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.

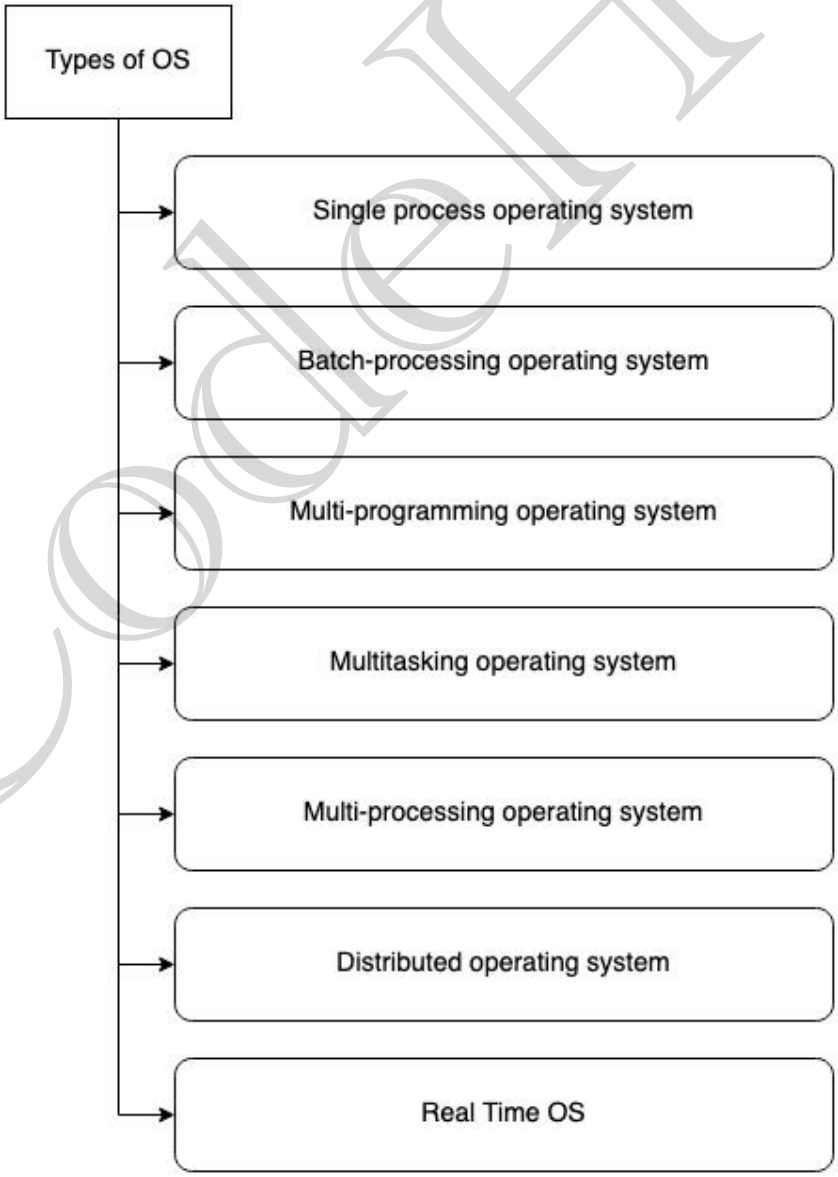
LEC-2: Types of OS

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]

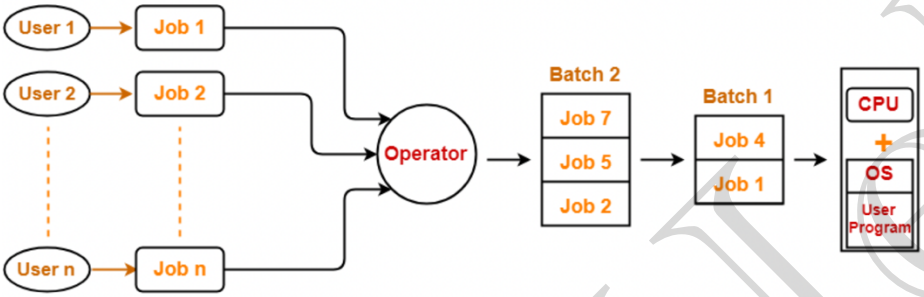


Single process OS, only 1 process executes at a time from the ready queue. [Oldest]

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.

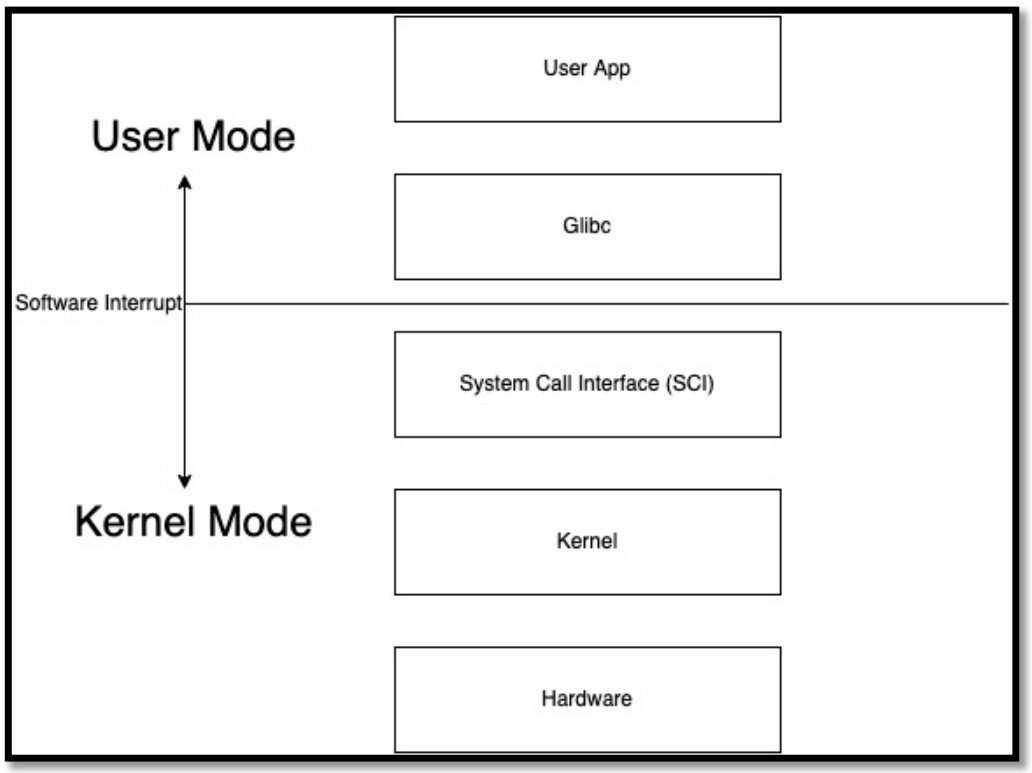
- User executes a process. (User space)
- Gets system call. (US)
- Exec system call to create a process. (KS)
- Return to US.

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?

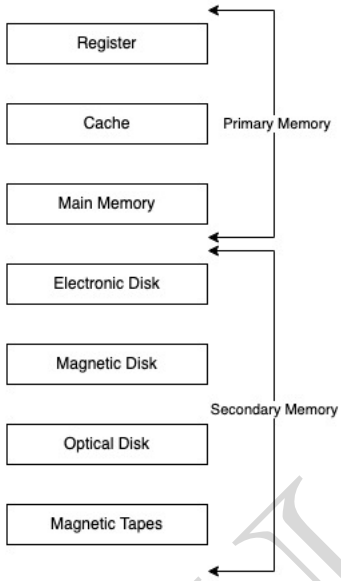
i. PC On
ii. CPU initializes itself and looks for a firmware program (BIOS) stored in BIOS Chip **(**Basic input-output system chip is a ROM chip found on mother board that allows to access & setup computer system at most basic level.)
    1. In modern PCs, CPU loads UEFI (Unified extensible firmware 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?



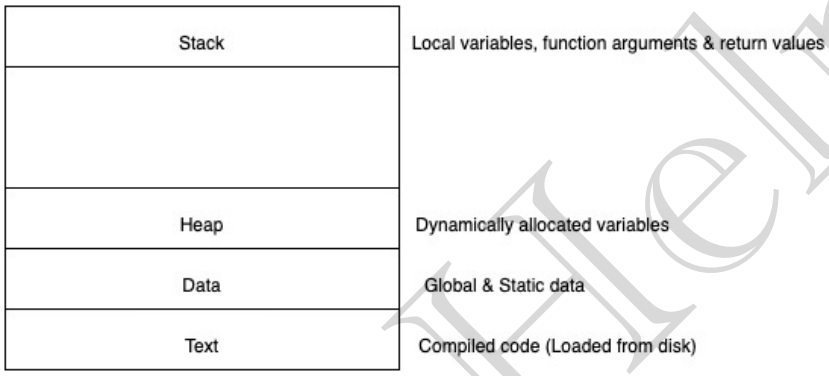
1. **Register**: Smallest unit of storage. It is a part of CPU itself.
   A register may hold an instruction, a storage address, or any data (such as bit sequence or individual characters).
   Registers are a type of computer memory used to quickly accept, store, and transfer data and instructions that 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

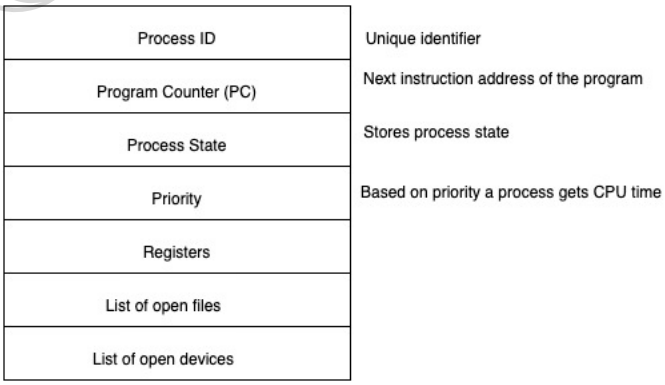
1. **Cost**:
   a. Primary storages are costly.
   b. Registers are most expensive due to expensive semiconductors & labour.
   c. Secondary storages are cheaper than primary.
2. **Access Speed**:
   a. Primary has higher access speed than secondary memory.
   b. Registers has highest access speed, then 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:



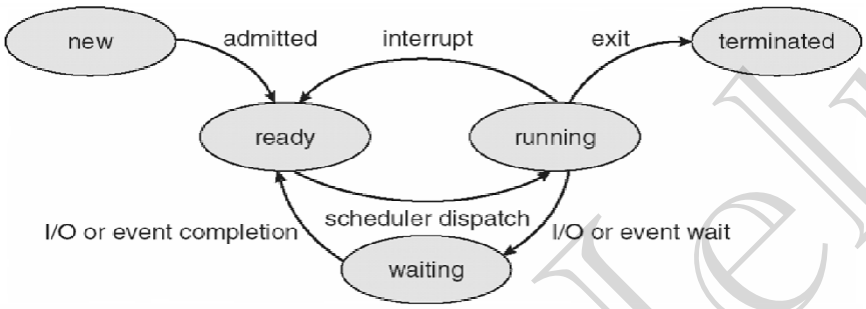
5. **Attributes** of process:
   a. Feature that allows identifying a process uniquely.
   b. Process table
      i. All processes are being tracked by OS using a table like data structure.
      ii. Each entry in that table is process control block (PCB).
   c. PCB: Stores info/attributes of a process.
      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

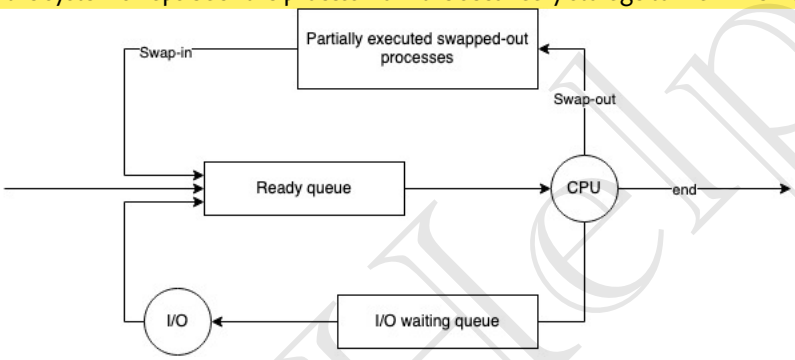
1. **Process States**: As process executes, it changes state. Each process may be in one of the following states.
    a. **New**: OS is about to pick the program & convert it into process. OR the process is being created.
    b. **Run**: Instructions are being executed; CPU is allocated.
    c. **Waiting**: 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.



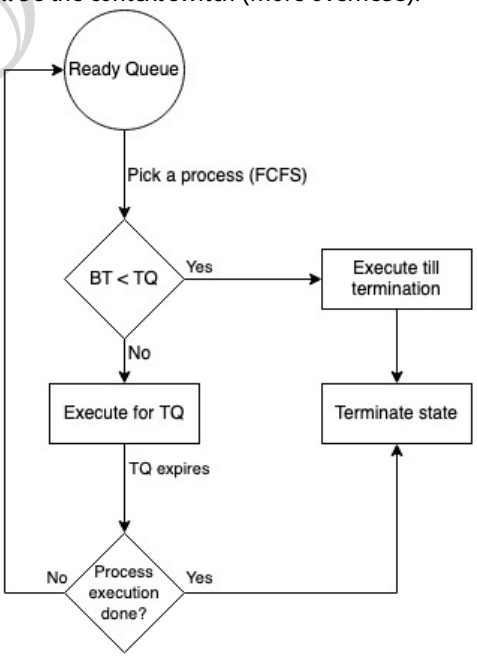
2. **Context-Switching**
   - a. Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process.
   - b. When this occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process 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.

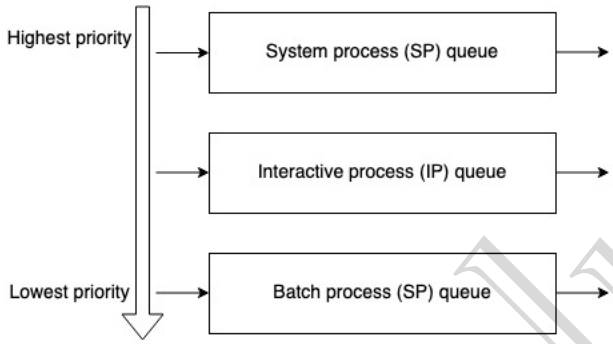
LEC-13: CPU Scheduling | SJF | Priority | RR

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).



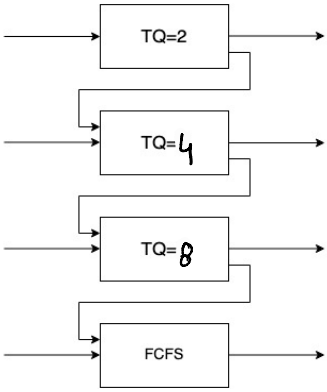
LEC-14: MLQ | MLFQ

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.

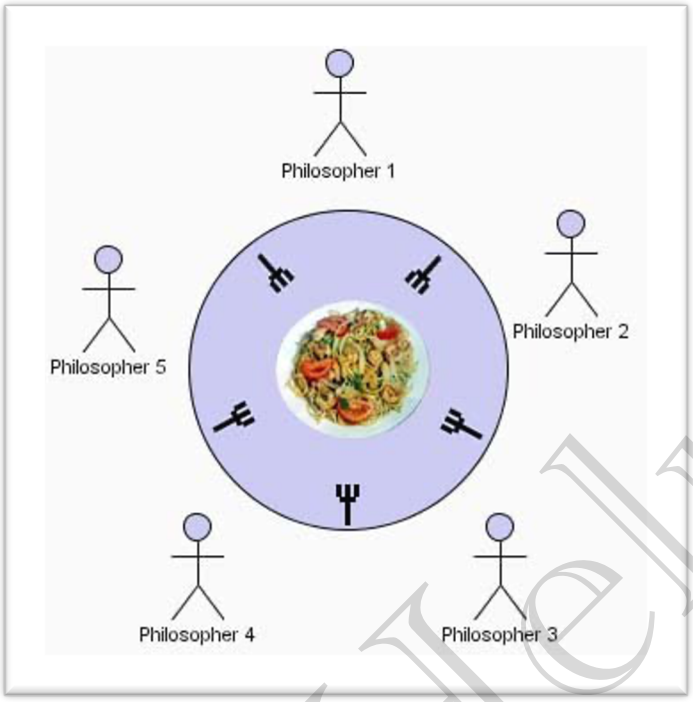
LEC-16: Critical Section Problem and How to address it

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



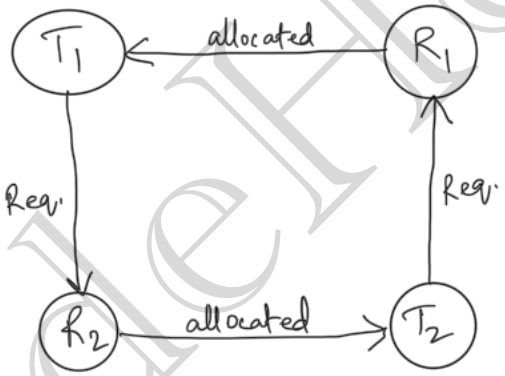
1. We have **5 philosophers.**
2. They spend their life just being in **two states:**
    a. Thinking
    b. Eating
3. They sit on a circular table surrounded by 5 chairs (1 each), in the center of table is a bowl of noodles, and the table is laid with 5 single forks.
4. **Thinking state**: When a ph. 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.**

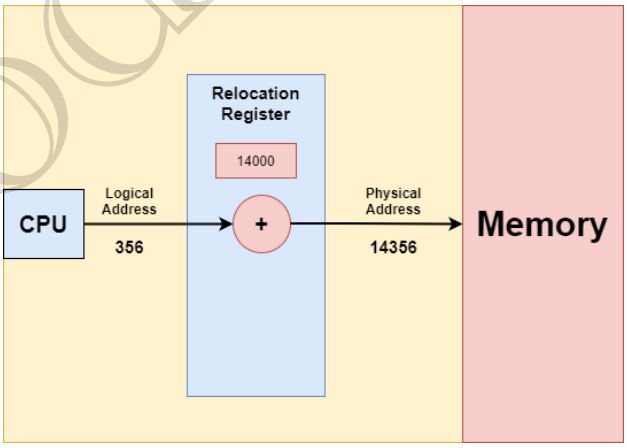
11. To ensure that deadlocks never occur, the system can use either a **deadlock prevention or deadlock avoidance scheme**.
12. **Deadlock Prevention**: by ensuring at least one of the necessary conditions cannot hold.
    a. **Mutual exclusion**
        i. Use locks (mutual exclusion) only for non-sharable 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.

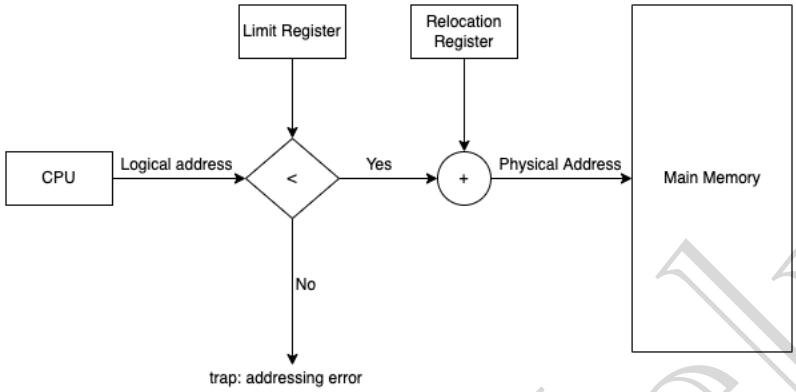
LEC-24: Memory Management Techniques | Contiguous Memory Allocation

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.

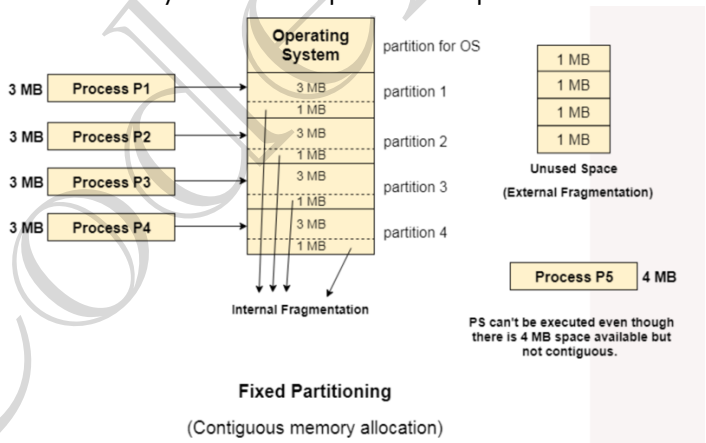


4. How OS manages the isolation and protect? **(Memory Mapping and Protection)**
    - a. OS provides this Virtual Address Space (VAS) concept.
    - b. To separate memory space, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can 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.

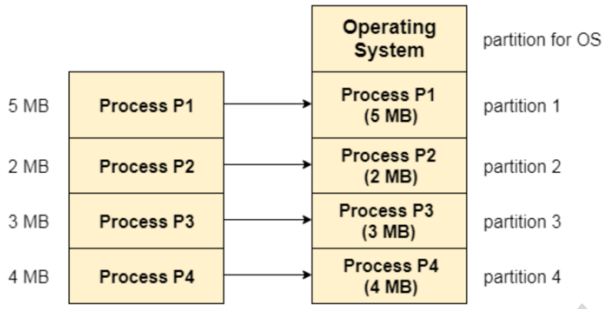


        ii.
        iii. Limitations:
            1. **Internal Fragmentation:** if the size of the process is lesser then the total size of the partition then some size of the partition gets wasted and remain unused. This is wastage of the memory and called internal fragmentation.
            2. **External 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)

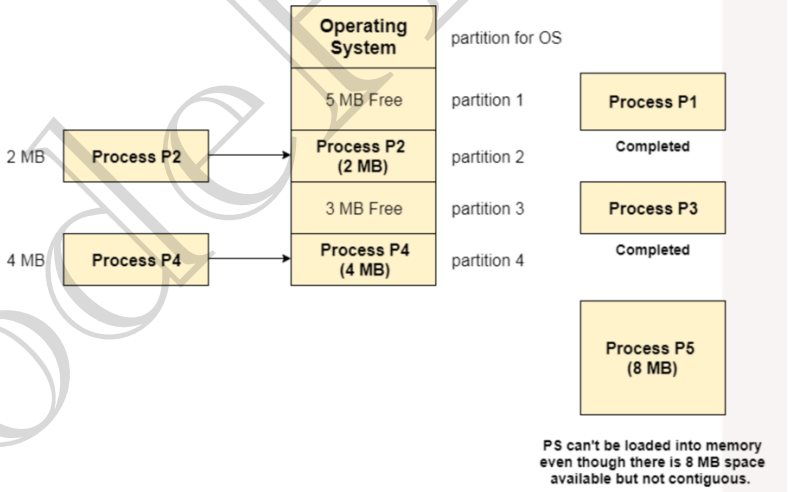
ii.

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

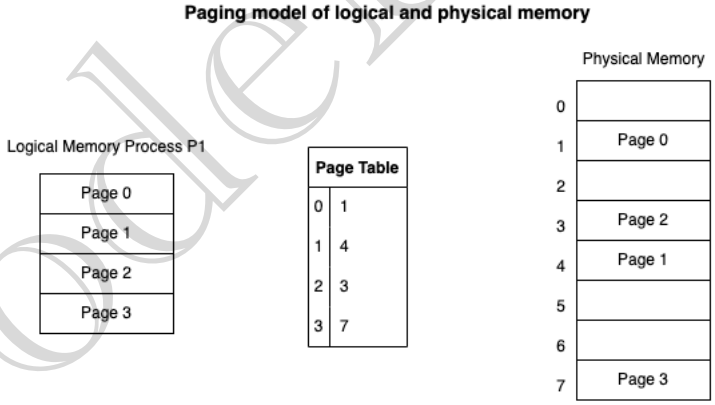
X

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

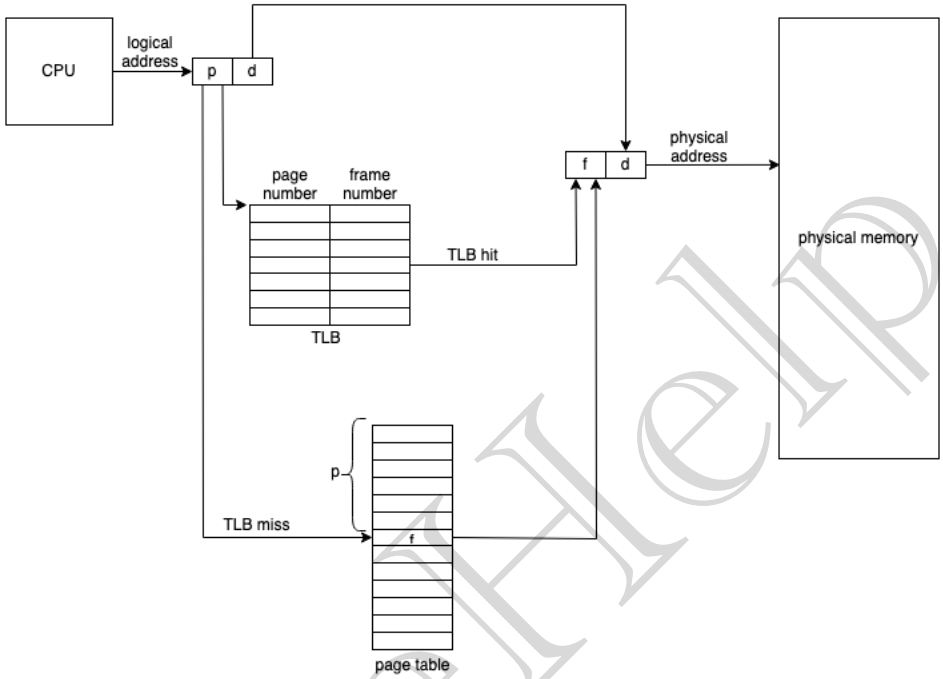
1. The main **disadvantage of Dynamic partitioning is External Fragmentation.**
    a. Can be removed by Compaction, but with overhead.
    b. We need more dynamic/flexible/optimal mechanism, to load processes in the partitions.
2. **Idea behind Paging**
    a. If we have only two small non-contiguous 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.

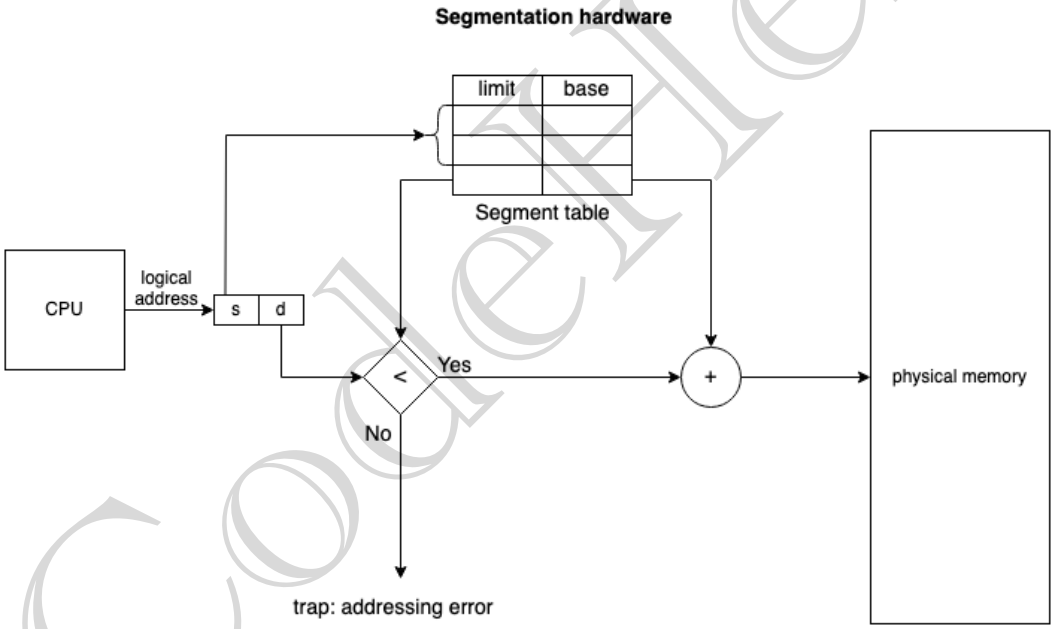
d. Page table is stores in main memory & because of this when the memory references is made the translation is slow.
e. When we are retrieving physical address using page table, after getting frame address corresponding to the page number, we put an entry of the into the TLB. So that next time, we can get the values from TLB directly without referencing actual page table. Hence, make paging process faster.

Paging hardware with TLB



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

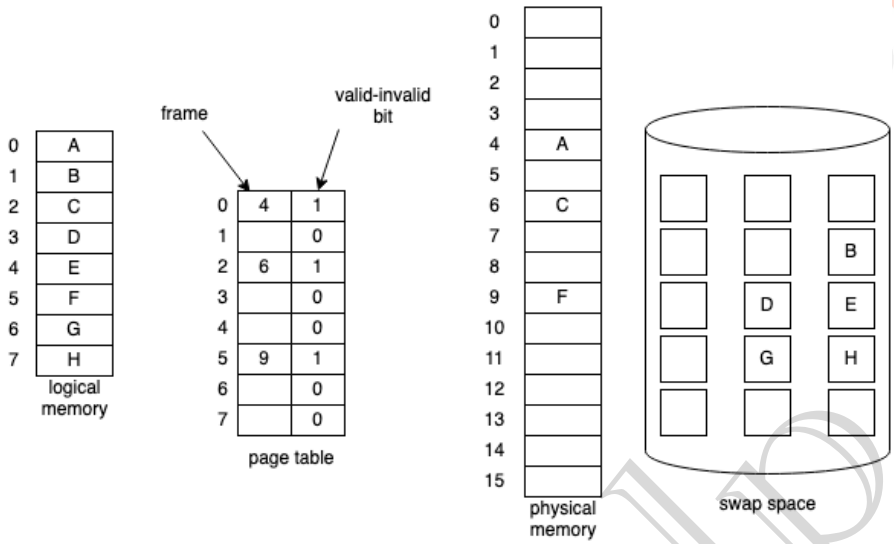
1. An important aspect of memory management that become unavoidable with paging is separation of user's view of memory from the actual physical memory.
2. Segmentation is memory management technique that supports the **user view of memory.**
3. A logical address space is a collection of segments, these 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.

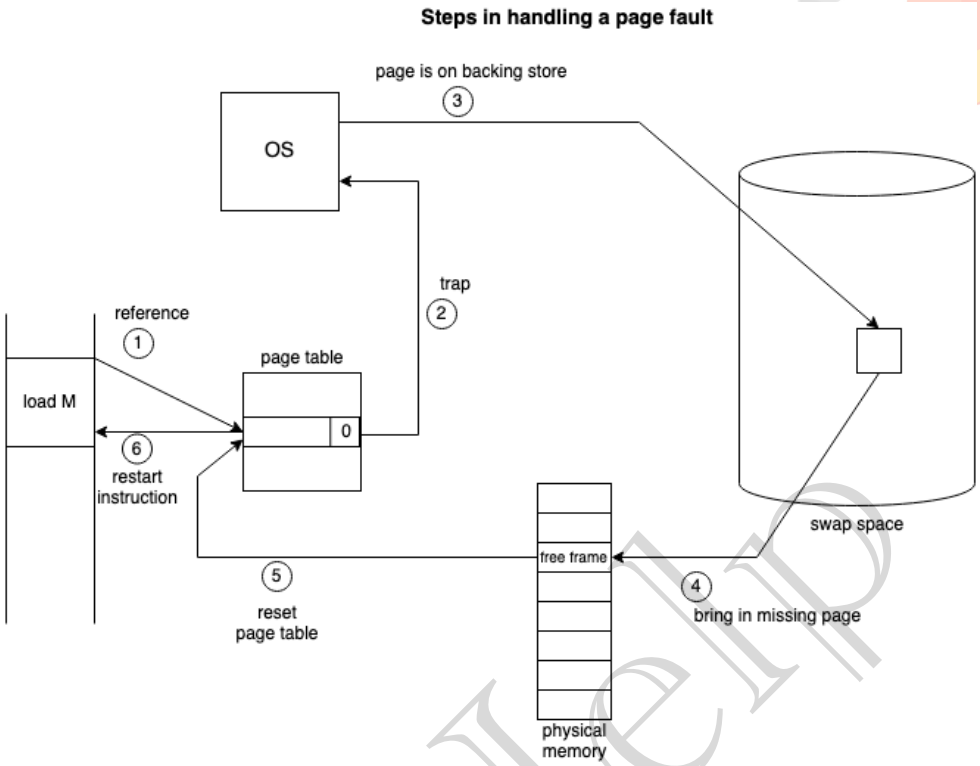
ii. If ref. was invalid process throws exception.
If ref. is valid, pager will swap-in the page.

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.

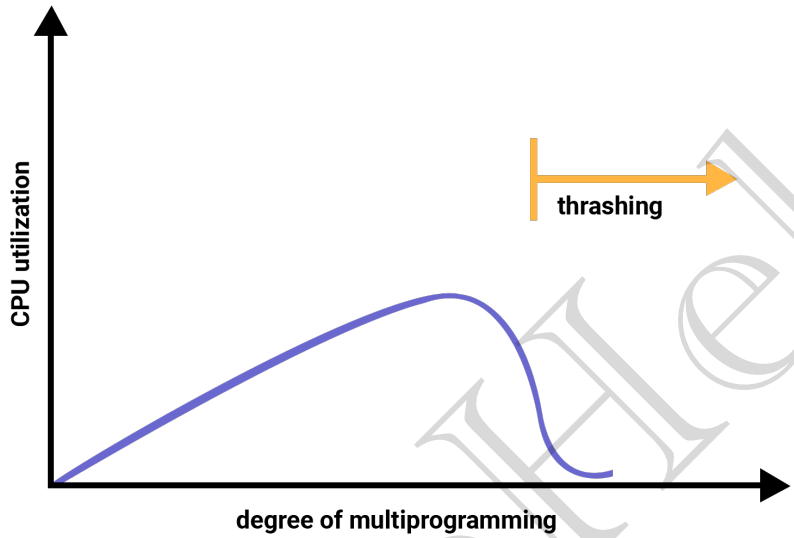
LEC-29: Page Replacement Algorithms

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

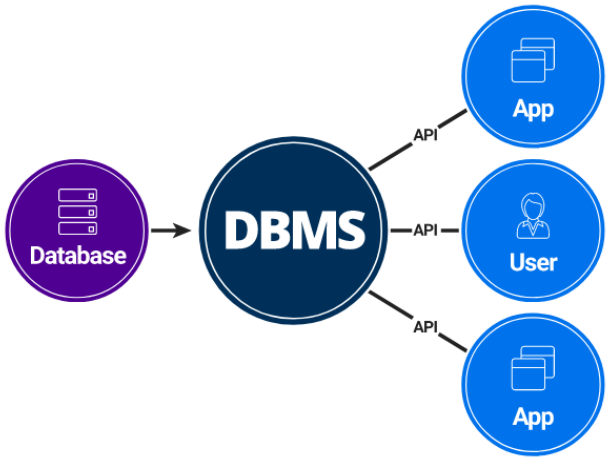
1. **Thrashing**
    a. If the process doesn't have the number of frames it needs to support pages in active use, it will quickly page-fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, 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.

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.

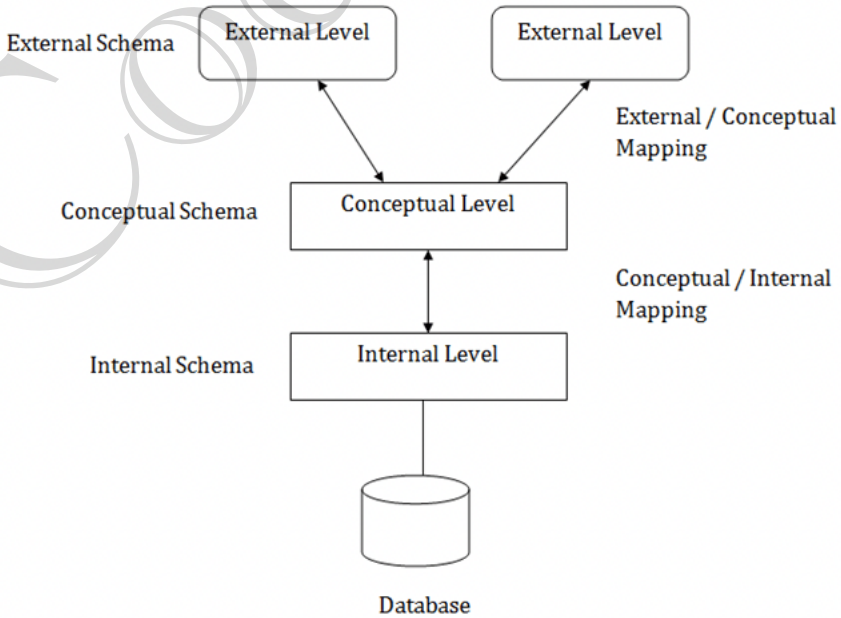


7.

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.

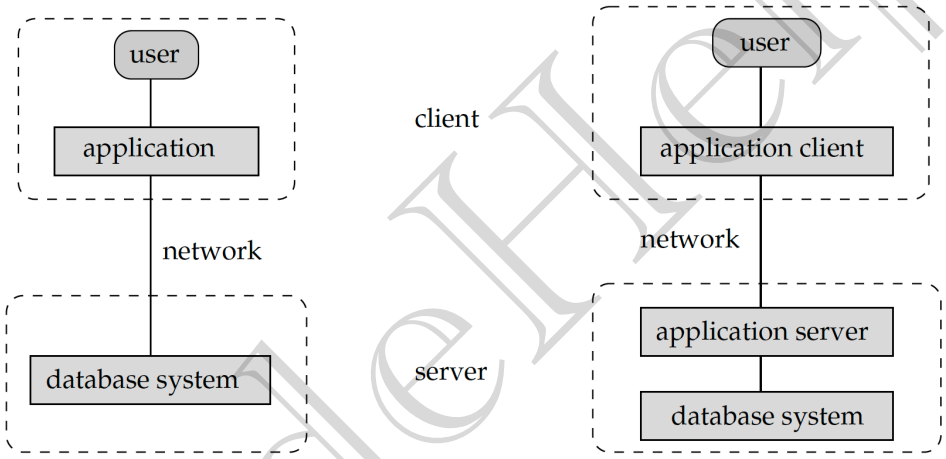


    g.
2. **Instances and Schemas**
    a. The collection of information stored in the DB at a particular moment is called an **instance** of DB.

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.

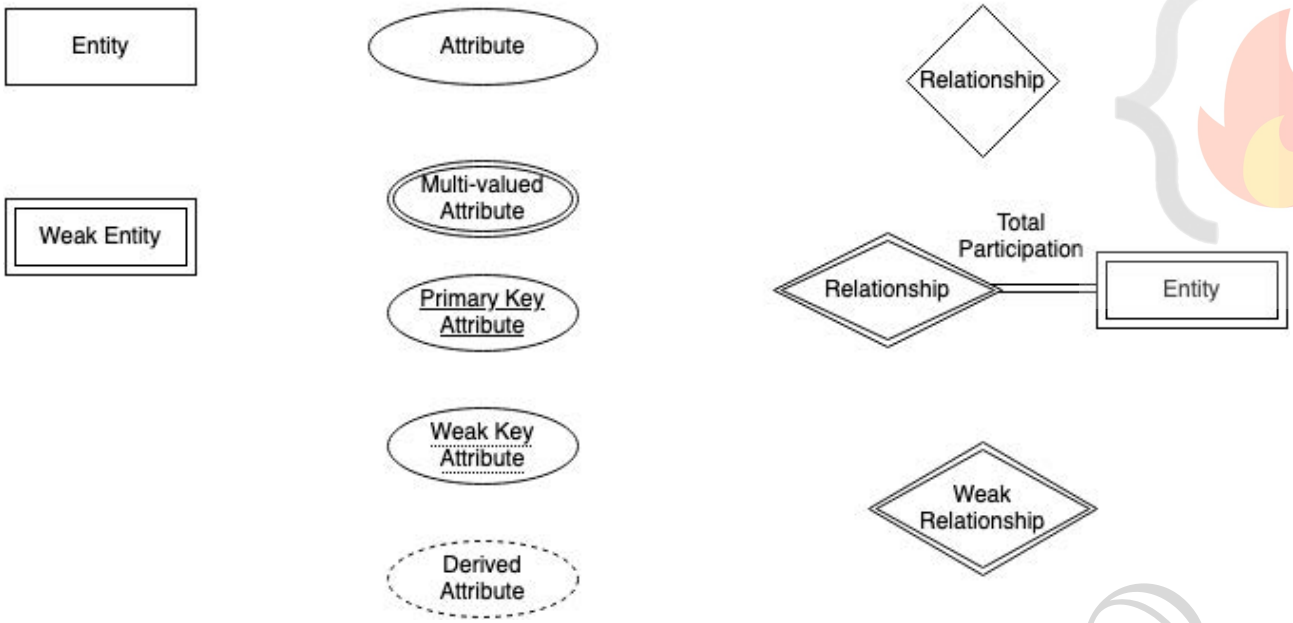
6. **NULL Value**
    1. An attribute takes a null value when an entity does not have a value for it.
    2. It may indicate "not applicable", value doesn't exist. e.g., person having no middle-name
    3. It may indicate "unknown".
        1. Unknown can indicate missing entry, e.g., name value of a 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.

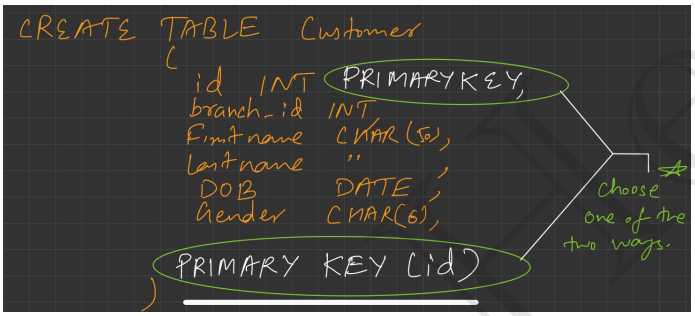
2. SQL is smart enough to realise that if you are using GROUP BY and not using any aggregation function, then you mean "DISTINCT".

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,

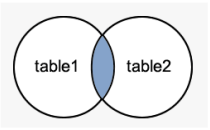
FOREIGN KEY(cust_id) REFERENCES customer(id) ON DELETE CASCADE
);

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
    );

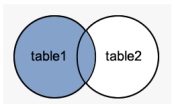
4. **REPLACE**
    1. Primarily used for already present tuple in a table.
    2. As UPDATE, using REPLACE with the help of WHERE clause in PK, then that row will be replaced.
    3. As INSERT, if there is no duplicate data new tuple will be inserted.
    4. REPLACE INTO student (id, class) VALUES(4,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

JOINING TABLES

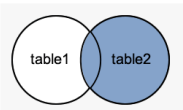
1. All **RDBMS** are relational in nature, we refer to other tables to get meaningful outcomes.
2. FK are used to do reference to other table.
3. **INNER JOIN**
    1. Returns a resultant table that has matching values from both the tables or all the tables.
    2. SELECT column-list FROM table1 INNER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



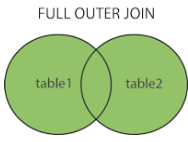
    3. **Alias in MySQL (AS)**
        1. Aliases in MySQL is used to give a temporary name to a table or a column in a table for the purpose of a particular query. It works as a nickname for expressing the tables or column names. It makes the query short and neat.
        2. SELECT col_name AS alias_name 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;



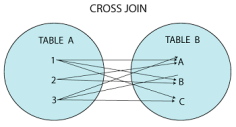
    2. **RIGHT JOIN**
        1. This returns a resulting table that all the data from right table and the matched data from the left table.
        2. SELECT columns FROM table RIGHT JOIN table2 ON join_cond;



    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

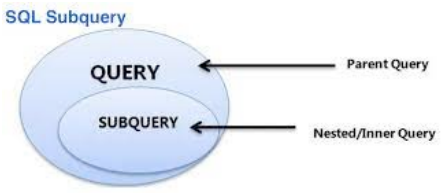
1. Used to combine multiple select statements.
2. Always gives distinct rows.

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).

LEC-11: Normalisation

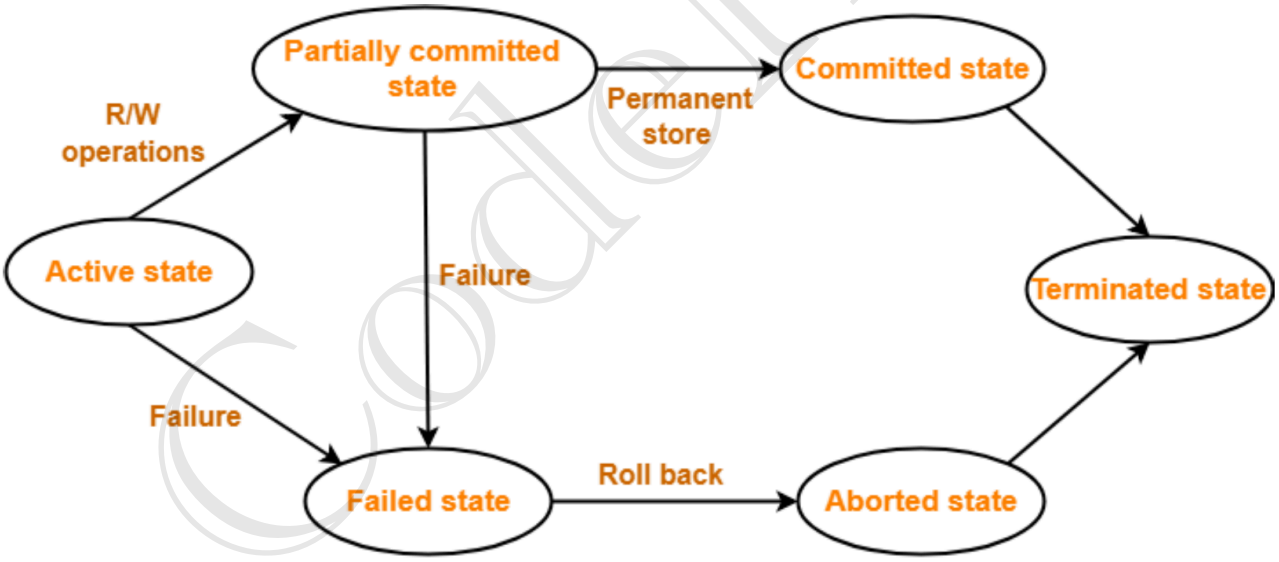
1. **Normalisation** is a step towards DB optimisation.
2. **Functional Dependency (FD)**
    1. It's a relationship between the primary key attribute (usually) of the relation to that of the other attribute of the relation.
    2. X -> Y, the left side of FD is known as a **Determinant**, the right 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**

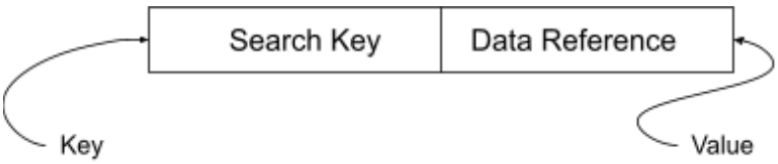
    1. When updates are made permanent on the DB. Then the T is said to be in the committed state. Rollback can't be done from the committed states. New consistent state is achieved at this stage.
4. **Failed state**
    1. When T is being executed and some failure occurs. Due to this it is impossible to 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

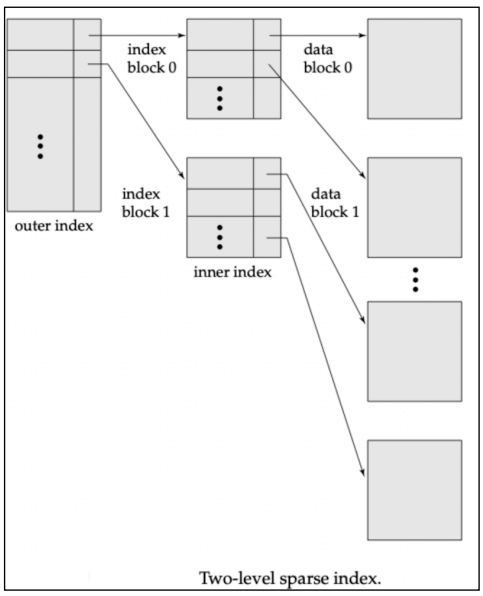
1. **Recovery Mechanism Component** of DBMS supports **atomicity and durability.**
2. **Shadow-copy scheme**
    1. Based on making copies of DB (aka, **shadow copies)**.
    2. Assumption only one Transaction (T) is active at a time.
    3. A pointer called **db-pointer** is maintained on the **disk**; 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

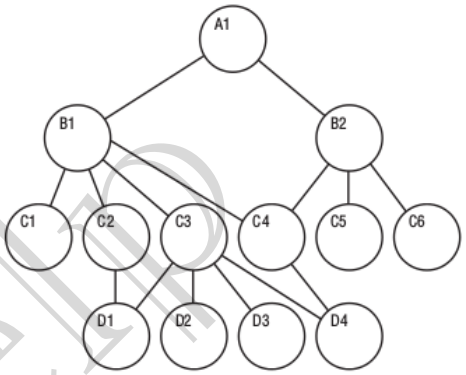
1. **Relational Databases**
    1. Based on Relational Model.
    2. Relational databases are quite **popular**, even though it was a system designed in the 1970s. Also known as relational database management systems (RDBMS), relational databases commonly use **Structured Query Language (SQL)** for operations 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)

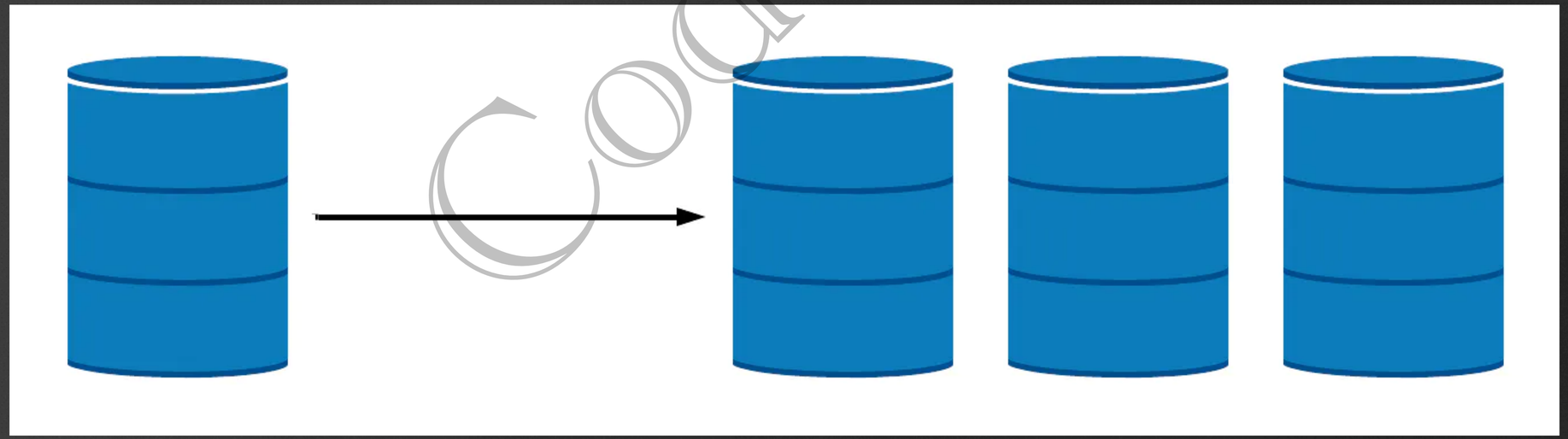
Database Scaling Patterns

Step by Step Scaling

- Lakshay

What will you learn?

- Step by Step manner, when to choose which Scaling option.
- Which Scaling option is feasible practically at the moment.



A Case Study

Cab Booking APP

- Tiny startup.
- ~10 customers onboard.
- A single small machine DB stores all customers, trips, locations, booking data, and customer trip history.
- ~1 trip booking in 5 mins.

Your App becoming famous, but…

The PROBLEM begins

- Requests scales upto 30 bookings per minute.
- Your tiny DB system has started performing poorly.
- API latency has increased a lot.
- Transactions facing Deadlock, Starvation, and frequent failure.
- Sluggish App experience.
- Customer dis-satisfaction.

Is there any solution?

- We have to apply some kind of performance optimisation measures.
- We might have to scale our system going forward.

Pattern 1

Query Optimisation & Connection Pool Implementation

- Cache frequently used non-dynamic data like, booking history, payment history, user profiles etc.
- Introduce Database Redundancy. (Or may be use NoSQL)
- Use connection pool libraries to **Cache DB connections**.
- Multiple application threads can use same DB connection.
- Good optimisations as of now.
- Scaled the business to one more city, and now getting ~100 booking per minute.

Pattern 2

Vertical Scaling or Scale-up

- Upgrading our initial tiny machine.
- RAM by 2x and SSD by 3x etc.
- Scale up is pocket friendly till a point only.
- More you scale up, cost increases exponentially.
- Good Optimisation as of now.
- Business is growing, you decided to scale it to 3 more cities and now getting 300 booking per minute.

Pattern 3

Command Query Responsibility Segregation (CQRS)

- The scaled up big machine is not able to handle all read/write requests.
- Separate read/write operations physical machine wise.
- 2 more machines as replica to the primary machine.
- All read queries to replicas.
- All write queries to primary.
- Business is growing, you decided to scale it to 2 more cities.
- Primary is not able to handle all write requests.
- Lag between primary and replica is impacting user experience.

Pattern 4

Multi Primary Replication

- Why not distribute write request to replica also?
- All machines can work as primary & replica.
- Multi primary configuration is a logical circular ring.
- Write data to any node.
- Read data from any node that replies to the broadcast first.
- You scale to 5 more cities & your system is in pain again. (~50 req/s)

Pattern 5

Partitioning of Data by Functionality

- What about separating the location tables in separate DB schema?
- What about putting that DB in separate machines with primary-replica or multi-primary configuration?
- Different DB can host data categorised by different functionality.
- Backend or application layer has to take responsibility to join the results.
- Planning to expand your business to other country.

Pattern 6

Horizontal Scaling or Scale-out

- Sharding - multiple shards.
- Allocate 50 machines - all having same DB schema - each machine just hold a part of data.
- Locality of data should be there.
- Each machine can have their own replicas, may be used in failure recovery.
- Sharding is generally hard to apply. But “No Pain, No Gain”
- Scaling the business across continents.

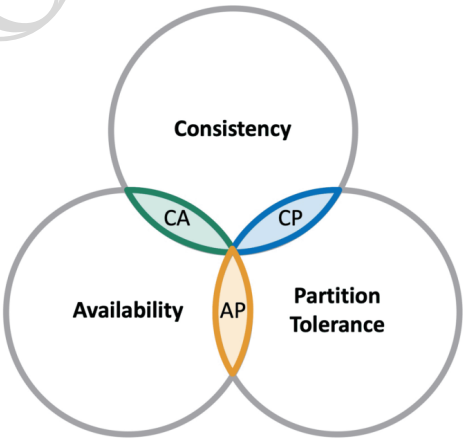
Pattern 7

Data Centre Wise Partition

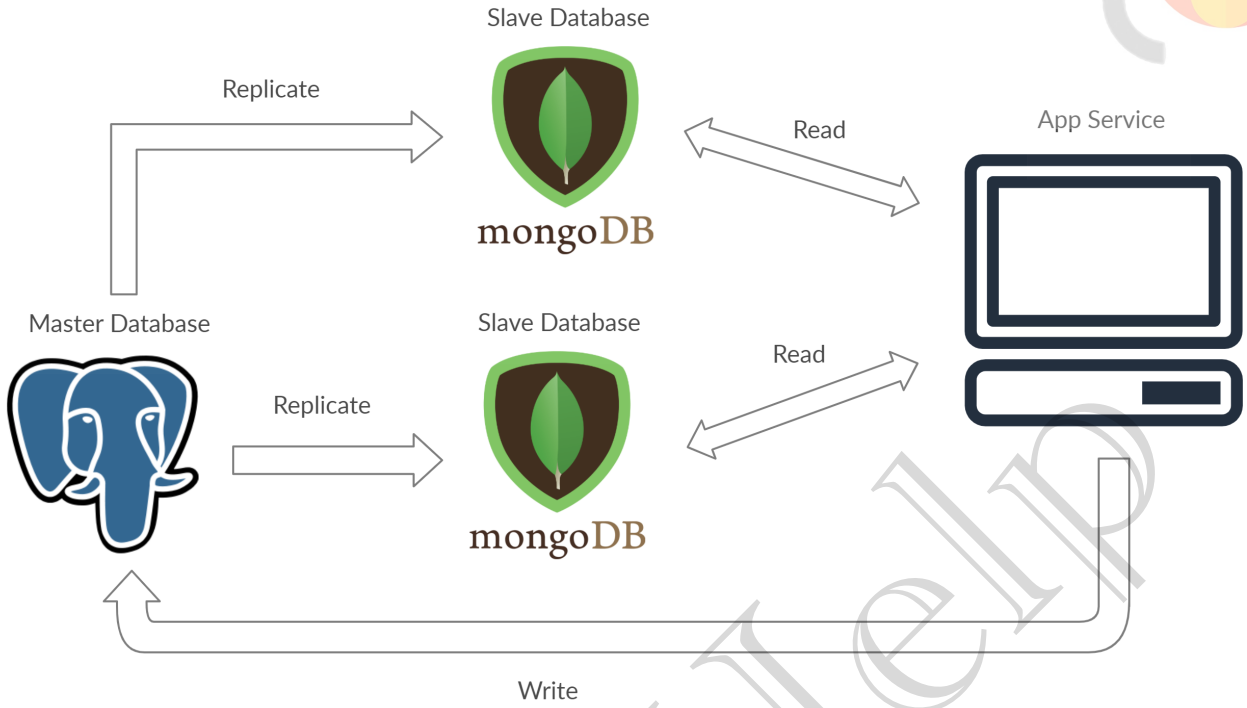
- Requests travelling across continents are having high latency.
- What about distributing traffic across data centres?
- Data centres across continents.
- Enable cross data centre replication which helps disaster recovery.
- This always maintain Availability of your system.
- Now, Plan for an IPO :p

LEC-20: CAP Theorem

1. Basic and one of the most important concept in **Distributed Databases.**
2. **Useful** to know this to design efficient distributed system for your given business logic.
3. Let's first breakdown CAP
    1. **Consistency**: In a consistent system, all nodes see the same data simultaneously. If we perform a read operation on a consistent system, it should return the value of the most recent write operation. The read should cause all nodes to return the same data. All users see the same data at the same time, regardless of the node they connect to. When data is written to a single node, it is then replicated across the other nodes in the system.
    2. **Availability**: When availability is present in a distributed system, it means that the system remains operational all of the time. Every request will get a response regardless of the individual state of the nodes. This means that the system will operate even if there are multiple nodes down. Unlike a consistent system, there's no guarantee that the response will be the most recent write operation.
    3. **Partition Tolerance**: When a distributed system encounters a partition, it means that there's a break in communication between nodes. If a system is partition-tolerant, the system does not fail, regardless of whether messages are dropped or delayed between nodes within the system. To have partition tolerance, the system must replicate records across combinations of nodes and networks.
4. What does the **CAP Theorem** says,
    1. The CAP theorem states that a distributed system can only provide **two of three properties** simultaneously: consistency, availability, and partition tolerance. The theorem formalises the **tradeoff between consistency and availability when there's a partition.**
5. **CAP Theorem NoSQL Databases:** NoSQL databases are great for distributed networks. They allow for horizontal scaling, and they can quickly scale across multiple nodes. When deciding which NoSQL database to use, it's important to keep the CAP theorem in mind.
    1. **CA Databases:** CA databases enable consistency and availability across all nodes. Unfortunately, CA databases can't deliver fault tolerance. In any distributed system, partitions are bound to happen, which means this type of database isn't a very practical choice. That being said, you still can find a CA database if you need one. Some relational databases, such as MySQL or PostgreSQL, allow for consistency and availability. You can deploy them to nodes using replication.
    2. **CP Databases:** CP databases enable consistency and partition tolerance, but not availability. When a partition occurs, the system has to turn off inconsistent nodes until the partition can be fixed. MongoDB is an example of a CP database. It's a NoSQL database management system (DBMS) that uses documents for data storage. It's considered schema-less, which means that it doesn't require a defined database schema. It's commonly used in big data and applications running in different locations. The CP system is structured so that there's **only one primary node that receives all of the write requests in a given replica set.** Secondary nodes replicate the data in the primary nodes, so if the primary node fails, a secondary node can stand-in. In banking system Availability is not as important as consistency, so we can opt it (MongoDB).
    3. **AP Databases:** AP databases enable availability and partition tolerance, but not consistency. In the event of a partition, all nodes are available, but they're not all updated. For example, if a user tries to access data from a bad node, they won't receive the most up-to-date version of the data. When the partition is eventually resolved, most AP databases will sync the nodes to ensure consistency across them. Apache Cassandra is an example of an AP database. It's a NoSQL database with no primary node, meaning that all of the nodes remain available. Cassandra allows for eventual consistency because users can re-sync their data right after a partition is resolved. For apps like Facebook, we value availability more than consistency, we'd opt for AP Databases like Cassandra or Amazon DynamoDB.



LEC-21: The Master-Slave Database Concept



1. Master-Slave is a general way to optimise IO in a system where number of requests goes way high that a single DB server is not able to handle it efficiently.
2. Its a Pattern 3 in LEC-19 (Database Scaling Pattern). ***(Command Query Responsibility Segregation)***
3. The true or latest data is kept in the Master DB thus write operations are directed there. Reading ops are done only from slaves. This architecture serves the purpose of safeguarding site ***eliability, availability, reduce latency etc .*** If a site receives a lot of traffic and the only available database is one master, it will be overloaded with reading and writing requests. Making the entire system slow for everyone on the site.
4. DB ***replication*** will take care of distributing data from Master machine to Slaves machines. This can be ***synchronous or asynchronous*** depending upon the system's need.