Cloud Storage

Introduction to Data Storage in the Cloud

Cloud systems utilize various forms of data storage:

- **Distributed file systems:** Examples include **Google File System** (GFS) and **Hadoop Distributed File System** (HDFS). These often build on concepts from Network File Systems and Parallel File Systems.
- **SQL and NoSQL databases**
- **Key-value storage systems:** Examples include **BigTable** and **Dynamo**.

Common goals for these cloud storage solutions include:

- Massive scaling on demand
- High availability
- Simplified application development and deployment

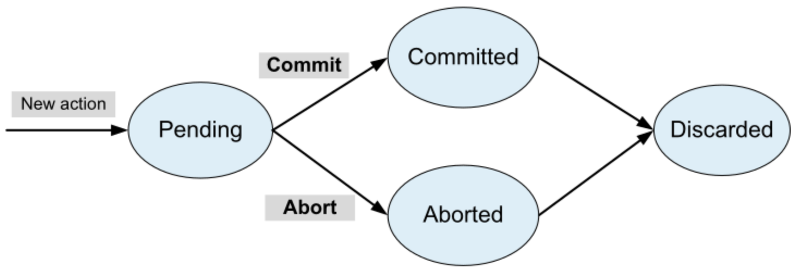
Atomic Transaction

It is a multi-step operation (transaction) that **should complete entirely without any interruption** (atomic).

- **Atomicity Requires:**
  - **Hardware (HW) support:** Non-interruptible ISA (Instruction Set Architecture) operations.
    - **Test-and-set:** Writes to a memory location and returns its old content as a non-interruptible operation.
    - **Compare-and-swap:** Compares a memory location's content to a given value and, only if they match, modifies the content to a new given value.
  - **Mechanisms for shared resource access** and critical section creation (e.g., Locks, semaphores, monitors).

Types of Atomicity

All-or-nothing atomicity



All-or-nothing means that either the entire atomic action is **carried out**, **or** the system is **left in the same state** it was before the atomic action was attempted.

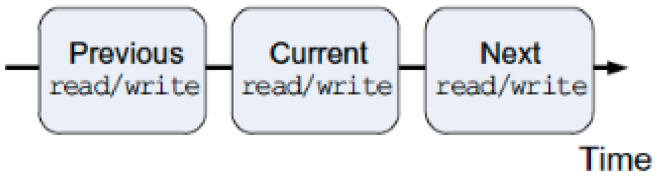
Consists of two (main) phases:

1. **Pre-commit phase: Preparatory actions** that can be **undone** (e.g., resource allocation, fetching a page, memory allocation).
2. **Commit-point:** The transition between pre and post phases.
3. **Post-commit phase:** Irreversible actions (e.g., altering the only copy of an object).

Maintaining a history of changes and a log of actions allows for dealing with system failures and ensuring consistency.

A **transaction lifecycle** goes from **Pending** to either **Committed**, so changes are applied (via Commit), or **Aborted**, so changes are undone using history of changes or log of action (via Abort), both leading to a Discarded state, this means the end of the transaction lifecycle.

Before-or-after atomicity:



From the point of view of an external observer, the effect of multiple actions is as though these actions have occurred one after another, in some order.

- *Example*: the concurrent execution of operations within a transaction should produce the same result as if the transaction were executed serially.

Storage Models & Desired Properties

- **Physical Storage:** Can be local disk, removable USB disk, or network-accessible disk; either solid-state (SSD) or magnetic (HDD).
- **Storage Model:** Describes the layout of a data structure in physical storage. Two main types:
  - **Cell** storage
  - **Journal** storage
- **Desired Properties for Storage Models:**
  - **Read/write coherence**: Ensures that the result of reading a memory cell is the same as the value from the most recent write to that specific cell. It focuses on the freshness and correctness of the data returned by a single read operation relative to previous write operations on that same location
  - **Before-or-after atomicity**: Ensures that concurrent operations appear to execute in some serial order, without their individual steps interleaving in a way that produces an inconsistent state. Each operation is treated as an indivisible unit.
  - **All-or-nothing atomicity**: Ensure that an operation is completely carried out or aborted.

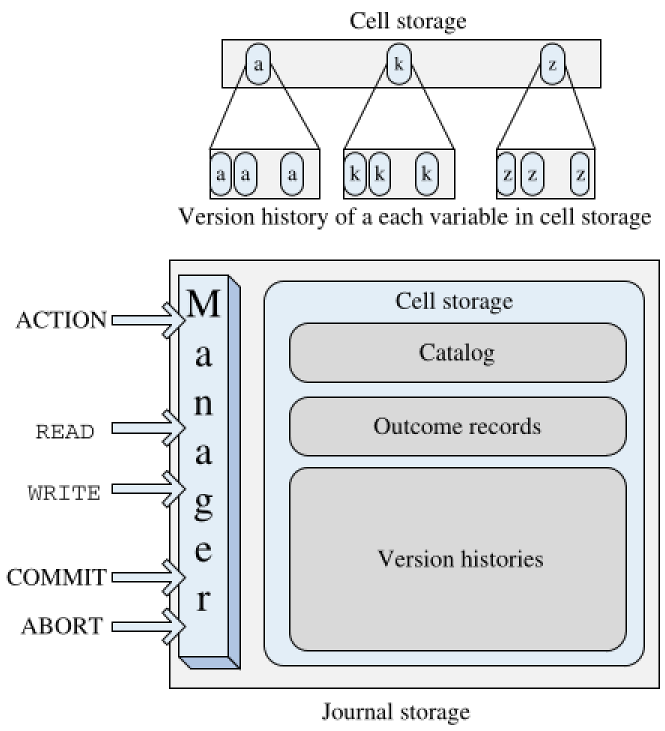
Cell Storage Model

- **Assumptions:**
  - Storage is composed of **cells of the same size**.
  - **Exactly an object** into a cell.

The unit for read/write operations is a sector or block, reflecting the physical organization of common storage media (e.g., primary memory (RAM) as an array of cells, secondary storage (DISK, SSD) in sectors/blocks).

- **Guarantees:**
    - **Read/write coherence**
    - **Before-or-after** atomicity.
- **Does not Guarantee:**
    - **All-or-nothing** atomicity: Once the content of a cell is changed, there is no way to abort the action and restore the original content of that cell via the cell storage model itself.

Journal Storage Model



- A model for storing objects like records consisting of multiple fields.
- Comprises a **manager** and uses **cell storage** as its underlying persistent store.
- The entire **history of a variable** (not only its current value) is **maintained** in the cell storage (often as version histories).

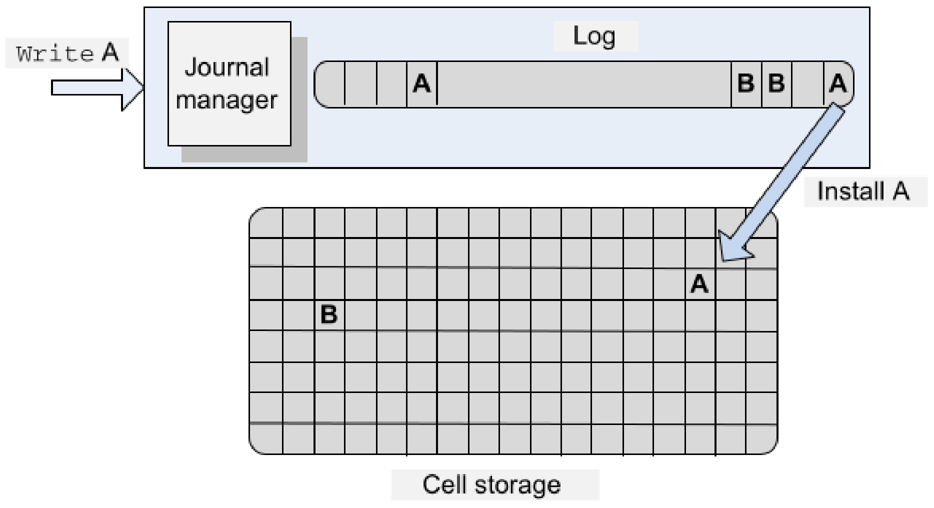
User Interaction

Users interact with the journal manager, not directly with the cell storage. User requests include:

- **ACTION**: Start a new action
- **READ**: Read the value of a cell
- **WRITE**: Write the value of a cell
- **COMMIT**: Commit an action
- **ABORT**: Abort an action

The journal manager translates these user requests into commands for the cell storage, such as: read a cell, write a cell, allocate a cell, or deallocate a cell.

The Log



- Contains the **history of all variables** in the cell store. Each update is a record appended to the log.
- The log is **authoritative**, meaning that it is considered the primary and most trustworthy record of all changes, and **allows reconstruction** of the cell store.
- For an all-or-nothing action, the action is:
  - **First**, recorded in the log (in journal storage)
  - **Then**, the change is installed in the cell storage by overwriting the previous version.
- The log is always kept on **nonvolatile storage**.
- The cell storage also typically resides on nonvolatile memory but **can be held in volatile memory** for real-time access or by using a write-through cache.

Guarantees

All-or-nothing atomicity: Achieved because an action first modifies the version history (log), and only then modifies the cell store (commit step). If any part of this process fails, the **action is discarded** (aborted).

All-or-nothing atomicity **implies read/write coherence and before-or-after atomicity**, so journal storage also guarantees these.

Consensus Protocols

Consensus: The process where a number of agents (e.g., processes in a distributed system) agree on one of several proposed alternatives, aiming for a single proposed value.

Consensus protocols aim to guarantee:

- **Consistency** and **Reliability (safety)**: Ensure that all nodes in a distributed system agree on a common state or decision, crucial for mantaining data intergrity and preventing conflicting updates.

Consensus protocols cannot guarantee:

- **Progress**: Cannot ensure that the process involved in trying to reach an agreement will eventually decide a value.

Paxos

A family of protocols for reaching consensus, often based on a finite state machine approach. It is core in supporting eventual consistency in NoSQL data stores.

Basic Paxos Goal is to enable a set of processes to reach consensus on a single proposed value. Clients send requests to processes, proposing a value, and then await a response.

Paxos protocol assumption

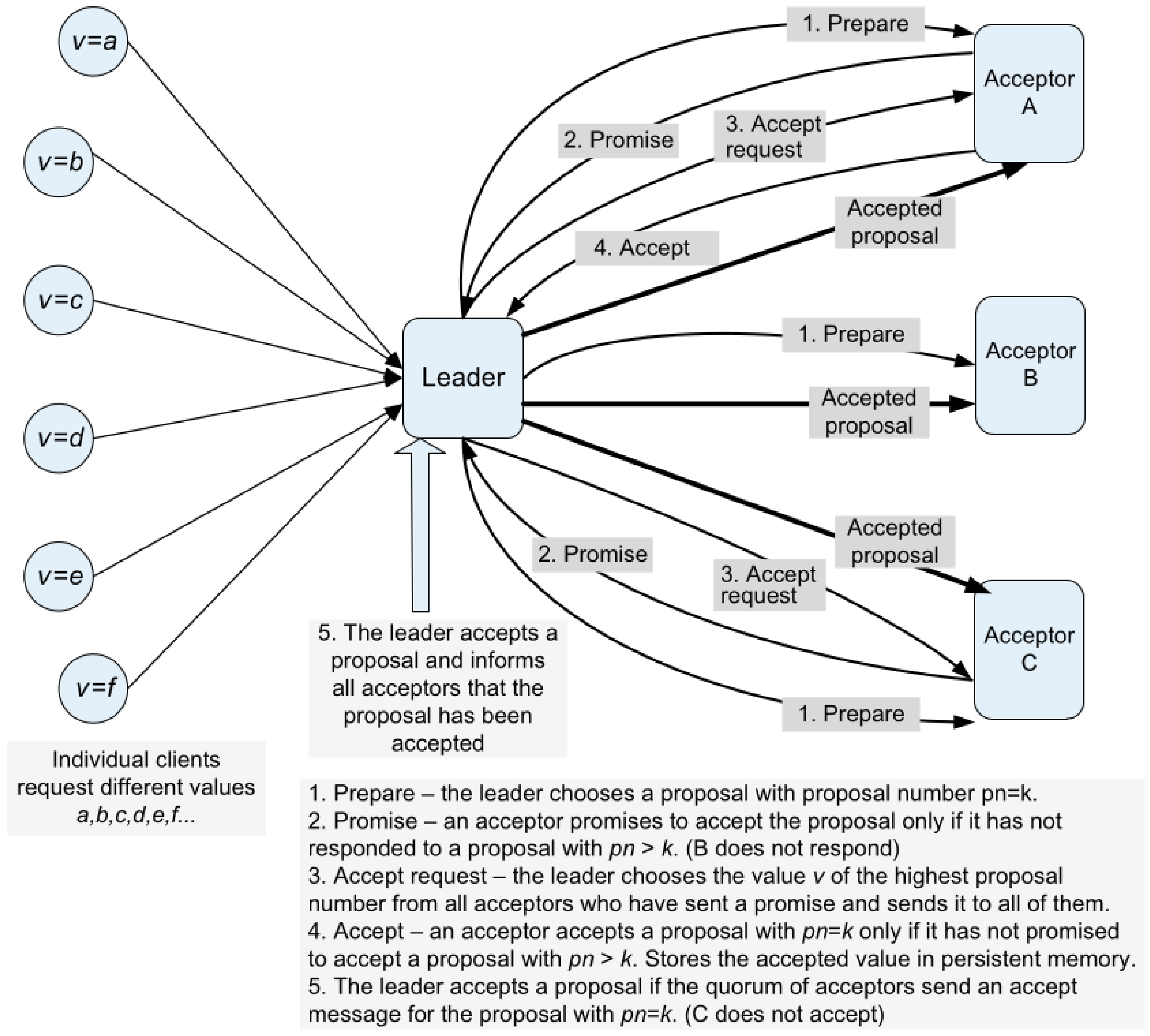
Basic Paxos protocol is based on several assumption about the processor and the network:

- Processors and the network may experience failures, but not **Byzantine failures** (where failed components can generate arbitrary, malicious data).
- **Processors:** Operate at arbitrary speeds, possess stable storage, may rejoin the protocol after failures, and can send messages to any other processor.
- **Network:** May lose, reorder, or duplicate messages; message delivery is asynchronous and can take an arbitrary amount of time.

Paxos Entities (Agents)

- **Client:** Issues a request and waits for a response.
- **Proposer:** Advocates a client's request, attempts to convince acceptors to agree on the proposed value, and acts as a coordinator to move the protocol forward, especially during conflicts.
- **Acceptor:** Acts as the fault-tolerant "memory" of the protocol.
- **Learner:** Acts as the replication factor; takes action once a request has been agreed upon.
- **Leader:** A distinguished proposer.
- Typically, a single entity in a deployment plays the roles of proposer, acceptor, and learner.
- **Quorum and Proposal:**
  - A **quorum** is a subset of all acceptors (typically a majority).
  - A **proposal** consists of a pair: a unique proposal number (pn) and a proposed value (v). Multiple proposals may suggest the same value.
  - A value is **chosen** if a simple majority of acceptors have accepted it.

Basic Paxos Algorithm



1. Prepare – the leader chooses a proposal with proposal number pn=k.
2. Promise – an acceptor promises to accept the proposal only if it has not responded to a proposal with *pn* > *k*. (B does not respond)
3. Accept request – the leader chooses the value *v* of the highest proposal number from all acceptors who have sent a promise and sends it to all of them.
4. Accept – an acceptor accepts a proposal with *pn*=*k* only if it has not promised to accept a proposal with *pn* > *k*. Stores the accepted value in persistent memory.
5. The leader accepts a proposal if the quorum of acceptors send an accept message for the proposal with *pn*=*k*. (C does not accept)

Phase 1: Proposal Preparation and Promise

1. **Proposal Preparation (Proposer):** A proposer (**leader**) selects a proposal number `pn=k` and sends a `prepare` message to a majority of acceptors.

   This message requests that acceptors:

   - **Do not accept** any proposal with a number less than `k`
   - Respond with the highest-numbered proposal (with `pn < k`) they have **already accepted**, if any.

2. **Proposal Promise (Acceptor):** An acceptor remembers the highest proposal number it has ever accepted and the highest proposal number it has ever responded to in a `prepare` request. It can promise to accept proposal `k` if and only if it has not already responded to a `prepare` message for a proposal with a number greater than `k`. If it has already replied to a higher proposal, it should not reply to the current one.

   - **Lost messages** are treated as if the acceptor chose not to respond.

Phase 2: Accept Request and Accept

1. **Accept Request (Proposer):** If the proposer receives promises from a majority of acceptors, it chooses a value `v` for its proposal as follow:

    - If any acceptors reported **previously accepted proposals**, the proposer must choose the value `v` associated with the highest proposal number among those responses.
    - If **no proposals were previously accepted** by the responding acceptors, the proposer can choose an arbitrary value (typically the one it originally wanted to propose).

    The proposer then sends an `accept request` message `(pn=k, v)` to a quorum of acceptors.

2. **Accept (Acceptor):** If an acceptor receives an `accept request` message for proposal `pn=k`, it must accept it if and only if it has not already promised (in Phase 1) to only consider proposals with a number greater than `k`.

    - If it **accepts**, it registers the value `v` (often in persistent storage) and sends an `accept` message to the proposer and to every learner.
    - If it **does not accept** the proposal, it should ignore the request.

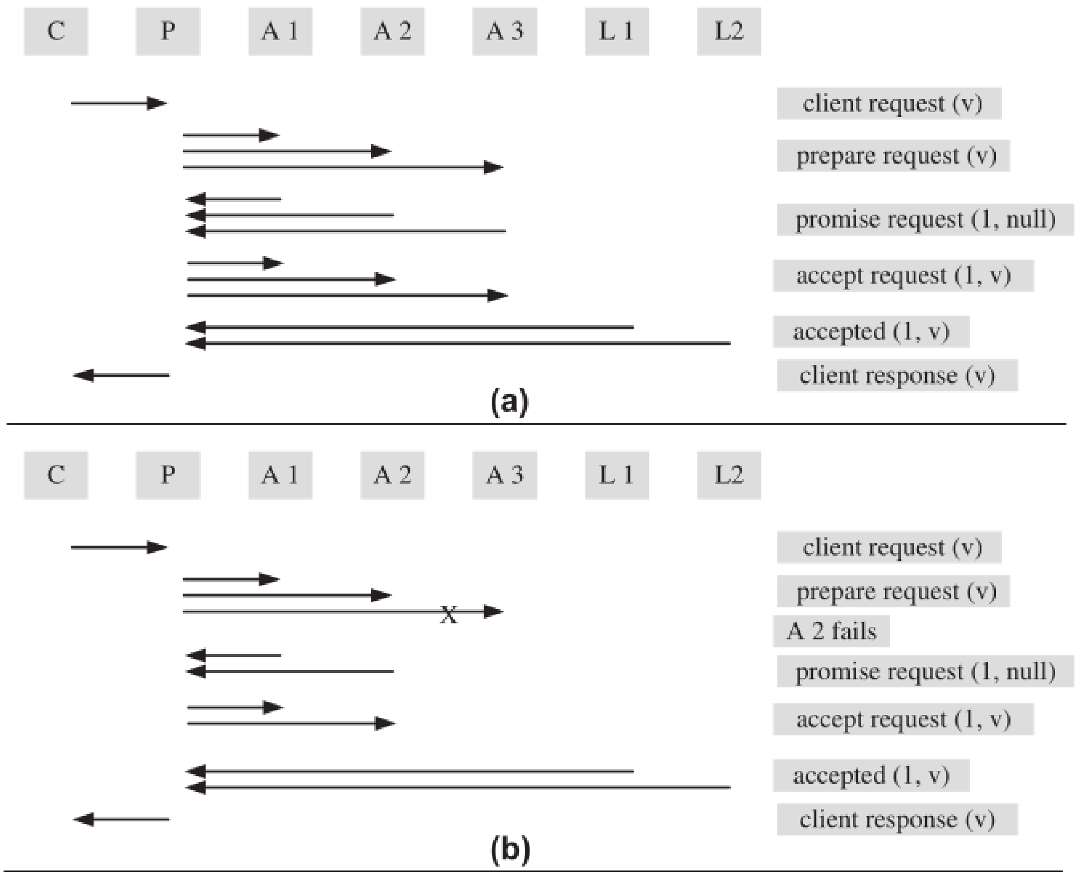


FIGURE 2.16

The basic Paxos with three actors: proposer (P), three acceptors (A1, A2, A3), and two learners (L1, L2). The client (C) sends a request to one of the actors playing the role of a proposer. The entities involved are (a) successful first round when there are no failures and (b) successful first round of Paxos when an acceptor fails.

Algorithm Properties (Correctness)

- A proposal number (`pn`) is **unique**.

- Any two **quorums** (sets of acceptors) have **at least one acceptor in common**.
- The value sent out in an `accept request` (Phase 2) by a proposer is the value of the **highest-numbered proposal** reported by acceptors in their Phase 1 `promise` responses (or the proposer's original value if no prior proposals were reported).
- **Message Flow and Failures:** The protocol is **designed to reach consensus** even if some messages are lost or some (non-Byzantine) nodes fail, as long as a proposer can communicate with a quorum of acceptors.

Google File System (GFS) (PAPER DA VEDERE)

A **distributed file system** designed to provide petabytes of storage using thousands of storage systems built from inexpensive commodity components.

- **Example**: Standard cheap and inexpensive HDD as opposed to expensive and high-end storage hardware.

Design Principles

Given the usage of many standard storage system, main concern was about **High Reliability** and **Fault Tollerance** against various failures (hardware, software, human error).

Careful **analysis of typical file characteristics** and **access models** in Google's environment led to the derivation of the most important aspects of GFS design.

Characteristic

- Files are very large (GBs to hundreds of TBs).
- Most common operations are dominated by appends rather than random writes.
- Sequential reads are common.
- High sustained throughput is more critical than low latency for individual operations, as data is often processed in bulk.
- A relaxed consistency model is used to simplify implementation without overburdening application developers.

Design

Above characteristic led to following design decisions:

- **File Structure:**
    - Files are divided into fixed-size segments called **chunks** (typically 64MB, much larger than traditional file system blocks).
    - Chunks are stored as files on standard Linux file systems on **chunk servers**.
    - Each chunk is **replicated on multiple chunk servers** (default is 3, configurable) for fault tolerance.
    - **Advantages of large chunk size:**
        - **Optimizes performance** for large files
        - **Reduces metadata** maintained by the master

- Increases **likelihood of multiple operations** targeting the same chunk (reducing network overhead by maintaining persistent connections)
- **Reduces disk fragmentation**: small files or the last chunk of a large file may only partially fill a chunk.

- **Architecture:**

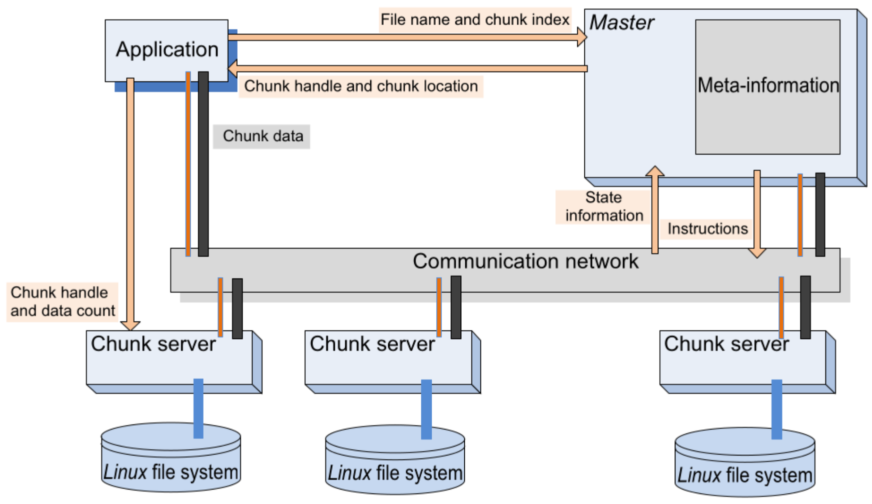


FIGURE 8.7

The architecture of a GFS cluster. The *master* maintains state information about all system components; it controls a number of *chunk servers*. A chunk server runs under *Linux*; it uses metadata provided by the *master* to communicate directly with the application. The data flow is decoupled from the control flow. The data and the control paths are shown separately, data paths with thick lines and control paths with thin lines. Arrows show the flow of control among the application, the *master*, and the chunk servers.

  - A single **Master** node:
    - Controls a large number of **Chunk Servers**.
    - Maintains all **filesystem metadata**, including filenames, access control information, the mapping from files to constituent chunks, the **locations of all replicas for every chunk**, and the state of individual chunk servers.
    - **Chunk locations** are stored only in the master's memory for fast access and are **updated at system startup** or **when a new chunk server joins**, this allows the master to have up-to-date info. about location of the chunks.
  - **Chunk Servers:** Store chunks on their local disks and handle read/write operations on these chunks as directed by clients or the master.

- **Reliability and Recovery:**

  - The master maintains an **operation log** for all metadata changes.

    These **changes are atomic** and made visible to clients only after being recorded durably on multiple replicas.

  - In case of **master failure**, it can be recovered by replaying this operation log.

  - To **minimize recovery time**, the master periodically **checkpoints** its state, so only log records after the last checkpoint need to be replayed.

File Access and Write Operations:

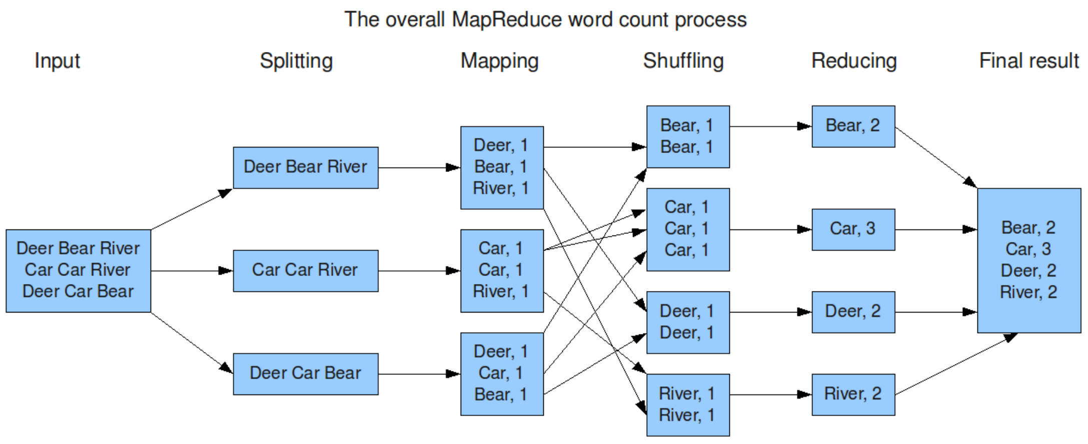
- **File Access (Reads):** The client asks the master for chunk locations. The master replies with the locations of replicas. The client then contacts a chunk server directly to read the data. The master is not involved in data transfer for reads.
- **File Creation:** Handled by the master.
- **Leases for Writes:** For mutations (writes or appends), the master grants a **lease** to one of the chunk replicas, designated as the **primary**. The primary is responsible for **serializing all mutations** to that chunk.
- **Write Process (Simplified):**
    1. The client asks the master to **identify the primary** and **secondary replicas** for the target chunk. The master **grants a lease** to the primary if one **doesn't exist**.
    2. The client pushes the data to **all replicas** (primary and secondaries). Data is temporary stored in an buffer on each chunk server.
    3. Once all replicas acknowledge receiving the data, the client sends a **write request** to the primary.
    4. The primary determines a serial order for mutations and applies the write to its local state.
    5. The primary forwards the write request (and the serial order) to all secondary replicas.
    6. Secondaries apply the mutation in the same order and acknowledge completion to the primary.
    7. The primary replies to the client after receiving ACKs from all secondaries (or on error).

Hadoop (PAPER ANCORA DA VEDERE)

An Apache Hadoop core component (along with **MapReduce** and YARN) designed to support distributed applications processing extremely large datasets (Big Data applications) based on MapReduce programming model.

Open-source framework for distributed storage and processing based on MapReduce programming model. Its main cores are **MapReduce** (Processing), **HDFS** (Storage) and **YARN** (Resource manager and Job Scheduler).

MapReduce



1. **Input Splitting:** The input data is divided into smaller chunks called splits.
2. **Mapping:** Each split is processed by a **Mapper** function, which transforms the input into **key-value pairs**.
3. **Shuffling:** The key-value pairs from all Mappers are sorted and **grouped by key**.
4. **Reducing:** Each unique key and its associated list of values are processed by a **Reducer** function to produce the final output.
5. **Output:** The results from the Reducers are written to the output storage.

HDFS

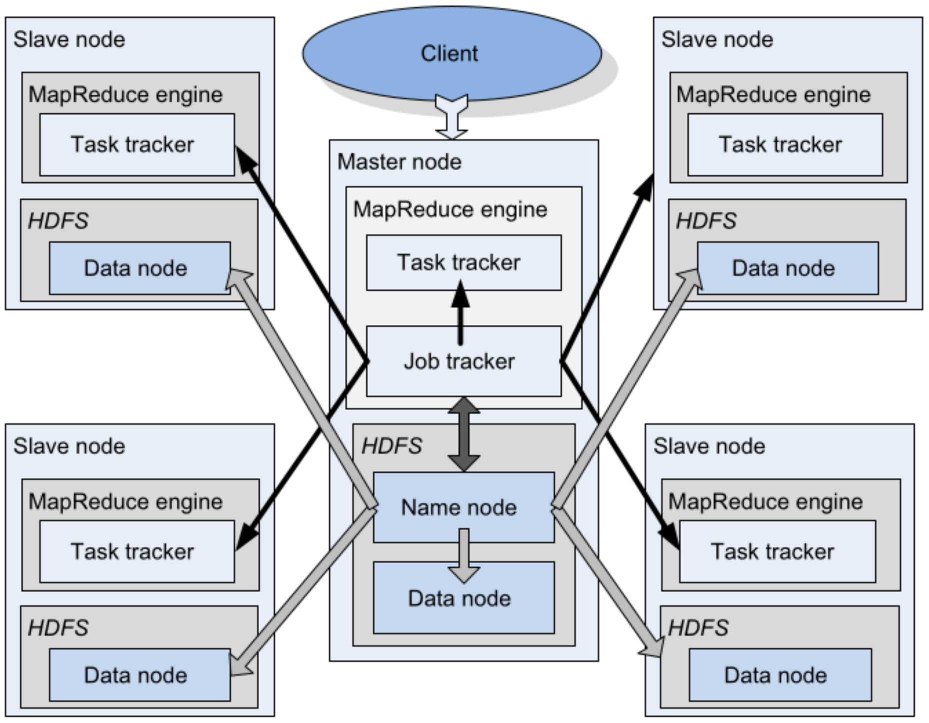


FIGURE 8.8

A *Hadoop* cluster using *HDFS*. The cluster includes a master and four slave nodes. Each node runs a *MapReduce* engine and a database engine, often *HDFS*. The *job tracker* of the master's engine communicates with the *task trackers* on all the nodes and with the *name node* of *HDFS*. The *name node* of *HDFS* shares information about data placement with the *job tracker* to minimize communication between the nodes on which data is located and the ones where it is needed.

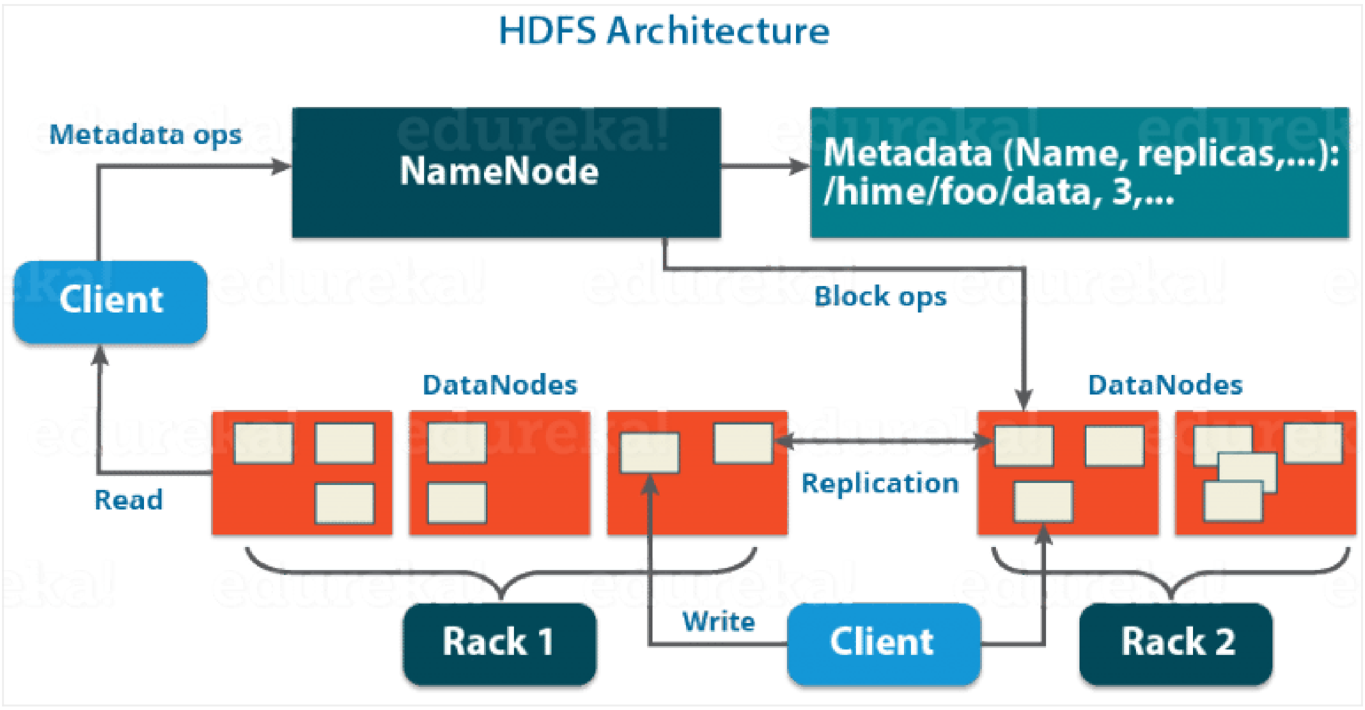
Distributed file system written in Java, making it portable across various platforms but cannot be directly mounted on an existing OS because not POSIX compliant.

Characteristics

- Uses **large block sizes** (typically 64MB or 128MB).
- **Distribute** large dataset on multiple nodes (default 3 replicas) for fault tolerance.

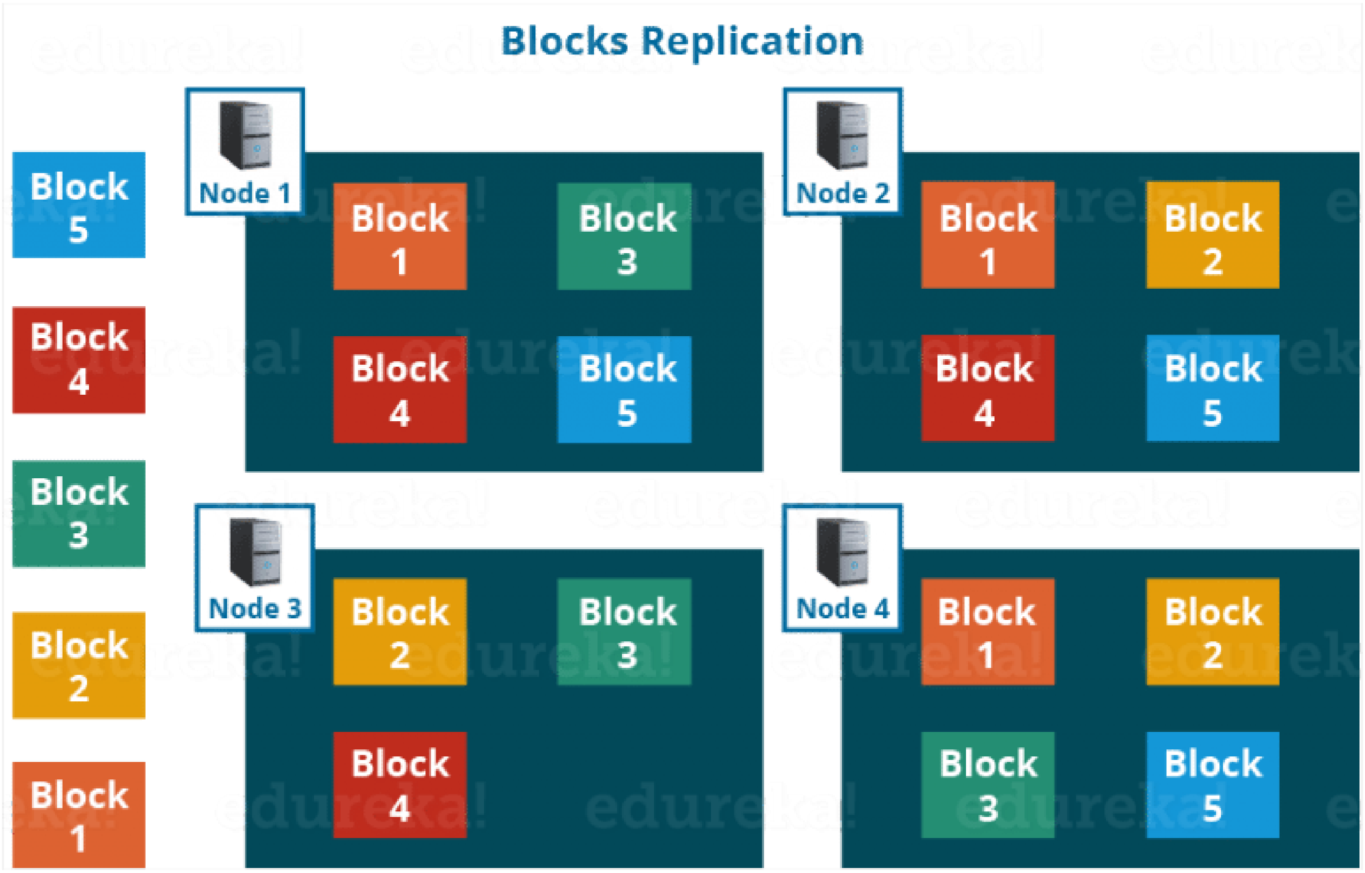
Architecture

HDFS follows a Master/Slave architecture, where a cluster comprise a single **NameNode** (Master) and all the other **DataNodes** (Slave).



- **NameNode (Master):**
  - Manage DataNodes.
  - Keep record of all the blocks in HDFS and in which nodes these blocks are located.
  - Manages the **file system namespace** (directory tree, file-to-block mapping).
  - Stores all **metadata** for the file system (e.g location of stored blocks, size of the files, etc.), there are 2 files associated with the metadata:
    - FsImage: Contains complete state of the file system namespace since the start of the DataNode
    - EditLogs: Contains recent modification w.r.t. to most recent FsImage.
  - Records all **changes to metadata** in an operation log (***EditLog***).
  - Monitors **DataNode liveness** through heartbeats.
    - *Example*: if a file is deleted in HDFS, the NameNode will immediately record this in the EditLog.
- **Secondary NameNode**:
  - Read and write file systems and metadata from Primary NameNode and writes it into the hard disk.
  - Responsible for combining, at regular intervals, the EditLogs with FsImage. Hence, it performs regular **checkpoints**.
- **DataNodes (Slaves):**
  - Manage storage attached to the nodes they run on.
  - Perform low-level r/w operations on data blocks as instructed by clients or the NameNode.
  - Store the actual data blocks.
  - Send heartbeats to the NameNode periodically to report overall HDFS health (default, 3 seconds).

Replication Management



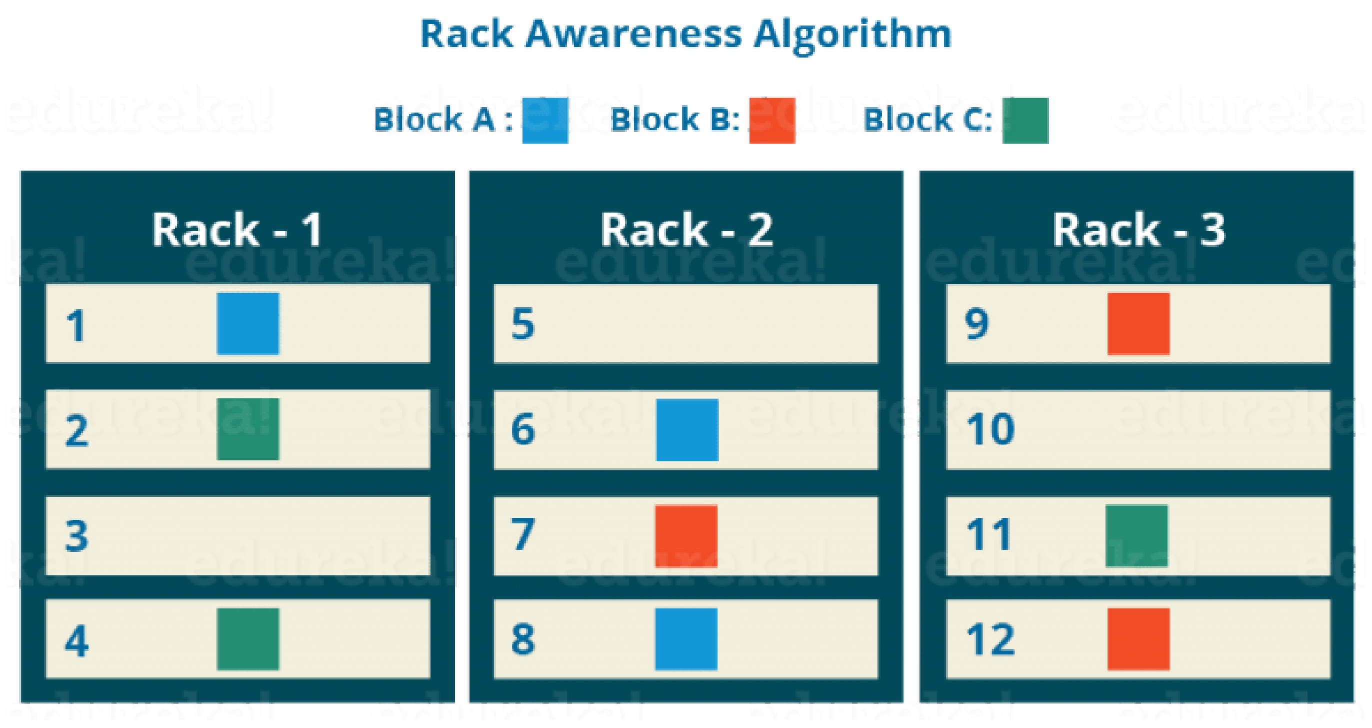
HDFS provides a reliable way to store huge data in a distributed environment as data blocks. The blocks are also **replicated to provide fault tolerance**.

Each block is replicated three times (default) and stored on different DataNodes.

Therefore, if you are storing a file of 128 MB in HDFS using the default configuration, you will end up occupying a space of 384 MB (3*128 MB) as the blocks will be replicated three times and each replica will be residing on a different DataNode.

The NameNode collects block report from DataNode periodically to maintain the replication factor. Therefore, whenever a block is over-replicated or under-replicated the NameNode deletes or add replicas as needed.

Rack Awareness



HDFS's replica placement policy is typically rack-aware. It tries to place replicas on different racks to ensure data availability even if an entire rack fails.

- *Example*: with 3 replicas, one might be on a local rack node, and two on nodes in a different rack.

HDFS Write Protocol (Pipelined)

1. **Setup of Pipeline:**
   - Client, wanting to write a block, requests the NameNode to **choose DataNodes for replicas**.
   - NameNode, provides a list of DataNodes **forming a pipeline** (e.g., DN1, DN4, DN6).
2. **Data Streaming and Replication:**
   - Client writes data packets for the block to the first DataNode (DN1) in the pipeline.
   - DN1 stores the packet and forwards it to the second DataNode (DN4) in the pipeline.
   - DN4 stores the packet and forwards it to the third DataNode (DN6), and **so on, for N replicas**.
3. **Shutdown of Pipeline (Acknowledgement):**
   - Acknowledgement packets flow back up the pipeline in the **reverse direction**. DN6 sends an ACK to DN4, DN4 to DN1, and DN1 to the client.
   - Once the client receives an ACK for the entire block from the first DataNode, it notifies the NameNode that the block write is complete. The **NameNode then updates its metadata**.

- For files larger than one block, this process is **repeated for each block**, and clients can pipeline writes for multiple blocks to improve throughput.

More at: https://www.edureka.co/blog/apache-hadoop-hdfs-architecture/

NoSQL Data Stores

- **Cloud Application Needs:** Cloud applications often require databases with low latency, high scalability, high availability, and specific consistency models.

- **DBMS Role:** A Database Management System (DBMS) should enforce data integrity, manage data access and concurrency control, and support recovery after failures.
- **Relational DBMS (RDBMS):** Typically guarantee ACID (Atomicity, Consistency, Isolation, Durability) properties but are often not easily scalable (in speed and size) for large cloud workloads.
- **NoSQL Database Models:** Suited for scenarios where:
  - The data structure does not require a rigid relational model.
  - The amount of data is very large (Big Data).
  - They may not guarantee full ACID properties, often opting for weaker consistency models like eventual consistency.
  - Include key-value stores, document store databases, and graph databases.
- **NoSQL Transaction Processing Characteristics:**
  - Designed for good scalability and no single point of failure.
  - Often have built-in support for consensus-based decisions (e.g., decisions based on a majority of votes).
  - Support partitioning and replication as fundamental primitives.
  - Employ a "soft-state" approach, allowing data to be temporarily inconsistent. The responsibility for implementing stricter ACID properties may fall on the application developer.
  - Data typically becomes **"eventually consistent"** at some future point, rather than enforcing immediate consistency upon transaction commit.
  - Data partitioning and replication enhance availability, reduce response time, and improve scalability.

Google BigTable

Bigtable is a **distributed storage system** developed at Google for managing massive amounts of structured data, scaling to petabytes across thousands of commodity servers.

Data Model

Bigtable is a sparse, distributed, persistent, multi-dimensional sorted map.

Data is indexed by a unique tuple:

$$(rowkey : string, columnkey : string, timestamp : int64)$$

the value is an uninterpreted string (array of bytes).

Rows

- Row keys are arbitrary strings (typically 10-100 bytes, up to 64KB).
- Data is maintained in **lexicographical order** by row key.
- Operations (reads/writes) under a single row key are **atomic**.
- A table's row range is dynamically partitioned into **tablets** (or *data chunks*), which are the **units of distribution** and **load balancing**, enabling efficient reads of short row ranges.
- Clients can choose row keys to **optimize data locality**.
  - *Example*: Web Pages in the same domain (.com, .it, .org) are grouped together into contigous rows by **reversing the components of URLs**.

Column Families

- Column keys are grouped into sets called **column families**, which are the basic **unit for access control** and **compression**.
- A column family must be created before data storage and is intended to be small in number (hundreds at most) and change rarely.
- A table can have a virtually **unbounded** number of columns within these families (**bounded** number), named using `family:qualifier` syntax.
- Access control, disk, and memory accounting are managed at the column-family level.

Timestamps

- Each cell (intersection of a row and column) can contain multiple versions of data, indexed by a 64-bit timestamp.
- Timestamps can be **auto-assigned** by Bigtable (microseconds "real time") or **manually assigned** by client applications.
- Versions are stored in **decreasing timestamp order**, allowing the most recent to be read first.
- **Garbage collection** for cell versions can be configured per column family (e.g., keep only the last 'n' versions or versions within a certain time window).

API

- Provides functions to **create/delete tables** and **column families**, and **modify metadata** like access control rights.
- Clients can **write**, **delete**, **lookup** values from individual rows, or **iterate** over data subsets.
- **RowMutation** allows **atomic** modifications to a single row.
- **Scanner** enables iteration over data, with various filtering options[cite: 67, 68].
- Supports **single-row transactions** (atomic read-modify-write sequences).
- General cross-row transactions are **not supported**, but client-side write batching is available.
- Cells can be used as atomic integer counters.
- Supports server-side execution of **client-supplied scripts** using Sawzall programming language for data transformation, filtering, and summarization (read-only access to Bigtable).
- Integrates with **MapReduce** as both an input source and an output target.

Building Blocks

- **Google File System (GFS):** Used for storing log and data files (SSTables).
- **SSTable File Format:**
  - Internally used **data format** to store Bigtable data as a persistent, ordered, immutable map from keys to values.
  - SSTables consist of **blocks** (e.g., 64KB) with an index for efficient lookups (typically one disk seek).
  - They can be **memory-mapped** for diskless access.
- **Cluster Management System:**
  - **Job scheduling**
  - **Resource management** on shared machines
  - Machine **failure handling** and **status monitoring** .
- **Chubby:**
  - A highly-available, persistent distributed lock service.
  - Uses **Paxos algorithm** for replica consistency.

- Provides a namespace for small files and directories usable as **locks**, with **r/w atomic operations** to file.
- Bigtable uses Chubby for:
  - Ensuring a **single active master**
  - Storing **bootstrap data** location
  - **Tablet server discovery** and **death finalization**
  - **Schema storage**
  - **Access control lists**.

Bigtable Implementation Details

Major Components

- **Client Library:** Linked into every client application.

- **Master Server (One):**

  - **Assigns tablets** to tablet servers
  - Detects server addition/expiration
  - **Load balancing**
  - Handles GFS **garbage collection**
  - Manages **schema changes** (e.g., table/column family creation).

  Clients rarely communicate with the master for data operations, as tablet locations are cached, resulting in light loaded master.

- **Tablet Servers (Many):**

  - **Manage** a set of tablets (10-1000 per server)
  - Handle read/write requests for (N.B.) **their tablets**
  - Split tablets that grow too large (default 100-200MB).

  A Bigtable cluster store a number of **tables**, which consists of set of tablets, each tablet constains all data associated with a row range.

  Initially, each table consist of a single tablet. As table grows, it is automatically split into multiple tablets

Tablet Location

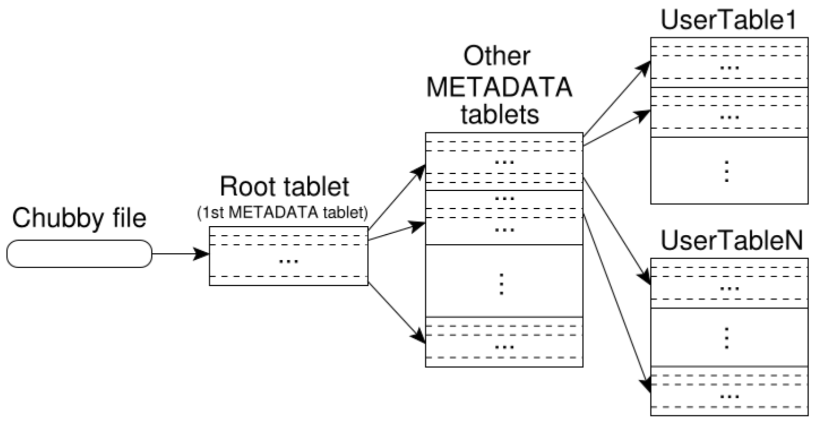


Figure 4: Tablet location hierarchy.

A **three-level tree** hierarchy is used to store tablet location info:

1. A **Chubby file** points to the **root tablet**.
2. The root tablet (first tablet of the METADATA table, **never split**) contains locations of all **METADATA tablets**.
3. METADATA tablets store locations of user tablets, keyed by

- **(table identifier, end row)**.
  - **Table Identifier**: Unique ID of the table
  - **End Row**: Each tablet contains data for a countigous range of rows (row range), the last row tell us where this range stop.

Clients cache these locations and prefetch them to reduce lookup latency.

Tablet Assignment

- The master assigns each tablet to one tablet server.
- The master keeps track of the set of live tablet server and their current tablet assignment.
- If a tablet is **unassigned**, the master assign it to a Tablet Server based on how much tablets are already assigned to it.
- The master uses **Chubby** to track live tablet servers (servers create an **exlusive lock on**, a uniquely-named files in a Chubby directory).
- The master detects **Tablet Server failures** (by asking for lock status or if unreachable) and reassigns their tablets after ensuring the failed server cannot serve **by deleting its Chubby file**.

Tablet Serving

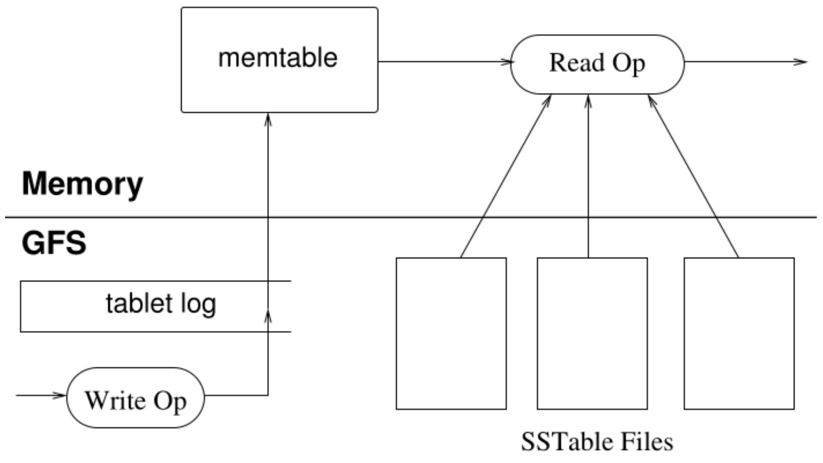


Figure 5: Tablet Representation

- Data for a tablet is stored in GFS as **SSTables**.
- Updates are written to a commit log and then to an in-memory (RAM) sorted buffer called a **memtable**. Older updates reside in SSTables (Disk).
- To **recover a tablet**, a tablet server reads its SSTable list and redo points from METADATA, then reconstructs the memtable by applying updates from commit logs.
- **Writes** are logged then put into the memtable.
- **Reads** operate on a merged view of SSTables and the memtable.

Compactions

- **Minor Compaction:** When a memtable fills, it's frozen, converted to an SSTable, and written to GFS. This frees memory and reduces recovery time.
- **Merging Compaction: Periodically** merges a few SSTables and the memtable into a new SSTable to limit the number of SSTables that read operations need to consult.
- **Major Compaction:** A merging compaction that rewrites all SSTables for a tablet into a single SSTable. This reclaims space from deleted data (as non-major compactions can contain deletion entries) and ensures timely removal of deleted data.

Refinements for Performance and Reliability

- **Locality Groups:** Column families can be grouped. A separate SSTable is generated per locality group per tablet, improving read efficiency for unrelated data. Groups can be declared "in-memory," causing their SSTables to be lazily loaded for diskless reads of **frequently accessed small data** (e.g., METADATA table's location info).
- **Compression:** Clients can specify compression format per locality group, applied to each SSTable block. This achieves good compression ratios, especially for pages from the same host or with multiple versions.
- **Caching:** Tablet servers use a two-level cache:
  - **Scan Cache** (key-value pairs) for repeated data reads.
  - **Block Cache** (SSTable blocks from GFS) for data locality reads.
- **Bloom Filters:** Can be created per SSTable in a locality group to quickly check if an SSTable **might contain data for a specific row/column**, reducing disk seeks for reads, especially for non-existent data.

- **Commit Log Implementation:** A single commit log per tablet server (co-mingling mutations) improves performance and group commit effectiveness. For recovery, this log is sorted in parallel by (table, row, sequence number) so each new tablet server can efficiently read entries for its assigned tablets. Each tablet server uses two log writing threads, switching if the active one performs poorly.
- **Tablet Recovery Speedup:** Before a tablet is moved, the source server performs minor compactions to reduce uncompacted state in the commit log, allowing the new server to load the tablet without log recovery.
- **Exploiting SSTable Immutability:** Simplifies concurrency (no locks needed for reading SSTables), allows efficient row concurrency control (memtable rows are copy-on-write), transforms permanent deletion into garbage collection of obsolete SSTables (master performs mark-and-sweep), and enables fast tablet splits (child tablets share parent's SSTables).

Amazon Dynamo

Dynamo is a highly available and scalable distributed key-value storage system developed and used by Amazon.com for its e-commerce platform. It's designed for services requiring an "always-on" experience, achieving high availability by, in some cases, sacrificing strong consistency.

System Assumptions and Requirements (Section 2.1)

- **Query Model:** Simple read/write operations for binary objects (blobs, usually <1MB) identified by unique keys; no relational schema or multi-item operations. This because Dynamo target application that need to store objects that are relatively small and require simple read/write op.
- **ACID Properties:** Prioritizes availability over strong consistency (the "C" in ACID). Provides no isolation guarantees and only single-key updates.
- **Efficiency:** Must run on commodity hardware (standard, less expensive HW), meeting stringent SLAs (often 99.9th percentile latency and throughput), because critical for service operations. Allows services to tunes their specific performance/cost/availability/durability needs, with consequent tradeoffs.
  - *Example*: Higher availabilty and durability (through data replication) translate in higher cost or slower write performance.

Service Level Agreements (SLA) (Section 2.2)

SLAs are negotiated contracts between client and service on expected request rates for a particular API and service latencies under those conditions. Amazon's SOA relies on each service involved meeting its SLA. Amazon measures SLAs at the 99.9th percentile to ensure a good experience for most customers, rather than using averages or medians. Dynamo's design allows services to tune system properties and let service make their own tradeoffs to meet these demanding SLAs.

Design Considerations (Section 2.3)

- **Consistency vs. Availability:** Strong consistency in traditional approaches, often reduces availability during failures. Dynamo uses ***optimistic replication*** (more below) for higher availability, where changes propagate in the background, tolerating temporary inconsistency that have to be solved later. This can lead to need of conflict resoultion.

- **Eventual Consistency:** Dynamo is designed as an eventually consistent datastore, that is all updates reach all replicas eventually.

- **Conflict Resolution:**
  - **When:** To achieve an "always writeable" store (critical for services like shopping carts that should never reject updates), Dynamo pushes conflict resolution complexity to reads, rather than rejecting writes.
  - **Who:** Resolution can be by the datastore (simpler approached like "last write wins") or by the application (which understands the data schema and can merge versions). Dynamo allows the application to choose, using better approaches, otherwise, data store do the job by applying simple approaches.

- **Other Principles:** Incremental scalability (add nodes one at a time), symmetry (all nodes have same responsibilities), decentralization (peer-to-peer over centralized control), and heterogeneity (distribute work based on server capabilities).

- **Optimistic vs Pessimistic Replication**: Optimistic replication prioritizes availability and "always writeable" characteristics, **accepting that inconsistencies might temporarily arise and will need to be resolved**. Pessimistic replication prioritizes strong consistency, potentially at the cost of availability or write latency if replicas are not reachable.

Related Work (Section 3)

Dynamo is compared to other decentralized storage systems, P2P systems (like Freenet, Gnutella, Pastry, Chord, OceanStore, PAST), distributed file systems (like Ficus, Coda, GFS, Farsite), and databases (like Bayou, Bigtable).

Key differentiators for Dynamo are its primary goals: 1. Being "always writeable" for high availability where no updates are rejected. 2. Its operation within a single administrative domain, where each node is assumed trusted. 3. Its focus on simple key-value access without hierarchical namespaces or complex relational schemas, 4. Its design for low-latency (99.9th percentile) operations, achieved partly by being a **zero-hop DHT**, each node mantains enough info. locally to route a request to the appropriate node directly to avoid multi-hop routing.

System Architecture (Section 4)

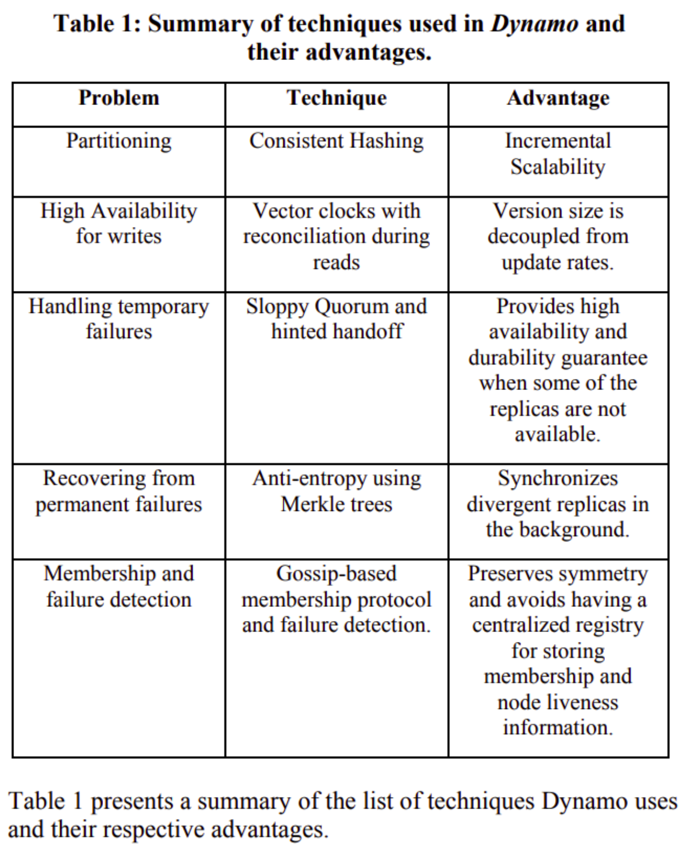


Table 1: Summary of techniques used in *Dynamo* and their advantages.

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

Table 1 presents a summary of the list of techniques Dynamo uses and their respective advantages.

Dynamo employs several core distributed systems techniques:

System Interface (Section 4.1):

Exposes `get(key)` and `put(key, context, object)` operations.

- `get(key)`: Locates replicas, returns a single object or a list of conflicting versions along with `context`.
- `put(key, context, object)`: Writes replicas; the `context` is metadata (including version info).

Keys are hashed (MD5) to 128-bit identifiers to **determine responsible storage nodes**.

Partitioning Algorithm (NOT CLEAR) (Section 4.2)

Uses **consistent hashing** to distribute keys across nodes on a logical "ring".

- **Consistent hashing**: Distributed hashing technique that allows for the distribution of data across a set of nodes (like servers in a distributed system) in such a way that **minimizes the number of keys that need to be remapped** when nodes are added or removed. Each node is assigned random positions (tokens) on the ring. To address non-uniform load distribution and heterogeneity, Dynamo uses **virtual nodes**: each physical node is responsible for multiple virtual nodes/tokens on the ring. This evens load distribution during node additions/removals and allows capacity-based token assignment.

Replication (Section 4.3)

Data is **replicated on N hosts** (N is configurable) for high availability and durability. A key's coordinator node stores the data locally and replicates it to N-1 clockwise successor nodes on the ring. The list of N nodes responsible for a key is its **preference list**, constructed to ensure distinct physical nodes, even with virtual nodes.

Data Versioning (Section 4.4)

To achieve **eventual consistency** (consistency model for distributed computing, more below) and handle concurrent updates, Dynamo treats **each modification as a new version of data**, allowing multiple versions to coexist.

- **Vector Clocks:** `(node, counter)` pairs are associated with each object version to capture **causality** (more below) between different versions of the same object. They help determine if versions are ancestors, descendants, or in conflict (parallel branches). When updating, clients pass the `context` (containing the vector clock) from a prior read.
- **Conflict Resolution:** If a read encounters **causally unrelated versions**, all are returned to the client for **semantic reconciliation** (e.g., merging shopping carts). The reconciled version then supersedes the divergent ones. Vector clock sizes are managed by truncating the oldest (node, counter) pair if a threshold is met, storing a timestamp with each pair to identify the oldest.
  - **Eventual Consistency**: consistency model in distributed computing that ensures all nodes in a system will eventually agree on the same data, even if updates are made concurrently.
  - **Causality**: Influence by which one event contributes to production of another.

STOP HERE IN SLIDE

Execution of `get()` and `put()` (Section 4.5)

- Requests can be routed via a **load balancer** or a **partition-aware client library** (for lower latency). If a non-coordinator node receives a request, it forwards it to a node in the key's preference list.
- Operations involve the first N healthy nodes in the preference list.
- A **sloppy quorum** system is used with configurable **R (minimum read replicas)** and **W (minimum write replicas)**. For consistency, often **R+W > N**.
- `put()`: Coordinator generates a new vector clock, writes locally, sends to N highest-ranked reachable nodes. **Success** if at least W-1 nodes respond.
- `get()`: Coordinator requests versions from N highest-ranked reachable nodes, waits for R responses, returns result (possibly with multiple versions for client reconciliation).

Handling Failures: Hinted Handoff (Section 4.6)

Hinted Handoff: If a target node in the preference list is down/unreachable, the replica is sent to a different healthy node (not necessarily in the top N) with a "hint" indicating the intended recipient.

The hinted node stores this replica temporarily and attempts to deliver it to the original target node once it recovers. This ensures R/W operations don't fail due to temporary issues. Setting W=1 maximizes write availability, even thought most Amazon services set a higher W to meet desidered durability.

Hinted Handoff works only if node **failures are transient**, in case of permanent failure, more complex techiniques have to be used.

Handling Permanent Failures: Replica Synchronization (Anti-entropy) (Section 4.7)

An anti-entropy protocol using **Merkle trees** synchronizes replicas and handles permanent failures or lost hinted replicas.

A Merkle tree is a hash tree where leaves are hashes of key values. Nodes exchange Merkle tree roots for common key ranges; if roots differ, they traverse down the trees to find and synchronize inconsistent keys, minimizing data transfer. Each node maintains a tree per key range it hosts.

Membership and Failure Detection (Section 4.8)

- **Ring Membership:** Node addition/removal is an explicit administrative operation issued to a Dynamo node. Changes are persisted and propagated via a **gossip-based protocol** for eventual consistency. Node-to-token mappings are also gossiped.
- **External Discovery (Seeds):** Some nodes are designated as "seeds" (discoverable externally) to prevent logical ring partitions during membership changes.
- **Failure Detection:** A local notion is used. If node A fails to communicate with node B, A considers B failed and uses alternate nodes for requests mapped to B's partitions, retrying B periodically. Explicit join/leave methods reduce the need for a globally consistent failure view.

Adding/Removing Storage Nodes (Section 4.9)

When a new node is added, it gets assigned random tokens. Existing nodes responsible for the key ranges now covered by the new node transfer the relevant data to it. This distributes bootstrapping load.

Implementation (Section 5)

Each Dynamo node has three Java components: request coordination, membership/failure detection, and a pluggable local persistence engine (e.g., Berkeley DB Transactional Data Store, MySQL) chosen based on application needs (object size, access patterns). Most instances use BDB. Request coordination uses an event-driven (SEDA-like) model with Java NIO, where each client request creates a state machine on the coordinator node.

Read repair is performed opportunistically: if a read coordinator receives stale versions from some replicas, it updates them with the latest version after responding to the client. Any of the top N nodes in a preference list can coordinate a write to distribute load and improve read-your-writes consistency by choosing the node that replied fastest to a previous read.

Experiences & Lessons Learned (Section 6)

Dynamo instances are configured with specific N (number of replicas, typically 3), R (read quorum), and W (write quorum) values, often (3,2,2), to meet application SLAs for performance, availability, and durability.

- **Usage Patterns:** Business logic specific reconciliation (client merges versions, e.g., shopping cart), timestamp-based reconciliation ("last write wins," e.g., session data), and as a high-performance read engine (R=1, W=N, e.g., product catalog cache).
- **Performance vs. Durability:** Dynamo targets 99.9% R/W within 300ms. Write latencies are higher than reads. An optimization using an in-memory write buffer can significantly lower 99.9th percentile latencies but trades some durability (server crash can lose buffered writes). This risk is mitigated by having one replica perform a "durable write" while the coordinator only waits for W responses.

- **Load Distribution:** Consistent hashing aims for uniform load. The paper discusses three partitioning strategies. Strategy 3 (Q/S tokens per node, equal-sized partitions) achieved the best load balancing efficiency and reduced metadata size compared to the initial strategy (T random tokens per node, partition by token value) because it decouples partitioning from placement, simplifies bootstrapping/recovery, and eases archival.
- **Divergent Versions:** Occur rarely, mostly due to many concurrent writers (e.g., busy robots) rather than system failures.
- **Coordination:** Client-driven coordination (client library manages R/W state machine) significantly reduces latency compared to server-driven (via load balancer) by eliminating an extra network hop.
- **Background vs. Foreground Tasks:** An admission controller manages resource slices for background tasks (replica sync, data handoff) based on monitored foreground operation performance to prevent contention.
- **Overall:** Dynamo provided high availability (99.9995% successful responses). Exposing consistency and reconciliation logic to developers was manageable as many Amazon apps were already designed for inconsistencies. The full membership model (gossiping full routing tables) scales to hundreds of nodes; larger scales might need hierarchical extensions.

Conclusions (Section 7)

Dynamo successfully combines decentralized techniques to provide a highly available and scalable key-value store, demonstrating that an eventually-consistent system can be a robust building block for demanding applications. It allows service owners to tune parameters (N,R,W) to meet specific SLAs.

Amazon S3 (Simple Storage Service)

Storage Model:

Data is stored as named **"objects"** which are grouped into named **"buckets"**. It can be conceptualized as a map: `bucket_name + object_key + version_id -> object_data`.

- **Buckets:**
  - Bucket names are **globally unique**.
  - Users can create **up to 100 buckets** by default.
  - Can be create, listed and deleted via API.
  - **Access Control Lists** (ACLs) control read/write permissions.
- **Objects:**
  - Can store any sequence of bytes, from 1 byte up to 5TB.
  - The **object key** (or key name) is unique identifier in a bucket. It can be thought of as a URI path name.
  - Objects are stored and retrieved (GET/PUT) in their **entirety** or by **specific byte** ranges.

Failure/Error Handling:

Applications using S3 must be designed to handle read/write failures and errors, typically by retrying requests.

S3 returns an **ETag** (which is the MD5 checksum of the object) with write acknowledgments and read responses.

- **Write Failure**: Clients should compute the MD5 of objects they write and compare it with the ETag to ensure data integrity against corruption during transmission or storage.
    - If a mismatch occurs, the operation should be **retried until succeed**.
- **Read Failure**: Successful read op. return the object requested along with MD5 stored in ETag. Client application (optional) can compute MD5 for comparison.
    - If mismatch is detected, client should retry request.

Data Consistency Model for Objects:

- **Strong read-after-write consistency** for:
    - PUTs of new objects
    - PUT requests that overwrite existing objects
    - DELETE requests.
      This means after a successful write or delete, any subsequent read request will immediately receive the new version or see the object as deleted.
- Updates to a single key are **atomic**. If two threads concurrently PUT and GET the same key, the GET will return either the old data or the new data, but never partial or corrupt data.
- A PUT request returns successfully after **all replicas are updated**.
- **Strong consistency** also applies to **read operations** on ACLs, Object Tags, and object metadata.
- **Concurrency and Locking:**
    - Amazon S3 **does not support object locking** for concurrent writers.
        - If two PUT requests are made simultaneously to the same key, the request with the **latest timestamp wins** (last-writer-wins semantics). Applications needing stricter concurrency control must implement their own locking mechanisms.
    - **Updates are key-based**; there is no way to make atomic updates across multiple keys unless built into the client application.

Data Consistency Model for Buckets:

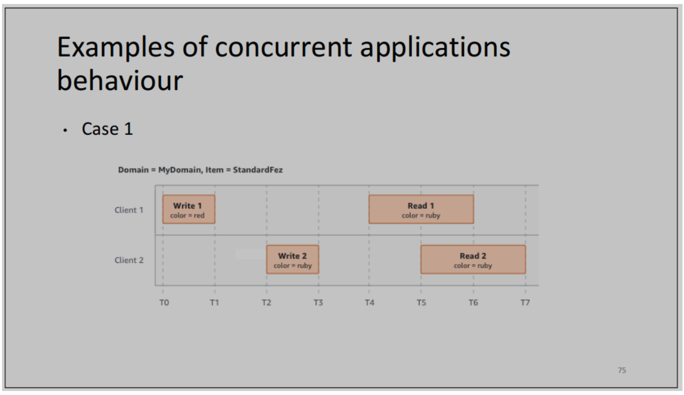
- Bucket configurations (e.g., listing buckets after deletion, enabling versioning) exhibit an **eventual consistency model**.
    - Changes might take a short time to fully propagate across the system.
    - A 15-minute waiting period is sometimes suggested before performing object operations after certain bucket configuration changes.

Concurrent Application Behavior Examples:**

General Setup:

- **Domain = MyDomain, Item = StandardFez:** This indicates that all operations are happening on the same object "StandardFez" within the "MyDomain".
- **Client 1 & Client 2:** Two different applications or actors are concurrently interacting with this object.

Case 1: Non-Overlapping Writes then Reads

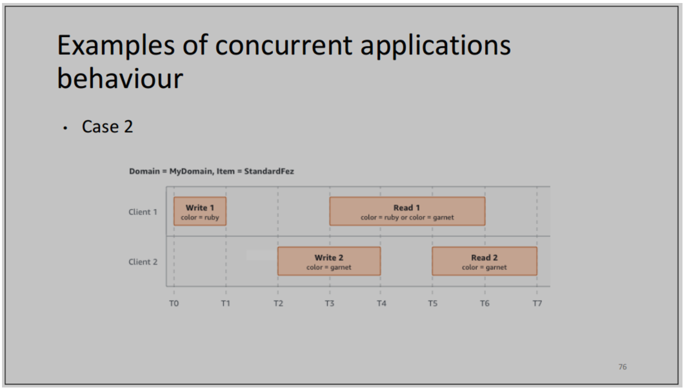


Examples of concurrent applications behaviour

- Case 1

In this scenario, the writes from Client 1 and Client 2 do not overlap in time. Client 1 writes first, and then Client 2 writes a different value. When each client reads after their respective write (and after the other client's write has completed), they observe the value they last wrote. This is a straightforward case where the order of operations is clear and there's no ambiguity due to concurrency.

Case 2: Overlapping Writes, Read After Both

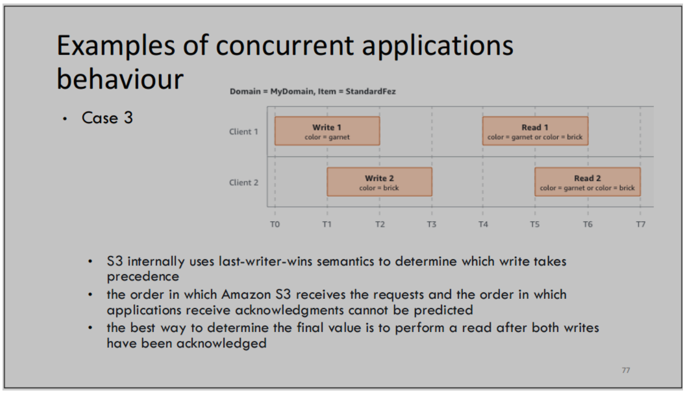


Examples of concurrent applications behaviour

- Case 2

- Client 1's read happens before Client 2's write is fully acknowledged or processed by the system. Therefore, Client 1 *might* read its own write ("ruby") or it *might* read the value that is in the process of being updated to "garnet," leading to the observation of "color = ruby or color = garnet."
    - The exact timing and system internals determine what Client 1 sees in this race condition.
- Client 2's read happens after its own write has completed and after Client 1's write has likely been overwritten. Thus, Client 2 consistently reads "color = garnet."

Case 3 (Image 3): Overlapping Writes, Reads After Both



Examples of concurrent applications behaviour

- Case 3

Domain = MyDomain, Item = StandardFez
Client 1
Write 1
color = garnet
Read 1
color = garnet or color = brick
Client 2
Write 2
color = brick
Read 2
color = garnet or color = brick
T0
T1
T2
T3
T4
T5
T6
T7

- S3 internally uses last-writer-wins semantics to determine which write takes precedence
- the order in which Amazon S3 receives the requests and the order in which applications receive acknowledgments cannot be predicted
- the best way to determine the final value is to perform a read after both writes have been acknowledged

77

Writes from Client 1 and Client 2 overlap. Client 2's write starts while Client 1's write is still in progress or has just completed.

Due to "**last-writer-wins**," the value written by Client 2 ("color = brick") will be the final value.

- The exact timing and system internals determine what Client 1 sees in this race condition, so, best way to determine the final value is to perform a read after both writes have been acknowledged

Key Takeaways from Case 3:

- **Last-Writer-Wins:** This is the crucial consistency model being demonstrated. The write that is received and processed last by the system determines the final value.
- **Unpredictable Order:** The order in which S3 receives requests and the order in which applications receive acknowledgments are not guaranteed to be the same as the order in which the writes were initiated by the clients. This highlights the challenges of concurrent systems.
- **Read After Both Writes:** The most reliable way to determine the final value in a concurrent write scenario is to perform a read *after* both write operations have been acknowledged by the system. This gives the "last-writer-wins" mechanism time to take effect and for the final state to be observable.

These scenarios effectively illustrate the complexities of concurrent operations in a distributed system with "last-writer-wins" consistency. They highlight the potential for intermediate or inconsistent reads and emphasize the importance of understanding the system's consistency model when designing concurrent applications.