OBJECT ORIENTED PROGRAMMING

- **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts defined below :

- **Class** is a user-defined data type which defines its properties and its functions. Class is the only logical representation of the data. For example, Human being is a class. The body parts of a human being are its properties, and the actions performed by the body parts are known as functions. The class does not occupy any memory space till the time an object is instantiated.

  C++ Syntax (for class) :

```
class student{
        public:
          int id; // data member
          int mobile;
          string name;

          int add(int x, int y){ // member functions
                  return x + y;
          }
};
```

- **Object** is a run-time entity. It is an instance of the class. An object can represent a person, place or any other item. An object can operate on both data members and member functions.

  C++ Syntax (for object):

```
student s = new student();
```

  Note : When an object is created **using** a new keyword, then space is allocated for the variable in a heap, and the starting address is stored in the stack memory. When an object is created **without** a new keyword, then space is not allocated in the heap memory, and the object contains the null value in the stack.

- **Inheritance**

  Inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such a way, you can **reuse, extend or modify** the attributes and behaviors which are defined in other classes.
  In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

  C++ Syntax :

  class derived_class :: visibility-mode base_class;
  visibility-modes = {private, protected, public}

  Types of Inheritance :

  1. Single inheritance : When one class inherits another class, it is known as single level inheritance
  2. Multiple inheritance : Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.
  3. Hierarchical inheritance : Hierarchical inheritance is defined as the process of deriving more than one class from a base class.
  4. Multilevel inheritance : Multilevel inheritance is a process of deriving a class from another derived class.
  5. Hybrid inheritance : Hybrid inheritance is a combination of simple, multiple inheritance and hierarchical inheritance.

- **Encapsulation**

  Encapsulation is the process of combining data and functions into a single unit called class. In Encapsulation, the data is not accessed directly; it is accessed through the functions present inside the class. In simpler words, attributes of the class are kept private and public getter and setter methods are provided to manipulate these attributes. Thus, encapsulation makes the concept of data hiding possible. (**Data hiding**: a language feature to restrict access to members of an object, reducing the negative effect due to dependencies. e.g. "protected", "private" feature in C++).

- **Abstraction**

  We try to obtain an **abstract view**, model or structure of a real life problem, and reduce its unnecessary details. With definition of properties of problems, including the data which are affected and the operations which are identified, the model abstracted from problems can be a standard solution to this type of problems. It is an efficient way since there are nebulous real-life problems that have similar properties.

Data binding **:** Data binding is a process of binding the application UI and business logic. Any change made in the business logic will reflect directly to the application UI.

- **Polymorphism**

  Polymorphism is the ability to present the same interface for differing underlying forms (data types). With polymorphism, each of these classes will have different underlying data. A point shape needs only two coordinates (assuming it's in a two-dimensional space of course). A circle needs a center and radius. A square or rectangle needs two coordinates for the top left and bottom right corners and (possibly) a rotation. An irregular polygon needs a series of lines. Precisely, Poly means 'many' and morphism means 'forms'.

  Types of Polymorphism **IMP**

  1. Compile Time Polymorphism (Static)
  2. Runtime Polymorphism (Dynamic)

Let's understand them one by one :

- **Compile Time Polymorphism** : The polymorphism which is implemented at the compile time is known as compile-time polymorphism. Example - Method Overloading

Method Overloading : Method overloading is a technique which allows you to have more than one function with the same function name but with different functionality. Method overloading can be possible on the following basis:

1. The return type of the overloaded function.
2. The type of the parameters passed to the function.
3. The number of parameters passed to the function.

Example :

```
#include<bits/stdc++.h>
using namespace std;

class Add {
    public:
            int add(int a,int b){
                    return (a + b);
            }
            int add(int a,int b,int c){
                    return (a + b + c);
            }
};
int main(){
      Add obj;
      int res1,res2;
      res1 = obj.add(2,3);
      res2 = obj.add(2,3,4);
      cout << res1 << " " << res2 << endl;
      return 0;
}

/*
Output : 5 9
add() is an overloaded function with a different number of parameters. */
```

- **Runtime Polymorphism** : Runtime polymorphism is also known as **dynamic polymorphism**. Function overriding is an example of runtime polymorphism. Function overriding means when the child class contains the method which is already present in the parent class. Hence, **the child class overrides the method of the parent class**. In case of function overriding, parent and child classes both contain the same function with a different definition. The call to the function is determined at runtime is known as runtime polymorphism.

C++ Sample Code :

```
#include <bits/stdc++.h>
using namespace std;

class Base_class{
    public:
      virtual void show(){
            cout << "Apni Kaksha base" << endl;
      }
};

class Derived_class : public Base_class{
    public:
            void show(){
                  cout << "Apni Kaksha derived" << endl;
            }
};

int main(){
    Base_class* b;
    Derived_class d;
    b = &d;
    b->show(); // prints the content of show() declared in derived
    class return 0;
```

```
}

// Output : Apni Kaksha derived
```

- **Constructor** : Constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new objects generally. The constructor in C++ has the same name as class or structure.

  There can be **two types** of constructors in C++.

  1. Default constructor : A constructor which has no argument is known as default constructor. It is invoked at the time of creating an object.
  2. Parameterized constructor : Constructor which has parameters is called a parameterized constructor. It is used to provide different values to distinct objects.
  3. Copy Constructor : A Copy constructor is an **overloaded** constructor used to declare and initialize an object from another object. It is of two types – default copy constructor and user defined copy constructor.

C++ Sample Code :

```
#include <bits/stdc++.h>
using namespace std;

class go {
        public:
                int x;
                go(int a){ // parameterized constructor.
```

```
            x=a;
        }
        go(go &i){ // copy constructor
            x = i.x;
        }
};
int main(){
    go a1(20); // Calling the parameterized constructor.
    go a2(a1); // Calling the copy constructor.
    cout << a2.x << endl;
    return 0;
}

// Output : 20
```

- **Destructor** : A destructor works opposite to constructor; it destructs the objects of classes. It can be defined **only once** in a class. Like constructors, it is invoked automatically. A destructor is defined like a constructor. It must have the same name as class, prefixed with a **tilde sign (~)**.

  Example :

```
#include<bits/stdc++.h>
using namespace std;

class A{
    public:
        // constructor and destructor are called automatically, once the object is instantiated
        A(){
            cout << "Constructor in use" << endl;
        }
        ~A(){
            cout << "Destructor in use" << endl;
        }
};
int main(){
```

```
    A a;
    A b;
    return 0;
}
/*
Output: Constructor in use
                Constructor in use
                Destructor in use
                Destructor in use
*/
```

- **'this' Pointer** : **this** is a keyword that refers to the **current instance of the class.** There can be 3 main uses of 'this' keyword:
  1. It can be used **to pass the current object as a parameter to another method**
  2. It can be used **to refer to the current class instance variable.**
  3. It can be used **to declare indexers.**

C++ Syntax :

```
struct node{
    int data;
    node *next;

    node(int x){
        this->data = x;
        this->next = NULL;
    }
}
```

- **Friend Function** : Friend function acts as a friend of the class. **It can access the private and protected members of the class.** The friend function is not

a member of the class, but it must be listed in the class definition. The non-member function cannot access the private data of the class. Sometimes, it is necessary for the non-member function to access the data. **The friend function is a non-member function and has the ability to access the private data of the class.**

Note :

1. A friend function cannot access the private members directly, it has to use an object name and dot operator with each member name.
2. Friend function uses objects as arguments.

Example IMP :

```
#include <bits/stdc++.h>
using namespace std;

class A{
        int a = 2;
        int b = 4;
        public:
                // friend function
                friend int mul(A k){
                return (k.a * k.b);
                }
};

int main(){
        A obj;
        int res = mul(obj);
        cout << res << endl;
        return 0;
}

// Output : 8
```

- **Aggregation :** It is a process in which one class defines another class as

any entity reference. **It is another way to reuse the class.** It is a form of association that represents the HAS-A relationship.

- **Virtual Function IMP**: A virtual function is used to replace the implementation provided by the base class. The replacement is always called whenever the object in question is actually of the derived class, even if the object is accessed by a base pointer rather than a derived pointer.
    1. **A virtual function is a member function which is present in the base class and redefined by the derived class.**
    2. When we use the same function name in both base and derived class, **the function in base class is declared with a keyword virtual.**
    3. When the function is made virtual, then C++ determines at run-time which function is to be called based on the type of the object pointed by the base class pointer. **Thus, by making the base class pointer to point to different objects, we can execute different versions of the virtual functions.**

Key Points :

1. Virtual functions cannot be static.
2. A class may have a virtual destructor but it cannot have a virtual constructor.

C++ Example :

```
#include <bits/stdc++.h>
```

```
using namespace std;

class base {
	public:
		// virtual function (re-defined in the derived class)
		virtual void print(){
			cout << "print base class" << endl;
		}

		void show(){
			cout << "show base class" << endl;
		}
};

class derived : public base {
	public:
		void print(){
			cout << "print derived class" << endl;
		}

		void show(){
			cout << "show derived class" << endl;
		}
};

int main(){
	base* bptr;
	derived d;
	bptr = &d;

	// virtual function, binded at runtime
	bptr->print();

	// Non-virtual function, binded at compile time
	bptr->show();
}

/*
output :
```

```
print derived class // (impact of virtual function)
show base class
*/
```

- **Pure Virtual Function** :
    1. A pure virtual function is not used for performing any task. It only serves as a placeholder.
    2. A pure virtual function is a function declared in the base class that has no definition relative to the base class.
    3. A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as **abstract base classes**.
    4. The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.

C++ Syntax :

```
virtual void display() = 0;
```

C++ Example :

```
#include<bits/stdc++.h>
using namespace std;

class Base{
    public:
        virtual void show() = 0;
};
class Derived : public Base {
    public:
        void show() {
            cout << "You can see me !" << endl;
        }
};
int main(){
    Base *bptr;
```

```
    Derived d;
    bptr = &d;
    bptr->show();
    return 0;
}

// output : You can see me !
```

- **Abstract Classes** : In C++ class is made abstract by declaring at least one of its functions as a **pure virtual function**. A pure virtual function is specified by placing "= 0" in its declaration. **Its implementation must be provided by derived classes.**

  Example :

```
#include<bits/stdc++.h>
using namespace std;

// abstract class
class Shape{
    public:
        virtual void draw()=0;
};
class Rectangle : Shape{
    public:
            void draw(){
                cout << "Rectangle" << endl;
            }
};
class Square : Shape{
    public:
            void draw(){
                cout << "Square" << endl;
            }
};

int main(){
    Rectangle rec;
    Square sq;
```

```
    rec.draw();
    sq.draw();
    return 0;
}

/*
Output :
Rectangle
Square
*/
```

- **Namespaces in C++** :
    1. The namespace is a logical division of the code which is designed to stop the naming conflict.
    2. The namespace defines the scope where the identifiers such as variables, class, functions are declared.
    3. **The main purpose of using namespace in C++ is to remove the ambiguity.** Ambiguity occurs when a different task occurs with the same name.
    4. For example: if there are two functions with the same name such as add(). In order to prevent this ambiguity, the namespace is used. Functions are declared in different namespaces.
    5. C++ consists of a standard namespace, i.e., std which contains inbuilt classes and functions. So, by using the statement "using namespace std;" includes the namespace "std" in our program.

C++ Example :

```
#include <bits/stdc++.h>
using namespace std;

// user-defined namespace
namespace Add {
    int a = 5, b = 5;
      int add(){
        return (a + b);
    }
}
```

```
int main(){
        int res = Add :: add(); // accessing the function inside namespace
        cout << res;
}

// output : 10
```

- **Access Specifiers IMP :** The access specifiers are used to define how functions and variables can be accessed outside the class. There are three types of access specifiers:

1. **Private**: Functions and variables declared as private can be accessed only within the same class, and they cannot be accessed outside the class they are declared.
2. **Public**: Functions and variables declared under public can be accessed from anywhere.
3. **Protected**: Functions and variables declared as protected cannot be accessed outside the class except a child class. This specifier is generally used in inheritance.

Key Notes

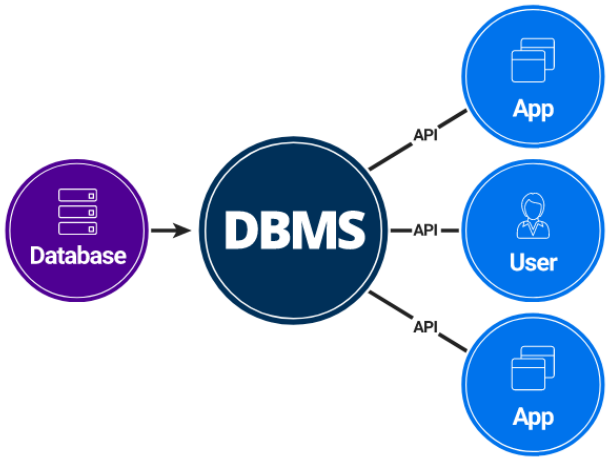
- **Delete** is used to release a unit of memory, **delete[]** is used to release an array.
- **Virtual inheritance** facilitates you to create only one copy of each object even if the object appears more than one in the hierarchy.
- **Function overloading:** Function overloading is defined as we can have more than one version of the same function. The versions of a function will have different signatures meaning that they have a different set of parameters.

Operator overloading: Operator overloading is defined as the standard operator can be redefined so that it has a different meaning when applied to the instances of a class.

- **Overloading** is static Binding, whereas Overriding is dynamic Binding. Overloading is nothing but the same method with different arguments, and it may or may not return the same value in the same class itself. **Overriding** is the same method name with the same arguments and return types associated with the class and its child class.

LEC-1: Introduction to DBMS

1. **What is Data?**
    a. Data is a collection of raw, unorganized facts and details like text, observations, figures, symbols, and descriptions of things etc.
    In other words, **data does not carry any specific purpose and has no significance by itself.**
    Moreover, data is measured in terms of bits and bytes – which are basic units of information in the context of computer storage and processing.
    b. Data can be recorded and doesn't have any meaning unless processed.
2. **Types of Data**
    a. Quantitative
        i. Numerical form
        ii. Weight, volume, cost of an item.
    b. Qualitative
        i. Descriptive, but not numerical.
        ii. Name, gender, hair color of a person.
3. What is **Information**?
    a. Info. Is **processed, organized, and structured data.**
    b. It provides **context of the data and enables decision making.**
    c. Processed data that make **sense** to us.
    d. Information is extracted from the data, by **analyzing and interpreting** pieces of data.
    e. E.g., you have data of all the people living in your locality, its Data, when you analyze and interpret the data and come to some conclusion that:
        i. There are 100 senior citizens.
        ii. The sex ratio is 1.1.
        iii. Newborn babies are 100.
        These are information.
4. **Data vs Information**
    a. Data is a collection of facts, while information puts those facts into context.
    b. While data is raw and unorganized, information is organized.
    c. Data points are individual and sometimes unrelated. Information maps out that data to provide a big-picture view of how it all fits together.
    d. Data, on its own, is meaningless. When it's analyzed and interpreted, it becomes meaningful information.
    e. Data does not depend on information; however, information depends on data.
    f. Data typically comes in the form of graphs, numbers, figures, or statistics. Information is typically presented through words, language, thoughts, and ideas.
    g. Data isn't sufficient for **decision-making**, but you can make decisions based on information.
5. What is **Database**?
    a. Database is an electronic place/system where data is stored in a way that it can be **easily accessed, managed, and updated.**
    b. To make real use Data, we need **Database management systems. (DBMS)**
6. What is **DBMS**?
    a. A database-management system (DBMS) is a collection of **interrelated data** and **a set of programs to access those data**. The collection of data, usually referred to as the **database**, contains information relevant to an enterprise. The primary goal of a DBMS is to provide a way to **store and retrieve database information** that is both convenient and efficient.
    b. A DBMS is the database itself, along with all the software and functionality. It is used to perform different operations, like **addition**, **access**, **updating**, and **deletion** of the data.

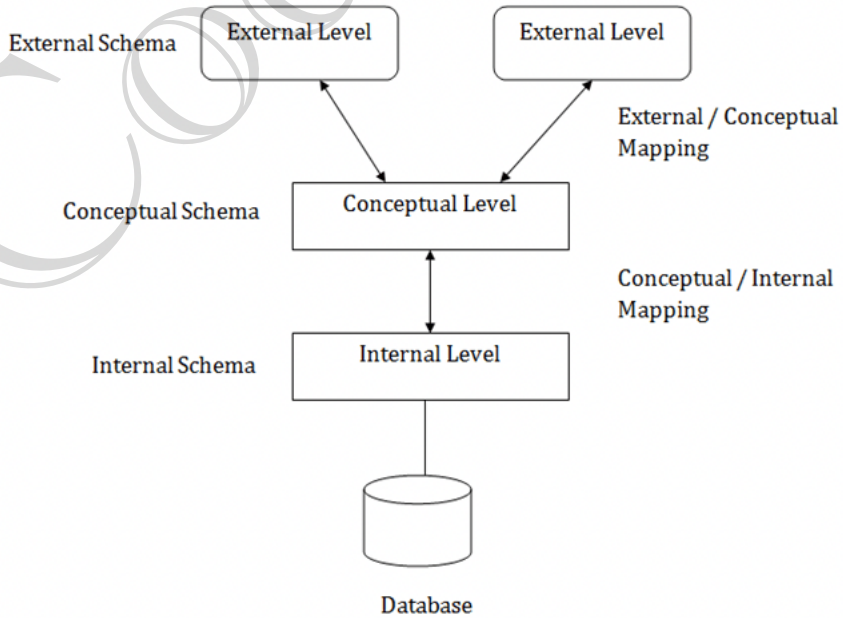


7.

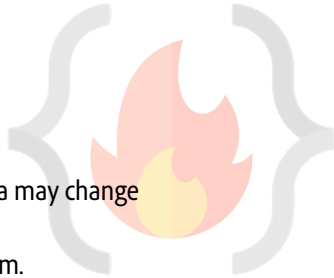
8. **DBMS vs File Systems**
    a. **File-processing systems** has major **disadvantages**.
        i. Data Redundancy and inconsistency
        ii. Difficulty in accessing data
        iii. Data isolation
        iv. Integrity problems
        v. Atomicity problems
        vi. Concurrent-access anomalies
        vii. Security problems
    b. Above 7 are also the **Advantages of DBMS** (answer to "**Why to use DBMS?**")

LEC-2: DBMS Architecture

1. **View of Data (Three Schema Architecture)**
    a. The major purpose of DBMS is to provide users with an **abstract view** of the data. That is, the system hides certain details of how the data is stored and maintained.
    b. To simplify user interaction with the system, abstraction is applied through several levels of abstraction.
    c. The **main objective** of three level architecture is to enable multiple users to access the same data with a personalized view while storing the underlying data only once
    d. Physical level / Internal level
        i. The lowest level of abstraction describes how the data are stored.
        ii. Low-level data structures used.
        iii. It has **Physical schema** which describes physical storage structure of DB.
        iv. Talks about: Storage allocation (N-ary tree etc), Data compression & encryption etc.
        v. **Goal**: We must define algorithms that allow efficient access to data.
    e. Logical level / Conceptual level:
        i. The **conceptual schema** describes the design of a database at the conceptual level, describes ***what*** data are stored in DB, and what ***relationships*** exist among those data.
        ii. User at logical level does not need to be aware about physical-level structures.
        iii. **DBA**, who must decide what information to keep in the DB use the logical level of abstraction.
        iv. **Goal**: ease to use.
    f. View level / External level:
        i. Highest level of abstraction aims to simplify users' interaction with the system by providing different view to different **end**-user.
        ii. Each **view schema** describes the database part that a particular user group is interested and hides the remaining database from that user group.
        iii. At the external level, a database contains several schemas that sometimes called as **subschema**. The subschema is used to describe the different view of the database.
        iv. At views also provide a **security** mechanism to prevent users from accessing certain parts of DB.

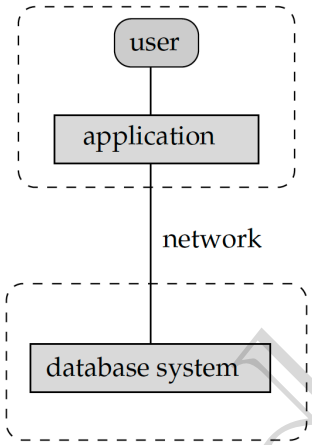


    g.
2. **Instances and Schemas**
    a. The collection of information stored in the DB at a particular moment is called an **instance** of DB.

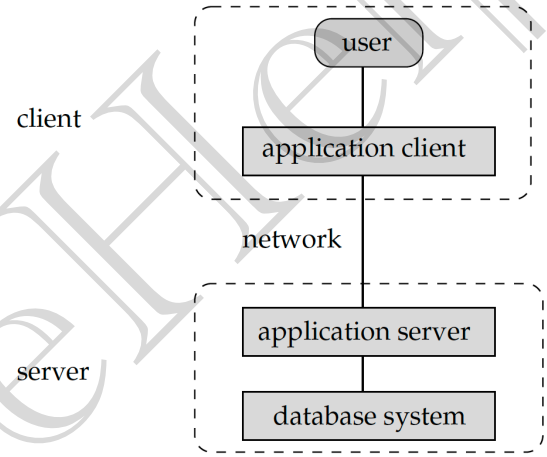


   b. The overall design of the DB is called the DB **schema**.
   c. Schema is **structural** description of data. Schema **doesn't change frequently**. Data may change frequently.
   d. **DB schema** corresponds to the variable declarations (along with type) in a program.
   e. We have 3 types of **Schemas**: **Physical, Logical**, several **view schemas** called subschemas.
   f. Logical schema is most **important** in terms of its effect on application programs, as programmers construct apps by using logical schema.
   g. **Physical data independence**, physical schema change should not affect logical schema/application programs.
3. **Data Models:**
   a. Provides a way to describe the **design** of a DB at **logical level.**
   b. Underlying the structure of the DB is the Data Model; a collection of conceptual tools for describing **data, data relationships, data semantics & consistency constraints**.
   c. E.g., **ER** model, **Relational** Model, **object-oriented** model, **object-relational** data model etc.
4. **Database Languages**:
   a. **Data definition language (DDL)** to specify the database schema.
   b. **Data manipulation language (DML)** to express database queries and updates.
   c. **Practically**, both language features are present in a single DB language, e.g., SQL language.
   d. DDL
      i. We specify consistency constraints, which must be checked, every time DB is updated.
   e. DML
      i. Data manipulation involves
         1. **Retrieval** of information stored in DB.
         2. **Insertion** of new information into DB.
         3. **Deletion** of information from the DB.
         4. **Updating** existing information stored in DB.
      ii. **Query language**, a part of DML to specify statement requesting the retrieval of information.
5. **How is Database accessed from Application programs?**
   a. Apps (written in host languages, C/C++, Java) interacts with DB.
   b. E.g., Banking system's module generating payrolls access DB by executing DML statements from the host language.
   c. API is provided to send DML/DDL statements to DB and retrieve the results.
      i. Open Database Connectivity (**ODBC**), Microsoft "C".
      ii. Java Database Connectivity (**JDBC**), Java.
6. **Database Administrator (DBA)**
   a. A person who has **central control** of both the data and the programs that access those data.
   b. **Functions** of DBA
      i. Schema Definition
      ii. Storage structure and access methods.
      iii. Schema and physical organization modifications.
      iv. Authorization control.
      v. Routine maintenance
         1. Periodic backups.
         2. Security patches.
         3. Any upgrades.
7. **DBMS Application Architectures**: Client machines, on which remote DB users work, and server machines on which DB system runs.
   a. **T1** Architecture
      i. The client, server & DB all present on the same machine.

b. **T2** Architecture
    i. App is partitioned into 2-components.
    ii. Client machine, which invokes DB system functionality at server end through query language statements.
    iii. API standards like **ODBC** & **JDBC** are used to interact between client and server.
c. **T3** Architecture
    i. App is partitioned into 3 logical components.
    ii. Client machine is just a frontend and doesn't contain any direct DB calls.
    iii. Client machine communicates with App server, and App server communicated with DB system to access data.
    iv. **Business** logic, what action to take at that condition is in App server itself.
    v. T3 architecture are best for **WWW** Applications.
    vi. **Advantages**:
        1. **Scalability** due to distributed application servers.
        2. **Data integrity**, App server acts as a middle layer between client and DB, which minimize the chances of data corruption.
        3. **Security**, client can't directly access DB, hence it is more secure.



a. two-tier architecture



b. three-tier architecture

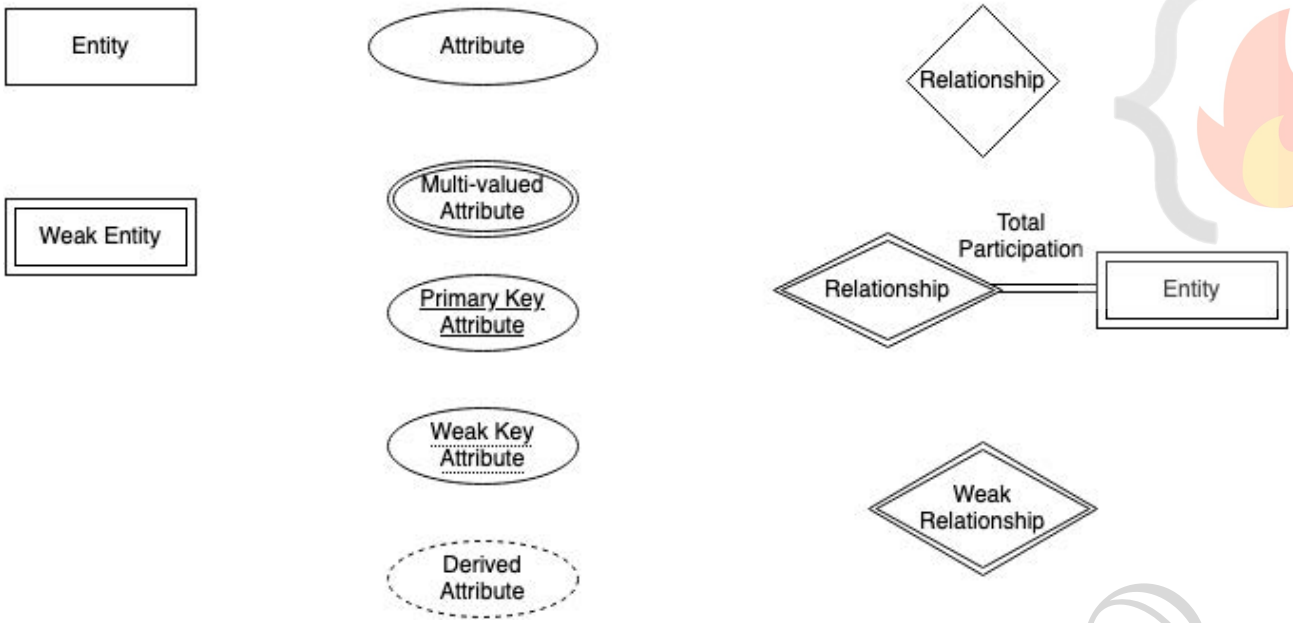
LEC-3: Entity-Relationship Model

1. **Data Model**: Collection of conceptual tools for **describing data, data relationships, data semantics, and consistency constraints.**
2. **ER Model**
    1. It is a high level data model based on a perception of a **real** world that consists of a collection of basic objects, called **entities** and of **relationships** among these objects.
    2. Graphical representation of ER Model is **ER diagram,** which acts as a **blueprint** of DB.
3. **Entity**: An Entity is a "**thing**" or "**object**" in the real world that is **distinguishable** from all other objects.
    1. It has **physical existence**.
    2. Each student in a college is an entity.
    3. Entity can be **uniquely** identified. (By a primary attribute, aka Primary Key)
    4. **Strong Entity**: Can be uniquely identified.
    5. **Weak Entity**: Can't be uniquely identified., depends on some other strong entity.
        1. It doesn't have sufficient attributes, to select a uniquely identifiable attribute.
        2. Loan -> Strong Entity, Payment -> Weak, as instalments are sequential number counter can be generated separate for each loan.
        3. **Weak entity depends on strong entity for existence.**
4. **Entity set**
    1. It is a set of entities of the **same** type that share the **same** properties, or attributes.
    2. E.g., Student is an entity set.
    3. E.g., Customer of a bank
5. **Attributes**
    1. An entity is represented by a set of attributes.
    2. Each entity has a value for each of its attributes.
    3. For each attribute, there is a set of **permitted values**, called the **domain**, or **value** set, of that attribute.
    4. E.g., Student Entity has following attributes
        A. Student_ID
        B. Name
        C. Standard
        D. Course
        E. Batch
        F. Contact number
        G. Address
    5. **Types of Attributes**
        1. **Simple**
            1. Attributes which can't be divided further.
            2. E.g., Customer's account number in a bank, Student's Roll number etc.
        2. **Composite**
            1. Can be divided into subparts (that is, other attributes).
            2. E.g., Name of a person, can be divided into first-name, middle-name, last-name.
            3. If user wants to refer to an entire attribute or to only a component of the attribute.
            4. Address can also be divided, street, city, state, PIN code.
        3. **Single-valued**
            1. Only one value attribute.
            2. e.g., Student ID, loan-number for a loan.
        4. **Multi-valued**
            1. Attribute having more than one value.
            2. e.g., phone-number, nominee-name on some insurance, dependent-name etc.
            3. Limit constraint may be applied, upper or lower limits.
        5. **Derived**
            1. Value of this type of attribute can be derived from the value of other related attributes.

    2. e.g., Age, loan-age, membership-period etc.
6. **NULL Value**
    1. An attribute takes a null value when an entity does not have a value for it.
    2. It may indicate "not applicable", value doesn't exist. e.g., person having no middle-name
    3. It may indicate "unknown".
        1. Unknown can indicate missing entry, e.g., name value of a customer is NULL, means it is missing as name must have some value.
        2. Not known, salary attribute value of an employee is null, means it is not known yet.

6. **Relationships**
    1. **Association** among two or more entities.
    2. e.g., Person has vehicle, Parent has Child, Customer borrow loan etc.
    3. **Strong Relationship**, between two independent entities.
    4. **Weak Relationship**, between weak entity and its owner/strong entity.
        1. e.g., Loan <instalment-payments> Payment.
    5. **Degree of Relationship**
        1. Number of entities participating in a relationship.
        2. **Unary**, Only one entity participates. e.g., Employee manages employee.
        3. **Binary**, two entities participates. e.g., Student takes Course.
        4. **Ternary** relationship, three entities participates. E.g, Employee works-on branch, employee works-on job.
        5. Binary are **common**.
7. **Relationships Constraints**
    1. **Mapping Cardinality / Cardinality Ratio**
        1. Number of entities to which another entity can be associated via a relationship.
        2. **One to one,** Entity in A associates with at most one entity in B, where A & B are entity sets. And an entity of B is associated with at most one entity of A.
            1. E.g., Citizen has Aadhar Card.
        3. **One to many,** Entity in A associated with N entity in B. While entity in B is associated with at most one entity in A.
            1. e.g., Citizen has Vehicle.
        4. **Many to one,** Entity in A associated with at most one entity in B. While entity in B can be associated with N entity in A.
            1. e.g., Course taken by Professor.
        5. **Many to many,** Entity in A associated with N entity in B. While entity in B also associated with N entity in A.
            1. Customer buys product.
            2. Student attend course.
    2. **Participation Constraints**
        1. Aka, **Minimum cardinality constraint.**
        2. **Types**, Partial & Total Participation.
        3. **Partial Participation,** not all entities are involved in the relationship instance.
        4. **Total Participation**, each entity must be involved in at least one relationship instance.
        5. e.g., Customer borrow loan, loan has total participation as it can't exist without customer entity. And customer has partial participation.
        6. **Weak entity has total participation constraint, but strong may not have total.**
8. **ER Notations**

Symbols used in ER Diagram



LEC-4: Extended ER Features

1. **Basic ER Features** studied in the **LEC-3,** can be used to model most DB features but when complexity increases, it is better to use some Extended ER features to model the DB Schema.
2. **Specialisation**
    1. In ER model, we may require to subgroup an entity set into other entity sets that are distinct in some way with other entity sets.
    2. **Specialisation** is **splitting** up the entity set into further **sub entity sets** on the basis of their **functionalities**, **specialities** and **features**.
    3. It is a **Top-Down** approach.
    4. e.g., **Person** entity set can be divided into **customer**, **student**, **employee**. Person is **superclass** and other specialised entity sets are **subclasses**.
        1. We have **"is-a"** relationship between superclass and subclass.
        2. Depicted by **triangle** component.
    5. **Why** Specialisation?
        1. Certain attributes may only be applicable to a few entities of the parent entity set.
        2. DB designer can show the distinctive features of the sub entities.
        3. To group such entities we apply Specialisation, to overall **refine** the DB blueprint.
3. **Generalisation**
    1. It is just a **reverse** of Specialisation.
    2. DB Designer, may encounter certain properties of two entities are overlapping. Designer may consider to make a new generalised entity set. That generalised entity set will be a super class.
    3. **"is-a"** relationship is present between subclass and super class.
    4. e.g., **Car**, **Jeep** and **Bus** all have some common attributes, to avoid data repetition for the common attributes. DB designer may consider to Generalise to a new entity set "**Vehicle**".
    5. It is a **Bottom-up** approach.
    6. **Why** Generalisation?
        1. Makes DB more **refined** and **simpler**.
        2. Common attributes are not **repeated**.
4. **Attribute Inheritance**
    1. **Both** Specialisation and Generalisation, has attribute inheritance.
    2. The attributes of higher level entity sets are inherited by lower level entity sets.
    3. E.g., **Customer & Employee** inherit the attributes of **Person**.
5. **Participation Inheritance**
    1. If a parent entity set participates in a relationship then its child entity sets will also participate in that relationship.
6. **Aggregation**
    1. **How to show relationships among relationships?** - Aggregation is the technique.
    2. **Abstraction** is applied to treat relationships as higher-level entities. We can call it Abstract entity.
    3. **Avoid redundancy** by aggregating relationship as an entity set itself.

LEC-7: Relational Model

1. Relational Model (RM) organises the data in the form of **relations** (**tables**).
2. A relational DB consists of **collection of tables**, each of which is assigned a **unique name.**
3. A **row** in a table represents a relationship among a set of values, and table is collection of such relationships.
4. **Tuple**: A single row of the table representing a single data point / a unique record.
5. **Columns**: represents the attributes of the relation. Each attribute, there is a permitted value, called **domain of the attribute.**
6. **Relation Schema**: defines the design and structure of the relation, contains the name of the relation and all the columns/attributes.
7. Common RM based DBMS systems, aka RDBMS: Oracle, IBM, **MySQL**, MS Access.
8. **Degree of table:** number of attributes/columns in a given table/relation.
9. **Cardinality**: Total no. of tuples in a given relation.
10. **Relational Key**: Set of attributes which can uniquely identify an each tuple.
11. Important **properties** of a **Table** in Relational Model
    1. The name of relation is distinct among all other relation.
    2. The values have to be atomic. Can't be broken down further.
    3. The name of each attribute/column must be unique.
    4. Each tuple must be unique in a table.
    5. The sequence of row and column has no significance.
    6. Tables must follow integrity constraints - it helps to maintain data consistency across the tables.
12. **Relational Model Keys**
    1. Super Key (**SK**): Any P&C of attributes present in a table which can uniquely identify each tuple.
    2. Candidate Key (**CK**): minimum subset of super keys, which can uniquely identify each tuple. It contains no redundant attribute.
        1. **CK value shouldn't be NULL.**
    3. Primary Key (**PK**):
        1. Selected out of CK set, has the least no. of attributes.
    4. Alternate Key (**AK**)
        1. All CK except PK.
    5. Foreign Key (**FK**)
        1. It creates relation between two tables.
        2. A relation, say r1, may include among its attributes the PK of an other relation, say r2. This attribute is called FK from r1 referencing r2.
        3. The relation r1 is aka **Referencing (Child) relation** of the FK dependency, and r2 is called **Referenced (Parent) relation** of the FK.
        4. FK helps to cross reference between two different relations.
    6. **Composite** Key: PK formed using at least 2 attributes.
    7. **Compound** Key: PK which is formed using 2 FK.
    8. **Surrogate** Key:
        1. Synthetic PK.
        2. Generated automatically by DB, usually an integer value.
        3. May be used as PK.
13. **Integrity Constraints**
    1. CRUD Operations must be done with some integrity policy so that DB is always consistent.
    2. Introduced so that we do not accidentally corrupt the DB.
    3. **Domain Constraints**
        1. Restricts the value in the attribute of relation, specifies the Domain.
        2. Restrict the Data types of every attribute.
        3. E.g., We want to specify that the enrolment should happen for candidate birth year < 2002.
    4. **Entity Constraints**
        1. Every relation should have PK. PK != NULL.

5. **Referential Constraints**
    1. Specified between two relations & helps maintain consistency among tuples of two relations.
    2. It requires that the value appearing in specified attributes of any tuple in referencing relation also appear in the specified attributes of at least one tuple in the referenced relation.
    3. If FK in referencing table refers to PK of referenced table then every value of the FK in referencing table must be NULL or available in referenced table.
    4. FK must have the matching PK for its each value in the parent table or it must be NULL.
6. **Key Constraints:** The six types of key constraints present in the Database management system are:-
    1. NOT NULL: This constraint will restrict the user from not having a NULL value. It ensures that every element in the database has a value.
    2. UNIQUE: It helps us to ensure that all the values consisting in a column are different from each other.
    3. DEFAULT: it is used to set the default value to the column. The default value is added to the columns if no value is specified for them.
    4. CHECK: It is one of the integrity constraints in DBMS. It keeps the check that integrity of data is maintained before and after the completion of the CRUD.
    5. PRIMARY KEY: This is an attribute or set of attributes that can uniquely identify each entity in the entity set. The primary key must contain unique as well as not null values.
    6. FOREIGN KEY: Whenever there is some relationship between two entities, there must be some common attribute between them. This common attribute must be the primary key of an entity set and will become the foreign key of another entity set. This key will prevent every action which can result in loss of connection between tables.

LEC-8: Transform - ER Model to Relational Model

1. Both **ER**-Model and **Relational** Model are abstract logical representation of real world enterprises. Because the two models implies the similar design principles, **we can convert ER design into Relational design**.
2. Converting a DB representation from an ER diagram to a table format is the way we arrive at Relational DB-design from an ER diagram.
3. ER diagram **notations to relations**:
    1. **Strong Entity**
        1. Becomes an **individual table** with entity name, attributes becomes columns of the relation.
        2. Entity's Primary Key (PK) is used as Relation's PK.
        3. **FK** are added to establish relationships with other relations.
    2. **Weak Entity**
        1. A table is formed with all the attributes of the entity.
        2. PK of its corresponding Strong Entity will be added as **FK**.
        3. **PK** of the relation will be a composite PK, {FK + Partial discriminator Key}.
    3. **Single Values Attributes**
        1. Represented as **columns** directly in the tables/relations.
    4. **Composite Attributes**
        1. Handled by **creating a separate attribute** itself in the original relation for each composite attribute.
        2. e.g., **Address**: {street-name, house-no}, is a composite attribute in customer relation, we add address-street-name & address-house-name as new columns in the attribute and ignore "address" as an attribute.
    5. **Multivalued Attributes**
        1. **New tables** (named as original attribute name) are created for each multivalued attribute.
        2. PK of the entity is used as column **FK** in the new table.
        3. Multivalued attribute's similar name is added as a column to define multiple values.
        4. **PK** of the new table would be {FK + multivalued name}.
        5. e.g., For Strong entity **Employee**, **dependent-name** is a multivalued attribute.
            1. New table named dependent-name will be formed with columns emp-id, and dname.
            2. PK: {emp-id, name}
            3. FK: {emp-id}
    6. **Derived Attributes**: Not considered in the tables.
    7. **Generalisation**
        1. **Method-1:** Create a table for the higher level entity set. For each lower-level entity set, create a table that includes a column for each of the attributes of that entity set plus a column for each attribute of the primary key of the higher-level entity set.
           For e.g., Banking System generalisation of Account - saving & current.
            1. Table 1: account (<u>account-number</u>, balance)
            2. Table 2: savings-account (<u>account-number</u>, interest-rate, daily-withdrawal-limit)
            3. Table 3: current-account (<u>account-number</u>, overdraft-amount, per-transaction-charges)
        2. **Method-2:** An alternative representation is possible, if the generalisation is disjoint and complete—that is, if no entity is a member of two lower-level entity sets directly below a higher-level entity set, and if every entity in the higher level entity set is also a member of one of the lower-level entity sets. Here, do not create a table for the higher-level entity set. Instead, for each lower-level entity set, create a table that includes a column for each of the attributes of that entity set plus a column for each attribute of the higher-level entity sets.
           Tables would be:
            1. Table 1: savings-account (<u>account-number</u>, balance, interest-rate, daily-withdrawal-limit)
            2. Table 2: current-account (<u>account-number</u>, balance, overdraft-amount, per-transaction-charges)
        3. **<u>Drawbacks of Method-2:</u>** If the second method were used for an overlapping generalisation, some values such as balance would be stored twice unnecessarily. Similarly, if the generalisation were not complete—that is, if some accounts were neither savings nor current accounts—then such accounts could not be represented with the second method.
    8. **Aggregation**

1. Table of the relationship set is made.
2. Attributes includes primary keys of entity set and aggregation set's entities.
3. Also, add descriptive attribute if any on the relationship.

LEC-9: SQL in 1-Video

1. **SQL**: Structured Query Language, used to access and manipulate data.
2. SQL used **CRUD** operations to communicate with DB.
    1. **CREATE** - execute INSERT statements to insert new tuple into the relation.
    2. **READ** - Read data already in the relations.
    3. **UPDATE** - Modify already inserted data in the relation.
    4. **DELETE** - Delete specific data point/tuple/row or multiple rows.
3. **SQL is not DB, is a query language.**
4. What is **RDBMS**? (Relational Database Management System)
    1. Software that enable us to implement designed relational model.
    2. e.g., MySQL, MS SQL, Oracle, IBM etc.
    3. Table/Relation is the simplest form of data storage object in R-DB.
    4. **MySQL** is open-source RDBMS, and it uses SQL for all CRUD operations
5. **MySQL** used client-server model, where client is CLI or frontend that used services provided by MySQL server.
6. **Difference between SQL and MySQL**
    1. SQL is Structured Query language used to perform CRUD operations in R-DB, while MySQL is a RDBMS used to store, manage and administrate DB (provided by itself) using SQL.

SQL DATA TYPES (Ref: https://www.w3schools.com/sql/sql_datatypes.asp)

1. In SQL DB, data is stored in the form of tables.
2. Data can be of different types, like INT, CHAR etc.

DATATYPE	Description
CHAR	string(0-255), string with size = (0, 255], e.g., CHAR(251)
VARCHAR	string(0-255)
TINYTEXT	String(0-255)
TEXT	string(0-65535)
BLOB	string(0-65535)
MEDIUMTEXT	string(0-16777215)
MEDIUMBLOB	string(0-16777215)
LONGTEXT	string(0-4294967295)
LONGBLOB	string(0-4294967295)
TINYINT	integer(-128 to 127)
SMALLINT	integer(-32768 to 32767)
MEDIUMINT	integer(-8388608 to 8388607)
INT	integer(-2147483648 to 2147483647)
BIGINT	integer (-9223372036854775808 to 9223372036854775807)
FLOAT	Decimal with precision to 23 digits
DOUBLE	Decimal with 24 to 53 digits

DATATYPE	Description
DECIMAL	Double stored as string
DATE	YYYY-MM-DD
DATETIME	YYYY-MM-DD HH:MM:SS
TIMESTAMP	YYYYMMDDHHMMSS
TIME	HH:MM:SS
ENUM	One of the preset values
SET	One or many of the preset values
BOOLEAN	0/1
BIT	e.g., BIT(n), n upto 64, store values in bits.

3. Size: TINY < SMALL < MEDIUM < INT < BIGINT.
4. **Variable length Data types** e.g., VARCHAR, are better to use as they occupy space equal to the actual data size.
5. Values can also be unsigned e.g., INT UNSIGNED.
6. **Types of SQL commands:**
    1. **DDL** (data definition language): defining relation schema.
        1. **CREATE**: create table, DB, view.
        2. **ALTER TABLE**: modification in table structure. e.g, change column datatype or add/remove columns.
        3. **DROP**: delete table, DB, view.
        4. **TRUNCATE**: remove all the tuples from the table.
        5. **RENAME**: rename DB name, table name, column name etc.
    2. **DRL/DQL** (data retrieval language / data query language): retrieve data from the tables.
        1. **SELECT**
    3. **DML** (data modification language): use to perform modifications in the DB
        1. **INSERT**: insert data into a relation
        2. **UPDATE**: update relation data.
        3. **DELETE**: delete row(s) from the relation.
    4. **DCL** (Data Control language): grant or revoke authorities from user.
        1. **GRANT**: access privileges to the DB
        2. **REVOKE**: revoke user access privileges.
    5. **TCL** (Transaction control language): to manage transactions done in the DB
        1. **START TRANSACTION**: begin a transaction
        2. **COMMIT**: apply all the changes and end transaction
        3. **ROLLBACK**: discard changes and end transaction
        4. **SAVEPOINT**: checkout within the group of transactions in which to rollback.

MANAGING DB (DDL)

1. **Creation of DB**
    1. **CREATE DATABASE IF NOT EXISTS db-name;**
    2. **USE** db-name; //need to execute to choose on which DB CREATE TABLE etc commands will be executed. //make switching between DBs possible.
    3. **DROP** DATABASE IF EXISTS db-name; //dropping database.
    4. **SHOW** DATABASES; //list all the DBs in the server.
    5. **SHOW** TABLES; //list tables in the selected DB.

DATA RETRIEVAL LANGUAGE (DRL)

1. Syntax: SELECT <set of column names> FROM <table_name>;
2. Order of execution from RIGHT to LEFT.
3. Q. Can we use SELECT keyword without using FROM clause?
    1. Yes, using DUAL Tables.
    2. Dual tables are dummy tables created by MySQL, help users to do certain obvious actions without referring to user defined tables.
    3. e.g., SELECT 55 + 11;
       SELECT now();
       SELECT ucase(); etc.
4. **WHERE**
    1. Reduce rows based on given conditions.
    2. E.g., SELECT * FROM customer WHERE age > 18;
5. **BETWEEN**
    1. SELECT * FROM customer WHERE age between 0 **AND** 100;
    2. In the above e.g., 0 and 100 are inclusive.
6. **IN**
    1. Reduces **OR** conditions;
    2. e.g., SELECT * FROM officers WHERE officer_name IN ('Lakshay', 'Maharana Pratap', 'Deepika');
7. **AND/OR/NOT**
    1. AND: WHERE cond1 AND cond2
    2. OR: WHERE cond1 OR cond2
    3. NOT: WHERE col_name NOT IN (1,2,3,4);
8. **IS NULL**
    1. e.g., SELECT * FROM customer WHERE prime_status is NULL;
9. **Pattern Searching / Wildcard ('%', '_')**
    1. '%', any number of character from 0 to n. Similar to '*' asterisk in regex.
    2. '_', only one character.
    3. SELECT * FROM customer WHERE name **LIKE** '%p_';
10. **ORDER BY**
    1. Sorting the data retrieved using **WHERE** clause.
    2. ORDER BY <column-name> DESC;
    3. DESC = Descending and ASC = Ascending
    4. e.g., SELECT * FROM customer ORDER BY name DESC;
11. **GROUP BY**
    1. GROUP BY Clause is used to collect data from multiple records and group the result by one or more column. It is generally used in a SELECT statement.
    2. Groups into category based on column given.
    3. SELECT c1, c2, c3 FROM sample_table WHERE cond GROUP BY c1, c2, c3.
    4. All the column names mentioned after SELECT statement shall be repeated in GROUP BY, in order to successfully execute the query.
    5. Used with aggregation functions to perform various actions.
        1. COUNT()
        2. SUM()
        3. AVG()
        4. MIN()
        5. MAX()
12. **DISTINCT**
    1. Find distinct values in the table.
    2. SELECT DISTINCT(col_name) FROM table_name;
    3. GROUP BY can also be used for the same
        1. "Select col_name from table GROUP BY col_name;" same output as above DISTINCT query.

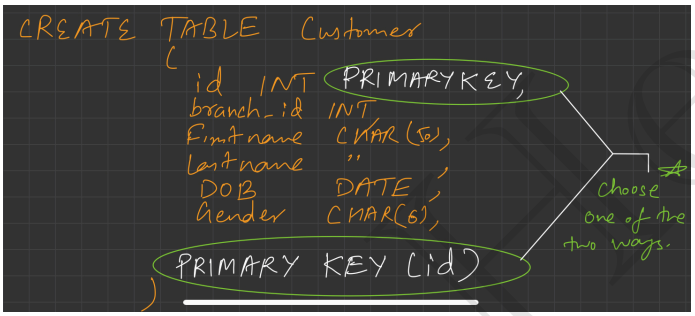
2. SQL is smart enough to realise that if you are using GROUP BY and not using any aggregation function, then you mean "DISTINCT".

13. GROUP BY HAVING

1. Out of the categories made by GROUP BY, we would like to know only particular thing (cond).
2. Similar to WHERE.
3. Select COUNT(cust_id), country from customer GROUP BY country HAVING COUNT(cust_id) > 50;
4. WHERE vs HAVING
    1. Both have same function of filtering the row base on certain conditions.
    2. WHERE clause is used to filter the rows from the table based on specified condition
    3. HAVING clause is used to filter the rows from the groups based on the specified condition.
    4. HAVING is used after GROUP BY while WHERE is used before GROUP BY clause.
    5. If you are using HAVING, GROUP BY is necessary.
    6. WHERE can be used with SELECT, UPDATE & DELETE keywords while GROUP BY used with SELECT.

CONSTRAINTS (DDL)

1. **Primary Key**



    1. PK is not null, unique and only one per table.
2. **Foreign Key**
    1. FK refers to PK of other table.
    2. Each relation can having any number of FK.
    3. CREATE TABLE ORDER (
    id INT PRIMARY KEY,
    delivery_date DATE,
    order_placed_date DATE,
    cust_id INT,
    FOREIGN KEY (cust_id) REFERENCES customer(id)
    );
3. **UNIQUE**
    1. Unique, can be null, table can have multiple unique attributes.
    2. CREATE TABLE customer (
    ...
    email VARCHAR(1024) UNIQUE,
    ...
    );
4. **CHECK**
    1. CREATE TABLE customer (
    ...
    CONSTRAINT age_check CHECK (age > 12),
    ...
    );
    2. "age_check", can also avoid this, MySQL generates name of constraint automatically.

5. **DEFAULT**
    1. Set default value of the column.
    2. CREATE TABLE account (
       ...
       saving-rate DOUBLE NOT NULL DEFAULT 4.25,
       ...
       );
6. An attribute can be **PK and FK both** in a table.
7. **ALTER OPERATIONS**
    1. Changes schema
    2. **ADD**
        1. **Add new column.**
        2. ALTER TABLE table_name ADD new_col_name datatype ADD new_col_name_2 datatype;
        3. e.g., ALTER TABLE customer ADD age INT NOT NULL;
    3. **MODIFY**
        1. **Change datatype of an attribute.**
        2. ALTER TABLE table-name MODIFY col-name col-datatype;
        3. E.g., VARCHAR TO CHAR
           ALTER TABLE customer MODIFY name CHAR(1024);
    4. **CHANGE COLUMN**
        1. **Rename column name.**
        2. ALTER TABLE table-name CHANGE COLUMN old-col-name new-col-name new-col-datatype;
        3. e.g., ALTER TABLE customer CHANGE COLUMN name customer-name VARCHAR(1024);
    5. **DROP COLUMN**
        1. **Drop a column completely.**
        2. ALTER TABLE table-name DROP COLUMN col-name;
        3. e.g., ALTER TABLE customer DROP COLUMN middle-name;
    6. **RENAME**
        1. **Rename table name itself.**
        2. ALTER TABLE table-name RENAME TO new-table-name;
        3. e.g., ALTER TABLE customer RENAME TO customer-details;

DATA MANIPULATION LANGUAGE (DML)

1. **INSERT**
    1. INSERT INTO table-name(col1, col2, col3) VALUES (v1, v2, v3), (val1, val2, val3);
2. **UPDATE**
    1. UPDATE table-name SET col1 = 1, col2 = 'abc' WHERE id = 1;
    2. Update multiple rows e.g.,
        1. UPDATE student SET standard = standard + 1;
    3. **ON UPDATE CASCADE**
        1. Can be added to the table while creating constraints. Suppose there is a situation where we have two tables such that primary key of one table is the foreign key for another table. if we update the primary key of the first table then using the ON UPDATE CASCADE foreign key of the second table automatically get updated.
3. **DELETE**
    1. DELETE FROM table-name WHERE id = 1;
    2. DELETE FROM table-name; //all rows will be deleted.
    3. **DELETE CASCADE** - (**to overcome DELETE constraint of Referential constraints**)
        1. What would happen to child entry if parent table's entry is deleted?
        2. CREATE TABLE ORDER (
           order_id int PRIMARY KEY,
           delivery_date DATE,
           cust_id INT,

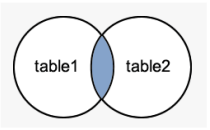
FOREIGN KEY(cust_id) REFERENCES customer(id) ON DELETE CASCADE
);

3. **ON DELETE NULL** - (**can FK have null values?**)
    1. CREATE TABLE ORDER (
    order_id int PRIMARY KEY,
    delivery_date DATE,
    cust_id INT,
    FOREIGN KEY(cust_id) REFERENCES customer(id) ON DELETE SET NULL
    );

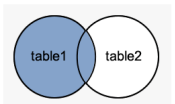
4. **REPLACE**
    1. Primarily used for already present tuple in a table.
    2. As UPDATE, using REPLACE with the help of WHERE clause in PK, then that row will be replaced.
    3. As INSERT, if there is no duplicate data new tuple will be inserted.
    4. REPLACE INTO student (id, class) VALUES(4, 3);
    5. REPLACE INTO table SET col1 = val1, col2 = val2;

JOINING TABLES

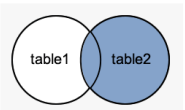
1. All **RDBMS** are relational in nature, we refer to other tables to get meaningful outcomes.
2. FK are used to do reference to other table.
3. **INNER JOIN**
    1. Returns a resultant table that has matching values from both the tables or all the tables.
    2. SELECT column-list FROM table1 INNER JOIN table2 ON condition1
    INNER JOIN table3 ON condition2
    ...;



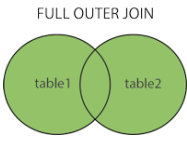
    3. **Alias in MySQL (AS)**
        1. Aliases in MySQL is used to give a temporary name to a table or a column in a table for the purpose of a particular query. It works as a nickname for expressing the tables or column names. It makes the query short and neat.
        2. SELECT col_name AS alias_name FROM table_name;
        3. SELECT col_name1, col_name2,... FROM table_name AS alias_name;
4. **OUTER JOIN**
    1. **LEFT JOIN**
        1. This returns a resulting table that all the data from left table and the matched data from the right table.
        2. SELECT columns FROM table LEFT JOIN table2 ON Join_Condition;



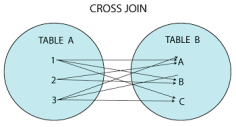
    2. **RIGHT JOIN**
        1. This returns a resulting table that all the data from right table and the matched data from the left table.
        2. SELECT columns FROM table RIGHT JOIN table2 ON join_cond;



    3. **FULL JOIN**
        1. This returns a resulting table that contains all data when there is a match on left or right table data.
        2. **Emulated** in MySQL using LEFT and RIGHT JOIN.
        3. LEFT JOIN UNION RIGHT JOIN.
        4. SELECT columns FROM table1 as t1 LEFT JOIN table2 as t2 ON t1.id = t2.id
        UNION
        SELECT columns FROM table1 as t1 RIGHT JOIN table2 as t2 ON t1.id = t2.id;
        5. UNION ALL, can also be used this will duplicate values as well while UNION gives unique values.



5. **CROSS JOIN**
    1. This returns all the cartesian products of the data present in both tables. Hence, all possible variations are reflected in the output.
    2. Used rarely in practical purpose.
    3. Table-1 has 10 rows and table-2 has 5, then resultant would have 50 rows.
    4. SELECT column-lists FROM table1 CROSS JOIN table2;



6. **SELF JOIN**

1. It is used to get the output from a particular table when the same table is joined to itself.
2. Used very less.
3. Emulated using INNER JOIN.
4. SELECT columns FROM table as t1 INNER JOIN table as t2 ON t1.id = t2.id;

7. Join without using join keywords.

1. SELECT * FROM table1, table2 WHERE condition;
2. e.g., SELECT artist_name, album_name, year_recordedFROM artist, albumWHERE artist.id = album.artist_id;

SET OPERATIONS

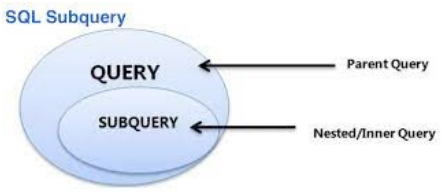
1. Used to combine multiple select statements.
2. Always gives distinct rows.

JOIN	SET Operations
Combines multiple tables based on matching condition.	Combination is resulting set from two or more SELECT statements.
Column wise combination.	Row wise combination.
Data types of two tables can be different.	Datatypes of corresponding columns from each table should be the same.
Can generate both distinct or duplicate rows.	Generate distinct rows.
The number of column(s) selected may or may not be the same from each table.	The number of column(s) selected must be the same from each table.
Combines results horizontally.	Combines results vertically.

3. **UNION**
   1. Combines two or more SELECT statements.
   2. SELECT * FROM table1
      UNION
      SELECT * FROM table2;
   3. Number of column, order of column must be same for table1 and table2.
4. **INTERSECT**
   1. Returns common values of the tables.
   2. Emulated.
   3. SELECT DISTINCT column-list FROM table-1 INNER JOIN table-2 USING(join_cond);
   4. SELECT DISTINCT * FROM table1 INNER JOIN table2 ON USING(id);
5. **MINUS**
   1. This operator returns the distinct row from the first table that does not occur in the second table.
   2. Emulated.
   3. SELECT column_list FROM table1 LEFT JOIN table2 ON condition WHERE table2.column_name IS NULL;
   4. e.g., SELECT id FROM table-1 LEFT JOIN table-2 USING(id) WHERE table-2.id IS NULL;

SUB QUERIES

1. Outer query depends on inner query.
2. Alternative to joins.
3. Nested queries.
4. SELECT column_list (s) FROM table_name WHERE column_name OPERATOR (SELECT column_list (s) FROM table_name [WHERE]);
5. e.g., SELECT * FROM table1 WHERE col1 IN (SELECT col1 FROM table1);
6. Sub queries exist mainly in 3 clauses
   1. Inside a WHERE clause.



2. Inside a FROM clause.
3. Inside a SELECT clause.

7. **Subquery using FROM clause**
   1. SELECT MAX(rating) FROM (SELECT * FROM movie WHERE country = 'India') as temp;
8. **Subquery using SELECT**
   1. SELECT (SELECT column_list(s) FROM T_name WHERE condition), columnList(s) FROM T2_name WHERE condition;
9. **Derived Subquery**
   1. SELECT columnLists(s) FROM (SELECT columnLists(s) FROM table_name WHERE [condition]) as new_table_name;
10. **Co-related sub-queries**
    1. With a normal nested subquery, the inner SELECT query runs first and executes once, returning values to be used by the main query. A correlated subquery, however, executes once for each candidate row considered by the outer query. In other words, the inner query is driven by the outer query.

```
SELECT column1, column2, ....
FROM table1 as outer
WHERE column1 operator
                    (SELECT column1, column2
                     FROM table2
                     WHERE expr1 =
                                 outer.expr2);
```

JOIN VS SUB-QUERIES

JOINS	SUBQUERIES
Faster	Slower
Joins maximise calculation burden on DBMS	Keeps responsibility of calculation on user.
Complex, difficult to understand and implement	Comparatively easy to understand and implement.
Choosing optimal join for optimal use case is difficult	Easy.

MySQL VIEWS

1. A view is a database object that has no values. Its contents are based on the base table. It contains rows and columns similar to the real table.
2. In MySQL, the View is a **virtual table** created by a query by joining one or more tables. It is operated similarly to the base table but does not contain any data of its own.
3. The View and table have one main difference that the views are definitions built on top of other tables (or views). If any changes occur in the underlying table, the same changes reflected in the View also.
4. CREATE VIEW view_name AS SELECT columns FROM tables [WHERE conditions];
5. ALTER VIEW view_name AS SELECT columns FROM table WHERE conditions;
6. DROP VIEW IF EXISTS view_name;
7. CREATE VIEW Trainer AS SELECT c.course_name, c.trainer, t.email FROM courses c, contact t WHERE c.id = t.id; (View using Join clause).

NOTE: We can also import/export table schema from files (.csv or json).

```
CREATE DATABASE ORG;
SHOW DATABASES;
USE ORG;

CREATE TABLE Worker (
        WORKER_ID INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
        FIRST_NAME CHAR(25),
        LAST_NAME CHAR(25),
        SALARY INT(15),
        JOINING_DATE DATETIME,
        DEPARTMENT CHAR(25)
);

INSERT INTO Worker
        (WORKER_ID, FIRST_NAME, LAST_NAME, SALARY, JOINING_DATE, DEPARTMENT) VALUES
                (001, 'Monika', 'Arora', 100000, '14-02-20 09.00.00', 'HR'),
                (002, 'Niharika', 'Verma', 80000, '14-06-11 09.00.00', 'Admin'),
                (003, 'Vishal', 'Singhal', 300000, '14-02-20 09.00.00', 'HR'),
                (004, 'Amitabh', 'Singh', 500000, '14-02-20 09.00.00', 'Admin'),
                (005, 'Vivek', 'Bhati', 500000, '14-06-11 09.00.00', 'Admin'),
                (006, 'Vipul', 'Diwan', 200000, '14-06-11 09.00.00', 'Account'),
                (007, 'Satish', 'Kumar', 75000, '14-01-20 09.00.00', 'Account'),
                (008, 'Geetika', 'Chauhan', 90000, '14-04-11 09.00.00', 'Admin');

SELECT * FROM Title;

CREATE TABLE Bonus (
        WORKER_REF_ID INT,
        BONUS_AMOUNT INT(10),
        BONUS_DATE DATETIME,
        FOREIGN KEY (WORKER_REF_ID)
```

```
                REFERENCES Worker(WORKER_ID)
    ON DELETE CASCADE
);

INSERT INTO Bonus
        (WORKER_REF_ID, BONUS_AMOUNT, BONUS_DATE) VALUES
                (001, 5000, '16-02-20'),
                (002, 3000, '16-06-11'),
                (003, 4000, '16-02-20'),
                (001, 4500, '16-02-20'),
                (002, 3500, '16-06-11');

CREATE TABLE Title (
        WORKER_REF_ID INT,
        WORKER_TITLE CHAR(25),
        AFFECTED_FROM DATETIME,
        FOREIGN KEY (WORKER_REF_ID)
                REFERENCES Worker(WORKER_ID)
    ON DELETE CASCADE
);

INSERT INTO Title
        (WORKER_REF_ID, WORKER_TITLE, AFFECTED_FROM) VALUES
 (001, 'Manager', '2016-02-20 00:00:00'),
 (002, 'Executive', '2016-06-11 00:00:00'),
 (008, 'Executive', '2016-06-11 00:00:00'),
 (005, 'Manager', '2016-06-11 00:00:00'),
 (004, 'Asst. Manager', '2016-06-11 00:00:00'),
 (007, 'Executive', '2016-06-11 00:00:00'),
 (006, 'Lead', '2016-06-11 00:00:00'),
 (003, 'Lead', '2016-06-11 00:00:00');
```

-- Q-1. Write an SQL query to fetch “FIRST_NAME” from Worker table using the alias name as <WORKER_NAME>.

```
select first_name AS WORKER_NAME from worker;
```

-- Q-2. Write an SQL query to fetch “FIRST_NAME” from Worker table in upper case.

```
select UPPER(first_name) from worker;
```

-- Q-3. Write an SQL query to fetch unique values of DEPARTMENT from Worker table.

```
SELECT distinct department from worker;
```

-- Q-4. Write an SQL query to print the first three characters of FIRST_NAME from Worker table.

```
select substring(first_name, 1, 3) from worker;
```

-- Q-5. Write an SQL query to find the position of the alphabet (‘b’) in the first name column ‘Amitabh’ from Worker table.

```
select INSTR(first_name, 'B') from worker where first_name = 'Amitabh';
```

-- Q-6. Write an SQL query to print the FIRST_NAME from Worker table after removing white spaces from the right side.

```
select RTRIM(first_name) from worker;
```

-- Q-7. Write an SQL query to print the DEPARTMENT from Worker table after removing white spaces from the left side.

```
select LTRIM(first_name) from worker;
```

-- Q-8. Write an SQL query that fetches the unique values of DEPARTMENT from Worker table and prints its length.

```
select distinct department, LENGTH(department) from worker;
```

-- Q-9. Write an SQL query to print the FIRST_NAME from Worker table after replacing ‘a’ with ‘A’.

```
select REPLACE(first_name, 'a', 'A')  from worker;
```

-- Q-10. Write an SQL query to print the FIRST_NAME and LAST_NAME from Worker table into a single column COMPLETE_NAME.

-- A space char should separate them.

```
select CONCAT(first_name, ' ', last_name) AS COMPLETE_NAME from worker;
```

-- Q-11. Write an SQL query to print all Worker details from the Worker table order by FIRST_NAME Ascending.

```
select * from worker ORDER by first_name;
```

-- Q-12. Write an SQL query to print all Worker details from the Worker table order by

-- FIRST_NAME Ascending and DEPARTMENT Descending.

```
select * from worker order by first_name, department DESC;
```

-- Q-13. Write an SQL query to print details for Workers with the first name as “Vipul” and “Satish” from Worker table.

```
select * from worker where first_name IN ('Vipul', 'Satish');
```

-- Q-14. Write an SQL query to print details of workers excluding first names, “Vipul” and “Satish” from Worker table.

```
select * from worker where first_name NOT IN ('Vipul', 'Satish');
```

-- Q-15. Write an SQL query to print details of Workers with DEPARTMENT name as “Admin*”.

```
select * from worker where department LIKE 'Admin%';
```

-- Q-16. Write an SQL query to print details of the Workers whose FIRST_NAME contains ‘a’.

```
select * from worker where first_name LIKE '%a%';
```

-- Q-17. Write an SQL query to print details of the Workers whose FIRST_NAME ends with ‘a’.

```
select * from worker where first_name LIKE '%a';
```

-- Q-18. Write an SQL query to print details of the Workers whose FIRST_NAME ends with ‘h’ and contains six alphabets.

```
select * from worker where first_name LIKE '_____h';
```

-- Q-19. Write an SQL query to print details of the Workers whose SALARY lies between 100000 and 500000.

```
select * from worker where salary between 100000 AND 500000;
```

-- Q-20. Write an SQL query to print details of the Workers who have joined in Feb'2014.

```
select * from worker where YEAR(joining_date) = 2014 AND MONTH(joining_date) = 02;
```

-- Q-21. Write an SQL query to fetch the count of employees working in the department 'Admin'.

```
select department, count(*) from worker where department = 'Admin';
```

-- Q-22. Write an SQL query to fetch worker full names with salaries >= 50000 and <= 100000.

```
select concat(first_name, ' ', last_name) from worker
where salary between 50000 and 100000;
```

-- Q-23. Write an SQL query to fetch the no. of workers for each department in the descending order.

```
select department, count(worker_id) AS no_of_worker from worker group by department
ORDER BY no_of_worker desc;
```

-- Q-24. Write an SQL query to print details of the Workers who are also Managers.

```
select w.* from worker as w inner join title as t on w.worker_id = t.worker_ref_id where
t.worker_title = 'Manager';
```

-- Q-25. Write an SQL query to fetch number (more than 1) of same titles in the ORG of different types.

```
select worker_title, count(*) as count from title group by worker_title having count > 1;
```

-- Q-26. Write an SQL query to show only odd rows from a table.

```
-- select * from worker where MOD (WORKER_ID, 2) != 0;
select * from worker where MOD (WORKER_ID, 2) <> 0;
```

-- Q-27. Write an SQL query to show only even rows from a table.

```
select * from worker where MOD (WORKER_ID, 2) = 0;
```

-- Q-28. Write an SQL query to clone a new table from another table.

```
CREATE TABLE worker_clone LIKE worker;
INSERT INTO worker_clone select * from worker;
select * from worker_clone;
```

-- Q-29. Write an SQL query to fetch intersecting records of two tables.

```
select worker.* from worker inner join worker_clone using(worker_id);
```

-- Q-30. Write an SQL query to show records from one table that another table does not have.

```
-- MINUS
select worker.* from worker left join worker_clone using(worker_id) WHERE
worker_clone.worker_id is NULL;
```

-- Q-31. Write an SQL query to show the current date and time.

```
-- DUAL
select curdate();
select now();
```

-- Q-32. Write an SQL query to show the top n (say 5) records of a table order by descending salary.

```
select * from worker order by salary desc LIMIT 5;
```

-- Q-33. Write an SQL query to determine the nth (say n=5) highest salary from a table.

```
select * from worker order by salary desc LIMIT 4,1;
```

-- Q-34. Write an SQL query to determine the 5th highest salary without using LIMIT keyword.

```
select salary from worker w1
WHERE 4 = (
SELECT COUNT(DISTINCT (w2.salary))
```

```
from worker w2
where w2.salary >= w1.salary
);

-- Q-35. Write an SQL query to fetch the list of employees with the same salary.
select w1.* from worker w1, worker w2 where w1.salary = w2.salary and w1.worker_id != w2.worker_id;

-- Q-36. Write an SQL query to show the second highest salary from a table using sub-query.
select max(salary) from worker
where salary not in (select max(salary) from worker);

-- Q-37. Write an SQL query to show one row twice in results from a table.
select * from worker
UNION ALL
select * from worker ORDER BY worker_id;

-- Q-38. Write an SQL query to list worker_id who does not get bonus.
select worker_id from worker where worker_id not in (select worker_ref_id from bonus);

-- Q-39. Write an SQL query to fetch the first 50% records from a table.
select * from worker where worker_id <= ( select count(worker_id)/2 from worker);

-- Q-40. Write an SQL query to fetch the departments that have less than 4 people in it.
select department, count(department) as depCount from worker group by department having depCount < 4;

-- Q-41. Write an SQL query to show all departments along with the number of people in there.
select department, count(department) as depCount from worker group by department;

-- Q-42. Write an SQL query to show the last record from a table.
select * from worker where worker_id = (select max(worker_id) from worker);
```

-- Q-43. Write an SQL query to fetch the first row of a table.

```
select * from worker where worker_id = (select min(worker_id) from worker);
```

-- Q-44. Write an SQL query to fetch the last five records from a table.

```
(select * from worker order by worker_id desc limit 5) order by worker_id;
```

-- Q-45. Write an SQL query to print the name of employees having the highest salary in each department.

```
select w.department, w.first_name, w.salary from
 (select max(salary) as maxsal, department from worker group by department) temp
inner join worker w on temp.department = w.department and temp.maxsal = w.salary;
```

-- Q-46. Write an SQL query to fetch three max salaries from a table using co-related subquery

```
select distinct salary from worker w1
where 3 >= (select count(distinct salary) from worker w2 where w1.salary <= w2.salary) order by w1.salary desc;
```

-- DRY RUN AFTER REVISING THE CORELATED SUBQUERY CONCEPT FROM LEC-9.

```
select distinct salary from worker order by salary desc limit 3;
```

-- Q-47. Write an SQL query to fetch three min salaries from a table using co-related subquery

```
select distinct salary from worker w1
where 3 >= (select count(distinct salary) from worker w2 where w1.salary >= w2.salary) order by w1.salary desc;
```

-- Q-48. Write an SQL query to fetch nth max salaries from a table.

```
select distinct salary from worker w1
where n >= (select count(distinct salary) from worker w2 where w1.salary <= w2.salary) order by w1.salary desc;
```

-- Q-49. Write an SQL query to fetch departments along with the total salaries paid for each of them.

```
select department , sum(salary) as depSal from worker group by department order by depSal desc;
```

-- Q-50. Write an SQL query to fetch the names of workers who earn the highest salary.

```
select first_name, salary from worker where salary = (select max(Salary) from worker);
```

LEC-11: Normalisation

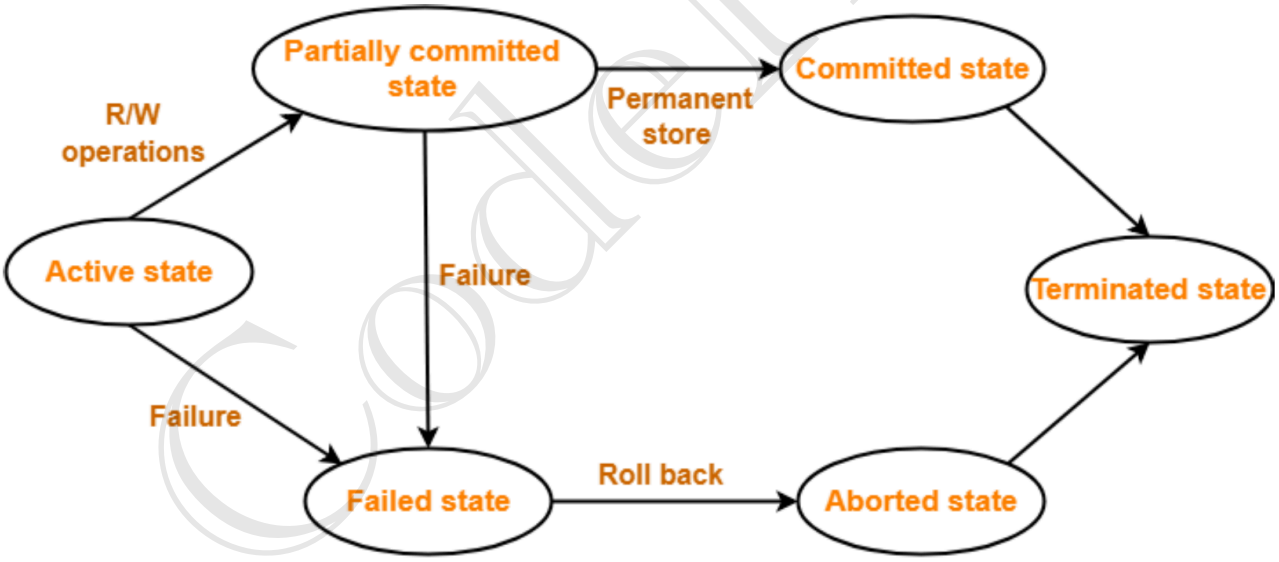
1. **Normalisation** is a step towards DB optimisation.
2. **Functional Dependency (FD)**
    1. It's a relationship between the primary key attribute (usually) of the relation to that of the other attribute of the relation.
    2. X -> Y, the left side of FD is known as a **Determinant**, the right side of the production is known as a **Dependent**.
    3. **Types of FD**
        1. **Trivial FD**
            1. A → B has trivial functional dependency if B is a subset of A. A->A, B->B are also Trivial FD.
        2. **Non-trivial FD**
            1. A → B has a non-trivial functional dependency if B is not a subset of A. [A intersection B is NULL].
    4. **Rules of FD (Armstrong's axioms)**
        1. **Reflexive**
            1. If 'A' is a set of attributes and 'B' is a subset of 'A'. Then, A→ B holds.
            2. If A ⊇ B then A → B.
        2. **Augmentation**
            1. If B can be determined from A, then adding an attribute to this functional dependency won't change anything.
            2. If A→ B holds, then AX→ BX holds too. 'X' being a set of attributes.
        3. **Transitivity**
            1. If A determines B and B determines C, we can say that A determines C.
            2. if A→ B and B→ C then A→ C.
3. **Why Normalisation?**
    1. To avoid redundancy in the DB, not to store redundant data.
4. **What happen if we have redundant data?**
    1. Insertion, deletion and updation anomalies arises.
5. **Anomalies**
    1. Anomalies means abnormalities, there are three types of anomalies introduced by data redundancy.
    2. **Insertion anomaly**
        1. When certain data (attribute) can not be inserted into the DB without the presence of other data.
    3. **Deletion anomaly**
        1. The delete anomaly refers to the situation where the deletion of data results in the unintended loss of some other important data.
    4. **Updation anomaly** (or modification anomaly)
        1. The update anomaly is when an update of a single data value requires multiple rows of data to be updated.
        2. Due to updation to many places, may be **Data inconsistency** arises, if one forgets to update the data at all the intended places.
    5. Due to these anomalies, **DB size increases** and **DB performance become very slow.**
    6. To rectify these anomalies and the effect of these of DB, we use **Database optimisation technique** called **NORMALISATION**.
6. **What is Normalisation?**
    1. Normalisation is used to minimise the redundancy from a relations. It is also used to eliminate undesirable characteristics like Insertion, Update, and Deletion Anomalies.
    2. Normalisation divides the composite attributes into individual attributes OR larger table into smaller and links them using relationships.
    3. The normal form is used to reduce redundancy from the database table.
7. **Types of Normal forms**
    1. **1NF**
        1. Every relation cell must have atomic value.
        2. Relation must not have multi-valued attributes.

2. **2NF**
    1. Relation must be in 1NF.
    2. There should not be any partial dependency.
        1. All non-prime attributes must be fully dependent on PK.
        2. Non prime attribute can not depend on the part of the PK.
3. **3NF**
    1. Relation must be in 2NF.
    2. No transitivity dependency exists.
        1. Non-prime attribute should not find a non-prime attribute.
4. **BCNF (Boyce-Codd normal form)**
    1. Relation must be in 3NF.
    2. FD: A -> B, A must be a super key.
        1. We must not derive prime attribute from any prime or non-prime attribute.

8. **Advantages of Normalisation**
    1. Normalisation helps to minimise data redundancy.
    2. Greater overall database organisation.
    3. Data consistency is maintained in DB.

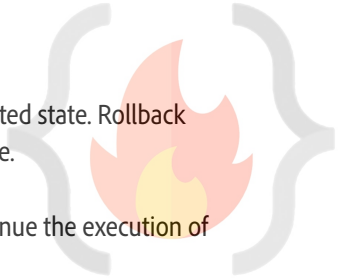
LEC-12: Transaction

1. **Transaction**
    1. A unit of work done against the DB in a logical sequence.
    2. Sequence is very important in transaction.
    3. It is a logical unit of work that contains one or more SQL statements. The result of all these statements in a transaction either gets completed successfully (all the changes made to the database are permanent) or if at any point any failure happens it gets rollbacked (all the changes being done are undone.)
2. **ACID Properties**
    1. To ensure integrity of the data, we require that the DB system maintain the following properties of the transaction.
    2. **Atomicity**
        1. Either all operations of transaction are reflected properly in the DB, or none are.
    3. **Consistency**
        1. Integrity constraints must be maintained before and after transaction.
        2. DB must be consistent after transaction happens.
    4. **Isolation**
        1. Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions Ti and Tj, it appears to Ti that either Tj finished execution before Ti started, or Tj started execution after Ti finished. Thus, each transaction is unaware of other transactions executing concurrently in the system.
        2. Multiple transactions can happen in the system in isolation, without interfering each other.
    5. **Durability**
        1. After transaction completes successfully, the changes it has made to the database persist, even if there are system failures.
3. **Transaction states**



Transaction States in DBMS

    1. **Active state**
        1. The very first state of the life cycle of the transaction, all the read and write operations are being performed. If they execute without any error the T comes to Partially committed state. Although if any error occurs then it leads to a Failed state.
    2. **Partially committed state**
        1. After transaction is executed the changes are saved in the buffer in the main memory. If the changes made are permanent on the DB then the state will transfer to the committed state and if there is any failure, the T will go to Failed state.
    3. **Committed state**



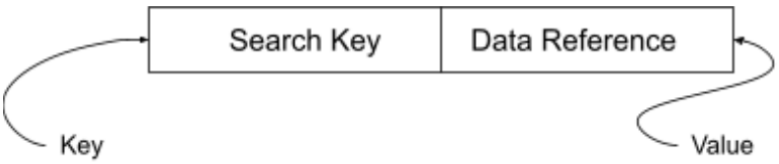
    1. When updates are made permanent on the DB. Then the T is said to be in the committed state. Rollback can't be done from the committed states. New consistent state is achieved at this stage.
4. **Failed state**
    1. When T is being executed and some failure occurs. Due to this it is impossible to continue the execution of the T.
5. **Aborted state**
    1. When T reaches the failed state, all the changes made in the buffer are reversed. After that the T rollback completely. T reaches abort state after rollback. DB's state prior to the T is achieved.
6. **Terminated state**
    1. A transaction is said to have terminated if has either committed or aborted.

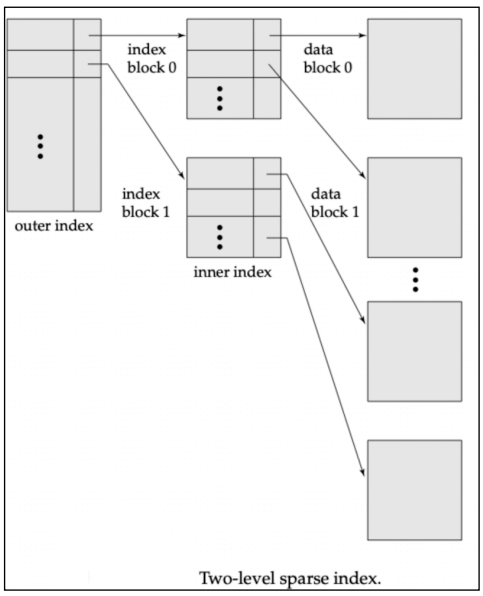
LEC-13: How to implement Atomicity and Durability in Transactions

1. **Recovery Mechanism Component** of DBMS supports **atomicity and durability.**
2. **Shadow-copy scheme**
    1. Based on making copies of DB (aka, **shadow copies)**.
    2. Assumption only one Transaction (T) is active at a time.
    3. A pointer called **db-pointer** is maintained on the **disk**; which at any instant points to current copy of DB.
    4. T, that wants to update DB first creates a complete copy of DB.
    5. All further updates are done on new DB copy leaving the original copy (shadow copy) untouched.
    6. If at any point the **T has to be aborted** the system deletes the new copy. And the old copy is not affected.
    7. If T success, it is committed as,
        1. OS makes sure all the pages of the new copy of DB written on the disk.
        2. DB system updates the db-pointer to point to the new copy of DB.
        3. New copy is now the current copy of DB.
        4. The old copy is deleted.
        5. The T is said to have been **COMMITTED** at the point where the updated db-pointer is written to disk.
    8. **Atomicity**
        1. If T fails at any time before db-pointer is updated, the old content of DB are not affected.
        2. T abort can be done by just deleting the new copy of DB.
        3. Hence, either all updates are reflected or none.
    9. **Durability**
        1. Suppose, system fails are any time before the updated db-pointer is written to disk.
        2. When the system restarts, it will read db-pointer & will thus, see the original content of DB and none of the effects of T will be visible.
        3. T is assumed to be successful only when db-pointer is updated.
        4. If **system fails after** db-pointer has been updated. Before that all the pages of the new copy were written to disk. Hence, when system restarts, it will read new DB copy.
    10. The implementation is dependent on write to the db-pointer being atomic. Luckily, disk system provide atomic updates to entire block or at least a disk sector. So, we make sure db-pointer lies entirely in a single sector. By storing db-pointer at the beginning of a block.
    11. **Inefficient**, as entire DB is copied for every Transaction.
3. **Log-based recovery methods**
    1. The log is a sequence of records. Log of each transaction is maintained in some **stable storage** so that if any failure occurs, then it can be recovered from there.
    2. If any operation is performed on the database, then it will be recorded in the log.
    3. But the process of storing the logs should be done **before** the actual transaction is applied in the database.
    4. **Stable storage** is a classification of computer data storage technology that guarantees atomicity for any given write operation and allows software to be written that is robust against some hardware and power failures.
    5. **Deferred DB Modifications**
        1. Ensuring **atomicity** by recording all the DB modifications in the log but deferring the execution of all the write operations until the final action of the T has been executed.
        2. Log information is used to execute deferred writes when T is completed.
        3. If system crashed before the T completes, or if T is aborted, the information in the logs are ignored.
        4. If T completes, the records associated to it in the log file are used in executing the deferred writes.
        5. If failure occur while this updating is taking place, we preform redo.
    6. **Immediate DB Modifications**
        1. DB modifications to be output to the DB while the T is still in active state.
        2. DB modifications written by active T are called uncommitted modifications.
        3. In the event of crash or T failure, system uses old value field of the log records to restore modified values.
        4. Update takes place only after log records in a stable storage.
        5. Failure handling
            1. System failure before T completes, or if T aborted, then old value field is used to undo the T.
            2. If T completes and system crashes, then new value field is used to redo T having commit logs in the logs.

LEC-14: Indexing in DBMS

1. **Indexing** is used to **optimise the performance** of a database by minimising the number of disk accesses required when a query is processed.
2. The index is a type of **data structure.** It is used to locate and access the data in a database table quickly.
3. **Speeds up operation** with read operations like **SELECT** queries, **WHERE** clause etc.
4. **Search Key**: Contains copy of primary key or candidate key of the table or something else.
5. **Data Reference**: Pointer holding the address of disk block where the value of the corresponding key is stored.
6. Indexing is **optional**, but increases access speed. It is not the primary mean to access the tuple, it is the secondary mean.
7. **Index file is always sorted.**
8. **Indexing Methods**
    1. **Primary Index (Clustering Index)**
        1. A file may have several indices, on different search keys. If the data file containing the records is sequentially ordered, a Primary index is an index whose search key also defines the sequential order of the file.
        2. **NOTE**: The term primary index is sometimes used to mean an index on a primary key. However, such usage is **nonstandard** and **should be avoided.**
        3. All files are ordered sequentially on some search key. It could be Primary Key or non-primary key.
        4. **Dense And Sparse Indices**
            1. **Dense Index**
                1. The dense index contains an index record for every search key value in the data file.
                2. The index record contains the search-key value and a pointer to the first data record with that search-key value. The rest of the records with the same search-key value would be stored sequentially after the first record.
                3. It needs more space to store index record itself. The index records have the search key and a pointer to the actual record on the disk.
            2. **Sparse Index**
                1. An index record appears for only some of the search-key values.
                2. Sparse Index helps you to resolve the issues of dense Indexing in DBMS. In this method of indexing technique, a range of index columns stores the same data block address, and when data needs to be retrieved, the block address will be fetched.
        5. Primary Indexing can be based on Data file is sorted w.r.t Primary Key attribute or non-key attributes.
        6. **Based on Key attribute**
            1. Data file is sorted w.r.t primary key attribute.
            2. PK will be used as search-key in Index.
            3. Sparse Index will be formed i.e., no. of entries in the index file = no. of blocks in datafile.
        7. **Based on Non-Key attribute**
            1. Data file is sorted w.r.t non-key attribute.
            2. No. Of entries in the index = unique non-key attribute value in the data file.
            3. This is dense index as, all the unique values have an entry in the index file.
            4. E.g., Let's assume that a company recruited many employees in various departments. In this case, clustering indexing in DBMS should be created for all employees who belong to the same dept.
        8. **Multi-level Index**
            1. Index with two or more levels.
            2. If the single level index become enough large that the binary search it self would take much time, we can break down indexing into multiple levels.





    2. **Secondary Index (Non-Clustering Index)**
        1. Datafile is unsorted. Hence, Primary Indexing is not possible.
        2. Can be done on key or non-key attribute.
        3. Called secondary indexing because normally one indexing is already applied.
        4. No. Of entries in the index file = no. of records in the data file.
        5. It's an example of Dense index.

9. **Advantages of Indexing**
    1. Faster access and retrieval of data.
    2. IO is less.
10. **Limitations of Indexing**
    1. Additional space to store index table
    2. Indexing Decrease performance in INSERT, DELETE, and UPDATE query.

LEC-15: NoSQL

1. **NoSQL databases** (aka "not only SQL") are non-tabular databases and store data differently than relational tables. NoSQL databases come in a variety of types based on their data model. The main types are document, key-value, wide-column, and graph. They provide **flexible schemas** and **scale** easily with **large amounts of data** and **high user loads**.
    1. They are schema free.
    2. Data structures used are not tabular, they are more flexible, has the ability to adjust dynamically.
    3. Can handle huge amount of data **(big data).**
    4. Most of the NoSQL are open sources and has the capability of horizontal scaling.
    5. **It just stores data in some format other than relational.**
2. **History behind NoSQL**
    1. NoSQL databases emerged in the late 2000s as the cost of storage dramatically decreased. Gone were the days of needing to create a complex, difficult-to-manage data model in order to avoid data duplication. Developers (rather than storage) were becoming the primary cost of software development, so NoSQL databases optimised for developer productivity.
    2. Data becoming unstructured more, hence structuring (defining schema in advance) them had becoming costly.
    3. NoSQL databases allow developers to store huge amounts of unstructured data, giving them a lot of flexibility.
    4. Recognising the need to rapidly adapt to changing requirements in a software system. Developers needed the ability to iterate quickly and make changes throughout their software stack — all the way down to the database. NoSQL databases gave them this flexibility.
    5. Cloud computing also rose in popularity, and developers began using public clouds to host their applications and data. They wanted the ability to distribute data across multiple servers and regions to make their applications resilient, to scale out instead of scale up, and to intelligently geo-place their data. Some NoSQL databases like MongoDB provide these capabilities.
3. **NoSQL Databases Advantages**
    A. **Flexible Schema**
        1. RDBMS has pre-defined schema, which become an issue when we do not have all the data with us or we need to change the schema. It's a huge task to change schema on the go.

    B. **Horizontal Scaling**
        1. Horizontal scaling, also known as scale-out, refers to bringing on additional nodes to share the load. This is difficult with relational databases due to the difficulty in spreading out related data across nodes. With non-relational databases, this is made simpler since collections are self-contained and not coupled relationally. This allows them to be distributed across nodes more simply, as queries do not have to "join" them together across nodes.
        2. Scaling horizontally is achieved through **Sharding** OR **Replica-sets**.

    C. **High Availability**
        1. NoSQL databases are highly available due to its auto replication feature i.e. whenever any kind of failure happens data replicates itself to the preceding consistent state.
        2. If a server fails, we can access that data from another server as well, as in NoSQL database data is stored at multiple servers.

    D. **Easy insert and read operations.**
        1. Queries in NoSQL databases can be faster than SQL databases. Why? Data in SQL databases is typically normalised, so queries for a single object or entity require you to join data from multiple tables. As your tables grow in size, the joins can become expensive. However, data in NoSQL databases is typically stored in a way that is optimised for queries. The rule of thumb when you use MongoDB is data that is accessed together should be stored together. Queries typically do not require joins, so the queries are very fast.
        2. But difficult delete or update operations.

    E. **Caching** mechanism**.**

    F. **NoSQL** use case is more for **Cloud** applications.
4. **When to use NoSQL?**
    1. Fast-paced Agile development
    2. Storage of structured and semi-structured data
    3. Huge volumes of data
    4. Requirements for scale-out architecture
    5. Modern application paradigms like micro-services and real-time streaming.
5. **NoSQL DB Misconceptions**
    1. Relationship data is best suited for relational databases.
        1. A common misconception is that NoSQL databases or non-relational databases don't store relationship data well. NoSQL databases can store relationship data — they just store it differently than relational databases do. In fact, when compared with relational databases, many find modelling relationship data in NoSQL databases to be easier than in relational databases, because related data doesn't have to be split between tables. NoSQL data models allow related data to be nested within a single data structure.
    2. NoSQL databases don't support ACID transactions.

1. Another common misconception is that NoSQL databases don't support ACID transactions. Some NoSQL databases like MongoDB do, in fact, support ACID transactions.

6. **Types of NoSQL Data Models**
    1. **Key-Value Stores**
        1. The simplest type of NoSQL database is a key-value store. Every data element in the database is stored as a key value pair consisting of an attribute name (or "key") and a value. In a sense, a key-value store is like a relational database with only two columns: the key or attribute name (such as "state") and the value (such as "Alaska").
        2. **Use cases** include shopping carts, user preferences, and user profiles.
        3. e.g., Oracle NoSQL, Amazon DynamoDB, MongoDB also supports Key-Value store, Redis.
        4. A key-value database associates a value (which can be anything from a number or simple string to a complex object) with a key, which is used to keep track of the object. In its simplest form, a key-value store is like a dictionary/array/map object as it exists in most programming paradigms, but which is stored in a persistent way and managed by a Database Management System (DBMS).
        5. Key-value databases use compact, efficient index structures to be able to quickly and reliably locate a value by its key, making them ideal for systems that need to be able to find and retrieve data in constant time.
        6. There are several use-cases where choosing a key value store approach is an optimal solution:
            a) Real time random data access, e.g., user session attributes in an online application such as gaming or finance.
            b) Caching mechanism for frequently accessed data or configuration based on keys.
            c) Application is designed on simple key-based queries.
    2. **Column-Oriented / Columnar / C-Store / Wide-Column**
        1. The data is stored such that each row of a column will be next to other rows from that same column.
        2. While a relational database stores data in rows and reads data row by row, a column store is organised as a set of columns. This means that when you want to run analytics on a small number of columns, you can read those columns directly without consuming memory with the unwanted data. Columns are often of the same type and benefit from more efficient compression, making reads even faster. Columnar databases can quickly aggregate the value of a given column (adding up the total sales for the year, for example). **Use cases** include analytics.
        3. e.g., Cassandra, RedShift, Snowflake.
    3. **Document Based Stores**
        1. This DB store data in documents similar to JSON (JavaScript Object Notation) objects. Each document contains pairs of fields and values. The values can typically be a variety of types including things like strings, numbers, booleans, arrays, or objects.
        2. **Use cases** include e-commerce platforms, trading platforms, and mobile app development across industries.
        3. Supports ACID properties hence, suitable for Transactions.
        4. e.g., MongoDB, CouchDB.
    4. **Graph Based Stores**
        1. A graph database focuses on the relationship between data elements. Each element is stored as a node (such as a person in a social media graph). The connections between elements are called links or relationships. In a graph database, connections are first-class elements of the database, stored directly. In relational databases, links are implied, using data to express the relationships.
        2. A graph database is optimised to capture and search the connections between data elements, overcoming the overhead associated with JOINing multiple tables in SQL.
        3. Very few real-world business systems can survive solely on graph queries. As a result graph databases are usually run alongside other more traditional databases.
        4. **Use cases** include fraud detection, social networks, and knowledge graphs.
7. **NoSQL Databases Dis-advantages**
    1. **Data Redundancy**
        1. Since data models in NoSQL databases are typically optimised for queries and not for reducing data duplication, NoSQL databases can be larger than SQL databases. Storage is currently so cheap that most consider this a minor drawback, and some NoSQL databases also support compression to reduce the storage footprint.
    2. Update & Delete operations are **costly**.
    3. All type of NoSQL Data model doesn't fulfil all of your application needs
        1. Depending on the NoSQL database type you select, you may not be able to achieve all of your use cases in a single database. For example, graph databases are excellent for analysing relationships in your data but may not provide what you need for everyday retrieval of the data such as range queries. When selecting a NoSQL database, consider what your use cases will be and if a general purpose database like MongoDB would be a better option.
    4. Doesn't support ACID properties in general.
    5. Doesn't support data entry with consistency constraints.

8. SQL vs NoSQL

	SQL Databases	NoSQL Databases
Data Storage Model	Tables with fixed rows and columns	Document: JSON documents, Key-value: key-value pairs, Wide-column: tables with rows and dynamic columns, Graph: nodes and edges
Development History	Developed in the 1970s with a focus on reducing data duplication	Developed in the late 2000s with a focus on scaling and allowing for rapid application change driven by agile and DevOps practices.
Examples	Oracle, MySQL, Microsoft SQL Server, and PostgreSQL	Document: MongoDB and CouchDB, Key-value: Redis and DynamoDB, Wide-column: Cassandra and HBase, Graph: Neo4j and Amazon Neptune
Primary Purpose	General Purpose	Document: general purpose, Key-value: large amounts of data with simple lookup queries, Wide-column: large amounts of data with predictable query patterns, Graph: analyzing and traversing relationships between connected data
Schemas	Fixed	Flexible
Scaling	Vertical (Scale-up)	Horizontal (scale-out across commodity servers)
ACID Properties	Supported	Not Supported, except in DB like MongoDB etc.
JOINS	Typically Required	Typically not required
Data to object mapping	Required object-relational mapping	Many do not require ORMs. MongoDB documents map directly to data structures in most popular programming languages.

LEC-16: Types of Databases

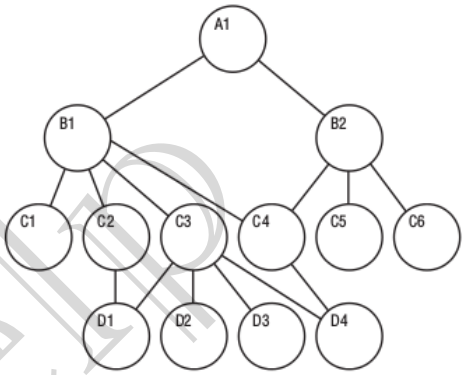
1. **Relational Databases**
    1. Based on Relational Model.
    2. Relational databases are quite **popular**, even though it was a system designed in the 1970s. Also known as relational database management systems (RDBMS), relational databases commonly use **Structured Query Language (SQL)** for operations such as **creating**, **reading**, **updating**, and **deleting** data. Relational databases store information in **discrete tables**, which can be **JOIN**ed together by fields known as **foreign** keys. For example, you might have a User table which contains information about all your users, and **join** it to a Purchases table, which contains information about all the purchases they've made. MySQL, Microsoft SQL Server, and Oracle are types of relational databases.
    3. they are ubiquitous, having acquired a steady user base since the 1970s
    4. they are highly optimised for working with structured data.
    5. they provide a stronger guarantee of data normalisation
    6. they use a well-known querying language through SQL
    7. **Scalability issues** (Horizontal Scaling).
    8. Data become huge, system become more complex.
2. **Object Oriented Databases**
    1. The object-oriented data model, is based on the **object-oriented-programming paradigm,** which is now in wide use. **Inheritance**, **object-identity**, and **encapsulation** (information hiding), with methods to provide an interface to objects, are among the key concepts of object-oriented programming that have found applications in data modelling. The object-oriented data model also supports a rich type system, including structured and collection types. While inheritance and, to some extent, complex types are also present in the E-R model, encapsulation and object-identity distinguish the object-oriented data model from the E-R model.
    2. Sometimes the database can be very complex, having multiple relations. So, maintaining a relationship between them can be tedious at times.
        1. In Object-oriented databases data is treated as an object.
        2. All bits of information come in one instantly available object package instead of multiple tables.
    3. **Advantages**
        1. Data storage and retrieval is easy and quick.
        2. Can handle complex data relations and more variety of data types that standard relational databases.
        3. Relatively friendly to model the advance real world problems
        4. Works with functionality of OOPs and Object Oriented languages.
    4. **Disadvantages**
        1. High complexity causes performance issues like read, write, update and delete operations are slowed down.
        2. Not much of a community support as isn't widely adopted as relational databases.
        3. Does not support views like relational databases.
    5. e.g., ObjectDB, GemStone etc.
3. **NoSQL Databases**
    1. NoSQL databases (aka "not only SQL") are non-tabular databases and store data differently than relational tables. NoSQL databases come in a variety of types based on their data model. The main types are document, key-value, wide-column, and graph. They provide flexible schemas and scale easily with large amounts of data and high user loads.
    2. They are schema free.
    3. Data structures used are not tabular, they are more flexible, has the ability to adjust dynamically.
    4. Can handle huge amount of data (big data).
    5. Most of the NoSQL are open sources and has the capability of horizontal scaling.
    6. It just stores data in some format other than relational.
    7. Refer **LEC-15** notes...
4. **Hierarchical Databases**
    1. As the name suggests, the hierarchical database model is most appropriate for use cases in which the main focus of information gathering is based on a **concrete hierarchy**, such as several individual employees reporting to a single department at a company.
    2. The schema for hierarchical databases is defined by its **tree-like** organisation, in which there is typically a **root** "parent" directory of data stored as records that links to various other subdirectory branches, and each subdirectory branch, or child record, may link to various other subdirectory branches.
    3. The hierarchical database structure dictates that, while a parent record can have several child records, each child record can only have **one parent** record. Data within records is stored in the form of fields, and each field can only contain one value. Retrieving hierarchical data from a hierarchical database architecture requires traversing the entire tree, starting at the root node.
    4. Since the **disk storage system** is also inherently a hierarchical structure, these models can also be used as physical models.
    5. The key **advantage** of a hierarchical database is its ease of use. The one-to-many organisation of data makes traversing the database simple and fast, which is ideal for use cases such as website drop-down menus or computer folders in systems like

Microsoft Windows OS. Due to the separation of the tables from physical storage structures, information can easily be added or deleted without affecting the entirety of the database. And most major programming languages offer functionality for reading tree structure databases.

6. The major **disadvantage** of hierarchical databases is their inflexible nature. The one-to-many structure is not ideal for complex structures as it cannot describe relationships in which each child node has multiple parents nodes. Also the tree-like organisation of data requires top-to-bottom sequential searching, which is time consuming, and requires repetitive storage of data in multiple different entities, which can be redundant.
7. e.g., IBM IMS.

5. Network Databases

1. **Extension** of Hierarchical databases
2. The child records are given the freedom to associate with multiple parent records.
3. Organised in a **Graph** structure.
4. Can handle complex relations.
5. Maintenance is tedious.
6. **M:N links** may cause slow retrieval.
7. Not much web community support.
8. e.g., Integrated Data Store (IDS), IDMS (Integrated Database Management System), Raima Database Manager, TurboIMAGE etc.



LEC-17: Clustering in DBMS

1. **Database Clustering** (making **Replica-sets**) is the process of combining more than one servers or instances connecting a single database. Sometimes one server may not be adequate to manage the amount of data or the number of requests, that is when a Data Cluster is needed. Database clustering, SQL server clustering, and SQL clustering are closely associated with SQL is the language used to manage the database information.
2. Replicate the same dataset on different servers.
3. **Advantages**
    1. **Data Redundancy:** Clustering of databases helps with data redundancy, as we store the same data at multiple servers. Don't confuse this data redundancy as repetition of the same data that might lead to some anomalies. The redundancy that clustering offers is required and is quite certain due to the synchronisation. In case any of the servers had to face a failure due to any possible reason, the data is available at other servers to access.
    2. **Load balancing:** or scalability doesn't come by default with the database. It has to be brought by clustering regularly. It also depends on the setup. Basically, what load balancing does is allocating the workload among the different servers that are part of the cluster. This indicates that more users can be supported and if for some reasons if a huge spike in the traffic appears, there is a higher assurance that it will be able to support the new traffic. One machine is not going to get all of the hits. This can provide **scaling** seamlessly as required. This **links directly to high availability.** Without load balancing, a particular machine could get overworked and traffic would slow down, leading to decrement of the traffic to zero.
    3. **High availability:** When you can access a database, it implies that it is available. High availability refers the amount of time a database is considered available. The amount of availability you need greatly depends on the number of transactions you are running on your database and how often you are running any kind of analytics on your data. With database clustering, we can reach extremely high levels of availability due to load balancing and have extra machines. In case a server got shut down the database will, however, be available.
4. **How does Clustering Work?**
    1. In cluster architecture, all requests are split with many computers so that an individual user request is executed and produced by a number of computer systems. The clustering is serviceable definitely by the ability of load balancing and high-availability. If one node collapses, the request is handled by another node. Consequently, there are few or no possibilities of absolute system failures.

LEC-18: Partitioning & Sharding in DBMS (DB Optimisation)

1. **A big problem** can be solved easily when it is chopped into several smaller sub-problems. That is what the partitioning technique does. It divides a big database containing data metrics and indexes into smaller and handy slices of data called partitions. The partitioned tables are directly used by SQL queries without any alteration. Once the database is partitioned, the data definition language can easily work on the smaller partitioned slices, instead of handling the giant database altogether. This is how partitioning cuts down the problems in managing large database tables.
2. **Partitioning** is the technique used to divide stored database objects into separate servers. Due to this, there is an increase in performance, controllability of the data. We can manage huge chunks of data optimally. When we horizontally scale our machines/servers, we know that it gives us a challenging time dealing with relational databases as it's quite tough to maintain the relations. But if we apply partitioning to the database that is already scaled out i.e. equipped with multiple servers, we can partition our database among those servers and handle the big data easily.
3. **Vertical Partitioning**
    1. Slicing relation vertically / column-wise.
    2. Need to access different servers to get complete tuples.
4. **Horizontal Partitioning**
    1. Slicing relation horizontally / row-wise.
    2. Independent chunks of data tuples are stored in different servers.
5. **When Partitioning is Applied?**
    1. Dataset become much huge that managing and dealing with it become a tedious task.
    2. The number of requests are enough larger that the single DB server access is taking huge time and hence the system's response time become high.
6. **Advantages of Partitioning**
    1. Parallelism
    2. Availability
    3. Performance
    4. Manageability
    5. Reduce Cost, as scaling-up or vertical scaling might be costly.
7. **Distributed Database**
    1. A single logical database that is, spread across multiple locations (servers) and logically interconnected by network.
    2. This is the product of applying DB optimisation techniques like **Clustering, Partitioning and Sharding.**
    3. Why this is needed? READ Point **5**.
8. **Sharding**
    1. Technique to implement Horizontal Partitioning.
    2. The **fundamental idea** of Sharding is the idea that instead of having all the data sit on one DB instance, we split it up and introduce a Routing layer so that we can forward the request to the right instances that actually contain the data.
    3. **Pros**
        1. Scalability
        2. Availability
    4. **Cons**
        1. Complexity, making partition mapping, Routing layer to be implemented in the system, Non-uniformity that creates the necessity of Re-Sharding
        2. Not well suited for Analytical type of queries, as the data is spread across different DB instances. (Scatter-Gather problem)

UNIT 1
AN INTRODUCTION TO OPERATING SYSTEMS

Application software performs specific task for the user.
System software operates and controls the computer system and provides a platform to run application software.

An **operating system** is a piece of software that manages all the resources of a computer system,both hardware and software, and provides an environment in which the user can execute his/her programs in a convenient and efficient manner by hiding underlying complexity of the hardware and acting as a resource manager.

Why OS?

1. What if there is no OS?
    a. Bulky and complex app. (Hardware interaction code must be in app's code base)
    b. Resource exploitation by 1 App.
    c. No memory protection.
2. What is an OS made up of?
    a. Collection of system software.

An operating system function -

- Access to the computer hardware.
- interface between the user and the computer hardware
- **Resource management (Aka, Arbitration) (memory, device, file, security, process etc)**
- **Hides the underlying complexity of the hardware. (Aka, Abstraction)**
- facilitates execution of application programs by providing isolation and protection.

User

Application programs
Operating system
Computer hardware

The operating system provides the means for proper use of the resources in the operation of the computer system.

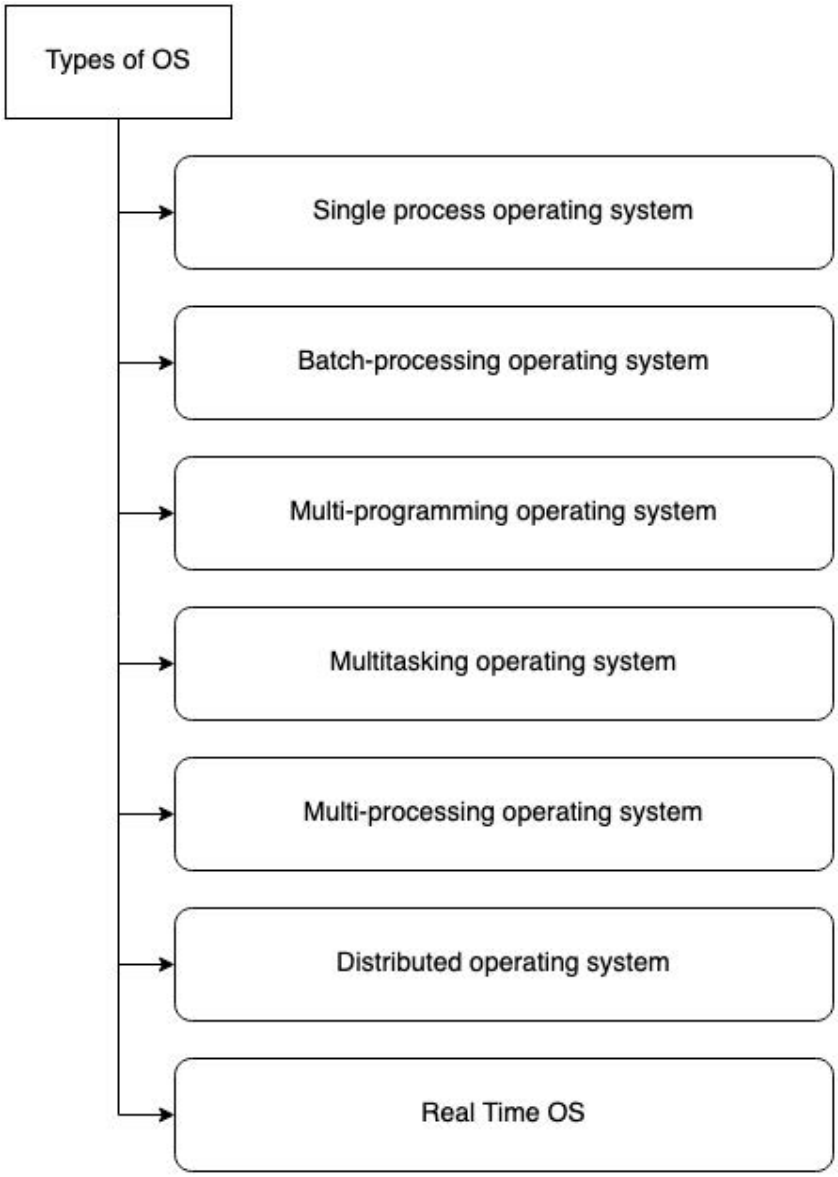
LEC-2: Types of OS

OS goals –

- Maximum CPU utilization
- Less process starvation
- Higher priority job execution

Types of operating systems –

- Single process operating system [MS DOS, 1981]
- Batch-processing operating system [ATLAS, Manchester Univ., late 1950s – early 1960s]
- Multiprogramming operating system [THE, Dijkstra, early 1960s]
- Multitasking operating system [CTSS, MIT, early 1960s]
- Multi-processing operating system [Windows NT]
- Distributed system [LOCUS]
- Real time OS [ATCS]

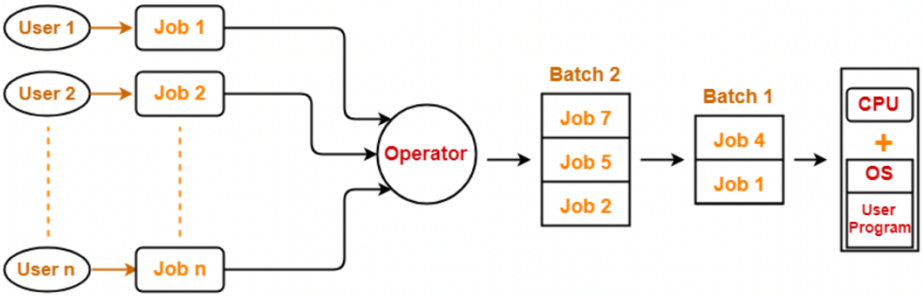


Single process OS, only 1 process executes at a time from the ready queue. [Oldest]

Batch-processing OS,

1. Firstly, user prepares his job using punch cards.
2. Then, he submits the job to the computer operator.
3. Operator collects the jobs from different users and sort the jobs into batches with similar needs.
4. Then, operator submits the batches to the processor one by one.
5. All the jobs of one batch are executed together.

- Priorities cannot be set, if a job comes with some higher priority.
- May lead to starvation. (A batch may take more time to complete)
- CPU may become idle in case of I/O operations.



Multiprogramming increases CPU utilization by keeping multiple jobs (code and data) in the**memory** so that the CPU always has one to execute in case some job gets busy with I/O.

- Single CPU
- Context switching for processes.
- Switch happens when current process goes to wait state.
- CPU idle time reduced.

Multitasking is a logical extension of multiprogramming.

- Single CPU
- Able to run more than one task simultaneously.
- Context switching and time sharing used.
- Increases responsiveness.
- CPU idle time is further reduced.

Multi-processing OS, more than 1 CPU in a single computer.

- Increases reliability, 1 CPU fails, other can work
- Better throughput.
- Lesser process starvation, (if 1 CPU is working on some process, other can be executed on other CPU.

Distributed OS,

- OS manages many bunches of resources, >=1 CPUs, >=1 memory, >=1 GPUs, etc
- **Loosely connected autonomous,** interconnected computer nodes.
- collection of independent, networked, communicating, and physically separate computational nodes.

RTOS

- **Real time** error free, computations within tight-time boundaries.
- Air Traffic control system, ROBOTS etc.

LEC-3: Multi-Tasking vs Multi-Threading

Program: A Program is an executable file which contains a certain set of instructions written to complete the specific job or operation on your computer.

- It's a compiled code. Ready to be executed.
- Stored in Disk

Process: Program under execution. Resides in Computer's primary memory (RAM).

Thread:

- Single sequence stream within a process.
- An independent path of execution in a process.
- Light-weight process.
- Used to achieve parallelism by dividing a process's tasks which are independent path of execution.
- E.g., Multiple tabs in a browser, text editor (When you are typing in an editor, spell-checking, formatting of text and saving the text are done concurrently by multiple threads.)

Multi-Tasking	**Multi-Threading**
The execution of more than one task simultaneously is called as multitasking.	A process is divided into several different sub-tasks called as threads, which has its own path of execution. This concept is called as multithreading.
Concept of more than 1 processes being context switched.	Concept of more than 1 thread. Threads are context switched.
No. of CPU 1.	No. of CPU >= 1. (Better to have more than 1)
Isolation and memory protection exists. OS must allocate separate memory and resources to each program that CPU is executing.	**No isolation and memory protection**, resources are shared among threads of that process. OS allocates memory to a process; multiple threads of that process share the same memory and resources allocated to the process.

Thread Scheduling:
Threads are scheduled for execution based on their priority. Even though threads are executing within the runtime, all threads are assigned processor time slices by the operating system.

Difference between Thread Context Switching and Process Context Switching:

Thread Context switching	Process context switching
OS saves current state of thread & switches to another thread of same process.	OS saves current state of process & switches to another process by restoring its state.

Doesn't includes switching of memory address space. (But Program counter, registers & stack are included.)	Includes switching of memory address space.
Fast switching.	Slow switching.
CPU's cache state is preserved.	CPU's cache state is flushed.

LEC-4: Components of OS

1. **Kernel:** A **kernel** is that part of the operating system which interacts directly with the hardware andperforms the most crucial tasks.
    a. Heart of OS/Core component
    b. Very first part of OS to load on start-up.
2. **User space:** Where application software runs, apps don't have privileged access to the underlying hardware. It interacts with kernel.
    a. GUI
    b. CLI

A **shell**, also known as a command interpreter, is that part of the operating system that receives commands from the users and gets them executed.

Functions of Kernel:

1. **Process management:**
    a. Scheduling processes and threads on the CPUs.
    b. Creating & deleting both user and system process.
    c. Suspending and resuming processes
    d. Providing mechanisms for process synchronization or process communication.
2. **Memory management:**
    a. Allocating and deallocating memory space as per need.
    b. Keeping track of which part of memory are currently being used and by which process.
3. **File management:**
    a. Creating and deleting files.
    b. Creating and deleting directories to organize files.
    c. Mapping files into secondary storage.
    d. Backup support onto a stable storage media.
4. **I/O management:** to manage and control I/O operations and I/O devices
    a. Buffering (data copy between two devices), caching and spooling.
        i. Spooling
            1. Within differing speed two jobs.
            2. Eg. Print spooling and mail spooling.
        ii. Buffering
            1. Within one job.
            2. Eg. Youtube video buffering
        iii. Caching
            1. Memory caching, Web caching etc.

Types of Kernels:

1. Monolithic kernel
    a. All functions are in kernel itself.
    b. Bulky in size.
    c. Memory required to run is high.
    d. Less reliable, one module crashes -> whole kernel is down.
    e. High performance as communication is fast. (Less user mode, kernel mode overheads)
    f. Eg. Linux, Unix, MS-DOS.

2. Micro Kernel
    a. Only major functions are in kernel.
        i. Memory mgmt.
        ii. Process mgmt.
    b. File mgmt. and IO mgmt. are in User-space.
    c. smaller in size.
    d. More Reliable
    e. More stable
    f. Performance is slow.
    g. Overhead switching b/w user mode and kernel mode.
    h. Eg. L4 Linux, Symbian OS, MINIX etc.

3. Hybrid Kernel:
    a. Advantages of both worlds. (File mgmt. in User space and rest in Kernel space.)
    b. Combined approach.
    c. Speed and design of mono.
    d. Modularity and stability of micro.
    e. Eg. MacOS, Windows NT/7/10
    f. IPC also happens but lesser overheads
4. Nano/Exo kernels…

Q. How will communication happen between user mode and kernel mode?
Ans. Inter process communication (**IPC**).

1. Two processes executing independently, having independent memory space (Memory protection), But some may need to communicate to work.

2. Done by shared memory and message passing.

LEC-5: System Calls

How do apps interact with Kernel? -> using system calls.

Eg. Mkdir laks

- Mkdir indirectly calls kernel and asked the file mgmt. module to create a new directory.
- Mkdir is just a wrapper of actual system calls.
- Mkdir interacts with kernel using system calls.

Eg. Creating a process.

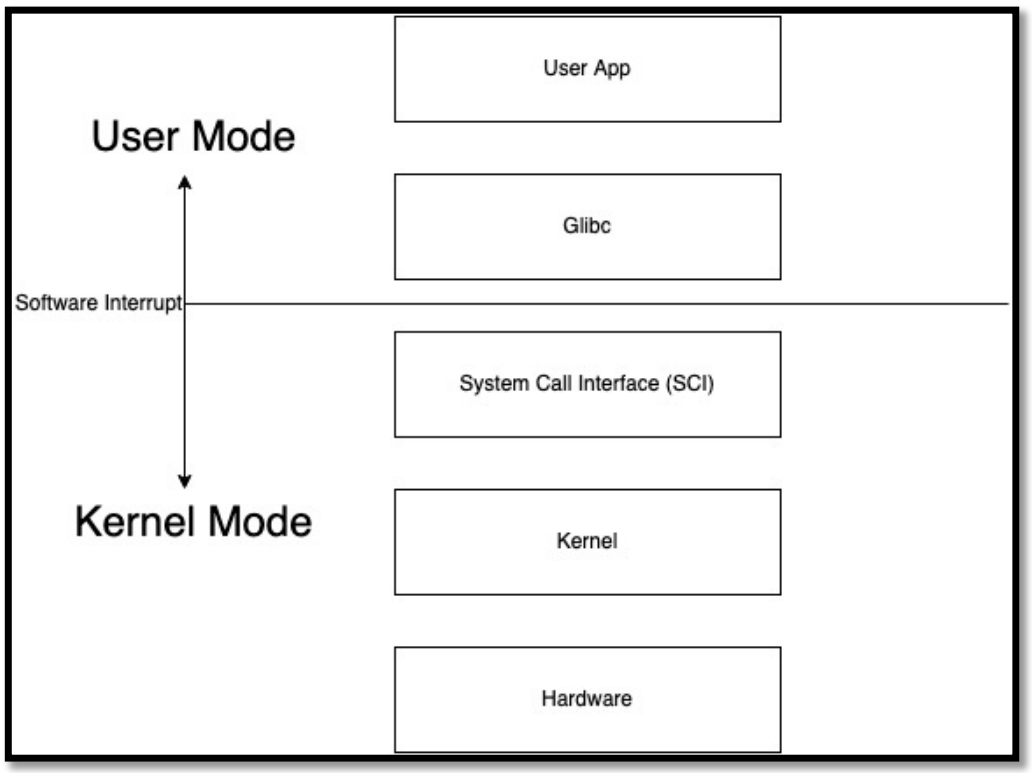
- User executes a process. (User space)
- Gets system call. (US)
- Exec system call to create a process. (KS)
- Return to US.

Transitions from US to KS done by software interrupts.

System calls are implemented in C.

A **system call** is a mechanism using which a user program can request a service from the kernel for which it does not have the permission to perform.
User programs typically do not have permission to perform operations like accessing I/O devices and communicating other programs.

System Calls are the only way through which a process can go into **kernel mode from user mode.**



Types of System Calls:

1) Process Control
   a. end, abort
   b. load, execute
   c. create process, terminate process
   d. get process attributes, set process attributes
   e. wait for time
   f. wait event, signal event
   g. allocate and free memory

2) File Management
   a. create file, delete file
   b. open, close
   c. read, write, reposition
   d. get file attributes, set file attributes

3) Device Management
   a. request device, release device
   b. read, write, reposition
   c. get device attributes, set device attributes
   d. logically attach or detach devices

4) Information maintenance
   a. get time or date, set time or date
   b. get system data, set system data
   c. get process, file, or device attributes
   d. set process, file, or device attributes

5) Communication Management
   a. create, delete communication connection
   b. send, receive messages
   c. transfer status information
   d. attach or detach remote devices

Examples of Windows & Unix System calls:

Category	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Management	CreateFile() ReadFile() WriteFile() CloseHandle() SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	open () read () write () close () chmod() umask(chown()
Device Management	SetConsoleMode() ReadConsole() WriteConsole()	ioctI() read() write()
Information Management	GetCurrentProcessID() SetTimer() Sleep()	getpid () alarm () sleep ()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe () shmget () mmap()

LEC-6: What happens when you turn on your computer?

i. PC On
ii. CPU initializes itself and looks for a firmware program (BIOS) stored in BIOS Chip **(**Basic input-output system chip is a ROM chip found on mother board that allows to access & setup computer system at most basic level.)
    1. In modern PCs, CPU loads UEFI (Unified extensible firmware interface)
iii. **CPU** runs the BIOS which tests and initializes system hardware. Bios loads configuration settings. If something is not appropriate (like missing RAM) error is thrown and boot process is stopped.
    This is called **POST** (Power on self-test) process.
    (UEFI can do a lot more than just initialize hardware; it's really a tiny operating system. For example, Intel CPUs have the Intel Management Engine. This provides a variety of features, including powering Intel's Active Management Technology, which allows for remote management of business PCs.)
iv. **BIOS** will handoff responsibility for booting your PC to your OS's bootloader.

    1. BIOS looked at the MBR (master boot record), a special boot sector at the beginning of a disk. The MBR contains code that loads the rest of the operating system, known as a "bootloader." The BIOS executes the bootloader, which takes it from there and begins booting the actual operating system—Windows or Linux, for example.
    In other words,
    the BIOS or UEFI examines a storage device on your system to look for a small program, either in the MBR or on an EFI system partition, and runs it.

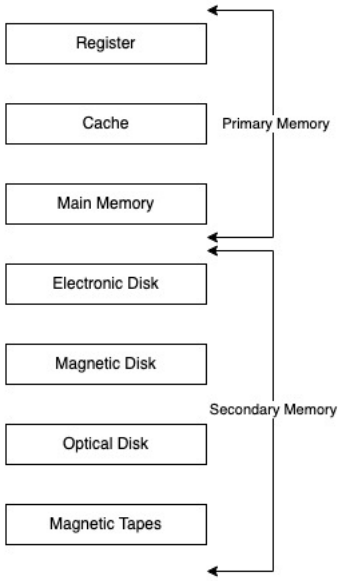
v. The bootloader is a small program that has the large task of booting the rest of the operating system (Boots Kernel then, User Space). Windows uses a bootloader named Windows Boot Manager (Bootmgr.exe), most Linux systems use GRUB, and Macs use something called boot.efi

Lec-7: 32-Bit vs 64-Bit OS

1. A 32-bit OS has 32-bit registers, and it can access 2^32 unique memory addresses. i.e., 4GB of physical memory.
2. A 64-bit OS has 64-bit registers, and it can access 2^64 unique memory addresses. i.e., 17,179,869,184 GB of physical memory.
3. 32-bit CPU architecture can process 32 bits of data & information.
4. 64-bit CPU architecture can process 64 bits of data & information.
5. Advantages of 64-bit over the 32-bit operating system:
    a. **Addressable Memory**: 32-bit CPU -> 2^32 memory addresses, 64-bit CPU -> 2^64 memory addresses.
    b. **Resource usage**: Installing more RAM on a system with a 32-bit OS doesn't impact performance. However, upgrade that system with excess RAM to the 64-bit version of Windows, and you'll notice a difference.
    c. **Performance**: All calculations take place in the registers. When you're performing math in your code, operands are loaded from memory into registers. So, having larger registers allow you to perform larger calculations at the same time.
    32-bit processor can execute 4 bytes of data in 1 instruction cycle while 64-bit means that processor can execute 8 bytes of data in 1 instruction cycle.
    (In 1 sec, there could be thousands to billons of instruction cycles depending upon a processor design)
    d. **Compatibility**: 64-bit CPU can run both 32-bit and 64-bit OS. While 32-bit CPU can only run 32-bit OS.
    e. **Better Graphics performance**: 8-bytes graphics calculations make graphics-intensive apps run faster.

Lec-8: Storage Devices Basics

What are the different memory present in the computer system?



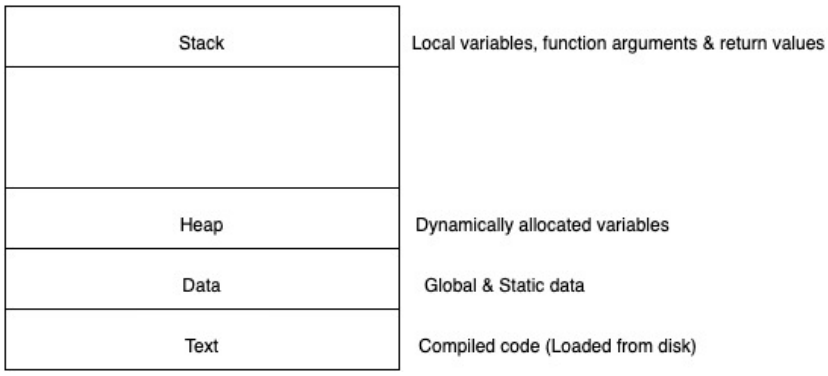
1. **Register**: Smallest unit of storage. It is a part of CPU itself.
   A register may hold an instruction, a storage address, or any data (such as bit sequence or individual characters).
   Registers are a type of computer memory used to quickly accept, store, and transfer data and instructions that are being used immediately by the CPU.
2. **Cache**: Additional memory system that temporarily stores frequently used instructions and data for quicker processing by the CPU.
3. **Main Memory**: RAM.
4. **Secondary Memory**: Storage media, on which computer can store data & programs.

Comparison

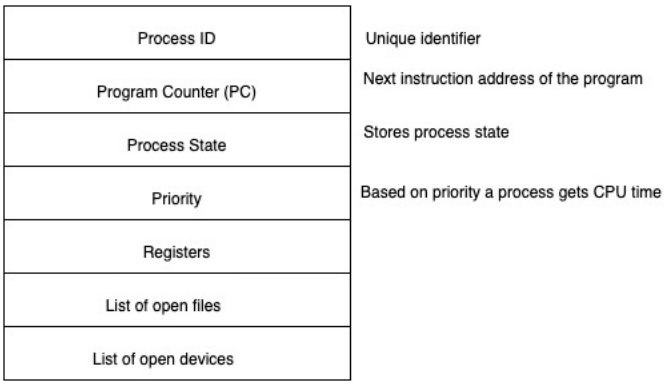
1. **Cost**:
   a. Primary storages are costly.
   b. Registers are most expensive due to expensive semiconductors & labour.
   c. Secondary storages are cheaper than primary.
2. **Access Speed**:
   a. Primary has higher access speed than secondary memory.
   b. Registers has highest access speed, then comes cache, then main memory.
3. **Storage size**:
   a. Secondary has more space.
4. **Volatility**:
   a. Primary memory is volatile.
   b. Secondary is non-volatile.

Lec-9: Introduction to Process

1. What is a program? Compiled code, that is ready to execute.
2. What is a process? Program under execution.
3. How OS creates a process? Converting program into a process.
   STEPS:
   a. Load the program & static data into memory.
   b. Allocate runtime stack.
   c. Heap memory allocation.
   d. IO tasks.
   e. OS handoffs control to main ().
4. **Architecture** of process:



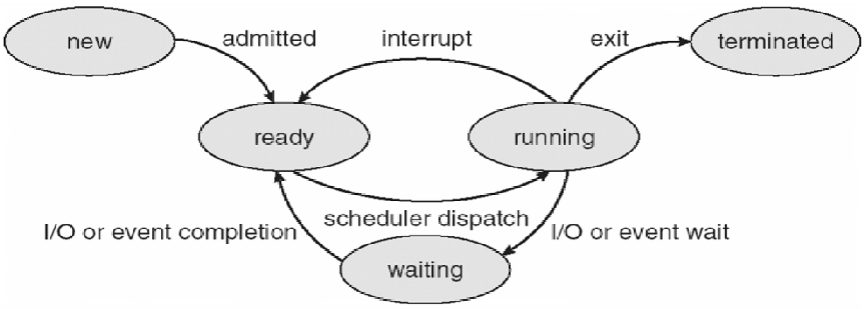
5. **Attributes** of process:
   a. Feature that allows identifying a process uniquely.
   b. Process table
      i. All processes are being tracked by OS using a table like data structure.
      ii. Each entry in that table is process control block (PCB).
   c. PCB: Stores info/attributes of a process.
      i. Data structure used for each process, that stores information of a process such as process id, program counter, process state, priority etc.
6. **PCB structure**:



Registers in the PCB, it is a data structure. When a processes is running and it's time slice expires, the current value of process specific registers would be stored in the PCB and the process would be swapped out. When the process is scheduled to be run, the register values is read from the PCB and written to the CPU registers. This is the main purpose of the registers in the PCB.

Lec-10: Process States | Process Queues

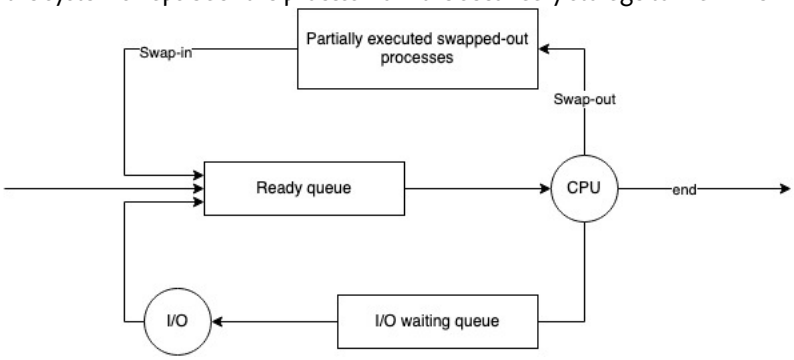
1. **Process States**: As process executes, it changes state. Each process may be in one of the following states.
    a. **New**: OS is about to pick the program & convert it into process. OR the process is being created.
    b. **Run**: Instructions are being executed; CPU is allocated.
    c. **Waiting**: Waiting for IO.
    d. **Ready**: The process is in memory, waiting to be assigned to a processor.
    e. **Terminated**: The process has finished execution. PCB entry removed from process table.



2. **Process Queues:**
    a. Job Queue:
        i. Processes in new state.
        ii. Present in secondary memory.
        iii. **Job Schedular (Long term schedular (LTS))** picks process from the pool and loads them into memory for execution.
    b. Ready Queue:
        i. Processes in Ready state.
        ii. Present in main memory.
        iii. **CPU Schedular (Short-term schedular)** picks process from ready queue and dispatch it to CPU.
    c. Waiting Queue:
        i. Processes in Wait state.
3. **Degree of multi-programming**: The number of processes in the memory.
    a. **LTS** controls degree of multi-programming.
4. **Dispatcher:** The module of OS that gives control of CPU to a process selected by **STS**.

LEC-11: Swapping | Context-Switching | Orphan process | Zombie process

1. **Swapping**
   - **a.** Time-sharing system may have medium term schedular (MTS).
   - **b.** Remove processes from memory to reduce degree of multi-programming.
   - **c.** These removed processes can be reintroduced into memory, and its execution can be continued where it left off. This is called **Swapping.**
   - **d.** Swap-out and swap-in is done by MTS.
   - **e.** Swapping is necessary to improve process mix or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up.
   - f. Swapping is a mechanism in which a process can be swapped temporarily out of main memory (or move) to secondary storage (disk) and make that memory available to other processes. At some later time, the system swaps back the process from the secondary storage to main memory.



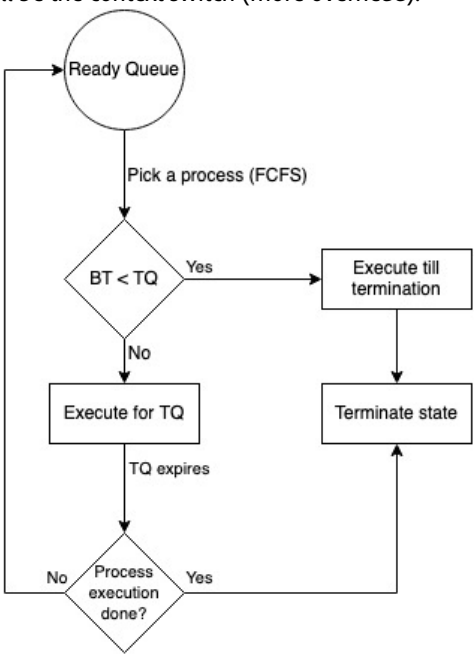
2. **Context-Switching**
   - a. Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process.
   - b. When this occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.
   - c. It is pure overhead, because the system does no useful work while switching.
   - d. Speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied etc.
3. **Orphan process**
   - **a.** The process whose parent process has been terminated and it is still running.
   - **b.** Orphan processes are adopted by init process.
   - **c.** Init is the first process of OS.
4. **Zombie process / Defunct process**
   - a. A zombie process is a process whose execution is completed but it still has an entry in the process table.
   - b. Zombie processes usually occur for child processes, as the parent process still needs to read its child's exit status. Once this is done using the wait system call, the zombie process is eliminated from the process table. This is known as **reaping** the zombie process.
   - c. It is because parent process may call wait () on child process for a longer time duration and child process got terminated much earlier.
   - d. As entry in the process table can only be removed, after the parent process reads the exit status of child process. Hence, the child process remains a zombie till it is removed from the process table.

LEC-12: Intro to Process Scheduling | FCFS | Convoy Effect

1. **Process Scheduling**
    a. Basis of Multi-programming OS.
    b. By switching the CPU among processes, the OS can make the computer more productive.
    c. Many processes are kept in memory at a time, when a process must wait or time quantum expires, the OS takes the CPU away from that process & gives the CPU to another process & this pattern continues.
2. **CPU Scheduler**
    a. Whenever the CPU become ideal, OS must select one process from the ready queue to be executed.
    b. Done by STS.
3. **Non-Preemptive scheduling**
    a. Once CPU has been allocated to a process, the process keeps the CPU until it releases CPU either by terminating or by switching to wait-state.
    b. Starvation, as a process with long burst time may starve less burst time process.
    c. Low CPU utilization.
4. **Preemptive scheduling**
    a. CPU is taken away from a process after time quantum expires along with terminating or switching to wait-state.
    b. Less Starvation
    c. High CPU utilization.
5. **Goals of CPU scheduling**
    a. Maximum CPU utilization
    b. Minimum Turnaround time (TAT).
    c. Min. Wait-time
    d. Min. response time.
    e. Max. throughput of system.
6. **Throughput**: No. of processes completed per unit time.
7. **Arrival time (AT)**: Time when process is arrived at the ready queue.
8. **Burst** time (BT): The time required by the process for its execution.
9. **Turnaround** time (TAT): Time taken from first time process enters ready state till it terminates. (CT - AT)
10. **Wait** time (WT): Time process spends waiting for CPU. (WT = TAT – BT)
11. **Response** time: Time duration between process getting into ready queue and process getting CPU for the first time.
12. **Completion** Time (CT): Time taken till process gets terminated.
13. **FCFS** (First come-first serve):
    a. Whichever process comes first in the ready queue will be given CPU first.
    b. In this, if one process has longer BT. It will have major effect on average WT of diff processes, called **Convoy effect.**
    c. Convoy Effect is a situation where many processes, who need to use a resource for a short time, are blocked by one process holding that resource for a long time.
        i. This cause poor resource management.

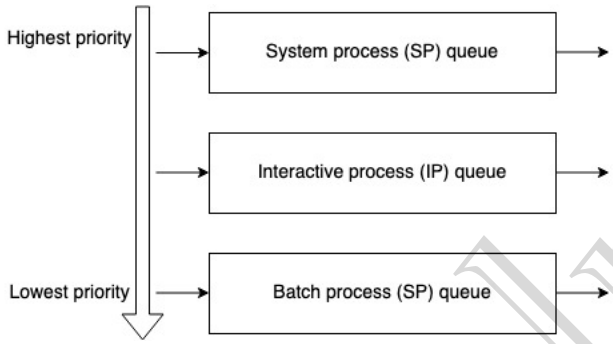
LEC-13: CPU Scheduling | SJF | Priority | RR

1. Shortest Job First (SJF) [Non-preemptive]
    a. Process with least BT will be dispatched to CPU first.
    b. Must do estimation for BT for each process in ready queue beforehand, Correct estimation of BT is an impossible task (ideally.)
    c. Run lowest time process for all time then, choose job having lowest BT at that instance.
    d. This will suffer from convoy effect as if the very first process which came is Ready state is having a large BT.
    e. Process starvation might happen.
    f. Criteria for SJF algos, AT + BT.
2. SJF [Preemptive]
    a. Less starvation.
    b. No convoy effect.
    c. Gives average WT less for a given set of processes as scheduling short job before a long one decreases the WT of short job more than it increases the WT of the long process.
3. Priority Scheduling [Non-preemptive]
    a. Priority is assigned to a process when it is created.
    b. SJF is a special case of general priority scheduling with priority inversely proportional to BT.
4. Priority Scheduling [Preemptive]
    a. Current RUN state job will be preempted if next job has higher priority.
    b. May cause indefinite waiting (Starvation) for lower priority jobs. (Possibility is they won't get executed ever). (True for both preemptive and non-preemptive version)
        i. Solution: Ageing is the solution.
        ii. Gradually increase priority of process that wait so long. E.g., increase priority by 1 every 15 minutes.
5. Round robin scheduling (RR)
    a. Most popular
    b. Like FCFS but preemptive.
    c. Designed for time sharing systems.
    d. Criteria: AT + time quantum (TQ), Doesn't depend on BT.
    e. No process is going to wait forever, hence very low starvation. [No convoy effect]
    f. Easy to implement.
    g. If TQ is small, more will be the context switch (more overhead).



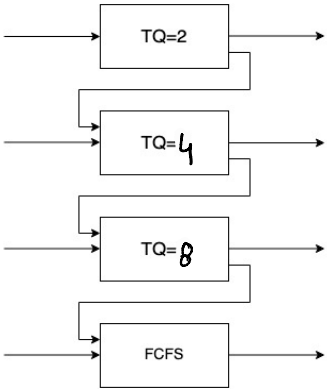
LEC-14: MLQ | MLFQ

1. Multi-level queue scheduling (**MLQ**)
    a. Ready queue is divided into multiple queues depending upon priority.
    b. A process is permanently assigned to one of the queues (inflexible) based on some property of process, memory, size, process priority or process type.
    c. Each queue has its own scheduling algorithm. E.g., SP -> RR, IP -> RR & BP -> FCFS.



    d. System process: Created by OS (Highest priority)
    Interactive process (Foreground process): Needs user input (I/O).
    Batch process (Background process): Runs silently, no user input required.
    e. Scheduling among different sub-queues is implemented as **fixed priority preemptive** scheduling. E.g., foreground queue has absolute priority over background queue.
    f. If an interactive process comes & batch process is currently executing. Then, batch process will be preempted.
    g. Problem: Only after completion of all the processes from the top-level ready queue, the further level ready queues will be scheduled.
    This came starvation for lower priority process.
    h. Convoy effect is present.

2. Multi-level feedback queue scheduling (**MLFQ**)
    a. Multiple sub-queues are present.
    b. Allows the process to move between queues. The idea is to separate processes according to the characteristics of their BT. If a process uses too much CPU time, it will be moved to lower priority queue. This scheme leaves I/O bound and interactive processes in the higher-priority queue.
    In addition, a process that waits too much in a lower-priority queue may be moved to a higher priority queue. This form of ageing prevents starvation.
    c. Less starvation then MLQ.
    d. It is flexible.
    e. Can be configured to match a specific system design requirement.
    Sample MLFQ design:



3. Comparison:

	FCFS	SJF	PSJF	Priority	P-Priority	RR	MLQ	MLFQ
Design	Simple	Complex	Complex	Complex	Complex	Simple	Complex	Complex
Preemption	No	No	Yes	No	Yes	Yes	Yes	Yes
Convoy effect	Yes	Yes	No	Yes	Yes	No	Yes	Yes
Overhead	No	No	Yes	No	Yes	Yes	Yes	Yes

LEC-15: Introduction to Concurrency

1. **Concurrency** is the execution of the multiple instruction sequences at the same time. It happens in the operating system when there are several process threads running in parallel.
2. **Thread**:
    - Single sequence stream within a process.
    - An independent path of execution in a process.
    - Light-weight process.
    - Used to achieve parallelism by dividing a process's tasks which are independent path of execution.
    - E.g., Multiple tabs in a browser, text editor (When you are typing in an editor, spell checking, formatting of text and saving the text are done concurrently by multiple threads.)
3. **Thread Scheduling:** Threads are scheduled for execution based on their priority. Even though threads are executing within the runtime, all threads are assigned processor time slices by the operating system.
4. **Threads context switching**
    - OS saves current state of thread & switches to another thread of same process.
    - Doesn't includes switching of memory address space. (But Program counter, registers & stack are included.)
    - Fast switching as compared to process switching
    - CPU's cache state is preserved.
5. **How each thread get access to the CPU?**
    - Each thread has its own program counter.
    - Depending upon the thread scheduling algorithm, OS schedule these threads.
    - OS will fetch instructions corresponding to PC of that thread and execute instruction.
6. **I/O or TQ, based context switching is done here as well**
    - We have TCB (Thread control block) like PCB for state storage management while performing context switching.
7. **Will single CPU system would gain by multi-threading technique?**
    - Never.
    - As two threads have to context switch for that single CPU.
    - This won't give any gain.
8. **Benefits of Multi-threading.**
    - Responsiveness
    - Resource sharing: Efficient resource sharing.
    - Economy: It is more economical to create and context switch threads.
        1. Also, allocating memory and resources for process creation is costly, so better to divide tasks into threads of same process.
    - Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

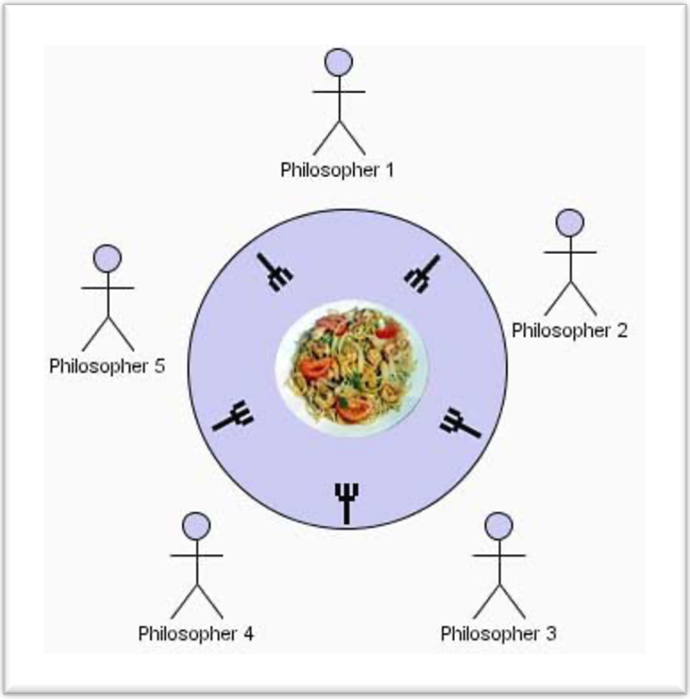
LEC-16: Critical Section Problem and How to address it

1. Process synchronization techniques play a key role in maintaining the consistency of shared data
2. **Critical Section (C.S)**
    a. The critical section refers to the segment of code where processes/threads access shared resources, such as common variables and files, and perform write operations on them. Since processes/threads execute concurrently, any process can be interrupted mid-execution.
3. **Major Thread scheduling issue**
    a. Race Condition
        i. A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e., both threads are "racing" to access/change the data.
4. **Solution to Race Condition**
    a. Atomic operations: Make Critical code section an atomic operation, i.e., Executed in one CPU cycle.
    b. Mutual Exclusion using locks.
    c. Semaphores
5. Can we use a simple flag variable to solve the problem of race condition?
    a. No.
6. **Peterson's solution** can be used to avoid race condition but holds good for only 2 process/threads.
7. **Mutex/Locks**
    a. Locks can be used to implement mutual exclusion and avoid race condition by allowing only one thread/process to access critical section.
    b. Disadvantages:
        i. Contention: one thread has acquired the lock, other threads will be busy waiting, what if thread that had acquired the lock dies, then all other threads will be in infinite waiting.
        ii. Deadlocks
        iii. Debugging
        iv. Starvation of high priority threads.

LEC-17: Conditional Variable and Semaphores for Threads synchronization

1. **Conditional variable**
    a. The condition variable is a synchronization primitive that lets the thread wait until a certain condition occurs.
    b. Works with a lock
    c. Thread can enter a wait state only when it has acquired a lock. When a thread enters the wait state, it will release the lock and wait until another thread notifies that the event has occurred. Once the waiting thread enters the running state, it again acquires the lock immediately and starts executing.
    d. Why to use conditional variable?
        i. To avoid busy waiting.
    e. Contention is not here.
2. **Semaphores**
    a. Synchronization method.
    b. An integer that is equal to number of resources
    c. Multiple threads can go and execute C.S concurrently.
    d. Allows multiple program threads to access the finite instance of resources whereas mutex allows multiple threads to access a single shared resource one at a time.
    e. Binary semaphore: value can be 0 or 1.
        i. Aka, mutex locks
    f. Counting semaphore
        i. Can range over an unrestricted domain.
        ii. Can be used to control access to a given resource consisting of a finite number of instances.
    g. To overcome the need for busy waiting, we can modify the definition of the wait () and signal () semaphore operations. When a process executes the wait () operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process car block itself. The block- operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the Waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.
    h. A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal () operation. The process is restarted by a wakeup () operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue.

Lec-20: The Dining Philosophers problem



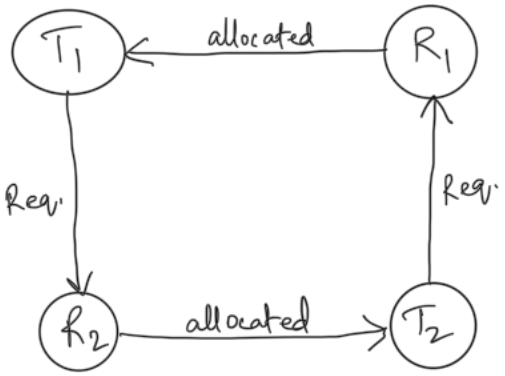
1. We have **5 philosophers.**
2. They spend their life just being in **two states:**
    a. Thinking
    b. Eating
3. They sit on a circular table surrounded by 5 chairs (1 each), in the center of table is a bowl of noodles, and the table is laid with 5 single forks.
4. **Thinking state**: When a ph. Thinks, he doesn't interact with others.
5. **Eating state:** When a ph. Gets hungry, he tries to pick up the 2 forks adjacent to him (Left and Right). He can pick one fork at a time.
6. One can't pick up a fork if it is already taken.
7. When ph. Has both forks at the same time, he eats without releasing forks.
8. Solution can be given using semaphores.
    a. Each fork is a binary semaphore.
    b. A ph. Calls wait() operation to acquire a fork.
    c. Release fork by calling signal().
    d. **Semaphore fork[5]{1};**
9. Although the semaphore solution makes sure that no two neighbors are eating simultaneously but it could still create **Deadlock**.
10. Suppose that all 5 ph. Become hungry at the same time and each picks up their left fork, then All fork semaphores would be 0.
11. When each ph. Tries to grab his right fork, he will be waiting for ever (Deadlock)
12. We must use **some methods to avoid Deadlock and make the solution work**
    a. Allow at most 4 ph. To be sitting simultaneously.
    b. Allow a ph. To pick up his fork only if both forks are available and to do this, he must pick them up in a critical section (atomically).

c. **Odd-even rule.**
an odd ph. Picks up first his left fork and then his right fork, whereas an even ph. Picks up his right fork then his left fork.

13. Hence, only semaphores are not enough to solve this problem.
We must add some enhancement rules to make deadlock free solution.

LEC-21: Deadlock Part-1

1. In Multi-programming environment, we have several processes competing for finite number of resources
2. Process requests a **resource (R)**, if R is not available (taken by other process), process enters in a waiting state. Sometimes that waiting process is never able to change its state because the resource, it has requested is busy (forever), called **DEADLOCK (DL)**
3. Two or more processes are waiting on some resource's availability, which will never be available as it is also busy with some other process. The Processes are said to be in **Deadlock.**
4. DL is a bug present in the process/thread synchronization method.
5. In DL, processes never finish executing, and the system resources are tied up, preventing other jobs from starting.
6. **Example of resources**: Memory space, CPU cycles, files, locks, sockets, IO devices etc.
7. Single resource can have multiple instances of that. E.g., CPU is a resource, and a system can have 2 CPUs.
8. How a process/thread utilize a resource?
    a. Request: Request the R, if R is free Lock it, else wait till it is available.
    b. Use
    c. Release: Release resource instance and make it available for other processes



9. **Deadlock Necessary Condition**: **4** Condition should hold simultaneously.
    a. Mutual Exclusion
        i. Only 1 process at a time can use the resource, if another process requests that resource, the requesting process must wait until the resource has been released.
    b. Hold & Wait
        i. A process must be holding at least one resource & waiting to acquire additional resources that are currently being held by other processes.
    c. No-preemption
        i. Resource must be voluntarily released by the process after completion of execution. (No resource preemption)
    d. Circular wait
        i. A set {P0, P1, … ,Pn} of waiting processes must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, and so on.
10. **Methods for handling Deadlocks:**
    a. Use a protocol to **prevent** or **avoid** deadlocks, ensuring that the system will never enter a deadlocked state.
    b. Allow the system to enter a deadlocked state, **detect it, and recover.**

c. Ignore the problem altogether and pretend that deadlocks never occur in system. (Ostrich algorithm) aka, **Deadlock ignorance.**

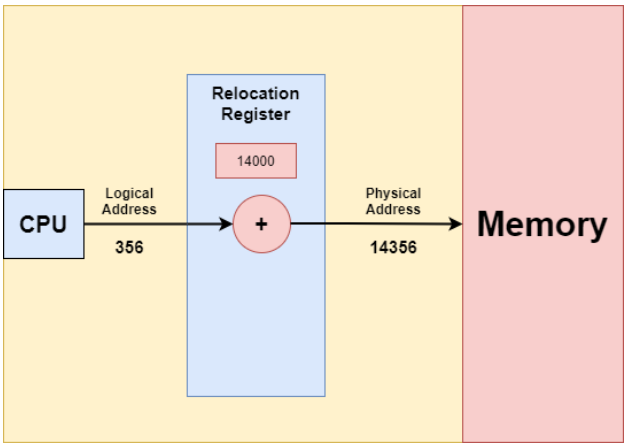
11. To ensure that deadlocks never occur, the system can use either a **deadlock prevention or deadlock avoidance scheme**.
12. **Deadlock Prevention**: by ensuring at least one of the necessary conditions cannot hold.
    a. **Mutual exclusion**
        i. Use locks (mutual exclusion) only for non-sharable resource.
        ii. Sharable resources like Read-Only files can be accessed by multiple processes/threads.
        iii. However, we can't prevent DLs by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable.
    b. **Hold & Wait**
        i. To ensure H&W condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it doesn't hold any other resource.
        ii. Protocol (A) can be, each process has to request and be allocated all its resources before its execution.
        iii. Protocol (B) can be, allow a process to request resources only when it has none. It can request any additional resources after it must have released all the resources that it is currently allocated.
    c. **No preemption**
        i. If a process is holding some resources and request another resource that cannot be immediately allocated to it, then all the resources the process is currently holding are preempted. The process will restart only when it can regain its old resources, as well as the new one that it is requesting. (Live Lock may occur).
        ii. If a process requests some resources, we first check whether they are available. If yes, we allocate them. If not, we check whether they are allocated to some other process that is waiting for additional resources. If so, preempt the desired resource from waiting process and allocate them to the requesting process.
    d. **Circular wait**
        i. To ensure that this condition never holds is to impose a proper ordering of resource allocation.
        ii. P1 and P2 both require R1 and R1, locking on these resources should be like, both try to lock R1 then R2. By this way which ever process first locks R1 will get R2.

LEC-22: Deadlock Part-2

1. **Deadlock Avoidance**: Idea is, the kernel be given in advance info concerning which resources will use in its lifetime.
   By this, system can decide for each request whether the process should wait.
   To decide whether the current request can be satisfied or delayed, the system must consider the resources currently available, resources currently allocated to each process in the system and the future requests and releases of each process.
   a. Schedule process and its resources allocation in such a way that the DL never occur.
   b. Safe state: A state is safe if the system can allocate resources to each process (up to its max.) in some order and still avoid DL.
      A system is in safe state only if there exists a safe sequence.
   c. In an Unsafe state, the operating system cannot prevent processes from requesting resources in such a way that any deadlock occurs. It is not necessary that all unsafe states are deadlocks; an unsafe state may lead to a deadlock.
   d. The main key of the deadlock avoidance method is whenever the request is made for resources then the request must only be approved only in the case if the resulting state is a safe state.
   e. In a case, if the system is unable to fulfill the request of all processes, then the state of the system is called unsafe.
   f. Scheduling algorithm using which DL can be avoided by finding safe state. (Banker Algorithm)
2. **Banker Algorithm**
   a. When a process requests a set of resources, the system must determine whether allocating these resources will leave the system in a safe state. If yes, then the resources may be allocated to the process. If not, then the process must wait till other processes release enough resources.
3. **Deadlock Detection:** Systems haven't implemented deadlock-prevention or a deadlock avoidance technique, then they may employ DL detection then, recovery technique.
   a. Single Instance of Each resource type **(wait-for graph method)**
      i. A deadlock exists in the system if and only if there is a cycle in the wait-for graph. In order to detect the deadlock, the system needs to maintain the wait-for graph and periodically system invokes an algorithm that searches for the cycle in the wait-for graph.
   b. Multiple instances for each resource type
      i. Banker Algorithm
4. **Recovery from Deadlock**
   a. Process termination
      i. Abort all DL processes
      ii. Abort one process at a time until DL cycle is eliminated.
   b. Resource preemption
      i. To eliminate DL, we successively preempt some resources from processes and give these resources to other processes until DL cycle is broken.

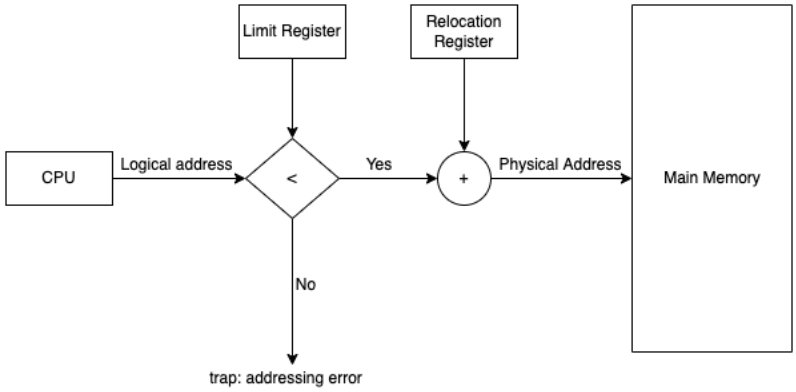
LEC-24: Memory Management Techniques | Contiguous Memory Allocation

1. In Multi-programming environment, we have multiple processes in the main memory (Ready Queue) to keep the CPU utilization high and to make computer responsive to the users.
2. To realize this increase in performance, however, we must keep several processes in the memory; that is, we must **share** the main memory. As a result, we must **manage** main memory for all the different processes.
3. **Logical versus Physical Address Space**
    - **a.** Logical Address
        - **i.** An address generated by the **CPU**.
        - **ii.** The logical address is basically the address of an instruction or data used by a process.
        - **iii.** User can access logical address of the process.
        - **iv.** User has indirect access to the physical address through logical address.
        - **v.** Logical address does not exist physically. Hence, aka, **Virtual address.**
        - vi. The set of all logical addresses that are generated by any program is referred to as Logical Address Space.
        - **vii.** **Range: 0 to *max*.**
    - b. Physical Address
        - i. An address loaded into the memory-address register of the physical memory.
        - ii. User can never access the physical address of the Program.
        - iii. The physical address is in the memory unit. It's a location in the main memory physically.
        - iv. A physical address can be accessed by a user indirectly but not directly.
        - v. The set of all physical addresses corresponding to the Logical addresses is commonly known as Physical Address Space.
        - vi. It is computed by the **Memory Management Unit (MMU).**
        - **vii.** **Range: (R + 0) to (R + max), for a base value R.**
    - **c.** **The runtime mapping from virtual to physical address is done by a hardware device called the memory-management unit (MMU).**
    - d. The user's program mainly generates the logical address, and the user thinks that the program is running in this logical address, but the program mainly needs physical memory in order to complete its execution.
    - e.

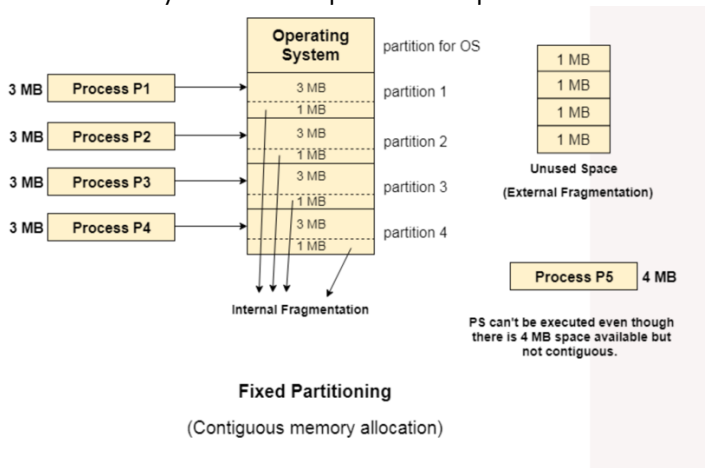


4. How OS manages the isolation and protect? **(Memory Mapping and Protection)**
    - a. OS provides this Virtual Address Space (VAS) concept.
    - b. To separate memory space, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses.
    - c. The relocation register contains value of smallest physical address (Base address [R]); the limit register contains the range of logical addresses (e.g., relocation = 100040 & limit = 74600).
    - d. Each logical address must be less than the limit register.

e. MMU maps the logical address dynamically by adding the value in the relocation register.
f. When CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Since every address generated by the CPU (Logical address) is checked against these registers, we can protect both OS and other users' programs and data from being modified by running process.
g. Any attempt by a program executing in user mode to access the OS memory or other uses' memory results in a trap in the OS, which treat the attempt as a fatal error.
h. **Address Translation**



5. **Allocation Method on Physical Memory**
   a. Contiguous Allocation
   b. Non-contiguous Allocation
6. **Contiguous Memory Allocation**
   a. In this scheme, each process is contained in a single contiguous block of memory.
   b. Fixed Partitioning
      i. The main memory is divided into partitions of equal or different sizes.

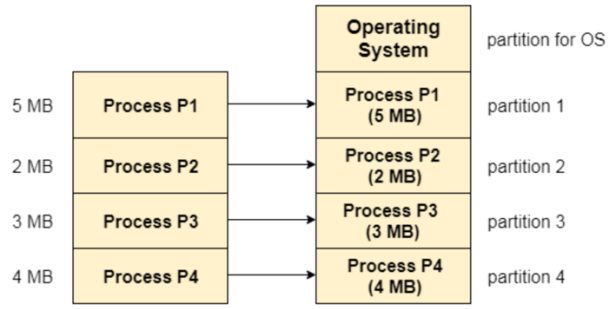


      ii.
      iii. Limitations:
         1. **Internal Fragmentation:** if the size of the process is lesser then the total size of the partition then some size of the partition gets wasted and remain unused. This is wastage of the memory and called internal fragmentation.
         2. **External Fragmentation:** The total unused space of various partitions cannot be used to load the processes even though there is space available but not in the contiguous form.
         3. Limitation on process size: If the process size is larger than the size of maximum sized partition then that process cannot be loaded into the memory. Therefore, a limitation can be imposed on the process size that is it cannot be larger than the size of the largest partition.

4. Low degree of multi-programming: In fixed partitioning, the degree of multiprogramming is fixed and very less because the size of the partition cannot be varied according to the size of processes.

c. Dynamic Partitioning

i. In this technique, the partition size is not declared initially. It is declared at the time of process loading.



Dynamic Partitioning

(Process Size = Partition Size)

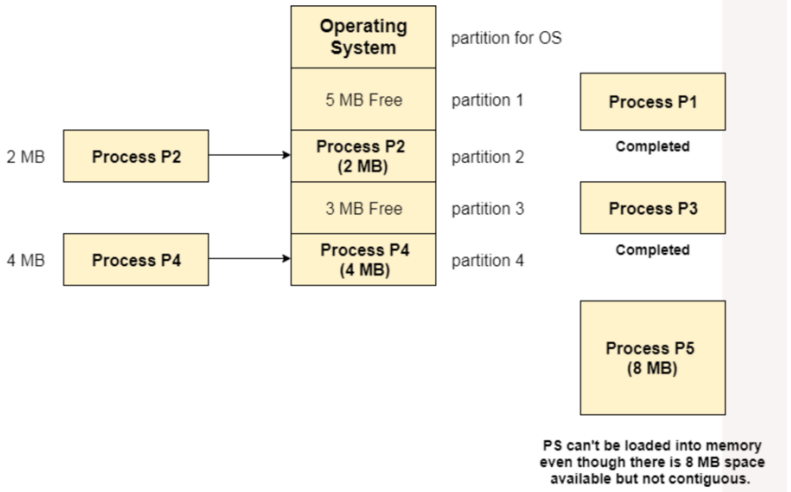
ii.

iii. Advantages over fixed partitioning

1. No internal fragmentation
2. No limit on size of process
3. Better degree of multi-programming

iv. Limitation

1. External fragmentation



External Fragmentation in Dynamic Partitioning

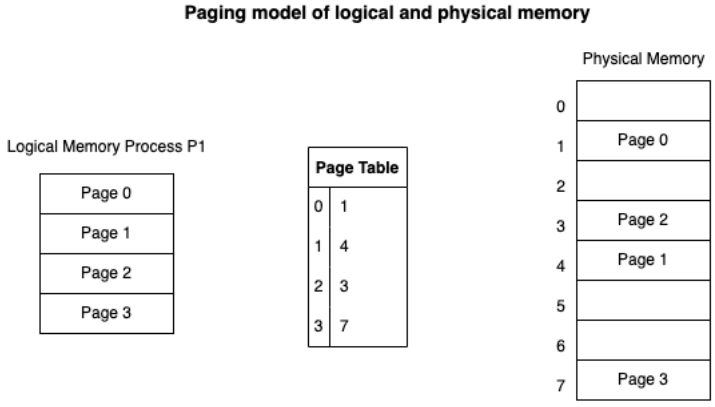
X

LEC-25: Free Space Management

1. **Defragmentation/Compaction**
    a. Dynamic partitioning suffers from external fragmentation.
    b. Compaction to minimize the probability of external fragmentation.
    c. All the free partitions are made contiguous, and all the loaded partitions are brought together.
    d. By applying this technique, we can store the bigger processes in the memory. The free partitions are merged which can now be allocated according to the needs of new processes. This technique is also called **defragmentation**.
    e. The efficiency of the system is decreased in the case of compaction since all the free spaces will be transferred from several places to a single place.
2. **How free space is stored/represented in OS?**
    a. Free holes in the memory are represented by a free list (Linked-List data structure).
3. **How to satisfy a request of a of n size from a list of free holes?**
    a. Various algorithms which are implemented by the Operating System in order to find out the holes in the linked list and allocate them to the processes.
    b. First Fit
        i. Allocate the first hole that is big enough.
        ii. Simple and easy to implement.
        iii. Fast/Less time complexity
    c. Next Fit
        i. Enhancement on First fit but starts search always from last allocated hole.
        ii. Same advantages of First Fit.
    d. Best Fit
        i. Allocate smallest hole that is big enough.
        ii. Lesser internal fragmentation.
        iii. May create many small holes and cause major external fragmentation.
        iv. Slow, as required to iterate whole free holes list.
    e. Worst Fit
        i. Allocate the largest hole that is big enough.
        ii. Slow, as required to iterate whole free holes list.
        iii. Leaves larger holes that may accommodate other processes.

LEC-26: Paging | Non-Contiguous Memory Allocation

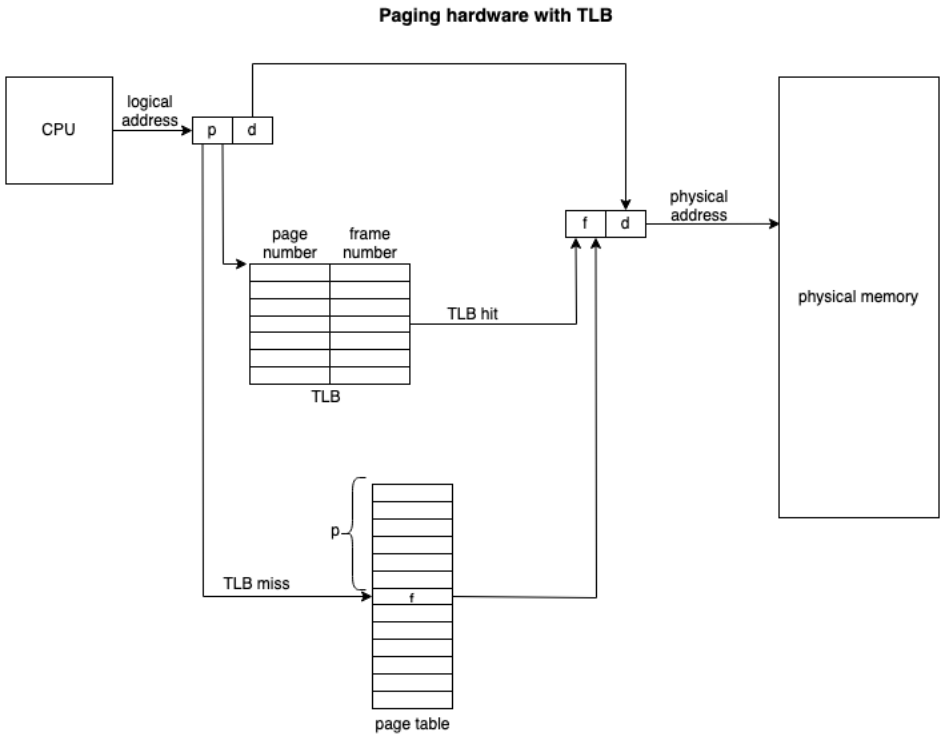
1. The main **disadvantage of Dynamic partitioning is External Fragmentation.**
    a. Can be removed by Compaction, but with overhead.
    b. We need more dynamic/flexible/optimal mechanism, to load processes in the partitions.
2. **Idea behind Paging**
    a. If we have only two small non-contiguous free holes in the memory, say 1KB each.
    b. If OS wants to allocate RAM to a process of 2KB, in contiguous allocation, it is not possible, as we must have contiguous memory space available of 2KB. (External Fragmentation)
    c. What if we divide the process into 1KB-1KB blocks?
3. Paging
    a. Paging is a memory-management scheme that permits the physical address space of a process to be non-contiguous.
    b. It avoids external fragmentation and the need of compaction.
    c. Idea is to divide the physical memory into fixed-sized blocks called **Frames**, along with divide logical memory into blocks of same size called **Pages**. (# Page size = Frame size)
    d. **Page size** is usually determined by the processor architecture. Traditionally, pages in a system had uniform size, such as 4,096 bytes. However, processor designs often allow two or more, sometimes simultaneous, page sizes due to its benefits.
    e. Page Table
        i. A Data structure stores which page is mapped to which frame.
        ii. The page table contains the base address of each page in the physical memory.
    f. Every address generated by CPU (logical address) is divided into two parts: a page number (p) and a page offset (d). The p is used as an index into a page table to get base address the corresponding frame in physical memory.



    g. Page table is stored in main memory at the time of process creation and its base address is stored in process control block (**PCB**).
    h. A page table base register (**PTBR**) is present in the system that points to the current page table. Changing page tables requires only this one register, at the time of context-switching.
4. **How Paging avoids external fragmentation?**
    a. Non-contiguous allocation of the pages of the process is allowed in the random free frames of the physical memory.
5. **Why paging is slow and how do we make it fast?**
    a. There are too many memory references to access the desired location in physical memory.
6. **Translation Look-aside buffer (TLB)**
    a. A Hardware support to speed-up paging process.
    b. It's a hardware cache, high speed memory.
    c. TBL has key and value.

d. Page table is stores in main memory & because of this when the memory references is made the translation is slow.

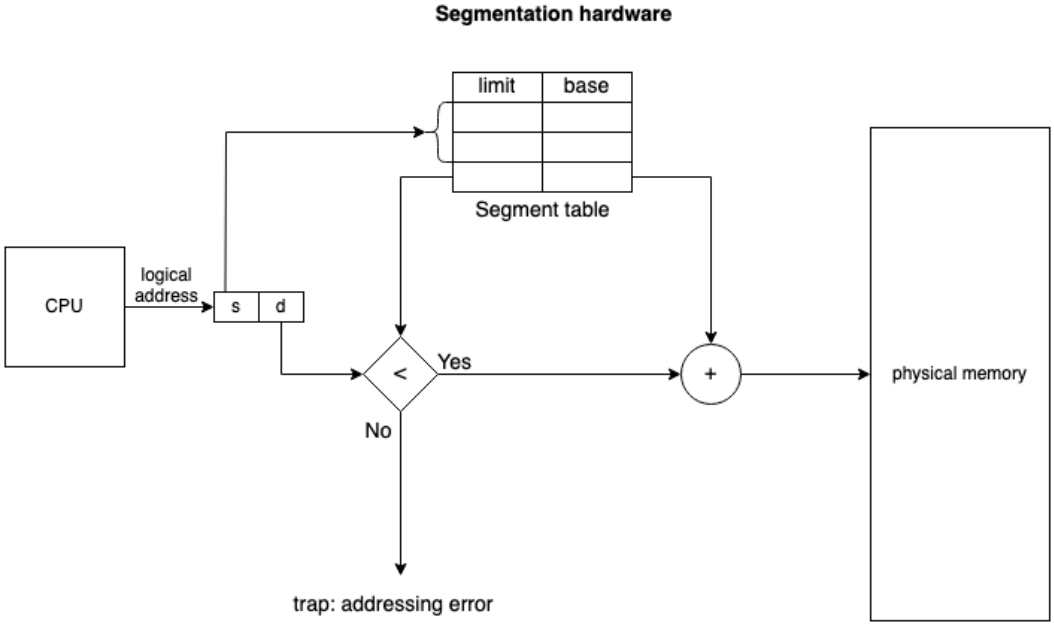
e. When we are retrieving physical address using page table, after getting frame address corresponding to the page number, we put an entry of the into the TLB. So that next time, we can get the values from TLB directly without referencing actual page table. Hence, make paging process faster.



f. **TLB hit, TLB contains the mapping for the requested logical address.**

g. **Address space identifier (ASIDs)** is stored in each entry of TLB. ASID uniquely identifies each process and is used to **provide address space protection and allow to TLB to contain entries for several different processes.** When TLB attempts to resolve virtual page numbers, it ensures that the ASID for the currently executing process matches the ASID associated with virtual page. If it doesn't match, the attempt is treated as TLB miss.

LEC-27: Segmentation | Non-Contiguous Memory Allocation

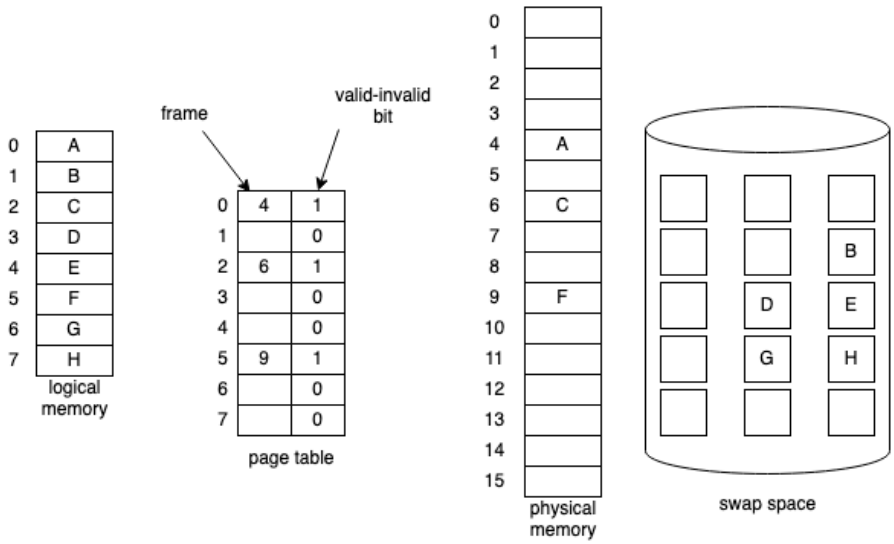
1. An important aspect of memory management that become unavoidable with paging is separation of user's view of memory from the actual physical memory.
2. Segmentation is memory management technique that supports the **user view of memory.**
3. A logical address space is a collection of segments, these segments are based on **user view** of logical memory.
4. Each segment has **segment number and offset,** defining a segment.
   <segment-number, offset> {s,d}
5. Process is divided into **variable segments based on user view.**
6. **Paging** is closer to the Operating system rather than the **User**. It divides all the processes into the form of pages although a process can have some relative parts of functions which need to be loaded in the same page.
7. Operating system doesn't care about the **User's view** of the process. It may **divide the same function into different page**s and those **pages may or may not be loaded at the same time into the memory.** It decreases the efficiency of the system.
8. It is better to have segmentation which divides the process into the segments. Each segment contains the same type of functions such as the main function can be included in one segment and the library functions can be included in the other segment.
9. 

10. **Advantages**:
    a. No internal fragmentation.
    b. One segment has a contiguous allocation, hence efficient working within segment.
    c. The size of segment table is generally less than the size of page table.
    d. It results in a more efficient system because the compiler keeps the same type of functions in one segment.
11. **Disadvantages**:
    a. External fragmentation.
    b. The different size of segment is not good that the time of swapping.
12. Modern System architecture provides both segmentation and paging implemented in some hybrid approach.

LEC-28: What is Virtual Memory? || Demand Paging || Page Faults

1. **Virtual memory** is a technique that allows the execution of processes that are not completely in the memory. It provides user an illusion of having a very big main memory. This is done by treating a part of secondary memory as the main memory. (Swap-space)
2. **Advantage** of this is, programs can be larger than physical memory.
3. It is required that instructions must be in physical memory to be executed. But it limits the size of a program to the size of physical memory. In fact, in many cases, the entire program is not needed at the same time. So, we want an ability to execute a program that is only partially in memory would give many benefits:
    a. A program would no longer be constrained by the amount of physical memory that is available.
    b. Because each user program could take less physical memory, more programs could be run at the same time, with a corresponding **increase in CPU utilization and throughput.**
    c. Running a program that is not entirely in memory would benefit **both the system and the user.**
4. Programmer is provided very large virtual memory when only a smaller physical memory is available.
5. **Demand Paging** is a popular method of **virtual memory management.**
6. In demand paging, the pages of a process which are least used, get stored in the secondary memory.
7. A page is copied to the main memory when its demand is made, or **page fault** occurs. There are various **page replacement algorithms** which are used to determine the pages which will be replaced.
8. Rather than swapping the entire process into memory, we use **Lazy Swapper.** A lazy swapper never swaps a page into memory unless that page will be needed.
9. We are viewing a process as a sequence of pages, rather than one large contiguous address space, using the term **Swapper is technically incorrect**. A swapper manipulates entire processes, whereas a **Pager** is concerned with individual pages of a process.
10. **How Demand Paging works?**
    a. When a process is to be swapped-in, the pager guesses which pages will be used.
    b. Instead of swapping in a whole process, the pager brings only those pages into memory. This, it avoids reading **into memory pages that will not be used anyway.**
    c. Above way**, OS decreases the swap time and the amount of physical memory needed.**
    d. The **valid-invalid bit scheme in the page table** is used to distinguish between pages that are in memory and that are on the disk.
        i. Valid-invalid bit **1** means, the associated page is both legal and in memory.
        ii. Valid-invalid bit **0** means, the page either is not valid (not in the LAS of the process) or is valid but is currently on the disk.

Page table when some pages are not in memory



e.

f. If a process never attempts **to access some invalid bit page, the process will be executed successfully without even the need pages present in the swap space.**

g. What happens if the process tries to access a page that was not brought into memory, access to a page marked invalid causes **page fault.** Paging hardware noticing invalid bit for a demanded page will cause **a trap to the OS.**

h. The procedure to handle the page fault:

i. Check an internal table (in PCB of the process) to determine whether the reference was valid or an invalid memory access.

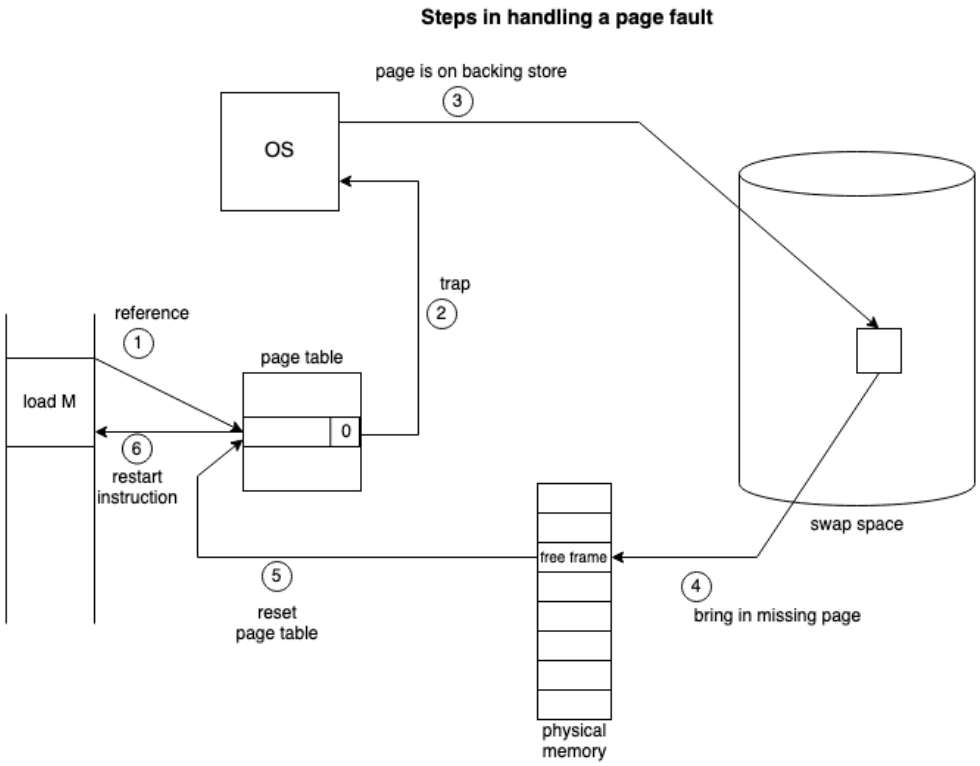
ii. If ref. was invalid process throws exception.
If ref. is valid, pager will swap-in the page.

iii. We find a free frame (from free-frame list)

iv. Schedule a disk operation to read the desired page into the newly allocated frame.

v. When disk read is complete, we modify the page table that, the page is now in memory.

vi. Restart the instruction that was interrupted by the trap. The process can now access the page as through it had always been in memory.



i.

j. **Pure Demand Paging**
    i. In extreme case, we can start executing a process with no pages in memory. When OS sets the instruction pointer to the first instruction of the process, which is not in the memory. The process immediately faults for the page and page is brought in the memory.
    ii. Never bring a page into memory until it is required.
k. We use **locality of reference** to bring out reasonable performance from demand paging.

11. **Advantages of Virtual memory**
    a. The degree of multi-programming will be increased.
    b. User can run large apps with less real physical memory.
12. **Disadvantages of Virtual Memory**
    a. The system can become slower as swapping takes time.
    b. **Thrashing** may occur.

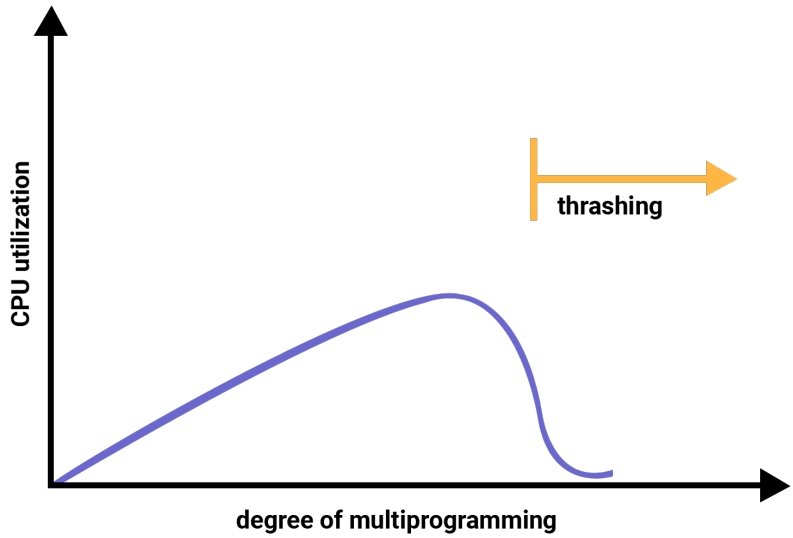
LEC-29: Page Replacement Algorithms

1. Whenever **Page Fault** occurs, that is, a process tries to access a page which is not currently present in a frame and OS must bring the page from swap-space to a frame.
2. OS must do page replacement to accommodate new page into a free frame, but there might be a possibility the system is working in high utilization and all the frames are busy, in that case OS must replace one of the pages allocated into some frame with the new page.
3. The **page replacement algorithm** decides which memory page is to be replaced. Some allocated page is swapped out from the frame and new page is swapped into the freed frame.
4. **Types** of Page Replacement Algorithm: (**AIM** is to have minimum page faults)
    a. **FIFO**
        i. Allocate frame to the page as it comes into the memory by **replacing the oldest page.**
        ii. Easy to implement.
        iii. Performance is **not** always good
            1. The page replaced may be an initialization module that was used long time ago **(Good replacement candidate)**
            2. The page may contain a heavily used variable that was initialized early and is in content use. (**Will again cause page fault)**
        iv. **Belady's** anomaly is present.
            1. **In the case of LRU and optimal page replacement algorithms, it is seen that the number of page faults will be reduced if we increase the number of frames.** However, Balady found that, In FIFO page replacement algorithm, the number of page faults will get increased with the increment in number of frames.
            2. This is the strange behavior shown by FIFO algorithm **in some of the cases.**
    b. **Optimal** page replacement
        i. Find if a page that is never referenced in future. If such a page exists, replace this page with new page.
        If no such page exists, find a page that is **referenced farthest in future**. Replace this page with new page.
        ii. **Lowest** page fault rate among any algorithm.
        iii. Difficult to implement as **OS requires future knowledge of reference string** which is kind of impossible. (Similar to SJF scheduling)
    c. Least-recently used (**LRU**)
        i. We can use recent past as an approximation of the near future then we can replace the page that has not been used for the longest period.
        ii. Can be implemented by two ways
            1. **Counters**
                a. Associate time field with each page table entry.
                b. Replace the page with smallest time value.
            2. **Stack**
                a. Keep a stack of page number.
                b. Whenever page is referenced, it is removed from the stack & put on the top.
                c. By this, most recently used is always on the top, & least recently used is always on the bottom.
                d. As entries might be removed from the middle of the stack, so Doubly linked list can be used.
    d. **Counting**-based page replacement – Keep a counter of the number of references that have been made to **each** page. (Reference counting)

i. Least frequently used (**LFU**)
    1. Actively used pages should have a large reference count.
    2. Replace page with the smallest count.
ii. Most frequently used (**MFU**)
    1. Based on the argument that the page with the smallest count was probably just brought in and has yet to be used.
iii. Neither MFU nor LFU replacement is common.

LEC-30: Thrashing

1. **Thrashing**
    a. If the process doesn't have the number of frames it needs to support pages in active use, it will quickly page-fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately.
    b. This **high paging activity is called Thrashing.**
    c. A system is Thrashing when it **spends more time servicing the page faults than executing processes.**



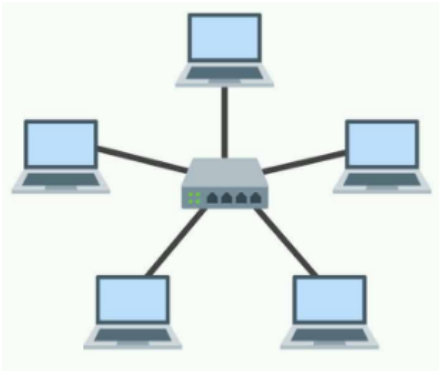
    d. **Technique to Handle Thrashing**
        i. **Working set model**
            1. This model is based on the concept of the **Locality** Model.
            2. The basic principle states that if we allocate enough frames to a process to accommodate its current locality, it will only fault whenever **it moves to some new locality.** But if the allocated frames are lesser than the size of the current locality, **the process is bound to thrash.**
        ii. **Page Fault frequency**
            1. **Thrashing** has a high page-fault rate.
            2. We want to **control** the page-fault rate.
            3. When it is too high, the process needs more frames. Conversely, if the page-fault rate is too low, then the process may have too many frames.
            4. We establish upper and lower bounds on the desired page fault rate.
            5. If pf-rate exceeds the upper limit, allocate the process another frame, if pf-rate fails falls below the lower limit, remove a frame from the process.
            6. By controlling pf-rate, thrashing can be prevented.

Computer Networks

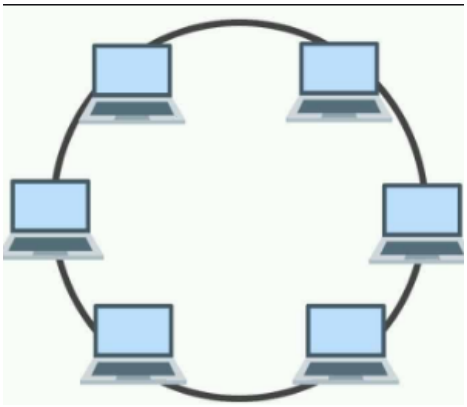
- **Network** : A network is a set of devices that are connected with a physical media link. In a network, two or more nodes are connected by a physical link or two or more networks are connected by one or more nodes. A network is a collection of devices connected to each other to allow the sharing of data.

- **Network Topology** : Network topology specifies the layout of a computer network. It shows how devices and cables are connected to each other.

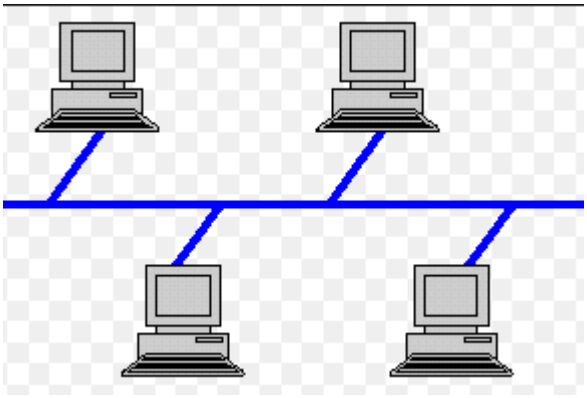
  Types of Network Topology :
  - **Star :**
    - Star topology is a network topology in which all the nodes are connected to a single device known as a central device.
    - Star topology requires more cable compared to other topologies. Therefore, it is more robust as a failure in one cable will only disconnect a specific computer connected to this cable.
    - If the central device is damaged, then the whole network fails.
    - Star topology is very easy to install, manage and troubleshoot. It is commonly used in office and home networks.



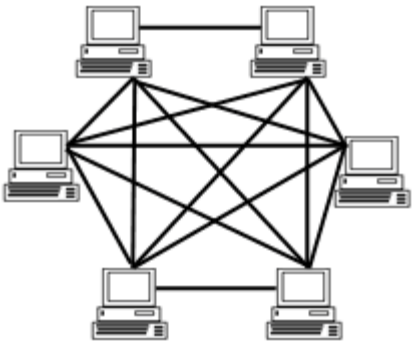
- **Ring** :
  1. Ring topology is a network topology in which nodes are exactly connected to two or more nodes and thus, forming a single continuous path for the transmission.
  2. It does not need any central server to control the connectivity among the nodes.
  3. If the single node is damaged, then the whole network fails.
  4. Ring topology is very rarely used as it is expensive, difficult to install and manage.
  5. Examples of Ring topology are SONET network, SDH network, etc.



- **Bus :**
    1. Bus topology is a network topology in which all the nodes are connected to a single cable known as a central cable or bus.
    2. It acts as a shared communication medium, i.e., if any device wants to send the data to other devices, then it will send the data over the bus which in turn sends the data to all the attached devices.
    3. Bus topology is useful for a small number of devices.
    4. As if the bus is damaged then the whole network fails.



- **Mesh** :
    1. Mesh topology is a network topology in which all the nodes are individually connected to other nodes.
    2. It does not need any central switch or hub to control the connectivity among the nodes.
    3. **Mesh topology is categorized into two parts**: Fully connected mesh topology: In this topology, all the nodes are connected to each other. Partially connected mesh topology: In this topology, all the nodes are not connected to each other.
    4. It is robust as a failure in one cable will only disconnect the specified computer connected to this cable.
    5. Mesh topology is rarely used as installation and configuration are difficult when connectivity gets more.
    6. Cabling cost is high as it requires bulk wiring.



- **Tree** :
    1. Tree topology is a combination of star and bus topology. It is also known as the expanded star topology.
    2. In tree topology, all the star networks are connected to a single bus.
    3. Ethernet protocol is used in this topology.
    4. In this, the whole network is divided into segments known as star networks which can be easily maintained. If one segment is damaged, there is no effect on other segments.
    5. Tree topology depends on the "main bus," and if it breaks, then the whole network gets damaged

- **Hybrid** :
    1. A hybrid topology is a combination of different topologies to form a resulting topology.
    2. If star topology is connected with another star topology, then it remains a star topology. If star topology is connected with different topology, then it becomes a Hybrid topology.
    3. It provides flexibility as it can be implemented in a different network environment.

- **<u>Different Types of Networks</u>** : (Imp) - Networks can be divided on the basis of area of distribution. For example:

- **PAN (Personal Area Network)**: Its range limit is up to 10 meters. It is created for personal use. Generally, personal devices are connected to this network. For example computers, telephones, fax, printers, etc.
- **LAN (Local Area Network)**: It is used for a small geographical location like office, hospital, school, etc.
- **HAN (House Area Network)**: It is actually a LAN that is used within a house and used to connect homely devices like personal computers, phones, printers, etc.
- **CAN (Campus Area Network)**: It is a connection of devices within a campus area which links to other departments of the organization within the same campus.
- **MAN (Metropolitan Area Network)**: It is used to connect the devices which span to large cities like metropolitan cities over a wide geographical area.
- **WAN (Wide Area Network)**: It is used over a wide geographical location that may range to connect cities and countries.
- **GAN (Global Area Network)**: It uses satellites to connect devices over the global area.

- **VPN (Virtual Private Network)** : VPN or the Virtual Private Network is a private WAN (Wide Area Network) built on the internet. It allows the creation of a secured tunnel (protected network) between different networks using the internet (public network). By using the VPN, a client can connect to the organization's network remotely.
- **Advantages of VPN** :
    1. VPN is used to connect offices in different geographical locations remotely and is cheaper when compared to WAN connections.
    2. VPN is used for secure transactions and confidential data transfer between multiple offices located in different geographical locations.
    3. VPN keeps an organization's information secured against any potential threats or intrusions by using virtualization.
    4. VPN encrypts the internet traffic and disguises the online identity.
- **Types of VPN** :

- **Access VPN:** Access VPN is used to provide connectivity to remote mobile users and telecommuters. It serves as an alternative to dial-up connections or ISDN (Integrated Services Digital Network) connections. It is a low-cost solution and provides a wide range of connectivity.
- **Site-to-Site VPN:** A Site-to-Site or Router-to-Router VPN is commonly used in large companies having branches in different locations to connect the network of one office to another in different locations. There are 2 sub-categories as mentioned below:
- **Intranet VPN:** Intranet VPN is useful for connecting remote offices in different geographical locations using shared infrastructure (internet connectivity and servers) with the same accessibility policies as a private WAN (wide area network).
- **Extranet VPN:** Extranet VPN uses shared infrastructure over an intranet, suppliers, customers, partners, and other entities and connects them using dedicated connections.

- **IPv4 Address** : An IP address is a 32-bit dynamic address of a node in the network. An IPv4 address has 4 octets of 8-bit each with each number with a value up to 255. IPv4 classes are differentiated based on the number of hosts it supports on the network. There are five types of IPv4 classes and are based on the first octet of IP addresses which are classified as Class A, B, C, D, or E.

IPv4 Class	IPv4 Start Address	IPv4 End Address	Usage
A	0.0.0.0	127.255.255.255	Used for Large Network
B	128.0.0.0	191.255.255.255	Used for Medium Size Network
C	192.0.0.0	223.255.255.255	Used for Local Area Network
D	224.0.0.0	239.255.255.255	Reserved for Multicasting
E	240.0.0.0	255.255.255.254	Study and R&D

- **OSI (Open System Interconnections)** (Imp) : It is a network architecture model based on the ISO standards. It is called the OSI model as it deals with connecting the systems that are open for communication with other systems. **The OSI model has seven layers.** The principles used to arrive at the seven layers can be summarized briefly as below:
    1. Create a new layer if a different abstraction is needed.
    2. Each layer should have a well-defined function.
    3. The function of each layer is chosen based on internationally standardized protocols.
- **Seven Layers** :

1. **Physical Layer**
    - It is the lowest layer of the OSI reference model.
    - It is used for the transmission of an unstructured raw bit stream over a physical medium.
    - Physical layer transmits the data either in the form of electrical/optical or mechanical form.
    - The physical layer is mainly used for the physical connection between the devices, and such physical connection can be made by using twisted-pair cable, fibre-optic or wireless transmission media.

2. **DataLink Layer**
    - It is used for transferring the data from one node to another node.
    - It receives the data from the network layer and converts the data into data frames and then attaches the physical address to these frames which are sent to the physical layer.
    - It enables the error-free transfer of data from one node to another node.

Functions of Data-link layer:

- **Frame synchronization**: Data-link layer converts the data into frames, and it ensures that the destination must recognize the starting and ending of each frame.
- **Flow control**: Data-link layer controls the data flow within the network.

- **Error control**: It detects and corrects the error occurred during the transmission from source to destination.
- **Addressing**: Data-link layers attach the physical address with the data frames so that the individual machines can be easily identified.
- **Link management**: Data-link layer manages the initiation, maintenance and termination of the link between the source and destination for the effective exchange of data.

3. Network Layer

- Network layer converts the logical address into the physical address.
- The routing concept means it determines the best route for the packet to travel from source to the destination.
  Functions of network layer :
- **Routing**: The network layer determines the best route from source to destination. This function is known as routing.
- **Logical addressing**: The network layer defines the addressing scheme to identify each device uniquely.
- **Packetizing**: The network layer receives the data from the upper layer and converts the data into packets. This process is known as packetizing.
- **Internetworking**: The network layer provides the logical connection between the different types of networks for forming a bigger network.
- **Fragmentation**: It is a process of dividing the packets into fragments..

4. Transport Layer

- It delivers the message through the network and provides error checking so that no error occurs during the transfer of data.
- **It provides two kinds of services**:
  - **Connection-oriented transmission**: In this transmission, the receiver sends the acknowledgement to the sender after the packet has been received.

- **Connectionless transmission**: In this transmission, the receiver does not send the acknowledgement to the sender.

5. Session Layer

- The main responsibility of the session layer is beginning, maintaining and ending the communication between the devices.
- Session layer also reports the error coming from the upper layers.
- Session layer establishes and maintains the session between the two users.

6. Presentation Layer

- The presentation layer is also known as a Translation layer as it translates the data from one format to another format.
- At the sender side, this layer translates the data format used by the application layer to the common format and at the receiver side, this layer translates the common format into a format used by the application layer.

  Functions of presentation layer:
  - Character code translation
  - Data conversion
  - Data compression
  - Data encryption

7. Application Layer

- Application layer enables the user to access the network.
- It is the topmost layer of the OSI reference model.
- Application layer protocols are file transfer protocol, simple mail transfer protocol, domain name system, etc.
- The most widely used application protocol is HTTP(Hypertext transfer protocol). A user sends the request for the web page using HTTP.

- **TCP/IP Reference Model** : It is a compressed version of the OSI model with only **4 layers**. It was developed by the US Department of Defence (DoD) in the 1860s. The name of this model is based on 2 standard protocols used i.e. TCP (Transmission Control Protocol) and IP (Internet Protocol).
    1. **Link** : Decides which links such as serial lines or classic Ethernet must be used to meet the needs of the connectionless internet layer. Ex - Sonet, Ethernet
    2. **Internet** : The internet layer is the most important layer which holds the whole architecture together. It delivers the IP packets where they are supposed to be delivered. Ex - IP, ICMP.
    3. **Transport** : Its functionality is almost the same as the OSI transport layer. It enables peer entities on the network to carry on a conversation. Ex - TCP, UDP (User Datagram Protocol)
    4. **Application** : It contains all the higher-level protocols. Ex - HTTP, SMTP, RTP, DNS.

- **HTTP and HTTPS** :

  HTTP is the HyperText Transfer Protocol which defines the set of rules and standards on how the information can be transmitted on the World Wide Web (WWW). It helps the web browsers and web servers for communication. It is a 'stateless protocol' where each command is independent with respect to the previous command. **HTTP is an application layer protocol built upon the TCP. It uses port 80 by default.**

  HTTPS is the HyperText Transfer Protocol Secure or Secure HTTP. It is an advanced and secured version of HTTP. On top of HTTP, SSL/TLS protocol is used to provide security. **It enables secure transactions by encrypting the communication and also helps identify network servers securely. It uses port 443 by default.**

- **DNS (Imp)** :
    1. DNS is an acronym that stands for Domain Name System.DNS was introduced by Paul Mockapetris and Jon Postel in 1983.
    2. It is a naming system for all the resources over the internet which includes physical nodes and applications. It is used to locate resources easily over a network.
    3. DNS is an internet which maps the domain names to their associated IP addresses.

4. Without DNS, users must know the IP address of the web page that you wanted to access.

- **Working of DNS** (**Imp**)**:** If you want to visit the website of "shaurya", then the user will type "https://www.shaurya.com" into the address bar of the web browser. Once the domain name is entered, then the domain name system will translate the domain name into the IP address which can be easily interpreted by the computer. Using the IP address, the computer can locate the web page requested by the user.

- **DNS Forwarder** : A forwarder is used with a DNS server when it receives DNS queries that cannot be resolved quickly. So it forwards those requests to external DNS servers for resolution. A DNS server which is configured as a forwarder will behave differently than the DNS server which is not configured as a forwarder.

- **SMTP Protocol** : SMTP is the **Simple Mail Transfer Protocol**. SMTP sets the rule for communication between servers. This set of rules helps the software to transmit emails over the internet. It supports both End-to-End and Store-and-Forward methods. It is in always-listening mode on port 25.

- **Difference Between TCP (Transmission Control Protocol) and UDP (User Datagram Protocol)**:
    1. **TCP** is a connection-oriented protocol, whereas **UDP** is a connectionless protocol. A key **difference between TCP and UDP** is speed, as **TCP** is comparatively slower than **UDP**. Overall, **UDP** is a much faster, simpler, and efficient protocol, however, retransmission of lost data packets is only possible with **TCP**
    2. TCP provides extensive error checking mechanisms. It is because it provides flow control and acknowledgment of data. UDP has only the basic error checking mechanism using checksums.

Important Protocols

A protocol is a set of rules which is used to govern all the aspects of information communication.The main elements of a protocol are:

- **Syntax**: It specifies the structure or format of the data. It also specifies the order in which they are presented.
- **Semantics**: It specifies the meaning of each section of bits.
- **Timing**: Timing specifies two characteristics: When data should be sent and how fast it can be sent.

- **DHCP**: DHCP is the **Dynamic Host Configuration Protocol**. It is an application layer protocol used to auto-configure devices on IP networks enabling them to use the TCP and UDP-based protocols. The DHCP servers auto-assign the IPs and other network configurations to the devices individually which enables them to communicate over the IP network. It helps to get the subnet mask, IP address and helps to resolve the DNS. It uses port 67 by default.
- **FTP** : FTP is a **File Transfer Protocol.** It is an application layer protocol used to transfer files and data reliably and efficiently between hosts. It can also be used to download files from remote servers to your computer. It uses port 27 by default.
- **ICMP** : ICMP is the **Internet Control Message Protocol.** It is a network layer protocol used for error handling. It is mainly used by network devices like routers for diagnosing the network connection issues and crucial for error reporting and testing if the data is reaching the preferred destination in time. It uses port 7 by default.
- **ARP** : ARP is **Address Resolution Protocol**. It is a network-level protocol used to convert the logical address i.e. IP address to the device's physical address i.e. MAC address. It can also be used to get the MAC address of devices when they are trying to communicate over the local network.
- **RIP** :RIP stands for Routing Information Protocol. It is accessed by the routers to send data from one network to another. RIP is a dynamic protocol which is used to find the best route from source to the destination over a network by using the hop count

algorithm. Routers use this protocol to exchange the network topology information. This protocol can be used by small or medium-sized networks.

- **MAC address and IP address** (Imp) :

  1. Both MAC (Media Access Control) Address and IP Address are used to **uniquely define a device on the internet.** NIC Card's Manufacturer provides the MAC Address, on the other hand Internet Service Provider provides IP Address.

  2. **The main difference between MAC and IP address** is that MAC Address is used to ensure the physical address of a computer. It uniquely identifies the devices on a network. While IP addresses are used to uniquely identify the connection of a network with that device taking part in a network.

- **Ipconfig and Ifconfig :**

  1. Ipconfig : Internet Protocol Configuration, It is a command used in Microsoft operating systems to view and configure network interfaces

  2. Ifconfig : Interface Configuration, It is a command used in MAC, Linux, UNIX operating systems to view and configure network interfaces

- **Firewall :** The firewall is a network security system that is used to monitor the incoming and outgoing traffic and blocks the same based on the firewall security policies. It acts as a wall between the internet (public network) and the networking devices (a private network). It is either a hardware device, software program, or a combination of both. It adds a layer of security to the network.

Important Key Points

1. **What happens when you enter google.com in the web browser? (Most Imp)**

Steps :

- Check the browser cache first if the content is fresh and present in the cache display the same.
- If not, the browser checks if the IP of the URL is present in the cache (browser and OS) if not then requests the OS to do a DNS lookup using UDP to get the corresponding IP address of the URL from the DNS server to establish a new TCP connection.
- A new TCP connection is set between the browser and the server using three-way handshaking.
- An HTTP request is sent to the server using the TCP connection.
- The web servers running on the Servers handle the incoming HTTP request and send the HTTP response.
- The browser processes the HTTP response sent by the server and may close the TCP connection or reuse the same for future requests.
- If the response data is cacheable then browsers cache the same.
- Browser decodes the response and renders the content.

2. **Hub:** Hub is a networking device which is used to transmit the signal to each port (except one port) to respond from which the signal was received. Hub is operated on a Physical layer. In this packet filtering is not available. It is of two types: Active Hub, Passive Hub.

   Switch: Switch is a network device which is used to enable the connection establishment and connection termination on the basis of need. Switch is operated on the Data link layer. In this packet filtering is available. It is a type of full duplex transmission mode and it is also called an efficient bridge.

3. A **subnet** is a network inside a network achieved by the process called subnetting which helps divide a network into subnets. It is used for getting a higher routing efficiency and enhances the security of the network. It reduces the time to extract the host address from the routing table.

4. **The reliability of a network can be measured by the following factors**:

- Downtime: The downtime is defined as the required time to recover.
- Failure Frequency: It is the frequency when it fails to work the way it is intended.
- Catastrophe: It indicates that the network has been attacked by some unexpected event such as fire, earthquake.

5. There are mainly two criteria which make a **network effective and efficient**:

- Performance : performance can be measured in many ways like transmit time and response time.
- Reliability: reliability is measured by frequency of failure.
- Robustness**:** robustness specifies the quality or condition of being strong and in good condition.
- Security: It specifies how to protect data from unauthorized access and viruses.

6. **Node and Link** : A network is a connection setup of two or more computers directly connected by some physical mediums like optical fiber or coaxial cable. This physical medium of connection is known as a link, and the computers that it is connected to are known as nodes.

7. **Gateway and router** : A node that is connected to two or more networks is commonly known as a gateway. It is also known as a router. It is used to forward messages from one network to another. **Both the gateway and router regulate the traffic in the network**. **Differences between gateway and router**: A router sends the data between two similar networks while gateway sends the data between two dissimilar networks.

8. **NIC (Imp)** : NIC stands for **Network Interface Card**. It is a peripheral card attached to the PC to connect to a network. Every NIC has its own MAC address that identifies the PC on the network. It provides a wireless connection to a local area network. NICs were mainly used in desktop computers.

9. **POP3 stands for Post Office Protocol version3**. POP is responsible for accessing the mail service on a client machine. POP3 works on two models such as Delete mode and Keep mode.

10. **Private IP Address** - There are three ranges of IP addresses that have been reserved for IP addresses. They are not valid for use on the internet. If you want to access the internet on these private IPs, you must use a proxy server or NAT server.

    Public IP Address - A public IP address is an address taken by the Internet Service Provider which facilitates communication on the internet.

11. **RAID** (Redundant Array of Inexpensive/Independent Disks): It is a method to provide Fault Tolerance by using multiple Hard Disc Drives.

12. **Netstat :** It is a command line utility program. It gives useful information about the current TCP/IP setting of a connection.

13. **Ping** : The "ping" is a utility program that allows you to check the connectivity between the network devices. You can ping devices using its IP address or name.

14. The processes on each machine that communicate at a given layer are called **peer-peer processes. (P2P).**

15. **Unicasting:** If the message is sent to a single node from the source then it is known as unicasting. This is commonly used in networks to establish a new connection.

    Anycasting: If the message is sent to any of the nodes from the source then it is known as anycasting. It is mainly used to get the content from any of the servers in the Content Delivery System.

    Multicasting: If the message is sent to a subset of nodes from the source then it is known as multicasting. Used to send the same data to multiple receivers.

    Broadcasting: If the message is sent to all the nodes in a network from a source then it is known as broadcasting. DHCP and ARP in the local network use broadcasting.

Introduction to React

React is an efficient, flexible, and open-source **JavaScript framework** library that allows developers to the creation of simple, fast, and scalable web applications. Jordan Walke, a software engineer who was working for Facebook created React. It was first deployed on the news feed of Facebook in 2011 and on Instagram in 2012. Developers from the Javascript background can easily develop web applications with the help of React.

React Hooks will allow you to use the state and other features of React in which requires a class to be written by you. In simple words, we can say that, React Hooks are the functions that will connect React state with the lifecycle features from the function components. React Hooks is among the features that are implemented latest in the version React 16.8.

Scope of React: The selection of the right technology for application or web development is becoming more challenging. React has been considered to be the fastest-growing Javascript framework among all. The tools of Javascript are firming their roots slowly and steadily in the marketplace and the React certification demand is exponentially increasing. React is a clear win for front-end developers as it has a quick learning curve, clean abstraction, and reusable components. Currently, there is no end in sight for React as it keeps evolving.

React Interview Questions for Freshers

1. What is React?

React is a front-end and open-source JavaScript library which is useful in developing user interfaces specifically for applications with a single page. It is helpful in building complex and reusable user interface(UI) components of mobile and web applications as it follows the component-based approach.

The important features of React are:

- It supports server-side rendering.
- It will make use of the virtual DOM rather than real DOM (Data Object Model) as RealDOM manipulations are expensive.
- It follows unidirectional data binding or data flow.
- It uses reusable or composable UI components for developing the view.

2. What are the advantages of using React?

MVC is generally abbreviated as Model View Controller.

- **Use of Virtual DOM to improve efficiency:** React uses virtual DOM to render the view. As the name suggests, virtual DOM is a virtual representation of the real DOM. Each

time the data changes in a react app, a new virtual DOM gets created. Creating a virtual DOM is much faster than rendering the UI inside the browser. Therefore, with the use of virtual DOM, the efficiency of the app improves.

- **Gentle learning curve:** React has a gentle learning curve when compared to frameworks like Angular. Anyone with little knowledge of javascript can start building web applications using React.
- **SEO friendly:** React allows developers to develop engaging user interfaces that can be easily navigated in various search engines. It also allows server-side rendering, which boosts the SEO of an app.
- **Reusable components:** React uses component-based architecture for developing applications. Components are independent and reusable bits of code. These components can be shared across various applications having similar functionality. The re-use of components increases the pace of development.
- **Huge ecosystem of libraries to choose from:** React provides you with the freedom to choose the tools, libraries, and architecture for developing an application based on your requirement.

3. What are the limitations of React?

The few limitations of React are as given below:

- React is not a full-blown framework as it is only a library.
- The components of React are numerous and will take time to fully grasp the benefits of all.
- It might be difficult for beginner programmers to understand React.
- Coding might become complex as it will make use of inline templating and JSX.

4. What is useState() in React?

The useState() is a built-in React Hook that allows you for having state variables in functional components. It should be used when the DOM has something that is dynamically manipulating/controlling.

In the below-given example code, The useState(0) will return a tuple where the count is the first parameter that represents the counter's current state and the second parameter setCounter method will allow us to update the state of the counter.

```
...
const [count, setCounter] = useState(0);
const [otherStuffs, setOtherStuffs] = useState(...);
...
const setCount = () => {
  setCounter(count + 1);
  setOtherStuffs(...);
```

```
  ...
};
```
We can make use of setCounter() method for updating the state of count anywhere. In this example, we are using setCounter() inside the setCount function where various other things can also be done. The idea with the usage of hooks is that we will be able to keep our code more functional and avoid class-based components if they are not required.

5. What are keys in React?

A key is a special string attribute that needs to be included when using lists of elements.

Example of a list using key -

```
const ids = [1,2,3,4,5];
const listElements = ids.map((id)=>{
return(
<li key={id.toString()}>
  {id}
</li>
)
})
```
Importance of keys -

- Keys help react identify which elements were added, changed or removed.
- Keys should be given to array elements for providing a unique identity for each element.
- Without keys, React does not understand the order or uniqueness of each element.
- With keys, React has an idea of which particular element was deleted, edited, and added.
- Keys are generally used for displaying a list of data coming from an API.

***Note- Keys used within arrays should be unique among siblings. They need not be globally unique.

6. What is JSX?

JSX stands for JavaScript XML. It allows us to write HTML inside JavaScript and place them in the DOM without using functions like appendChild( ) or createElement( ).

As stated in the official docs of React, JSX provides syntactic sugar for React.createElement( ) function.

Note- We can create react applications without using JSX as well.

Let's understand **how JSX works**:

Without using JSX, we would have to create an element by the following process:

```
const text = React.createElement('p', {}, 'This is a text');
const container = React.createElement('div','{}',text );
ReactDOM.render(container,rootElement);
```

Using JSX, the above code can be simplified:

```
const container = (
<div>
  <p>This is a text</p>
</div>
);
ReactDOM.render(container,rootElement);
```

As one can see in the code above, we are directly using HTML inside JavaScript.

7. What are the differences between functional and class components?

Before the introduction of Hooks in React, functional components were called stateless components and were behind class components on a feature basis. After the introduction of Hooks, functional components are equivalent to class components.

Although functional components are the new trend, the react team insists on keeping class components in React. Therefore, it is important to know how these components differ.

On the following basis let's compare functional and class components:

- **Declaration**

Functional components are nothing but JavaScript functions and therefore can be declared using an arrow function or the function keyword:

```
function card(props){
 return(
    <div className="main-container">
      <h2>Title of the card</h2>
    </div>
  )
 }
 const card = (props) =>{
```

```
  return(
    <div className="main-container">
      <h2>Title of the card</h2>
    </div>
  )
 }
```

Class components, on the other hand, are declared using the ES6 class:

```
class Card extends React.Component{
 constructor(props){
   super(props);
 }
  render(){
    return(
      <div className="main-container">
        <h2>Title of the card</h2>
      </div>
    )
  }
 }
```

- **Handling props**

Let’s render the following component with props and analyse how functional and class components handle props:

```
<Student Info name="Vivek" rollNumber="23" />
```

In functional components, the handling of props is pretty straightforward. Any prop provided as an argument to a functional component can be directly used inside HTML elements:

```
function StudentInfo(props){
  return(
    <div className="main">
      <h2>{props.name}</h2>
      <h4>{props.rollNumber}</h4>
    </div>
  )
}
```

In the case of class components, props are handled in a different way:

```
class StudentInfo extends React.Component{
  constructor(props){
    super(props);
  }
```

```
  render(){
    return(
      <div className="main">
        <h2>{this.props.name}</h2>
        <h4>{this.props.rollNumber}</h4>
      </div>
    )
  }
}
```

As we can see in the code above, **this** keyword is used in the case of class components.

- **Handling state**

Functional components use React hooks to handle state. It uses the useState hook to set the state of a variable inside the component:

```
function ClassRoom(props){
  let [studentsCount,setStudentsCount] = useState(0);
   const addStudent = () => {
     setStudentsCount(++studentsCount);
  }
   return(
     <div>
       <p>Number of students in class room: {studentsCount}</p>
       <button onClick={addStudent}>Add Student</button>
     </div>
   )
  }
```

Since useState hook returns an array of two items, the first item contains the current state, and the second item is a function used to update the state.

In the code above, using array destructuring we have set the variable name to studentsCount with a current value of “0” and setStudentsCount is the function that is used to update the state.

For reading the state, we can see from the code above, the variable name can be directly used to read the current state of the variable.

We cannot use React Hooks inside class components, therefore state handling is done very differently in a class component:

Let’s take the same above example and convert it into a class component:

```
class ClassRoom extends React.Component{
```

```
    constructor(props){
      super(props);
      this.state = {studentsCount : 0};

      this.addStudent = this.addStudent.bind(this);
    }

      addStudent(){
      this.setState((prevState)=>{
        return {studentsCount: prevState.studentsCount++}
      });
    }

      render(){
       return(
         <div>
           <p>Number of students in class room: {this.state.studentsCount}</p>
           <button onClick={this.addStudent}>Add Student</button>
         </div>
       )
     }
    }
```

In the code above, we see we are using **this.state** to add the variable studentsCount and setting the value to “0”.

For reading the state, we are using **this.state.studentsCount**.

For updating the state, we need to first bind the addStudent function to **this**. Only then, we will be able to use the **setState** function which is used to update the state.

8. What is the virtual DOM? How does react use the virtual DOM to render the UI?

As stated by the react team, virtual DOM is a concept where a virtual representation of the real DOM is kept inside the memory and is synced with the real DOM by a library such as ReactDOM.

Why was virtual DOM introduced?

DOM manipulation is an integral part of any web application, but DOM manipulation is quite slow when compared to other operations in JavaScript. The efficiency of the application gets affected when several DOM manipulations are being done. Most JavaScript frameworks update the entire DOM even when a small part of the DOM changes.

For example, consider a list that is being rendered inside the DOM. If one of the items in the list changes, the entire list gets rendered again instead of just rendering the item that was changed/updated. This is called inefficient updating.

To address the problem of inefficient updating, the react team introduced the concept of virtual DOM.

How does it work?

For every DOM object, there is a corresponding virtual DOM object(copy), which has the same properties. The main difference between the real DOM object and the virtual DOM object is that any changes in the virtual DOM object will not reflect on the screen directly. Consider a virtual DOM object as a blueprint of the real DOM object. Whenever a JSX element gets rendered, every virtual DOM object gets updated.

**Note- One may think updating every virtual DOM object might be inefficient, but that's not the case. Updating the virtual DOM is much faster than updating the real DOM since we are just updating the blueprint of the real DOM.

React uses two virtual DOMs to render the user interface. One of them is used to store the current state of the objects and the other to store the previous state of the objects. Whenever the virtual DOM gets updated, react compares the two virtual DOMs and gets to know about which virtual DOM objects were updated. After knowing which objects were updated, react renders only those objects inside the real DOM instead of rendering the complete real DOM. This way, with the use of virtual DOM, react solves the problem of inefficient updating.

9. What are the differences between controlled and uncontrolled components?

Controlled and uncontrolled components are just different approaches to handling input from elements in react.

Feature	Uncontrolled	Controlled	Name attrs
One-time value retrieval (e.g. on submit)	✔	✔	✔

Validating on submit	✔️	✔️	✔️
Field-level Validation	❌	✔️	✔️
Conditionally disabling submit button	❌	✔️	✔️
Enforcing input format	❌	✔️	✔️
several inputs for one piece of data	❌	✔️	✔️
dynamic inputs	❌	✔️	🤔

- **Controlled component:** In a controlled component, the value of the input element is controlled by React. We store the state of the input element inside the code, and by using event-based callbacks, any changes made to the input element will be reflected in the code as well.

When a user enters data inside the input element of a controlled component, onChange function gets triggered and inside the code, we check whether the value entered is valid or invalid. If the value is valid, we change the state and re-render the input element with the new value.

Example of a controlled component:

```
function FormValidation(props) {
let [inputValue, setInputValue] = useState("");
let updateInput = e => {
  setInputValue(e.target.value);
};
return (
  <div>
    <form>
```

```
    <input type="text" value={inputValue} onChange={updateInput} />
   </form>
  </div>
);
}
```
As one can see in the code above, the value of the input element is determined by the state of the **inputValue** variable. Any changes made to the input element is handled by the **updateInput** function.

- **Uncontrolled component:** In an uncontrolled component, the value of the input element is handled by the DOM itself. Input elements inside uncontrolled components work just like normal HTML input form elements.

The state of the input element is handled by the DOM. Whenever the value of the input element is changed, event-based callbacks are not called. Basically, react does not perform any action when there are changes made to the input element.

Whenever use enters data inside the input field, the updated data is shown directly. To access the value of the input element, we can use **ref**.

Example of an uncontrolled component:

```
function FormValidation(props) {
let inputValue = React.createRef();
let handleSubmit = e => {
  alert(`Input value: ${inputValue.current.value}`);
  e.preventDefault();
};
return (
  <div>
    <form onSubmit={handleSubmit}>
      <input type="text" ref={inputValue} />
      <button type="submit">Submit</button>
    </form>
  </div>
);
}
```
As one can see in the code above, we are **not** using **onChange** function to govern the changes made to the input element. Instead, we are using **ref** to access the value of the input element.

10. What are props in React?

The props in React are the inputs to a component of React. They can be single-valued or objects having a set of values that will be passed to components of React during creation by

using a naming convention that almost looks similar to HTML-tag attributes. We can say that props are the data passed from a parent component into a child component.

The main purpose of props is to provide different component functionalities such as:

- Passing custom data to the React component.
- Using through this.props.reactProp inside render() method of the component.
- Triggering state changes.

For example, consider we are creating an element with reactProp property as given below:
<Element reactProp = "1" />
This reactProp name will be considered as a property attached to the native props object of React which already exists on each component created with the help of React library: props.reactProp;.

11. Explain React state and props.

Props	State
Immutable	Owned by its component
Has better performance	Locally scoped
Can be passed to child components	Writeable/Mutable
	has setState() method to modify properties
	Changes to state can be asynchronous
	can only be passed as props

- **React State**
  Every component in react has a built-in state object, which contains all the property values that belong to that component.
  In other words, the state object controls the behaviour of a component. Any change in the property values of the state object leads to the re-rendering of the component.

Note- State object is not available in functional components but, we can use React Hooks to add state to a functional component.

How to declare a state object?

Example:

```
class Car extends React.Component{
constructor(props){
  super(props);
  this.state = {
    brand: "BMW",
    color: "black"
  }
}
}
```

How to use and update the state object?

```
class Car extends React.Component {
constructor(props) {
  super(props);
  this.state = {
    brand: "BMW",
    color: "Black"
  };
}
changeColor() {
  this.setState(prevState => {
    return { color: "Red" };
  });
}
render() {
  return (
    <div>
      <button onClick={() => this.changeColor()}>Change Color</button>
      <p>{this.state.color}</p>
    </div>
  );
}
}
```

As one can see in the code above, we can use the state by calling **this.state.propertyName** and we can change the state object property using **setState** method.

- **React Props**

Every React component accepts a single object argument called props (which stands for “properties”). These props can be passed to a component using HTML attributes and the component accepts these props as an argument.

Using props, we can pass data from one component to another.

Passing props to a component:

While rendering a component, we can pass the props as an HTML attribute:

```
<Car brand="Mercedes"/>
```

The component receives the props:

In Class component:

```
class Car extends React.Component {
constructor(props) {
  super(props);
  this.state = {
    brand: this.props.brand,
    color: "Black"
  };
}
}
```

In Functional component:

```
function Car(props) {
let [brand, setBrand] = useState(props.brand);
}
```

Note- Props are read-only. They cannot be manipulated or changed inside a component.

12. Explain about types of side effects in React component.

There are two types of side effects in React component. They are:

- **Effects without Cleanup:** This side effect will be used in useEffect which does not restrict the browser from screen update. It also improves the responsiveness of an application. A few common examples are network requests, Logging, manual DOM mutations, etc.
- **Effects with Cleanup:** Some of the Hook effects will require the cleanup after updating of DOM is done. For example, if you want to set up an external data source subscription, it requires cleaning up the memory else there might be a problem of memory leak. It is a

known fact that React will carry out the cleanup of memory when the unmounting of components happens. But the effects will run for each render() method rather than for any specific method. Thus we can say that, before execution of the effects succeeding time the React will also cleanup effects from the preceding render.

13. What is prop drilling in React?

Sometimes while developing React applications, there is a need to pass data from a component that is higher in the hierarchy to a component that is deeply nested. To pass data between such components, we pass props from a source component and keep passing the prop to the next component in the hierarchy till we reach the deeply nested component.

The **disadvantage** of using prop drilling is that the components that should otherwise be not aware of the data have access to the data.

14. What are error boundaries?

Introduced in version 16 of React, Error boundaries provide a way for us to catch errors that occur in the render phase.

- **What is an error boundary?**

Any component which uses one of the following lifecycle methods is considered an error boundary.
In what places can an error boundary detect an error?

1. Render phase
2. Inside a lifecycle method
3. Inside the constructor

Without using error boundaries:

```
class CounterComponent extends React.Component{
constructor(props){
  super(props);
  this.state = {
    counterValue: 0
  }
  this.incrementCounter = this.incrementCounter.bind(this);
}
incrementCounter(){
  this.setState(prevState => counterValue = prevState+1);
```

```
}
render(){
  if(this.state.counter === 2){
    throw new Error('Crashed');
  }
  return(
    <div>
      <button onClick={this.incrementCounter}>Increment Value</button>
      <p>Value of counter: {this.state.counterValue}</p>
    </div>
  )
}
}
```

In the code above, when the counterValue equals 2, we throw an error inside the render method.

When we are not using the error boundary, instead of seeing an error, we see a blank page. Since any error inside the render method leads to unmounting of the component. To display an error that occurs inside the render method, we use error boundaries.

With error boundaries: As mentioned above, error boundary is a component using one or both of the following methods: **static getDerivedStateFromError and componentDidCatch.**

Let's create an error boundary to handle errors in the render phase:

```
class ErrorBoundary extends React.Component {
constructor(props) {
  super(props);
  this.state = { hasError: false };
}
static getDerivedStateFromError(error) {
  return { hasError: true };
}
 componentDidCatch(error, errorInfo) {
  logErrorToMyService(error, errorInfo);
}
render() {
  if (this.state.hasError) {
    return <h4>Something went wrong</h4>
  }
  return this.props.children;
}
}
```

In the code above, **getDerivedStateFromError** function renders the fallback UI interface when the render method has an error.

componentDidCatch logs the error information to an error tracking service.

Now with the error boundary, we can render the CounterComponent in the following way:

```
<ErrorBoundary>
 <CounterComponent/>
</ErrorBoundary>
```

15. What is React Hooks?

React Hooks are the built-in functions that permit developers for using the state and lifecycle methods within React components. These are newly added features made available in React 16.8 version. Each lifecycle of a component is having 3 phases which include mount, unmount, and update. Along with that, components have properties and states. Hooks will allow using these methods by developers for improving the reuse of code with higher flexibility navigating the component tree.

Using Hook, all features of React can be used without writing class components. ***For example***, before React version 16.8, it required a class component for managing the state of a component. But now using the useState hook, we can keep the state in a functional component.

16. Explain React Hooks.

What are Hooks? Hooks are functions that let us “hook into” React state and lifecycle features from a **functional component.**

React Hooks **cannot** be used in class components. They let us write components without class.

Why were Hooks introduced in React?

React hooks were introduced in the 16.8 version of React. Previously, functional components were called stateless components. Only class components were used for state management and lifecycle methods. The need to change a functional component to a class component, whenever state management or lifecycle methods were to be used, led to the development of Hooks.

Example of a hook: **useState hook:**

In functional components, the useState hook lets us define a state for a component:

```
function Person(props) {
// We are declaring a state variable called name.
// setName is a function to update/change the value of name
let [name, setName] = useState('');
}
```
The state variable “name” can be directly used inside the HTML.

17. What are the rules that must be followed while using React Hooks?

There are 2 rules which must be followed while you code with Hooks:

- React Hooks must be called only at the top level. It is not allowed to call them inside the nested functions, loops, or conditions.
- It is allowed to call the Hooks only from the React Function Components.

18. What is the use of useEffect React Hooks?

The useEffect React Hook is used for performing the side effects in functional components. With the help of useEffect, you will inform React that your component requires something to be done after rendering the component or after a state change. The function you have passed(can be referred to as “effect”) will be remembered by React and call afterwards the performance of DOM updates is over. Using this, we can perform various calculations such as data fetching, setting up document title, manipulating DOM directly, etc, that don’t target the output value. The useEffect hook will run by default after the first render and also after each update of the component. React will guarantee that the DOM will be updated by the time when the effect has run by it.

The useEffect React Hook will accept 2 arguments: useEffect(callback,[dependencies]);

Where the first argument callback represents the function having the logic of side-effect and it will be immediately executed after changes were being pushed to DOM. The second argument dependencies represent an optional array of dependencies. The useEffect() will execute the callback only if there is a change in dependencies in between renderings.

Example:

```
import { useEffect } from 'react';
function WelcomeGreetings({ name }) {
 const msg = `Hi, ${name}!`;     // Calculates output
 useEffect(() => {
   document.title = `Welcome to you ${name}`;    // Side-effect!
 }, [name]);
 return <div>{msg}</div>;         // Calculates output
}
```

The above code will update the document title which is considered to be a side-effect as it will not calculate the component output directly. That is why updating of document title has been placed in a callback and provided to useEffect().

Consider you don't want to execute document title update each time on rendering of WelcomeGreetings component and you want it to be executed only when the name prop changes then you need to supply name as a dependency to useEffect(callback, [name]).

19. Why do React Hooks make use of refs?

Earlier, refs were only limited to class components but now it can also be accessible in function components through the useRef Hook in React.

The refs are used for:

- Managing focus, media playback, or text selection.
- Integrating with DOM libraries by third-party.
- Triggering the imperative animations.

20. What are Custom Hooks?

A Custom Hook is a function in Javascript whose name begins with 'use' and which calls other hooks. It is a part of React v16.8 hook update and permits you for reusing the stateful logic without any need for component hierarchy restructuring.

In almost all of the cases, custom hooks are considered to be sufficient for replacing render props and HoCs (Higher-Order components) and reducing the amount of nesting required. Custom Hooks will allow you for avoiding multiple layers of abstraction or wrapper hell that might come along with Render Props and HoCs.

The **disadvantage** of Custom Hooks is it cannot be used inside of the classes.

React Interview Questions for Experienced

1. How to perform automatic redirect after login?

The react-router package will provide the component <Redirect> in React Router. Rendering of a <Redirect> component will navigate to a newer location. In the history stack, the current location will be overridden by the new location just like the server-side redirects.

```
import React, { Component } from 'react'
import { Redirect } from 'react-router'
export default class LoginDemoComponent extends Component {
 render() {
```

```
  if (this.state.isLoggedIn === true) {
    return <Redirect to="/your/redirect/page" />
  } else {
    return <div>{'Please complete login'}</div>
  }
 }
}
```

Conclusion

React has got more popularity among the top IT companies like Facebook, PayPal, Instagram, Uber, etc., around the world especially in India. Hooks is becoming a trend in the React community as it removes the state management complexities.

This article includes the most frequently asked ReactJS and React Hooks interview questions and answers that will help you in interview preparations. Also, remember that your success during the interview is not all about your technical skills, it will also be based on your state of mind and the good impression that you will make at first. All the best!!

2. How to pass data between sibling components using React router?

Passing data between sibling components of React is possible using React Router with the help of history.push and match.params.

In the code given below, we have a Parent component AppDemo.js and have two Child Components HomePage and AboutPage. Everything is kept inside a Router by using React-router Route. It is also having a route for /about/{params} where we will pass the data.

```
import React, { Component } from 'react';
class AppDemo extends Component {
render() {
 return (
   <Router>
     <div className="AppDemo">
     <ul>
       <li>
         <NavLink to="/"  activeStyle={{ color:'blue' }}>Home</NavLink>
       </li>
       <li>
         <NavLink to="/about"  activeStyle={{ color:'blue' }}>About
</NavLink>
       </li>
</ul>
           <Route path="/about/:aboutId" component={AboutPage} />
           <Route path="/about" component={AboutPage} />
```

```
            <Route path="/" component={HomePage} />
      </div>
    </Router>
  );
}
}
export default AppDemo;
```

The HomePage is a functional component with a button. On button click, we are using props.history.push(‘/about/’ + data) to programmatically navigate into /about/data.

```
export default function HomePage(props) {
 const handleClick = (data) => {
  props.history.push('/about/' + data);
 }
return (
  <div>
    <button onClick={() => handleClick('DemoButton')}>To About</button>
  </div>
)
}
```

Also, the functional component AboutPage will obtain the data passed by props.match.params.aboutId.

```
export default function AboutPage(props) {
if(!props.match.params.aboutId) {
    return <div>No Data Yet</div>
}
return (
  <div>
    {`Data obtained from HomePage is ${props.match.params.aboutId}`}
  </div>
)
}
```

After button click in the HomePage the page will look like below:

3. How to re-render the view when the browser is resized?

It is possible to listen to the resize event in **componentDidMount()** and then update the width and height dimensions. It requires the removal of the event listener in the **componentWillUnmount()** method.

Using the below-given code, we can render the view when the browser is resized.

```
class WindowSizeDimensions extends React.Component {
 constructor(props){
   super(props);
   this.updateDimension = this.updateDimension.bind(this);
 }

 componentWillMount() {
   this.updateDimension()
 }
 componentDidMount() {
   window.addEventListener('resize', this.updateDimension)
 }
 componentWillUnmount() {
   window.removeEventListener('resize', this.updateDimension)
 }
 updateDimension() {
   this.setState({width: window.innerWidth, height: window.innerHeight})
 }
 render() {
   return <span>{this.state.width} x {this.state.height}</span>
 }
}
```

4. How to create a switching component for displaying different pages?

A switching component refers to a component that will render one of the multiple components. We should use an object for mapping prop values to components.

A below-given example will show you how to display different pages based on page prop using switching component:

```
import HomePage from './HomePage'
import AboutPage from './AboutPage'
import FacilitiesPage from './FacilitiesPage'
import ContactPage from './ContactPage'
import HelpPage from './HelpPage'
const PAGES = {
 home: HomePage,
 about: AboutPage,
 facilitiess: FacilitiesPage,
 contact: ContactPage
 help: HelpPage
}
const Page = (props) => {
 const Handler = PAGES[props.page] || HelpPage
```

```
 return <Handler {...props} />
}
// The PAGES object keys can be used in the prop types for catching errors during dev-time.
Page.propTypes = {
 page: PropTypes.oneOf(Object.keys(PAGES)).isRequired
}
```

5. Explain how to create a simple React Hooks example program.

I will assume that you are having some coding knowledge about JavaScript and have installed Node on your system for creating a below given React Hook program. An installation of Node comes along with the command-line tools: npm and npx, where npm is useful to install the packages into a project and npx is useful in running commands of Node from the command line. The npx looks in the current project folder for checking whether a command has been installed there. When the command is not available on your computer, the npx will look in the npmjs.com repository, then the latest version of the command script will be loaded and will run without locally installing it. This feature is useful in creating a skeleton React application within a few key presses.

Open the Terminal inside the folder of your choice, and run the following command:

```
npx create-react-app react-items-with-hooks
```

Here, the create-react-app is an app initializer created by Facebook, to help with the easy and quick creation of React application, providing options to customize it while creating the application? The above command will create a new folder named react-items-with-hooks and it will be initialized with a basic React application. Now, you will be able to open the project in your favourite IDE. You can see an src folder inside the project along with the main application component App.js. This file is having a single function App() which will return an element and it will make use of an extended JavaScript syntax(JSX) for defining the component.

JSX will permit you for writing HTML-style template syntax directly into the JavaScript file. This mixture of JavaScript and HTML will be converted by React toolchain into pure JavaScript that will render the HTML element.

It is possible to define your own React components by writing a function that will return a JSX element. You can try this by creating a new file src/SearchItem.jsand put the following code into it.

```
import React from 'react';
export function SearchItem() {
 return (
  <div>
   <div className="search-input">
    <input type="text" placeholder="SearchItem"/>
```

```
    </div>
    <h1 className="h1">Search Results</h1>
    <div className="items">
      <table>
        <thead>
          <tr>
            <th className="itemname-col">Item Name</th>
            <th className="price-col">Price</th>
            <th className="quantity-col">Quantity</th>
          </tr>
        </thead>
        <tbody></tbody>
      </table>
    </div>
  </div>
 );
}
```

This is all about how you can create a component. It will only display the empty table and doesn't do anything. But you will be able to use the Search component in the application. Open the file src/App.js and add the import statement given below to the top of the file.

```
import { SearchItem } from './SearchItem';
```

Now, from the logo.svg, import will be removed and then contents of returned value in the function App() will be replaced with the following code:

```
<div className="App">
 <header>
   Items with Hooks
 </header>
 <SearchItem/>
</div>
```

You can notice that the element <SearchItem/> has been used just similar to an HTML element. The JSX syntax will enable for including the components in this approach directly within the JavaScript code. Your application can be tested by running the below-given command in your terminal.

```
npm start
```

This command will compile your application and open your default browser into http://localhost:4000. This command can be kept on running when code development is in progress to make sure that the application is up-to-date, and also this browser page will be reloaded each time you modify and save the code.

This application will work finely, but it doesn't look nice as it doesn't react to any input from the user. You can make it more interactive by adding a state with React Hooks, adding authentication, etc.

6. Explain conditional rendering in React.

Conditional rendering refers to the dynamic output of user interface markups based on a condition state. It works in the same way as JavaScript conditions. Using conditional rendering, it is possible to toggle specific application functions, API data rendering, hide or show elements, decide permission levels, authentication handling, and so on.

There are different approaches for implementing conditional rendering in React. Some of them are:

- Using if-else conditional logic which is suitable for smaller as well as for medium-sized applications
- Using ternary operators, which takes away some amount of complication from if-else statements
- Using element variables, which will enable us to write cleaner code.

7. Can React Hook replaces Redux?

The React Hook cannot be considered as a replacement for Redux (It is an open-source, JavaScript library useful in managing the application state) when it comes to the management of the global application state tree in large complex applications, even though the React will provide a useReducer hook that manages state transitions similar to Redux. Redux is very useful at a lower level of component hierarchy to handle the pieces of a state which are dependent on each other, instead of a declaration of multiple useState hooks.

In commercial web applications which is larger, the complexity will be high, so using only React Hook may not be sufficient. Few developers will try to tackle the challenge with the help of React Hooks and others will combine React Hooks with the Redux.

8. What is React Router?

React Router refers to the standard library used for routing in React. It permits us for building a single-page web application in React with navigation without even refreshing the page when the user navigates. It also allows to change the browser URL and will keep the user interface in sync with the URL. React Router will make use of the component structure for calling the components, using which appropriate information can be shown. Since React is a component-based framework, it's not necessary to include and use this package. Any other compatible routing library would also work with React.

The major components of React Router are given below:

- **BrowserRouter:** It is a router implementation that will make use of the HTML5 history API (pushState, popstate, and event replaceState) for keeping your UI to be in sync with the URL. It is the parent component useful in storing all other components.
- **Routes:** It is a newer component that has been introduced in the React v6 and an upgrade of the component.
- **Route:** It is considered to be a conditionally shown component and some UI will be rendered by this whenever there is a match between its path and the current URL.
- **Link:** It is useful in creating links to various routes and implementing navigation all over the application. It works similarly to the anchor tag in HTML.

9. Do Hooks cover all the functionalities provided by the classes?

Our goal is for Hooks to cover all the functionalities for classes at its earliest. There are no Hook equivalents for the following methods that are not introduced in Hooks yet:

- getSnapshotBeforeUpdate()
- getDerivedStateFromError()
- componentDidCatch()

Since it is an early time for Hooks, few third-party libraries may not be compatible with Hooks at present, but they will be added soon.

10. How does the performance of using Hooks will differ in comparison with the classes?

- React Hooks will avoid a lot of overheads such as the instance creation, binding of events, etc., that are present with classes.
- Hooks in React will result in smaller component trees since they will be avoiding the nesting that exists in HOCs (Higher Order Components) and will render props which result in less amount of work to be done by React.

11. Differentiate React Hooks vs Classes.

React Hooks	Classes
It is used in functional components of React.	It is used in class-based components of React.
It will not require a declaration of any kind of constructor.	It is necessary to declare the constructor inside the class component.

It does not require the use of this keyword in state declaration or modification.	Keyword this will be used in state declaration (this.state) and in modification (this.setState()).
It is easier to use because of the useState functionality.	No specific function is available for helping us to access the state and its corresponding setState variable.
React Hooks can be helpful in implementing Redux and context API.	Because of the long setup of state declarations, class states are generally not preferred.

12. Explain about types of Hooks in React.

There are two types of Hooks in React. They are:

1. Built-in Hooks: The built-in Hooks are divided into 2 parts as given below:

- **Basic Hooks:**
  - useState(): This functional component is used to set and retrieve the state.
  - useEffect(): It enables for performing the side effects in the functional components.
  - useContext(): It is used for creating common data that is to be accessed by the components hierarchy without having to pass the props down to each level.
- **Additional Hooks:**
  - useReducer() : It is used when there is a complex state logic that is having several sub-values or when the upcoming state is dependent on the previous state. It will also enable you to optimization of component performance that will trigger deeper updates as it is permitted to pass the dispatch down instead of callbacks.
  - useMemo() : This will be used for recomputing the memoized value when there is a change in one of the dependencies. This optimization will help for avoiding expensive calculations on each render.
  - useCallback() : This is useful while passing callbacks into the optimized child components and depends on the equality of reference for the prevention of unneeded renders.
  - useImperativeHandle(): It will enable modifying the instance that will be passed with the ref object.

    - useDebugValue(): It is used for displaying a label for custom hooks in React DevTools.
    - useRef() : It will permit creating a reference to the DOM element directly within the functional component.
    - useLayoutEffect(): It is used for the reading layout from the DOM and re-rendering synchronously.

2. Custom Hooks: A custom Hook is basically a function of JavaScript. The Custom Hook working is similar to a regular function. The “use” at the beginning of the Custom Hook Name is required for React to understand that this is a custom Hook and also it will describe that this specific function follows the rules of Hooks. Moreover, developing custom Hooks will enable you for extracting component logic from within reusable functions.

13. Does React Hook work with static typing?

Static typing refers to the process of code check during the time of compilation for ensuring all variables will be statically typed. React Hooks are functions that are designed to make sure about all attributes must be statically typed. For enforcing stricter static typing within our code, we can make use of the React API with custom Hooks.

14. What are the lifecycle methods of React?

React lifecycle hooks will have the methods that will be automatically called at different phases in the component lifecycle and thus it provides good control over what happens at the invoked point. It provides the power to effectively control and manipulate what goes on throughout the component lifecycle.

For example, if you are developing the YouTube application, then the application will make use of a network for buffering the videos and it consumes the power of the battery (assume only these two). After playing the video if the user switches to any other application, then you should make sure that the resources like network and battery are being used most efficiently. You can stop or pause the video buffering which in turn stops the battery and network usage when the user switches to another application after video play.

So we can say that the developer will be able to produce a quality application with the help of lifecycle methods and it also helps developers to make sure to plan what and how to do it at different points of birth, growth, or death of user interfaces.

The various lifecycle methods are:

- constructor(): This method will be called when the component is initiated before anything has been done. It helps to set up the initial state and initial values.

- getDerivedStateFromProps(): This method will be called just before element(s) rendering in the DOM. It helps to set up the state object depending on the initial props. The getDerivedStateFromProps() method will have a state as an argument and it returns an object that made changes to the state. This will be the first method to be called on an updating of a component.
- render(): This method will output or re-render the HTML to the DOM with new changes. The render() method is an essential method and will be called always while the remaining methods are optional and will be called only if they are defined.
- componentDidMount(): This method will be called after the rendering of the component. Using this method, you can run statements that need the component to be already kept in the DOM.
- shouldComponentUpdate(): The Boolean value will be returned by this method which will specify whether React should proceed further with the rendering or not. The default value for this method will be True.
- getSnapshotBeforeUpdate(): This method will provide access for the props as well as for the state before the update. It is possible to check the previously present value before the update, even after the update.
- componentDidUpdate(): This method will be called after the component has been updated in the DOM.
- componentWillUnmount(): This method will be called when the component removal from the DOM is about to happen.

15. What are the different phases of the component lifecycle?

There are four different phases in the lifecycle of React component. They are:

- **Initialization:** During this phase, React component will prepare by setting up the default props and initial state for the upcoming tough journey.
- **Mounting:** Mounting refers to putting the elements into the browser DOM. Since React uses VirtualDOM, the entire browser DOM which has been currently rendered would not be refreshed. This phase includes the lifecycle methods componentWillMount and componentDidMount.
- **Updating:** In this phase, a component will be updated when there is a change in the state or props of a component. This phase will have lifecycle methods like componentWillUpdate, shouldComponentUpdate, render, and componentDidUpdate.
- **Unmounting:** In this last phase of the component lifecycle, the component will be removed from the DOM or will be unmounted from the browser DOM. This phase will have the lifecycle method named componentWillUnmount.

16. What are Higher Order Components?

Simply put, Higher-Order Component(HOC) is a function that takes in a component and returns a new component.

When do we need a Higher Order Component?

While developing React applications, we might develop components that are quite similar to each other with minute differences. In most cases, developing similar components might not be an issue but, while developing larger applications we need to keep our code **DRY**, therefore, we want an **abstraction** that allows us to define this logic in a single place and share it across components. HOC allows us to create that abstraction.

Example of a HOC:

Consider the following components having similar functionality. The following component displays the list of articles:

```
// "GlobalDataSource" is some global data source
class ArticlesList extends React.Component {
 constructor(props) {
  super(props);
  this.handleChange = this.handleChange.bind(this);
  this.state = {
    articles: GlobalDataSource.getArticles(),
  };
}
 componentDidMount() {
  // Listens to the changes added
  GlobalDataSource.addChangeListener(this.handleChange);
}
 componentWillUnmount() {
  // Listens to the changes removed
  GlobalDataSource.removeChangeListener(this.handleChange);
}
 handleChange() {
  // States gets Update whenver data source changes
  this.setState({
    articles: GlobalDataSource.getArticles(),
  });
}
 render() {
  return (
    <div>
      {this.state.articles.map((article) => (
        <ArticleData article={article} key={article.id} />
      ))}
```

```
    </div>
  );
 }
}
```

The following component displays the list of users:

```
// "GlobalDataSource" is some global data source
class UsersList extends React.Component {
 constructor(props) {
  super(props);
  this.handleChange = this.handleChange.bind(this);
  this.state = {
    users: GlobalDataSource.getUsers(),
  };
 }
 componentDidMount() {
  // Listens to the changes added
  GlobalDataSource.addChangeListener(this.handleChange);
 }
 componentWillUnmount() {
  // Listens to the changes removed
  GlobalDataSource.removeChangeListener(this.handleChange);
 }
 handleChange() {
  // States gets Update whenver data source changes
  this.setState({
    users: GlobalDataSource.getUsers(),
  });
 }
 render() {
  return (
    <div>
      {this.state.users.map((user) => (
        <UserData user={user} key={user.id} />
      ))}
    </div>
  );
 }
}
```

Notice the above components, both have similar functionality but, they are calling different methods to an API endpoint.

Let’s create a Higher Order Component to create an abstraction:

```
// Higher Order Component which takes a component
// as input and returns another component
// "GlobalDataSource" is some global data source
function HOC(WrappedComponent, selectData) {
 return class extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {
      data: selectData(GlobalDataSource, props),
    };
  }
  componentDidMount() {
    // Listens to the changes added
    GlobalDataSource.addChangeListener(this.handleChange);
  }
  componentWillUnmount() {
    // Listens to the changes removed
    GlobalDataSource.removeChangeListener(this.handleChange);
  }
  handleChange() {
    this.setState({
      data: selectData(GlobalDataSource, this.props),
    });
  }
  render() {
    // Rendering the wrapped component with the latest data data
    return <WrappedComponent data={this.state.data} {...this.props} />;
  }
 };
}
```

We know HOC is a function that takes in a component and returns a component.

In the code above, we have created a function called HOC which returns a component and performs functionality that can be shared across both the **ArticlesList** component and **UsersList** Component.

The second parameter in the HOC function is the function that calls the method on the API endpoint.

We have reduced the duplicated code of the **componentDidUpdate** and **componentDidMount** functions.

Using the concept of Higher-Order Components, we can now render the **ArticlesList** and **UsersList** components in the following way:

```
const ArticlesListWithHOC = HOC(ArticlesList, (GlobalDataSource) =>
GlobalDataSource.getArticles());
const UsersListWithHOC = HOC(UsersList, (GlobalDataSource) =>
GlobalDataSource.getUsers());
```

Remember, we are not trying to change the functionality of each component, we are trying to share a single functionality across multiple components using HOC.

17. How to pass data between react components?

Parent Component to Child Component (using props)

With the help of props, we can send data from a parent to a child component.

How do we do this?

Consider the following Parent Component:

```
import ChildComponent from "./Child";
  function ParentComponent(props) {
   let [counter, setCounter] = useState(0);

   let increment = () => setCounter(++counter);

   return (
     <div>
       <button onClick={increment}>Increment Counter</button>
       <ChildComponent counterValue={counter} />
     </div>
   );
  }
```

As one can see in the code above, we are rendering the child component inside the parent component, by providing a prop called counterValue. The value of the counter is being passed from the parent to the child component.

We can use the data passed by the parent component in the following way:

```
function ChildComponent(props) {
return (
 <div>
```

```
  <p>Value of counter: {props.counterValue}</p>
 </div>
);
}
```
We use the **props.counterValue** to display the data passed on by the parent component.

Child Component to Parent Component (using callbacks)

This one is a bit tricky. We follow the steps below:

- Create a callback in the parent component which takes in the data needed as a parameter.
- Pass this callback as a prop to the child component.
- Send data from the child component using the callback.

We are considering the same example above but in this case, we are going to pass the updated counterValue from child to parent.

Step1 and Step2: Create a callback in the parent component, pass this callback as a prop.

```
function ParentComponent(props) {
let [counter, setCounter] = useState(0);
let callback = valueFromChild => setCounter(valueFromChild);
return (
 <div>
  <p>Value of counter: {counter}</p>
  <ChildComponent callbackFunc={callback} counterValue={counter} />
 </div>
);
}
```
As one can see in the code above, we created a function called callback which takes in the data received from the child component as a parameter.

Next, we passed the function callback as a prop to the child component.

Step3: Pass data from the child to the parent component.

```
function ChildComponent(props) {
let childCounterValue = props.counterValue;
return (
 <div>
  <button onClick={() => props.callbackFunc(++childCounterValue)}>
```

```
    Increment Counter
   </button>
  </div>
);
}
```

In the code above, we have used the props.counterValue and set it to a variable called childCounterValue.

Next, on button click, we pass the incremented childCounterValue to the **props.callbackFunc**.

This way, we can pass data from the child to the parent component.

18. Name a few techniques to optimize React app performance.

There are many ways through which one can optimize the performance of a React app, let's have a look at some of them:

- **Using useMemo( )** -
  - It is a React hook that is used for caching CPU-Expensive functions.
  - Sometimes in a React app, a CPU-Expensive function gets called repeatedly due to re-renders of a component, which can lead to slow rendering. useMemo( ) hook can be used to cache such functions. By using useMemo( ), the CPU-Expensive function gets called only when it is needed.
- **Using React.PureComponent -**
  - It is a base component class that checks the state and props of a component to know whether the component should be updated.
  - Instead of using the simple React.Component, we can use React.PureComponent to reduce the re-renders of a component unnecessarily.
- **Maintaining State Colocation -**
  - This is a process of moving the state as close to where you need it as possible.
  - Sometimes in React app, we have a lot of unnecessary states inside the parent component which makes the code less readable and harder to maintain. Not to forget, having many states inside a single component leads to unnecessary re-renders for the component.
  - It is better to shift states which are less valuable to the parent component, to a separate component.
- **Lazy Loading -**
  - It is a technique used to reduce the load time of a React app. Lazy loading helps reduce the risk of web app performances to a minimum.

19. What are the different ways to style a React component?

There are many different ways through which one can style a React component. Some of the ways are :

- **Inline Styling:** We can directly style an element using inline style attributes. Make sure the value of style is a JavaScript object:

```
class RandomComponent extends React.Component {
 render() {
  return (
    <div>
      <h3 style={{ color: "Yellow" }}>This is a heading</h3>
      <p style={{ fontSize: "32px" }}>This is a paragraph</p>
    </div>
  );
 }
}
```

- **Using JavaScript object:** We can create a separate JavaScript object and set the desired style properties. This object can be used as the value of the inline style attribute.

```
class RandomComponent extends React.Component {
 paragraphStyles = {
  color: "Red",
  fontSize: "32px"
 };
 headingStyles = {
  color: "blue",
  fontSize: "48px"
 };
 render() {
  return (
    <div>
      <h3 style={this.headingStyles}>This is a heading</h3>
      <p style={this.paragraphStyles}>This is a paragraph</p>
    </div>
  );
 }
}
```

- **CSS Stylesheet:** We can create a separate CSS file and write all the styles for the component inside that file. This file needs to be imported inside the component file.

```
import './RandomComponent.css';
class RandomComponent extends React.Component {
 render() {
  return (
    <div>
      <h3 className="heading">This is a heading</h3>
```

```
      <p className="paragraph">This is a paragraph</p>
    </div>
  );
 }
}
```

- **CSS Modules:** We can create a separate CSS module and import this module inside our component. Create a file with ".module.css"' extension, styles.module.css:

```
.paragraph{
 color:"red";
 border:1px solid black;
}
```

We can import this file inside the component and use it:

```
import styles from  './styles.module.css';
class RandomComponent extends React.Component {
 render() {
  return (
    <div>
      <h3 className="heading">This is a heading</h3>
      <p className={styles.paragraph} >This is a paragraph</p>
    </div>
  );
 }
}
```

20. How to prevent re-renders in React?

- **Reason for re-renders in React:**
  - Re-rendering of a component and its child components occur when props or the state of the component has been changed.
  - Re-rendering components that are not updated, affects the performance of an application.
- **How to prevent re-rendering:**

Consider the following components:

```
class Parent extends React.Component {
state = { messageDisplayed: false };
componentDidMount() {
  this.setState({ messageDisplayed: true });
}
render() {
  console.log("Parent is getting rendered");
```

```
  return (
    <div className="App">
      <Message />
    </div>
  );
}
}
class Message extends React.Component {
constructor(props) {
  super(props);
  this.state = { message: "Hello, this is vivek" };
}
render() {
  console.log("Message is getting rendered");
  return (
    <div>
      <p>{this.state.message}</p>
    </div>
  );
}
}
```

- The **Parent** component is the parent component and the **Message** is the child component. Any change in the parent component will lead to re-rendering of the child component as well. To prevent the re-rendering of child components, we use the shouldComponentUpdate( ) method:

**Note- Use shouldComponentUpdate( ) method only when you are sure that it's a static component.

```
class Message extends React.Component {
constructor(props) {
  super(props);
  this.state = { message: "Hello, this is vivek" };
}
shouldComponentUpdate() {
  console.log("Does not get rendered");
  return false;
}
render() {
  console.log("Message is getting rendered");
  return (
    <div>
      <p>{this.state.message}</p>
```

```
   </div>
  );
}
}
```
As one can see in the code above, we have returned **false** from the shouldComponentUpdate( ) method, which prevents the child component from re-rendering.

21. Explain Strict Mode in React.

StrictMode is a tool added in **version 16.3** of React to highlight potential problems in an application. It performs additional checks on the application.

```
function App() {
 return (
   <React.StrictMode>
     <div classname="App">
       <Header/>
       <div>
         Page Content
       </div>
       <Footer/>
     </div>
   </React.StrictMode>
 );
}
```
To enable StrictMode, <React.StrictMode> tags need to be added inside the application:

```
import React from "react";
import ReactDOM from "react-dom";
import App from "./App";
const rootElement = document.getElementById("root");
ReactDOM.render(
<React.StrictMode>
  <App />
</React.StrictMode>,
rootElement
);
```
StrictMode currently helps with the following issues:

- **Identifying components with unsafe lifecycle methods:**
    - Certain lifecycle methods are unsafe to use in asynchronous react applications. With the use of third-party libraries, it becomes difficult to ensure that certain lifecycle methods are not used.

    - StrictMode helps in providing us with a warning if any of the class components use an unsafe lifecycle method.
- **Warning about the usage of legacy string API:**
    - If one is using an older version of React, **callback ref** is the recommended way to manage **refs** instead of using the **string refs**. StrictMode gives a warning if we are using **string refs** to manage refs.
- **Warning about the usage of findDOMNode:**
    - Previously, findDOMNode( ) method was used to search the tree of a DOM node. This method is deprecated in React. Hence, the StrictMode gives us a warning about the usage of this method.
- **Warning about the usage of legacy context API (because the API is error-prone).**

1. What is Node.js and how it works?

Node.js is a virtual machine that uses JavaScript as its scripting language and runs Chrome's V8 JavaScript engine. Basically, Node.js is based on an event-driven architecture where I/O runs asynchronously making it lightweight and efficient. It is being used in developing desktop applications as well with a popular framework called electron as it provides API to access OS-level features such as file system, network, etc.

2. What tools can be used to assure consistent code style?

ESLint can be used with any IDE to ensure a consistent coding style which further helps in maintaining the codebase.

3. What is a first class function in Javascript?

When functions can be treated like any other variable then those functions are first-class functions. There are many other programming languages, for example, scala, Haskell, etc which follow this including JS. Now because of this function can be passed as a param to another function(callback) or a function can return another function(higher-order function). map() and filter() are higher-order functions that are popularly used.

---

4. How do you manage packages in your node.js project?

It can be managed by a number of package installers and their configuration file accordingly. Out of them mostly use npm or yarn. Both provide almost all libraries of javascript with extended features of controlling environment-specific configurations. To maintain versions of libs being installed in a project we use package.json and package-lock.json so that there is no issue in porting that app to a different environment.

5. How is Node.js better than other frameworks most popularly used?

- Node.js provides simplicity in development because of its non-blocking I/O and event-based model results in short response time and concurrent processing, unlike

other frameworks where developers have to use thread management.

- It runs on a chrome v8 engine which is written in c++ and is highly performant with constant improvement.

- Also since we will use Javascript in both the frontend and backend the development will be much faster.

6. Explain the steps how “Control Flow” controls the functions calls?

- Control the order of execution
- Collect data
- Limit concurrency
- Call the following step in the program.

7. What are some commonly used timing features of Node.js?

- setTimeout/clearTimeout – This is used to implement delays in code execution.
- setInterval/clearInterval – This is used to run a code block multiple times.
- setImmediate/clearImmediate – Any function passed as the setImmediate() argument is a callback that's executed in the next iteration of the event loop.
- process.nextTick – Both setImmediate and process.nextTick appear to be doing the same thing; however, you may prefer one over the other depending on your callback’s urgency.

8. What are the advantages of using promises instead of callbacks?

The main advantage of using promise is you get an object to decide the action that needs to be taken after the async task completes. This gives more manageable code and avoids callback hell.

9. What is fork in node JS?

A fork in general is used to spawn child processes. In node it is used to create a new instance of v8 engine to run multiple workers to execute the code.

10. Why is Node.js single-threaded?

Node.js was created explicitly as an experiment in async processing. This was to try a new theory of doing async processing on a single thread over the existing thread-based implementation of scaling via different frameworks.

11. How do you create a simple server in Node.js that returns Hello World?

```
var http = require("http");
http.createServer(function (request, response) {
  response.writeHead(200, {'Content-Type': 'text/plain'});
  response.end('Hello World\n');
}).listen(3000);
```

12. How many types of API functions are there in Node.js?

There are two types of API functions:

- Asynchronous, non-blocking functions - mostly I/O operations which can be fork out of the main loop.
- Synchronous, blocking functions - mostly operations that influence the process running in the main loop.

13. What is REPL?

PL in Node.js stands for Read, Eval, Print, and Loop, which further means evaluating code on the go.

14. List down the two arguments that async.queue takes as input?

- Task Function
- Concurrency Value

15. What is the purpose of module.exports?

This is used to expose functions of a particular module or file to be used elsewhere in the project. This can be used to encapsulate all similar functions in a file which further improves the project structure.

For example, you have a file for all utils functions with util to get solutions in a different programming language of a problem statement.

```
const getSolutionInJavaScript = async ({
 problem_id
}) => {
...
};
const getSolutionInPython = async ({
 problem_id
}) => {
...
};
module.exports = { getSolutionInJavaScript, getSolutionInPython }
```

Thus using module.exports we can use these functions in some other file:

```
const { getSolutionInJavaScript, getSolutionInPython} = require("./utils")
```

Intermediate Node.js Interview Questions

1. Explain the concept of stub in Node.js?

Stubs are used in writing tests which are an important part of development. It replaces the whole function which is getting tested.

This helps in scenarios where we need to test:

- External calls which make tests slow and difficult to write (e.g HTTP calls/ DB calls)
- Triggering different outcomes for a piece of code (e.g. what happens if an error is thrown/ if it passes)

For example, this is the function:

```
const request = require('request');
const getPhotosByAlbumId = (id) => {
const requestUrl = `https://jsonplaceholder.typicode.com/albums/${id}/photos?_limit=3`;
return new Promise((resolve, reject) => {
    request.get(requestUrl, (err, res, body) => {
        if (err) {
            return reject(err);
```

```
        }
        resolve(JSON.parse(body));
    });
});
};
module.exports = getPhotosByAlbumId;
To test this function this is the stub
const expect = require('chai').expect;
const request = require('request');
const sinon = require('sinon');
const getPhotosByAlbumId = require('./index');
describe('with Stub: getPhotosByAlbumId', () => {
before(() => {
    sinon.stub(request, 'get')
        .yields(null, null, JSON.stringify([
            {
                "albumId": 1,
                "id": 1,
                "title": "A real photo 1",
                "url": "https://via.placeholder.com/600/92c952",
                "thumbnailUrl": "https://via.placeholder.com/150/92c952"
            },
            {
                "albumId": 1,
                "id": 2,
                "title": "A real photo 2",
                "url": "https://via.placeholder.com/600/771796",
                "thumbnailUrl": "https://via.placeholder.com/150/771796"
            },
            {
                "albumId": 1,
                "id": 3,
                "title": "A real photo 3",
                "url": "https://via.placeholder.com/600/24f355",
                "thumbnailUrl": "https://via.placeholder.com/150/24f355"
            }
        ]));
});
after(() => {
    request.get.restore();
});
it('should getPhotosByAlbumId', (done) => {
    getPhotosByAlbumId(1).then((photos) => {
        expect(photos.length).to.equal(3);
```

```
        photos.forEach(photo => {
            expect(photo).to.have.property('id');
            expect(photo).to.have.property('title');
            expect(photo).to.have.property('url');
        });
        done();
    });
});
});
```

2. Describe the exit codes of Node.js?

Exit codes give us an idea of how a process got terminated/the reason behind termination.

A few of them are:

- Uncaught fatal exception - (code - 1) - There has been an exception that is not handled
- Unused - (code - 2) - This is reserved by bash
- Fatal Error - (code - 5) - There has been an error in V8 with stderr output of the description
- Internal Exception handler Run-time failure - (code - 7) - There has been an exception when bootstrapping function was called
- Internal JavaScript Evaluation Failure - (code - 4) - There has been an exception when the bootstrapping process failed to return function value when evaluated.

3. For Node.js, why Google uses V8 engine?

Well, are there any other options available? Yes, of course, we have Spidermonkey from Firefox, Chakra from Edge but Google's v8 is the most evolved(since it's open-source so there's a huge community helping in developing features and fixing bugs) and fastest(since it's written in c++) we got till now as a JavaScript and WebAssembly engine. And it is portable to almost every machine known.

4. Why should you separate Express app and server?

The server is responsible for initializing the routes, middleware, and other application logic whereas the app has all the business logic which will be served by the routes initiated by the server. This ensures that the business logic is encapsulated and decoupled from the application logic which makes the project more readable and maintainable.

5. Explain what a Reactor Pattern in Node.js?

Reactor pattern again a pattern for nonblocking I/O operations. But in general, this is used in any event-driven architecture.

There are two components in this: 1. Reactor 2. Handler.

Reactor: Its job is to dispatch the I/O event to appropriate handlers
Handler: Its job is to actually work on those events

6. What is middleware?

Middleware comes in between your request and business logic. It is mainly used to capture logs and enable rate limit, routing, authentication, basically whatever that is not a part of business logic. There are third-party middleware also such as body-parser and you can write your own middleware for a specific use case.

7. What are node.js buffers?

In general, buffers is a temporary memory that is mainly used by stream to hold on to some data until consumed. Buffers are introduced with additional use cases than JavaScript's Unit8Array and are mainly used to represent a fixed-length sequence of bytes. This also supports legacy encodings like ASCII, utf-8, etc. It is a fixed(non-resizable) allocated memory outside the v8.

8. What is node.js streams?

Streams are instances of EventEmitter which can be used to work with streaming data in Node.js. They can be used for handling and manipulating streaming large files(videos, mp3, etc) over the network. They use buffers as their temporary storage.

There are mainly four types of the stream:

- Writable: streams to which data can be written (for example, fs.createWriteStream()).
- Readable: streams from which data can be read (for example, fs.createReadStream()).
- Duplex: streams that are both Readable and Writable (for example, net.Socket).

- Transform: Duplex streams that can modify or transform the data as it is written and read (for example, zlib.createDeflate()).

9. How can we use async await in node.js?

Here is an example of using async-await pattern:

```
// this code is to retry with exponential backoff
function wait (timeout) {
 return new Promise((resolve) => {
setTimeout(() => {
  resolve()
}, timeout);
 });
}
async function requestWithRetry (url) {
 const MAX_RETRIES = 10;
 for (let i = 0; i <= MAX_RETRIES; i++) {
try {
  return await request(url);
} catch (err) {
  const timeout = Math.pow(2, i);
  console.log('Waiting', timeout, 'ms');
  await wait(timeout);
  console.log('Retrying', err.message, i);
}
 }
}
```

10. How does Node.js overcome the problem of blocking of I/O operations?

Since the node has an event loop that can be used to handle all the I/O operations in an asynchronous manner without blocking the main function.

So for example, if some network call needs to happen it will be scheduled in the event loop instead of the main thread(single thread). And if there are multiple such I/O calls each one will be queued accordingly to be executed separately(other than the main thread).

Thus even though we have single-threaded JS, I/O ops are handled in a nonblocking way.

11. Differentiate between process.nextTick() and setImmediate()?

Both can be used to switch to an asynchronous mode of operation by listener functions.

process.nextTick() sets the callback to execute but setImmediate pushes the callback in the queue to be executed. So the event loop runs in the following manner

timers–>pending callbacks–>idle,prepare–>connections(poll,data,etc)–>check–>close callbacks

In this process.nextTick() method adds the callback function to the start of the next event queue and setImmediate() method to place the function in the check phase of the next event queue.

12. If Node.js is single threaded then how does it handle concurrency?

The main loop is single-threaded and all async calls are managed by libuv library.

For example:

```
const crypto = require("crypto");
const start = Date.now();
function logHashTime() {
 crypto.pbkdf2("a", "b", 100000, 512, "sha512", () => {
console.log("Hash: ", Date.now() - start);
 });
}
logHashTime();
logHashTime();
logHashTime();
logHashTime();
```

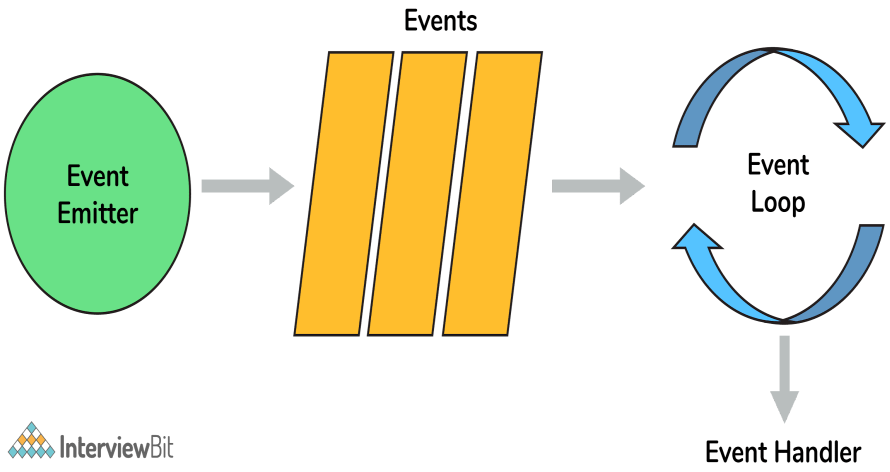
This gives the output:

```
Hash: 1213
Hash: 1225
Hash: 1212
Hash: 1222
```

This is because libuv sets up a thread pool to handle such concurrency. How many threads will be there in the thread pool depends upon the number of cores but you can override this.

13. What is an event-loop in Node JS?

Whatever that is async is managed by event-loop using a queue and listener. We can get the idea using the following diagram:



So when an async function needs to be executed(or I/O) the main thread sends it to a different thread allowing v8 to keep executing the main code. Event loop involves different phases with specific tasks such as timers, pending callbacks, idle or prepare, poll, check, close callbacks with different FIFO queues. Also in between iterations it checks for async I/O or timers and shuts down cleanly if there aren't any.

14. What do you understand by callback hell?

```
async_A(function(){
  async_B(function(){
    async_C(function(){
      async_D(function(){
        ....
        });
    });
  });
});
```

For the above example, we are passing callback functions and it makes the code unreadable and not maintainable, thus we should change the async logic to avoid this.

Advanced Node.js Interview Questions

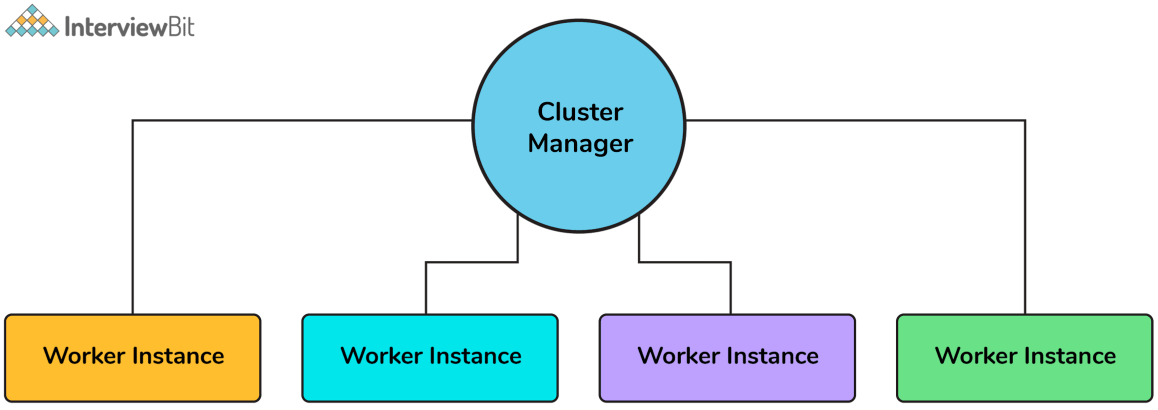
1. What is an Event Emitter in Node.js?

EventEmitter is a Node.js class that includes all the objects that are basically capable of emitting events. This can be done by attaching named events that are emitted by the object using an eventEmitter.on() function. Thus whenever this object throws an even the attached functions are invoked synchronously.

```
const EventEmitter = require('events');
class MyEmitter extends EventEmitter {}
const myEmitter = new MyEmitter();
myEmitter.on('event', () => {
 console.log('an event occurred!');
});
myEmitter.emit('event');
```

2. Enhancing Node.js performance through clustering.

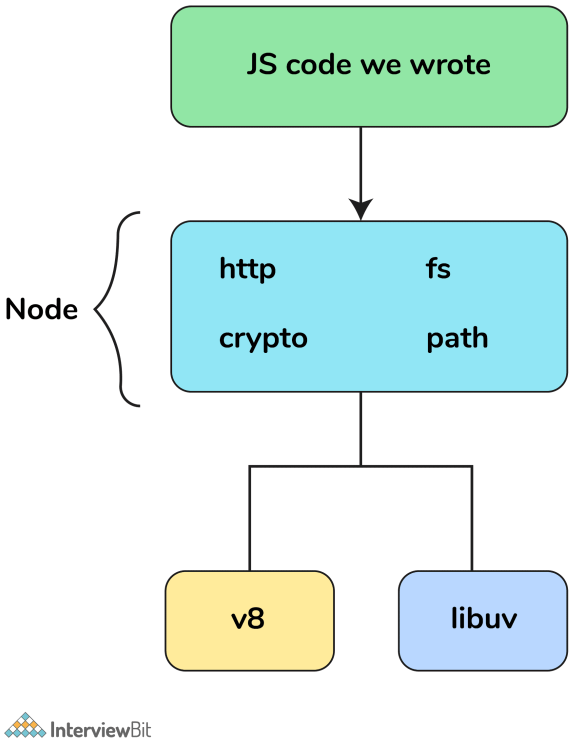
Node.js applications run on a single processor, which means that by default they don't take advantage of a multiple-core system. Cluster mode is used to start up multiple node.js processes thereby having multiple instances of the event loop. When we start using cluster in a nodejs app behind the scene multiple node.js processes are created but there is also a parent process called the cluster manager which is responsible for monitoring the health of the individual instances of our application.



Clustering in Node.js

3. What is a thread pool and which library handles it in Node.js

The Thread pool is handled by the libuv library. libuv is a multi-platform C library that provides support for asynchronous I/O-based operations such as file systems, networking, and concurrency.



4. What is WASI and why is it being introduced?

Web assembly provides an implementation of WebAssembly System Interface specification through WASI API in node.js implemented using WASI class. The introduction of WASI was done by keeping in mind its possible to use the underlying operating system via a collection of POSIX-like functions thus further enabling the application to use resources more efficiently and features that require system-level access.

5. How are worker threads different from clusters?

Cluster:

- There is one process on each CPU with an IPC to communicate.
- In case we want to have multiple servers accepting HTTP requests via a single port, clusters can be helpful.
- The processes are spawned in each CPU thus will have separate memory and node instance which further will lead to memory issues.

Worker threads:

- There is only one process in total with multiple threads.
- Each thread has one Node instance (one event loop, one JS engine) with most of the APIs accessible.
- Shares memory with other threads (e.g. SharedArrayBuffer)
- This can be used for CPU-intensive tasks like processing data or accessing the file system since NodeJS is single-threaded, synchronous tasks can be made more efficient leveraging the worker's threads.

6. How to measure the duration of async operations?

Performance API provides us with tools to figure out the necessary performance metrics. A simple example would be using async_hooks and perf_hooks

```
'use strict';
const async_hooks = require('async_hooks');
const {
 performance,
 PerformanceObserver
} = require('perf_hooks');
const set = new Set();
const hook = async_hooks.createHook({
 init(id, type) {
if (type === 'Timeout') {
  performance.mark(`Timeout-${id}-Init`);
  set.add(id);
}
 },
 destroy(id) {
if (set.has(id)) {
  set.delete(id);
  performance.mark(`Timeout-${id}-Destroy`);
  performance.measure(`Timeout-${id}`,
```

```
                `Timeout-${id}-Init`,
                `Timeout-${id}-Destroy`);
}
 }
});
hook.enable();
const obs = new PerformanceObserver((list, observer) => {
 console.log(list.getEntries()[0]);
 performance.clearMarks();
 observer.disconnect();
});
obs.observe({ entryTypes: ['measure'], buffered: true });
setTimeout(() => {}, 1000);
```

This would give us the exact time it took to execute the callback.

7. How to measure the performance of async operations?

Performance API provides us with tools to figure out the necessary performance metrics.

A simple example would be:

```
const { PerformanceObserver, performance } = require('perf_hooks');
const obs = new PerformanceObserver((items) => {
 console.log(items.getEntries()[0].duration);
 performance.clearMarks();
});
obs.observe({ entryTypes: ['measure'] });
performance.measure('Start to Now');
performance.mark('A');
doSomeLongRunningProcess(() => {
 performance.measure('A to Now', 'A');
 performance.mark('B');
 performance.measure('A to B', 'A', 'B');
});
```

What Is MongoDB?

MongoDB is a popular open-source, NoSQL (non-relational) database management system that is created to store, retrieve, and manage data flexibly and scalable. MongoDB is classified as a document database, storing data in a format similar to JSON (JavaScript Object Notation) documents.

1. Document-Oriented: MongoDB stores data in collections that contain documents. Each document is a JSON-like object, and these documents can have varying structures within the same collection. This flexibility makes it well-suited for handling data with dynamic or evolving schemas.
2. Schema-less: Unlike traditional relational databases, MongoDB doesn't require a predefined schema for data. You can insert documents with different fields in the same collection without altering the schema.
3. Scalability: MongoDB is designed for horizontal scalability. You can distribute data across multiple servers and clusters to handle large volumes of data and high traffic loads.
4. High Performance: MongoDB can provide high read and write throughput, especially for certain types of applications where rapid data access is critical.
5. Rich Query Language: MongoDB supports a powerful query language for retrieving and manipulating data. You can perform complex queries, indexing, and aggregation operations.
6. Geospatial Data: MongoDB has built-in support for geospatial data and allows you to perform geospatial queries, making it suitable for location-based applications.
7. Replication and High Availability: MongoDB supports replication for data redundancy and high availability. It can automatically recover from hardware failures and provide continuous service.
8. Flexible Indexing: You can create custom indexes to optimize query performance for specific use cases.
9. Community and Enterprise Editions: MongoDB provides a freely available Community Edition and a premium Enterprise Edition, which includes extra functionalities and comprehensive support.
10. Large Ecosystem: MongoDB boasts a thriving and engaged community, comprehensive documentation, and diverse drivers and connectors tailored to numerous programming languages and frameworks.

MongoDB is commonly used in web and mobile applications, content management systems, real-time analytics, and other scenarios where flexibility, scalability, and speed are essential. It's a popular choice for developers and organizations looking to work with data that doesn't fit neatly into traditional relational databases. Now, let’s look at the most popular MongoDB Interview Questions and Answers for 2024.

MongoDB Basic Interview Questions

1. How does MongoDB differ from traditional relational databases?

- MongoDB is a NoSQL database, while traditional relational databases are SQL-based.
- It stores data in flexible, schema-less documents, whereas relational databases use structured tables with fixed schemas.
- It is designed for horizontal scalability and can handle large volumes of data, while relational databases typically scale vertically.

2. Can you explain what a document in MongoDB is?

A document is a JSON-like data structure that stores and represents data. It can contain key-value pairs, arrays, and nested documents. Documents are stored in collections, equivalent to tables in relational databases.

3. What is a collection in MongoDB?

A collection in MongoDB is a grouping of documents. Collections are schema-less, meaning documents in the same collection can have different structures. Collections are similar to tables in traditional relational databases.

4. How does MongoDB store data?

MongoDB stores data in BSON (Binary JSON) format, a binary-encoded serialization of JSON-like documents. These documents are stored in collections within databases.

5. What is a primary key in MongoDB?

In MongoDB, the `_id` field serves as the primary key for a document. It must be unique within a collection and is automatically generated if not provided during document insertion.

6. Can you explain the concept of sharding in MongoDB?

Sharding in MongoDB is a strategy used to distribute data horizontally across numerous servers or clusters, efficiently managing extensive datasets and heavy workloads. In this approach, data is divided into distinct subsets known as shards, and MongoDB's query router directs queries to the relevant shard as needed.

7. What are indexes in MongoDB?

MongoDB employs data structures known as indexes to enhance query performance, enabling the database to swiftly locate documents according to the indexed fields. MongoDB offers support for a range of index types.

8. How do you create a database in MongoDB?

You create a database implicitly by switching to it or explicitly by running the `use <database_name>` command in the MongoDB shell. When you insert data into it, MongoDB will create the database if it doesn't already exist.

9. How do you insert data into a MongoDB collection?

You can insert data into a MongoDB collection using the `insertOne()` or `insertMany()` method. You provide a document or an array of documents to be inserted.

10. What is a replica set in MongoDB?

It is a group of servers that maintain the same data. It provides data redundancy and high availability. One server acts as the primary, while others are secondary servers that replicate data from the primary.

11. What are the data types supported by MongoDB?

MongoDB supports various data types, including string, number, boolean, date, array, object, null, regex, and more. It also helps geospatial and binary data types.

12. How do you update documents in MongoDB?

You can update documents in MongoDB using methods like `updateOne()`, `updateMany(),` or `findOneAndUpdate().` You specify the query to select the documents to update and provide an update operation.

13. What is the role of `_id` in MongoDB documents?

The `_id` field uniquely identifies each document in a collection. MongoDB uses it as the primary key, and if not provided during document insertion, MongoDB generates a unique value for it.

14. How do you delete data from a MongoDB collection?

You can delete data from a MongoDB collection using methods like `deleteOne()`, `deleteMany()`, or `findOneAndDelete()`. You specify a query to select the documents to delete.

15. What is a cursor in MongoDB, and when is it used?

A cursor in MongoDB is an iterator to retrieve and process documents from query results. Cursors are used when fetching large result sets, allowing you to retrieve documents in batches.

16. Can you explain the concept of data modeling in MongoDB?

Data modeling in MongoDB involves designing the structure of your documents and collections to represent your data best and meet your application's requirements. It includes defining document schemas, relationships, and indexing strategies.

17. How is data consistency maintained in MongoDB?

MongoDB provides strong consistency within a single document but offers eventual consistency for distributed data across multiple nodes or shards. It controls data consistency levels by using mechanisms like write concern and read preferences.

18. What is the role of collections in MongoDB?

Collections in MongoDB are containers for organizing and storing related documents. They act as the equivalent of tables in relational databases, grouping similar data.

19. How do you perform a query in MongoDB?

You can perform queries in MongoDB using the `find()` method, where you specify criteria to filter documents. You can also use various query operators to refine your queries.

20. Can you explain the concept of aggregation in MongoDB?

MongoDB's aggregation framework is a powerful tool designed for processing and transforming documents within a collection. With it, you can execute various operations such as grouping, sorting, and computing aggregate values on your dataset.

21. What is the difference between MongoDB and MySQL?

- MongoDB is a NoSQL database, while MySQL is a traditional relational database.
- MongoDB stores data in flexible, schema-less documents; MySQL uses structured tables with fixed schemas.
- MongoDB is designed for horizontal scalability, while MySQL typically scales vertically.
- MongoDB is often used for unstructured or semi-structured data, while MySQL is commonly used for structured data.

22. How do you backup a MongoDB database?

You can back up a MongoDB database using tools like `mongodump` or by configuring regular snapshots at the file system or cluster level.

23. What are the main features of MongoDB?

Some prominent features of MongoDB include flexibility in data modeling, horizontal scalability, support for unstructured data, powerful query language, automatic sharding, high availability with replica sets, and geospatial capabilities.

24. What is the purpose of using MongoDB over other databases?

MongoDB is chosen over other databases for its ability to handle flexible, unstructured, and rapidly changing data. It excels in scenarios where scalability, speed, and agility are essential, such as web and mobile applications, real-time analytics, and content management systems. Its horizontal scaling capabilities also make it suitable for large-scale data storage and processing.

- MongoDB Intermediate Interview Questions

1. How does MongoDB ensure high availability?

MongoDB guarantees robust availability via replica sets consisting of multiple MongoDB servers that store identical data. This setup offers redundancy and seamless failover capabilities. In the event of a primary node failure, an automatic process elects one of the secondary nodes to take over as the new primary, thus ensuring uninterrupted service.

2. What is the role of a sharding key in MongoDB?

A sharding key determines how data is distributed across multiple shards (database partitions) in a sharded cluster. MongoDB uses a field in the document to decide which shard should store the document. Choosing an appropriate sharding key is crucial for even data distribution and efficient queries.

3. Can you explain replica set elections in MongoDB?

Replica set elections occur when the primary node in a replica set becomes unavailable. In such cases, the replica set members vote to elect a new primary. The node with the most votes becomes the new primary, ensuring data availability and continuity of service.

4. How does MongoDB handle transactions?

MongoDB introduced multi-document transactions in version 4.0, allowing you to perform ACID-compliant transactions. Transactions ensure that a series of operations succeeds or fails, maintaining data consistency.

5. What are the different types of indexes in MongoDB?

MongoDB supports various indexes, including single-field indexes, compound indexes, geospatial indexes, text indexes, hashed indexes, and wildcard indexes.

6. Can you explain the aggregation pipeline in MongoDB?

The Aggregation Pipeline is a robust framework for performing data transformations and computations on data stored in MongoDB. It consists of stages, each processing and transforming data before passing it to the next stage. It's commonly used for complex data analysis and aggregation operations.

7. How do you monitor the performance of a MongoDB database?

You can monitor MongoDB using various tools and techniques. MongoDB provides built-in metrics and logs, and external monitoring tools like MongoDB Atlas, MMS, and third-party solutions can help track performance, query execution, and resource usage.

8. What is journaling in MongoDB?

In MongoDB, journaling is a durability feature that ensures data is written to a journal (write-ahead log) before it's written to data files. This provides crash recovery and data consistency guarantees.

9. How does MongoDB handle replication and failover?

MongoDB uses replica sets for replication and failover. Data is replicated to secondary nodes, and when a primary node failure occurs, one of the secondaries is automatically elected as the new primary to maintain high availability.

10. What are the different types of sharding strategies in MongoDB?

MongoDB supports various sharding strategies, including range-based sharding, hash-based sharding, and tag-aware sharding. The choice of strategy depends on the data distribution and query patterns.

11. Can you explain the read and write concerns in MongoDB?

Read and Write concerns in MongoDB allow you to specify the data consistency and acknowledgment required for read and write operations. They include options like "majority," "acknowledged," and "unacknowledged."

12. How do you scale a MongoDB database?

You can scale MongoDB horizontally by adding more servers to a cluster, vertically by upgrading server hardware, or by using sharding to distribute data across multiple servers in a sharded cluster.

13. What is the role of the WiredTiger storage engine in MongoDB?

Since version 3.2 of MongoDB, WiredTiger has served as the primary storage engine responsible for data storage, compression, and caching, thereby enhancing both performance and concurrency.

14. How do you implement security in MongoDB?

MongoDB provides a range of security capabilities, including authentication, role-based access control (RBAC), SSL/TLS encryption, auditing, and network security, ensuring data safeguarding and preventing unauthorized access.

15. Can you explain how MongoDB handles large data sets?

MongoDB can handle large data sets using horizontal scaling (sharding), optimized indexing, and efficient storage mechanisms like WiredTiger. It also provides tools for data partitioning and distribution.

16. What is the difference between embedded documents and references in MongoDB?

Embedded documents are nested within another document, while references are links or references to documents in separate collections. Embedded documents are used for denormalization and improved query performance, while references maintain data integrity.

17. How do you optimize query performance in MongoDB?

You can optimize query performance by creating appropriate indexes, using the Aggregation Pipeline, minimizing the number of queries, and optimizing query patterns to leverage the query planner.

18. What are capped collections in MongoDB?

Capped collections are fixed-size collections that maintain data insertion order. Once the collection reaches its size limit, old data is automatically overwritten by new data. They are often used for logging and event tracking.

19. How does MongoDB handle schema migrations?

MongoDB's flexible schema makes it easier to evolve the data model over time. When schema changes are required, applications can handle data migration using techniques like in-place updates or background processes.

20. What are the common pitfalls in MongoDB data modeling?

Common pitfalls include not choosing an appropriate sharding key, not understanding query patterns, not considering index size, and failing to denormalize data when necessary.

21. Can you explain the concept of GridFS in MongoDB?

GridFS represents a MongoDB standard designed to handle storing and retrieving substantial files, such as images, videos, and binary data. This approach involves breaking down large files into smaller segments and then saving them as individual documents within collections. This method enables the efficient handling, retrieval, and administration of such files.

22. How do you manage sessions in MongoDB?

MongoDB provides a session management API for managing multi-statement transactions. Sessions allow you to start and commit transactions, ensuring data consistency.

23. What are the best practices for index creation in MongoDB?

Best practices include creating indexes based on query patterns, avoiding too many indexes, using compound indexes effectively, and periodically reviewing and maintaining indexes for optimal performance.

24. How does MongoDB integrate with other data analysis tools?

MongoDB can integrate with various data analysis tools and frameworks through connectors, drivers, and plugins. Popular tools like Apache Spark and Hadoop have connectors for MongoDB data.

25. What is the role of Oplog in MongoDB replication?

Oplog (short for "operation log") is a capped collection that records all write operations in the primary node of a replica set. Secondary nodes use the oplog to replicate changes and maintain data consistency with the primary. It plays a crucial role in replication and failover processes.

MongoDB Advanced Interview Questions

1. How do you design a sharded MongoDB architecture for a large-scale application?

- To design a sharded MongoDB architecture for a large-scale application, consider the following steps:
- Identify a sharding key that evenly distributes data across shards.
- Set up a shard cluster with multiple shard servers.
- Configure a shard router (mongos) to route queries to the appropriate shards.
- Implement replica sets within each shard for high availability.
- Monitor and scale the cluster as needed to maintain performance.

2. Can you explain the complexities involved in MongoDB data sharding?

MongoDB data sharding introduces complexities such as choosing the right shard key, managing data distribution, ensuring data consistency, and handling shard rebalancing. Handling shard keys and ensuring balanced data distribution are key challenges.

3. What are the strategies for handling data consistency in distributed MongoDB deployments?

In distributed MongoDB deployments, you can achieve data consistency through various strategies:

- Read Preference: Specify read preferences to control which data is read.
- Write Concern: Use write concern levels to control the acknowledgment of write operations.
- Transactions: MongoDB supports multi-document transactions to ensure consistency across documents.

4. How do you handle data migration in a live MongoDB environment?

Use tools like MongoDB's `mongodump` and `mongorestore` to perform live data migrations. These tools allow you to export data from one cluster and import it into another while minimizing downtime.

5. Can you explain the internals of the WiredTiger storage engine?

In MongoDB, WiredTiger is the default storage engine. It supports document-level locking, compression, and data durability through write-ahead logging (WAL). It uses B-trees and LSM trees for data storage.

6. What are the best practices for disaster recovery in MongoDB?

Disaster recovery best practices in MongoDB include regular backups, offsite storage, automated backup processes, and testing backup restoration procedures. Implementing replication and having a well-defined recovery plan is crucial.

7. How do you perform advanced data aggregation operations in MongoDB?

MongoDB offers the Aggregation Framework, allowing for complex data aggregation operations. You can use operators like `$group`, `$project`, and `$lookup` to perform operations like filtering, grouping, and joining data.

8. What are the considerations for choosing shard keys in a highly distributed environment?

Consider even data distribution, query patterns, and scalability when choosing shard keys. Avoid monotonically increasing keys to prevent hotspots. Use hashed shard keys for better distribution.

9. How do you troubleshoot performance issues in a sharded MongoDB cluster?

Troubleshooting performance in a sharded MongoDB cluster involves monitoring metrics, identifying slow queries, optimizing indexes, and scaling resources where needed. Analyzing the query execution plan is crucial.

10. Can you explain the process of tuning Read and Write operations in high-load environments?

In high-load environments, you can optimize read and write operations by adjusting the MongoDB configuration parameters, using appropriate indexes, and employing caching mechanisms like Redis or Memcached.

11. How does MongoDB handle network partitioning and split-brain scenarios?

MongoDB uses a replica set and an internal consensus algorithm to handle network partitioning scenarios. In split-brain scenarios, priority settings and automatic failover can help maintain data consistency.

12. What are the best practices for securing a MongoDB cluster in a public cloud environment?

Best practices for securing MongoDB in a public cloud environment include network security groups, authentication, role-based access control, rest and transit encryption, and regularly applying security patches.

13. How do you automate MongoDB deployments in a DevOps environment?

Automation tools like Ansible, Terraform, or Kubernetes can be used to automate MongoDB deployments in a DevOps environment. Infrastructure as Code (IaC) principles are often applied.

14. Can you discuss the challenges of integrating MongoDB with big data technologies?

Integrating MongoDB with big data technologies like Hadoop, Spark, or Kafka can be challenging. You may use connectors or ETL tools to transfer and process data between MongoDB and these systems.

15. How do you optimize MongoDB for IoT applications with high ingestion rates?

To optimize MongoDB for IoT applications, use sharding, time-series data models, and proper indexing. Implement data retention policies and consider using edge computing for data preprocessing.

16. What are the trade-offs between different replication strategies in MongoDB?

MongoDB offers primary-secondary replication, replica sets, and sharding. Each has trade-offs regarding data consistency, failover, and read scalability. Choose the replication strategy that suits your application's needs.

17. How do you manage large-scale data migrations in MongoDB?

For large-scale data migrations, use tools like MongoDB Atlas Data Lake or data pipeline solutions like Apache Kafka. Plan for data validation and verification to ensure data integrity.

18. What are the advanced techniques for monitoring MongoDB clusters?

Use monitoring tools like MongoDB Cloud Manager, Prometheus, or Grafana to track key performance metrics, resource utilization, and cluster health. Set up alerts for proactive issue detection.

19. How do you ensure data integrity in a MongoDB transaction?

MongoDB supports multi-document transactions to ensure data integrity. You can use transactions to group multiple operations into a single unit of work, allowing for atomicity, consistency, isolation, and durability (ACID).

20. Can you explain the role of consensus algorithms in MongoDB cluster management?

MongoDB uses the Raft consensus algorithm to replicate set elections and leader selection. Raft ensures that the cluster maintains a consistent state and can recover from failures.

21. How do you handle schema evolution in MongoDB for agile development practices?

MongoDB's flexible schema allows for agile development practices. Developers can evolve the schema by adding or removing fields as needed, and versioning data structures may be necessary for compatibility.

22. What are the challenges and solutions for backup and restoration in large MongoDB deployments?

Challenges in large MongoDB deployments include data volume, backup frequency, and retention policies. Solutions involve using incremental backups, snapshots, and offsite storage with efficient data deduplication.

23. How does MongoDB interact with microservices architectures?

MongoDB can be used as a data store in microservices architectures. Each microservice can have its database or share it with others, depending on data isolation and coupling requirements.

24. Can you discuss the impact of network latency on MongoDB’s performance and scalability?

Network latency can impact MongoDB's performance and scalability, especially in geographically distributed deployments. Techniques like read preference configuration and sharding can help mitigate latency issues.

25. What are the future trends and expected developments in MongoDB?

While I cannot provide real-time information, MongoDB's future trends may include enhanced support for multi-cloud deployments, further improvements in scalability and performance, and new features to address evolving application needs in data management and analysis.