Unit # 1

- CAP Theorem
- Big Tables
- Chubby
- Bloom Filter

Assignment (1) :
- Motivation for Big data ?

Assignment (2) :
- RDB Principles ?

CAP Theorem's Core Idea

- The CAP Theorem is a fundamental principle in distributed systems, introduced by computer scientist Eric Brewer in 2000. It stands for ***Consistency, Availability***, and ***Partition Tolerance***, and asserts that a distributed system can simultaneously achieve at most two out of these three properties.
- A distributed system like Cloud system is a network that stores data on more than one node (physical or virtual machines) at the same time. It’s essential to understand the CAP theorem when designing a cloud system so that we can choose a data management system that delivers the characteristics our application needs most.

1. Consistency (C)

✓ Every read from the system returns the most recent write. In other words, all nodes in the distributed system see the same data at the same time i.e If we read data after a write operation, we should get the latest value that was written, regardless of which node we read from.

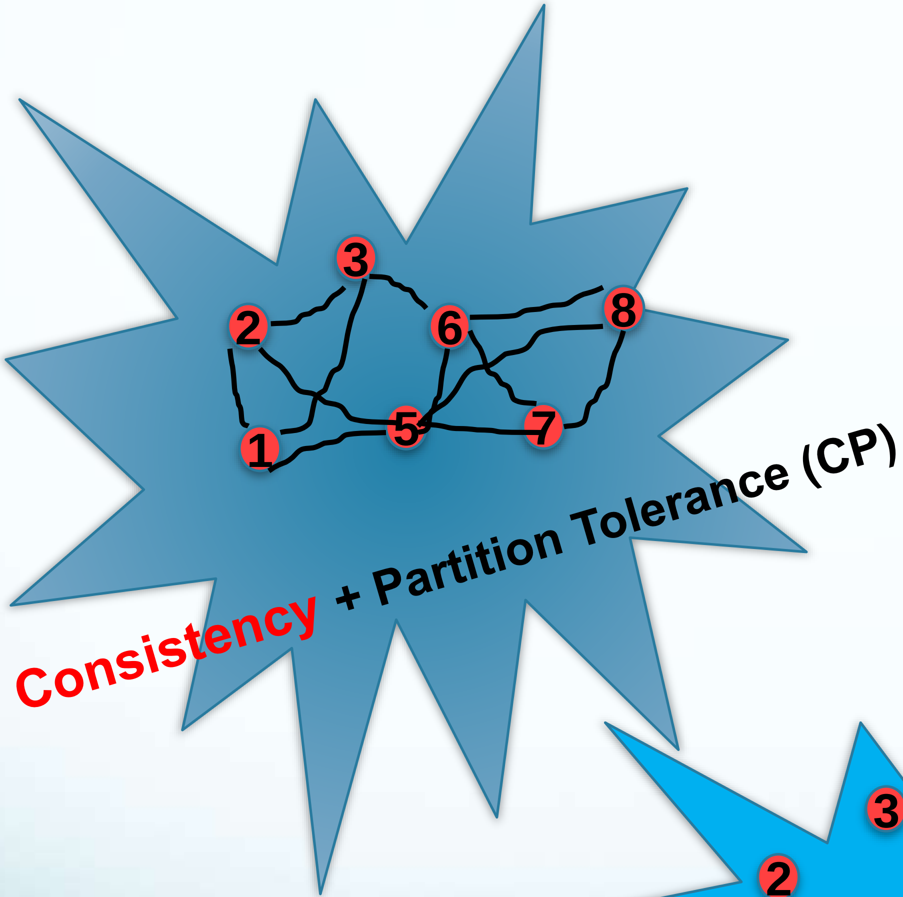
2. Availability (A)

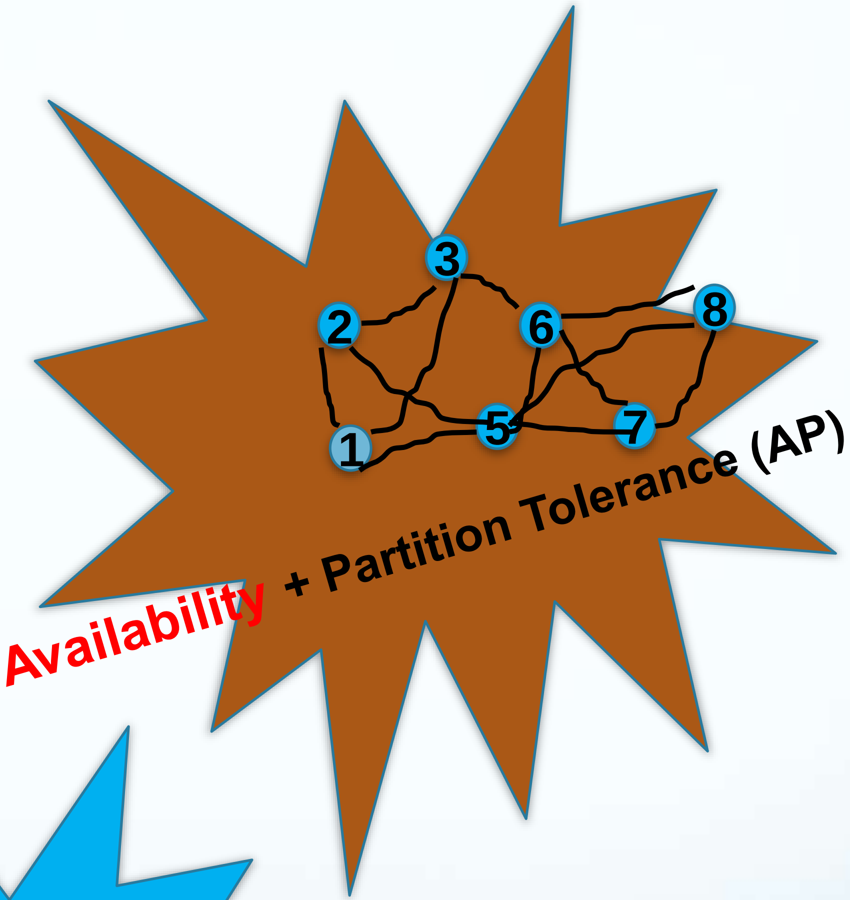
✓ Every request (read or write) receives a response, even if some of the nodes are down or unreachable. i.e the system is always available to serve read and write requests, and the data we read might not be the most recent if the system is partitioned.

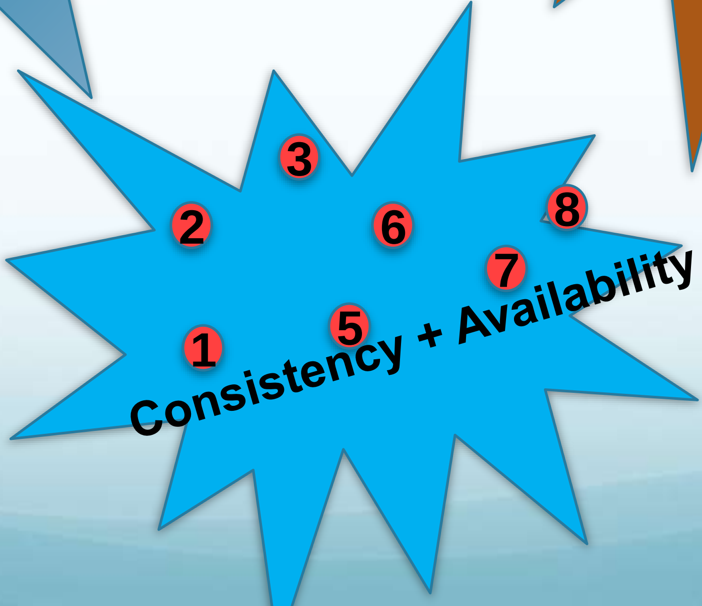
3. Partition Tolerance (P)

✓ The system continues to operate, despite network partitions or communication breakdowns between nodes in the distributed system i.e the system can handle the failure of parts of the network that prevent some nodes from communicating with others.

CAP Theorem's Core Idea (contd)







CAP Theorem's application

"Apply the CAP Theorem to a Bank ATM service and discuss scenarios arising"

Scenario 1 : Consistency + Partition Tolerance (CP)

In this scenario , we assume that the bank prioritizes **Consistency** and **Partition** tolerance, sacrificing **Availability** in certain situations.

Consistency: The bank ensures that every transaction is immediately and correctly reflected across all ATMs and the central database. If an ATM is processing a withdrawal, it communicates with the central server to update the account balance.
Partition Tolerance: Even if there's a network issue or partition between the ATM and the centra server, the ATM will either wait for confirmation from the server before dispensing cash or deny the transaction.
Availability : The Tradeoff is that availability is sacrificed. During a network partition, the ATM might become unavailable, preventing customers from withdrawing cash until the network is restored and consistency can be guaranteed. This ensures that the account balance remains accurate and consistent but could frustrate customers due to service unavailability.

Scenario 2 : Availability + Partition Tolerance (AP)

In this scenario , we assume that the bank shall prioritizes **Availability** and **Partition** tolerance, sacrificing **Consistency** in certain situations.

CAP Theorem's Core Idea (contd)

Availability : The ATM shall remains operational even if there is a network outage and shall allows the customer to withdraw money, even though it cannot immediately communicate with the central server.
Partition Tolerance: Even if there's a network issue the ATM can still function and dispense cash, despite the network partition.
Consistency : The Tradeoff is that consistency is sacrificed. During a network outage , the ATM should have become unavailable but in this case it is continuing working. This shall raise serious concern that the customer's withdrawal might not be immediately reflected in the central database, leading to a potential discrepancy in the account balance across different systems. For example, the same customer might withdraw cash from another ATM that's connected to a different server, potentially causing the account to be overdrawn.

Scenario 3 : Consistency + Availability (CA)

In this case we see that the bank prioritizes **Consistency** and **Availability**, with the understanding that **Partition** tolerance may not be possible during a network issue. The bank aims to ensure that transactions are always consistent and that ATMs remain available, assuming no network partitions occur.

CAP Theorem's Significance

- In the context of Big Data, the CAP Theorem is crucial because it influences the design and behavior of distributed databases and data processing systems, which are foundational in managing and analyzing large-scale data.
- Depending on the application, some Big Data systems may prioritize consistency to ensure data accuracy, while others may prioritize availability to ensure that data can be accessed quickly, even during failures. Partition tolerance is non-negotiable in Big Data systems due to the inherent distributed nature and scale of the data across multiple nodes and geographies.
- For example, in Hadoop, consistency and partition tolerance are often prioritized. Data is processed in large batches, and consistency across nodes is crucial to ensure accurate analysis. Availability might be sacrificed because the system can afford to delay processing if nodes are temporarily unreachable, as the focus is on accuracy and completeness of data.
- Whereas in systems where uptime and responsiveness are more critical than immediate consistency, such as social media platforms, real-time analytics, or Big Data systems like Cassandra and DynamoDB , they are designed to handle real-time data across multiple nodes with high availability and less focus on consistency. Such systems are widely used in scenarios where data needs to be available even during network issues. In this case Consistency is relaxed as different nodes might have slightly different versions of data during a partition, but the system ensures that it remains operational and responsive.

CAP Theorem's Core Idea (contd)

Asignment-3

“Apply the CAP Theorem to a Social messaging platform like Whatsapp or Twitter and discuss scenarios arising”

Scenario 1 : Consistency + Partition Tolerance (CP)

Scenario 2 : Availability + Partition Tolerance (AP)

Scenario 3 : Consistency + Availability (CA)

Big Tables

- Originally developed by Google in 2004, Big table was built to meet the demands of applications that require real-time, high-throughput access to large volumes of data. It is nothing but is a distributed, scalable, and high-performance **NoSQL database** designed to handle massive amounts of structured data across many machines.
- It serves as the underlying database for many of Google's applications, including Google Search, Google Analytics, Google Earth, and Gmail.
- Big Tables in the context of Big Data refers to large, distributed, and scalable data storage systems designed to handle vast amounts of structured data across many servers. This concept is primarily associated with Google's Big table, which is a foundational technology in the realm of Big Data storage systems.
- Today, Google Cloud Big table is offered as a managed service on the Google Cloud Platform **(GCP),** and is a cornerstone of modern big data infrastructure, especially for applications that require low-latency reads and writes with high scalability.

Key Features :

1. Scalability

Big Tables have characteristics that they can be scaled horizontally by adding more machines to the cluster, enabling users to manage massive datasets efficiently. That is how Big tables can manage petabytes of data across thousands of servers. Additional servers can be added or turned off at will.

Big Tables (contd)

2. *Column-Family Format*

- Big table are sparse, distributed, multi-dimensional sorted map where data is stored in a flexible, column-family format. This structure allows for efficient data storage and retrieval, even if the data is sparse.
- Every row in Big table is uniquely identified by a row key. Data is stored in lexicographic order based on these row keys, allowing fast lookups by the primary key.
- Each table has one or more **column families**, which group related data together. Column families help in organizing and optimizing data reads and writes. Inside each column family, you can have different columns that store specific data about the product.
- Every cell in Big table can store multiple versions of data, each tagged with a **timestamp**. This feature is useful for versioning, allowing access to historical data.

Stu_Id	Student	Student	Marks	Marks	Result

Big Tables (contd)

Stu_Id	Student	Student	Exam	Marks	Marks	Result
001	Ram	“IT”	Inter	10		P
002	Sham	“CS”	Exter		20	P
003	manoj	“Elect”	Exter		20	P

Student : This column family contains basic information about the students , such as his name (Student:Name) and stream (Student :stream).

Similiarly , the second column family is Marks that distinguish between External and Internal exam obtained marks.(Marks:Inter) , (Marks:Exter)

In Bigtable, rows can have different columns, and columns are only created if there is data. For example, above in first row there is no Marks:Inter data because no marks is applied to Intern . Similarly no column applicable for row 2 & 3 for Internal exam marks as
Note that the system doesn’t need to store a null value; it just skips that column for this row, making the system space-efficient.
This is not possible in traditional database where sparse data is discouraged.

Big Tables (contd)

3. *High Availability and Fault Tolerance:*

- Bigtable replicates data across multiple machines for fault tolerance. If one machine or node fails, the data is available from another node, ensuring high availability.
- When nodes fail or go down for maintenance, Bigtable's failover mechanisms ensure that requests are automatically rerouted to healthy nodes without user intervention.
- Bigtable ensures strong **consistency** at the row level. This means that updates to a row are atomic, and any read operation on that row will return the latest written data.
- For systems that prioritize availability, Bigtable allows for eventual consistency across rows, meaning data written to different rows may not be immediately consistent across all replicas but will eventually synchronize.

4. *No SQL structure:*

- Unlike traditional relational databases, in Big table different rows in the same table can have different columns, and new columns can be added on the fly without altering the schema or affecting existing rows. In a relational databases, the schema the structure of tables and columns must be predefined and are rigid.
- The flexible schema allows each user to store only the information that they have provided, without requiring all rows to have the same set of columns. **This means no storage space is wasted on fields that don't apply to certain users.**

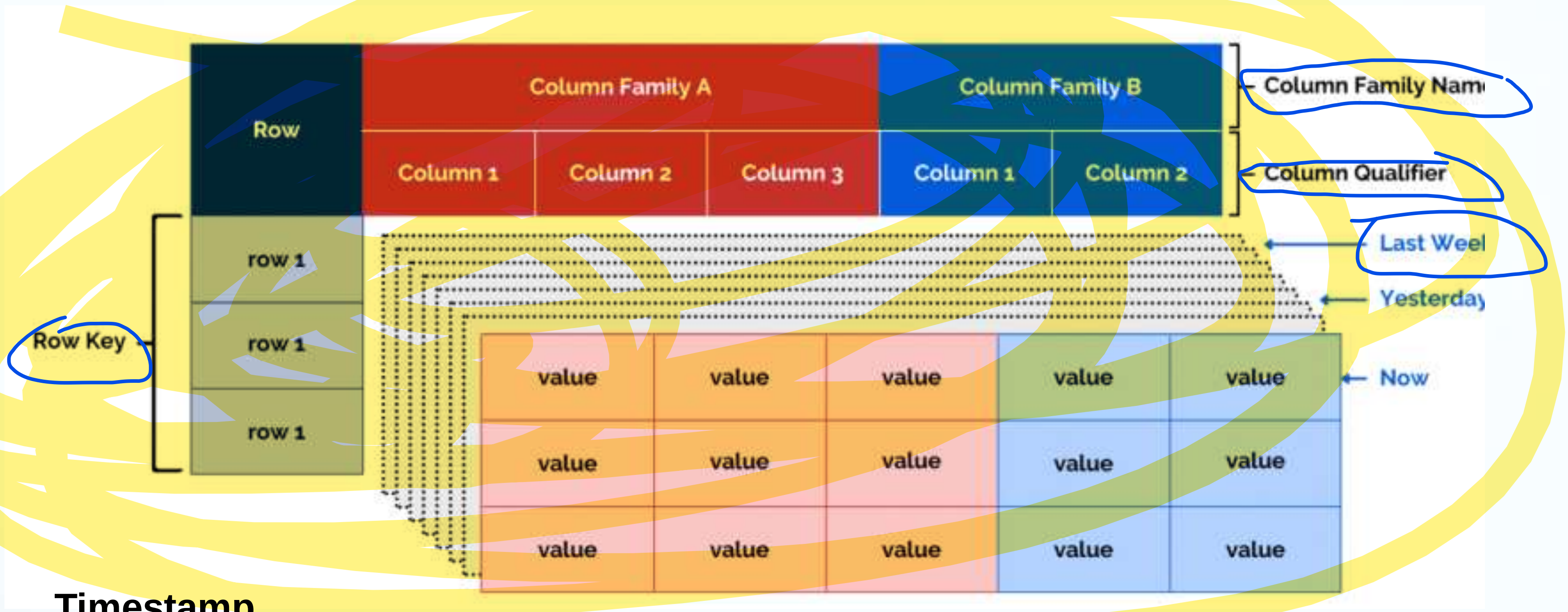
Big Tables (contd)

Id	Basic_Info	Basic_Info	Contact_info	Contact_info	Food	Food
001	Ram	ram@gmail.com	9863456789	Timarpur,Delhi	V	P
002	Sham	sham@yahoo.com		Civil lines 54		
003	manoj	manoj@rediff.com	8975623490			NV

Say if a new field needs to be added, such as a "Profile Picture," it can be added as a new column in the "BasicInfo" column family without modifying the schema or affecting existing users. You can add it for only the users who provide that information. This flexibility means that Sham's profile doesn't need an empty "ProfilePicture" field, saving storage and keeping the schema adaptable.

Id	Basic_Info	Basic_Info	Basic_Info	Contact_info	Contact_info
001	Ram	ram@gmail.com	Ram.jpeg	9863456789	Timarpur,Delhi
002	Sham	sham@yahoo.com			Civil lines 54
003	manoj	manoj@rediff.com	Manoj.png	8975623490	

Big Tables (contd)



Timestamp

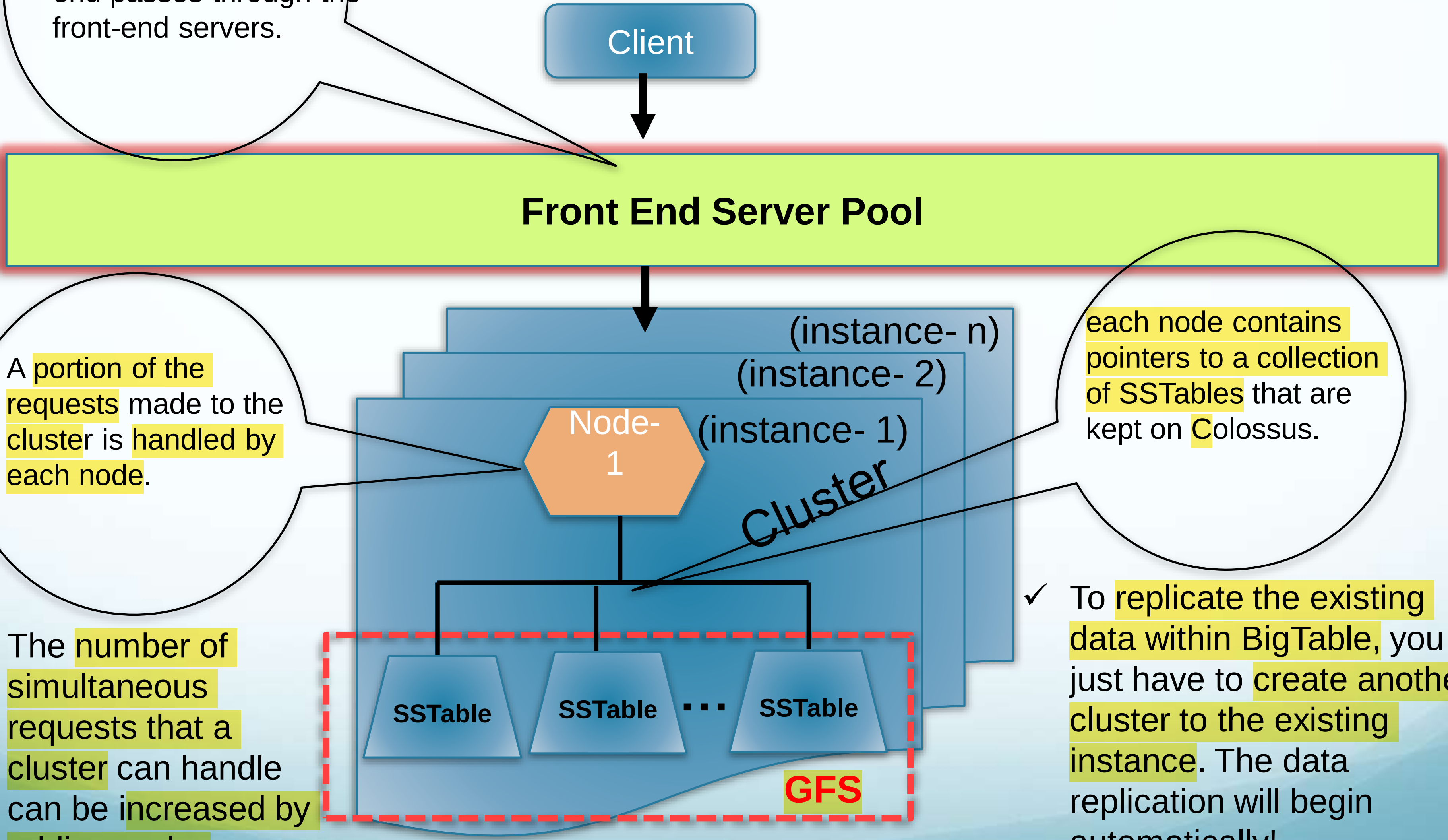
The time dimension in Big Tables makes Big Tables different from the relational databases as it creates multiple columns for the same rows (key). This capability makes the Big Table to capture real time data that is ever changing and ever growing and it does not requires the schema to be changed with time. Timestamps make it possible to access most recent or past values or use values among a specific range of timestamp. It makes Big tables not just any ordinary database but a special business tool. It literally help us go back in time to know the best interactions for ideal analysis. Cloud BigTable thus get the ability to do Big data Analytics and tap the capability of Machine Learning algorithms with speed.

Architecture of Big Tables

First, the requests that come from the client's end passes through the front-end servers.

A portion of the requests made to the cluster is handled by each node.

each node contains pointers to a collection of SSTables that are kept on Colossus.



- ✓ The number of simultaneous requests that a cluster can handle can be increased by adding nodes.
- ✓ To replicate the existing data within BigTable, you just have to create another cluster to the existing instance. The data replication will begin automatically!

Architecture of Big Tables (contd)

The architecture of Big table consists of several key components that ensure its scalability, availability, and performance. However the three main components are Front end server pool , Tablet server and SSTable. They are discussed below :

1. Front-End Server Pool

- The Front-End Server Pool (also called the Front-End Layer) acts as the gateway or interface between the client (user, application, or API) and the internal components of the Big Table system, including the tablet servers.
- They handle authentication and security checks, ensuring that only authorized clients can access the data.
- After authentication it routes client requests to the correct tablet server. They are typically the first point of contact when a client sends a read or write request.
- It also ensures that client requests are **load-balanced** across available tablet servers. It also **aggregates data** from different tablet servers when necessary, especially for complex queries.
- Front-end servers may include caching mechanisms to reduce the load on tablet servers by storing frequently accessed data in memory, improving response times for the clients.

2. Tablet Servers:

- A Tablet Server is a critical component in Google Bigtable, which is a distributed NoSQL database. The tablet server is responsible for managing and serving data to clients (applications or users) in a distributed environment. It plays a key role in how Bigtable stores, retrieves, and processes data across multiple machines.

Architecture of Big Tables (contd)

- A tablet server stores data in tablets, which are portions of a larger table. Each tablet contains a range of rows from the **tablet**.
- When an application requests data, the tablet server retrieves the appropriate data from memory (Memtable) or disk (SSTables).
- If a tablet becomes too large, the tablet server splits it into smaller tablets. The system redistributes these smaller tablets across multiple tablet servers to ensure **load balancing.**

3. SSTable :

- SSTable (Sorted String Table) is a file format used to store large amounts of data in a sorted and efficient way. It is an immutable data structure, meaning that once data is written to an SSTable, it cannot be modified. They allow for fast and efficient **read operations** while minimizing the need for complex data structures like traditional B-trees used in relational databases.
- The data in an SSTable is always stored in a **sorted order** by key (usually the row key in databases like Bigtable or HBase). This makes searching through the data much faster.
- SSTables are stored **on disk** and are used to hold large datasets that don't fit in memory. They provide efficient access to data even for very large tables.
- Before data is written to an SSTable, it is first stored in memory in a Memtable. When the Memtable gets filled up (reaches a certain size), it is flushed to disk as a new SSTable.

Architecture of Big Tables (contd)

- Each SSTable contains two primary components: The Data block that holds the actual data (e.g., rows of a table). The second component is Index block that helps quickly locate data within the SSTable. It contains pointers (references) to the positions of certain keys in the data block.
- When reading data, the system uses the index to jump directly to the part of the SSTable that contains the requested key, making lookups faster.
- When a **read request** comes in The system first checks the **Memtable .** If the data is in the Memtable, it is returned immediately. If the data is not in the Memtable, the system checks the **SSTables** stored on disk. Because the data in each SSTable is sorted, the system can perform a **binary search** to locate the row efficiently.

SSTable and Tablet Server relationship

Both SSTable and Tablet server work closely together to manage and store data efficiently. A Tablet Server is responsible for managing data and handling requests (reading and writing) for a portion of the data. It manages tablets, which are smaller pieces (or sections) in the larger table that are ultimately saved on SSTable.
When a client writes new data, it first goes into memory (called a Memtable) in the tablet server. Over time, when this memory is full, the data is flushed to disk and stored as an SSTable. So in nutshell , So, the **SSTable** is where the tablet server stores the tablet's data **on disk**.

GFS (Google File System)

GFS is yet another proprietary system designed at google which is actually used by the Bigtable to store tablet data. It can store petabytes of data and handle millions of file operations (such as reads, writes, and appends) across a large cluster of servers. The salient points of GFS are :

1. **Redundancy**

GFS assumes all sorts of hardware failures (such as disk crashes, server failures, and network outages) are common in large-scale systems. So GFS saves each piece of data in **multiple replicas** (typically three) across different servers. This redundancy ensures that if one machine fails, the data can still be accessed from other replicas.

2. High Throughput

GFS is optimized for large, sequential read and write operations rather than small random access operations. This makes it particularly suitable for batch processing systems like MapReduce.

3. Handling large files

GFS is optimized for very large files, often gigabytes or even terabytes in size. It handles files that are too large to fit on a single machine by breaking them into chunks. Each chunk is stored across multiple machines, and GFS keeps track of which machines store each chunk.

GFS (contd)

4. Master-Slave Architecture
GFS uses a master-slave architecture where a single master server manages the metadata (information about the files and their locations), while the chunkservers store the actual data. The client requests data from the master, which then directs the client to the appropriate chunkserver to retrieve the file.

5. Efficient Append Operations:
GFS is designed to efficiently handle concurrent appends. This is useful in scenarios where multiple clients need to append data to the same file (e.g., log files). It ensures that all appends are atomic and that data is not lost or overwritten when multiple clients write to the same file simultaneously.

Tablet Server , SSTable & GFS relationship

GFS is the underlying storage system that manages the physical storage of data files, in ***SSTables*** created by ***tablet servers***. SSTables are stored on GFS. The GFS manages the physical storage of these files, ensuring that the SSTables are replicated and stored on multiple machines (replicas) to provide fault tolerance. In other words when the tablet server writes an SSTable to disk, it is actually writing the data to **GFS**,

Chubby Lock System

- Chubby is a distributed lock service developed by Google to manage coordination and synchronization between different systems and services in distributed environments.
- It provides exclusive locks or shared locks to prevent multiple clients from accessing the same resource simultaneously, ensuring data consistency and preventing race conditions.
- In Bigtable , Chubby is used for coordination between its tablet servers and the master server. Specifically, Chubby is used to elect the master server and to ensure that only one master server is active at a time.
- Tablet servers also use Chubby to register their presence. They create ephemeral files in Chubby to indicate their availability, and these files are automatically deleted if the tablet server crashes or becomes unreachable, allowing Bigtable’s master to detect and reassign the tablets it was managing.
- Tablet servers also use Chubby to register their presence. They create ephemeral files in Chubby to indicate their availability, and these files are automatically deleted if the tablet server crashes or becomes unreachable, allowing Bigtable’s master to detect and reassign the tablets it was managing.
- Chubby can be used for leader election. We know in any distributed system we must ensure that there is one and only one node that can act as a leader (e.g., to manage writes, perform coordination tasks, or manage resources). Nodes competing for leadership can request a lock in Chubby, and the one that acquires the lock becomes the leader.

Chubby Lock System (contd)

Working of Chubby in Big Table :
Chubby is used by Tablet server and the GFS of Big table.

Tablet server :
Bigtable uses Chubby for coordination between its **tablet servers** and the **master** server. Specifically, Chubby is used to elect the master server (which manages tablet assignments) and to ensure that only one master server is active at a time. When a Master Server starts up, it tries to acquire a Chubby lock that designates it as the leader. If the lock is available, the Master Server becomes the leader. If the lock is already held by another Master Server, the new instance has to wait. If the active Master Server fails (due to hardware failure or network partition), Chubby release the lock, and another Master Server can then take over by acquiring the lock, ensuring high availability and fault tolerance.

Google File System (GFS) :
Chubby is used to help coordinate chunk servers and the GFS master server. The master server can maintain exclusive control over metadata and manage chunk placement, replication, and recovery using Chubby's distributed locking. In GFS, chunk servers store the actual data (in the form of chunks). They frequently go online or offline due to scaling or failures. The GFS Master Server needs to know which chunk servers are available and what chunks they store. Chubby is used to register chunk servers and help the Master Server keep track of them.

Bloom Filter

Mathematically speaking , A Bloom filter is a space-efficient data structure used to test whether an element is a member of a set. It is designed to offer fast, space-efficient membership checks with false positives but no false negatives. This means a Bloom filter can confirm that an element may be in the set or definitely is not i.e it cannot guarantee membership with absolute certainty but can guarantee definetely
non membership.

- Bloom filters are commonly used in systems where fast lookups are required but exact accuracy is not necessary, and saving memory is a priority. They are particularly useful in distributed systems, databases, caches, and network security applications.
- A fixed-size bit array (e.g., 1,000 bits) initialized to all zeros.
- Multiple hash functions maps an input element to a position in the bit array.
- When we add an element to a Bloom filter , the element is passed through each of the hash functions. Each hash function generates an index corresponding to a position in the bit array. The bit at each of these positions in the bit array is set to 1.
- To check if an element is in the Bloom filter , the element is passed through the same hash functions to generate positions in the bit array. The bits at those positions are checked. If all the bits are set to 1, the element is **probably** in the set. If any of the bits are 0, the element is **definitely not** in the set.
- A Bloom filter can return false positives but never false negatives. If the Bloom filter indicates that an element is not in the set, it is definitely not in the set.

Bloom Filter (contd)

Bloom filters are highly space-efficient because they only require a fixed-size bit array and a few hash functions, making them suitable for use in memory-constrained environments.

Application of Bloom Filter :

Lets consider we are managing a large, distributed cache system (e.g., for a web service like a CDN or a database). Checking whether a resource is in the cache (i.e., membership testing) could be slow if we need to check multiple nodes, especially if the resource isn't in the cache at all. When a user requests a resource (like an image or webpage), the system can first check a Bloom filter to see if the resource is in the cache.

- If the Bloom filter says the resource is **not** in the cache (bits are 0 for at least one hash), then the system can immediately know to fetch the resource from the original source.
- If the Bloom filter says the resource **may** be in the cache (all bits are 1), the system performs a more expensive cache lookup.
- This optimization reduces unnecessary cache lookups for non-existent items, saving time and resources.

Email systems uses Bloom filters to detect whether a sender's email address has already been flagged as spam. If the Bloom filter indicates the address is probably in the spam list, the system can check more rigorously.

Thanks