) {
    this.age = age;
  }

  public void setBreed(String breed) {
    this.breed = breed;
  }

}
```

A Java class that defines a dog by its name, age and parameter.

To store all of these dog objects in the database, we create a table to represent the dog class. The table rows correspond to dog instances and the columns correspond to the dog class' attributes. When we create a dog record in the table, it's automatically assigned an ID value.

ID	Name	Age	Breed
1	Bailey	5	Labrador Retriever
2	Max	8	Poodle
3	Charlie	3	Boxer
4	Luna	10	Husky
5	Rocky	4	German Shepard

The table stores the attributes of the different dog classes.

Advantages of Object-Relational Mapping

1. **Handles the logic required to interact with database:** ORM tools automatically generate the necessary SQL queries and handle database transactions, so developers don't need to write SQL code themselves.
2. **Speed up development time for teams:** ORM tools abstract away the low-level details of working with a database, which allows developers to focus on writing business logic instead of SQL queries. This shift results in faster development times because developers can write and test code more quickly.
3. **Decreases the cost of development**: Because ORM tools automate many of the repetitive tasks involved in working with databases, developers can write more code in less time, thereby leading to a lower cost of development.
4. **Improves security:** ORM tools provide built-in security features that help prevent SQL injection attacks, which are a common security vulnerability in database-driven applications.
5. **Requires less code:** ORM tools allow developers to interact with databases using a higher-level programming language like Java or Python instead of writing SQL queries. This makes the code easier for other team members to read; the code is also easier to maintain because developers don't need to write as much SQL code.

Disadvantages of Object-Relational Mapping

1. **Steep learning curve**

- ✓ ORM tools can have a steep learning curve, especially for developers who are not familiar with object-oriented programming concepts.
- ✓ Developers have to understand how to map their object-oriented code to the database schema, which can require a significant amount of time and effort to learn.
- ✓ Additionally, some ORM tools have their own specific syntax and APIs, which developers must learn to use.

2. **Struggles with complex queries**

- ✓ ORM tools simplify working with databases, but they might not be as good as writing SQL directly, especially for complex queries.
- ✓ Some ORM tools don't support advanced SQL features, which can be a limitation for certain applications.

3. **Slow performance speed**

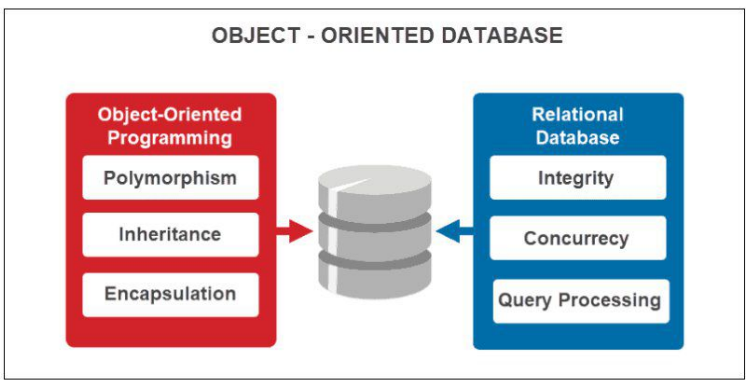
- ✓ ORM tools generate SQL queries automatically, which can result in queries that are less efficient than those written by an experienced SQL developer. This lack of efficiency can result in slower performance for database-driven applications.
- ✓ Additionally, ORM tools often generate more database queries than necessary, which can slow down performance even further.

ORM tools

- ➢ ***JAVA:*** Hibernate, Apache OpenJPA, EclipseLink, jOOQ, Oracle TopLink
- ➢ ***PYTHON:*** Django, web2py, SQLObject, SQLAlchemy

Object Oriented Databases

- ✓ Object-oriented databases add database functionality to object programming languages, creating more manageable code bases.
- ✓ An object-oriented database is managed by an object-oriented database management system (OODBMS). The database combines object-oriented programming concepts with relational database principles.
- ✓ Objects are the basic building block and an instance of a class, where the type is either built-in or user defined.
- ✓ Classes provide a schema or blueprint for objects, defining the behavior.
- ✓ Methods determine the behavior of a class.
- ✓ Pointers help access elements of an object database and establish relations between objects.



Object-Oriented Programming Concepts

Object-oriented databases closely relate to object-oriented programming concepts. The four main ideas of object-oriented programming are:

- ➢ Polymorphism
- ➢ Inheritance
- ➢ Encapsulation
- ➢ Abstraction

Example of Object-oriented databases

- ➢ GemStone/S
- ➢ ObjectDB
- ➢ ObjectDatabase++
- ➢ Objectivity/DB
- ➢ ObjectStore

Advantages

- ✓ Complex data and a wider variety of data types compared to MySQL data types.
- ✓ Easy to save and retrieve data quickly.
- ✓ Seamless integration with object-oriented programming languages.
- ✓ Easier to model the advanced real-world problems.
- ✓ Extensible with custom data types.

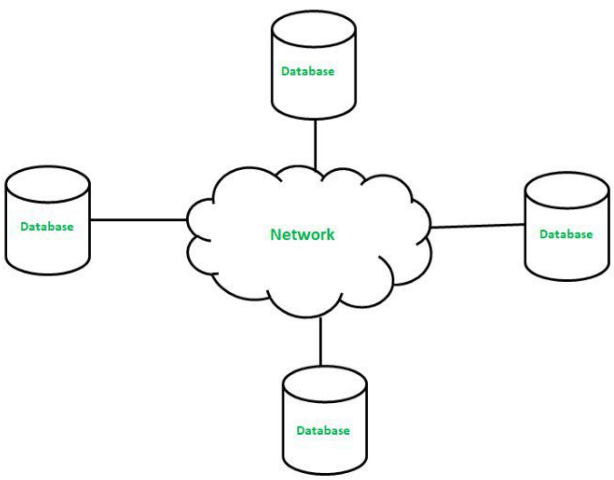
Disadvantages

- ✓ Not as widely adopted as relational databases.
- ✓ No universal data model. Lacks theoretical foundations and standards.
- ✓ Does not support views.
- ✓ High complexity causes performance issues.
- ✓ An adequate security mechanism and access rights to objects do not exist.

Distributed Databases

A distributed database is basically a type of database which consists of multiple databases that are connected with each other and are spread across different physical locations. The data that is stored in various physical locations can thus be managed independently of other physical locations. The communication between databases at different physical locations is thus done by a computer network.

This may be required when a particular database needs to be accessed by various users globally. It needs to be managed such that for the users it looks like one single database.



Types:

1. Homogeneous Database:

In a homogeneous database, all different sites store databases identically. The operating system, database management system, and the data structures used – all are the same at all sites. Hence, they're easy to manage.

2. Heterogeneous Database:

In a heterogeneous distributed database, different sites can use different schema and software that can lead to problems in query processing and transactions. Also, a particular site might be completely unaware of the other sites. Different computers may use a different operating system, different database application. They may even use different data models for the database.

Distributed Data Storage

There are two ways in which data can be stored at different sites. These are,

i) **Replication:** As the name suggests, the system stores copies of data at different sites. If an entire database is available on multiple sites, it is a fully redundant database.

ii) **Fragmentation**

In Fragmentation, the relations are fragmented, which means they are split into smaller parts. Each of the fragments is stored on a different site, where it is required. In this, the data is not replicated, and no copies are created.

The prerequisite for fragmentation is to make sure that the fragments can later be reconstructed into the original relation without losing any data.

There are two types of fragmentation,

Horizontal Fragmentation – Splitting by rows.

Vertical fragmentation – Splitting by columns.

- ✓ **Horizontal Fragmentation**

The relation schema is fragmented into group of rows, and each group is then assigned to one fragment.

- ✓ **Vertical Fragmentation**

The relation schema is fragmented into group of columns, called smaller schemas. These smaller schemas are then assigned to each fragment.

Each fragment must contain a common candidate key to guarantee a lossless join.

Advantages of Distributed Database

Better Reliability: Distributed databases offers better reliability than centralized databases. When database failure occurs in a centralized database, the system comes to a complete stop. But in the case of distributed databases, the system functions even when a failure occurs, only performance-related issues occur which are negotiable.

Modular Development: It implies that the system can be expanded by adding new computers and local data to the new site and connecting them to the distributed system without interruption.

Lower Communication Cost: Locally storing data reduces communication costs for data manipulation in distributed databases. In centralized databases, local storage is not possible.

Better Response Time: As the data is distributed efficiently in distributed databases, this provides a better response time when user queries are met locally. While in the case of centralized databases, all of the queries have to pass through the central machine which increases response time.

Disadvantages of Distributed Database

Costly Software: Maintaining a distributed database is costly because we need to ensure data transparency, coordination across multiple sites which requires costly software.

Large Overhead: Many operations on multiple sites require complex and numerous calculations, causing a lot of processing overhead.

Improper Data Distribution: If data is not properly distributed across different sites, then responsiveness to user requests is affected. This in turn increases the response time.

Distributed Ledger Technology

- ✓ Distributed Ledger Technology (DLT) is also known as a "shared ledger" or simply distributed ledger.
- ✓ It is a digital system that lets users and systems record transactions related to assets. A distributed ledger technology stores the information at multiple locations at any given point of time.
- ✓ DLT, unlike traditional databases, does not have any central place to store information. This is what differentiates it from a traditional database.
- ✓ At the core, DLT originates from the peer-to-peer(P2P) network. In any P2P network, peers communicate with each other without the need for a centralized entity. Technically, distributed ledger technology is possible through a peer-to-peer network.

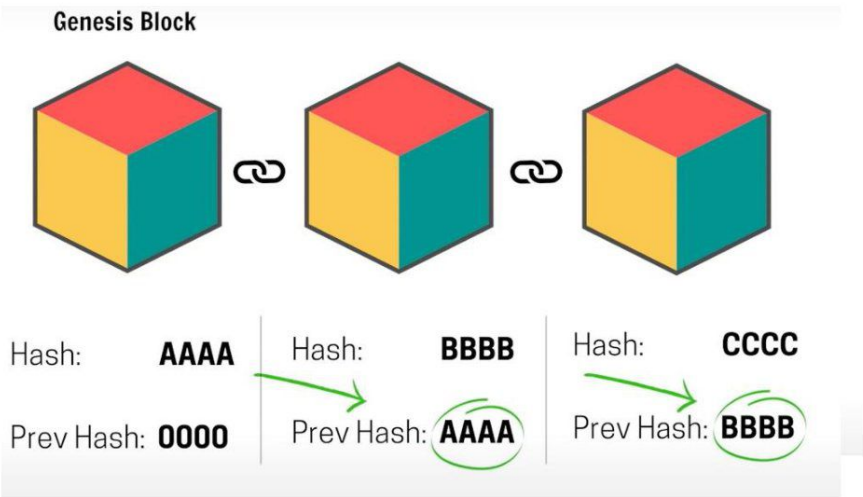
Key features of distributed ledger technology:

- ➢ **Decentralized:** It is a decentralized technology and every node will maintain the ledger, and if any data changes happen, the ledger will get updated. The process of updating takes place independently at each node. Even small updates or changes made to the ledger are reflected and the history of that change is sent to all participants in a matter of seconds.
- ➢ **Immutable:** Distributed ledger uses cryptography to create a secure database in which data once stored cannot be altered or changed. In this process, every node or contributor of the Ledger will try to verify the transactions with the various consensus algorithms or voting. The voting or participation of all the nodes depends on the rules of that Ledger. With Bitcoin, the Proof of Work consensus mechanism is used for the participation of each node.
- ➢ **Distributed:** This technology doesn't rely on a single central authority or server to manage its database, which makes it very open and transparent.
- ➢ **Fault Tolerance:** Distributed ledgers are highly fault-tolerant because of their decentralized nature. If one node or participant fails, the data remains available on other nodes.
- ➢ **Security:** Distributed ledgers are highly secure because of their cryptographic nature. Every transaction is recorded with a cryptographic signature that ensures that it cannot be altered. This makes the technology highly secure and resistant to fraud.

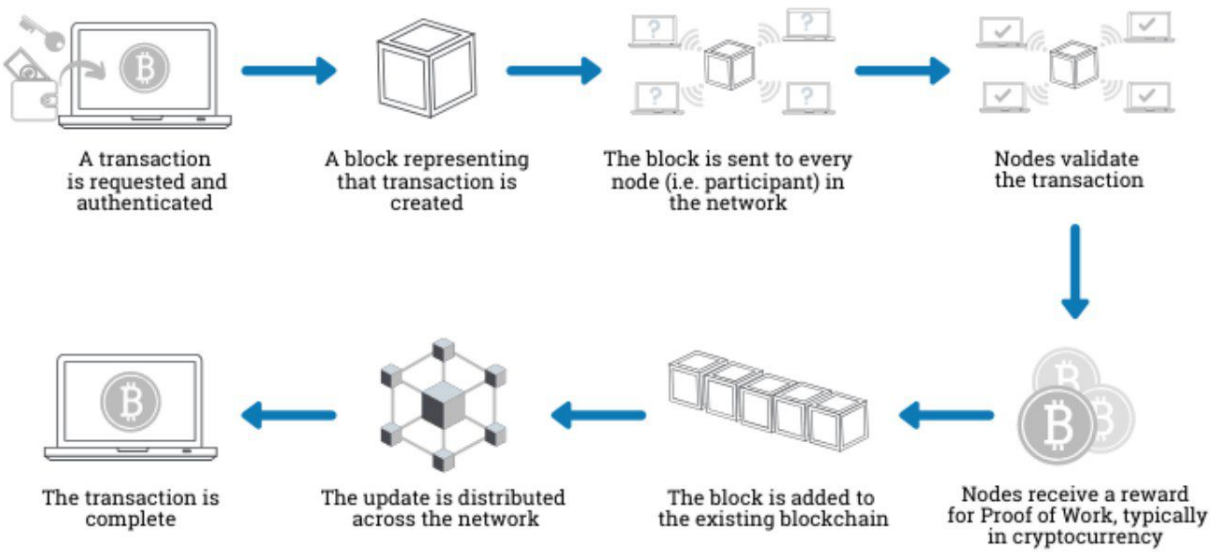
Blockchain

One of the technologies that are under the umbrella of DLT is blockchain. In 2008, an individual under the pseudonym Satoshi Nakamoto authored and released a white paper titled "Bitcoin: A Peer-to-Peer Electronic Cash System." This document introduced the concept of a distributed blockchain, expanding upon existing models. Nakamoto's proposal laid the foundation for decentralized digital currency systems, with Bitcoin being the first implementation.

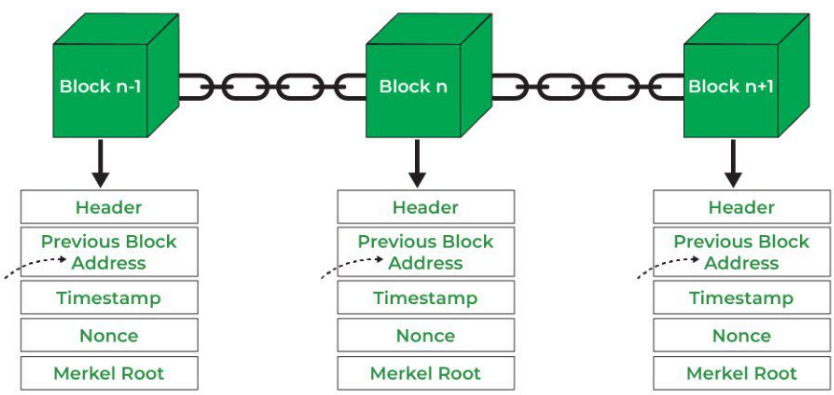
- ✓ A blockchain is a chain of blocks where each block stores information and is associated with a unique hash address, serving as proof of valid transactions.
- ✓ It is a distributed ledger that stores data, including transactions, and is publicly shared across all nodes within the network.
- ✓ Transactions are verified by a network of participating computers within the blockchain.
- ✓ Thus, there is no involvement of any central authority or middlemen, satisfying the decentralization property of blockchain.
- ✓ Once a block of information is created in the chain, it can't be changed or deleted. This makes the blockchain very secure and trustworthy.



How does a transaction get into the blockchain?



Blockchain architecture



1. **Header:** It is used to identify the particular block in the entire blockchain. It handles all blocks in the blockchain.
2. **Previous Block Address/ Hash:** It is a reference to the hash of the previous (parent) block in the chain.
3. **Timestamp:** It is a system verify the data into the block and assigns a time or date of creation for digital documents.
4. **Nonce:** A nonce is a number used only once, crucial for the proof of work in a block. Miners test numerous nonces per second, aiming to find one that meets the required criteria and is valid.
5. **Merkle Root:** A Merkle Root is a data structure that organizes different blocks of data into a tree. It creates a digital fingerprint of all the transactions within a block, enabling users to verify transaction inclusion in a block.

Blockchain properties

- **Decentralization:** One of the fundamental properties of blockchain is decentralization. Instead of relying on a central authority (like a bank or a government), blockchain networks are distributed across a network of nodes. Each node typically maintains a copy of the entire blockchain, and decisions are made through consensus mechanisms rather than by a single central entity.
- **Immutability**: Once data is recorded on a blockchain, it's extremely difficult to alter or delete it. Each block contains a cryptographic hash of the previous block, creating a chain of blocks that are linked together. Any attempt to alter the data in a block would require changing all subsequent blocks in the chain, which is practically infeasible due to the computational resources required. This property ensures the integrity of the data stored on the blockchain.
- **Transparency:** Blockchain networks are often transparent, meaning that the data stored on the blockchain is visible to all participants in the network. Anyone can view the entire transaction history, which promotes trust and accountability within the network.
- **Security:** Blockchain networks use cryptographic techniques to secure transactions and maintain the integrity of the data. Consensus mechanisms such as Proof of Work (PoW) ensure that only valid transactions are added to the blockchain, and the decentralized nature of the network makes it resistant to censorship and attacks.
- **Irreversibility:** Once a transaction is confirmed and added to the blockchain, it becomes practically irreversible. This property is particularly valuable in financial transactions, where it eliminates the need for intermediaries and reduces the risk of fraud.
- **Consensus Mechanisms:** Blockchain networks rely on consensus mechanisms to achieve agreement among nodes on the validity of transactions and the state of the blockchain.
- **Smart Contracts:** Smart contracts are simply programs stored on a blockchain that run when predetermined conditions are met. They typically are used to automate the execution of an agreement so that all participants can be immediately certain of the outcome, without any intermediary's involvement or time loss.
- **Auditability:** Blockchain operates on the Unspent Transaction Output (UTXO) model, where each transaction refers to previous unspent transactions. Once a transaction is recorded in the blockchain, the referenced unspent transactions become spent. This process enables easy tracking of transactions and ensures data integrity between them.

Application areas of blockchain:

- **Banking:** Blockchain enables secure, low-cost peer-to-peer transactions, eliminating the need for intermediaries like banks. Cryptocurrencies facilitate borderless transactions.
- **Cyber Security**: Blockchain enhances security by quickly detecting and preventing cyber threats through decentralized, encrypted data storage. It ensures data integrity while protecting privacy.
- **Supply Chain Management:** Blockchain improves transparency and traceability in supply chains, enabling seamless verification of transactions across the entire chain. It reduce issues like counterfeiting and ensures product authenticity.
- **Healthcare**: Blockchain facilitates secure and instant access to medical data by decentralizing storage and encryption. It prevents data manipulation and enhances privacy and interoperability of medical records.
- **Government:** Blockchain revolutionizes voting systems by enabling secure, anonymous voting and ensuring the accuracy and transparency of election processes. It prevents fraud and manipulation, empowering citizens in democratic participation.

SOME WIDELY USED TERMS IN BLOCKCHAIN

Proof of Work: It is a consensus mechanism used in blockchain networks to validate and confirm transactions and produce new blocks. In this system, miners compete to solve complex mathematical puzzles, requiring significant computational power. The first miner to solve the puzzle broadcasts their solution to the network. Other miners verify the solution, and if it's correct, the new block is added to the blockchain.

Consensus mechanism /Algorithm:A consensus mechanism is a procedure through which all peers of the blockchain network reach a common agreement about the current state of the distributed ledger. In this way, consensus mechanisms achieve reliability in the blockchain network and establish trust between unknown peers in a distributed computing environment. Essentially, the consensus protocol ensures that every new block added to the blockchain is the one and only version of the truth agreed upon by all nodes in the blockchain

Miners: Miners are individuals or entities that participate in the process of validating and adding transactions to the blockchain. They play a crucial role in maintaining the integrity and security of the blockchain network. Miners use specialized hardware and software to solve complex mathematical puzzles in a process known as mining.

When a miner successfully solves a puzzle, they create a new block of transactions, which is then added to the blockchain. In return for their efforts, miners are rewarded with cryptocurrency tokens, such as Bitcoin, as well as transaction fees associated with the transactions included in the block.

Miners compete with each other to be the first to solve the puzzle and add a new block to the blockchain. This competition ensures that no single entity has control over the network and helps to prevent fraudulent or malicious activities.

Cryptocurrency

- ✓ Cryptocurrency refers to a digital or virtual form of currency that utilizes cryptography for security and operates independently of a central authority, such as a government or financial institution.
- ✓ Cryptocurrencies typically operate on decentralized networks based on blockchain technology, which is a distributed ledger that records all transactions across a network of computers.
- ➢ The most well-known cryptocurrency is Bitcoin, which was created in 2009 by an unknown person or group using the pseudonym. Other examples of cryptocurrency are Ethereum, Litecoin Ripple
- ✓ Cryptocurrencies can be stored in digital wallets, which provide users with a private key to access and manage their funds securely.

People can acquire cryptocurrencies through various means:

- ❖ **Mining:** Cryptocurrencies are generated through a process known as mining. Miners use specialized computer systems to solve complex mathematical puzzles. Upon successful completion, they are rewarded with bitcoins or other cryptocurrencies.
- ❖ **Buying, Selling, and Storing:** Users have the option to purchase cryptocurrencies from centralized exchanges, brokers, or individual sellers. Similarly, they can sell cryptocurrencies through these entities. Cryptocurrencies are typically stored in digital wallets, which may be online, offline, hardware-based, or software-based.
- ❖ **Investing:** Cryptocurrencies can be transferred between digital wallets for various purposes, including:
  - ✓ Purchasing goods and services.
  - ✓ Trading on cryptocurrency exchanges.
  - ✓ Converting them into traditional fiat currencies.

Benefits of using cryptocurrency

- ➢ **Decentralization**: Cryptocurrencies operate on decentralized networks, typically utilizing blockchain technology. This means that they are not controlled by any single authority, such as a government or financial institution. This decentralization can lead to increased transparency, security, and resilience against censorship or manipulation.
- ➢ **Lower Transaction Fees:** Traditional financial transactions often involve intermediary institutions that charge fees for their services. Cryptocurrency transactions, on the other hand, can be conducted peer-to-peer, reducing or eliminating the need for intermediaries and lowering transaction fees.
- ➢ **Fast and Borderless Transactions:** Cryptocurrency transactions can be processed quickly, often within minutes, regardless of geographic location. This can be particularly advantageous for international transfers, as cryptocurrencies are not subject to the same delays and fees associated with traditional cross-border transactions.

- **Financial Inclusion:** Cryptocurrencies have the potential to provide access to financial services for individuals who are underserved or excluded by traditional banking systems. People without access to bank accounts or credit cards can participate in the global economy through the use of cryptocurrencies and digital wallets.
- **Security and Privacy:** Cryptocurrency transactions are secured through cryptographic techniques and recorded on immutable, tamper-resistant ledgers (blockchains). This provides a high level of security against fraud and counterfeiting. Additionally, while transactions are recorded on public blockchains, the identities of the parties involved can remain pseudonymous, offering a degree of privacy.
- **Inflation Hedge**: Some cryptocurrencies, like Bitcoin, are designed with a fixed supply or a controlled inflation rate. This makes them potentially attractive as a hedge against inflation, as their value may not be eroded by the debasement of fiat currencies.