Big Data Technological Stack

By-Shraddha Nikam

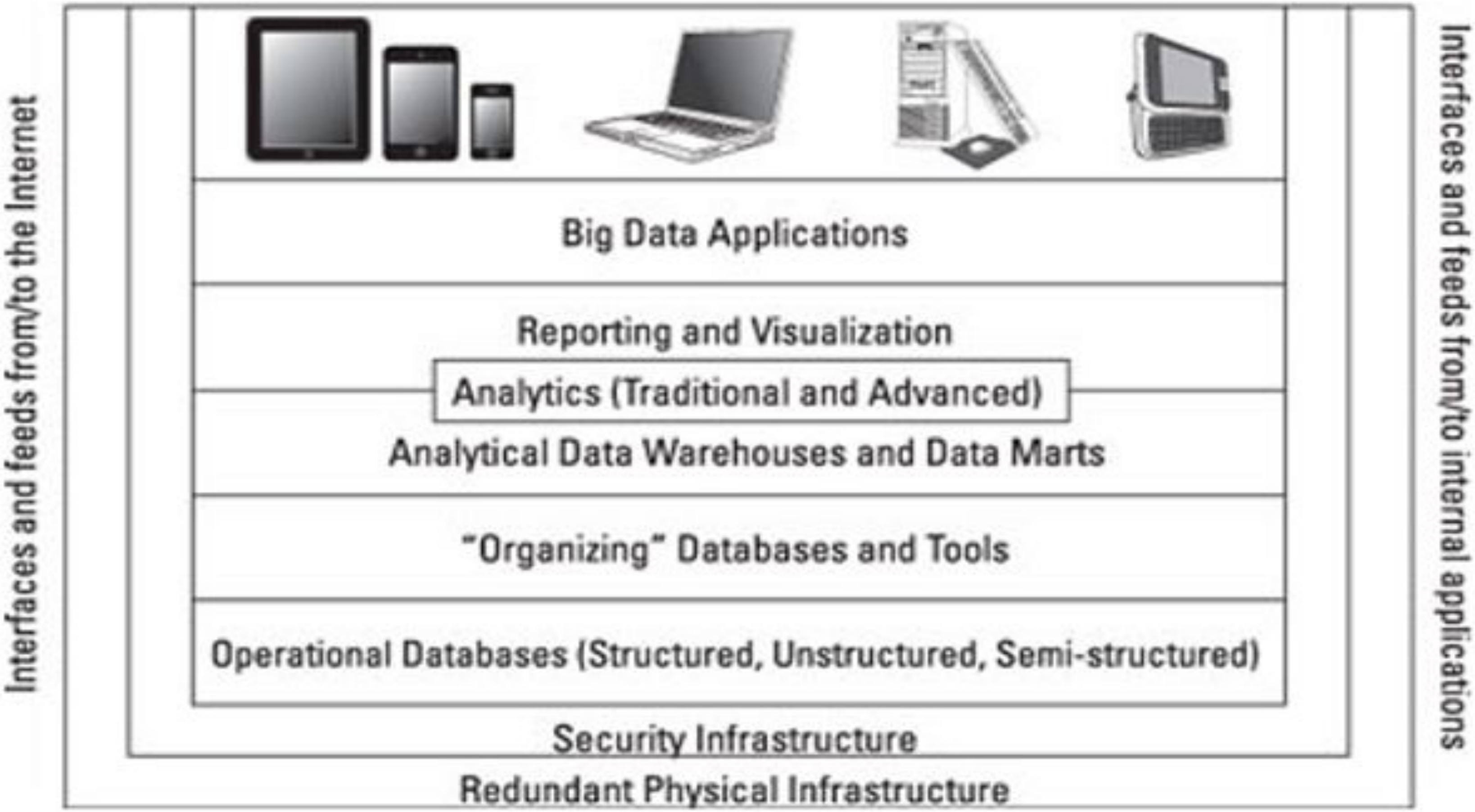
Why there is a need of Big Data Technology Stack?

When the data growth is hige or when we have variety of (data in different formats) then our traditional system are not able to store it or handle this huge data and that is why we need the Big Data Technology Stack.

The Big Data Stack makes use of Distribute Storage and Distributed Processing which makes it distributed in nature which is not the case in case of traditional systems.

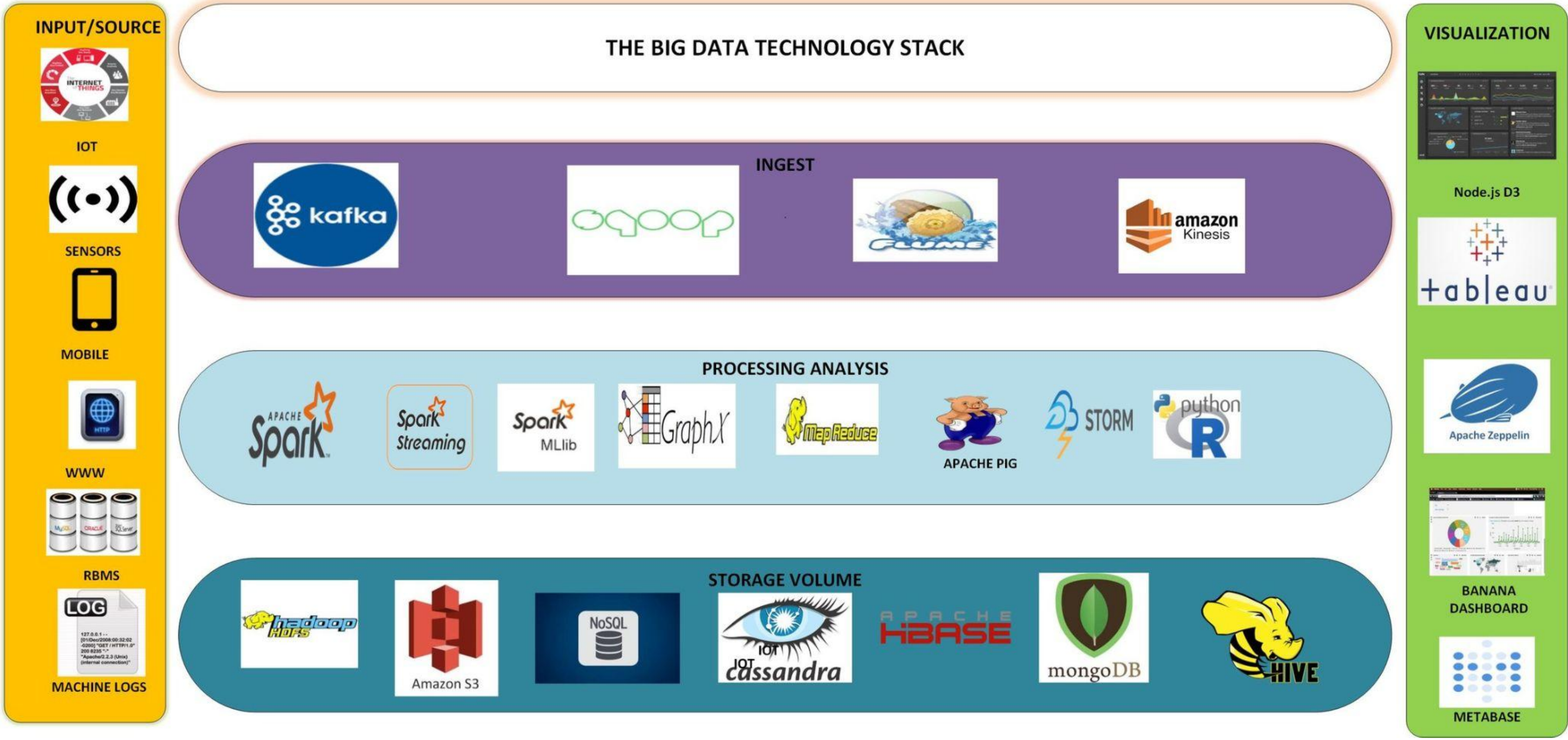
What is Big Data Stack?

Big data stack is a framework for the big data technologies that may satisfy the functional needs for big data projects is referred to as a technological stack. To understand big data, it helps to see how it stacks up that is, to set out the components of the architecture. Big data management architecture must incorporate a number of services that enable firms to make use of diverse data sources in a timely and effective manner


Interfaces and feeds from/to the Internet
Big Data Applications
Reporting and Visualization
Analytics (Traditional and Advanced)
Analytical Data Warehouses and Data Marts
"Organizing" Databases and Tools
Operational Databases (Structured, Unstructured, Semi-structured)
Security Infrastructure
Redundant Physical Infrastructure
Interfaces and feeds from/to internal applications

- **Security infrastructure** - The information about your constituents must be protected in order to comply with regulatory requirements as well as to protect their privacy.
- **Operational data sources** - A relational database was used to store highly structured data that was handled by the line of business. Operational data sources were used to store highly-structured data.
- **Organizing Databases and tools** - structured database and tools used to organize the data and process this.
- **Analytical Data warehouse -** The addition of an analytical data warehouse simplifies the data for the development of reports.
- **Reporting and visualization** - Enable the processing of data while providing a user-friendly depiction of the results.

THE BIG DATA TECHNOLOGY STACK



INPUT/SOURCE

IOT

SENSORS

MOBILE

WWW

RBMS

MACHINE LOGS

INGEST

PROCESSING ANALYSIS

APACHE PIG

STORAGE VOLUME

VISUALIZATION

Node.js D3

BANANA DASHBOARD

METABASE

Big Data Technology Stack can be divided into 4 layers

- Data Ingestion (Extract Transform Load)
- Data Processing / Analytics (Transformation, Machine Learning,Streaming)
- Data Storage (Data Center, Cluster Nodes)
- Data Visualization (Reporting , Alert, Recommendation, Dashboard, BI, Search,NoteBook)

Data Ingestion

Big data ingestion is about moving data - and especially unstructured data - from where it is originated, into a system where it can be stored and analyzed such as Hadoop. Data ingestion may be continuous or asynchronous, real-time or batched or both depending upon the characteristics of the source and the sink (destination). Real time ingestion with Apache flume.

Apache Flume: Flume is a tool/service/data ingestion mechanism for collecting aggregating and transporting large amounts of streaming data such as log files, events (etc...) from various sources to a centralized data store. It is reliable, fault tolerant, scalable, manageable, and customizable. It supports a large set of sources and destinations types. It supports multi-hop flows, fan-in fan-out flows, contextual routing, etc. Flume can be scaled horizontally.

Kafka: Apache Kafka is a distributed publish-subscribe messaging system and a robust queue that can handle a high volume of data and enables you to pass messages from one end-point to another. Kafka is suitable for both offline and online message consumption. Kafka messages are persisted on the disk and replicated within the cluster to prevent data loss. Kafka accepts each stream of events and stores it in an append-only queue called a **log**. Information in the log is immutable hence enables continuous, real-time processing and transformation of these streams and makes the results available system-wide.

Data Computation and Analytics

Once the data is available, the system can begin processing the data to surface actual information. The computation layer is perhaps the most diverse part of the system as the requirements and best approach can vary significantly depending on what type of insights desired and use case. Data can be processed either by Batch Process or Real time Streaming.

Hadoop MapReduce: The process involves breaking work up into smaller pieces, scheduling each piece on an individual machine, reshuffling the data based on the intermediate results, and then calculating and assembling the final result. These steps are often referred to individually as splitting, mapping, shuffling, reducing, and assembling, or collectively as a distributed map reduce algorithm.

Apache Spark: Apache Spark is a lightning-fast cluster computing technology, designed for fast computation. It is based on Hadoop MapReduce and it extends the MapReduce model to efficiently use it for more types of computations, which includes interactive queries and stream processing. The main feature of Spark is its **in-memory cluster computing** that increases the processing speed of an application. Spark has following features speed, support multiple languages and advanced Analytics. Spark has different components namely Spark Core, Spark SQL, SparkMLib, Spark Streaming, Graph X.

Apache Storm: is a distributed real-time big data-processing system. Storm is designed to process vast amount of data in a fault-tolerant and horizontal scalable method. It is a streaming data framework that has the capability of highest ingestion rates. It is simple and you can execute all kinds of manipulations on real-time data in parallel.

R: is a programming language and software environment for statistical analysis, graphics representation and reporting. R was created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, and is currently developed by the R Development Core Team. R has an effective data handling and storage facility, R provides a large, coherent and integrated collection of tools for data analysis. R is world's most widely used statistics programming language. It's the # 1 choice of data scientists and supported by a vibrant and talented community of contributors.

Python: is a high-level, interpreted, interactive and object-oriented scripting language. It is designed to be highly readable .It has fewer syntactical constructions than other languages.It has become a common language for data science and production of web based analytics. It is second most used language for data scientists after **R**. My experience is python compare to other languages has been good. It has rich set of libaries and you write few codes to get the job done. I personally fall in love with **Python**.

Data Storage

The ingestion processes typically hand the data off to the components that manage storage, so that it can be reliably persisted to disk. While this seems like it would be a simple operation, the volume of incoming data, the requirements for availability, and the distributed computing layer make more complex storage systems necessary.The following are some data storage technology in Big Data ecosystem.

Hadoop: is an open-source software framework for storing data and running applications on clusters of commodity hardware. It provides massive storage for any kind of data, enormous processing power and the ability to handle virtually limitless concurrent tasks or jobs.

HBase: is a distributed column-oriented database built on top of the Hadoop file system. It is an open-source project and is horizontally scalable. HBase is a data model that is similar to Google’s big table designed to provide quick random access to huge amounts of structured data. It leverages the fault tolerance provided by the Hadoop File System (HDFS). It is a part of the Hadoop ecosystem that provides random real-time read/write access to data in the Hadoop File System.

Hive: is a data warehouse infrastructure tool to process structured data in Hadoop. It resides on top of Hadoop to summarize Big Data, and makes querying and analyzing easy.Initially Hive was developed by Facebook, later the Apache Software Foundation took it up and developed it further as an open source under the name Apache Hive. It is used by different companies. For example, Amazon uses it in Amazon Elastic MapReduce.

Cassandra: is a highly scalable, high-performance distributed database designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure. It is a type of NoSQL database.It is scalable, fault-tolerant, and consistent.t is a column-oriented database

MongoDB: is a cross-platform, document oriented database that provides, high performance, high availability, and easy scalability. MongoDB works on concept of collection and document.

Amozon S3: is an online storage facility. It is cheap, fast and easy to setup. And since it’s a service provided by e-commerce giant Amazon, you can be rest-assured whatever you stored at S3 is secured. It is not and open source. It is base on software as service (SAS licence model) offering. You must pay to use. No free lunch in freetown.

Data Visualization

Due to the type of information being processed in big data systems, recognizing trends or changes in data over time is often more important than the values themselves. Visualizing data is one of the most useful ways to spot trends and make sense of a large number of data points.

Real-time processing is frequently used to visualize application and server metrics. The data changes frequently and large deltas in the metrics typically indicate significant impacts on the health of the systems or organization. Some of popular visualization tools are **Tabulue**, Metabase, **Banana** a fork from **kibana** or do it your self custom dashboard with **Node.js** and **D3**.Another visualization technology typically used for interactive data science work is a data "notebook". These projects allow for interactive exploration and visualization of the data in a format conducive to sharing, presenting, or collaborating. Popular examples of this type of visualization interface are **Jupyter** Notebook and Apache **Zeppelin.**

Overview of big data distribution packages

There are several popular big data distribution packages available that facilitate the management, processing, and analysis of large datasets. Some of the prominent ones include:

Apache Hadoop: Apache Hadoop is an open-source framework for distributed storage and processing of large datasets across clusters of computers. It includes modules like Hadoop Distributed File System (HDFS) for storage and MapReduce for processing.

Apache Spark: Apache Spark is an open-source cluster computing framework that provides an interface for programming entire clusters with implicit data parallelism and fault tolerance. Spark offers libraries for various tasks such as SQL, streaming data, machine learning, and graph processing.

Apache Flink: Apache Flink is an open-source stream processing framework for distributed, high-performing, and fault-tolerant stream processing of big data. It supports batch processing as well as event-driven applications.

Apache Kafka: Apache Kafka is an open-source distributed event streaming platform used for building real-time data pipelines and streaming applications. It is often used as a message broker for handling large volumes of data streams.

Apache Cassandra: Apache Cassandra is an open-source distributed NoSQL database management system designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure.

Hortonworks Data Platform (HDP): HDP is an open-source data management platform based on Apache Hadoop. It includes various components such as HDFS, YARN, Hive, HBase, Spark, and others for processing and analyzing big data.

Cloudera Data Platform (CDP): Cloudera Data Platform is a comprehensive big data management and analytics platform that includes various components for data ingestion, storage, processing, and analysis. It supports both on-premises and cloud deployments.

MapR Data Platform: MapR Data Platform is a converged data platform that integrates Hadoop and Spark with global event streaming, real-time database capabilities, and enterprise storage. It offers a comprehensive set of tools for managing and analyzing big data.

These packages provide a range of tools and capabilities for handling big data, and the choice of the most suitable one depends on specific requirements, such as scalability, performance, real-time processing needs, and ease of integration with existing systems.

Big Data Platforms

WHAT IS A BIG DATA PLATFORM?

A big data platform acts as an organized storage medium for large amounts of data. Big data platforms utilize a combination of data management hardware and software tools to store aggregated data sets, usually onto the cloud.

A big data platform works to wrangle this amount of information, storing it in a manner that is organized and understandable enough to extract useful insights. Big data platforms utilize a combination of data management hardware and software tools to aggregate data on a massive scale, usually onto the cloud.

Benefits of Big Data Platforms

How does Netflix or Spotify know exactly what you want to stream next? This is due in a large part to big data platforms working behind the scenes.

Understanding big data has become an asset in nearly every industry, ranging from healthcare to retail and beyond. Companies increasingly rely on these platforms to collect loads of data and turn them into categorized, actionable business decisions. This helps firms get a better view of their customers, target audiences, discover new markets and make predictions about future steps.

Big data platforms provide infrastructure and tools for storing, processing, and analyzing large and complex datasets. Here are some of the prominent big data platforms:

Amazon Web Services (AWS):

- Amazon EMR (Elastic MapReduce): Managed Hadoop framework on AWS.
- Amazon Redshift: Fully managed data warehouse service.
- Amazon Athena: Interactive query service for analyzing data in Amazon S3.
- Amazon Kinesis: Real-time data streaming platform.

Microsoft Azure:

- Azure HDInsight: Managed Apache Hadoop, Spark, HBase, and Storm service.
- Azure Synapse Analytics: Enterprise analytics service that brings together big data and data warehousing.
- Azure Databricks: Unified analytics platform for big data and machine learning.
- Azure Stream Analytics: Real-time data streaming and analytics service.

Google Cloud Platform (GCP):

- Google BigQuery: Fully managed serverless data warehouse for analytics.
- Google Cloud Dataproc: Managed Apache Spark and Hadoop service.
- Google Cloud Dataflow: Stream and batch processing service for real-time data analytics.
- Google Cloud Pub/Sub: Real-time messaging service for event-driven systems.

Cloudera Data Platform (CDP): Comprehensive big data platform that includes various components for data management, processing, and analytics, both on-premises and in the cloud.

Hortonworks Data Platform (HDP): Open-source big data platform based on Apache Hadoop, providing various components for managing and analyzing big data.

MapR Data Platform: Converged data platform that integrates Hadoop, Spark, global event streaming, and real-time database capabilities.

IBM Cloud Pak for Data: Integrated data and AI platform that enables organizations to collect, organize, and analyze data at scale.

Databricks: Unified analytics platform that combines data engineering, data science, and analytics in the cloud, built on Apache Spark.

Snowflake: Cloud-based data warehousing platform that allows users to store and analyze data using SQL queries.

MongoDB Atlas: Fully managed cloud database service for MongoDB, suitable for storing and analyzing large volumes of unstructured data.

These platforms offer a range of services and tools to meet the needs of different use cases, such as data warehousing, real-time analytics, machine learning, and batch processing. The choice of platform depends on factors such as scalability, performance, cost, and integration requirements.

What is Big Data Storage?

Big Data Storage is a new technology poised to revolutionize how we store data. The technology was first developed in the early 2000s when companies were faced with storing massive amounts of data that they could not keep on their servers.

The problem was that traditional storage methods couldn't handle storing all this data, so companies had to look for new ways to keep it. That's when Big Data Storage came into being. It's a way for companies to store large amounts of data without worrying about running out of space.

Big Data Storage Challenges

Big data is a hot topic in IT. Every month, more companies are adopting it to help them improve their businesses. But with any new technology comes challenges and questions, and big data is no exception.

The first challenge is how much storage you'll need for your extensive data system. If you're going to store large amounts of information about your customers and their behavior, you'll need a lot of space for that data to live.

It's not uncommon for large companies like Google or Facebook to have petabytes (1 million gigabytes) of storage explicitly dedicated to their big data needs, and that's only one company!

Another challenge with big data is how quickly it grows. Companies are constantly gathering new types of information about their customer's habits and preferences, and they're looking at ways they can use this information to improve their products or services.

As a result, big data systems will continue growing exponentially until something stops them. It means it's essential for companies who want to use this technology effectively to plan how they'll deal with it later on down the road when it becomes too much for them alone!

Data Storage Methods

Warehouse and cloud storage are two of the most popular options for storing big data. Warehouse storage is typically done on-site, while cloud storage involves storing your data offsite in a secure location.

Warehouse Storage

Warehouse storage is one of the more common ways to store large amounts of data, but it has drawbacks. For example, if you need immediate access to your data and want to avoid delays or problems accessing it over the internet, there might be better options than this. Also, warehouse storage can be expensive if you're looking for long-term contracts or need extra personnel to manage your warehouse space.

Cloud Storage

Cloud storage is an increasingly popular option since it's easier than ever to use this method, thanks to advancements in technology such as Amazon Web Services (AWS). With AWS, you can store unlimited data without worrying about how much space each file takes up on their servers. They'll automatically compress them before sending them over, so they take up less space overall!

Big Data Storage Platforms

For large-scale data storage in big data environments, several platforms are specifically designed to handle massive volumes of data efficiently. Here are some popular ones:

> **Hadoop Distributed File System (HDFS):** HDFS is a distributed file system designed to store large datasets reliably across commodity hardware. It is the primary storage system used by the Apache Hadoop framework.
>
> **Amazon Simple Storage Service (S3):** Amazon S3 is an object storage service that provides scalability, high availability, and security for storing large amounts of data in the cloud. It is widely used for big data storage and analytics on the AWS platform.

Google Cloud Storage: Google Cloud Storage offers a scalable and highly available object storage solution for storing large datasets in the cloud. It provides features such as global storage buckets, versioning, and lifecycle management.

Microsoft Azure Blob Storage: Azure Blob Storage is a massively scalable object storage service that is part of the Azure cloud platform. It offers tiered storage options, encryption, and integration with other Azure services for big data processing and analytics.

Ceph: Ceph is an open-source distributed storage system that provides object, block, and file storage in a unified platform. It is designed for scalability, reliability, and performance, making it suitable for large-scale data storage.

Apache Cassandra: Cassandra is a distributed NoSQL database designed for high availability and scalability. It is optimized for storing large volumes of data across multiple nodes and is often used for time-series data and real-time analytics.

Apache HBase: HBase is a distributed, column-oriented database built on top of Hadoop and HDFS. It provides random, real-time read/write access to large datasets and is commonly used for storing semi-structured or sparse data.

IBM Cloud Object Storage: IBM Cloud Object Storage is a scalable and secure object storage service that offers flexible storage options for big data workloads. It is designed for high throughput and low-latency access to data in the cloud.

MapR-FS: MapR-FS is a distributed file system that provides high-performance storage for big data applications. It offers features such as snapshots, data mirroring, and POSIX compatibility for seamless integration with existing workflows.

SwiftStack: SwiftStack is a software-defined storage platform that provides scalable object storage for big data environments. It is built on top of the OpenStack Swift project and offers features such as multi-tenancy, data encryption, and policy-based data management.

These storage platforms offer various features and capabilities to meet the storage needs of big data applications, including scalability, reliability, performance, and cost-effectiveness. The choice of platform depends on factors such as the specific requirements of the application, the volume and type of data, and the desired level of integration with other big data technologies.

CAP Theorem

The CAP theorem, also known as Brewer's theorem, is a fundamental concept in distributed computing that states that it's impossible for a distributed data store to simultaneously provide all three of the following guarantees:

> **Consistency (C):** Every read receives the most recent write or an error. In other words, all nodes in the system have the same data at the same time, regardless of which node is queried.
> **Availability (A):** Every request receives a response, without the guarantee that it contains the most recent write. The system remains operational despite node failures.

Partition tolerance (P): The system continues to operate despite network partitions (communication failures) that may occur between nodes, causing them to be unable to communicate with each other.

In the context of big data systems, the CAP theorem is particularly relevant because many of these systems, such as distributed databases and storage platforms, operate across multiple nodes and handle large volumes of data.

Eventual Consistency

Eventual consistency is a consistency model employed in distributed systems, including many big data platforms, that allows data replicas to diverge temporarily and then converge to a consistent state over time. In an eventually consistent system:

> **Updates Propagate Eventually:** When a change is made to the data in one replica, that change is eventually propagated to all other replicas in the system. However, there's no guarantee of how long it will take for this propagation to occur.
>
> **Reads May Be Inconsistent:** During the period of inconsistency, different replicas may return different values for the same data item. This means that, at any given moment, different clients may observe different states of the data

- **Convergence Ensures Consistency:** Despite the temporary inconsistency, eventual consistency guarantees that all replicas will eventually converge to a consistent state where all clients will observe the same data.

In the context of big data systems, eventual consistency is often preferred over strong consistency because it allows for higher availability and better fault tolerance. Here's how eventual consistency is typically implemented and used in big data platforms:

- **Distributed Databases:** Many distributed databases used in big data systems, such as Apache Cassandra and Amazon DynamoDB, employ eventual consistency to ensure high availability and fault tolerance. These systems use techniques like hinted handoff, read repair, and anti-entropy mechanisms to achieve eventual consistency.

Distributed File Systems: Big data storage platforms like Apache Hadoop Distributed File System (HDFS) may also rely on eventual consistency to replicate data across multiple nodes. Updates made to one replica are eventually propagated to others, ensuring data availability and fault tolerance.

Stream Processing: Eventual consistency is also relevant in stream processing systems like Apache Kafka, where data streams are distributed across multiple partitions. Consumers may observe data in different states at different times, but eventual consistency ensures that all consumers eventually see the same data.

Data Replication: In big data analytics, eventual consistency is often used in data replication processes to ensure that data warehouses, data lakes, and other analytical systems have consistent copies of data, even when updates occur asynchronously across distributed sources.

Consistency Trade-Offs

In big data analytics, there are several consistency tradeoffs that organizations must consider when designing and implementing their systems. These tradeoffs often involve balancing data consistency, availability, and latency to meet the requirements of specific use cases and workloads. Here are some common consistency tradeoffs in big data analytics:

Strong Consistency vs. Eventual Consistency:

- Strong consistency ensures that all reads and writes to the data store reflect the most recent state of the data. Achieving strong consistency often requires coordination and synchronization across distributed nodes, which can impact latency and availability.
- Eventual consistency, on the other hand, allows for temporary inconsistencies between replicas, with the guarantee that all replicas will eventually converge to a consistent state. This approach can improve availability and latency but may lead to temporary data inconsistencies.

Consistency vs. Availability:

- Consistency and availability are often considered opposing goals in distributed systems, including big data analytics. Strict consistency guarantees can lead to increased latency and decreased availability due to the need for coordination among distributed nodes.
- Relaxing consistency requirements in favor of availability can improve system responsiveness and fault tolerance, but it may result in eventual consistency or even data divergence in some cases.

Synchronous vs. Asynchronous Replication:

- Synchronous replication ensures that data changes are propagated to all replicas before acknowledging a write operation. This approach guarantees strong consistency but can introduce latency and reduce availability, especially in scenarios with high network latency or node failures.
- Asynchronous replication allows data changes to be propagated to replicas asynchronously, which can improve availability and reduce latency. However, it may lead to temporary data inconsistencies between replicas until updates are synchronized.

Consistency Models in Distributed Databases:

- Distributed databases used in big data analytics often offer various consistency models, such as strong consistency, eventual consistency, and various levels of consistency in between (e.g., causal consistency, session consistency). Organizations must evaluate these consistency models based on their specific requirements for data consistency, availability, and latency.

Data Partitioning and Sharding:

- Partitioning and sharding data across distributed nodes can improve scalability and performance in big data analytics systems. However, it can also introduce consistency challenges, such as ensuring consistent data distribution, managing data locality, and handling partitioning-related data skew.

Data Replication Strategies:

- Choosing an appropriate data replication strategy is crucial for maintaining consistency and availability in big data analytics. Organizations must consider factors such as replication topology (e.g., master-slave, multi-master), replication protocols (e.g., gossip-based, quorum-based), and conflict resolution mechanisms to achieve the desired balance between consistency and availability.

Overall, achieving the right consistency tradeoffs in big data analytics requires careful consideration of the specific requirements, workload characteristics, and architectural constraints of the system. Organizations must strike a balance between consistency, availability, and latency to ensure that their big data analytics platforms meet performance goals while providing accurate and timely insights.

ACID Properties

A **transaction** is a single logical unit of work that accesses and possibly modifies the contents of a database. Transactions access data using read and write operations.

In order to maintain consistency in a database, before and after the transaction, certain properties are followed. These are called **ACID** properties.

Atomicity:

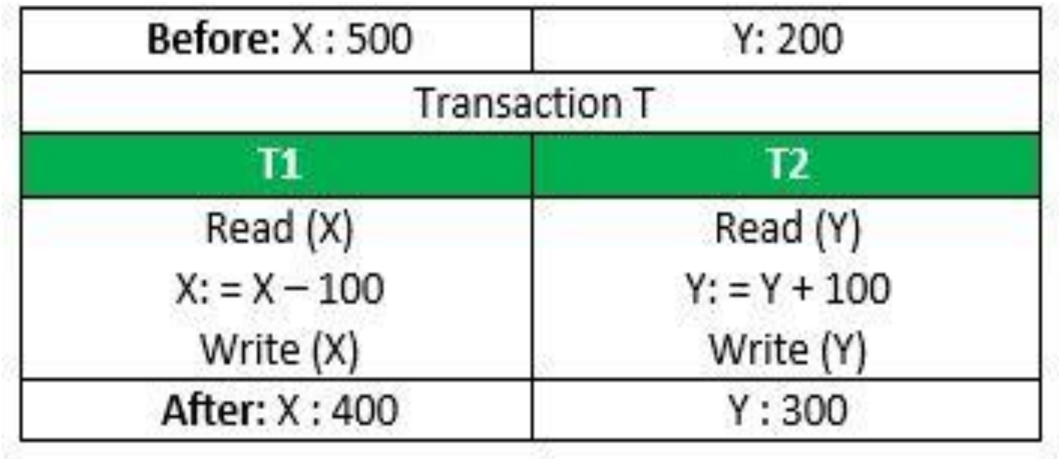
By this, we mean that either the entire transaction takes place at once or doesn’t happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves the following two operations.

—**Abort**: If a transaction aborts, changes made to the database are not visible.

—**Commit**: If a transaction commits, changes made are visible.

Atomicity is also known as the ‘All or nothing rule’.

Consider the following transaction **T** consisting of **T1** and **T2**: Transfer of 100 from account **X** to account **Y**.



Before: X : 500	Y: 200
Transaction T	
T1	T2
Read (X) X: = X − 100 Write (X)	Read (Y) Y: = Y + 100 Write (Y)
After: X : 400	Y : 300

If the transaction fails after completion of **T1** but before completion of **T2**.(say, after **write(X)** but before **write(Y)**), then the amount has been deducted from **X** but not added to **Y**. This results in an inconsistent database state. Therefore, the transaction must be executed in its entirety in order to ensure the correctness of the database state.

Consistency:

This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to the correctness of a database. Referring to the example above,

The total amount before and after the transaction must be maintained.

Total **before T** occurs = **500 + 200 = 700**.

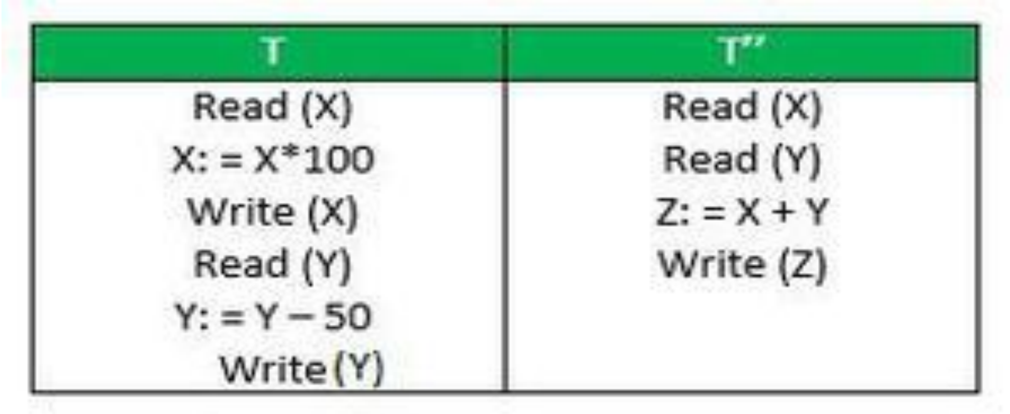
Total **after T occurs** = **400 + 300 = 700**.

Therefore, the database is **consistent**. Inconsistency occurs in case **T1** completes but **T2** fails. As a result, T is incomplete.

Isolation:

This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of the database state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed. This property ensures that the execution of transactions concurrently will result in a state that is equivalent to a state achieved these were executed serially in some order.

Let **X**= 500, **Y** = 500. Consider two transactions **T** and **T”.**



T	T”
Read (X)	Read (X)
X: = X*100	Read (Y)
Write (X)	Z: = X + Y
Read (Y)	Write (Z)
Y: = Y – 50	
Write (Y)	

Suppose **T** has been executed till **Read (Y)** and then **T”** starts. As a result, interleaving of operations takes place due to which **T”** reads the correct value of **X** but the incorrect value of **Y** and sum computed by

T”: (X+Y = 50, 000+500=50, 500)

is thus not consistent with the sum at end of the transaction:

T: (X+Y = 50, 000 + 450 = 50, 450).

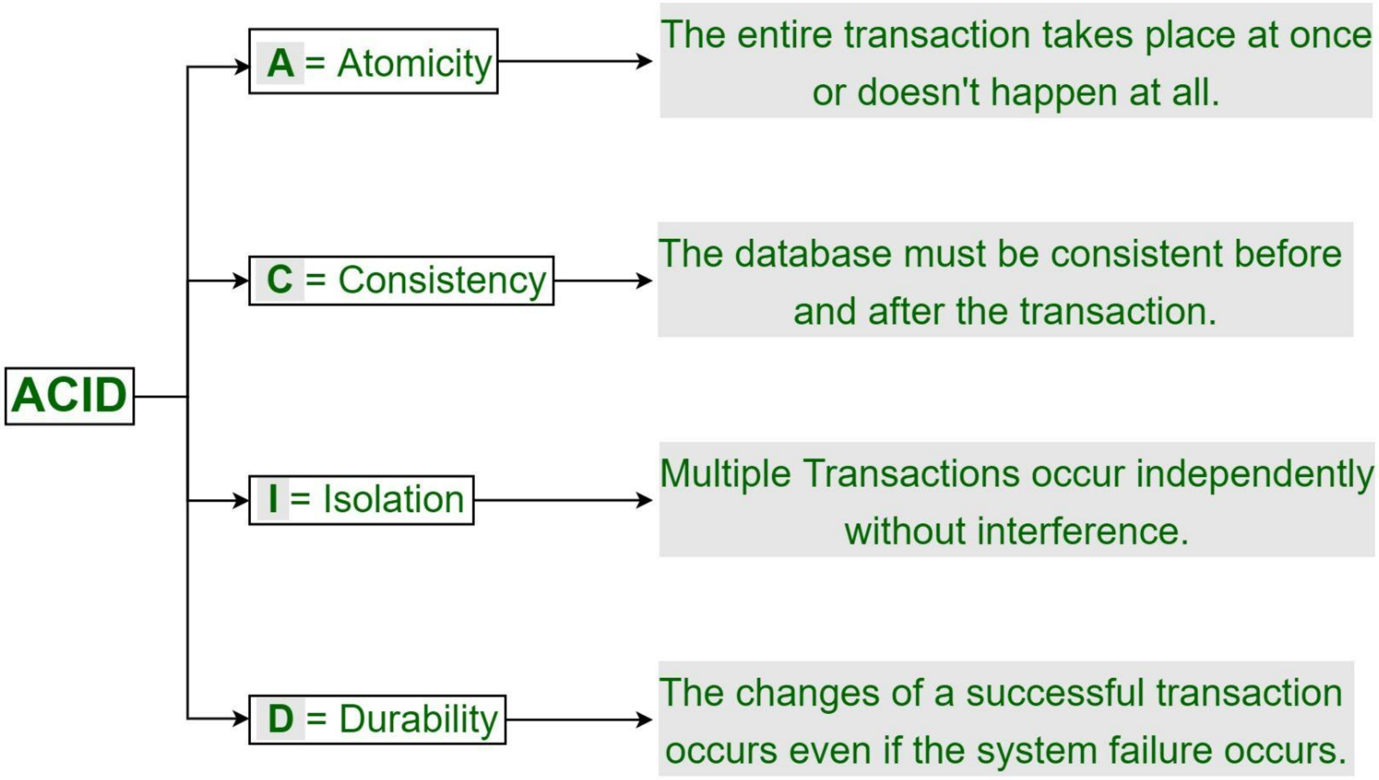
This results in database inconsistency, due to a loss of 50 units. Hence, transactions must take place in isolation and changes should be visible only after they have been made to the main memory.

Durability:

This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if a system failure occurs. These updates now become permanent and are stored in non-volatile memory. The effects of the transaction, thus, are never lost.

The **ACID** properties, in totality, provide a mechanism to ensure the correctness and consistency of a database in a way such that each transaction is a group of operations that acts as a single unit, produces consistent results, acts in isolation from other operations, and updates that it makes are durably stored.

ACID Properties in DBMS



ACID

A = Atomicity

The entire transaction takes place at once or doesn't happen at all.

C = Consistency

The database must be consistent before and after the transaction.

I = Isolation

Multiple Transactions occur independently without interference.

D = Durability

The changes of a successful transaction occurs even if the system failure occurs.

Advantages of ACID Properties in DBMS:

1. **Data Consistency:** ACID properties ensure that the data remains consistent and accurate after any transaction execution.
2. **Data Integrity:** ACID properties maintain the integrity of the data by ensuring that any changes to the database are permanent and cannot be lost.
3. **Concurrency Control:** ACID properties help to manage multiple transactions occurring concurrently by preventing interference between them.
4. **Recovery:** ACID properties ensure that in case of any failure or crash, the system can recover the data up to the point of failure or crash.

Disadvantages of ACID Properties in DBMS:

1. **Performance:** The ACID properties can cause a performance overhead in the system, as they require additional processing to ensure data consistency and integrity.
2. **Scalability:** The ACID properties may cause scalability issues in large distributed systems where multiple transactions occur concurrently.
3. **Complexity:** Implementing the ACID properties can increase the complexity of the system and require significant expertise and resources.

Overall, the advantages of ACID properties in DBMS outweigh the disadvantages. They provide a reliable and consistent approach to data

Application of ACID Model:

The ACID model is ideal for systems that require transactional integrity, such as banking systems, financial systems, and airline reservation systems. These systems need to ensure that data remains accurate and consistent, and the ACID model is well-suited to meet these requirements.

BASE Properties

The BASE properties of a database management system are a set of principles that guide the design and operation of modern databases.

The acronym BASE stands for Basically Available, Soft State, and Eventual Consistency.

Basically Available:

This property refers to the fact that the database system should always be available to respond to user requests, even if it cannot guarantee immediate access to all data. The database may experience brief periods of unavailability, but it should be designed to minimize downtime and provide quick recovery from failures.

Soft State:

This property refers to the fact that the state of the database can change over time, even without any explicit user intervention. This can happen due to the effects of background processes, updates to data, and other factors. The database should be designed to handle this change gracefully, and ensure that it does not lead to data corruption or loss.

Eventual Consistency:

This property refers to the eventual consistency of data in the database, despite changes over time. In other words, the database should eventually converge to a consistent state, even if it takes some time for all updates to propagate and be reflected in the data. This is in contrast to the immediate consistency required by traditional ACID-compliant databases.

Uses of BASE Databases

BASE databases are used in modern, highly-available, and scalable systems that handle large amounts of data. Examples of such systems include online shopping websites, social media platforms, and cloud-based services.

Advantages of BASE Model:

1. High availability
2. Good scalability
3. Can handle large volumes of data and high concurrency
4. Allows for flexibility in handling data

Disadvantages of BASE Model:

1. Data may be inconsistent for a short period of time
2. Can lead to conflicts in data
3. Can be difficult to maintain

Application of BASE Model:

The BASE model is ideal for systems that require high availability, such as social media platforms and e-commerce websites. These systems need to be able to handle large volumes of data and a high degree of concurrency, and the BASE model is well-suited to meet these requirements.

Difference between Base Properties and ACID Properties

ACID	BASE
ACID (Atomicity, Consistency, Isolation, Durability) is a set of properties that guarantee the integrity and consistency of data in a traditional database.	The BASE properties are a more relaxed version of ACID that trade off some consistency guarantees for greater scalability and availability.
The primary difference between the two is that ACID requires immediate consistency,	while BASE only requires eventual consistency.
ACID is better suited to traditional transactional databases.	The BASE is more suitable for use in large-scale, highly-available systems,

What is Apache ZooKeeper?

Zookeeper is a distributed, open-source coordination service for distributed applications. It exposes a simple set of primitives to implement higher-level services for synchronization, configuration maintenance, and group and naming.

In a distributed system, there are multiple nodes or machines that need to communicate with each other and coordinate their actions.

ZooKeeper provides a way to ensure that these nodes are aware of each other and can coordinate their actions.

It does this by maintaining a hierarchical tree of data nodes called “**Znodes**“, which can be used to store and retrieve data and maintain state information.

ZooKeeper provides a set of primitives, such as locks, barriers, and queues, that can be used to coordinate the actions of nodes in a distributed system.

It also provides features such as leader election, failover, and recovery, which can help ensure that the system is resilient to failures.

ZooKeeper is widely used in distributed systems such as Hadoop, Kafka, and HBase, and it has become an essential component of many distributed applications.

Why do we need it?

- **Coordination services**: The integration/communication of services in a distributed environment.
- Coordination services are complex to get right. They are especially prone to errors such as race conditions and deadlock.
- **Race condition-**Two or more systems trying to perform some task.
- **Deadlocks**– Two or more operations are waiting for each other.
- To make the coordination between distributed environments easy, developers came up with an idea called zookeeper so that they don’t have to relieve distributed applications of the responsibility of implementing coordination services from scratch.

What is distributed system?

- Multiple computer systems working on a single problem.
- It is a network that consists of autonomous computers that are connected using distributed middleware.
- **Key Features**: Concurrent, resource sharing, independent, global, greater fault tolerance, and price/performance ratio is much better.
- **Key Goal**s: Transparency, Reliability, Performance, Scalability.
- **Challenges**: Security, Fault, Coordination, and resource sharing.

Coordination Challenge

- Why is coordination in a distributed system the hard problem?
- Coordination or configuration management for a distributed application that has many systems.
- Master Node where the cluster data is stored.
- Worker nodes or slave nodes get the data from this master node.
- single point of failure.
- synchronization is not easy.
- Careful design and implementation are needed.

Apache Zookeeper

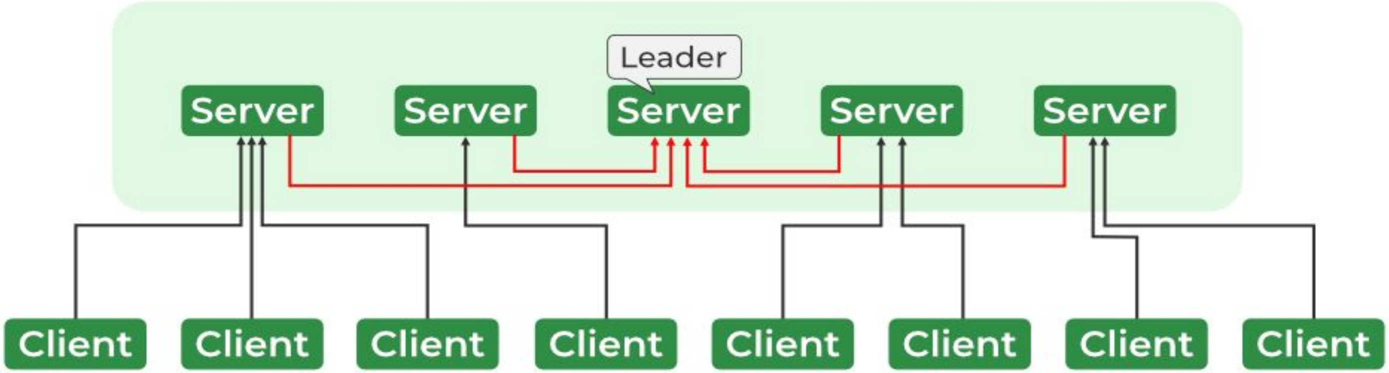
Apache Zookeeper is a distributed, open-source coordination service for distributed systems. It provides a central place for distributed applications to store data, communicate with one another, and coordinate activities.

Zookeeper is used in distributed systems to coordinate distributed processes and services. It provides a simple, tree-structured data model, a simple API, and a distributed protocol to ensure data consistency and availability.

Zookeeper is designed to be highly reliable and fault-tolerant, and it can handle high levels of read and write throughput.

Zookeeper is implemented in Java and is widely used in distributed systems, particularly in the Hadoop ecosystem. It is an Apache Software Foundation project and is released under the Apache License 2.0.

Architecture of Zookeeper



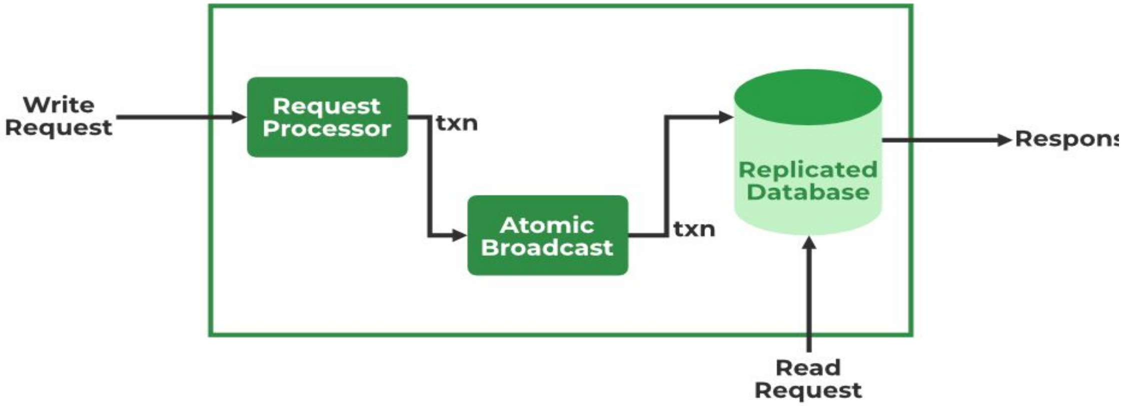
Zookeeper Services

The ZooKeeper architecture consists of a hierarchy of nodes called znodes, organized in a tree-like structure. Each znode can store data and has a set of permissions that control access to the znode.

The znodes are organized in a hierarchical namespace, similar to a file system. At the root of the hierarchy is the root znode, and all other znodes are children of the root znode.

The hierarchy is similar to a file system hierarchy, where each znode can have children and grandchildren, and so on.

Important Components

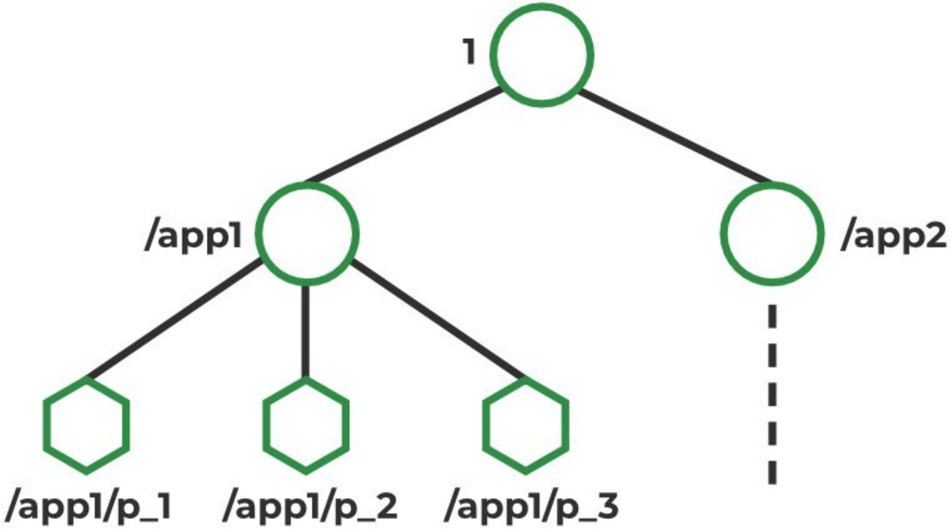


- **Leader & Follower**
- **Request Processor** – Active in Leader Node and is responsible for processing write requests. After processing, it sends changes to the follower nodes
- **Atomic Broadcast** – Present in both Leader Node and Follower Nodes. It is responsible for sending the changes to other Nodes.
- **In-memory Databases** (Replicated Databases)-It is responsible for storing the data in the zookeeper. Every node contains its own databases. Data is also written to the file system providing recoverability in case of any problems with the cluster.

Other Components

- **Client** – One of the nodes in our distributed application cluster. Access information from the server. Every client sends a message to the server to let the server know that client is alive.
- **Server**– Provides all the services to the client. Gives acknowledgment to the client.
- **Ensemble**– Group of Zookeeper servers. The minimum number of nodes that are required to form an ensemble is 3.

Zookeeper Data Model



ZooKeeper data model

In Zookeeper, data is stored in a hierarchical namespace, similar to a file system. Each node in the namespace is called a Znode, and it can store data and have children.

Znodes are similar to files and directories in a file system. Zookeeper provides a simple API for creating, reading, writing, and deleting Znodes.

It also provides mechanisms for detecting changes to the data stored in Znodes, such as watches and triggers.

Znodes maintain a stat structure that includes: Version number, ACL, Timestamp, Data Length

Types of Znodes:

- **Persistence**: Alive until they’re explicitly deleted.
- **Ephemeral**: Active until the client connection is alive.
- **Sequential**: Either persistent or ephemeral.

Why do we need ZooKeeper in the Hadoop?

Zookeeper is used to manage and coordinate the nodes in a Hadoop cluster, including the NameNode, DataNode, and ResourceManager. In a Hadoop cluster, Zookeeper helps to:

- Maintain configuration information: Zookeeper stores the configuration information for the Hadoop cluster, including the location of the NameNode, DataNode, and ResourceManager.
- Manage the state of the cluster: Zookeeper tracks the state of the nodes in the Hadoop cluster and can be used to detect when a node has failed or become unavailable.

- Coordinate distributed processes: Zookeeper can be used to coordinate distributed processes, such as job scheduling and resource allocation, across the nodes in a Hadoop cluster.

Zookeeper helps to ensure the availability and reliability of a Hadoop cluster by providing a central coordination service for the nodes in the cluster.

How ZooKeeper in Hadoop Works?

ZooKeeper operates as a distributed file system and exposes a simple set of APIs that enable clients to read and write data to the file system. It stores its data in a tree-like structure called a znode, which can be thought of as a file or a directory in a traditional file system.

ZooKeeper uses a consensus algorithm to ensure that all of its servers have a consistent view of the data stored in the Znodes. This means that if a client writes data to a znode, that data will be replicated to all of the other servers in the ZooKeeper ensemble.

One important feature of ZooKeeper is its ability to support the notion of a “watch.” A watch allows a client to register for notifications when the data stored in a znode changes. This can be useful for monitoring changes to the data stored in ZooKeeper and reacting to those changes in a distributed system.

In Hadoop, ZooKeeper is used for a variety of purposes, including:

- **Storing configuration information:** ZooKeeper is used to store configuration information that is shared by multiple Hadoop components. For example, it might be used to store the locations of NameNodes in a Hadoop cluster or the addresses of JobTracker nodes.
- **Providing distributed synchronization:** ZooKeeper is used to coordinate the activities of various Hadoop components and ensure that they are working together in a consistent manner. For example, it might be used to ensure that only one NameNode is active at a time in a Hadoop cluster.

- **Maintaining naming:** ZooKeeper is used to maintain a centralized naming service for Hadoop components. This can be useful for identifying and locating resources in a distributed system.

ZooKeeper is an essential component of Hadoop and plays a crucial role in coordinating the activity of its various subcomponents.

Reading and Writing in Apache Zookeeper

ZooKeeper provides a simple and reliable interface for reading and writing data. The data is stored in a hierarchical namespace, similar to a file system, with nodes called znodes. Each znode can store data and have children znodes. ZooKeeper clients can read and write data to these znodes by using the getData() and setData() methods, respectively. Here is an example of reading and writing data using the ZooKeeper Java API:

```
// Connect to the ZooKeeper ensemble
ZooKeeper zk = new ZooKeeper("localhost:2181", 3000, null);

// Write data to the znode "/myZnode"
String path = "/myZnode";
String data = "hello world";
zk.create(path, data.getBytes(), Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);

// Read data from the znode "/myZnode"
byte[] bytes = zk.getData(path, false, null);
String readData = new String(bytes);

// Prints "hello world"
System.out.println(readData);

// Closing the connection
// to the ZooKeeper ensemble
zk.close();
```

Session and Watches

Session

- Requests in a session are executed in FIFO order.
- Once the session is established then the **session id** is assigned to the client.
- Client sends **heartbeats** to keep the session valid
- session timeout is usually represented in milliseconds

Watches

- Watches are mechanisms for clients to get notifications about the changes in the Zookeeper
- Client can watch while reading a particular znode.
- Znodes changes are modifications of data associated with the znodes or changes in the znode’s children.
- Watches are triggered only once.
- If the session is expired, watches are also removed.

Paxos

Consensus algorithms are fundamental to distributed computing. In a distributed system, multiple nodes communicate with each other to perform a common task.

Consensus algorithms help these nodes agree on a shared value, even when some nodes may fail or be unreliable.

The importance of consensus algorithms has grown significantly over the past few decades, particularly with the advent of blockchain technology.

Consensus algorithms are used to ensure that all nodes in a blockchain network agree on the state of the ledger, preventing double-spending and other forms of fraud.

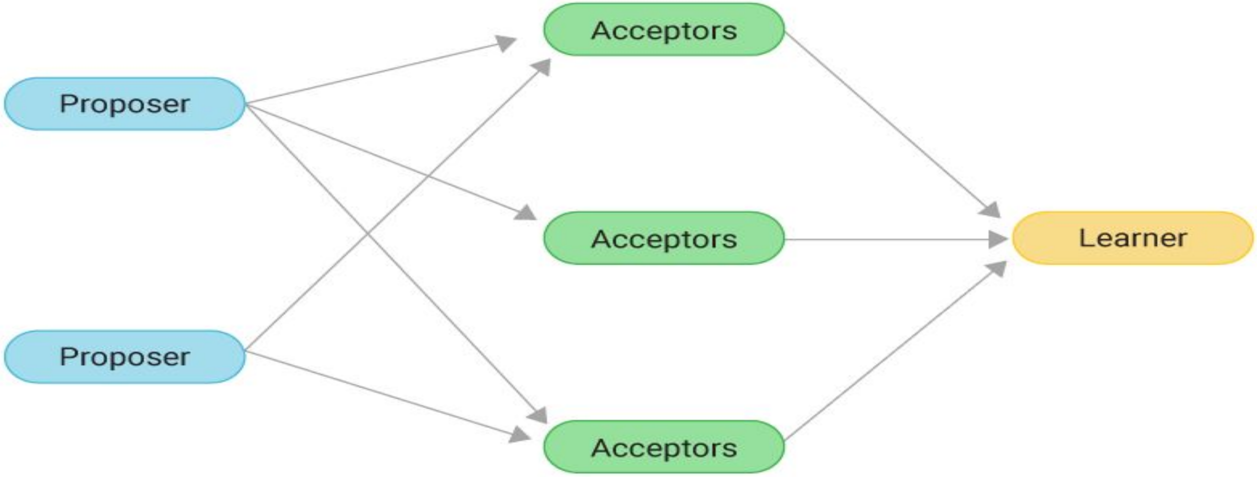
One of the earliest and most well-known consensus algorithms is the Paxos algorithm, developed by Leslie Lamport in 1990.

Paxos is a family of algorithms that can be used to achieve consensus in a distributed system, even when some nodes may fail or be unreliable.

The algorithm is widely used in distributed databases and other systems that require high availability.

Paxos Algorithm:

The algorithm works by allowing a group of nodes to agree on a single value, even if some nodes fail or are unresponsive.



How it works

Essentially the Paxos protocol compares each write request to a proposal. In terms of entities, the Paxos protocol has the following entities:

1. **Proposers:** Receive requests (values/proposals) from clients and try to convince acceptors to accept their proposed values
2. **Acceptors:** Accept certain proposed values from proposers and let proposers know if something else was accepted. A response represents a vote for a particular proposal
3. **Learners:** Announces the outcome

The algorithm consists of three phases:

Phase 1 (Prepare Phase): A node that wants to propose a value as the agreed value sends a prepare message to all other nodes. This message contains a proposal number, which is a unique identifier for this particular proposal. Each node that receives the prepare message responds with a promise to not accept any proposals with a lower proposal number than the one in the prepare message.

Phase 2 (Accept Phase): Once a node receives promises from a majority of nodes, it can send an accept message to all nodes containing the proposed value and the proposal number. Each node that receives this accept message will accept the proposed value only if it has not already promised to accept a higher numbered proposal.

Phase 3 (Commit Phase): Once a node receives accept messages from a majority of nodes, it can send a commit message to all nodes containing the agreed-upon value. The other nodes will then update their state with this agreed-upon value.

It is important to note that this algorithm requires a majority of nodes to agree on a value for consensus to be achieved. This is to prevent a situation where two different values are agreed upon by different subsets of nodes.

Example:

Let us consider an example of how the Paxos algorithm would work in a simple distributed system with three nodes: A, B, and C. Assume that node A wants to propose a value to be agreed upon by the system.

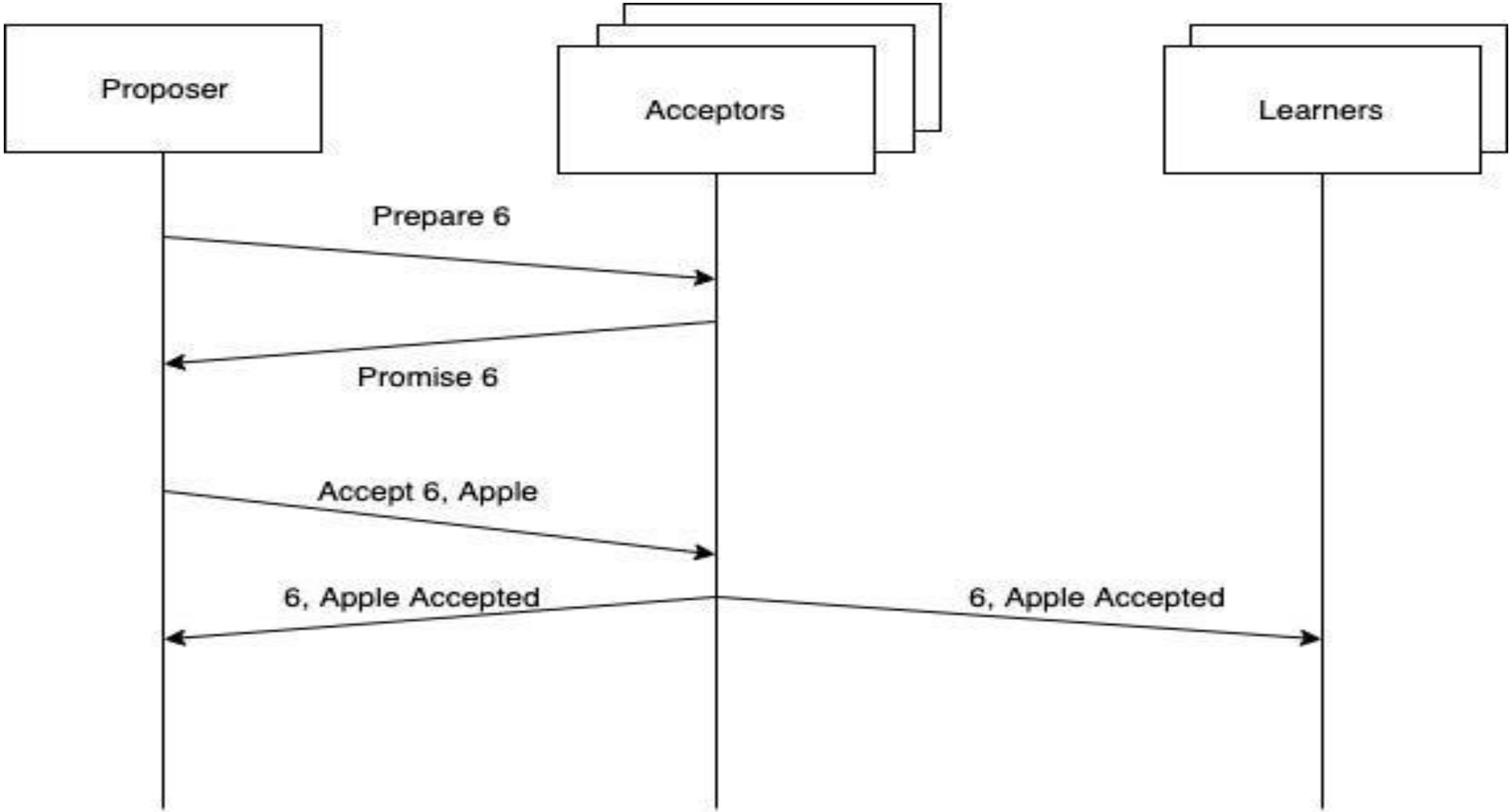
Phase 1 (Prepare Phase): Node A sends a prepare message with a proposal number of 1 to nodes B and C. Nodes B and C both respond with a promise to not accept any proposals with a lower proposal number than 1.

Phase 2 (Accept Phase): Once node A receives promises from a majority of nodes (in this case, nodes B and C), it can send an accept message to all nodes containing the proposed value (let's say the value is "hello world") and the proposal number 1. Nodes B and C will both accept the proposed value.

Phase 3 (Commit Phase): Once node A receives accept messages from a majority of nodes (in this case, nodes B and C), it can send a commit message to all nodes containing the agreed-upon value "hello world". Nodes B and C will then update their state with this agreed-upon value.

This example shows how the Paxos algorithm ensures that all nodes in a distributed system agree on a shared value, even in the presence of node failures or unresponsiveness.

Example of Paxos



Phase 1

Proposer send ‘Prepare Id’ message to all the acceptors along with the chosen Id. Acceptors, on receiving the ‘Prepare Id’ message checks if it already promised for an Id greater than the Id sent by the current proposer. If there is no promise with greater Id, then the acceptor promises not to accept for an Id less than the current Id. If there was a promise with greater Id, the acceptor ignores to respond. If there was a promise with lesser Id, then the acceptor responds with a promise and the accepted value.

Phase 2

Proposer, on receiving the ‘Promise Id’ message from the majority of the Acceptors checks if there is any value associated with the promise. If there are values from the acceptors, then the value with the highest Id is chosen as the accepted value. If there is no value from the acceptors, then the value that the Proposer proposed is chosen as the value. Proposer, on choosing the accepted value issues an ‘Accept’ message to all the acceptors along with the value it chose.

Acceptors, on receiving the ‘Accept Id’ message with a value checks if it already promised not to accept the Id. If it had already promised not to accept then the message is ignored. If not, then the consensus is reached and the Acceptors accepts the **value** given by the Proposer. On accepting a value, the acceptors broadcast the accepted Id and value to all the learners

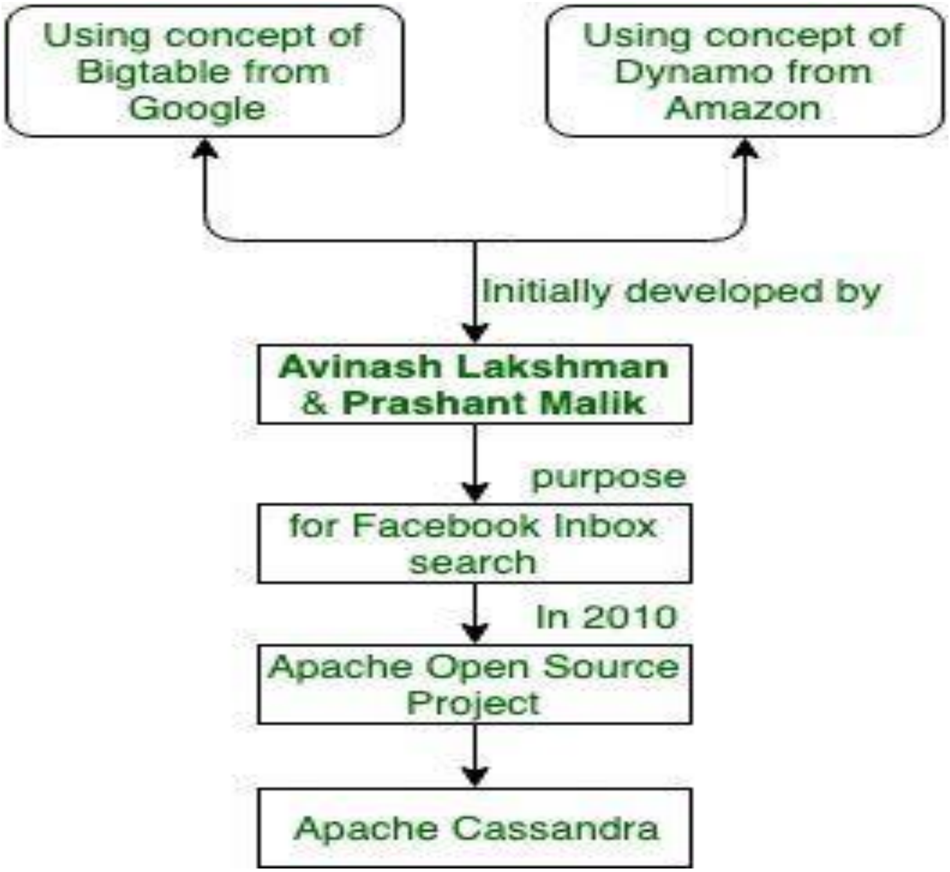
Cassandra

Cassandra is a distributed database management system which is open source with wide column store, NoSQL database to handle large amount of data across many commodity servers which provides high availability with no single point of failure.

It is written in Java and developed by Apache Software Foundation.

Avinash Lakshman & **Prashant Malik** initially developed the Cassandra at Facebook to power the Facebook inbox search feature. Facebook released Cassandra as an open source project on Google code in July 2008.

In March 2009 it became an Apache Incubator project and in February 2010 it becomes a top-level project. Due to its outstanding technical features Cassandra becomes so popular.



Using concept of Bigtable from Google

Using concept of Dynamo from Amazon

Initially developed by

Avinash Lakshman & **Prashant Malik**

purpose

for Facebook Inbox search

In 2010

Apache Open Source Project

Apache Cassandra

Introduction to Cassandra

Apache Cassandra is used to manage very large amounts of structure data spread out across the world. It provides highly available service with no single point of failure. Listed below are some points of Apache Cassandra:

- It is scalable, fault-tolerant, and consistent.
- It is column-oriented database.
- Its distributed design is based on Amazon’s Dynamo and its data model on Google’s Big table.
- It is Created at Facebook and it differs sharply from relational database management systems.

Cassandra implements a Dynamo-style replication model with no single point of failure but its add a more powerful “column family” data model.

Cassandra is being used by some of the biggest companies such as Facebook, Twitter, Cisco, Rackspace, eBay, Netflix, and more. The design goal of a Cassandra is to handle big data workloads across multiple nodes without any single point of failure.

Cassandra has peer-to-peer distributed system across its nodes, and data is distributed among all the nodes of the cluster. All the nodes of Cassandra in a cluster play the same role. Each node is independent, at the same time interconnected to other nodes.

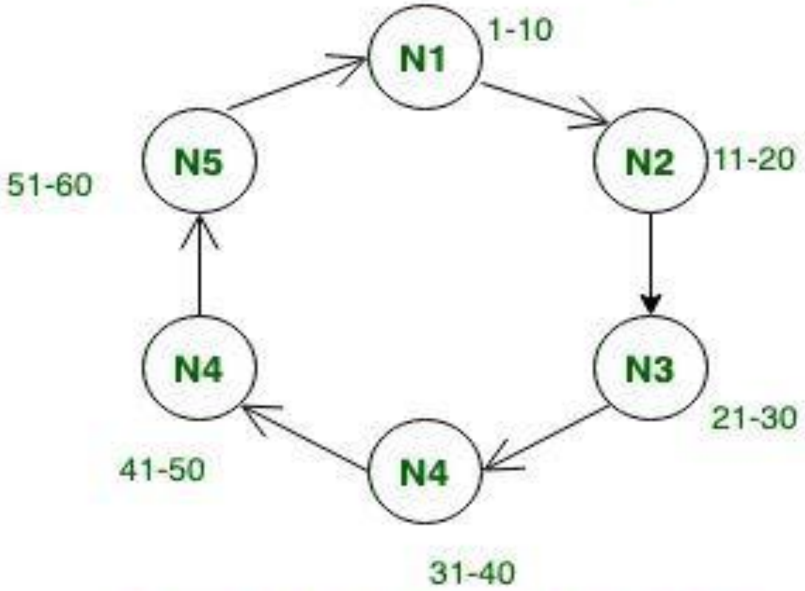
Each node in a cluster can accept read and write requests, regardless of where the data is actually located in the cluster. When a node goes down, read/write request can be served from other nodes in the network.

Features of Cassandra

Cassandra has become popular because of its technical features. There are some of the features of Cassandra:

Easy data distribution – It provides the flexibility to distribute data where you need by replicating data across multiple data centers. for example: If there are 5 node let say N1, N2, N3, N4, N5 and by using partitioning algorithm we will decide the token range and distribute data accordingly. Each node have specific token range in which data will be distribute. let’s have a look on diagram for better understanding.

Token range



Example : Ring structure with token range.

Flexible data storage – Cassandra accommodates all possible data formats including: structured, semi-structured, and unstructured. It can dynamically accommodate changes to your data structures accordingly to your need.

Elastic scalability – Cassandra is highly scalable and allows to add more hardware to accommodate more customers and more data as per requirement.

Fast writes – Cassandra was designed to run on cheap commodity hardware. Cassandra performs blazingly fast writes and can store hundreds of terabytes of data, without sacrificing the read efficiency.

Always on Architecture – Cassandra has no single point of failure and it is continuously available for business-critical applications that can’t afford a failure.

Fast linear-scale performance – Cassandra is linearly scalable therefore it increases your throughput as you increase the number of nodes in the cluster. It maintains a quick response time.

Transaction support – Cassandra supports properties like Atomicity, Consistency, Isolation, and Durability (ACID) properties of transactions.