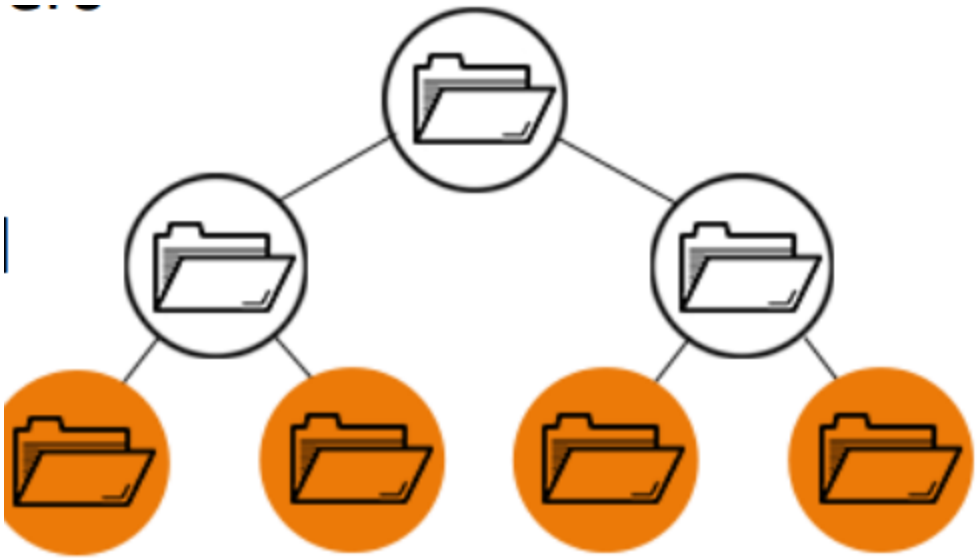
Unit-3

File Storage System

- **Definition**: Organizes and represents data as a hierarchy of files in folders.



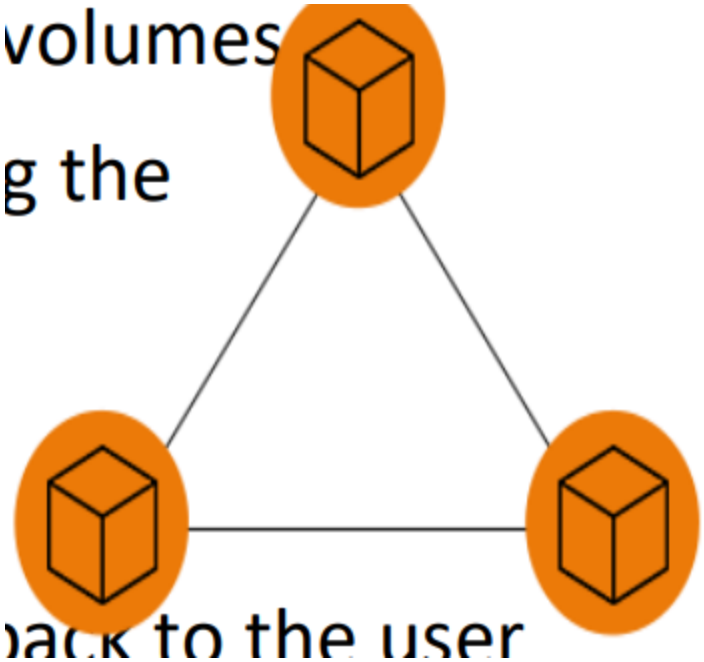
- **Data Handling**:
  - Data is stored as a single piece of information inside a folder, similar to organizing pieces of paper inside a manila folder.

- When you need to access that piece of data, your computer needs to know the path to find it.
- Data stored in files is organized and retrieved using a limited amount of metadata that tells the computer exactly where the file itself is kept.

- **Environment**:
  - Network Attached Storage (NAS) or Direct Attached Storage (DAS) are examples of file storage systems.
- **Scaling**:
  - Scaling with file storage is typically scale-out, as there is a limit to the addition of capacity to a system.

Block Storage

- **Definition**: Block storage chunks data into arbitrarily organized, evenly sized volumes.

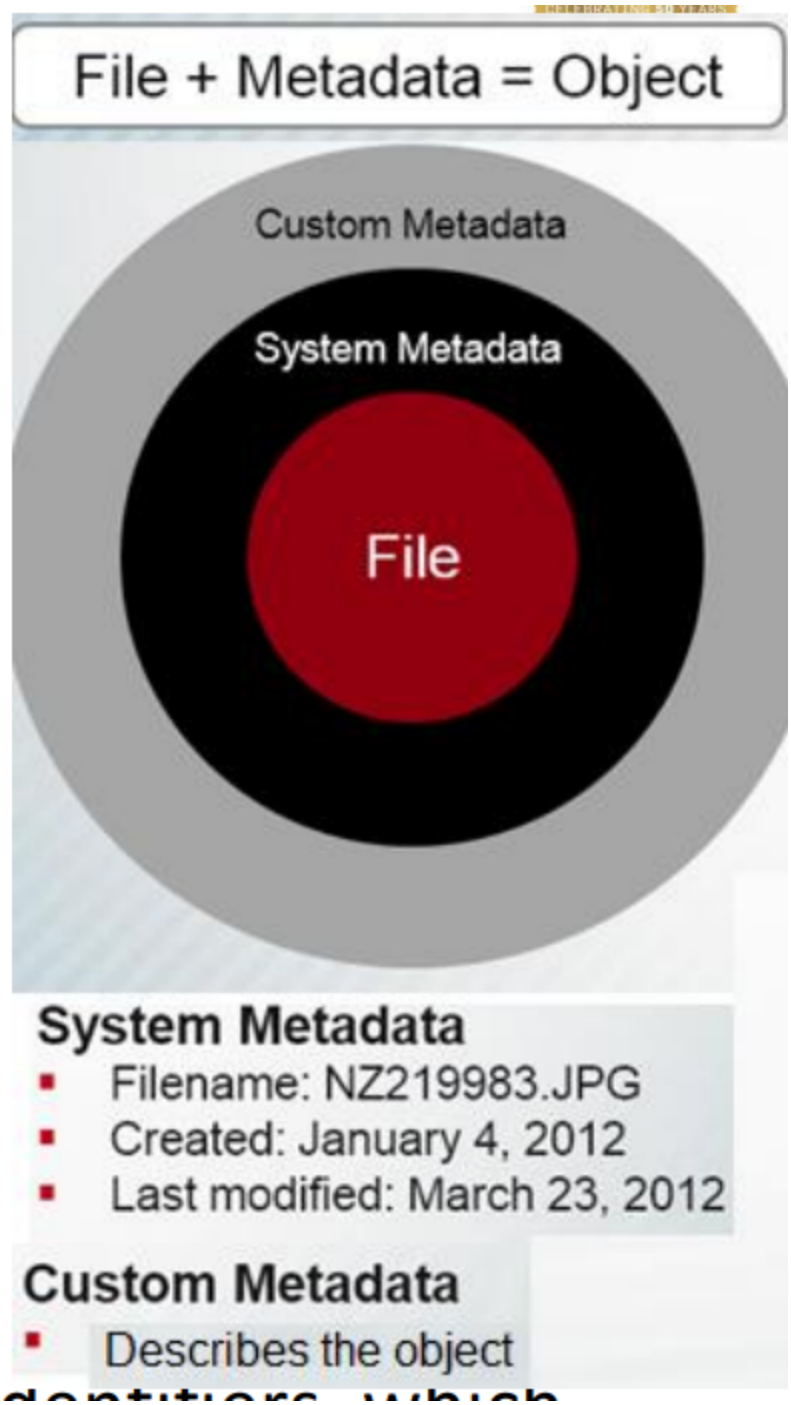


- **Data Handling**:
  - Data is chopped into blocks with each block given a unique identifier, allowing the storage system to place the data wherever convenient.

    - Block storage decouples data from the user's environment. When data is requested, the underlying storage software reassembles the blocks of data from these environments and presents them back to the user.
- **Environment**:
    - Usually used with SAN (Storage Area Network) environments.
    - It can be retrieved quickly and accessed by any environment.
- **Advantages**:
    - Efficient and reliable way to store data.
    - Easy to use and manage.
    - Works well with enterprises performing big transactions and those that deploy huge databases. The more data you need to store, the better off you'll be with block storage.
- **Disadvantages**:
    - Can be expensive.
    - Limited capability to handle metadata, which means it needs to be dealt with at the application or database level.

Object Storage

- **Definition**: Object storage manages data and links it to associated metadata.



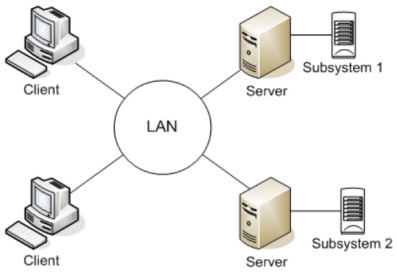
- **Structure**:
  - Object storage, also known as object-based storage, is a flat structure with data broken into discrete units called objects, kept in a single repository instead of being stored as files in folders or blocks.
- **Object Storage Volumes**:

    - Object storage volumes work as modular units:
        - Each is a self-contained repository that owns the data, a unique identifier that allows the object to be found over a distributed system, and metadata that describes the data.
- **Metadata**:
    - Metadata includes details at two levels:
        - General details like age, privacy/security settings, access contingencies, etc.
        - Custom information about the object (data) itself.
- **Data Retrieval**:
    - To retrieve the data, the storage operating system uses the metadata and identifiers, which distributes the load better and allows administrators to apply policies that perform more robust searches.

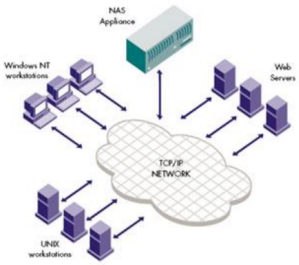
Feature	Block Storage	File Storage	Object Storage
Data Organization	Data is chopped into blocks with unique identifiers.	Data is stored as files in folders.	Data is broken into discrete units called objects.
Retrieval	Blocks are reassembled by storage software.	Computer navigates through folder structure to find files.	Storage operating system uses metadata and identifiers for retrieval.
Metadata	Limited metadata for each block.	Limited metadata for each file.	Rich metadata for each object, including custom information.
Structure	Chunks data into evenly sized volumes.	Organizes data as a hierarchy of files in folders.	Flat structure with data broken into discrete objects.
Environment	Typically used with SAN environments.	Commonly used with NAS or DAS.	Suitable for distributed systems.

Scaling	Typically scale-up.	Typically scale-out.	Scales horizontally, distributing load better.
Use Cases	Ideal for enterprises with large databases and big transactions.	Suited for general-purpose file storage needs.	Well-suited for applications with vast amounts of unstructured data.

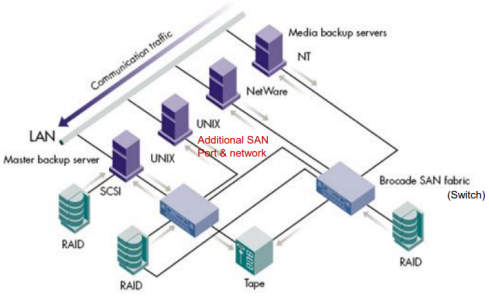
Directly Attached Storage (DAS)



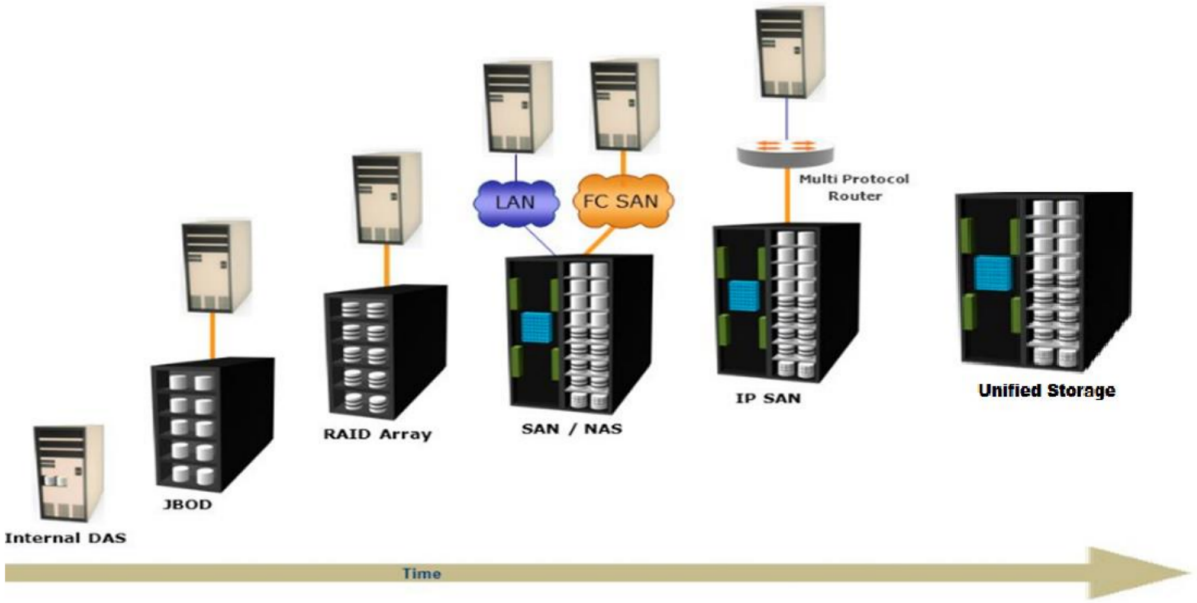
Network Attached Storage (NAS)



Storage Area Network (SAN)

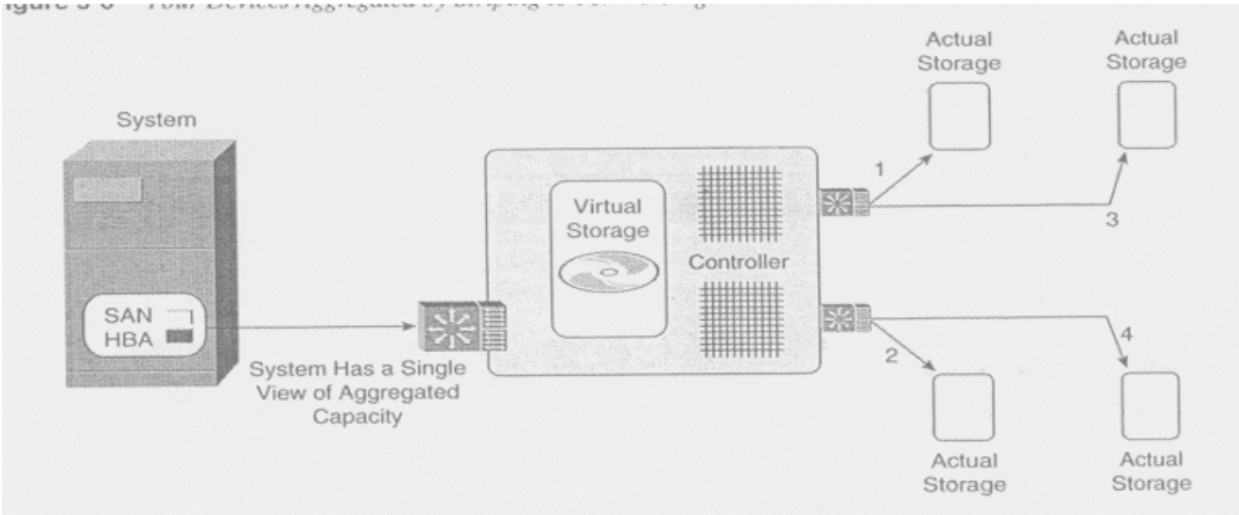


Evolution of Storage System



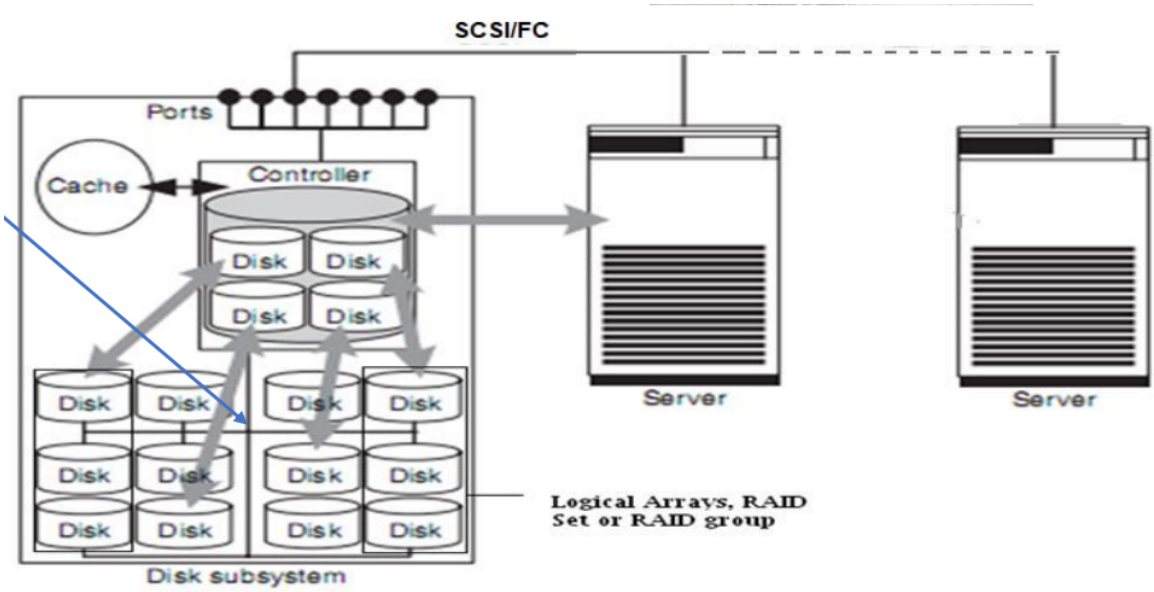
Typical Architecture of a Storage Device/Subsystem in Cloud Computing

- **Controller Placement**:
  - The figure shows a controller positioned between the disks and the ports.
  - External ports are extended to disks through internal I/O channels.



- **Controller Functions**:
  - Increase data availability and access performance through RAID (Redundant Array of Independent Disks).
  - Utilize caches to speed up read and write access to the server.
- **Components**:
  - Redundant controllers to ensure high availability and fault tolerance.
  - Significant cache for improved performance.
  - Storage disks capable of supporting petabytes of data.
  - Consolidated disks to provide better utilization of storage capacity.
- **Size and Weight**:

- The storage subsystem could weigh over a ton and have the size of a large wardrobe.

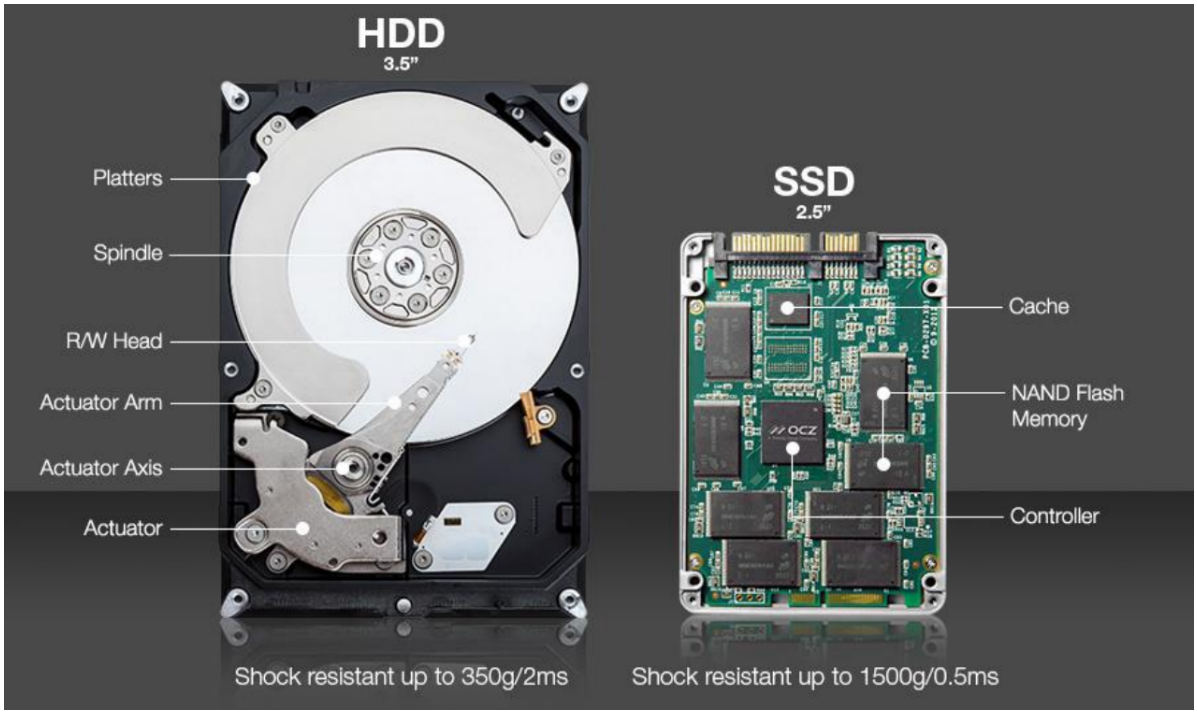


RAID (Redundant Array of Independent Disks) in Cloud Computing:

- **Definition**:
  - RAID is a data storage virtualization technology that combines multiple physical disk drive components into one or more logical units for data redundancy and performance improvement.
- **Data Distribution**:
  - Data is distributed across the drives in one of several ways, referred to as RAID levels, depending on the required level of redundancy and performance.
- **RAID Levels**:
  - Common RAID levels include RAID 0, RAID 1, RAID 4, and others.

For example:

- RAID 0: Striping without parity or mirroring.
- RAID 1: Mirroring without striping or parity.
- RAID 4: Block-level striping with dedicated parity.

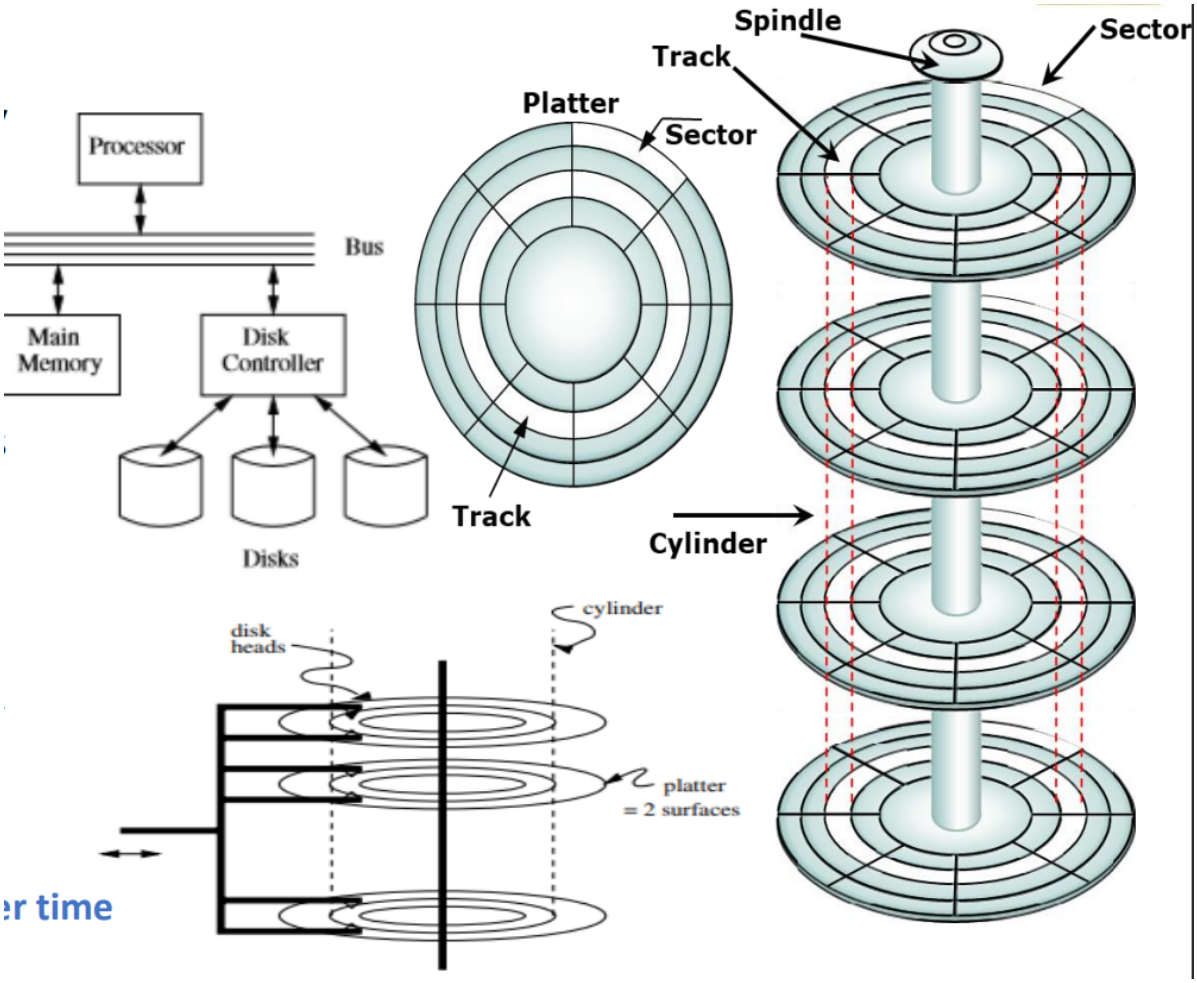


Reading or writing a block involves three steps:

1. **Seek Time**:
    - The disk controller positions the head assembly at the cylinder containing the track on which the block is located.
    - The time taken for this positioning is the seek time.
2. **Rotational Latency**:
    - The disk controller waits while the first sector of the block moves under the head.
    - This time is called the rotational latency.
3. **Transfer Time**:
    - All the sectors and the gaps between them pass under the head while the disk controller reads or writes data in these sectors.

- This delay is called the transfer time.

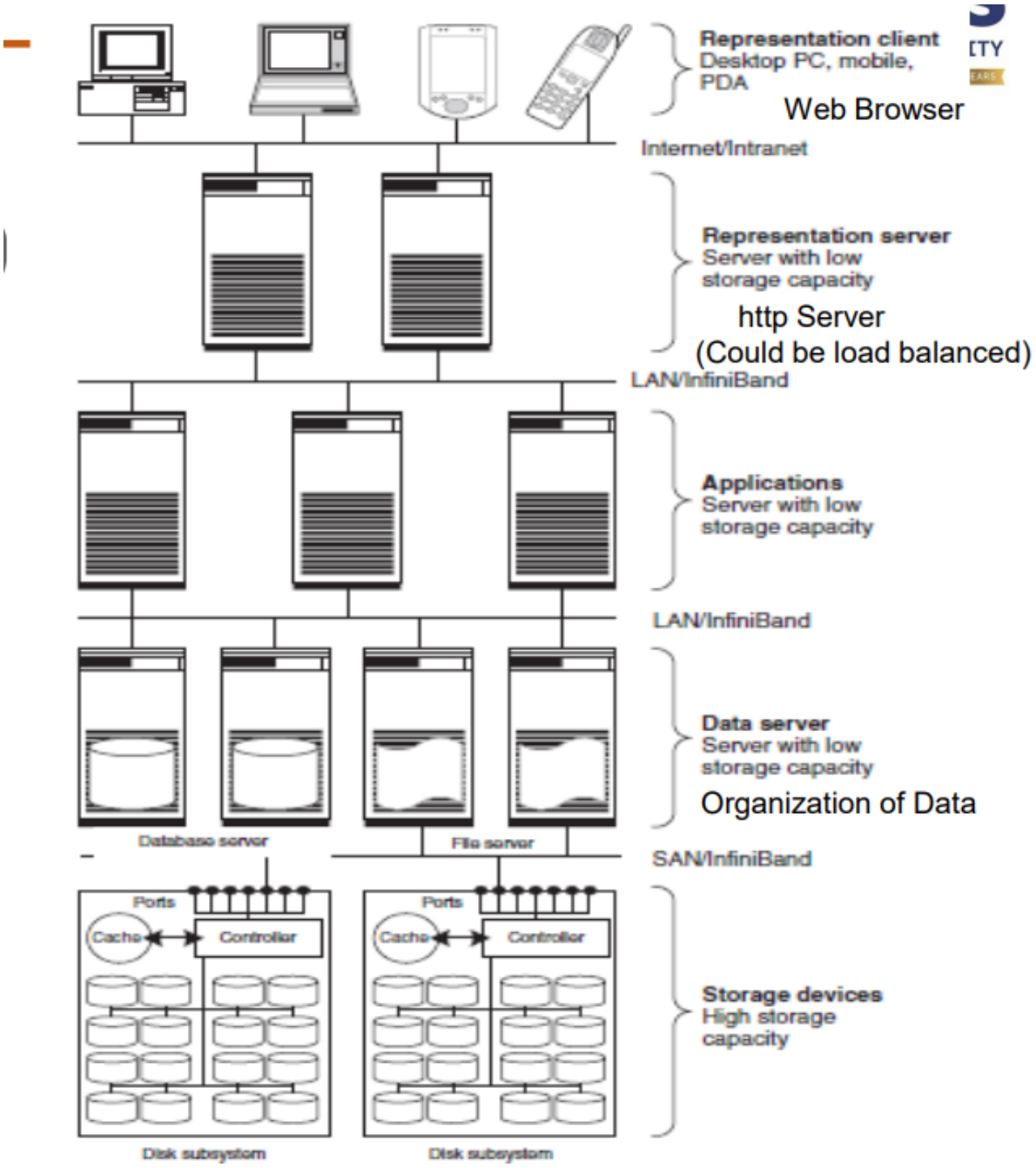
Therefore, Disk Latency = Seek Time + Rotational Latency + Transfer Time.



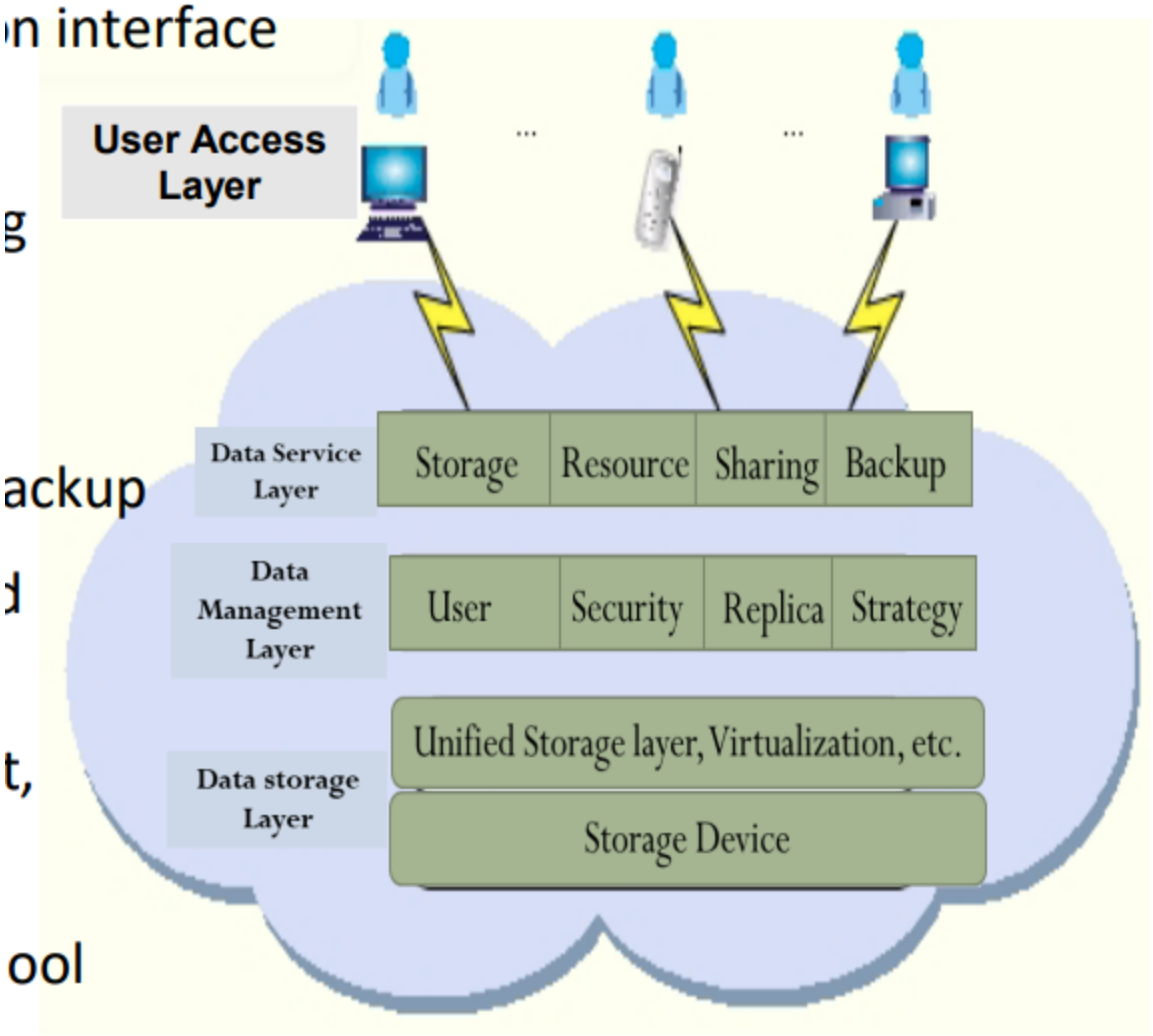
Cloud Storage:

- **Definition**:
  - Cloud Storage provides an ability for applications running somewhere to save, write, or read data in an off-site location.
  - This data can be accessed either through the public internet or a dedicated private network connection from anywhere.
- **Storage Virtualization**:
  - Storage virtualization supports access, utilization, availability, and other features needed by applications.

- **Terminology**:
  - Private cloud, public cloud, hybrid cloud, internal cloud, external cloud can be applied to storage based on where the storage is located.
  - However, ultimately, it's storage systems having the capability of virtual storage pools and multi-tenancy.
- **Infrastructure**:
  - Cloud Storage infrastructure includes both hardware and software cloud components.
  - Object-based storage is the prominent approach, and access to the infrastructure is via web services API.
- **Responsibilities**:
  - Cloud storage providers are responsible for:
    - Keeping the data available and accessible.
    - Protecting and running the physical environment.
- **Usage**:
  - People and organizations buy or lease storage capacity from the providers to store user, organization, or application data.
- **Virtual Storage Pool**:
  - Cloud storage or cloud-enabled storage can be visualized as a virtual storage pool.
- **Examples**:
  - Object storage services like Amazon S3 and Microsoft Azure Storage.
  - Object storage software like OpenStack Swift.
  - Object storage systems like EMC Atmos, EMC ECS, and Hitachi Content Platform.
  - Distributed storage research projects like OceanStore and VISION Cloud.



Architecture of a cloud storage platform



1. User Access Layer:
An authorized user can log into the cloud storage platform from any location via a standard public application interface and access cloud storage.

2. Data Service Layer:
This layer deals directly with users. Depending on user demands, different application interfaces can be developed to provide services such as:

- Data storage
- Space leasing
- Public resource
- Multi-user data sharing

- Data backup

3. Data Management:
This layer provides the upper layer with a unified public management interface for different services. It includes functions such as:

- User management
- Security management
- Replica management
- Strategy management

4. Data Storage:
Data stored in the system forms a massive pool and needs to be organized. This is the core of the cloud storage platform.

Cloud Storage Requirements

1. Dramatic Reduction in TCO:
Cloud storage cuts storage cost by more than 10 times compared to block or file storage.

2. Unlimited Scalability:
Cloud storage, built using distributed technologies, has unlimited scalability. It allows you to seamlessly add or remove storage systems from the pool.

3. Elasticity:
Storage virtualization in cloud storage decouples and abstracts the storage pool from its physical implementation. This results in a virtual, elastic (can grow and shrink as required), and unified storage pool.

4. On-Demand:
Cloud storage uses a pay-as-you-go model, where you pay only for the data stored and the data accessed. For a private cloud, there is a minimal cluster to start with, beyond which it is on-demand. This can result in huge cost savings for the storage user.

5. Universal Access:
Traditional storage has limitations like for block storage, the server needing to be

on the same SAN. But Cloud storage offers flexibility on the number of users and from where to access the same.

6. Multitenancy:
Cloud Storage is typically multi-tenant and supports centralized management, higher storage utilization, and lower costs.

7. Data Durability and Availability:
Cloud storage runs on commodity hardware but is still highly available even with partial failures of the storage system. This is supported by a software layer providing the availability.

8. Usability:
Cloud storage is designed to be user-friendly and requires minimal setup.

9. Disaster Recovery:
Cloud storage can be used as a backup plan by businesses, allowing for a quick recovery of data.

Cloud Storage Enablers

1. Storage Virtualization:
Storage, like compute resources, is enabled for the cloud by virtualization. Storage virtualization could be implemented in hardware or software. It could also be implemented in the server, in the storage device, and in the network carrying the data.

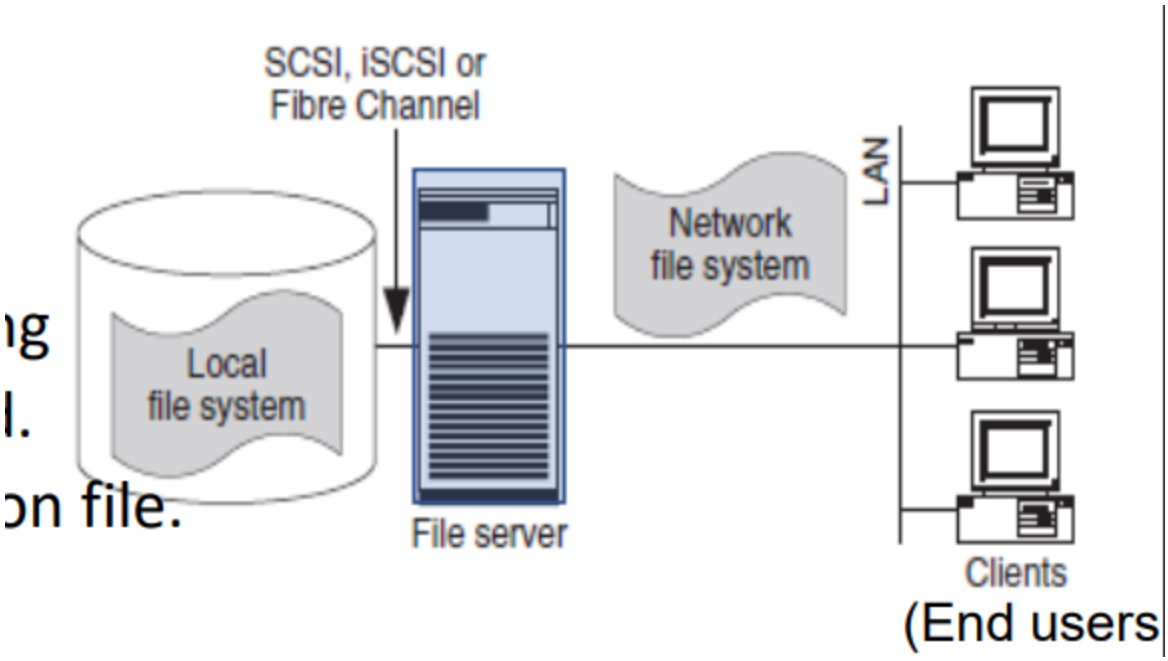
2. Server Storage Virtualization:
Techniques and components like File Systems, Volume Manager, Logical Volume Managers enable storage virtualization in the server.

3. Storage Device Virtualization:
Techniques like RAID (Redundant Array of Independent Disks) and Logical Volume Management are also used with storage virtualization in the storage device.

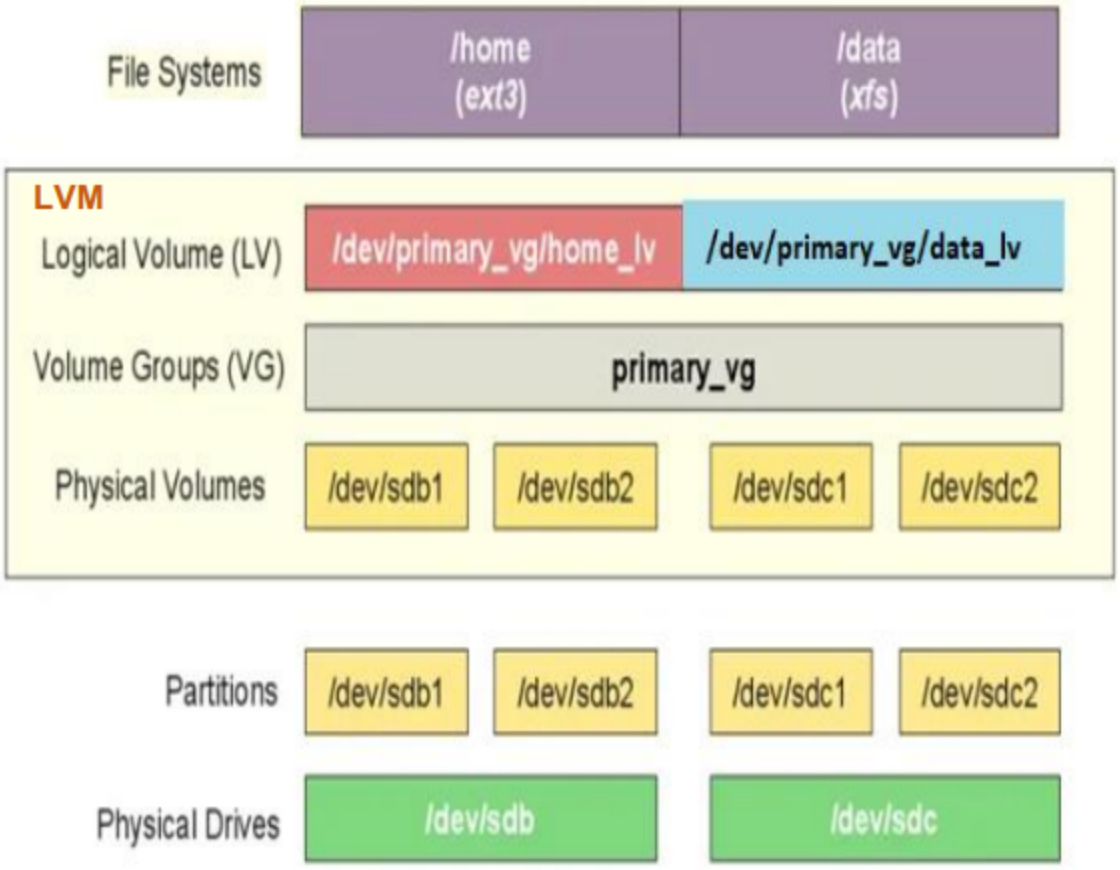
Enablers for Storage Virtualization – File Systems:

- **File System Basics**:
    - A block is the basic unit of storage for IO (Read/Write) operations.
    - Data is separated and grouped into pieces and given a name called a file.

    - File systems manage the groups of information (files) and their names, controlling how data is stored and retrieved.
    - Without a file system, information placed in a storage medium would be one large body of data with no way to tell where one piece of information stops and the next begins.
- **Local File Systems**:
    - File systems can be used on numerous different types of storage devices that use different kinds of media.
    - There are many different kinds of file systems.
    - Some file systems are used on local data storage devices, while others provide file access via a network interface and are responsible for arranging storage space.
- **Network File Systems (NFS)**:
    - Network file systems make local files and directories available over the LAN, allowing several end users to work on common files.
    - NFS supports applications to share and access files remotely from various computers.
    - NFS functionality, in a typical server dedicated to hosting files and directories and making them available across the network, has a daemon process called `nfsd`.
    - The server administrator exports the directories and advertises them in a configuration file.
    - NFS client requests to the exported directories with the mount command.
    - Once mounted, it's transparent where the data is being accessed and can be controlled by permissions.

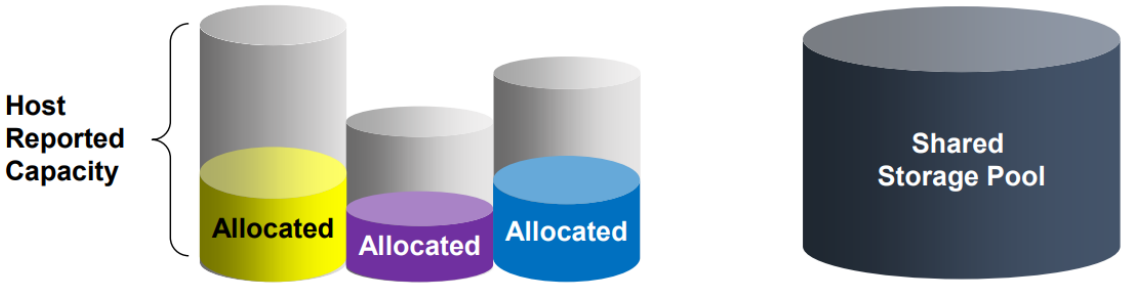


- **Logical Volume Manager (LVM)**:
  - LVM provides an independent layer between the File System and the disk drives.
  - It allows you to:
    - Create partitions on the physical disk and create physical volumes.
    - Group these physical volumes into a volume group.
    - Break up this volume group into logical volumes.
    - Create a file system on these logical volumes and mount them.

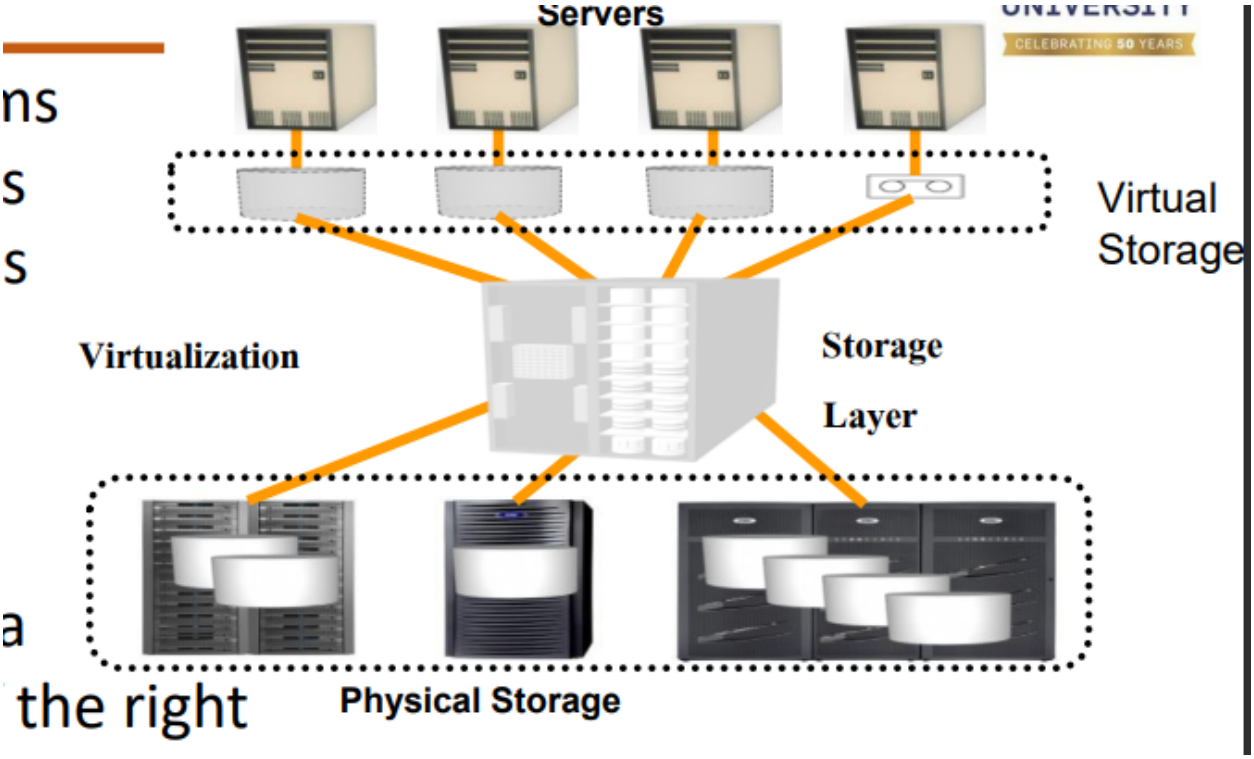


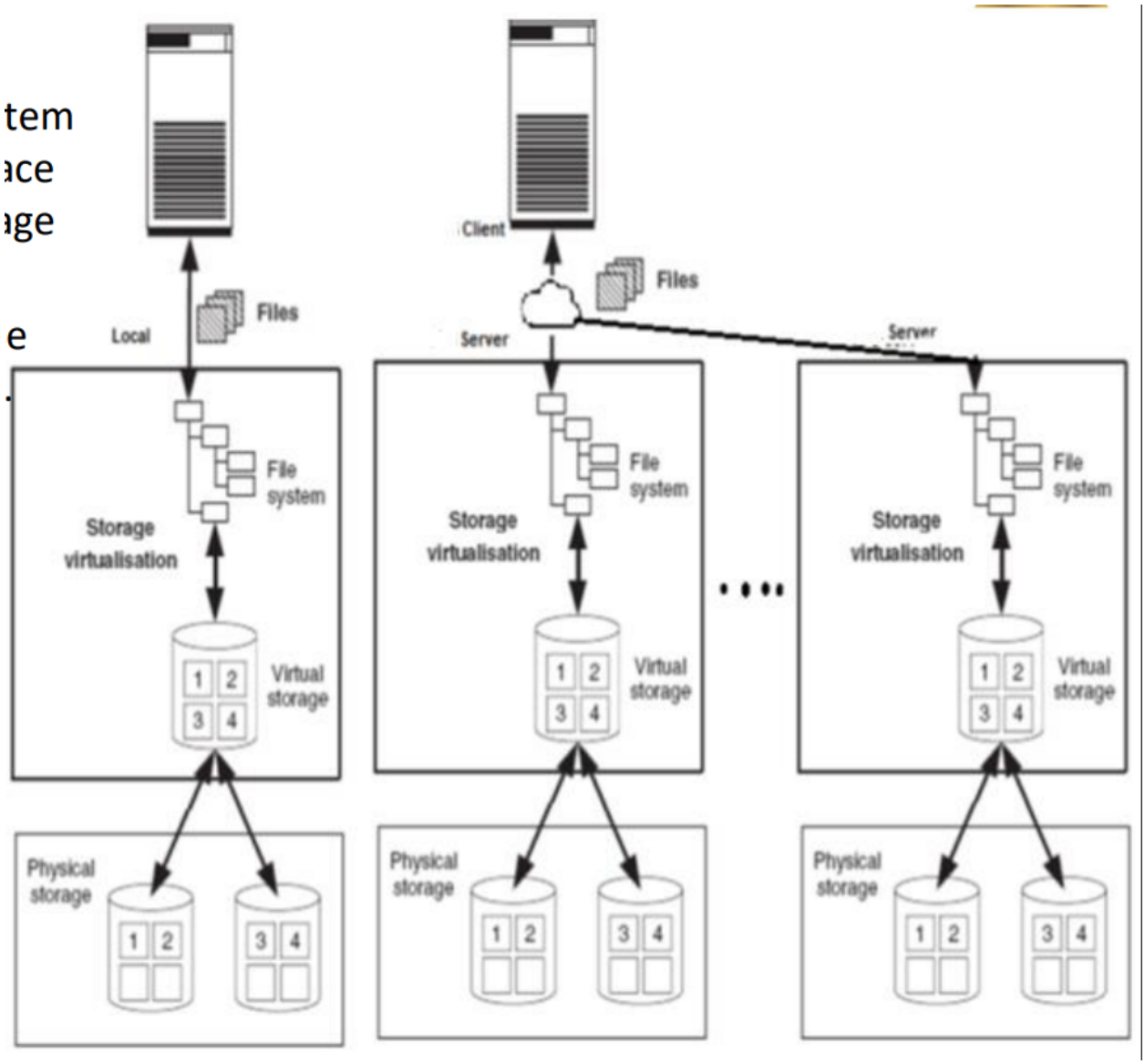
Enablers for Storage Virtualization – Logical Volume Manager (LVM):

- **Thin Provisioning or Virtual Provisioning**:
  - Capacity-on-demand from a shared storage pool.
  - Logical units presented to hosts have more capacity than physically allocated (the physical resources are thinly or virtually provisioned).
  - Physical storage is allocated only when the host requires it.
  - Provisioning decisions are not bound by currently available storage.



Storage – Virtualization





- **Definition**:
  - Storage virtualization abstracts physical storage subsystems (disks, tapes, etc.) from the user's application and presents them as logical entities.
  - It hides the underlying complexity of storage subsystems and the nature of access, network, or changes to physical devices.
- **Aggregation of Capacity**:
  - It aggregates the capacity of multiple storage devices into storage pools.

- Aggregates multiple resources as one addressable entity (pool) or divides a resource into multiple addressable entities, enabling easy provisioning of the right storage for performance or cost.

- **Examples**:
  - Examples include creating single virtual large disks from multiple small disks or many smaller virtual disks from a large disk.
- **Location Independence**:
  - Virtualization of storage helps achieve location independence by abstracting the physical location of the data.
  - The virtualization layer presents to the user a logical space for data storage and handles the mapping to the actual physical location.
- **Responsibilities of Virtualization Software**:
  - Virtualization software maintains a consistent view of all mapping information and keeps it consistent.
  - Mapping information is often part of metadata.
- **Virtualization Layer**:
  - The virtualization layer can be implemented in hardware or software.

Storage – Categories of Storage Virtualization

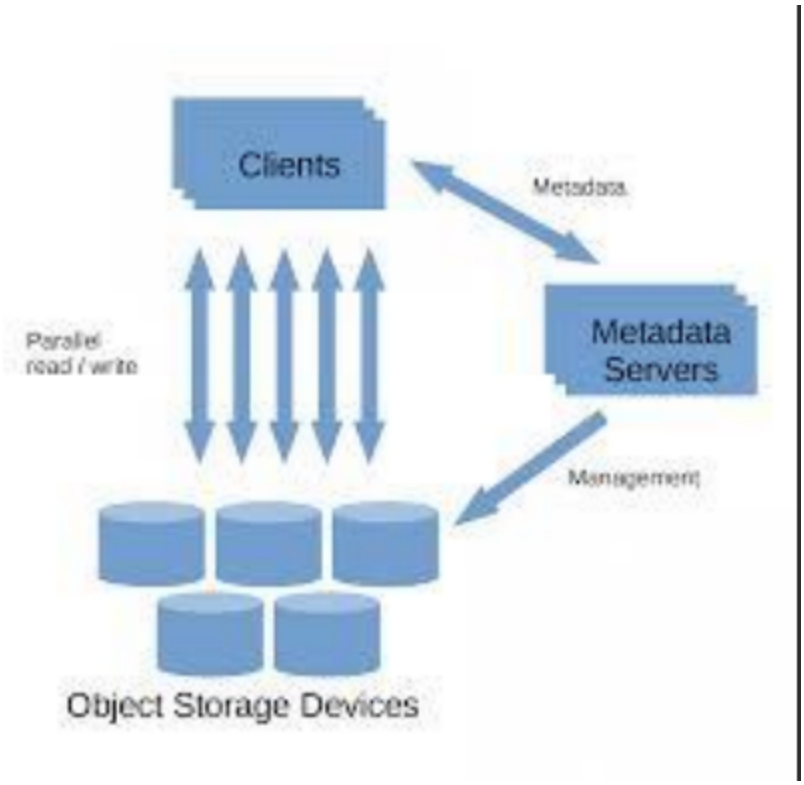
File-level Virtualization:

- A file system virtualization provides an abstraction of a file system to the application (with a standard file-serving protocol interface such as NFS or CIFS) and manages changes to distributed storage hardware underneath the file system implementation.
- Eliminates dependencies between the data accessed at the file level and the location where the files are physically stored.

- The virtualization layer manages files, directories, or file systems across multiple servers and allows administrators to present users with a single logical file system.
- A typical implementation of a virtualized file system is as a network file system that supports sharing of files over a standard protocol with one or more file servers enabling access to individual files.
- File-serving protocols typically employed are NFS, CIFS, and web interfaces such as HTTP or WebDAV.
- Advantages: This provides opportunities to optimize storage usage, server consolidation, and non-disruptive file migration.

Distributed File System (DFS):

- **Definition**:
  - A distributed file system (DFS) is a network file system wherein the file system is distributed across multiple servers.
  - DFS enables location transparency, file directory replication, and tolerance to faults.
  - Some implementations may also cache recently accessed disk blocks for improved performance.

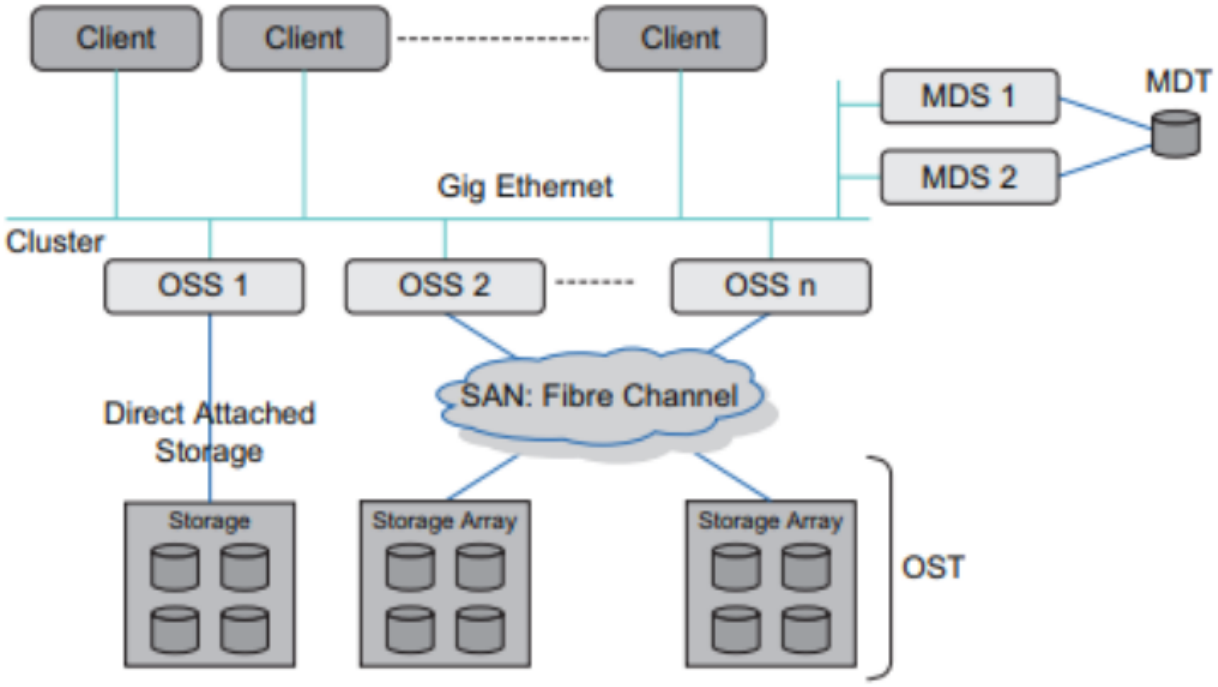


- **Management of Metadata**:
  - Efficient management of metadata is crucial for overall file system performance.
  - Two important techniques for managing metadata for highly scalable file virtualization:
    a. Separate data from metadata with a centralized metadata server (used in Lustre).
    b. Distribute data and metadata on multiple servers (used in Distributed File Systems with Centralized Metadata).
- **Centralized Metadata Management Scheme**:
  - Achieves scalable DFS with a dedicated metadata server to which all metadata operations performed by clients are directed.
  - Lock-based synchronization is used in every read or write operation from the clients.
  - In centralized metadata systems, the metadata server can become a bottleneck if there are too many metadata operations.

- For workloads with large files, centralized metadata systems perform and scale very well.

Lustre - Distributed File Systems with Centralized Metadata:

- **Overview**:
  - Lustre is a massively parallel, scalable distributed file system for Linux, which employs a cluster-based architecture with centralized metadata.
  - It is a software solution capable of scaling over thousands of clients for a storage capacity of petabytes with high-performance I/O throughput.
- **Architecture**:
  - Lustre architecture consists of three main functional components, which can be on the same nodes or distributed on separate nodes communicating over a network:
    1. Object storage servers (OSSes), which store file data on object storage targets (OSTs).
    2. A single metadata target (MDT) that stores metadata on one or more Metadata servers (MDS).
    3. Lustre Clients that access the data over the network using a POSIX interface.



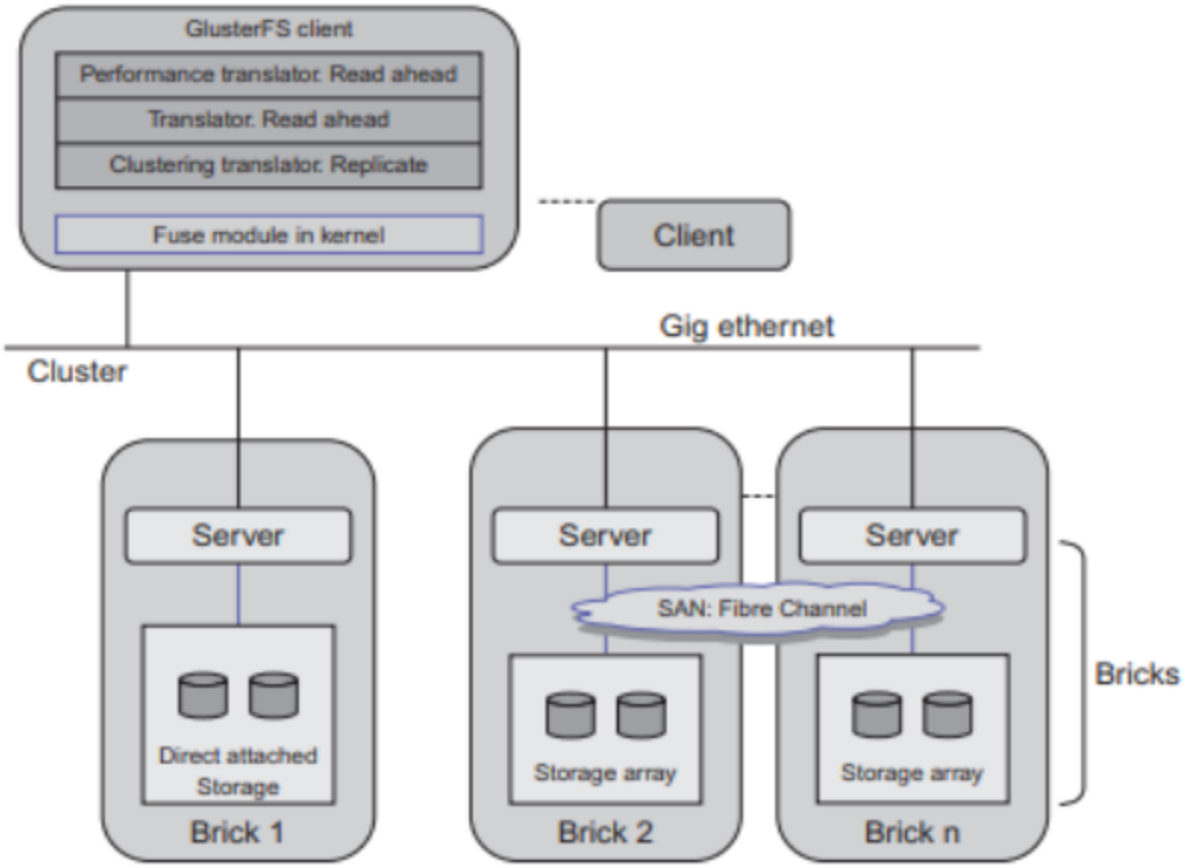
- **Functioning**:
  - When a client accesses a file, it does a filename lookup on an MDS.
  - Then, the MDS creates a metadata file on behalf of the client or returns the layout of an existing file.
  - The client then passes the layout to a logical object volume (LOV) for read or write operations.
  - The LOV maps the offset and size to one or more objects, each residing on a separate OST.
  - The client then locks the file range being operated on and executes one or more parallel read or write operations directly to the OSTs.

GlusterFS - Distributed File Systems with Distributed Metadata:

- **Overview**:
  - GlusterFS is an open-source, distributed cluster file system without a centralized metadata server.

- It is capable of scaling to thousands of clients, petabytes of capacity, and is optimized for high performance.

- **Architecture**:
  - GlusterFS employs a modular architecture with a stackable user-space design.
  - It aggregates multiple storage bricks on a network (over Infiniband RDMA or TCP/IP interconnects) and delivers as a network file system with a global namespace.



- **Components**:
  - It consists of just two major components: a Client and a Server.
  - The Gluster server clusters all the physical storage servers and exports the combined diskspace of all servers as a Gluster File System.

    - The Gluster client is optional and can be used to implement highly available, massively parallel access to every storage node and handles failure of any single node transparently.

- **Functioning**:
    - GlusterFS uses the concept of a storage brick consisting of a server that is attached to storage directly (DAS) or through a SAN.
    - Local file systems (ext3, ext4) are created on this storage.
    - Gluster employs a mechanism called translators to implement the file system capabilities.
    - Translators are programs inserted between the actual content of a file and the user accessing the file as a basic file system interface.
    - Each translator implements a particular feature of GlusterFS, and they can be loaded both in client and server side appropriately to improve or achieve new functionalities.
    - GlusterFS supports file replication with the Automatic File Replication (AFR) translator, which keeps identical copies of a file/directory on all its subvolumes.

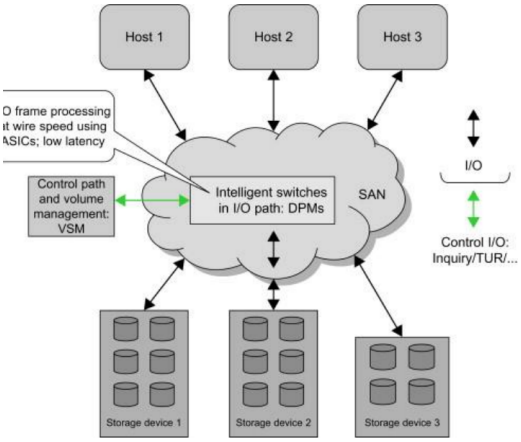
Block Virtualization

Block level Virtualization:

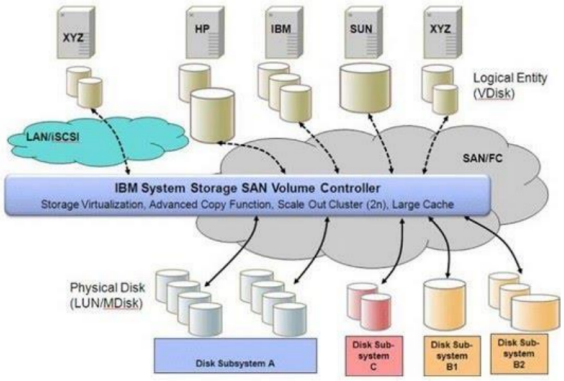
- Virtualizes multiple physical disks and presents them as a single logical disk.
- Data blocks are mapped to one or more physical disk subsystems, appearing as a single logical storage device.

Block level storage virtualization can be performed at three levels:
a. Host-Based
b. Storage Device level
c. Network level

Switch-based network virtualization



Appliance-based network virtualization



Host-Based block virtualization:

- Uses a Logical Volume Manager (LVM) to create a storage pool by combining multiple disks.
- Allows transparent allocation and management of disk space for file systems or raw data.

Storage-Device level virtualization:

- Creates Virtual Volumes over the physical storage space of the specific storage subsystem.
- Storage disk arrays provide this form of virtualization using RAID techniques.
- Array controllers create Logical Units (LUNs) spanning across multiple disks in the array in RAID Groups.

Network-Based block virtualization:

- Most commonly implemented form of scalable virtualization.
- Implemented within the network connecting hosts and storage, such as a Fibre Channel Storage Area Network (SAN).

Switch-based network virtualization:

- Virtualization functionality is implemented within the network, either in switches or routers.

- Switch-based network virtualization occurs in an intelligent switch in the fabric, working in conjunction with a metadata manager in the network.

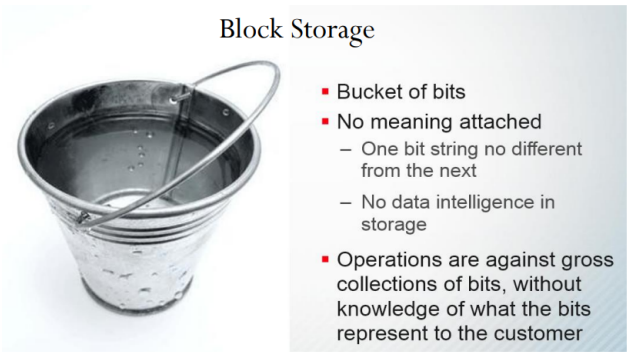
Appliance-based network virtualization:

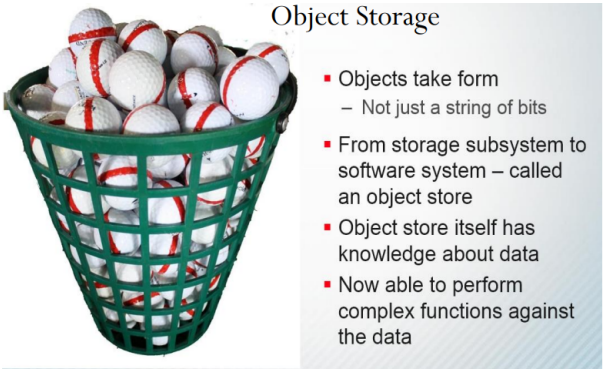
- I/O flows through an appliance that controls the virtualization layer.
- Example: IBM SAN Volume Controller.

Variations of appliance-based implementation:

- In-band: All I/O requests and their data pass through the virtualization device.
- Out-of-band: The appliance comes in between for metadata management, while the data path is directly from the client to each host.

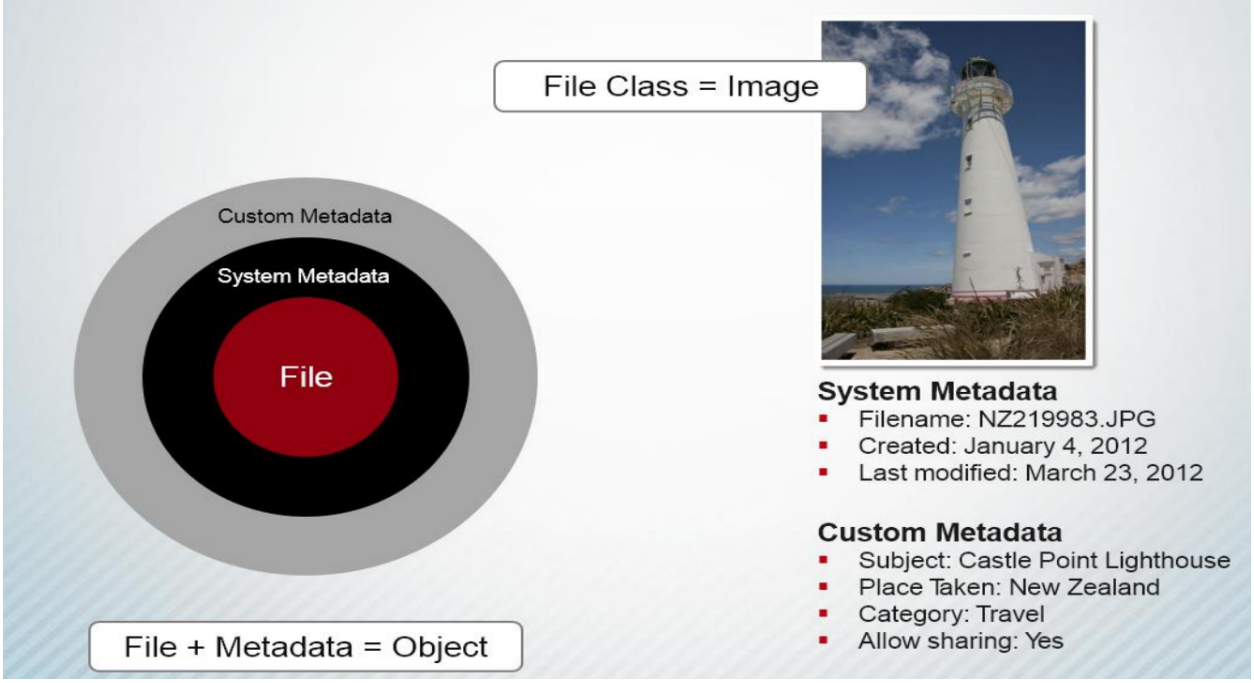
Object Storage





- **Overview**:
  - Object storage is a prominently used approach in building cloud storage systems.
  - It differs from block or file storage as it allows users to store data in the form of objects, essentially files in a logical view, by virtualizing the physical implementation in a flat namespace, eliminating name collisions using REST HTTP APIs.
- **Key Characteristics**:
  - Data manipulated using GET, PUT, DELETE, and UPDATE operations.

- Uses Representational State Transfer (REST) APIs, which are a distillation of the way the Web already works.
- Resources are identified by uniform resource identifiers (URIs) and manipulated through their representations.
- Messages are self-descriptive and stateless (XML).
- Multiple representations are accepted or sent.
- Objects contain additional descriptive properties that can be used for better indexing or management.



- **Architecture**:
  - Built using scale-out distributed systems where each node runs on a local filesystem.
  - Does not require specialized and expensive hardware; architecture allows for the use of commodity hardware.
- **Critical Tasks**:
  - **Data Placement**:
    - Objects are stored across multiple nodes.

- **Automating Management Tasks**:
  - Includes ensuring durability and availability.
  - Object storage software takes care of the durability model by:
    - Creating multiple copies of the object and chunking it.
    - Creating erasure codes.
- **Management**:
  - Supports activities such as periodic health checks, self-healing, and data migration.
  - Management is made easy by using a single flat namespace, allowing a storage administrator to manage the entire cluster as a single entity.
  - Erasure coding (EC) is a method of data protection in which data is broken into fragments, expanded and encoded with redundant data pieces and stored across a set of different locations or storage media.

- **Operation**:
  - Typically, a user sends their HTTP GET, PUT, POST, HEAD, or DELETE request to any one node from a set of nodes.
  - The request is translated to physical nodes by the object storage software.

Amazon Simple Storage Service (S3)

- **Overview**:
  - Amazon S3 is a highly reliable, highly available, scalable, and fast storage service in the cloud for storing and retrieving large amounts of data through simple web services.
  - There are three ways of using S3:
    - AWS console: Most common operations can be performed via the AWS console, which provides a GUI interface to AWS.

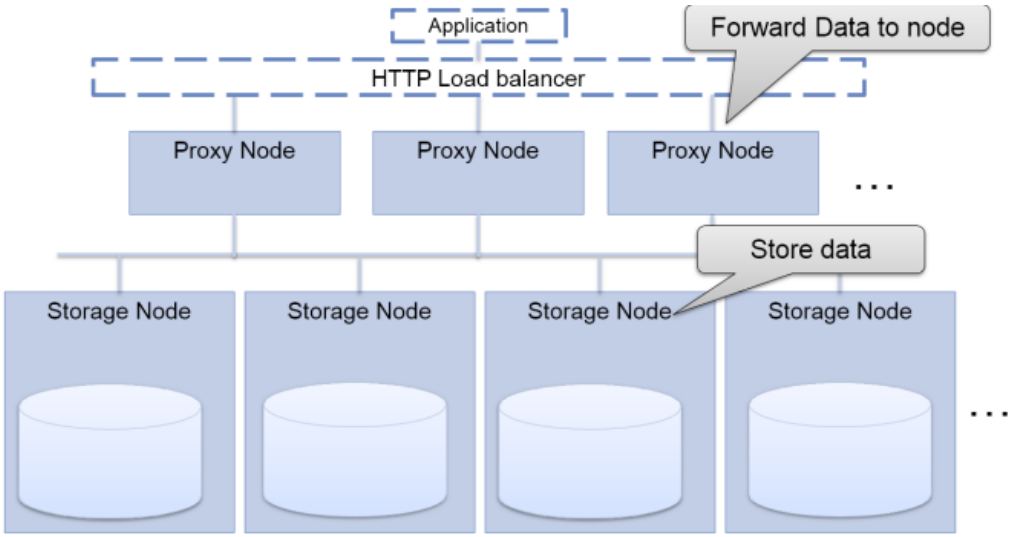
        - RESTful API: For use within applications, Amazon provides a RESTful API with familiar HTTP operations such as GET, PUT, DELETE, and HEAD.
        - Libraries and SDKs: There are libraries and SDKs for various languages that abstract these operations.
    - Several S3 browsers exist that allow users to explore their S3 account as if it were a directory. There are also file system implementations that let users treat their S3 account as just another directory on their local disk.
- **Organizing Data in S3: Buckets, Objects, and Keys**:
    - Data is stored as objects in S3.
    - Objects are stored in resources called buckets.
    - S3 objects can be up to 5 Terabytes in size, and there are no limits on the number of objects that can be stored.
    - Objects in S3 are replicated across multiple geographic locations to make it resilient to several types of failures.
    - Objects are referred to with keys, which are basically an optional directory path name followed by the name of the object.
    - If object versioning is enabled, recovery from inadvertent deletions and modifications is possible.
- **Buckets**:
    - Buckets provide a way to keep related objects in one place and separate them from others.
    - There can be up to 100 buckets per account and an unlimited number of objects in a bucket.
    - Each object has a key, which can be used as the path to the resource in an HTTP URL.
    - By convention, slash-separated keys are used to establish a directory-like naming scheme for convenient browsing in S3.

- **Security:**
  Users can ensure the security of their S3 data by two methods
  - **Access control to objects:** Users can set permissions that allow others to access their objects. This is accomplished via the AWS Management Console.
  - **Audit logs:** S3 allows users to turn on logging for a bucket, in which case it stores complete access logs for the bucket in a different bucket. This allows users to see which AWS account accessed the objects, the time of access, the IP address from which the accesses took place and the operations that were performed. Logging can be enabled from the AWS Management Console
- **Data Protection**
  1. **Replication**:
     - By default, S3 replicates data across multiple storage devices and is designed to survive two replica failures.
     - Reduced Redundancy Storage (RRS) option is available for noncritical data. RRS data is replicated twice and is designed to survive one replica failure.
     - S3 provides strong consistency if used in that mode and guarantees consistency among the replicas.
  2. **Regions**:
     - For performance, legal, availability, and other reasons, it may be desirable to have S3 data running in specific geographic locations.
     - This can be accomplished at the bucket level by selecting the region that the bucket is stored in during its creation.
  3. **Versioning**:
     - If versioning is enabled on a bucket, then S3 automatically stores the full history of all objects in the bucket from that time onwards.
     - The object can be restored to a prior version, and even deletes can be undone. This guarantees that data is never inadvertently lost.

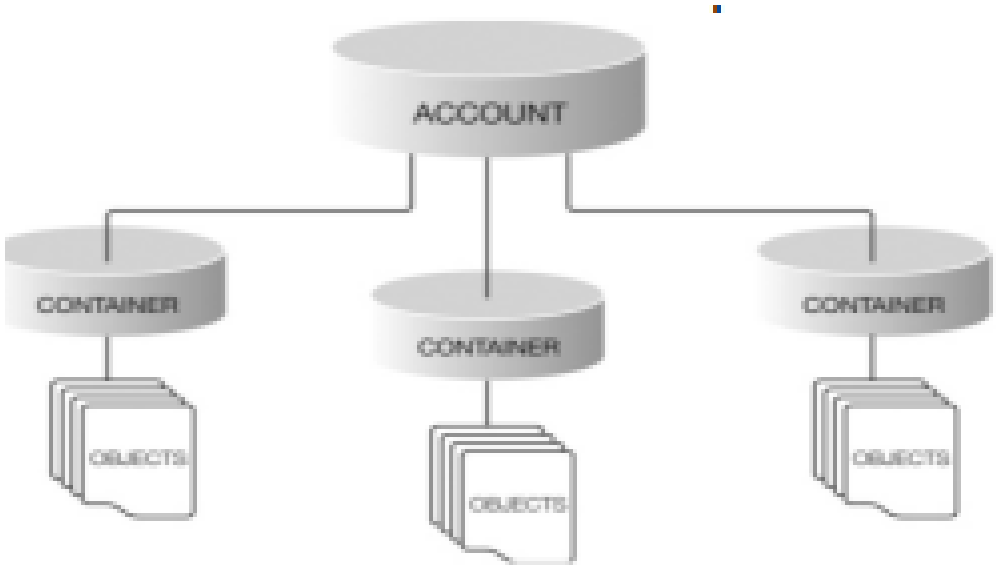
Large Objects and Multi-part Uploads:

- S3 provides APIs that allow the developer to write a program that splits a large object into several parts and uploads each part independently.
- These uploads can be parallelized for greater speed to maximize network utilization. If a part fails to upload, only that part needs to be re-tried.

Swift



- **Definition**:
  - Swift partitions break the storage available into locations where data will be located, including account databases, container databases, or objects.
  - They are the core of the replication system and are distributed across all disks.



- **Analogy**:
  - It's like a moving bin in a warehouse.
  - These are not Linux partitions.
  - Adding a node leads to reassigning of partitions.
- **Components**:
  - **Account**:
    - An account is a user in the storage system.
    - Unlike volumes, Swift creates accounts which enable multiple users and applications to access the storage system at the same time.
  - **Container**:
    - Containers are where accounts create and store data.
    - Containers are namespaces used to group objects, conceptually like directories.
  - **Object**:
    - Actual data is stored on the disk. This could be photos, documents, etc.
- **Ring**:

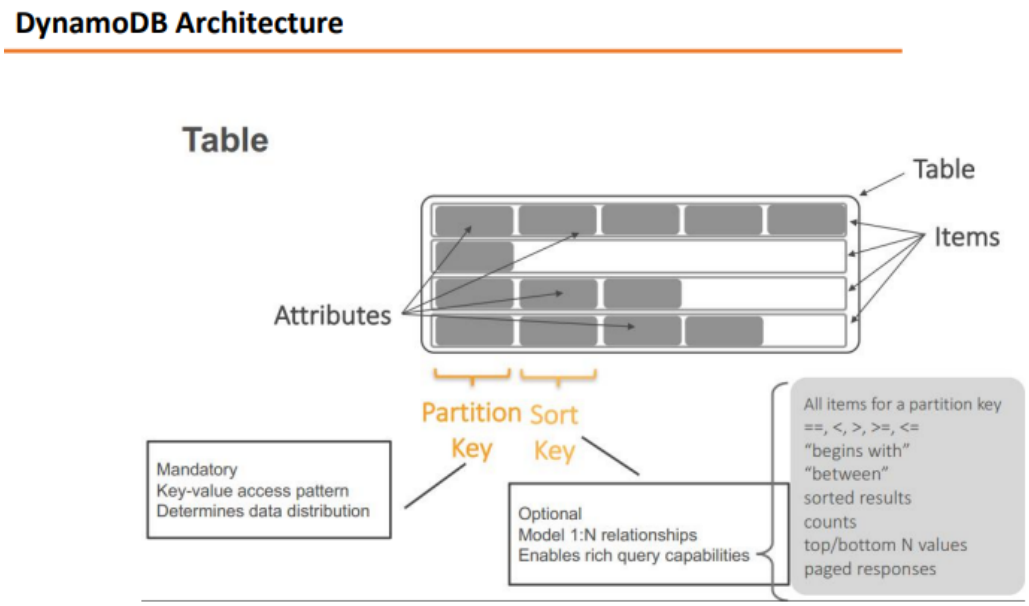
- The ring maps the partition space to physical locations on disk.
- It's like an encyclopedia; instead of letters, Swift uses hash for searching.

NoSQL Database - DynamoDB

- **Definition**:
  - DynamoDB is a NoSQL fully managed cloud-based document and Key-Value database available through Amazon Web Services (AWS).
  - All data in DynamoDB is stored in tables that you have to create and define in advance, though tables have some flexible elements and can be modified later.
  - DynamoDB requires users to define only some aspects of tables, most importantly the structure of keys and local secondary indexes, while retaining a schema-less flavor.
  - DynamoDB enables users to query data based on secondary indexes too, rather than solely on the basis of a primary key.
  - DynamoDB supports only item-level consistency, which is analogous to row-level consistency in RDBMSs.
  - If consistency across items is a necessity, DynamoDB is not the right choice.
  - DynamoDB has no concept of joins between tables. A table is the highest level at which data can be grouped and manipulated, and any join-style capabilities that you need will have to be implemented on the application side.
- **Core Components**:
  - In DynamoDB, tables, items, and attributes are the core components.
  - A table is a collection of items, and each item is a collection of attributes.
  - DynamoDB uses primary keys to uniquely identify each item in a table and secondary indexes to provide more querying flexibility.

- **Primary Keys**:
  - DynamoDB supports two different kinds of primary keys:
    - **Partition Key**:
      - A simple primary key, composed of one attribute known as the partition key.
      - DynamoDB uses the partition key's value as input to an internal hash function. The output from the hash function determines the partition (physical storage internal to DynamoDB) in which the item will be stored.
    - **Partition Key and Sort Key**:
      - Referred to as a composite primary key, this type of key is composed of two attributes.
      - The first attribute is the partition key, and the second attribute is the sort key.
      - All items with the same partition key value are stored together in sorted order by sort key value.
- **Secondary Indexes**:
  - Users can create one or more secondary indexes on a table.
  - A secondary index lets users query the data in the table using an alternate key, in addition to queries against the primary key.
- **Partitions and Data Distribution**:
  - DynamoDB stores data in partitions.
  - A partition is an allocation of storage for a table, backed by solid-state drives (SSDs), and automatically replicated across multiple Availability Zones within an AWS Region.
  - Partition management is handled entirely by DynamoDB.

DynamoDB Architecture

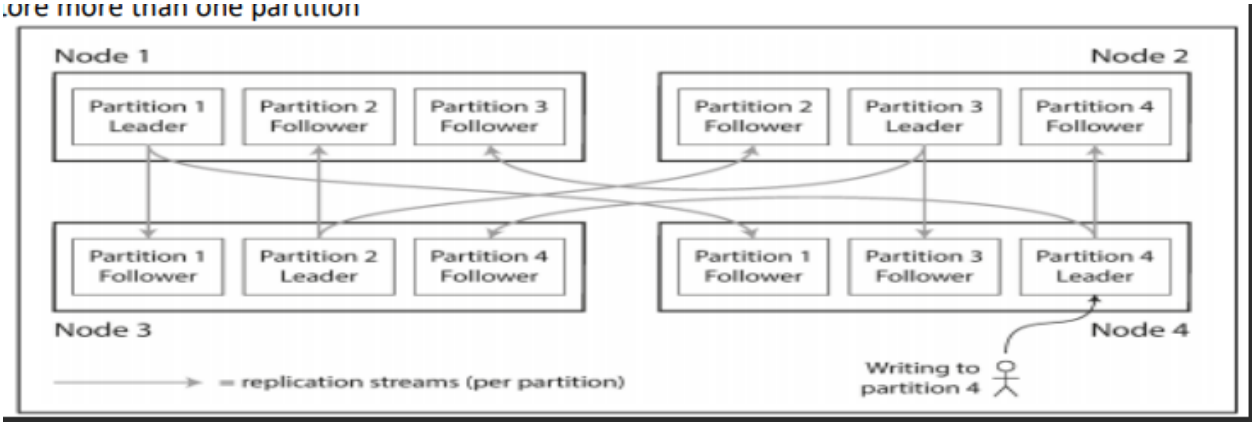


Partitioning

- **Definition**:
  - Cloud applications typically scale using elastic computing resources to address variations in workload. However, this can lead to a bottleneck in the backend if the same data store is used.
  - Partitioning is a method of intentionally breaking a large volume of data into smaller partitions, where each partition can be placed on a single node or distributed across different nodes in a cluster. This allows query or IO operations to be distributed across multiple nodes, supporting the performance and throughput expectations of applications.
  - Partitions are defined so that each piece of data (e.g., each record, row, or document in a database) belongs to exactly one partition. Many operations may simultaneously affect partitions.
  - Each node can independently execute queries or perform IO operations on its own partition, enabling scaling of throughput with the addition of more nodes.

- For example, partitioning a large piece of data to be written to disk into multiple partitions and distributing these partitions to multiple disks will lead to better total IO performance.

- **Benefits**:
  - Large complex queries or large IO workloads can potentially be parallelized across many nodes.
  - Partitioning is usually combined with replication so that copies of each partition are stored on multiple nodes. This ensures fault tolerance, even though each record belongs to exactly one partition and may still be stored on several different nodes.
- **Terminologies**:
  - **Goal of Partitioning**:
    - The goal of partitioning is to spread the data and query load evenly across nodes.
  - **Skew**:
    - If some partitions have more data than others, it is called skew.
    - Skew makes partitioning much less effective. In an extreme case, all the load could end up on one partition.
  - **Hot Spot**:
    - A partition with disproportionately high load is called a hot spot.



Different Approaches of Partitioning

1. **Vertical Partitioning**
    - In this approach, data is partitioned vertically.
    - Also known as column partitioning, where a set of columns are stored on one data store and others to a different data store (could be on a different node), and data is distributed accordingly.
    - In this approach, no two critical columns are stored together, which improves performance.
2. **Workload Driven Partitioning**
    - Data access patterns generated from the application are analyzed, and partitions are formed according to that.
    - This improves the scalability of transactions in terms of throughput and response time.
3. **Partitioning by Random Assignment**
    - In this approach, records are randomly assigned to nodes, which is the simplest approach for avoiding hot spots.
    - This distributes the data evenly across the nodes.
    - The disadvantage is that when trying to read a particular item, you have no way of knowing which node it is on, so you have to query all nodes in parallel.
4. **Horizontal Partitioning**
    - This is a static approach of horizontally partitioning data to store it on different nodes.
    - Once it's partitioned, this would not change.
    - Techniques used for horizontal partitioning include:
        1. Partitioning by Key Range
        2. Partitioning using the Schema
        3. Partitioning using Graph Partitioning

4. Partitioning using Hashing

4.1 Partitioning by Key Range

- Range partitioning involves partitioning cloud data by using a range of keys, assigning a continuous range of keys to each partition (from some minimum to some maximum).
- The values of the keys should be adjacent but not overlapped.
- The range of keys is decided based on some conditions or operators.
- Knowing the boundaries between the ranges allows easy determination of which partition contains a given key.
- If we also know which partition is assigned to which node, we can make the request directly to the appropriate node.
- The ranges of keys are not necessarily evenly spaced because data may not be evenly distributed. Within each partition, we can keep keys in sorted order.

Example:

- Consider an application that stores data from a network of sensors, where the key is the timestamp of the measurement (year-month-day-hour-minute-second).
- Range scans are very useful when all the readings for a particular month need to be fetched.
- The disadvantage of key range partitioning is that certain access patterns can lead to hot spots.
- If the key is a timestamp, all writes end up going to the same partition (the one for today), leading to an overload of writes while other partitions are idle.
- To avoid this problem, we can use some other attribute other than the timestamp as the first element of the key.

4.2 Schema Based Partitioning

- Schema Partitioning is designed to minimize distributed transactions.
- In this approach, the database schema is partitioned so that related rows are kept in the same partition instead of separating them into different partitions.

4.3 Graph Partitioning

- Graph partitioning is a workload-based static partitioning method where partitions are determined by analyzing the pattern of data access.
- Once partitioning is done, the workload is not observed for changes, and there is no repartitioning.

4.4 Partitioning by Hash of Key

- Hash partitioning maps data to partitions based on a hashing algorithm applied to the partitioning key.
- The hashing algorithm evenly distributes rows among partitions, ensuring partitions are approximately the same size and avoiding skew and hot spots.
- It is an easy-to-use alternative to range partitioning, especially when data to be partitioned is not historical or lacks an obvious partitioning key.
- Using a suitable hash function for keys, each partition is assigned a range of hashes, and every key whose hash falls within a partition's range will be stored in that partition.

Different Approaches of Partitioning: Partitioning by Hash of Key (Continued)

- By using the hash of the key for partitioning, the ability to efficiently perform range queries is lost.
- One approach to circumvent this limitation is to use a concatenated index (or a composite key) approach, enabling one-to-many relationships.
- For example, on a social media site, if the primary key for updates is (user_id, update_timestamp), all updates made by a particular user within some time interval can be efficiently retrieved, sorted by timestamp.
- Different users may be stored on different partitions, but within each user, the updates are stored ordered by timestamp on a single partition.

Example:

```
CREATE PARTITION FUNCTION myRangePF1 (int)
AS RANGE LEFT FOR VALUES (1, 100, 1000);
GO
CREATE PARTITION SCHEME myRangePS1
AS PARTITION myRangePF1
TO (test1fg, test2fg, test3fg, test4fg);
GO
CREATE TABLE PartitionTable (col1 int, col2 char(10))
ON myRangePS1 (col1);
GO
```

- Partition `test1fg` will contain tuples with `col1` values <= 1.
- Partition `test2fg` will contain tuples with `col1` values > 1 and <= 100.
- Partition `test3fg` will contain tuples with `col1` values > 100 and <= 1000.
- Partition `test4fg` will contain tuples with `col1` values > 1000.

Distributed Hashing

- In distributed hashing, the hash table may need to be split into several parts and stored on different servers to work around the memory limitations of a single system.
- Data partitions and their keys (the hash table) are distributed among many servers.
- The mechanism for distributing keys onto different servers could be a simple hash modulo of the number of servers in the environment.
- In this setup, a pool of caching servers typically hosts many key/value pairs, providing fast access to data. If the query for a key for partition information is not available in the cache, the data is retrieved from the distributed hashing table as a cache miss.
- If a client needs to retrieve the partition with a key, it looks it up in the cache and gets the location of the partition. If the cache entry for the key does not

exist, then it computes the hash and modulo of the number of servers to determine which server the partition exists on and then retrieves the partition information, also populating it back into the cache for future use.

Distributed Hashing - Rehashing Problem

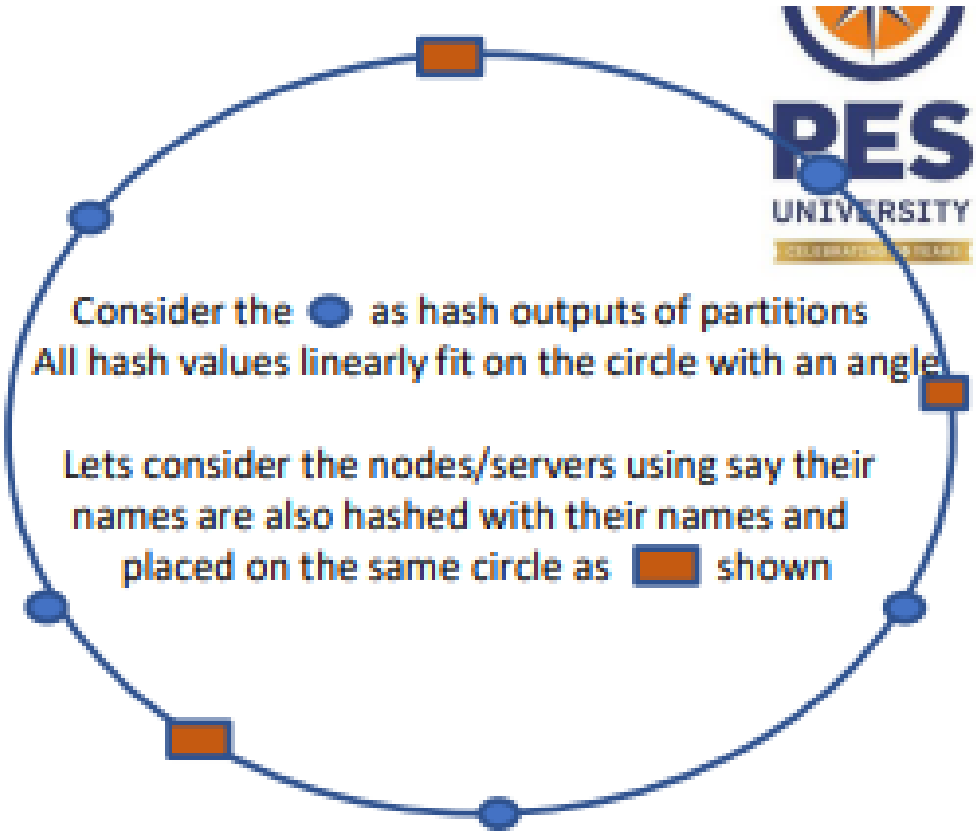
This Distributed hashing scheme is simple, intuitive, and works fine, but may have issues in certain scenarios:

1. **Scaling:** Adding a few more servers into the pool for scaling can be challenging.
2. **Server Failure:** If one of the servers fails, keys need to be redistributed to account for the missing server or to account for new servers in the pool.

While this is a common issue with any distribution scheme, the problem with our simple modulo distribution is that when the number of servers changes, most hashes modulo N will change. Therefore, most keys will need to be moved to a different server. Even if a single server is removed or added, all keys will likely need to be rehashed into a different server.

For example, let's say Server 2 (of 0, 1, and 2) failed. In this case, all key locations change, not only the ones from server C representing 2 in the HASH mod 3.

Consistent Hashing



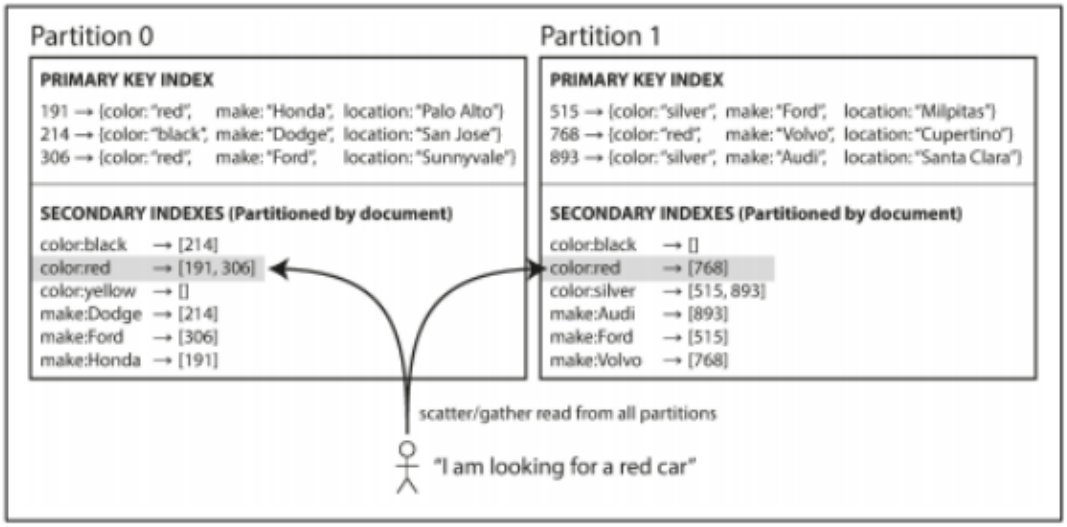
Addressing issues with traditional distributed hashing schemes requires a distribution scheme that does not directly depend on the number of servers. Consistent Hashing is one such solution.

How Consistent Hashing works:

- **Abstract Circle (Hash Ring):** In Consistent Hashing, servers and objects are assigned positions on an abstract circle, or hash ring.
- **Mapping Hash Outputs:** Hash outputs are linearly mapped onto the edge of this circle. The minimum hash value corresponds to an angle of zero, and the maximum hash value corresponds to an angle of 360 degrees.
- **Mapping Keys and Servers:** Keys and servers are hashed and placed on this circle. Each object key is associated with the server whose key is closest in a counterclockwise (or clockwise) direction.
- **Scaling:** When a new server is added, only the keys closest to it in the clockwise direction need to be moved, and the rest remain unchanged. Similarly, if a server fails, only the keys behind it need to be relocated.

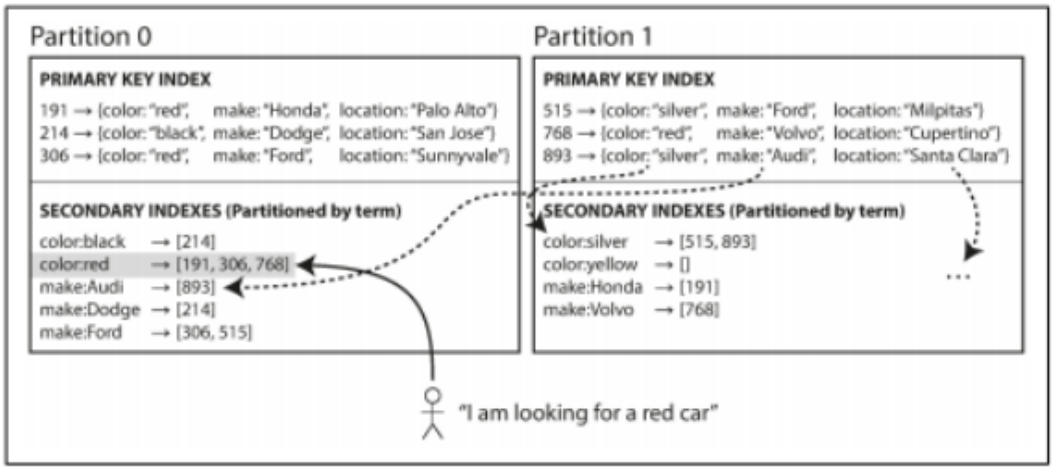
Partitioning and Secondary Indexes

- **Document-based Partitioning:**
  - In this approach, each partition maintains its own secondary indexes, covering only the documents in that partition.
  - It is also known as a local index because each partition is completely separate.
  - Reading from a document-partitioned index requires querying all partitions and combining the results, often known as scatter/gather. This can make read queries on secondary indexes expensive.



- **Term-based Partitioning:**
  - In term-based partitioning, a global index covering data in all partitions is constructed.
  - The index can be partitioned either by the term itself or using a hash of the term.
  - Partitioning by the term itself is useful for range scans, while partitioning on a hash of the term ensures a more even distribution of the load.
  - A global index makes reads more efficient, as the client only needs to make a request to the partition containing the term it wants.

- However, writes are slower and more complicated because a write to a single document may affect multiple partitions of the index.
- The global index needs to be updated immediately after a document is written to ensure consistency, which might require asynchronous updates as distributed transactions across all affected partitions are not supported in all databases.



Rebalancing Partitions and Request Routing

Skewed Workloads and Relieving Hot Spots

While hashing keys for partitioning helps reduce hot spots, it doesn't eliminate them entirely. In cases where all reads and writes are for the same key, all requests will be routed to the same partition, causing performance issues. To address this, simple techniques like adding a random number to the beginning or end of the key can be employed. However, this may lead to lower performance as data needs to be read from all keys and combined.

Rebalancing Partitions

Over time, various changes such as an increase in query throughput, dataset size, addition of new machines, or failure of existing ones may necessitate the movement of data and requests from one node to another. This process is known as rebalancing.

Rebalancing should meet the following requirements:

- After rebalancing, the load (data storage, read and write requests) should be fairly shared between the nodes.
- The database should continue to accept reads and writes during rebalancing.
- Only necessary data should be moved between nodes to minimize network and disk I/O load.

Strategies for Rebalancing Partitions

1. **Hash-based Rebalancing (hash mod N):**
    - In this approach, keys are partitioned based on the hash of the key, and the % operator is used to associate a partition to a node.
    - However, changing the number of nodes requires most keys to be moved from one node to another, making this approach excessively expensive.
2. **Fixed Number of Partitions:**
    - Create more partitions than nodes and assign several partitions to each node.
    - When a new node is added, it can take partitions from existing nodes until partitions are fairly distributed.
    - Only entire partitions are moved between nodes, and the assignment of keys to partitions remains unchanged.
3. **Dynamic Partitioning:**
    - Suitable for databases using key range partitioning.
    - Partitions are dynamically created or split when they exceed a configured size.
    - If a partition grows too large, it is split into two, and one half is transferred to another node to balance the load.
    - Conversely, if lots of data is deleted, a partition can be merged with an adjacent one.
4. **Partitioning Proportionally to Nodes:**

- The number of partitions is proportional to the number of nodes.
- When a new node joins the cluster, it randomly chooses a fixed number of existing partitions to split and takes ownership of one half of each split partition.
- With a fixed number of partitions, the size of each partition is proportional to the size of the dataset.

In both the above cases, the number of partitions is independent of the number of nodes. Partitioning proportionally to nodes ensures that as the dataset size grows or the number of nodes changes, the size of each partition remains fairly stable, maintaining balanced loads across the cluster.

Request Routing

Consider a partitioned dataset distributed across multiple nodes running on different machines. When a client wants to make a request, it needs to know which node to connect to. As partitions are rebalanced, the assignment of partitions to nodes changes, requiring a mechanism to stay updated on these changes.

Approaches to Service Discovery:

1. **Approach 1:**
    - Clients can contact any node.
    - If the contacted node owns the partition for the request, it handles it directly; otherwise, it forwards the request to the appropriate node and returns the reply to the client.
2. **Approach 2:**
    - All requests from clients are sent to a routing tier first.
    - The routing tier determines the node that should handle each request and forwards it accordingly.
    - The routing tier acts as a partition-aware load balancer but does not handle any requests itself.

3. **Approach 3:**
    - Clients are aware of the partitioning and the assignment of partitions to nodes.
    - Clients can connect directly to the appropriate node without any intermediary.

Approach 4: ZooKeeper

- Many distributed data systems use a separate coordination service like ZooKeeper to manage cluster metadata.
- Each node registers itself in ZooKeeper, which maintains the authoritative mapping of partitions to nodes.
- Other actors, such as the routing tier or partition-aware clients, can subscribe to this information in ZooKeeper.
- Whenever a partition changes ownership or a node is added or removed, ZooKeeper notifies the routing tier to keep its routing information up to date.

Systems Using ZooKeeper:

- HBase, SolrCloud, and Kafka use ZooKeeper to track partition assignment.
- LinkedIn's Espresso uses Helix for cluster management (which relies on ZooKeeper), implementing a routing tier.
- Cassandra and Riak use a gossip protocol among nodes to disseminate any changes in the cluster state. Requests can be sent to any node, which then forwards them to the appropriate node for the requested partition.

Replication

In cloud computing, replication involves keeping multiple copies of the same data on different machines connected via a network. Replication serves several purposes:

1. **Reducing Latency:** By keeping data geographically close to users.
2. **Increasing Availability:** Ensuring the system continues to work even if some parts fail.

3. **Scaling Read Throughput:** Allowing more machines to serve read queries.

Replication is especially beneficial for read-only data, improving performance and providing fault tolerance.

Key Concepts:

- **Replica:** Each node storing a copy of the dataset.
- **Write Operations:** Every write must be processed by every replica to ensure all nodes hold the same data.

Challenges:

- **Handling Data Changes:**
  - Should there be a leader replica, and if so, how many?
  - Synchronous or asynchronous propagation of updates among replicas?
  - Handling failed replicas and leader failures, including resurrections.

Replication Algorithms:

1. **Leader-Based (Single-Leader) Replication:**
2. **Synchronous Replication:**
3. **Asynchronous Replication:**
4. **Multi-Leader Replication:**
5. **Leaderless Replication:**

Replication strategies vary based on the specific requirements of the system in terms of consistency, availability, and performance.

1. Replication: Leader-Based Replication

Leader-based replication, also known as Leader-Follower or master-slave replication, ensures data consistency across multiple replicas by designating one replica as the leader.

Key Components:

- **Leader:**
  - Dedicated compute node responsible for propagating changes.

- Also known as master or primary.
- Accepts both read and write queries.
- Sends changes as replication logs to followers.

- **Follower:**
  - General replica.
  - Also known as slave, secondary, or hot standby.
  - Accepts only read queries and responds with data from local storage/copy.
  - Receives changes from the leader(s) and updates the local copy accordingly:
    - Applies all writes in the same order as applied on the leader.

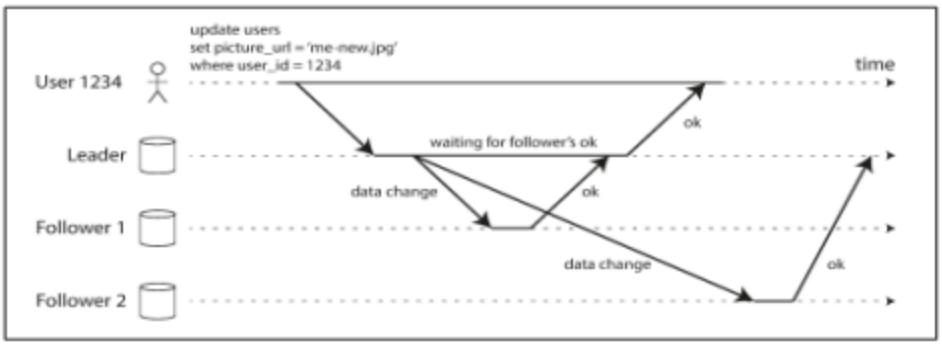
Replication Process:

- Users or clients send write requests to the leader.
- The leader writes new data to its local storage and sends the data change to all followers as part of a replication log.
- Each follower takes the log from the leader and updates its local copy of the database by applying all writes in the same order as they were processed on the leader.
- Clients can read from anywhere (leader or the followers), but writes are only accepted by the leader.

Synchronous vs. Asynchronous Replication:

- **Synchronous Replication:**
  - The leader waits until followers confirm that they received the write before reporting success to the user and making the write visible to other clients.
  - Example: Replication to follower 1 is synchronous.
- **Asynchronous Replication:**
  - The leader sends the message to its follower(s) but doesn't wait for a response from the followers before answering success to the user.

- Example: Replication to follower 2 is asynchronous.



Leader based replication with one synchronous & one asynchronous follower

Implementation of Replication Logs:

1. **Statement-based replication:**
   - The leader logs every write request it executes and sends that statement log to its followers.
2. **Write-ahead log (WAL) shipping:**
   - The leader writes the log, which is an append-only sequence of bytes containing all writes to the database, to disk and sends it across the network to its followers.
   - Followers process this log and update their local copy of the database, building an exact copy of the data structures found on the leader.
3. **Change data capture (CDC) based replication:**
   - Logical log replication captures changes made to database tables at the granularity of rows.
   - It involves identifying, capturing, and delivering changes made to the database.
   - Replicas can run on different versions or storage engines but use different log formats for different storage engines.
   - CDC-based replication is easier to parse for external applications.
   - A logical log for a relational database is usually a sequence of records describing writes to database tables at the granularity of a row.

4. **Trigger-based replication (application layer):**
    - Triggers allow users to register custom application code that is automatically executed when a data change (write transaction) occurs in a database system.
    - The trigger logs the change, which can be read by an external process to apply necessary application logic and replicate the data change to another system.

Potential Issues and Their Handling:

Follower Failure: Catch-up Recovery:

- Each follower keeps a log of the data changes it has received from the leader on its local disk.
- If a follower crashes and is restarted or if the network between the leader and the follower is temporarily interrupted, the follower can recover easily from its log.
- The follower knows the last transaction processed before the fault occurred and can connect to the leader to request all data changes that occurred during the disconnection.
- Once the follower has applied these changes, it has caught up to the leader and can continue receiving data changes as before.

Leader Failure - Failover:

- If the leader fails, one of the followers needs to be promoted as the new leader.
- Clients need to be reconfigured to send their writes to the new leader, and other followers need to start consuming data changes from the new leader.
- Failover can happen manually or automatically:
    - **Automatic Failover Process:**
        1. Determining that the leader has failed.
        2. Choosing a new leader.

3. Reconfiguring the system to use the new leader.

Replication Lag

In leader-based replication, all writes go to the leader, but read-only queries can go to any replica. This makes it attractive for scalability, latency, and fault-tolerance.

For read-mainly workloads:

- Distribute reads across multiple followers to remove load from the leader.
- Allows read requests to be served by nearby replicas.

However, this approach is realistic only for asynchronous replication; otherwise, the system will not be available.

Potential Issues and their Handling

Replication Lag:

- If an application reads from an asynchronous follower, it may see outdated information if the follower has fallen behind.
- This leads to apparent inconsistencies in the database. Running the same query on the leader and a follower at the same time may yield different results because not all writes would have been reflected in the follower.

Identification of Inconsistencies due to Replication Lag

Reading Your Own Writes:

- Reading your own writes ensures that users always see any updates they submitted themselves.
- Different consistency models like Read-after-write consistency and various eventual consistency models help bring in consistency.

Potential Solutions for Replication Lag:

1. **Always read critical data from the leader and the rest from a follower:**
    - Negates the benefit of read scaling.
2. **Monitor the replication lag on followers and prevent queries on any follower with significant lag behind the leader.**

3. **Timestamp-based solution:**
    - The client can remember the timestamp of its most recent write.
    - The system ensures that the replica serving any reads for that user reflects updates at least until that timestamp.
4. **Monotonic reads:**
    - Ensure that each user always makes their reads from the same replica.
5. **Consistent prefix reads:**
    - If a sequence of writes happens in a certain order, then anyone reading those writes should see them appear in the same order.

Multi-Leader Replication

Recap: Replication is a means of keeping a copy of the same data on multiple machines connected via a network. We discussed three popular algorithms for replicating changes between nodes: single-leader, multi-leader, and leaderless replication.

Leader Based Replication:

- Single bottleneck in the leader.
- All writes must go through it. If there is a network interruption between the user and the leader, no writes are allowed.

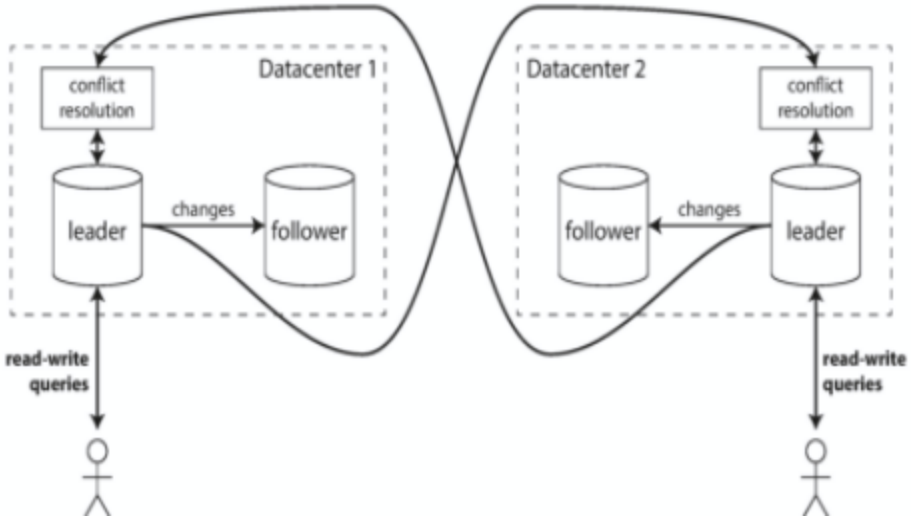
Multi-Leader Replication:

- Each node can accept writes, allowing more than one leader.
- Each node that processes a write must forward that data change to all the other nodes.
- Also known as master/master or active/active replication.
- Each leader simultaneously acts as a follower to the other leaders.

Use Cases:

- **Multi-Datacenter Operation:**
    - A leader in each datacenter.

- Each datacenter has regular leader-follower replication.
- Between datacenters, each datacenter's leader replicates its changes to the leaders in other datacenters.



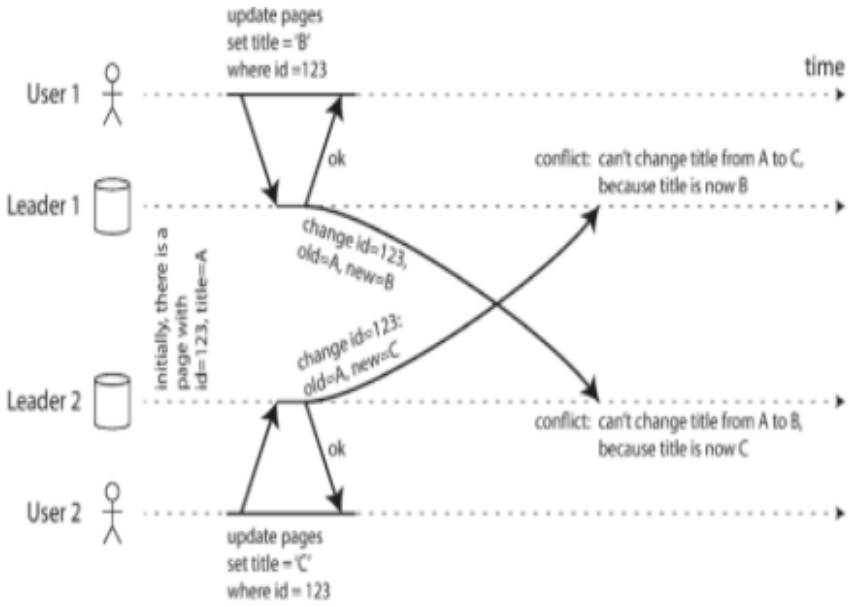
- **Clients with Offline Operation:**
  - Every device has a local database acting as a leader.
- **Collaborative Editing:**
  - Enables multiple users to collaborate on a document simultaneously.

Multi-Leader Replication Challenges

A problem with multi-leader replication is that it can lead to write conflicts:

1. **Conflict Avoidance:**
   - Ensure that all writes for a particular record go through the same leader.
2. **Converging Toward a Consistent State:**
   - Give each write a unique ID (e.g., a timestamp, a long random number, a UUID, or a hash of the key and value).
   - Pick the write with the highest ID as the winner and discard the other writes.
3. **Custom Conflict Resolution Logic:**

- Implement write conflict resolution logic in the application code.
- This code may be executed on write or on read.



Single-Leader vs. Multi-Leader Replication

Performance:

- **Single-Leader Configuration:**
  - Every write must go over the internet to the datacenter with the leader, adding significant latency.
- **Multi-Leader Configuration:**
  - Every write can be processed in the local datacenter and is asynchronously replicated to other datacenters.
  - Inter-datacenter network delay is hidden from users, improving perceived performance.

Tolerance of Datacenter Outages:

- **Single-Leader Configuration:**
  - Failover can promote a follower in another datacenter to be the leader if the datacenter with the leader fails.

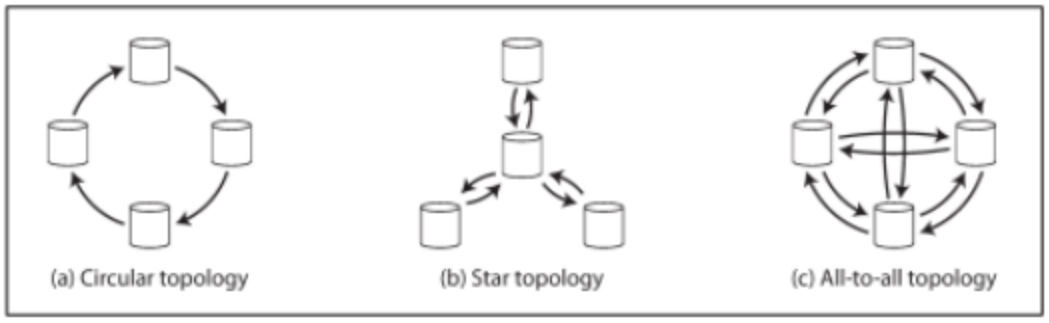
- **Multi-Leader Configuration:**
    - Each datacenter can continue operating independently, and replication catches up when the failed datacenter comes back online.

Tolerance of Network Problems:

- **Single-Leader Configuration:**
    - Writes are made synchronously over the inter-datacenter link, making it sensitive to network problems.
- **Multi-Leader Configuration:**
    - Asynchronous replication can usually tolerate network problems better as temporary network interruptions do not prevent writes from being processed.

Multi-Leader Replication Topologies:

- **Circular Topology:**
    - Each node receives writes from one node and forwards those writes (plus any writes of its own) to one other node.
- **Star Topology:**
    - One designated root node forwards writes to all other nodes.
- **All-to-All Topology:**
    - Allows messages to travel along different paths, avoiding a single point of failure.



Leaderless Replication

Replication (Recap)

Introduction:

- Replication is a method of maintaining a copy of the same data on multiple machines connected via a network.
- Reasons for replication:
  1. Reduce latency by keeping data geographically close to users.
  2. Increase availability by allowing the system to continue working even if some parts fail.
  3. Scale out the number of machines that can serve read queries, increasing read throughput.

Popular Replication Algorithms:

- Leader-based replication:
  - Synchronous replication
  - Asynchronous replication
  - Multi-leader replication
  - Leaderless replication

Leader Based Replication (Recap):

- In leader-based replication, one replica is designated as the leader.
- Users/clients send write requests to the leader, which writes the new data to its local storage and then sends the data change to all other replicas (followers).
- Potential issues: leader failure, follower failure, replication lag.
- Techniques to address issues: failover, catch-up recovery, conflict resolution.
- Single leader creates a bottleneck.

Single Leader vs. Multi-Leader Replication:

- **Performance:**

    - Single-Leader Configuration:
        - Writes must go over the internet to the datacenter with the leader, adding latency.
    - Multi-Leader Configuration:
        - Writes can be processed locally and asynchronously replicated to other datacenters, improving perceived performance.
- **Tolerance of Datacenter Failures:**
    - Single-Leader Configuration:
        - Failover can promote a follower in another datacenter to be the leader.
    - Multi-Leader Configuration:
        - Each datacenter can operate independently, and replication catches up when the failed datacenter comes back online.
- **Tolerance of Network Problems:**
    - Single-Leader Configuration:
        - Sensitive to problems in inter-datacenter link due to synchronous writes.
    - Multi-Leader Configuration:
        - Asynchronous replication tolerates network problems better.

Introduction:

- Leaderless replication is an alternative to single leader and multi-leader replication strategies.
- In a leaderless replication setup, nodes are considered peers, and all nodes accept both writes and reads from clients.
- Leaderless replication offers better availability compared to leader-based approaches.

Architecture:

- Every write must be sent to every replica.

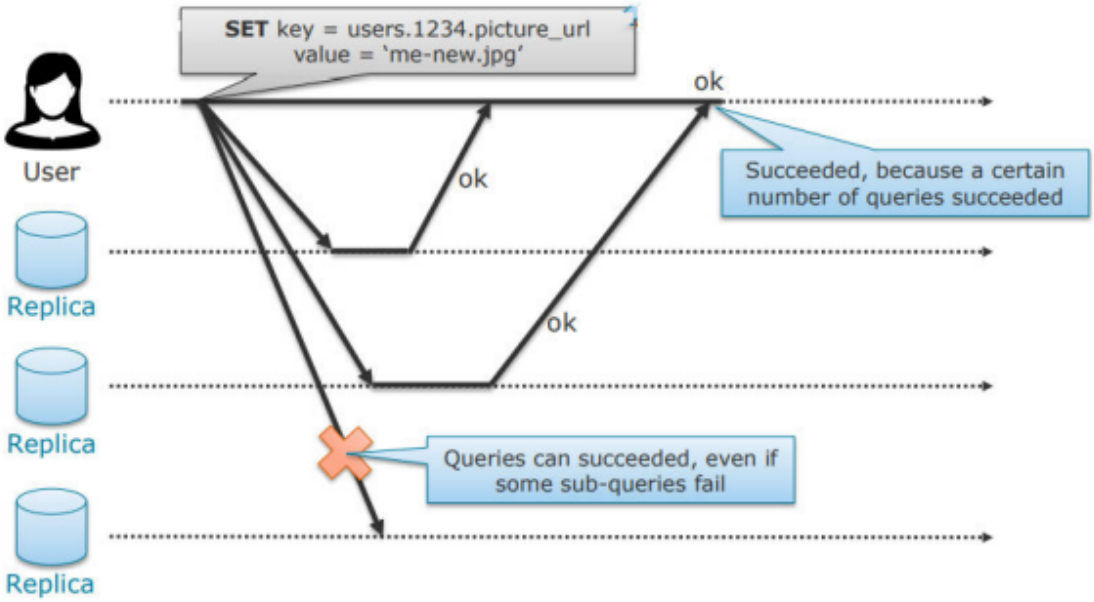
- A write is considered successful when acknowledged by (a quorum of) at least *k* out of *n* replicas.
- Read operations also require agreement among a quorum of replicas.

Advantages:

- Parallel writes are possible, avoiding the bottleneck of a single leader.

Writes with Leaderless Replication:

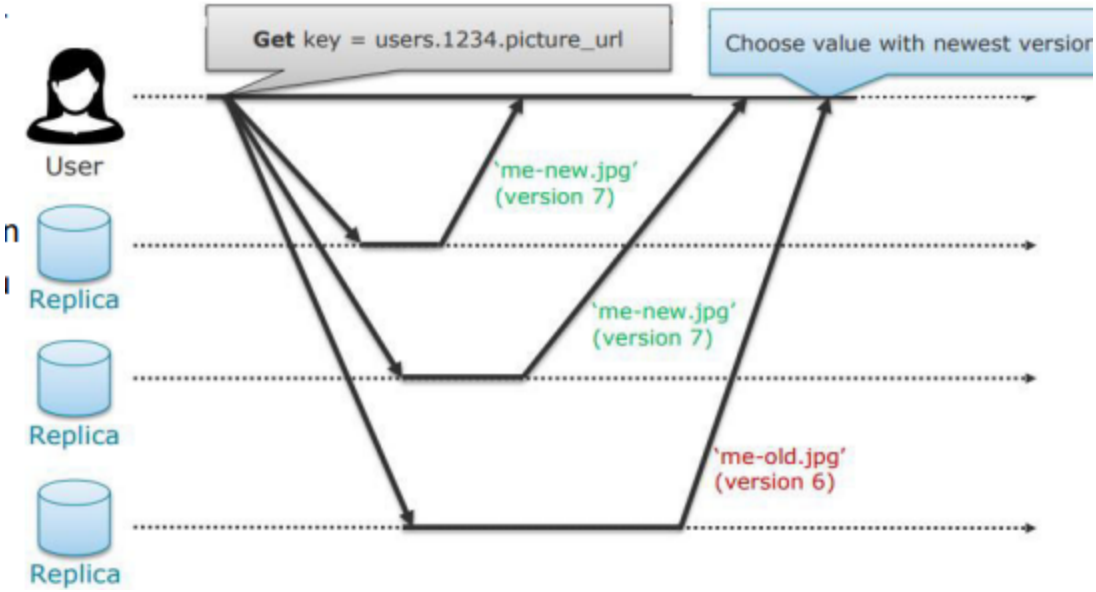
- Upon a write, the client broadcasts the request to all replicas and waits for a certain number of ACKs.
- The write is considered successful when acknowledged by at least *k* out of *n* replicas.



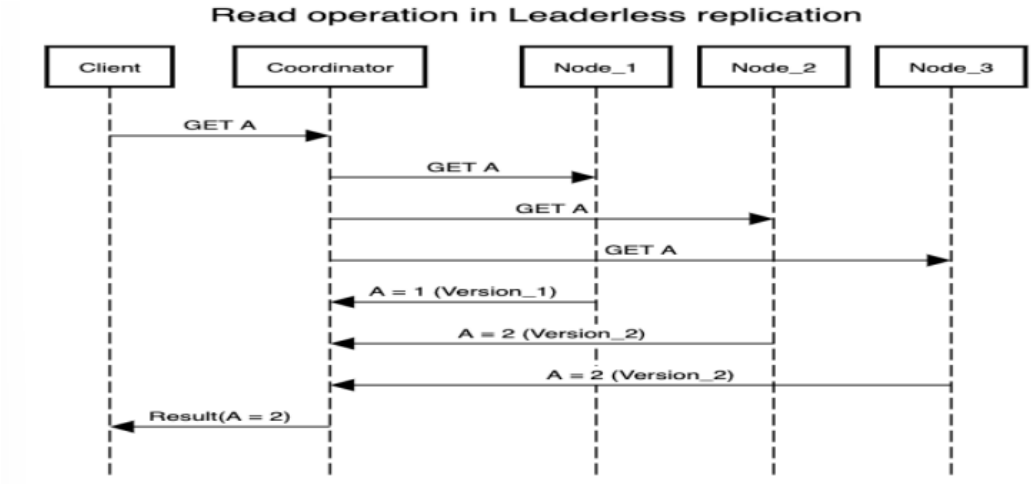
Reads with Leaderless Replication:

- Upon a read, the client contacts all replicas and waits for responses.
- At least *k* out of *n* reads must agree on a value for the read to be considered successful.
- The approach of waiting for a quorum of responses is known as quorum. The value *k* is known as the quorum.

- *k* value when reading is known as read quorum, and *k* value when writing is known as write quorum.



Consistency:



- Quorum settings can be configured to provide consistency to the copies.

Leaderless Replication: Potential Pitfalls

1. Writing to the Database When a Node Is Down:

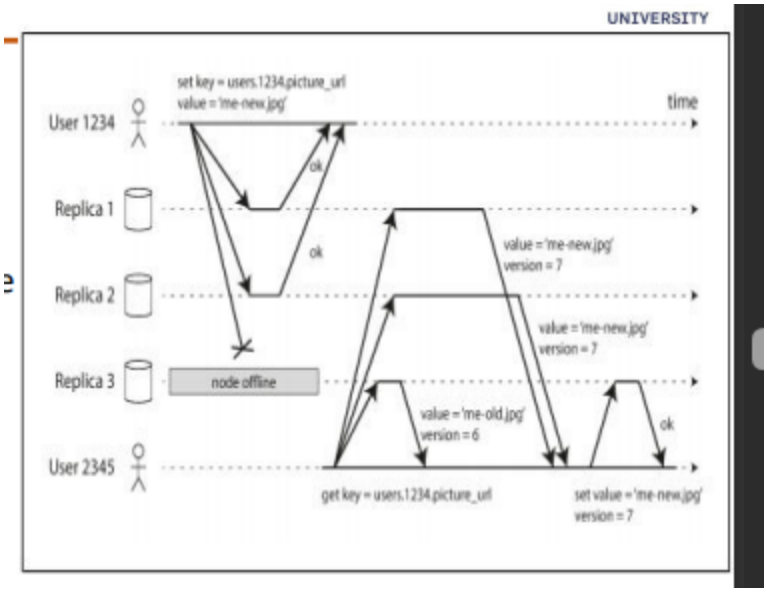
- Scenario: Three replicas, one currently unavailable.

- Client sends write to all three replicas in parallel, and two available replicas accept it, but the unavailable one misses it.
- The write is considered successful since two out of three replicas acknowledged it.
- When the unavailable node comes back online, clients reading from it may get stale values.

Approach to Address:

- Read requests also sent to several nodes in parallel.
- Client may get different responses from different nodes, with version numbers used to determine which value is newer.

2. Catching Up on Missed Writes:



- **Read Repair:**
  - Client reads from several nodes in parallel and detects stale responses. It writes the newer value back to the replica with the stale value.
- **Anti-entropy Process:**
  - Background process looks for differences in data between replicas and copies any missing data from one replica to another.

3. Detecting Concurrent Writes:

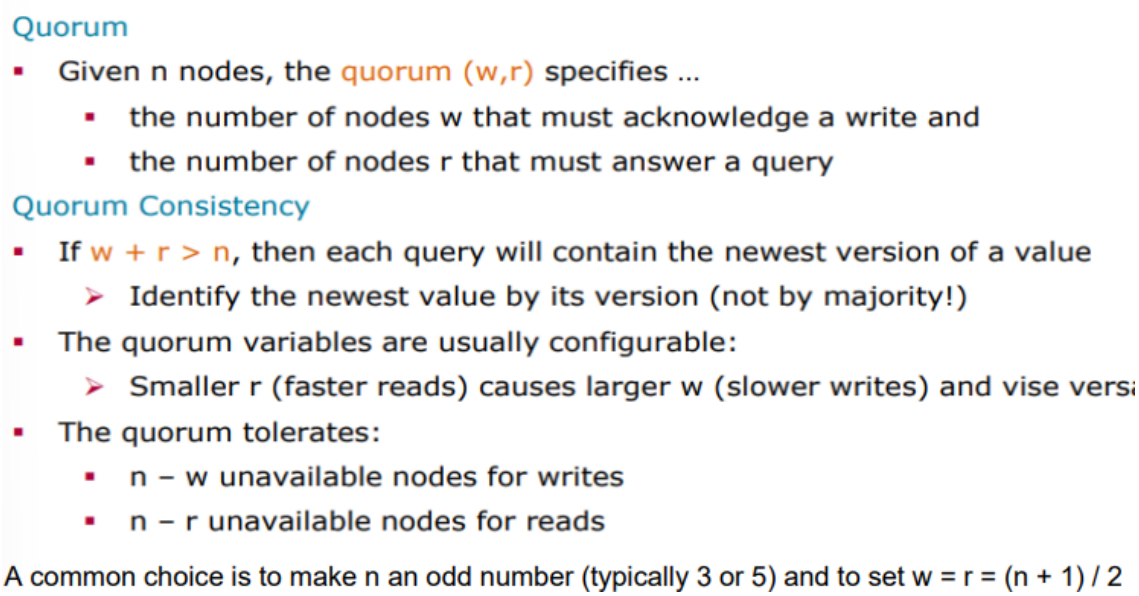
- Concurrent writes from several clients to the same key can lead to conflicts.
- Events may arrive in different orders at different nodes due to network delays and partial failures.
- Last write wins approach: Older values overwritten and discarded.

Approaches to Address:

- **Timestamps:** Used to resolve write conflicts, but may face synchronization challenges.
- **Determining Concurrency:**
    - Operations are concurrent if B knows about A, depends on A, or builds upon A in some way.
    - Server determines concurrency by looking at version numbers.
    - Version numbers per replica per key stored in a version vector.

Algorithm for Determining Concurrency:

- Server maintains a version number for every key, increments it every time the key is written.
- When a client reads a key, the server returns all values not overwritten and the latest version number.
- Client must read a key before writing.
- When a client writes a key, it includes the version number from the prior read and merges together all values received.
- Server overwrites all values with version number or below and keeps all values with a higher version number.



A common choice is to make n an odd number (typically 3 or 5) and to set w = r = (n + 1) / 2

Leaderless Replication: Monitoring Staleness

- **Leader-based Replication:**
  - Database typically exposes metrics for replication lag.
  - Replication lag is measured by subtracting a follower's current position from the leader's current position.
- **Leaderless Replication:**
  - No fixed order in which writes are applied, making monitoring more difficult.

Leaderless Replication: Multi-datacenter Operation

- Suitable for multi-datacenter operation:
  - Tolerates conflicting concurrent writes, network interruptions, and latency spikes.
- Number of replicas (n) includes nodes in all datacenters.
- Number of replicas configured in each datacenter.
- Each write sent to all replicas, regardless of datacenter.
- Client waits for acknowledgment from a quorum of nodes within its local datacenter to remain unaffected by delays and interruptions on the cross-

datacenter link.

- Higher-latency writes to other datacenters often configured to happen asynchronously.

Consistency Models

Consistency and Eventual Consistency

- **Consistency:**
  - Refers to the consistency of values in different copies of the same data item in a replicated distributed system.
  - Can be lost due to network issues or differences in the time taken to write into different copies of the same data.
  - Consistency model is a contract between a distributed data store and processes, ensuring certain rules are followed.
  - Some applications require strong consistency, where values across copies must be the same, often impacting performance.
- **Consistency Models - Consistency Guarantees:**
  - **Eventual consistency:**
    - Guarantees that copies of data will eventually be consistent once all current operations have been processed, but they don't always have to be identical.
    - Provides improved performance.
    - Most replicated databases provide eventual consistency, meaning that if you stop writing to the database and wait for some unspecified length of time, eventually all read requests will return the same value.
    - Doesn't specify when replicas will converge.
    - Edge cases of eventual consistency become apparent during system faults or high concurrency.

Consistency Models – Different Types

Sequential Consistency (Lamport):

- In a shared-memory system, supports the sequential consistency model if all processes see the same order of all memory access operations on the shared memory.
- The exact order of memory access operations doesn't matter.
- Conceptually, there is one global memory and a switch that connects an arbitrary processor to the memory at any time. Each processor issues memory operations in program order, and the switch provides global serialization among all processors.
- Not deterministic because multiple executions of the distributed program might lead to a different order of operations.

Causal Consistency:

- Two events are causally related if one can influence the other.
- Relaxes the requirement of the sequential model for better concurrency.
- In the causal consistency model, all processes see only those memory reference operations in the same correct order that are potentially causally related.
- Memory reference operations that are not potentially causally related may be seen by different processes in different orders.

PRAM (Pipelined Random-Access Memory) Consistency:

- Ensures that all write operations performed by a single process are seen by all other processes in the order in which they were performed, as if all the write operations performed by a single process are in a pipeline.
- Write operations performed by different processes may be seen by different processes in different orders.

Strict Consistency (Strong consistency or Linearizability):

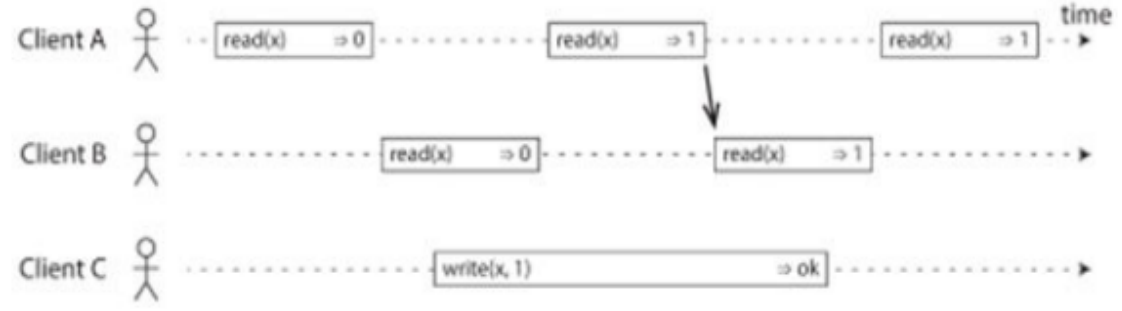
- A shared-memory system supports strict consistency if the value returned by a read operation on a memory address is always the same as the value written by the most recent write operation to that address, irrespective of the locations of the processes performing the read and write operations.

- Similar to sequential consistency, but the execution order of programs between processors must be the order in which those operations were issued.
- If each program in each processor is deterministic, then the distributed program is deterministic.

Linearizability

- The basic idea is to make a system appear as if there were only one copy of the data, and all operations on it are atomic.
- Also known as atomic consistency, strong consistency, immediate consistency, or external consistency.
- Linearizability is a recency guarantee: a read is guaranteed to see the latest value written.
- In a linearizable system, as soon as one client successfully completes a write, all clients reading from the database must be able to see the value just written.
- Maintaining the illusion of a single copy of the data means guaranteeing that the value read is the most recent, up-to-date value and doesn't come from a stale cache or replica.

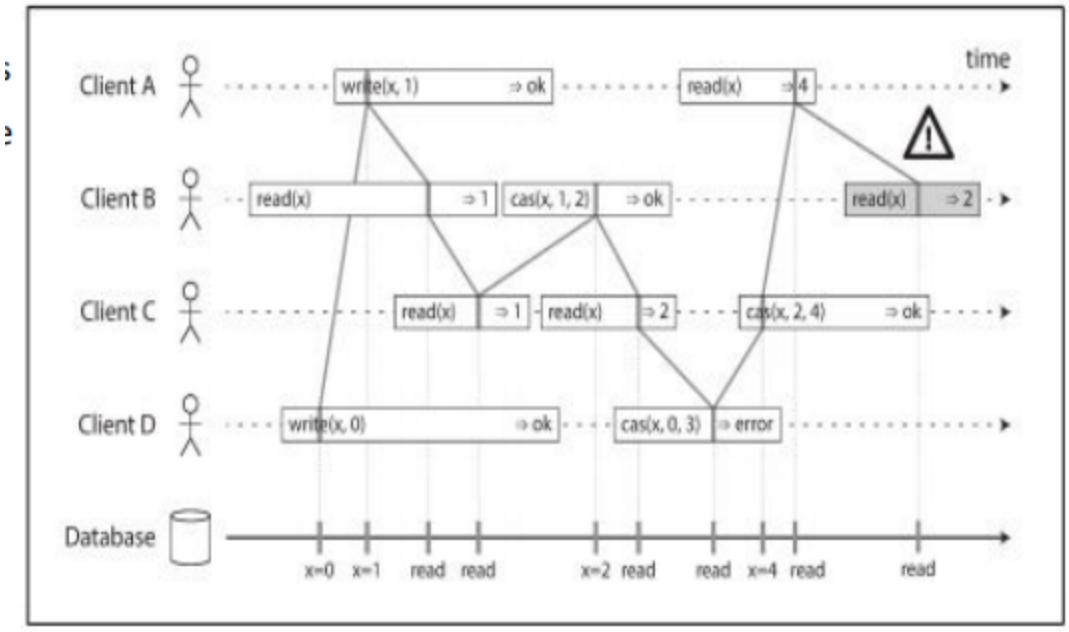
ed the new value, all following reads (on the same or other



Compare and Set (cas)

- Adds a third type of operation besides read and write.
- `cas(x, vold, vnew) ⇒ r` means the client requested an atomic compare-and-set operation. If the current value of the register `x` equals `vold`, it should be atomically set to `vnew`. If `x ≠ vold`, then the operation should leave the register unchanged and return an error. `r` is the database's response (ok or error).

Visualizing the points in time at which the reads and writes appear to have taken effect



- The final read by B is not linearizable.
- It is possible to test whether a system's behavior is linearizable by recording the timings of all requests and responses and checking whether they can be arranged into a valid sequential order.

Single-leader replication (potentially linearizable)

- If you make reads from the leader or from synchronously updated followers, they have the potential to be linearizable.

Consensus algorithms (linearizable)

- Consensus protocols contain measures to prevent split brain and stale replicas.
- Consensus algorithms can implement linearizable storage safely.
- Example: ZooKeeper.

Multi-leader replication (not linearizable)

Leaderless replication (probably not linearizable)

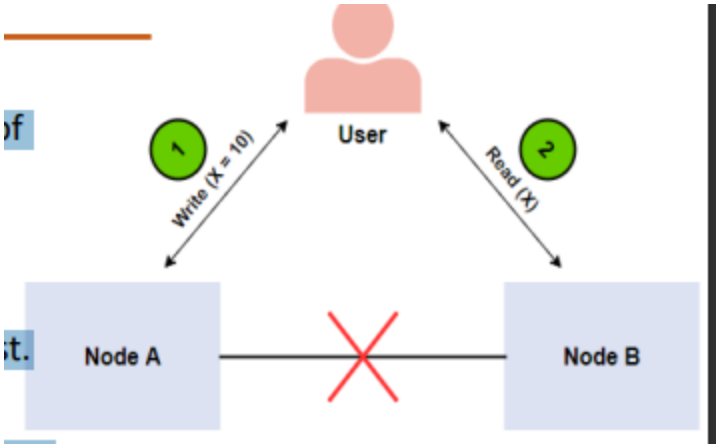
Consistency Model	Description	Use Cases	Advantages	Disadvantages
Sequential Consistency (Lamport)	All processes see the same order of all memory access operations on the shared memory.	Systems where the order of operations is crucial	Easy to understand and reason about	May limit concurrency and performance
Causal Consistency	All processes see only those memory reference operations in the same (correct) order that are potentially causally related.	Systems where causality is important	Better concurrency than sequential consistency	Requires understanding of causality among operations
PRAM (Pipelined Random-Access Memory) Consistency	Write operations performed by a single process are seen by all other processes in the order they were performed, while write operations performed by different processes may be seen in different orders.	Systems where writes by a single process need to be seen in order	Improves concurrency	Limited to writes performed by a single process
Strict Consistency	The value returned by a read	Systems where strong	Ensures the latest value is	May impact performance

(Strong consistency, Linearizability)	operation on a memory address is always the same as the value written by the most recent write operation to that address, regardless of the locations of the processes performing the read and write operations.	consistency is crucial	always returned	and scalability
Eventual Consistency	Systems are designed to eventually guarantee that all copies of data are consistent once all current operations have been processed, but they don’t always have to be identical.	Systems where low latency and high availability are important	Improved performance and availability	May lead to temporary inconsistencies during updates
Linearizability	Provides the illusion of a single copy of the data, ensuring that all operations on it are atomic and that reads see the latest value written.	Systems where strong consistency and atomicity are required	Ensures that reads return the most recent value	May impact performance and scalability due to synchronization requirements

CAP Theorem

CAP Theorem

- The CAP theorem, originally introduced as the CAP principle, can be used to explain some of the competing requirements in a distributed system with replication.
- It is a tool used to make system designers aware of the trade-offs while designing “network shared-data systems” or a distributed system that stores data on more than one node (physical/virtual machines) at the same time.
- The three letters in CAP refer to three desirable properties of distributed systems with replicated data:
  - **C** - Consistency (among replicated copies)
  - **A** - Availability (of the system for read and write operations)
  - **P** - Partition tolerance (in the face of the nodes in the system being partitioned by a network fault).



- The CAP theorem states that it is not possible to guarantee all three of the desirable properties – consistency, availability, and partition tolerance at the same time in a distributed system with data replication. We can strongly support only two of the three properties.

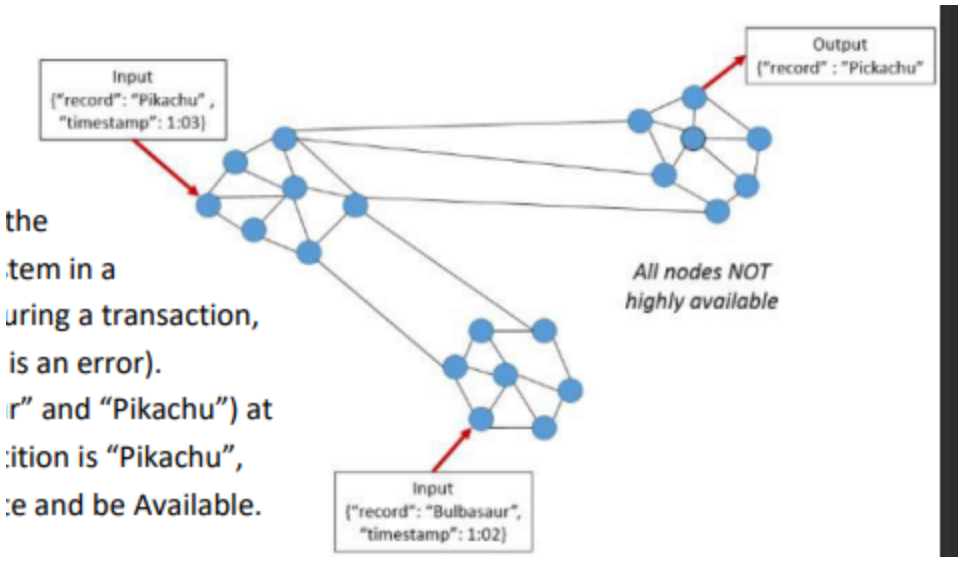
CAP Theorem Example

- Imagine a distributed system consisting of two nodes:
  - The distributed system acts as a plain register with the value of variable X.
  - There’s a network failure that results in a network partition between the two nodes in the system.

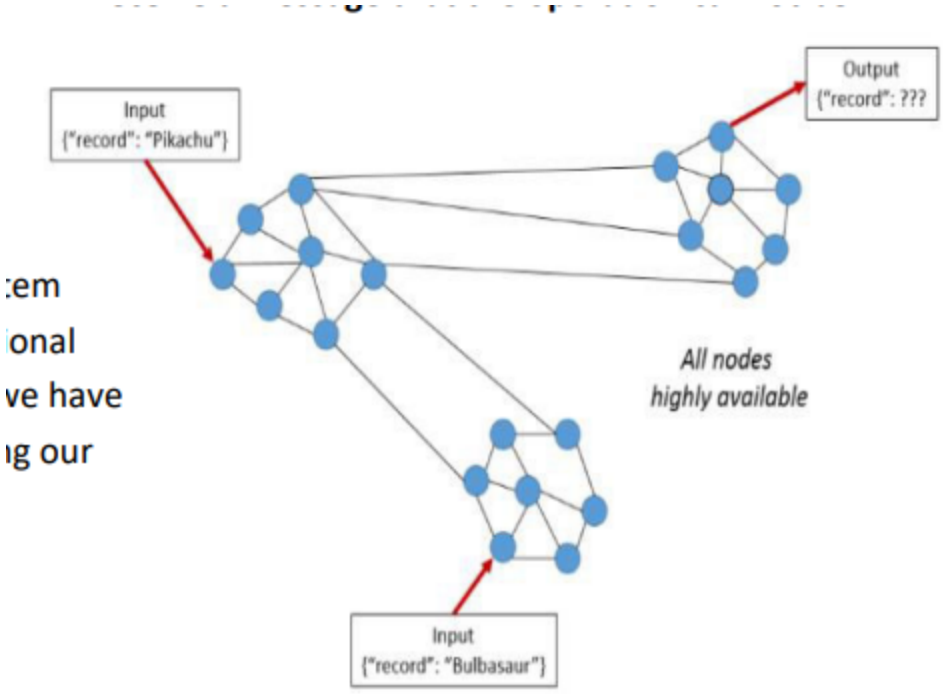
    - An end-user performs a write request, and then a read request.
- Let’s examine a case where a different node of the system processes each request. In this case, our system has two options:
    - It can fail at one of the requests, breaking the system’s availability.
    - It can execute both requests, returning a stale value from the read request and breaking the system’s consistency.
- The system can’t process both requests successfully while also ensuring that the read returns the latest value written by the write.
- This is because the results of the write operation can’t be propagated from node A to node B because of the network partition.

CAP Theorem

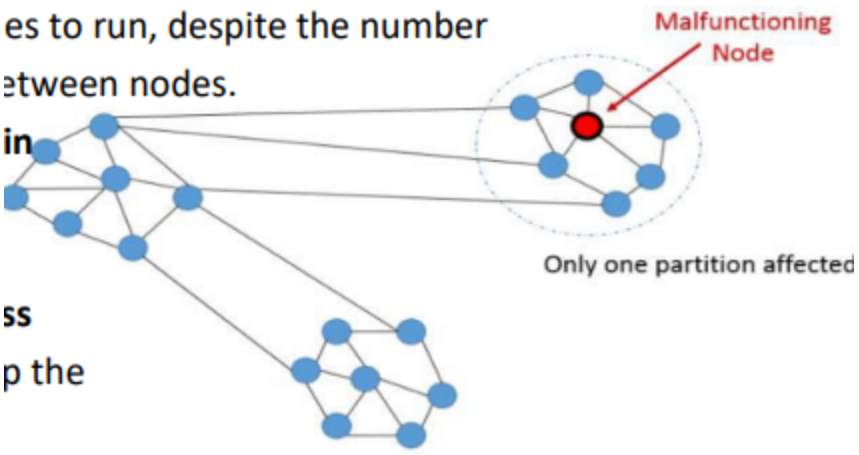
1. **Consistency (C)**
    - Consistency means that the nodes will have the same copies of a replicated data item visible for various transactions. It’s the guarantee that every node in a distributed cluster returns the same, most recently updated value of a successful write at any logical time.
    - In CAP, the term consistency refers to the consistency of the values in different copies of the same data item in a replicated distributed system. This can be verified if all reads initiated after a successful write return the same and latest value at any given logical time.
    - Performing a read operation will return the value of the most recent write operation causing all nodes to return the same data.
    - A system has consistency if a transaction starts with the system in a consistent state, and ends with the system in a consistent state (may have an inconsistent state during a transaction, but the entire transaction gets rolled back if there is an error).
    - In the image, we have 2 different records (“Bulbasaur” and “Pikachu”) at different timestamps. The output on the third partition is “Pikachu”, the latest input although it will need time to update and be Available.



1. **Availability (A)**
    - Availability means that each read or write request for a data item from a client to a node will either be processed successfully or will receive a message that the operation cannot be completed (if not in failed state).
    - All working nodes in the distributed system return a valid response for any request, without exception.
    - Achieving availability in a distributed system requires that the system remains operational 100% of the time, which may need that we have “x” servers beyond the “n” servers serving our application.



2. **Partition Tolerance (P)**
    - Partition tolerance requires that a system be able to re-route a communication when there are temporary breaks or failures in the network (network partitions).
    - It means that the system continues to function and upholds its consistency guarantees in spite of network partitions.
    - Distributed systems guaranteeing partition tolerance can gracefully recover from partitions once the partition heals.
    - This condition states that the system continues to run, despite the number of messages being delayed by the network between nodes.
    - A system that is partition-tolerant can sustain any amount of network failure that doesn't result in a failure of the entire network.
    - Data records are sufficiently replicated across combinations of nodes and networks to keep the system up through intermittent outages.



Tradeoffs between Requirements (Illustrations)

1. **Availability and Partition-Tolerance (Compromised Consistency):**
   - **Scenario:** Two nodes in the system with a severed link between them.
   - **Design Decision:** Design the system to accept requests on each node independently.
   - **Result:** Each node will issue its own results, compromising consistency for high availability and partition tolerance.
2. **Consistent and Partition-Tolerant (Compromised Availability):**
   - **Scenario:** Three nodes in the system, one node loses its link with the other two.
   - **Design Decision:** Create a rule that a result will be returned only when a majority of nodes agree.
   - **Result:** The system will return a consistent result, but since the separated node won't be able to reach consensus, it won't be available.
3. **Consistent and Available (Compromised on Partition-Tolerance):**
   - **Scenario:** System compromises on partition tolerance to maintain consistency and availability.
   - **Result:** The system may have to block on a partition to ensure consistency and availability.

CAP Theorem and Some of the Criticisms

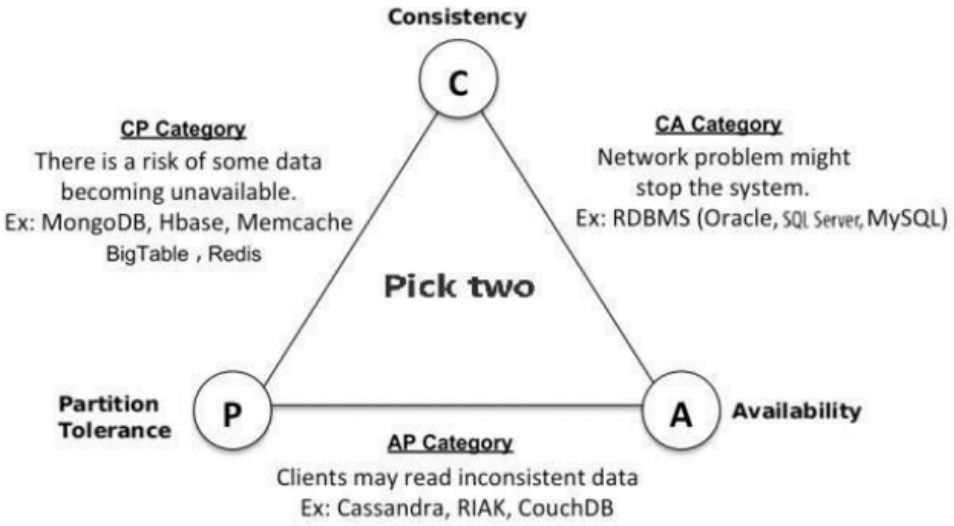
1. **Ignoring Latency:**
    - The CAP Theorem doesn't specify an upper bound on response time for availability. In practice, however, there exists a timeout.
    - CAP Theorem ignores latency, which is an important consideration. Timeouts are often implemented in services. During a partition, if a request is canceled, consistency is maintained, but availability is forfeited. Latency can be seen as another word for availability.
2. **Belief in Eventual Consistency:**
    - In NoSQL distributed databases, CAP Theorem has led to the belief that eventual consistency provides better availability than strong consistency. This belief may be considered outdated. It's better to factor in sensitivity to network delays.
3. **Binary Decision vs. Continuum:**
    - CAP Theorem suggests a binary decision, but in reality, it's more of a continuum. There are different degrees of consistency implemented via "read your writes, monotonic reads, and causal consistency."

CAP Theorem and Decision Making

- CAP Theorem's trade-offs lead to important questions with real-world consequences from both technical and business perspectives:
    - Is it important to avoid throwing up errors to the client?
    - Are we willing to sacrifice the visible user experience to ensure consistency?
    - Is consistency an essential part of the user’s experience?
    - Can we achieve what we want with a relational database and avoid the need for partition tolerance altogether?
- Understanding the overall goals of the project and the context in which your database solution is operating is crucial.

CAP Theorem and Its Relationship to Cloud Computing

- **Microservices Architecture:**
  - Microservices architecture is prevalent on both cloud servers and on-premises data centers.
  - Microservices are loosely coupled, independently deployable application components that incorporate their own stack, including their own database and database model, and communicate with each other over a network.
  - Understanding the CAP theorem can help in choosing the best database when designing a microservices-based application running from multiple locations.
  - For example, if the ability to quickly iterate the data model and scale horizontally is essential to your application, but you can tolerate eventual (as opposed to strict) consistency, databases like Cassandra or Apache CouchDB (supporting AP - Availability and Partitioning) can meet your requirements and simplify your deployment.
  - On the other hand, if your application depends heavily on data consistency, such as in an eCommerce application or a payment service, you might opt for a relational database like PostgreSQL.



Distributed Transactions

Transactions in Cloud Computing

1. Definition of a Transaction:

- A transaction is an operation composed of a number of discrete steps.
- All steps must be completed for the transaction to be committed. If not, the transaction is aborted, and the system reverts to its state before the transaction started.
- Example: Buying a house.

2. Basic Operations of Transactions:

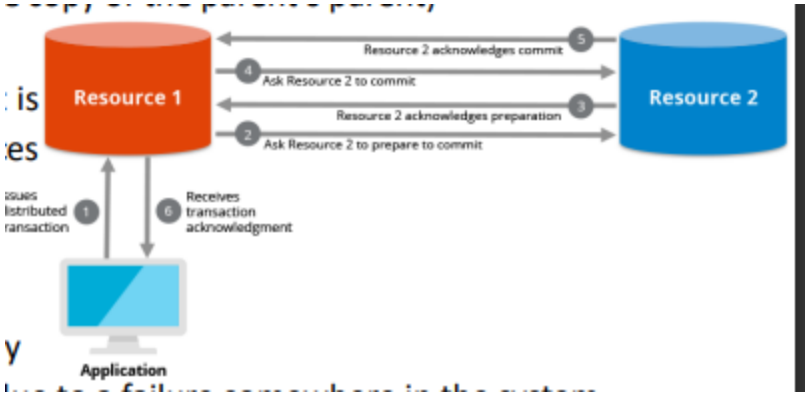
- **Transaction Primitives:**
  - **Begin Transaction:** Marks the start of a transaction.
  - **End Transaction:** Marks the end of a transaction and tries to commit it.
  - **Abort Transaction:** Terminates the transaction and restores old values.
  - **Read/Write Data:** Involves reading from or writing to files (or object stores). Data needs to be restored if the transaction is aborted.

3. Properties of Transactions (ACID):

- **Atomic:**
  - The transaction occurs as a single indivisible action. Intermediate results are not visible to others. It's an all-or-nothing operation.
- **Consistent:**
  - If the system has invariants, they must hold after the transaction. For example, the total amount of money in all accounts must remain the same before and after a "transfer funds" transaction.
- **Isolated (Serializable):**
  - If transactions run simultaneously, the final result must be the same as if they were executed in some serial order.
- **Durable:**

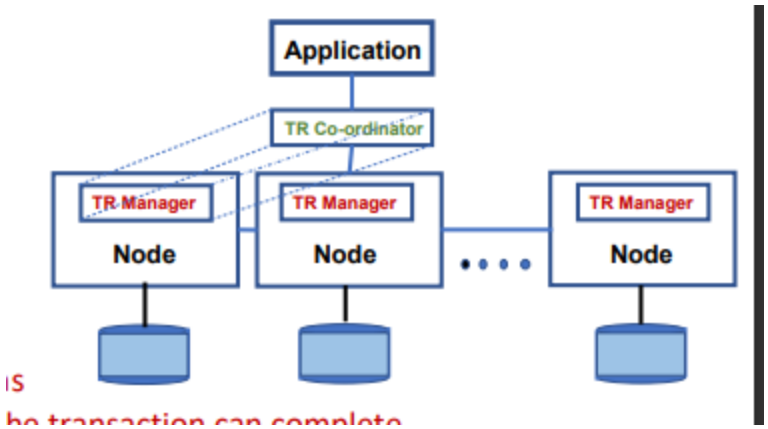
- Once a transaction commits, the results are made permanent. No failures after a commit will cause the results to revert.

Nested Transactions



- **Definition:**
  - A nested transaction is a top-level transaction that may create sub-transactions.
- **Problem:**
  - Sub-transactions may commit (making their results durable), but the parent transaction may abort.
- **Solution: Private Workspace**
  - Each sub-transaction is given a private copy of every object it manipulates.
  - On commit, the private copy displaces the parent's copy (which may also be a private copy of the parent's parent).

Distributed Transactions



- **Definition:**
  - A distributed transaction is a set of operations on data performed across two or more data repositories or resources across different systems.
- **Challenges:**
  - Handling machine, software, and network failures while preserving transaction integrity.
- **Possible Outcomes:**
  - All operations successfully complete.
  - None of the operations are performed at all due to a failure somewhere in the system.
    - In this outcome, any work completed prior to the failure will be reversed to ensure no net work was done, preserving data integrity according to the "ACID" principles.

Transactions and Different Types of Transactions

- **System Architecture:**
  - A system supporting distributed transactions has multiple data repositories hosted on different nodes connected by a network.
  - Transactions may access data at several nodes/sites.
- **Local Transaction Manager:**
  - Responsible for:

        - Maintaining a log for recovery purposes.
        - Coordinating the concurrent execution of transactions executing at that site.
        - Handling sub-transactions on that system.
        - Performing prepare, commit, and abort calls for sub-transactions.
    - Each sub-transaction must agree to commit changes before the transaction can complete.
- **Transaction Coordinator:**
    - Coordinates activities across the data repositories.
    - Periodically nominates a local transaction manager as a local coordinator.
    - Responsible for:
        - Starting the execution of transactions originating at the site.
        - Distributing sub-transactions to appropriate sites for execution.
        - Coordinating the termination of each transaction originating at the site, which may result in the transaction being committed at all sites or aborted at all sites.

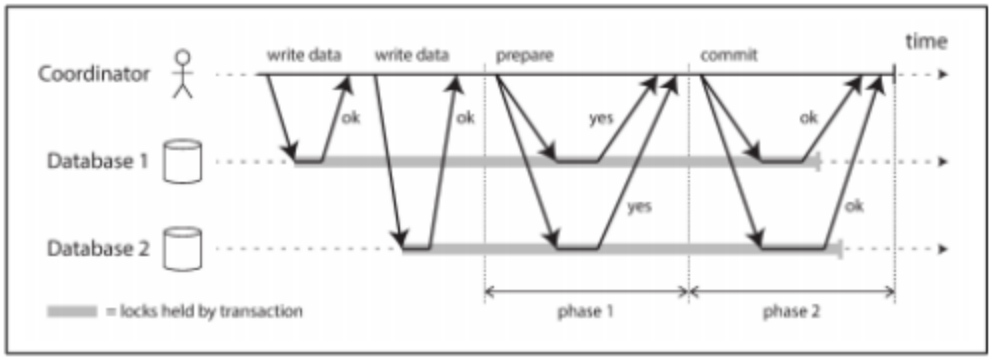
Distributed Transaction System Architecture

Concurrency Control

- All concurrency mechanisms must:
    - Preserve data consistency.
    - Complete each atomic action in finite time.
- Important capabilities include:
    - Resilience to site and communication link failures.
    - Allowing parallelism to enhance performance requirements.
    - Incurring optimal cost and optimizing communication delays.

- Placing constraints on atomic action.

Commit Protocols



- **Definition:**
  - Commit protocols ensure atomicity across sites in distributed transactions.
  - A transaction executed at multiple sites must either be committed at all sites or aborted at all sites.
  - It's not acceptable to have a transaction committed at one site and aborted at another.

Two-Phase Commit Protocol (2PC)

Phase 1

1. **Coordinator Actions:**
   - Places a log record `prepare T` on the log at its site.
   - Sends the message `prepare T` to each component's site.
2. **Component's Site Actions:**
   - Upon receiving the `prepare T` message:
     - If ready to commit:
       - Enters the pre-committed state for T and places `Ready T` in the local log.
       - Sends `Ready T` message to the coordinator.

- If not ready to commit:
  - Logs the record `Don't Commit T`.
  - Sends the message `Don't commit T` to the coordinator.

Phase 2

1. **Coordinator Actions:**
   - If it has received `Ready T` from all components of T:
     - Decides to commit T.
     - Logs `Commit T` at its site and sends `commit T` message to all sites involved in T.
   - If it has received `Don't commit T` from one or more sites:
     - Logs `Abort T` at its site and sends `abort T` messages to all sites involved in T.
2. **Component's Site Actions:**
   - Upon receiving the message:
     - If `Commit T` is received:
       - Commits the component of T at that site, releases the locks, and logs `Commit T`.
     - If `Abort T` is received:
       - Aborts T, rolls back all changes, releases locks, and logs `Abort T`.

Two-Phase Commit Protocol (2PC): Phase 1

- **Coordinator Related Nodes (Sites):**
  - Write `prepare to commit T` to log at its site.
  - Send `prepare to commit` message.
  - Work on the components towards T.
  - If ready to commit, get into pre-committed state for T, place `Ready T` in the local log, and send `Ready T` message to the coordinator (holds locks).

- If not ready to commit, place `Don't commit T` in the local log and send `Don't commit T` message to the coordinator.
- Wait for a message from the coordinator.
- Wait for a reply from each related node.

Two-Phase Commit Protocol (2PC): Phase 2

- **Coordinator Related Nodes (Sites):**
  - If `Ready T` has been received from all nodes:
    - Write `commit T` to the local log and send `commit T` message.
  - If `Don't commit T` is received from any of the related nodes:
    - Write `Abort T` to the local log and send `Abort T` to all related nodes (sites).
  - Wait for the `Done` message and clear up all states (if in the Pre-Committed State, continue to hold the locks).
  - Receive `Commit T` or `Abort T` message.
  - If `Commit T` is received, commit the component of T at the site, release locks, place `Commit T` to the local log, and send `Done` message to the coordinator.
  - If `Abort T` is received, roll back all changes, release locks, place `Abort T` to the local log, and send `Done` message to the Coordinator.
- **Consistency:** Ensures that either all tasks are completed (committed) or none are (aborted).
- **Reliability:** Prevents a situation where some tasks are completed while others are not, leading to inconsistencies.