CLOUD COMPUTING

Distributed Storage

Suresh Jamadagni

Department of Computer Science and Engineering

CLOUD COMPUTING

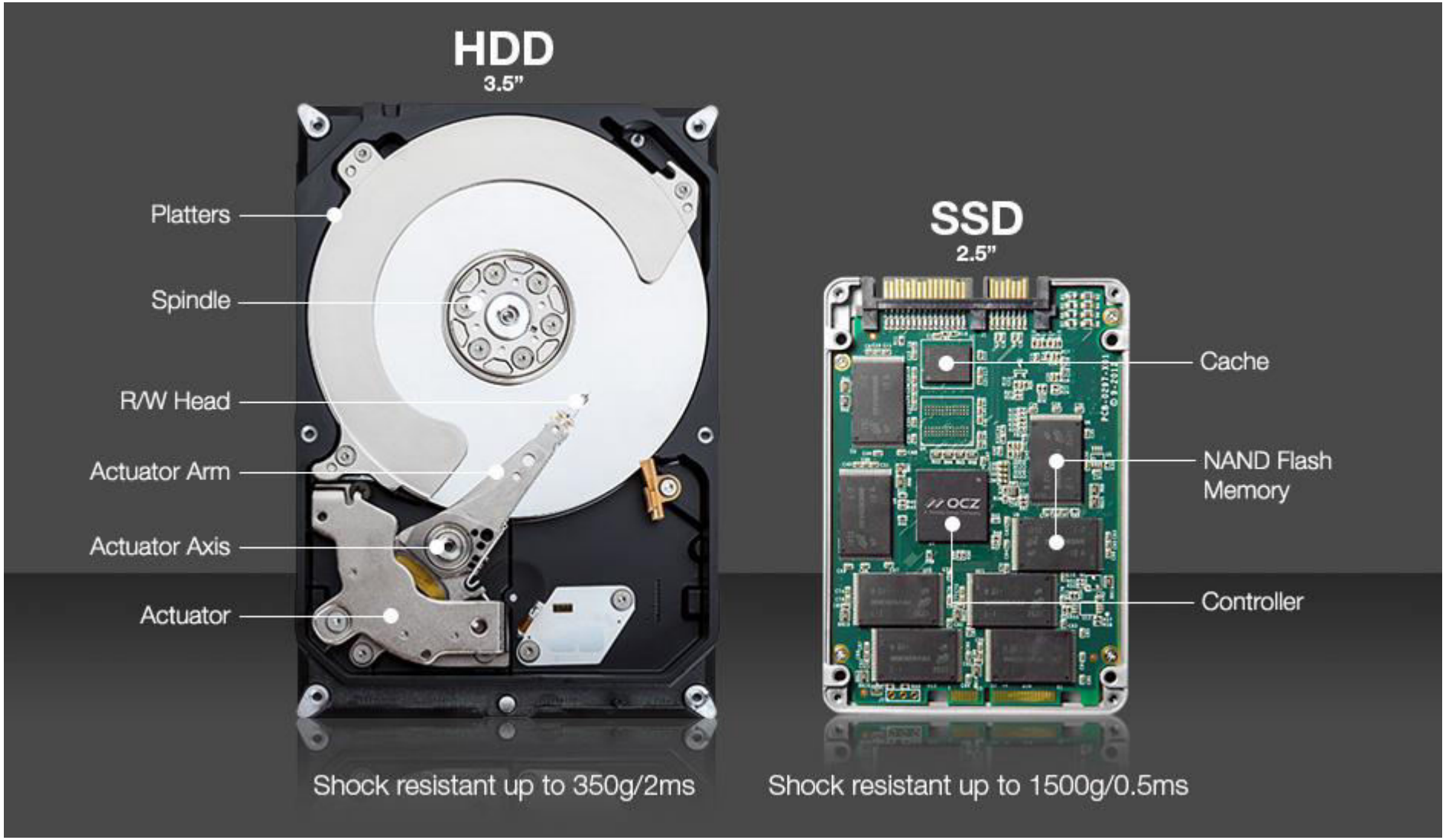
Distributed Storage

Suresh Jamadagni

Department of Computer Science and Engineering

CLOUD COMPUTING

Storage Devices



DATABASE TECHNOLOGIES

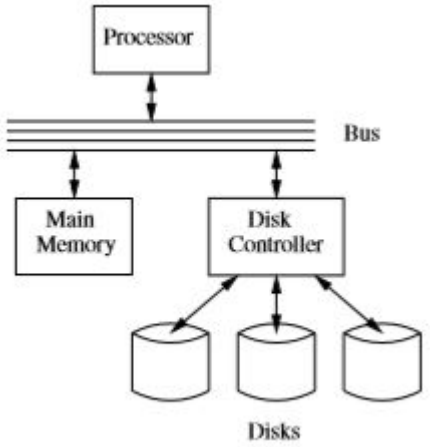
Disk Latency

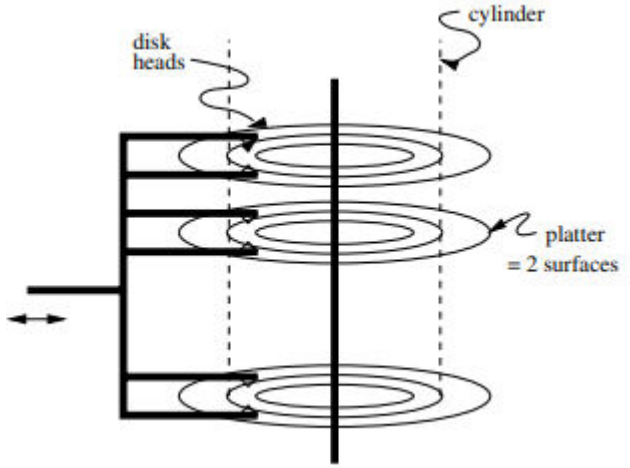
Reading or writing a block involves three steps

1. The disk controller positions the head assembly at the cylinder containing the track on which the block is located. The time to do so is the **seek time**

2. The disk controller waits while the first sector of the block moves under the head. This time is called the **rotational latency**.

3. 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**

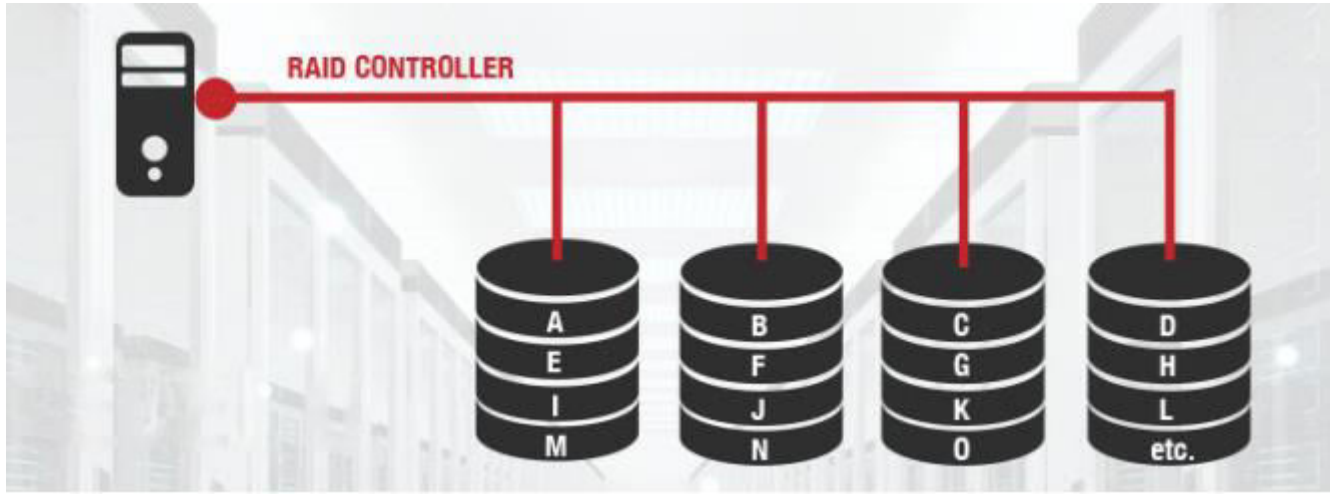




Disk Latency = Seek time + Rotational latency + Transfer time

CLOUD COMPUTING

RAID – Redundant Array of Independent Disks



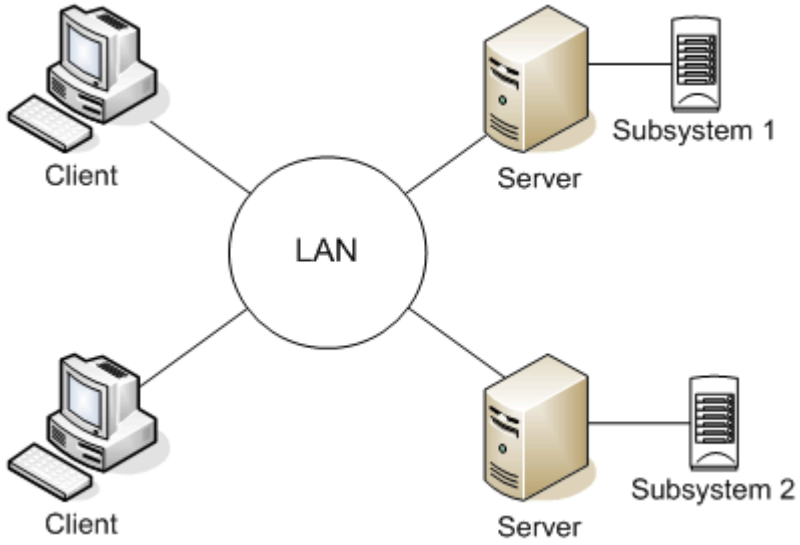
RAID is a **data storage virtualization technology** that combines multiple physical disk drive components into one or more logical units for the purposes of data redundancy and performance improvement.

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

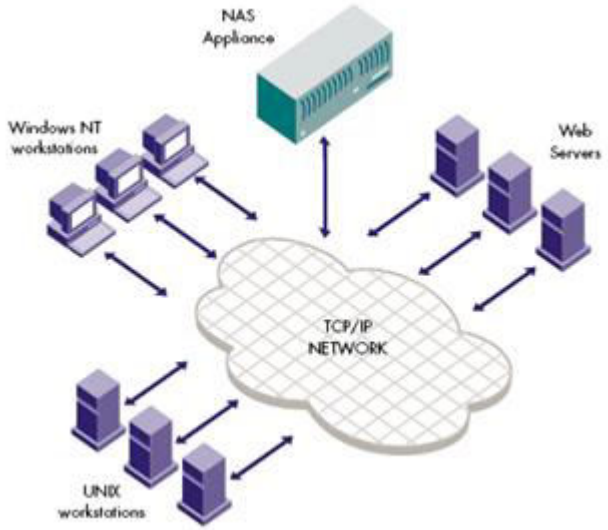
CLOUD COMPUTING

Storage Architecture

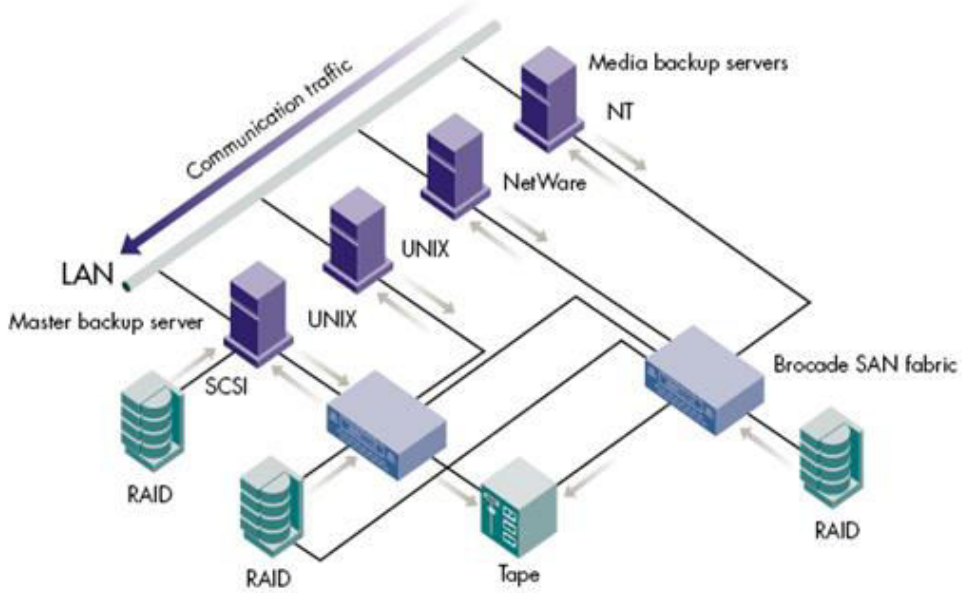
Directly Attached Storage (DAS)



Network Attached Storage (NAS)

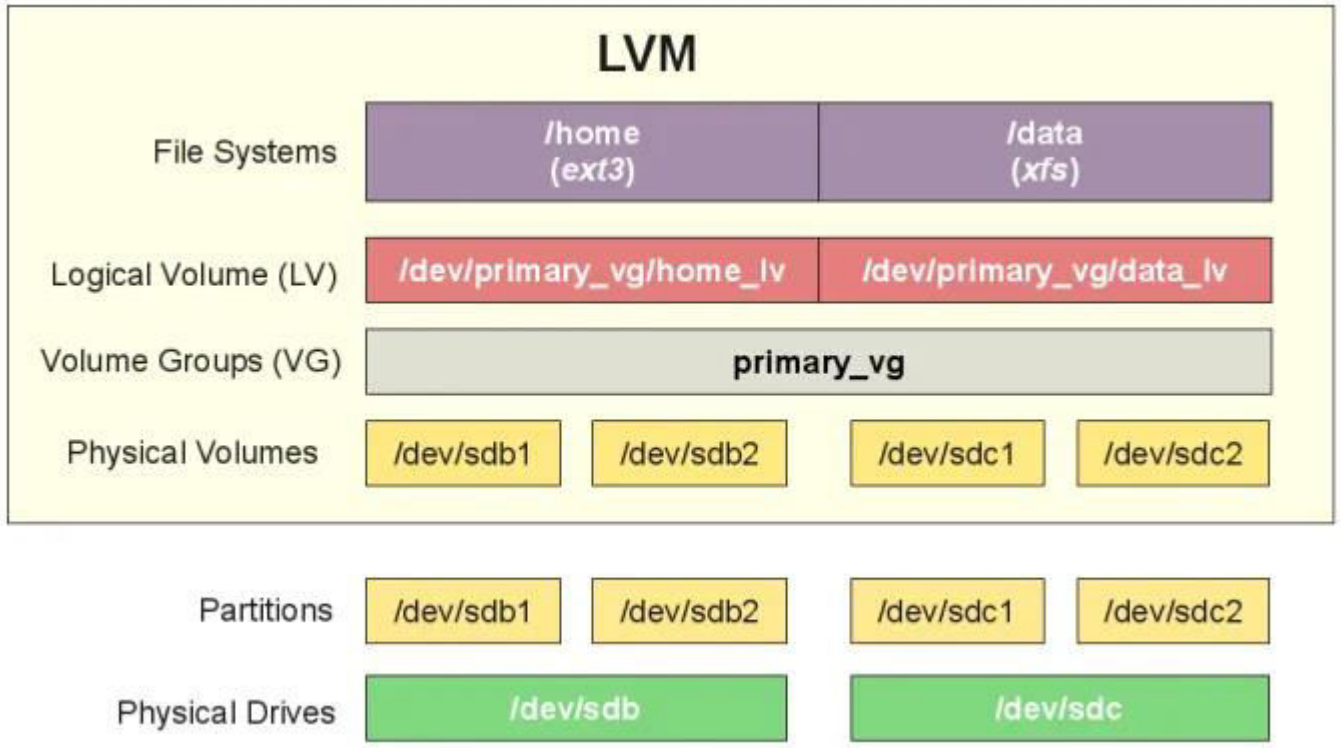


Storage Area Network (SAN)



CLOUD COMPUTING

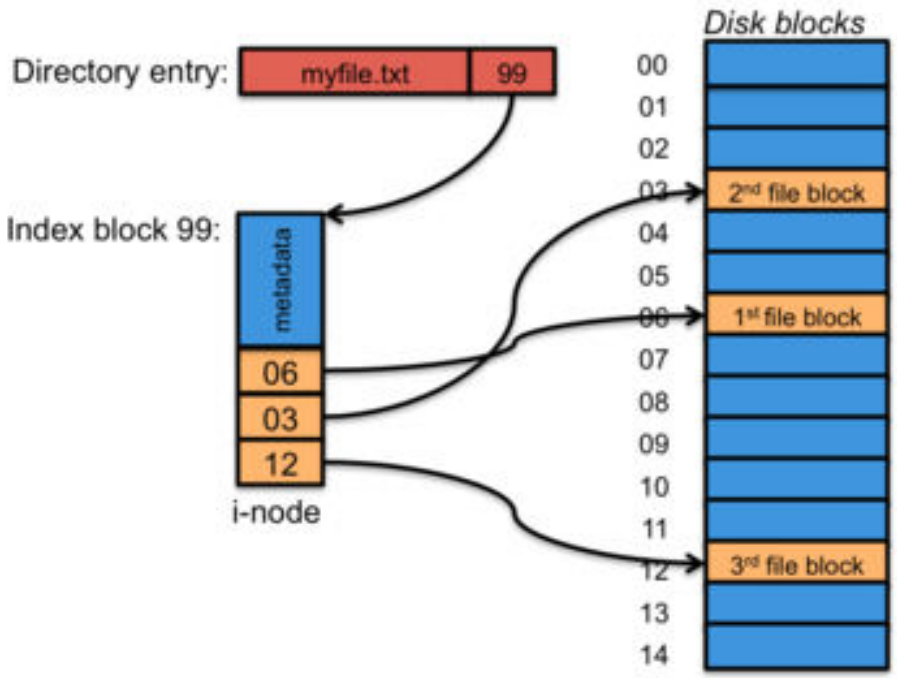
Storage – File Systems

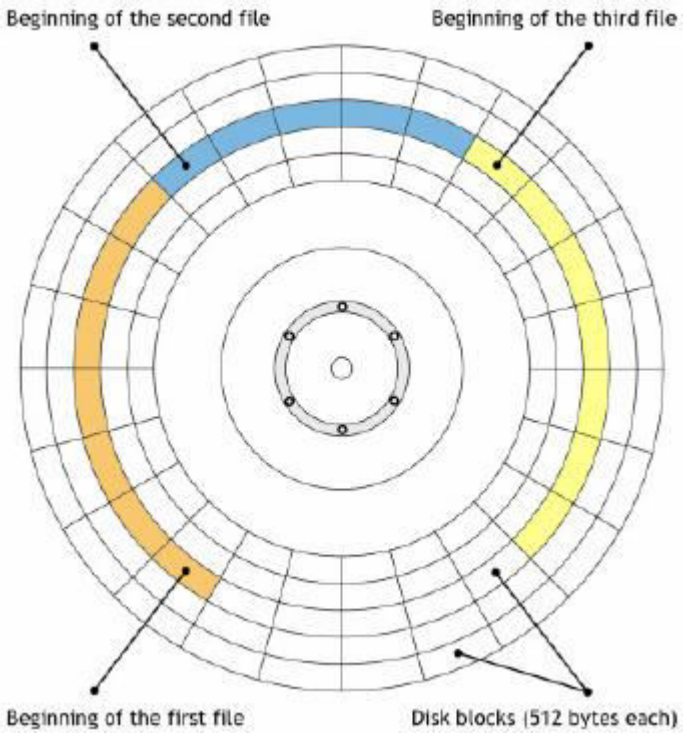


https://opensource.com/business/16/9/linux-users-guide-lvm

CLOUD COMPUTING

Storage – File Allocation Table





Block is a basic unit of storage for IO (Read/Write) operations

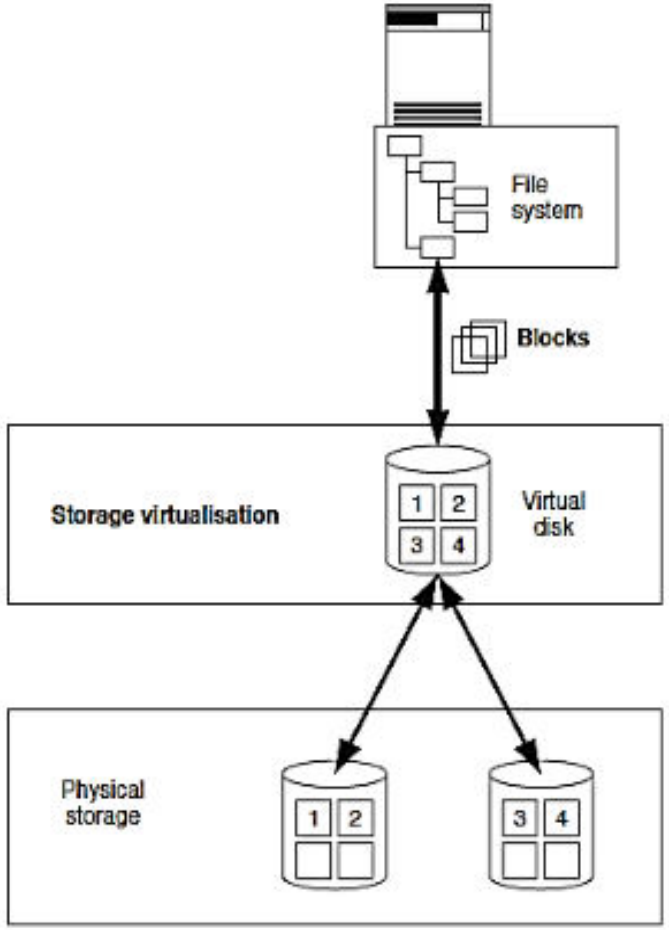
CLOUD COMPUTING

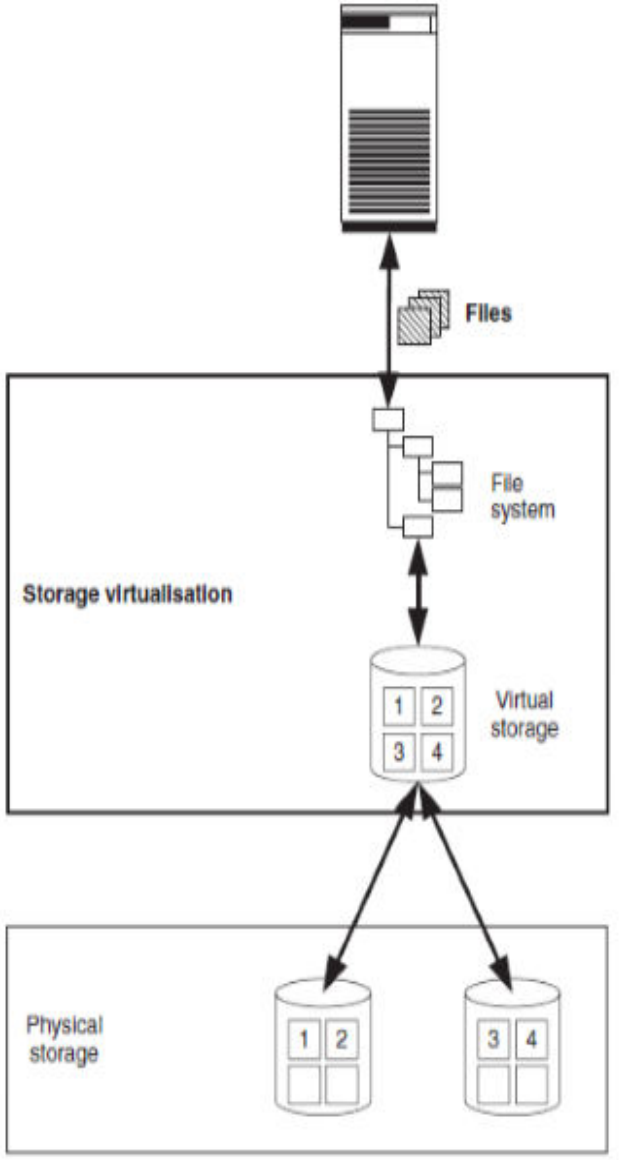
Storage – Virtualization

- Storage Virtualization is a means through which **physical storage subsystems are abstracted from the user's application and presented as logical entities**, hiding the underlying complexity of the storage subsystems and nature of access, network or direct to the physical devices.
- This abstraction helps applications to perform normally despite changes to the storage hardware.
- Storage virtualization enables
  - **Higher resource utilization** by aggregating the capacity of multiple heterogeneous storage devices into storage pools
  - Easy provisioning of the right storage for **performance or cost**, as well as provides ability to **centrally manage pools of storage** and associated services.

CLOUD COMPUTING

Storage – File Level Virtualization vs Block Level Virtualization





https://www.ques10.com/p/20616/explain-block-level-and-file-level-storage-virtual/

Storage – Virtualization

Categories of Storage Virtualization:

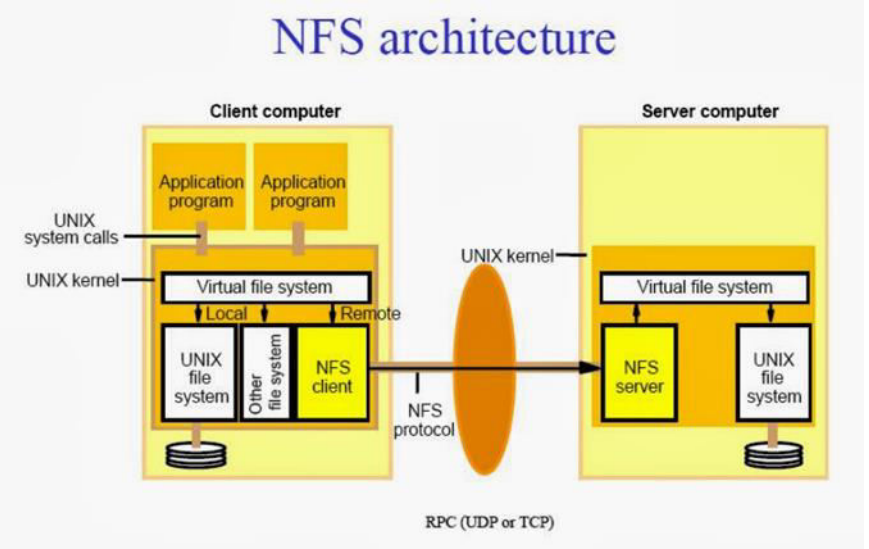
- **File level** - A file system virtualization provides an abstraction of a file system to the application (with a standard file-serving protocol interface such as NFS or CIFS) and manages changes to distributed storage hardware underneath the file system implementation.

- **Block level** - virtualizes multiple physical disks and presents the same as a single logical disk. The data blocks of this logical disk may be internally mapped to one or more physical disks or may reside on multiple storage subsystems.

CLOUD COMPUTING

Storage – File Virtualization

- File virtualization creates an **abstraction layer between file servers and their clients.**
- This 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 multiple file servers enabling access to individual files.
- File-serving protocols that are typically employed are NFS, CIFS and Web interfaces such as HTTP or WebDAV

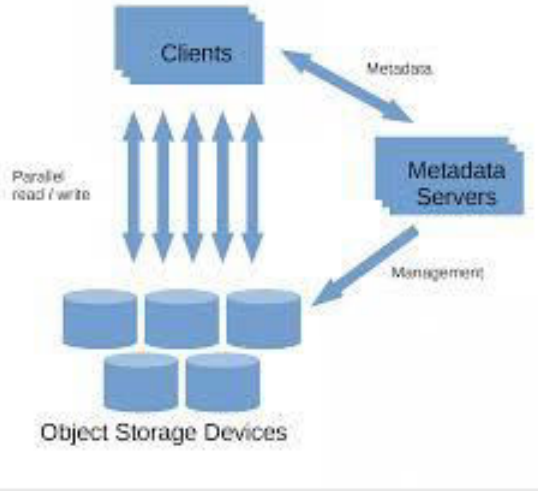


CLOUD COMPUTING

Storage – File Virtualization

Distributed File System

- A distributed file system (DFS) is a network file system wherein the file system is distributed across multiple servers.
- DFS enables location transparency and file directory replication as well as tolerance to faults.
- Some implementations may also cache recently accessed disk blocks for improved performance. Though distribution of file content increases performance considerably, 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 Gluster)



Storage – File Virtualization

Distributed File Systems with Centralized Metadata

- A 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

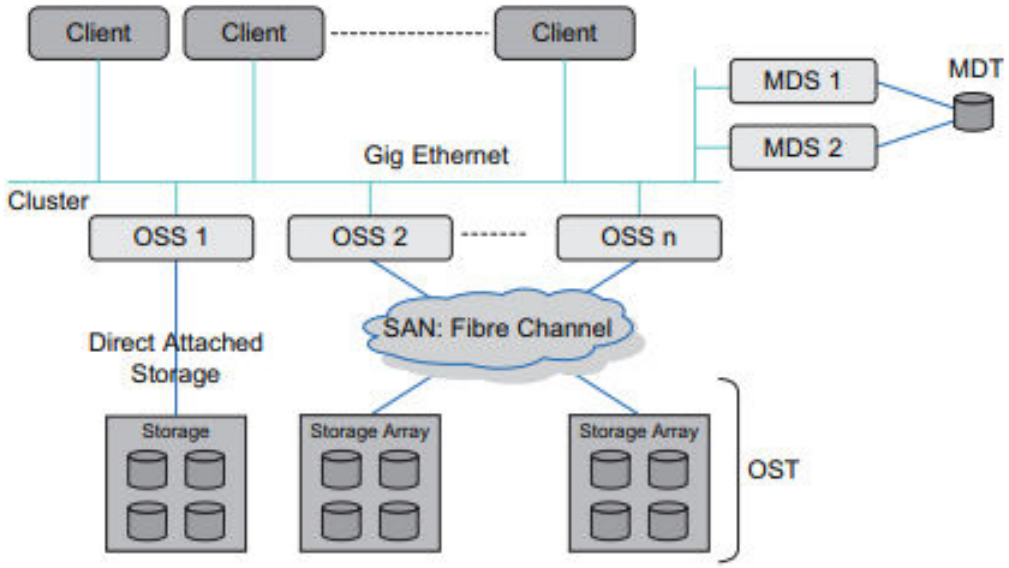
Lustre - Distributed File Systems with Centralized Metadata

- Lustre is a massively parallel, scalable distributed file system for Linux which employs a cluster-based architecture with centralized metadata.
- This is a software solution with an ability to scale over thousands of clients for a storage capacity of petabytes with high performance I/O throughput.
- The architecture of Lustre includes the following three main functional components, which can either 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

CLOUD COMPUTING

Storage – File Virtualization – Lustre architecture

- When a client accesses a file, it does a filename lookup on a MDS.
- Then, 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.



Distributed File Systems with Distributed Metadata

- In distributed metadata management, metadata is distributed across all nodes in the system, rather than using centralized metadata servers.
- Such systems have greater complexity than centralized metadata systems, since the metadata management is spread over all the nodes in the system.

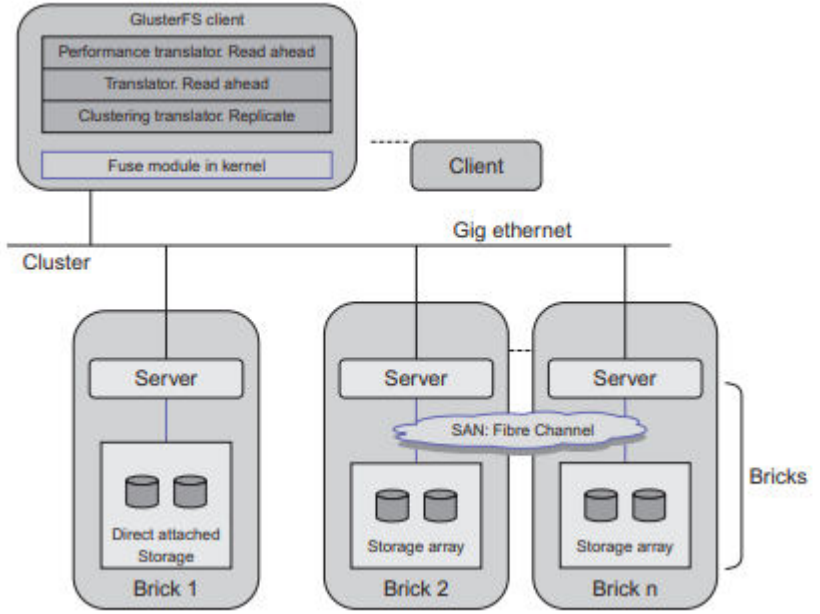
GlusterFS - Distributed File Systems with Distributed Metadata

- GlusterFS is an open-source, distributed cluster file system without a centralized metadata server.
- It is also capable of scaling to thousands of clients, Petabytes of capacity and is optimized for high performance.
- 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 name space.
- 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 actually optional and can be used to implement highly available, massively parallel access to every storage node and handles failure of any single node transparently

CLOUD COMPUTING

Storage – File Virtualization – GlusterFS architecture

- 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 (like filters) 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.
- Translators can be loaded both in client and server side appropriately to improve or achieve new functionalities
- Gluster performs very good load balancing of operations using the I/O Scheduler translators
- GlusterFS also supports file replication with the Automatic File Replication (AFR) translator, which keeps identical copies of a file/directory on all its subvolumes



CLOUD COMPUTING

Storage – Block Virtualization

- Block-level virtualization technique virtualizes multiple physical disks and presents the same as a single logical disk.
- The data blocks are mapped to one or more physical disks sub-systems.
- These block addresses may reside on multiple storage sub-systems, appearing however as a single storage (logical) storage device.
- Block level storage virtualization can be performed at three levels:
  a. Host-Based
  b. Storage Level
  c. Network level

CLOUD COMPUTING

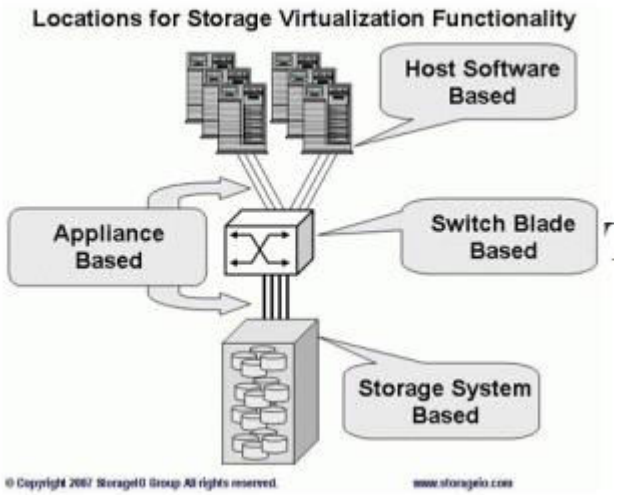
Storage – Block Virtualization

Host-Based block virtualization

- Uses a **Logical Volume Manager** (LVM), a virtualization layer that supports allocation and management of disk space for file systems or raw data with capabilities to dynamically shrink or increase physical volumes, or combine small chunks of unused space from multiple disks or create a logical volume that is greater than the size of the physical disk, all these transparently.

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. Some disk arrays also virtualize third-party external storage devices attached to the array.
- This technique is generally host-agnostic and has low latency since the virtualization is a part of the storage device itself and in the firmware of the device.



Storage – Block Virtualization

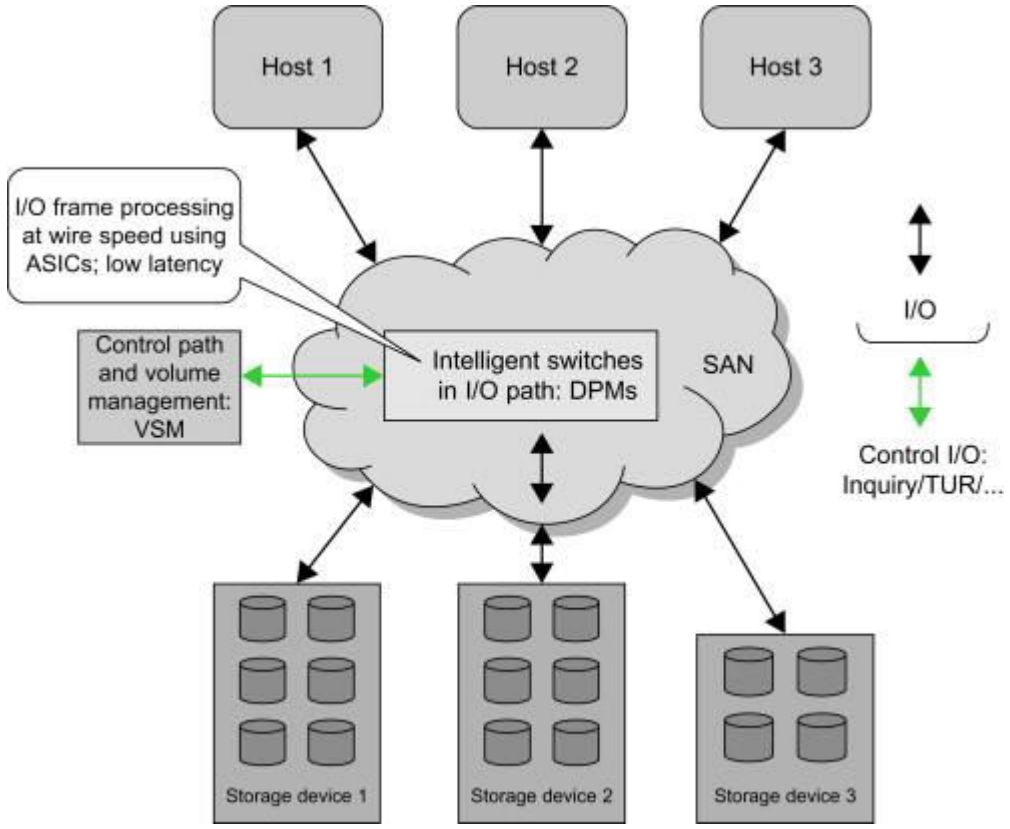
Network-Based block virtualization

- This is the most commonly implemented form of scalable virtualization.
- Virtualization functionality is implemented within the network connecting hosts and storage, say a Fibre Channel Storage Area Network (SAN).
- There are broadly two categories **based on where the virtualization functions are implemented**: either in **switches** (routers) or in **appliances** (servers).
  - In a switch-based network virtualization, the actual virtualization occurs in an intelligent switch in the fabric and the functionality is achieved when it works in conjunction with a metadata manager in the network. Example**: HP StorageWorks SAN Virtualization Services Platform**
  - In an appliance-based approach, the I/O flows through an appliance that controls the virtualization layer. Example: **IBM SAN Volume Controller**

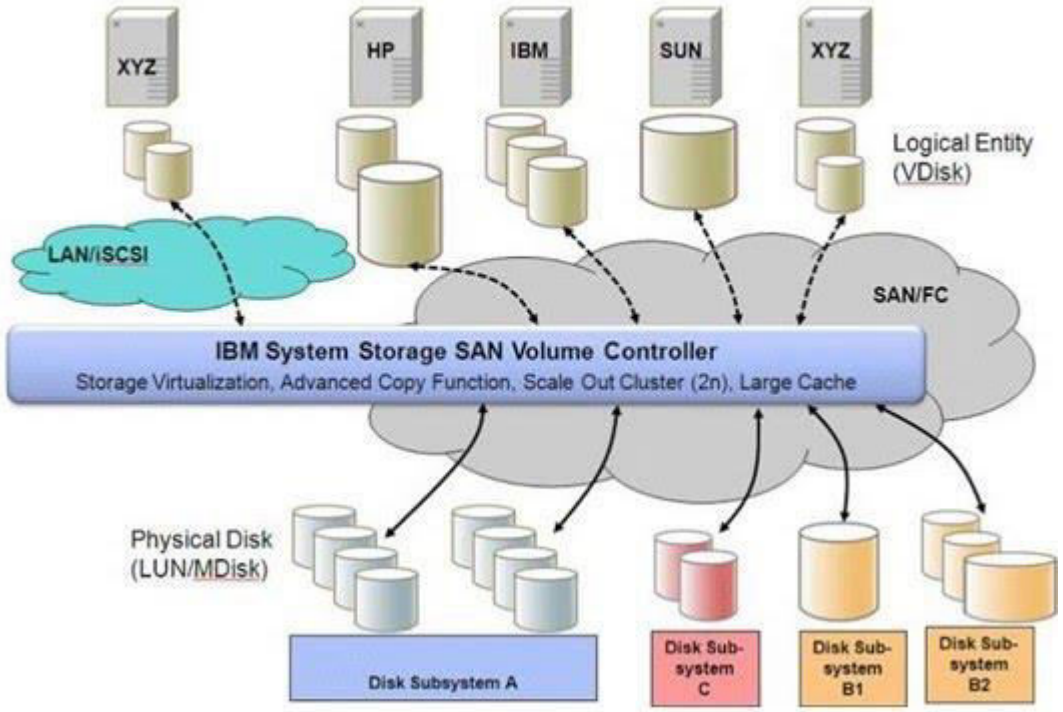
CLOUD COMPUTING

Storage – Block Virtualization

Switch-based network virtualization



Appliance-based network virtualization



Storage – Block Virtualization

Network-Based block virtualization

- There are broadly two variations of an appliance-based implementation.
- The appliance can either be **in-band** or **out-of-band**.
- In **in-band**, all I/O requests and their data pass through the virtualization device and the clients do not interact with the storage device at all. All I/O is performed by the appliance on behalf of the clients
- In **out-of-band** usage, the appliance only comes in between for metadata management (control path), while the data (I/O) path is directly from the client to each host (with agents on each host/client).

THANK YOU

Suresh Jamadagni

Department of Computer Science and Engineering

CLOUD COMPUTING

Distributed Storage

K.S. Srinivas
Suresh Jamadagni
Venkatesh Prasad

Department of Computer Science and Engineering

CLOUD COMPUTING

Distributed Storage - Replication

K. Srinivas
Suresh Jamadagni
Venkatesh Prasad

Department of Computer Science and Engineering

Amazon Simple Storage Service (S3)

- Amazon S3 is a highly reliable, highly available, scalable and fast storage in the cloud for storing and retrieving large amounts of data just through simple web services.
- There are three ways of using S3. Most common operations can be performed via the AWS console (GUI interface to AWS)
- For use of S3 within applications, Amazon provides a REST-ful API with familiar HTTP operations such as GET, PUT, DELETE, and HEAD.
- There are libraries and SDKs for various languages that abstract these operations
- Since S3 is a storage service, several S3 browsers exist that allow users to explore their S3 account as if it were a directory (or a folder). There are also file system implementations that let users treat their S3 account as just another directory on their local disk.

Object Storage - Amazon Simple Storage Service (S3)

Organizing Data In S3: Buckets, Objects and Keys

- Files are called **objects** in S3.
- Objects are referred to with **keys** – basically an optional directory path name followed by the name of the object.
- Objects in S3 are replicated across multiple geographic locations to make it resilient to several types of failures.
- If object versioning is enabled, recovery from inadvertent deletions and modifications is possible.
- S3 objects can be up to 5 Terabytes in size and there are no limits on the number of objects that can be stored.
- All objects in S3 must be stored in a **bucket**.
- 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.

Object Storage - Amazon Simple Storage Service (S3)

Organizing Data In S3: Buckets, Objects and Keys

- Each object has a key, which can be used as the path to the resource in an HTTP URL.
- **Example**: if the bucket is named johndoe and the key to an object is resume.doc, then its HTTP URL is http://s3.amazonaws.com/johndoe/resume.doc or alternatively, http://johndoe.s3.amazonaws.com/resume.doc
- By convention, slash-separated keys are used to establish a directory-like naming scheme for convenient browsing in S3

Object Storage - Amazon Simple Storage Service (S3)

Security

- Users can ensure the security of their S3 data by two methods
  1. **Access control to objects**: Users can set permissions that allow others to access their objects. This is accomplished via the AWS Management Console.
  2. **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

Object Storage - Amazon Simple Storage Service (S3)

Data Protection

1. **Replication**:
   - By default, S3 replicates data across multiple storage devices, and is designed to survive two replica failures.
   - It is also possible to request Reduced Redundancy Storage(RRS) for noncritical data. RRS data is replicated twice, and is designed to survive one replica failure.
   - S3 does not guarantee consistency among the replicas.
2. **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

Data Protection

3. **Regions**:
    - For performance, legal 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.

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 the network utilization. If a part fails to upload, only that part needs to be re-tried

CLOUD COMPUTING

Object Storage – NoSQL Database - DynamoDB

- DynamoDB is a cloud-based 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 schemaless flavor.
- DynamoDB enables users to query data based on secondary indexes 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. 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

CLOUD COMPUTING

Object Storage - DynamoDB

- 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
- 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.

People

Secondary Indexes

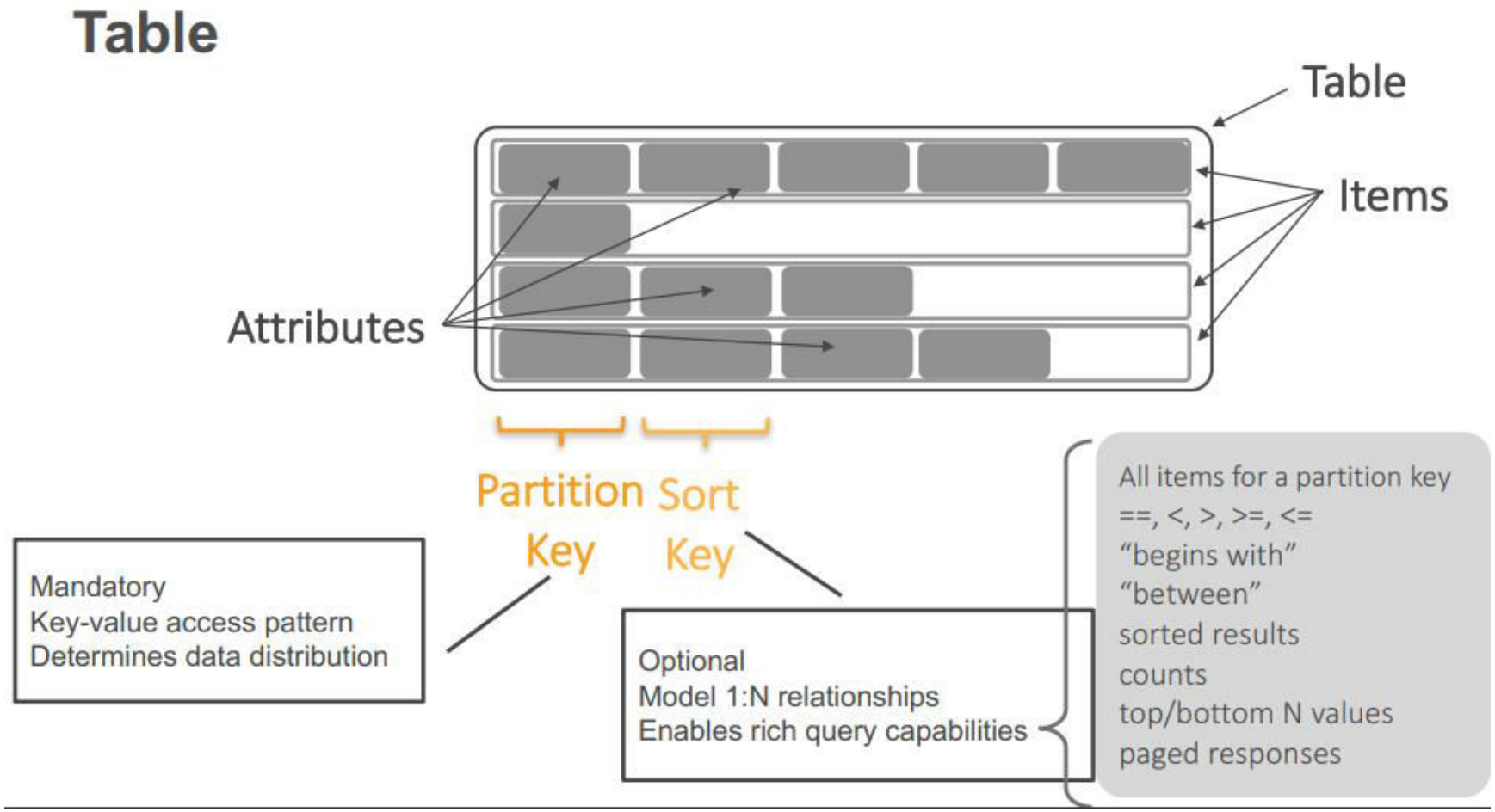
- Users can create one or more secondary indexes on a table.
- A secondary index lets users query the data in the table using an alternate key, in addition to queries against the primary key.

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

CLOUD COMPUTING

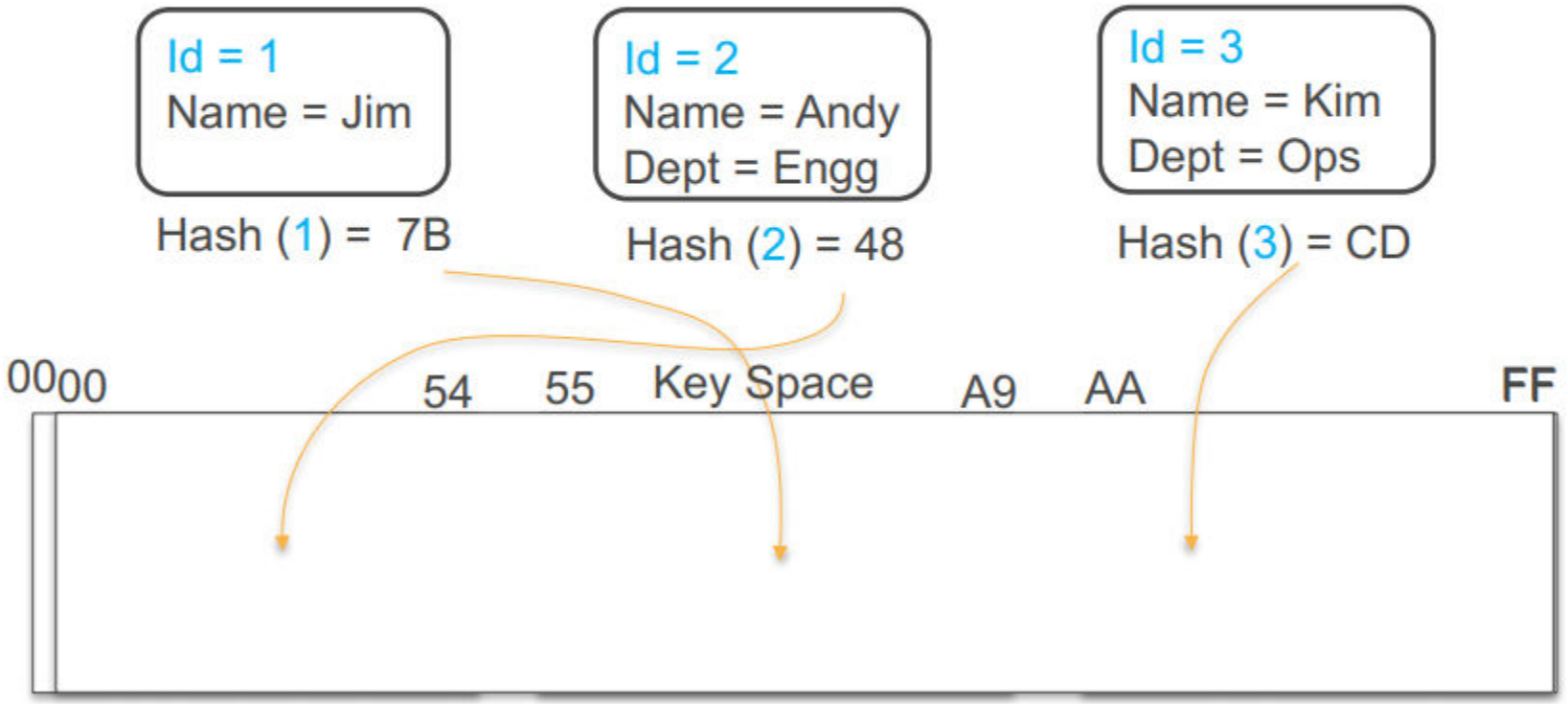
Object Storage - DynamoDB Architecture



Object Storage - DynamoDB Architecture

Partition table

Partition key uniquely identifies an item
Partition key is used for building an unordered hash index
Table can be partitioned for scale

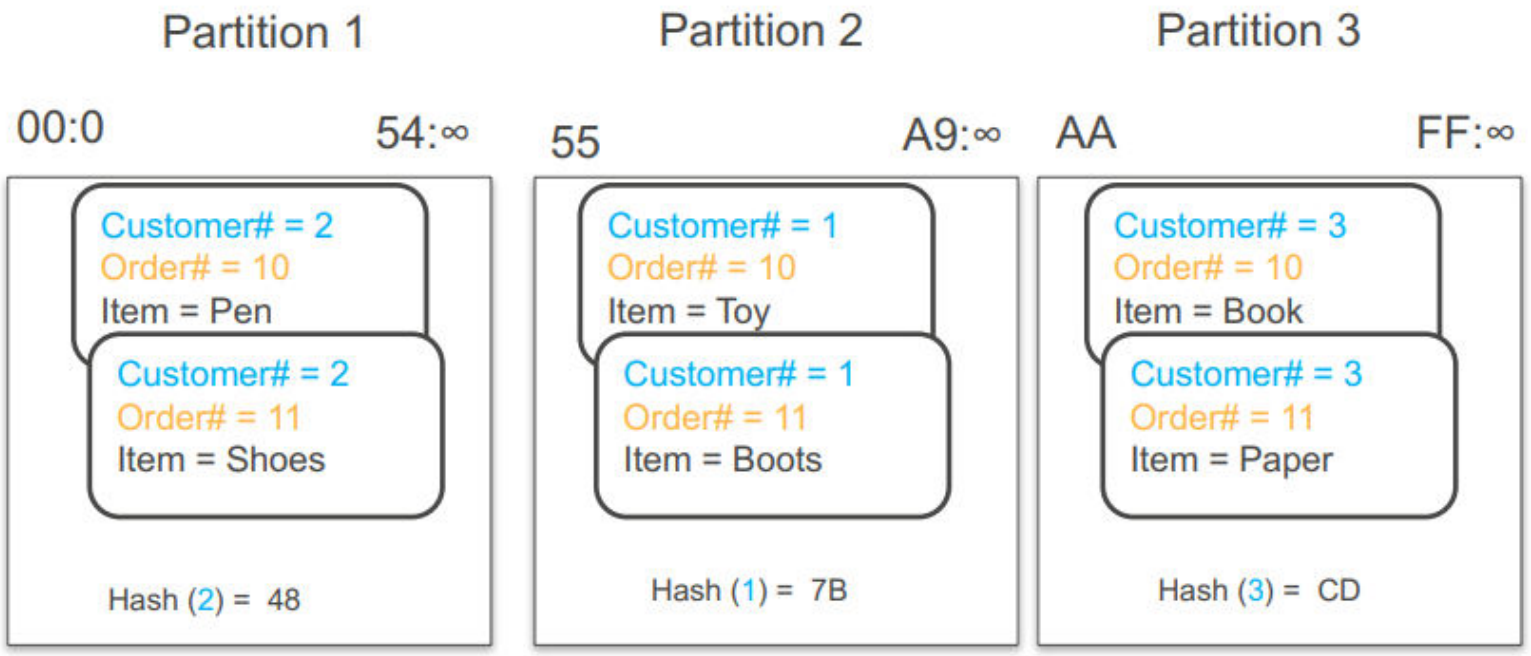


Object Storage - DynamoDB Architecture

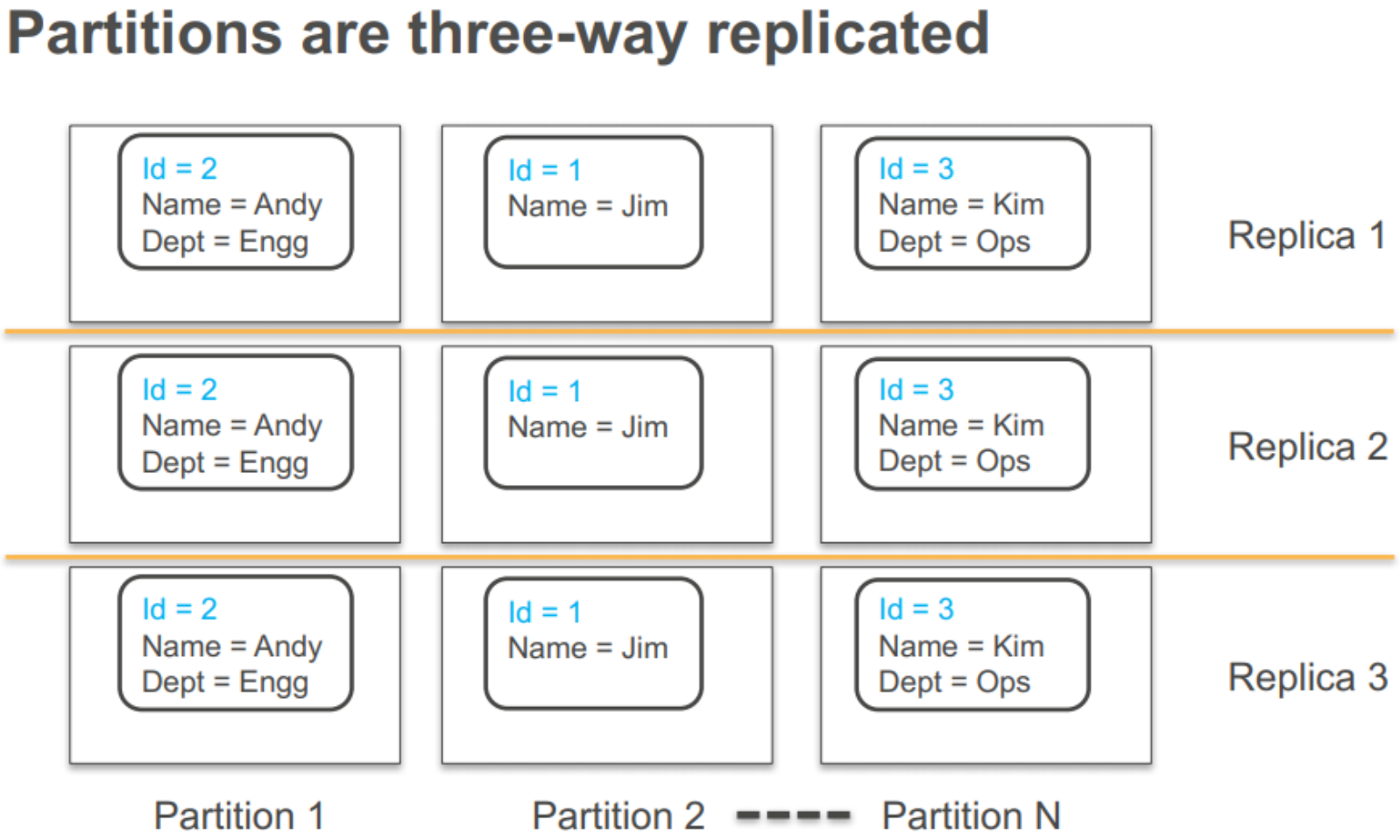
Partition-sort key table

Partition key and sort key together uniquely identify an Item
Within unordered partition key-space, data is sorted by the sort key
No limit on the number of items (∞) per partition key

- Except if you have local secondary indexes



DynamoDB Architecture



CLOUD COMPUTING

Object Storage – Amazon RDS (Relational Database Service)

- Amazon Relational Database Service (RDS) provides a relational database abstraction in the cloud.
- RDS provides six familiar database engines to choose from, including Amazon Aurora, PostgreSQL, MySQL, MariaDB, Oracle Database and SQL Server.
- AWS performs many of the administrative tasks associated with maintaining a database for the user. The database is backed up at configurable intervals. AWS also provides the capability to snapshot the database as needed.
- RDS provides encryption at rest and in transit.
- RDS provides APIs for application development.

THANK YOU

K.S. Srinivas
Suresh Jamadagni
Venkatesh Prasad

Department of Computer Science and Engineering

CLOUD COMPUTING

Distributed Storage

Suresh Jamadagni
Department of Computer Science and Engineering

CLOUD COMPUTING

Distributed Storage – Partitioning

Suresh Jamadagni

Department of Computer Science and Engineering

CLOUD COMPUTING

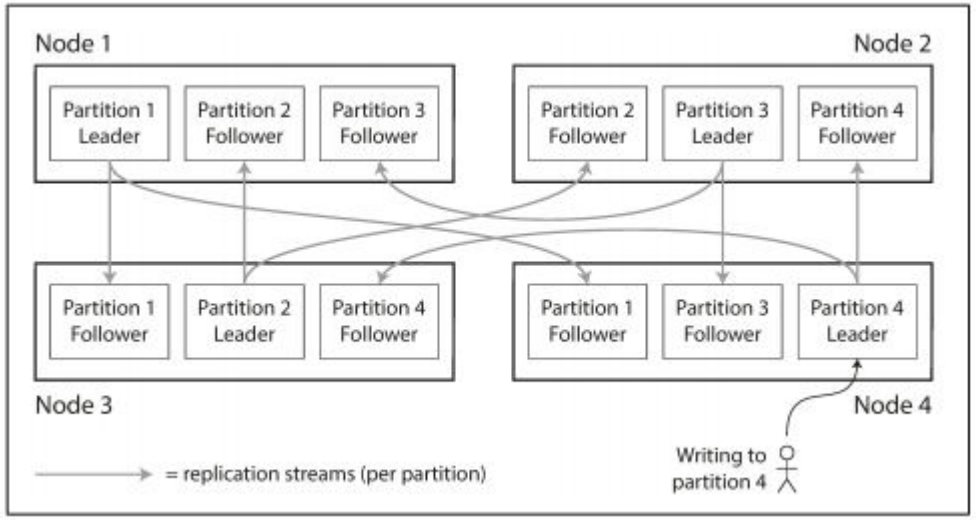
Partitioning

- Partitioning, is a way of intentionally breaking a large database down into smaller ones
- Partitions are defined in such a way that each piece of data (each record, row, or document) belongs to exactly one partition.
- In effect, each partition is a small database of its own, although the database may support operations that touch multiple partitions at the same time
- The main reason for wanting to partition data is scalability.
- Different partitions can be placed on different nodes in a cluster
- Thus, a large dataset can be distributed across many disks, and the query load can be distributed across many processors.
- Each node can independently execute the queries that operate on a single partition (own partition), so query throughput can be scaled by adding more nodes.

CLOUD COMPUTING

Partitioning

- Large complex queries 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 means that, even though each record belongs to exactly one partition, it may still be stored on several different nodes for fault tolerance.
- A node may store more than one partition



Partitioning of Key-Value data

- Goal with partitioning is to spread the data and the query load evenly across nodes.
- If some partitions have more data than others, we call it **skewed**
- The presence of skew makes partitioning much less effective. In an extreme case, all the load could end up on one partition.
- A partition with disproportionately high load is called a **hot spot**
- The simplest approach for avoiding hot spots would be to assign records to nodes randomly. That would distribute the data quite evenly across the nodes, but it has a big disadvantage. 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.

Partitioning of Key-Value data

Partitioning by key range

- Assign a continuous range of keys (from some minimum to some maximum) to each partition
- If we know the boundaries between the ranges, we can easily determine which partition contains a given key
- If we also know which partition is assigned to which node, then 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.

Partitioning of Key-Value data

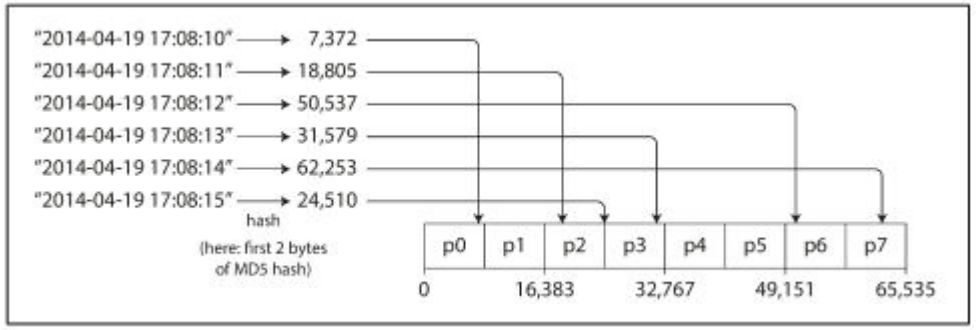
Partitioning by key range

- **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 in the case where all the readings for a particular month need to be fetched.
- The downside 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), so that partition can be overloaded with writes while others are idle
- To avoid this problem, use some other attribute other than the timestamp as the first element of the key

Partitioning of Key-Value data

Partitioning by hash of key

- Because of the risk of skew and hot spots, many distributed datastores use a hash function to determine the partition for a given key
- A good hash function takes skewed data and makes it uniformly distributed
- Using a suitable hash function for keys, you can assign each partition a range of hashes (rather than a range of keys), and every key whose hash falls within a partition's range will be stored in that partition.



Partitioning of Key-Value data

Partitioning by hash of key

- By using the hash of the key for partitioning we lose the ability to do efficient range queries
- The concatenated index (composite key) approach enables an elegant data model for one-to-many relationships.
- Example: On a social media site, one user may post many updates. If the primary key for updates is chosen to be (user_id, update_timestamp), then you can efficiently retrieve all updates made by a particular user within some time interval, 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.

CLOUD COMPUTING

Example: Partitioning of Key-Value data

```
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

THANK YOU

Suresh Jamadagni

Department of Computer Science and Engineering

CLOUD COMPUTING

Distributed Storage

Suresh Jamadagni
Department of Computer Science and Engineering

CLOUD COMPUTING

Distributed Storage – Partitioning

Suresh Jamadagni

Department of Computer Science and Engineering

CLOUD COMPUTING

Consistent Hashing

- Consistent hashing is a way of evenly distributing load across an internet-wide system of caches such as a content delivery network (CDN).
- It uses randomly chosen partition boundaries to avoid the need for central control or distributed consensus.
- Note that consistent here has nothing to do with replica consistency or ACID consistency, but rather describes a particular approach to rebalancing the partitions.

Skewed Workloads and Relieving Hot Spots

- Hashing a key to determine its partition can help reduce hot spots. However, it can't avoid them entirely
- In the extreme case where all reads and writes are for the same key, all requests will be routed to the same partition.
- Most data systems are not able to automatically compensate for such a highly skewed workload, so it's the responsibility of the application to reduce the skew.
- If one key is known to be very hot, a simple technique is to add a random number to the beginning or end of the key
- Having split the writes across different keys, any reads now have to do additional work, as they have to read the data from all the keys and combine it

Partitioning and Secondary Indexes

- If records are only accessed via their primary key, we can determine the partition from that key and use it to route read and write requests to the partition responsible for that key.
- A **secondary index** usually doesn't identify a record uniquely but rather is a way of searching for occurrences of a particular value
- Secondary indexes don't map neatly to partitions.
- Two main approaches to partitioning a database with secondary indexes:
  1. Document-based partitioning
  2. Term-based partitioning

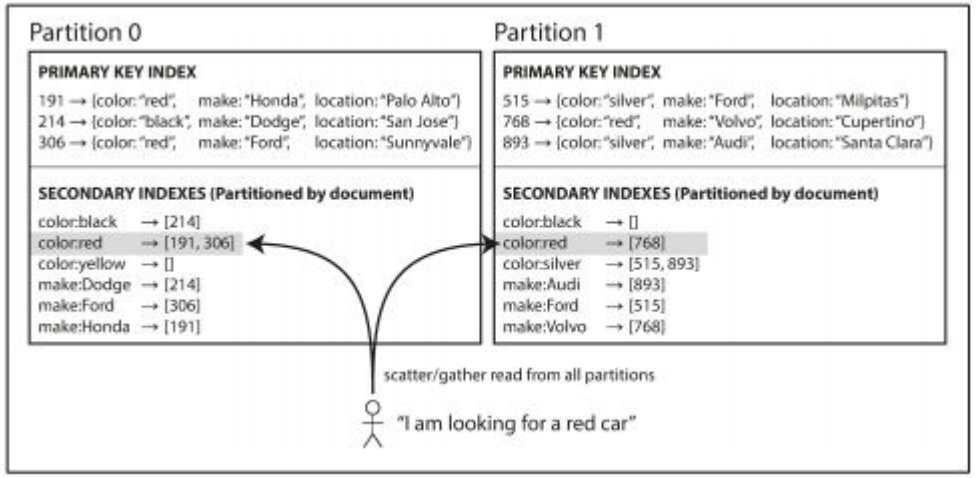
Partitioning Secondary Indexes by Document

- In this indexing approach, each partition is completely separate
- Each partition maintains its own secondary indexes, covering only the documents in that partition. It doesn't care what data is stored in other partitions.
- Whenever you need to write to the database, you only need to deal with the partition that contains the document ID that you are writing. For that reason, a document-partitioned index is also known as a **local index**
- Reading from a document-partitioned index requires all partitions to be queried and combining all the results.
- This approach to querying a partitioned database is sometimes known as **scatter/ gather** and it can make read queries on secondary indexes quite expensive.

CLOUD COMPUTING

Partitioning Secondary Indexes by Document

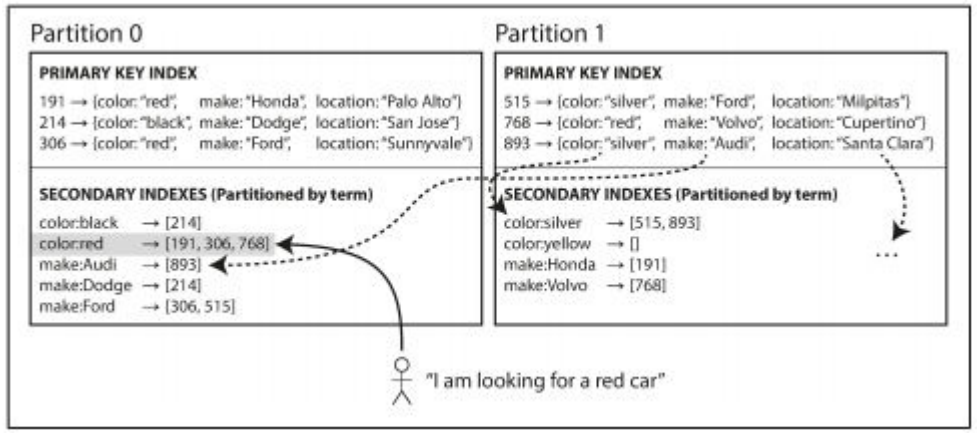
- Even if the partitions are queried in parallel, scatter/gather is prone to tail latency amplification
- Most database vendors recommend that you structure your partitioning scheme so that secondary index queries can be served from a single partition, but that is not always possible, especially when we're using multiple secondary indexes in a single query



CLOUD COMPUTING

Partitioning Secondary Indexes by Term

- Construct a global index that covers data in all partitions
- A global index must also be partitioned, but it can be partitioned differently from the primary key index.
- This kind of index is called **term-partitioned**, because the term we're looking for determines the partition of the index.



Partitioning Secondary Indexes by Term

- We can partition the index by the term itself or using a hash of the term.
- Partitioning by the term itself can be useful for range scans, whereas partitioning on a hash of the term gives a more even distribution of load.
- The advantage of a global (term-partitioned) index over a document-partitioned index is that it can make reads more efficient: rather than doing scatter/gather over all partitions, a client only needs to make a request to the partition containing the term that it wants.
- The downside of a global index is that writes are slower and more complicated, because a write to a single document may now affect multiple partitions of the index (every term in the document might be on a different partition on a different node).

Partitioning Secondary Indexes by Term

- The index needs to be up to date so that every document written to the database would immediately be reflected in the index.
- In a term partitioned index, that would require a distributed transaction across all partitions affected by a write, which is not supported in all databases
- In practice, updates to global secondary indexes are often asynchronous
- Therefore, if you read the index shortly after a write, the change you just made may not yet be reflected in the index

THANK YOU

Suresh Jamadagni

Department of Computer Science and Engineering

CLOUD COMPUTING

Distributed Storage

Suresh Jamadagni
Department of Computer Science and Engineering

CLOUD COMPUTING

Distributed Storage – Partitioning

Suresh Jamadagni

Department of Computer Science and Engineering

Rebalancing Partitions

- Over time, many things change in a database
- These changes call for data and requests to be moved from one node to another.
- The process of moving load from one node in the cluster to another is called **rebalancing**.
- Rebalancing is expected to meet some minimum requirements regardless of the partitioning scheme:
  - After rebalancing, the load (data storage, read and write requests) should be shared fairly between the nodes in the cluster.
  - While rebalancing is in progress, the database should continue to accept reads and writes.
  - No more data than necessary should be moved between nodes, to make rebalancing fast and to minimize the network and disk I/O load.

Strategies for Rebalancing Partitions

hash mod N

- hash(key) mod N would return a number between 0 and N-1. An easy way of assigning each key to a node.
- Problem with this approach is that if the number of nodes N changes, most of the keys will need to be moved from one node to another
- Such frequent moves make rebalancing excessively expensive.

Strategies for Rebalancing Partitions

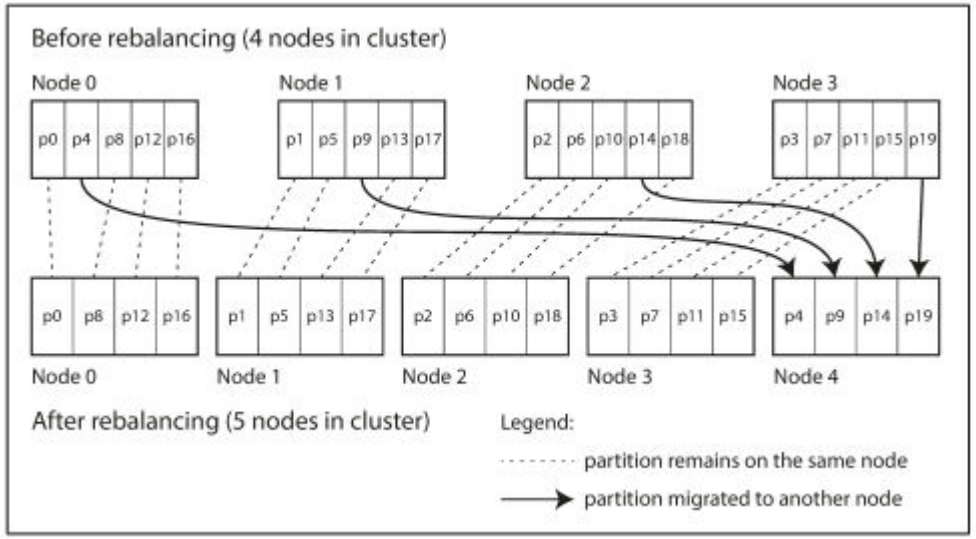
Fixed number of partitions

- Create many more partitions than there are nodes and assign several partitions to each node
- If a node is added to the cluster, the new node can steal a few partitions from every existing node until partitions are fairly distributed once again
- Only entire partitions are moved between nodes.
- The number of partitions does not change, nor does the assignment of keys to partitions. The only thing that changes is the assignment of partitions to nodes
- This change of assignment is not immediate. It takes some time to transfer a large amount of data over the network so the old assignment of partitions is used for any reads and writes that happen while the transfer is in progress.

Strategies for Rebalancing Partitions

Fixed number of partitions

- The number of partitions is usually fixed when the database is first set up and not changed afterward. Although in principle it's possible to split and merge partitions
- A fixed number of partitions is operationally simpler. So many fixed-partition databases choose not to implement partition splitting



Strategies for Rebalancing Partitions

Fixed number of partitions

- Choosing the right number of partitions is difficult if the total size of the dataset is highly variable
- The best performance is achieved when the size of partitions is “just right,” neither too big nor too small, which can be hard to achieve if the number of partitions is fixed but the dataset size varies.

Strategies for Rebalancing Partitions

Dynamic partitioning

- For databases that use key range partitioning , a fixed number of partitions with fixed boundaries would be very inconvenient if the boundaries are wrong.
- All of the data could end up in one partition and all of the other partitions could remain empty
- When a partition grows to exceed a configured size , it is split into two partitions so that approximately half of the data ends up on each side of the split
- If lots of data is deleted and a partition shrinks below some threshold, it can be merged with an adjacent partition
- Each partition is assigned to one node and each node can handle multiple partitions

Strategies for Rebalancing Partitions

Dynamic partitioning

- After a large partition has been split, one of its two halves can be transferred to another node in order to balance the load.
- Advantage of dynamic partitioning is that the number of partitions adapts to the total data volume
- An empty database starts off with a single partition, all writes have to be processed by a single node while the other nodes sit idle
- Dynamic partitioning is not only suitable for key range partitioned data, but can equally well be used with hash partitioned data

Strategies for Rebalancing Partitions

Partitioning proportionally to nodes

- With dynamic partitioning, the number of partitions is proportional to the size of the dataset, since the splitting and merging processes keep the size of each partition between some fixed minimum and maximum
- 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
- Make the number of partitions proportional to the number of nodes—in other words, to have a fixed number of partitions per node
- The size of each partition grows proportionally to the dataset size while the number of nodes remains unchanged, but when you increase the number of nodes, the partitions become smaller again
- Since a larger data volume generally requires a larger number of nodes to store, this approach also keeps the size of each partition fairly stable.

Strategies for Rebalancing Partitions

Partitioning proportionally to nodes

- When a new node joins the cluster, it randomly chooses a fixed number of existing partitions to split, and then takes ownership of one half of each of those split partitions while leaving the other half of each partition in place
- The randomization can produce unfair splits, but when averaged over a larger number of partitions, the new node ends up taking a fair share of the load from the existing nodes.
- Picking partition boundaries randomly requires that hash-based partitioning is used
- This approach corresponds most closely to the original definition of **consistent hashing**

THANK YOU

Suresh Jamadagni

Department of Computer Science and Engineering

CLOUD COMPUTING

Distributed Storage

Suresh Jamadagni
Department of Computer Science and Engineering

CLOUD COMPUTING

Distributed Storage – Request Routing

Suresh Jamadagni

Department of Computer Science and Engineering

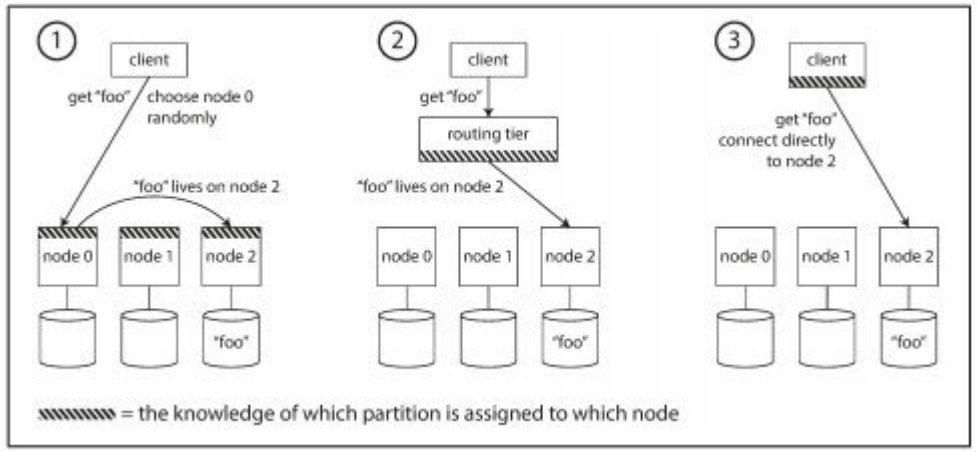
CLOUD COMPUTING

Request Routing

- Consider a partitioned dataset across multiple nodes running on multiple machines
- When a client wants to make a request, how does it know which node to connect to?
- As partitions are rebalanced, the assignment of partitions to nodes change.
- Distributed Database Catalogs
  - Contains metadata of nodes, partitions, key ranges etc

CLOUD COMPUTING

Request Routing

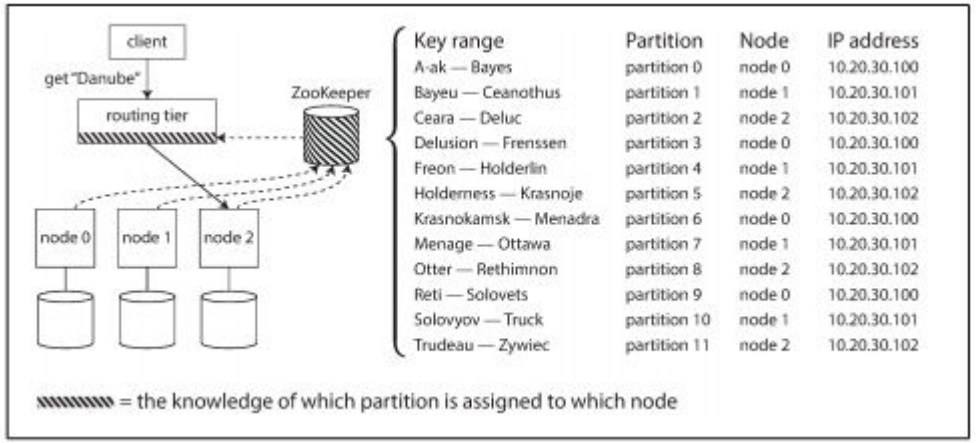


1. Allow clients to contact any node (e.g., via a round-robin load balancer). If that node coincidentally owns the partition to which the request applies, it can handle the request directly; otherwise, it forwards the request to the appropriate node, receives the reply and passes the reply along to the client.
2. Send all requests from clients to a routing tier first, which determines the node that should handle each request and forwards it accordingly. This routing tier does not itself handle any requests; it only acts as a partition-aware load balancer.
3. Require that clients be aware of the partitioning and the assignment of partitions to nodes. In this case, a client can connect directly to the appropriate node, without any intermediary

Request Routing

ZooKeeper

- Many distributed data systems rely on a separate coordination service such as ZooKeeper to keep track of this 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 the partitioning-aware client, can subscribe to this information in ZooKeeper.
- Whenever a partition changes ownership or a node is added or removed, ZooKeeper notifies the routing tier so that it can keep its routing information up to date.



Key range	Partition	Node	IP address
A-ak — Bayes	partition 0	node 0	10.20.30.100
Bayeu — Ceanothus	partition 1	node 1	10.20.30.101
Ceara — Deluc	partition 2	node 2	10.20.30.102
Delusion — Frenssen	partition 3	node 0	10.20.30.100
Freon — Holderlin	partition 4	node 1	10.20.30.101
Holderness — Krasnoje	partition 5	node 2	10.20.30.102
Krasnokamsk — Menadra	partition 6	node 0	10.20.30.100
Menage — Ottawa	partition 7	node 1	10.20.30.101
Otter — Rethimnon	partition 8	node 2	10.20.30.102
Reti — Solovets	partition 9	node 0	10.20.30.100
Solovyov — Truck	partition 10	node 1	10.20.30.101
Trudeau — Zywiec	partition 11	node 2	10.20.30.102

ZooKeeper - A Distributed Coordination Service for Distributed Applications

https://cwiki.apache.org/confluence/display/ZOOKEEPER/Index

- HBase, SolrCloud and Kafka use ZooKeeper to track partition assignment
- LinkedIn's Espresso uses Helix for cluster management (which in turn relies on ZooKeeper), implementing a routing tier

- Cassandra and Riak use a gossip protocol among the nodes to disseminate any changes in cluster state. Requests can be sent to any node, and that node forwards them to the appropriate node for the requested partition

THANK YOU

Suresh Jamadagni

Department of Computer Science and Engineering

CLOUD COMPUTING

Distributed Storage

Suresh Jamadagni

Department of Computer Science and Engineering

CLOUD COMPUTING

Distributed Storage - Replication

Suresh Jamadagni

Department of Computer Science and Engineering

CLOUD COMPUTING

Replication

- **Replication** means keeping a copy of the same data on multiple machines that are connected via a network.
- Reasons for replicating data:
  1. To keep data geographically close to the users (and thus reduce latency)
  2. To allow the system to continue working even if some of its parts have failed (and thus increase availability)
  3. To scale out the number of machines that can serve read queries (and thus increase read throughput)
- Three popular algorithms for replicating changes between nodes: **single-leader, multi-leader,** and **leaderless replication**

Replication

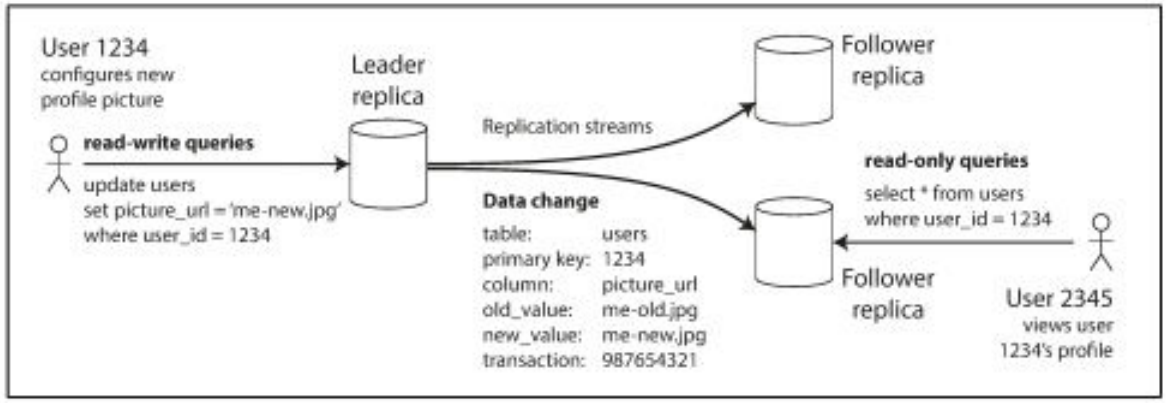
Leader based replication (also known as active/passive or master/slave replication)

- Each node that stores a copy of the database is called a replica.
- How do we ensure that all the data is consistent across multiple replicas?
  - One of the replicas is designated the leader (also known as master or primary).
  - When clients want to write to the database, they must send their requests to the leader, which first writes the new data to its local storage.
  - The other replicas are known as followers (read replicas, slaves, secondaries, or hot standbys).
  - Whenever the leader writes new data to its local storage, it also sends the data change to all of its followers as part of a replication log or change stream.
  - Each follower takes the log from the leader and updates its local copy of the data- base accordingly, by applying all writes in the same order as they were processed on the leader.

Replication

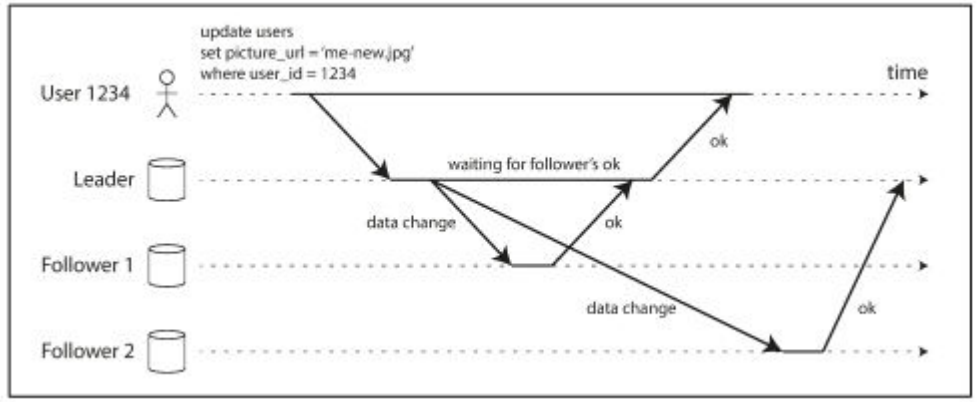
Leader based replication (also known as active/passive or master/slave replication)

- When a client wants to read from the database, it can query either the leader or any of the followers. However, writes are only accepted on the leader.
- This mode of replication is a built-in feature of many relational databases such as PostgreSQL, MySQL, Oracle Data Guard and SQL Server's AlwaysOn. It is also used in some non-relational databases including MongoDB, RethinkDB and Espresso. Leader-based replication is also used in distributed message brokers such as Kafka and RabbitMQ.



Synchronous Versus Asynchronous Replication

- In **synchronous** replication, the leader waits until followers have confirmed that it received the write before reporting success to the user and before making the write visible to other clients.
- In **asynchronous** replication, the leader sends the message but doesn't wait for a response from the followers.



Leader-based replication with one synchronous and one asynchronous follower

Follower Failure: Catch-up recovery

- On its local disk, each follower keeps a log of the data changes it has received from the leader.
- If a follower crashes and is restarted, or if the network between the leader and the follower is temporarily interrupted, the follower can recover quite easily from its log
- Follower knows the last transaction that was processed before the fault occurred. Thus, the follower can connect to the leader and request all the data changes that occurred during the time when the follower was disconnected.
- When the follower has applied these changes, it has caught up to the leader and can continue receiving a stream of data changes as before

Leader failure: Failover

- One of the followers is to be promoted to be the new leader
- Clients need to be reconfigured to send their writes to the new leader and the other followers need to start consuming data changes from the new leader.
- Failover can happen manually (an administrator is notified that the leader has failed and takes the necessary steps to make a new leader) or automatically
- Steps followed in an automatic failover process:
    1. Determining that the leader has failed.
    2. Choosing a new leader
    3. Reconfiguring the system to use the new leader

CLOUD COMPUTING

Replication - Implementation

1. **Statement based replication -** The leader logs every write request that it executes and sends that statement log to its followers.
2. **Write-ahead log (WAL) shipping -** The log is an append-only sequence of bytes containing all writes to the database. The leader writes the log to disk and sends it across the network to its followers. When the follower processes this log, it builds a copy of the exact same data structures as found on the leader.
3. **Logical (row-based) log replication -** Uses different log formats for replication and for the storage engine. The replication log is called a **logical log**. 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 -** A trigger lets users register custom application code that is automatically executed when a data change (write transaction) occurs in a database system. The trigger has the opportunity to log this change into a separate table, from which it can be read by an external process. That external process can then apply any necessary application logic and replicate the data change to another system.

CLOUD COMPUTING

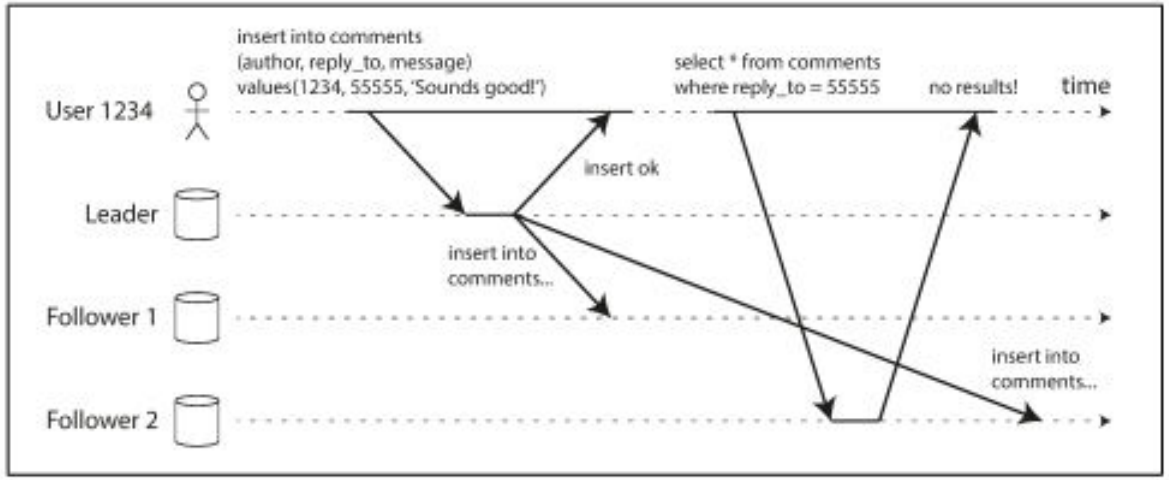
Replication Lag

- 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: if you run the same query on the leader and a follower at the same time, you may get different results, because not all writes would have been reflected in the follower.
- This inconsistency is just temporary—if there are no writes to the database in a while, the followers will eventually catch up and become consistent with the leader. This effect is known as **eventual consistency**
- In normal operation, the delay between a write happening on the leader and the same being reflected on a follower is known as the **replication lag.** This may be only a fraction of a second and not noticeable in practice
- When the lag is large, the inconsistencies it introduces are not just a theoretical issue but a real problem for applications.

CLOUD COMPUTING

Replication Lag

Reading Your Own Writes



- Read-after-write consistency, also known as read-your-writes consistency
- This is a guarantee that if the user re-reads the data, they will always see any updates they submitted themselves.
- It makes no promises about other users: other users' updates may not be visible until some later time.

Replication Lag

Solutions:

- A simple rule: always read critical data from the leader and rest from a follower (negates the benefit of read scaling)
- Monitor the replication lag on followers and prevent queries on any follower with significant lag behind the leader.
- The client can remember the timestamp of its most recent write—then the system can ensure that the replica serving any reads for that user reflects updates at least until that timestamp
- Monotonic reads - make sure that each user always makes their reads from the same replica
- Consistent prefix reads - if a sequence of writes happen in a certain order, then anyone reading those writes should see them appear in the same order

THANK YOU

Suresh Jamadagni

Department of Computer Science and Engineering

CLOUD COMPUTING

Distributed Storage

Suresh Jamadagni

Department of Computer Science and Engineering

CLOUD COMPUTING

Distributed Storage – Multi Leader Replication

Suresh Jamadagni

Department of Computer Science and Engineering

CLOUD COMPUTING

Multi-Leader Replication

- Leader-based replication has one major downside: there is only one leader and all writes must go through it. If an application cannot connect to the leader for any reason, you can't write to the database.
- A natural extension of the leader-based replication model is to allow more than one node to accept writes
- 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.

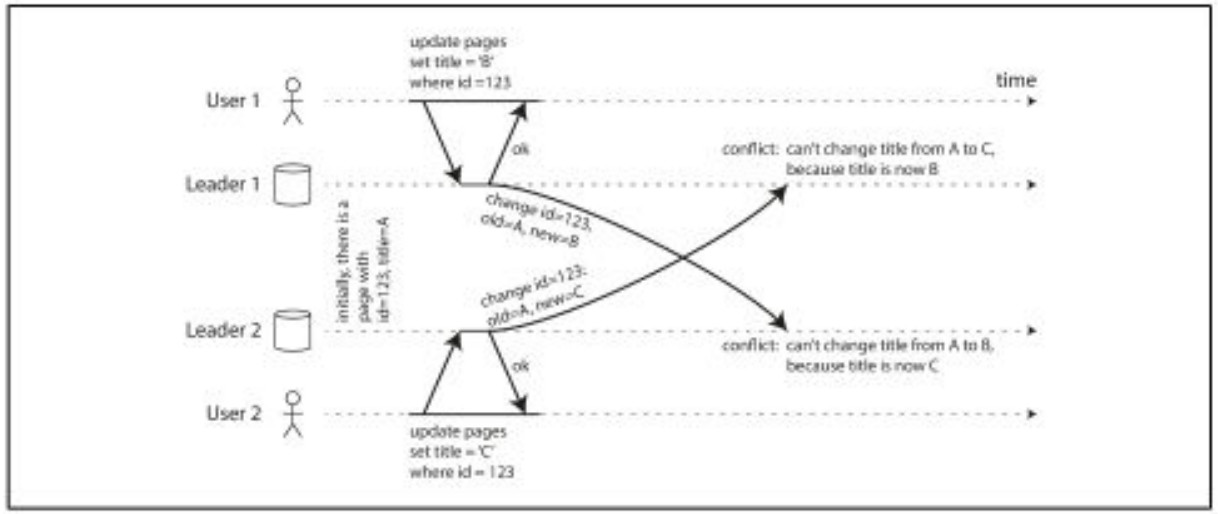
Use cases:

- **Multi-datacenter operation** - can have a leader in each datacenter. Within each datacenter, regular leader–follower replication is used; between datacenters, each datacenter's leader replicates its changes to the leaders in other datacenters.
- **Clients with offline operation**
- **Collaborative editing**

CLOUD COMPUTING

Multi-Leader Replication Challenges

Handling Write Conflicts



- **Conflict avoidance** - ensure that all writes for a particular record go through the same leader
- **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 throw away the other writes.
- **Custom conflict resolution logic -** Write conflict resolution logic in application code. That code may be executed on write or on read

CLOUD COMPUTING

Single-Leader vs. Multi-Leader Replication

Performance

- In a single-leader configuration, every write must go over the internet to the datacenter with the leader. This can add significant latency to writes and might contravene the purpose of having multiple datacenters in the first place.
- In a multi-leader configuration, every write can be processed in the local datacenter and is replicated asynchronously to the other datacenters. Thus, the interdatacenter network delay is hidden from users, which means the perceived performance may be better.

Tolerance of datacenter outages

- In a single-leader configuration, if the datacenter with the leader fails, failover can promote a follower in another datacenter to be leader.
- In a multi-leader configuration, each datacenter can continue operating independently of the others, and replication catches up when the failed datacenter comes back online.

Single-Leader vs. Multi-Leader Replication

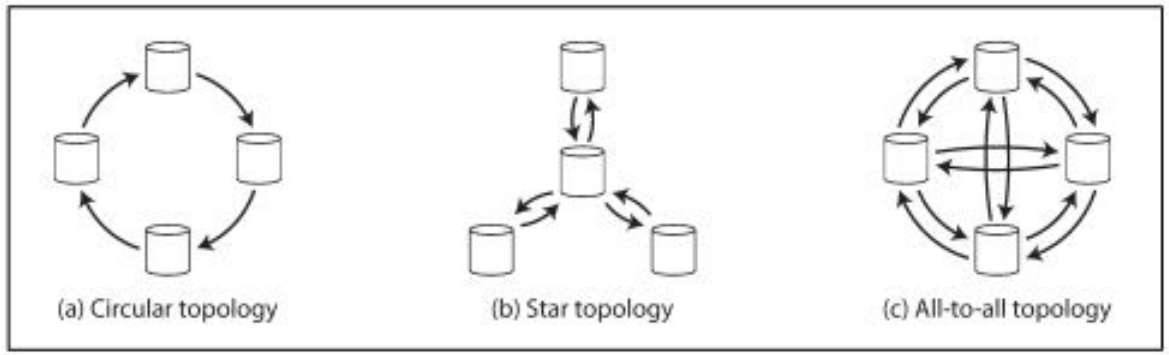
Tolerance of network problems

- Traffic between datacenters usually goes over the public internet, which may be less reliable than the local network within a datacenter.
- A single-leader configuration is very sensitive to problems in this inter-datacenter link, because writes are made synchronously over this link.
- A multi-leader configuration with asynchronous replication can usually tolerate network problems better: a temporary network interruption does not prevent writes being processed.

CLOUD COMPUTING

Multi-Leader Replication Topologies

- A replication topology describes the communication paths along which writes are propagated from one node to another.



- **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 of the other nodes.
- **All-to-all topology** - allows messages to travel along different paths, avoiding a single point of failure.

THANK YOU

Suresh Jamadagni

Department of Computer Science and Engineering

CLOUD COMPUTING

Distributed Storage

Suresh Jamadagni

Department of Computer Science and Engineering

CLOUD COMPUTING

Distributed Storage – Leaderless Replication

Suresh Jamadagni

Department of Computer Science and Engineering

CLOUD COMPUTING

Leaderless Replication

- In some leaderless implementations, the client directly sends its writes to several replicas
- In others, a coordinator node does this on behalf of the client. However, unlike a leader database that coordinator does not enforce a particular ordering of writes.

Writing to the Database When a Node Is Down

- Consider three replicas with one of the replicas being currently unavailable.
- The client sends the write to all three replicas in parallel, and the two available replicas accept the write but the unavailable replica misses it.
- Consider the write to be successful since two out of three replicas have acknowledged the write
- The client simply ignores the fact that one of the replicas missed the write
- Suppose the unavailable node comes back online and clients start reading from it. Any writes that happened while the node was down will be missing from that node. Thus, stale (outdated) values may be read from that node as response.

Leaderless Replication

Writing to the Database When a Node Is Down

- To solve this problem, read requests are also sent to several nodes in parallel.
- The client may get different responses from different nodes; i.e., the up-to-date value from one node and a stale value from another.
- Version numbers are used to determine which value is newer

CLOUD COMPUTING

Leaderless Replication

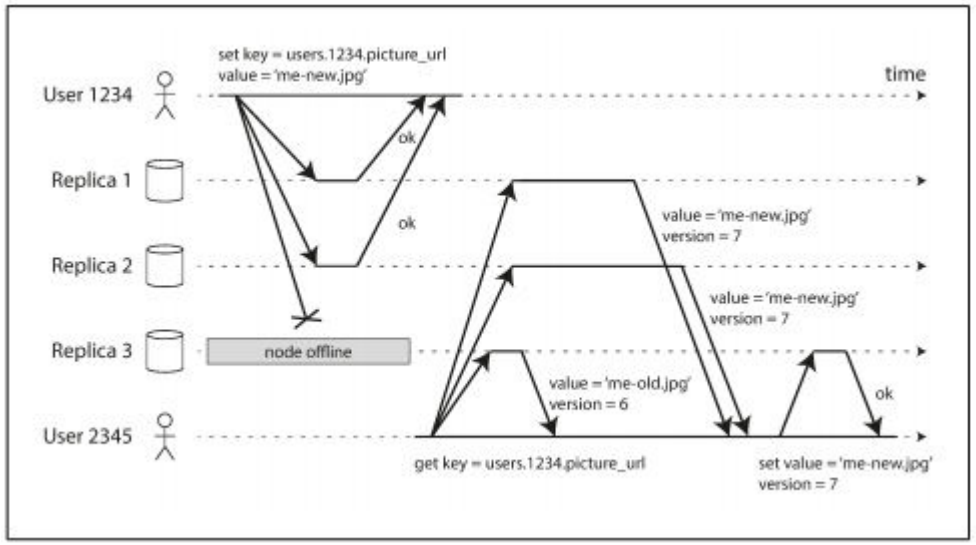
How does a node catch up on the writes that it missed?

Read repair and anti-entropy

- When a client makes a read from several nodes in parallel, it can detect any stale responses. User 2345 gets a version 6 value from replica 3 and a version 7 value from replicas 1 and 2. The client sees that replica 3 has a stale value and writes the newer value back to that replica.

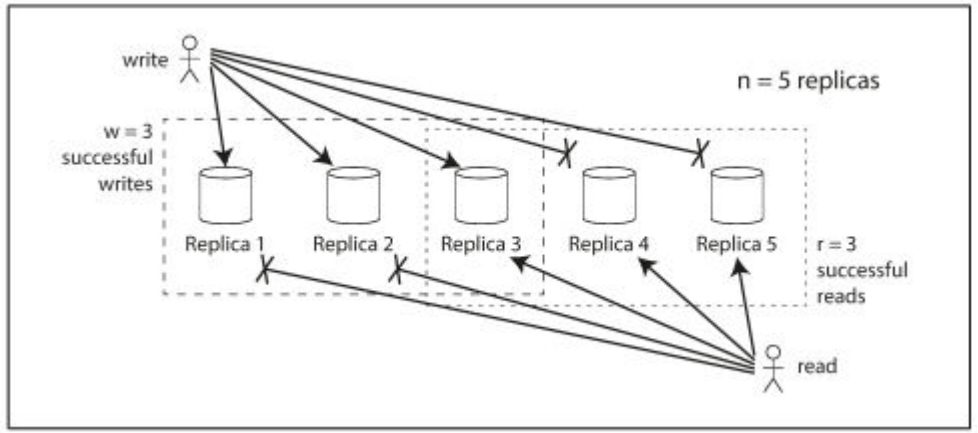
Anti-entropy process

- Some datastores have a background process that constantly looks for differences in the data between replicas and copies any missing data from one replica to another



Quorums for reading and writing

- If there are n replicas, every write must be confirmed by **w** nodes to be considered successful, and we must query at least **r** nodes for each read. As long as **w + r > n**, we expect to get an up-to-date value when reading, because at least one of the r nodes we're reading from must be up to date
- Reads and writes that obey these r and w values are called quorum reads and writes
- A common choice is to make n an odd number (typically 3 or 5) and to set w = r = (n + 1) / 2



- With n = 5, w = 3, r = 3 we can tolerate two unavailable nodes.

Leaderless Replication

Monitoring staleness

- For leader-based replication, the database typically exposes metrics for the replication lag. By subtracting a follower's current position from the leader's current position, you can measure the amount of replication lag.
- For leaderless replication, there is no fixed order in which writes are applied, which makes monitoring more difficult

Leaderless Replication

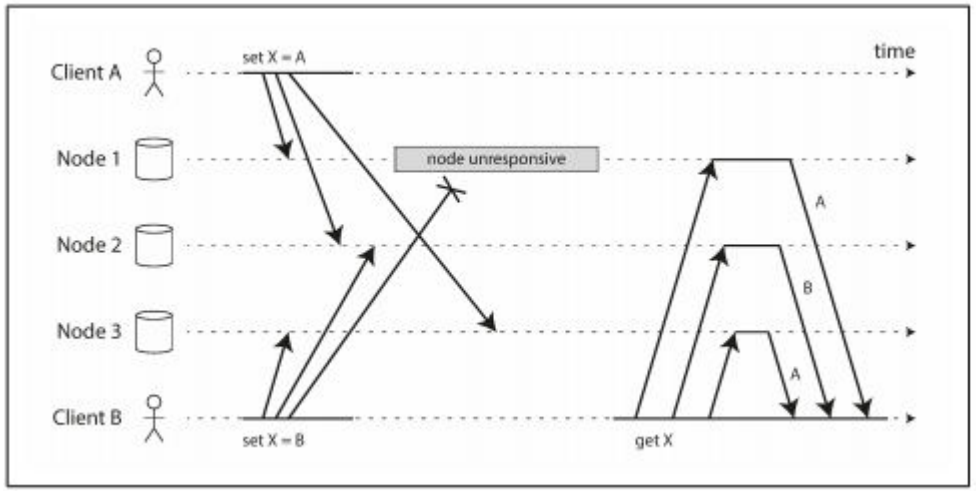
Multi-datacenter operation

- Leaderless replication is also suitable for multi-datacenter operation, since it is designed to tolerate conflicting concurrent writes, network interruptions and latency spikes.
- The number of replicas **n** includes nodes in all datacenters, and the number of replicas you want to have in each datacenter can be configured.
- Each write from a client is sent to all replicas, regardless of datacenter, but the client usually only waits for acknowledgment from a quorum of nodes within its local datacenter so that it is unaffected by delays and interruptions on the cross-datacenter link.
- The higher-latency writes to other datacenters are often configured to happen asynchronously

Leaderless Replication

Detecting Concurrent Writes

- Databases allow several clients to concurrently write to the same key, which means that conflicts will occur even if strict quorums are used
- Events may arrive in a different order at different nodes, due to variable network delays and partial failures
- If each node simply overwrote the value for a key whenever it received a write request from a client, the nodes would become permanently inconsistent



Detecting Concurrent Writes

- In order to become eventually consistent, the replicas should converge toward the same value
- **Last write wins** (discarding concurrent writes) - One approach for achieving eventual convergence is to declare that each replica need only store the most “recent” value and allow “older” values to be overwritten and discarded.
- **The “happens-before” relationship and concurrency** - How do we decide whether two operations are concurrent or not?
  - An operation A happens before another operation B if B knows about A, or depends on A, or builds upon A in some way. Whether one operation happens before another operation is the key to defining what concurrency means.
  - Server can determine whether two operations are concurrent by looking at the version numbers

CLOUD COMPUTING

Leaderless Replication

Algorithm for determining whether two operations are concurrent by looking at the version numbers

- The server maintains a version number for every key, increments the version number every time that key is written, and stores the new version number along with the value written.
- When a client reads a key, the server returns all values that have not been overwritten, as well as the latest version number. A client must read a key before writing.
- When a client writes a key, it must include the version number from the prior read, and it must merge together all values that it received in the prior read. (The response from a write request can be like a read, returning all current values, which allows to chain several writes.)
- When the server receives a write with a particular version number, it can overwrite all values with that version number or below (since it knows that they have been merged into the new value), but it must keep all values with a higher version number (because those values are concurrent with the incoming write).

CLOUD COMPUTING

Leaderless Replication

Version vectors

- In case of multiple replicas, we need to use a version number per replica as well as per key.
- Each replica increments its own version number when processing a write, and also keeps track of the version numbers it has seen from each of the other replicas. This information indicates which values to overwrite and which values to keep as siblings.
- The collection of version numbers from all the replicas is called a **version vector**
- Version vectors are sent from the database replicas to clients when values are read, and need to be sent back to the database when a value is subsequently written
- The version vector allows the database to distinguish between overwrites and concurrent writes
- The version vector structure ensures that it is safe to read from one replica and subsequently write back to another replica. Doing so, may result in siblings being created
- No data is lost as long as siblings are merged correctly.

THANK YOU

Suresh Jamadagni

Department of Computer Science and Engineering

CLOUD COMPUTING

Distributed Storage

Suresh Jamadagni
Department of Computer Science and Engineering

CLOUD COMPUTING

Distributed Storage – Consistency Models

Suresh Jamadagni

Department of Computer Science and Engineering

Consistency Models

Consistency Guarantees

- Most replicated databases provide **eventual consistency**, which means that if you stop writing to the database and wait for some **unspecified length of time**, then eventually all read requests will return the same value. That is, all replicas will eventually converge to the same value
- This is a very **weak guarantee** as it doesn't say anything about when the replicas will converge
- The edge cases of eventual consistency only become apparent when there is a fault in the system or at high concurrency.

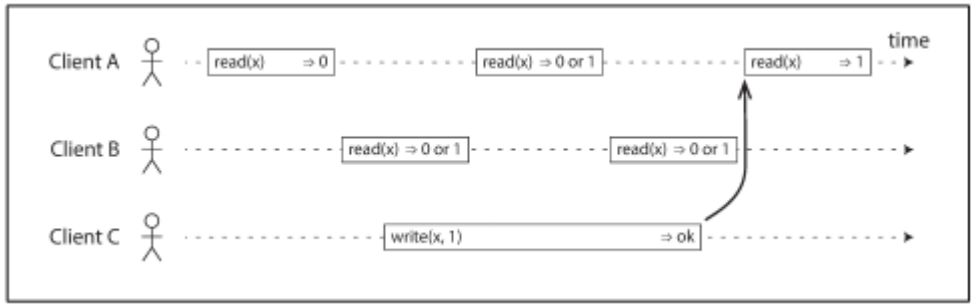
Linearizability

- 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.
- With this guarantee, even though there may be multiple replicas in reality, the application does not need to worry about them.
- Also known as atomic consistency, strong consistency, immediate consistency or external consistency
- 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

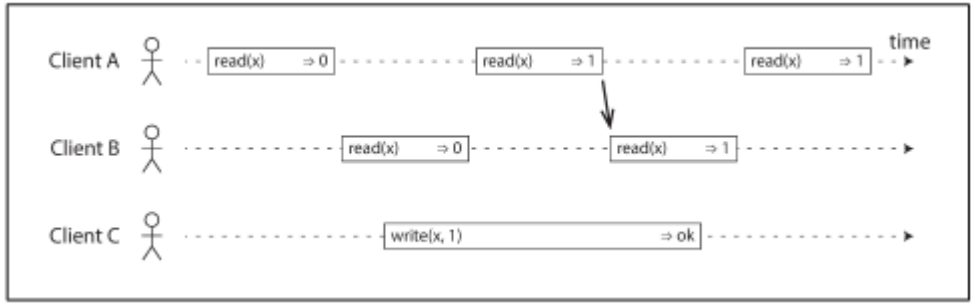
CLOUD COMPUTING

Linearizability

- If a read request is concurrent with a write request, it may return either the old or the new value



- After any one read has returned the new value, all following reads (on the same or other clients) must also return the new value.



Linearizability

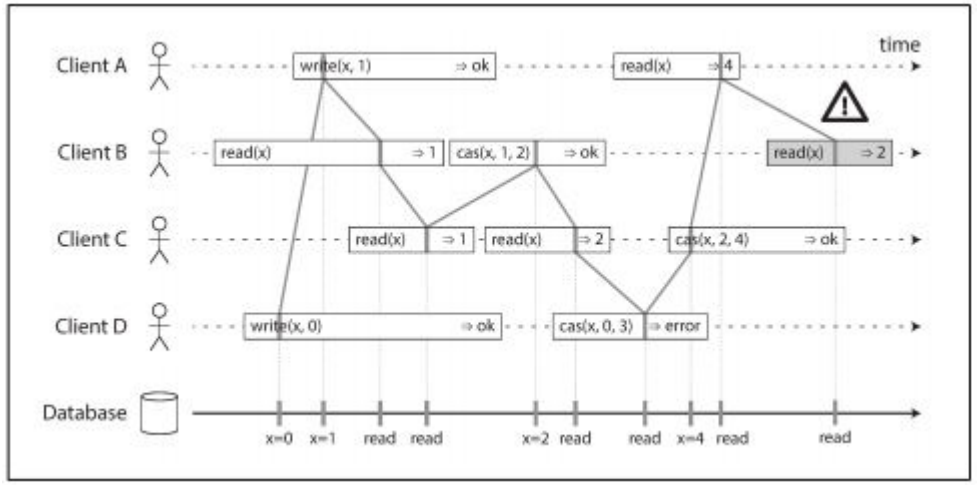
- Add a third type of operation besides read and write

- **cas(x, v_{old}, v_{new}) ⇒ r** means the client requested an atomic compare-and-set operation . If the current value of the register x equals v_{old}, it should be atomically set to v_{new}. If $x \neq v_{old}$ then the operation should leave the register unchanged and return an error. r is the database's response (ok or error).

CLOUD COMPUTING

Linearizability

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

Linearizability

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)

THANK YOU

Suresh Jamadagni

Department of Computer Science and Engineering

CLOUD COMPUTING

Distributed Storage

Suresh Jamadagni
Department of Computer Science and Engineering

CLOUD COMPUTING

Distributed Storage – CAP Theorem

Suresh Jamadagni

Department of Computer Science and Engineering

CAP Theorem

- If an application **requires linearizability**, and some replicas are disconnected from the other replicas due to a network problem, then some replicas cannot process requests while they are disconnected. They must either wait until the network problem is fixed, or return an error
- If an application **does not require linearizability**, then each replica can process requests independently even if it is disconnected from other replicas. In this case, the application can remain available in the face of a network problem, but its behavior is not linearizable.
- Therefore, applications that don't require linearizability can be more tolerant of network problems. This insight is popularly known as the **CAP theorem**

CLOUD COMPUTING

CAP Theorem

The CAP theorem states that any networked shared-data system can have at most two of three desirable properties:

1. **Consistency** (C) requires that all reads initiated after a successful write return the same and latest value at any given logical time.
2. **Availability** (A) requires that every node (not in failed state) always execute queries. Let's say we have "n" servers serving our application. To ensure better availability we would add an additional "x" servers.
3. **Partition Tolerance** (P) requires that a system be able to re-route a communication when there are temporary breaks or failures in the network. The goal is to maintain synchronization among the involved nodes.

Tradeoffs Between Requirements

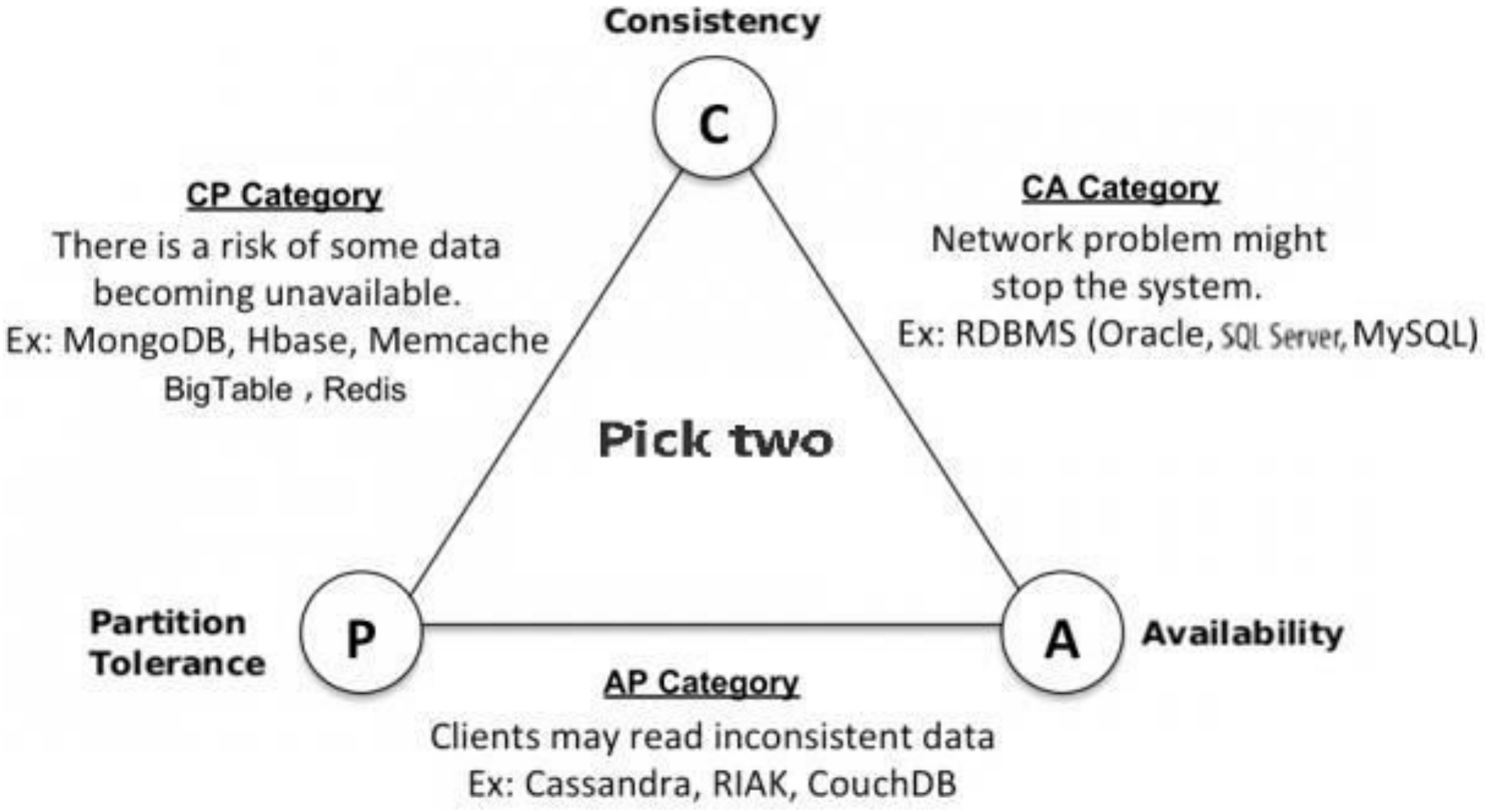
1. ***Available and Partition-Tolerant***: Say you have two nodes and the link between the two is severed. Since both nodes are up, you can design the system to accept requests on each of the nodes, which will make the system available despite the network being partitioned. However, each node will issue its own results, so by providing high availability and partition tolerance you'll compromise consistency.

2. ***Consistent and Partition-Tolerant***: Say you have three nodes and one node loses its link with the other two. You can create a rule that a result will be returned only when a majority of nodes agree. So, despite having a partition, the system will return a consistent result. However, since the separated node won't be able to reach consensus it won't be available even though it's up.

3. Finally, a system can be both ***consistent and available***, but it may have to block on a partition.

CLOUD COMPUTING

CAP Theorem



CAP Theorem

- The “Pick Two” expression of CAP opened the minds of designers to a wider range of systems and tradeoffs
- The modern CAP goal is to maximize combinations of consistency and availability that make sense for the specific application. Such an approach incorporates plans for operation during a partition unavailability and for recovery afterward, thus helping designers think about CAP beyond its historically perceived limitations.

THANK YOU

Suresh Jamadagni

Department of Computer Science and Engineering

CLOUD COMPUTING

Distributed Storage

K.S. Srinivas
Suresh Jamadagni
Venkatesh Prasad

Department of Computer Science and Engineering

CLOUD COMPUTING

Distributed Storage

K. Srinivas
Suresh Jamadagni
Venkatesh Prasad

Department of Computer Science and Engineering

CLOUD COMPUTING

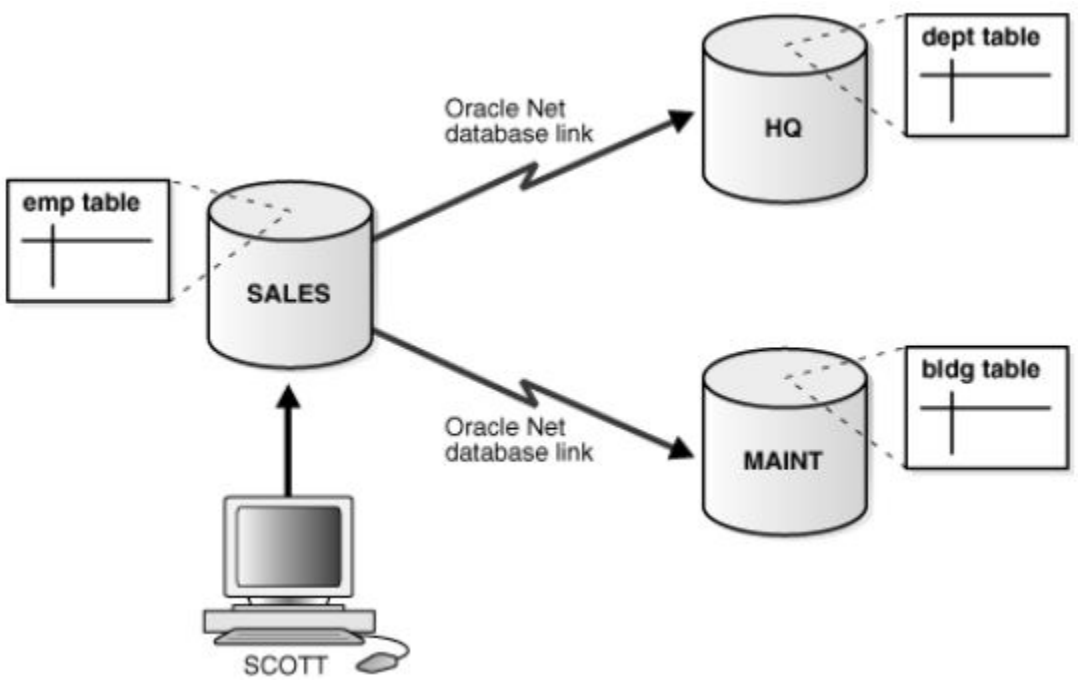
Distributed Transactions

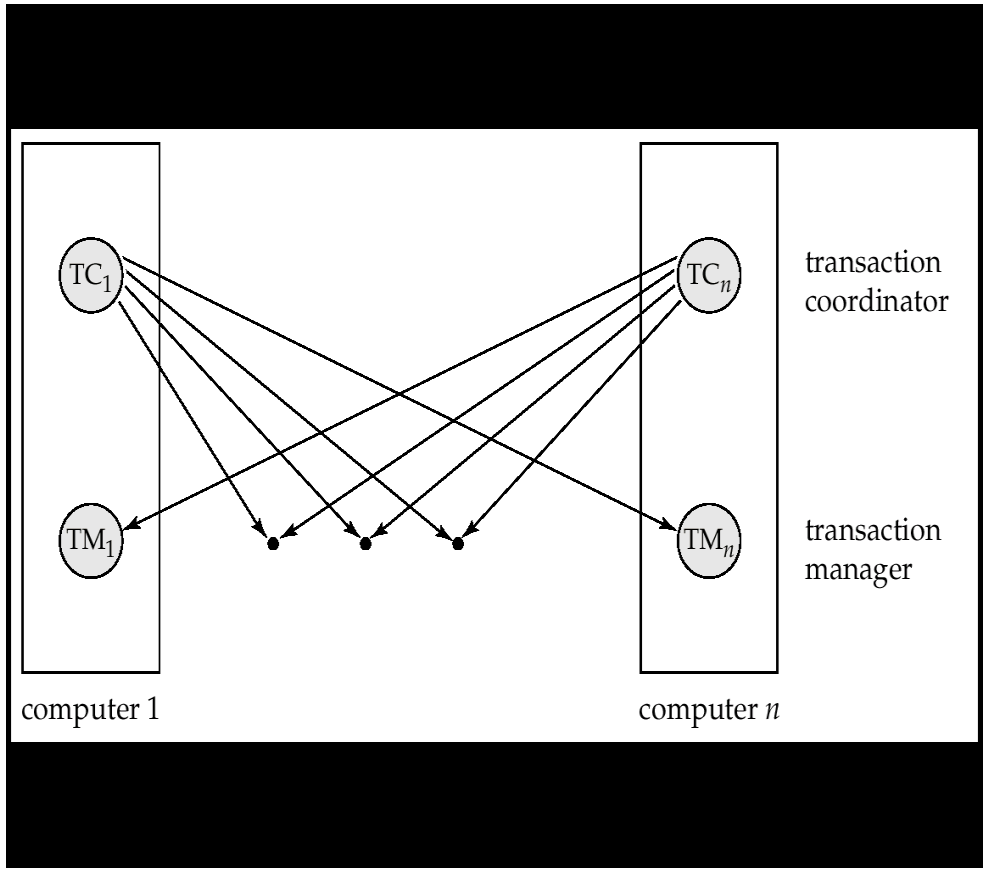
Transaction may access data at several sites.

- Each site has a local transaction manager responsible for:
  - Maintaining a log for recovery purposes
  - Participating in coordinating the concurrent execution of the transactions executing at that site.
- Each site has a transaction coordinator, which is responsible for:
  - Starting the execution of transactions that originate at the site.
  - Distributing sub transactions at appropriate sites for execution.
  - Coordinating the termination of each transaction that originates at the site, which may result in the transaction being committed at all sites or aborted at all sites.

CLOUD COMPUTING

Distributed Transaction System Architecture





Concurrency Control

- All concurrency mechanisms must preserve data consistency and complete each atomic action in finite time
- Important capabilities are
  a) Be resilient to site and communication link failures.
  b) Allow parallelism to enhance performance requirements.
  c) Incur optimal cost and optimize communication delays
  d) Place constraints on atomic action.

Commit Protocols

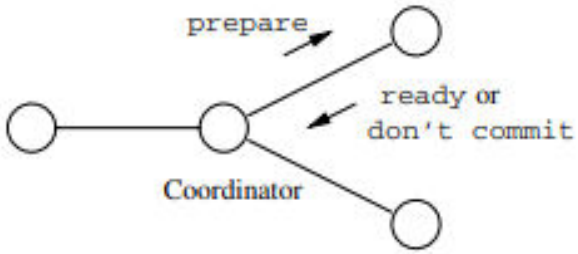
- Commit protocols are used to ensure atomicity across sites
  - A transaction which executes at multiple sites must either be committed at all the sites, or aborted at all the sites.
  - Not acceptable to have a transaction committed at one site and aborted at another.

- The two-phase commit (2 PC) protocol is widely used.

CLOUD COMPUTING

Two Phase Commit Protocol - Phase 1

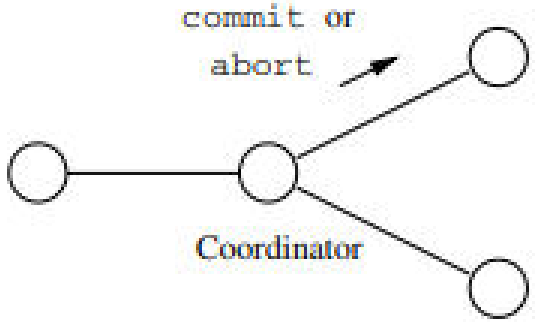
1. The coordinator places a log record **<Prepare T>** on the log at its site.
2. The coordinator sends to each component's site the message prepare T.
3. Each site receiving the message prepare T decides whether to commit or abort its component of T.
4. If a site wants to commit its component, it must enter a state called pre-committed. Once in the pre-committed state, the site cannot abort its component of T without a directive to do so from the coordinator
   a. Perform whatever steps necessary to be sure the local component of T will not have to abort
   b. Place the record **<Ready T>** on the local log and flush the log to disk.
   c. Send ready T message to the coordinator
5. If the site wants to abort its component of T, then it logs the record **<Don't Commit T>** and sends the message don't commit T to the coordinator



CLOUD COMPUTING

Two Phase Commit Protocol - Phase 2

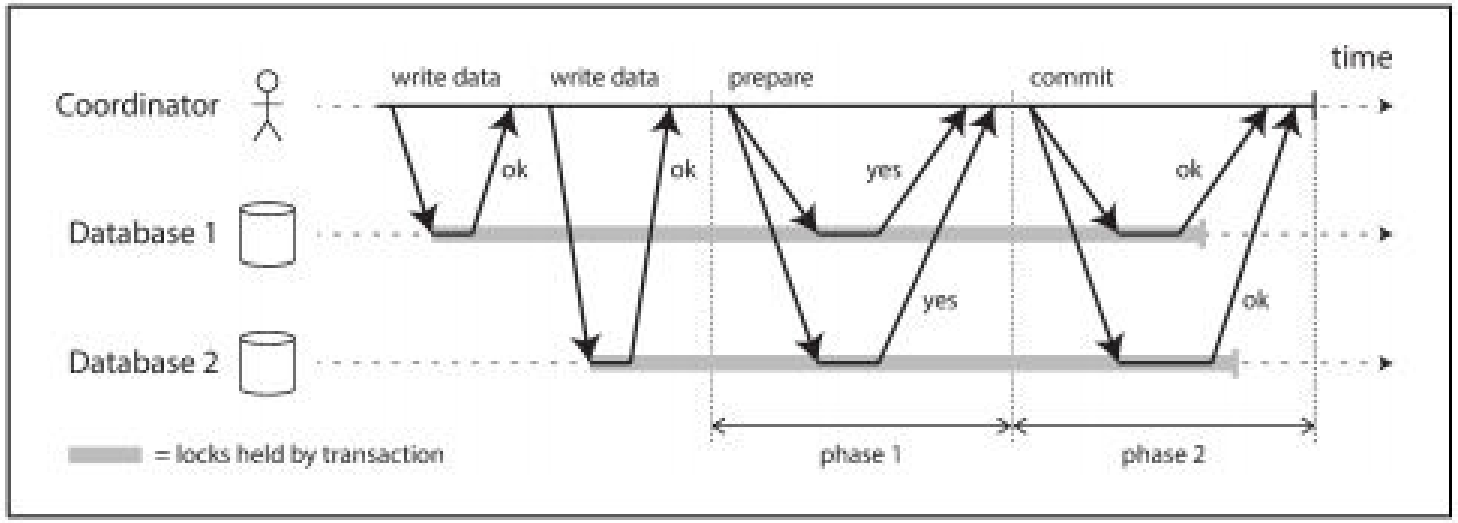
1. If the coordinator has received ready T from all components of T, then it decides to commit T. The coordinator logs **<Commit T>** at its site and then sends message commit T to all sites involved in T
2. If the coordinator has received don't commit T from one or more sites, it logs **<Abort T>** at its site and then sends abort T messages to all sites involved in T
3. If a site receives a commit T message, it commits the component of T at that site, logging **<Commit T>** as it does.
4. If a site receives the message abort T, it aborts T and writes the log record **<Abort T>**



CLOUD COMPUTING

Two Phase Commit Protocol

A successful execution of two-phase commit



THANK YOU

K.S. Srinivas
Suresh Jamadagni
Venkatesh Prasad

Department of Computer Science and Engineering