A Comparative Analysis of Popular Distributed Key-Value Stores

Ramprasad Chinthekindi[1], Shyam Burkule[2], Ashok Kumar Chintakindi[3]

[1]Email: *ramprasad.ch[at]gmail.com*

[2]Email: *shyam.burkule[at]gmail.com*

[3]Email: *ashokkumar.chintakindi[at]gmail.com*

Abstract: ***Distributed key - value stores have become increasingly popular in recent years due to their ability to provide high availability, scalability, and fault tolerance for large - scale data storage and retrieval. This paper examines several prominent distributed key - value stores, such as Apache Cassandra, Amazon DynamoDB, and CockroachDB, and presents a comparative analysis of their architectures, design principles, and performance characteristics. The goal is to provide insights into the architecture, features, trade - offs, and suitability of these systems for different use cases, aiding researchers, developers, and system architects in making informed decisions.***

Keywords: Distributed Systems, Key Value Stores, Databases, Cloud Computing

1. Introduction

Distributed data storage is an essential component of modern computing, enabling the storage and retrieval of large amounts of data across multiple machines or nodes in a distributed system. Distributed key - value stores, a type of distributed database, allow for the storage and retrieval of data in the form of key - value pairs, where the key is a unique identifier and the value is the actual data. Distributed data storage is an important aspect of modern computing for several reasons. First, it allows for the storage and retrieval of large amounts of data across multiple machines or nodes in a distributed system. This can provide horizontal scaling and increased availability, as the data can be spread across multiple nodes and accessed from any node in the system. Second, distributed data storage can improve performance by allowing data to be stored closer to where it is needed. For example, if a web application has users all over the world, it may be beneficial to store the data in multiple locations so that users can access it more quickly. Third, distributed data storage can provide fault tolerance and disaster recovery capabilities. If one node in the system fails, the data can still be accessed from other nodes, ensuring that the system remains available. Additionally, data can be replicated across multiple nodes to protect against data loss in the event of a disaster.

Distributed key - value stores are widely used in industry for a variety of applications, such as web caching, content delivery networks, real - time analytics, and applications requiring fast and scalable data access. As the demand for these systems continues to grow, it is important to understand the architectural differences, design principles, and performance characteristics of the various distributed key - value stores available. This paper aims to provide a comparative analysis of several popular distributed key - value stores, including Apache Cassandra [1], Amazon DynamoDB [2], and CockroachDB [3]. By examining the key features, consistency models, scalability, and performance of these systems, the goal is to offer insights that can guide researchers, developers, and system architects in selecting the most appropriate distributed key - value store for their specific use cases.

This paper is structured as follows. Section II talks about the Apache Cassandra architecture, its consistency model and scalability and performance. Section III presents the Amazon DynamoDB architecture and its consistency model and scalability and performance. Section IV presents the architecture of CockroachDB distributed key - value store. Section V describes the comparative analysis of these systems and suitable use cases for these systems. Finally section VI concludes the distributed key value stores.

1.1 Apache Cassandra

A. Architecture

Apache Cassandra [1], [4] is a distributed NoSQL database, designed to handle large amounts of data across many commodity servers. It employs a decentralized, masterless architecture, i. e. it’s a peer - to - peer distributed system, where all nodes in the cluster are equal and communicate with each other directly. Its design focuses on providing high availability, fault tolerance and linear scalability. The main components of the Cassandra architecture are:

1) Cluster: A Cassandra cluster is a group of nodes that work together to store and manage data. Each node in the cluster runs an instance of the Cassandra server, and all nodes are equal and communicate with each other directly.
2) Node: A node is a single instance of the Cassandra server running on a physical or virtual machine. Each node is responsible for storing and managing a portion of the data in the cluster.
3) Data partitioning: Cassandra provides ability to scale incrementally, with the increasing workloads, which requires dynamic partitioning of data across the set of nodes in the cluster. It uses Consistent Hashing [5], a partitioning scheme to dynamically partition data across the nodes in the cluster. Data is partitioned based on the partition key, which is a column or set of columns in the data that is used to determine the node where the data should be stored.

4) Replication: Cassandra uses replication to achieve high availability and durability. Cassandra provides replication by creating multiple copies of the data and distributing them across different nodes in the cluster. This ensures that the data remains available even if one or more nodes fail. The data is replicated to N nodes, where N is the replication factor, which is configurable in Cassandra. Each key is assigned to a coordinator Node, which saves one copy locally and is responsible for replicating to N - 1 nodes in the cluster. Cassandra supports various replication topologies, such as "Rack Aware", "Rack Unaware", "Datacenter Aware", etc. to support different levels of reliability amidst various types of failures.
5) Consistency: Cassandra offers tunable consistency, allowing users to choose between strong consistency (all nodes see the same data at the same time) and eventual consistency (data may be temporarily inconsistent but will eventually become consistent). It also offers tunable consistency at a per operation level. For example, a user needs a particular transaction to be available on all nodes to mark the transaction complete vs a less critical data be available eventually, providing relaxed consistency guarantees.
6) Data modeling: Cassandra uses a denormalized data model, where data is modeled as a collection of tables with a flexible schema. This allows for efficient querying and high write throughput.
7) Query language: Cassandra uses the Cassandra Query Language (CQL) for querying and manipulating data. CQL is similar to SQL and provides a simple and intuitive way to interact with the database.

Overall, the architecture of Cassandra is designed to provide scalability, high availability, and fault tolerance for large - scale data storage and processing applications. It is widely used in industry for applications such as web analytics, IoT telemetry, and real - time data processing.

B. Consistency Model

Cassandra follows the eventual consistency model, allowing tunable consistency levels based on the application's requirements, as explained in section.

Handling writes: The writes to Cassandra are first written to the on - disk commit log on the coordinator node and are simultaneously written to the in - memory write - back cache called the memtable. The coordinator node sends the write request to the identified replica nodes. Each replica node independently writes the data to its commit log and memtable. The write operation is acknowledged once a specified number of replica nodes have successfully acknowledged the write.
Handling reads: The read path involves first looking up the data in the memtable, similar to the standard LSM KV stores [6], [7]. If the requested data is available in the memtable, it is retrieved directly. Else, Cassandra searches the SST tables (ondisk storage files). Read operations in Cassandra allow users to specify a consistency level. The consistency level determines how many replicas must respond to the read request before it is considered successful. Consistency levels can be adjusted to balance between consistency and availability based on the application's requirements.

Quorum Read: A common practice in Cassandra is to use a quorum read, where the coordinator node sends read requests to a majority of the replicas. This ensures consistency by requiring acknowledgement from a majority of the replicas, preventing stale or inconsistent data from being returned.

C. Scalability and Performance

As per NetFlix's cloud benchmark [8], the scalability is linear as shown in Figure 1. Each client system generates about 17, 500 write requests per second, and there are no bottlenecks as the traffic was scaled up. Each client ran 200 threads to generate traffic across the cluster.

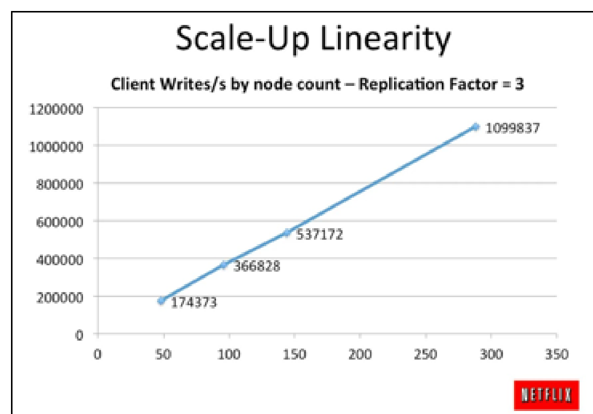

Figure 1: Performance results from Netflix's cloud benchmark on Cassandra

1.2 Amazon DynamoDB

A. Architecture

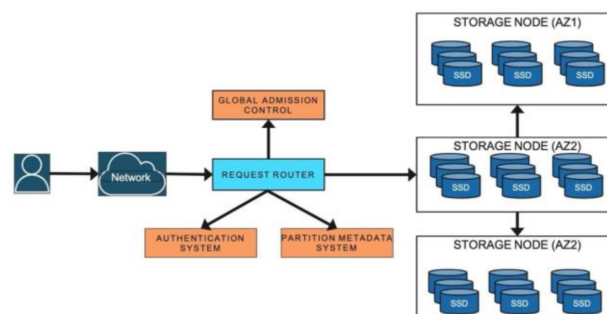

Figure 2: DynamoDB Architecture

Amazon DynamoDB [2] is a fully managed, NoSQL database service provided by Amazon Web Services (AWS). It is designed to provide fast and predictable performance with seamless scalability. The main components of Amazon DynamoDB architecture are:

1) Partition: DynamoDB is designed to scale incrementally. DynamoDB uses consistent hashing to split the total key set into partitions, where each partition represents a contiguous and partial key range. Each storage host is assigned one or more partitions. Each data item identified by a key is assigned to a node by hashing the data item's key to yield its position on the ring, and then walking the ring clockwise to find the first node with a position larger than the item's position. For uniform distribution of load per storage host, DynamoDB creates multiple virtual nodes per each node and places them on the consistent hash ring. It dynamically adjusts the total virtual nodes

Paper ID: SR24426081530 DOI: https://dx.doi.org/10.21275/SR24426081530

on the ring, for uniform load balancing, as the load distribution changes in the production.

2) Replication: To achieve high availability and durability, DynamoDB replicates its partitions on multiple hosts across different Availability Zones. The replicas for a partition form the replication group. The replication group uses Multi - Paxos [9] for leader election and consensus. Any replica can trigger a round of the election. Once elected leader, a replica can maintain leadership as long as it periodically renews its leadership lease. The writes go to the leader replica, which writes it to the write - ahead log and sends to peer replica nodes.
3) Consistency: DynamoDB supports strong and eventually consistent reads. A write is acknowledged to the application once the quorum of nodes persist the data in their local write - ahead logs. The writes and strongly consistent reads always go to the leader replica node.
4) Data Model: DynamoDB tables don't have a fixed schema but instead allow each data item to contain any number of attributes with varying types, including multivalued attributes. Tables use a key - value or document data model.
5) Failure Detection for leader election: During the leader election process, the replica is not available for writes and consistent read traffic, affecting the availability. False positive failures affect the overall availability of the system, possibly due to gray network failures. In order to reduce the false positives, DynamoDB triggers failover only when all the replica nodes are unable to communicate with the leader.

B. Consistency Model

DynamoDB offers strong consistency and eventual consistency options, allowing users to choose based on their application's needs.

C. Scalability and Performance

Amazon DynamoDB [2] is designed for seamless scalability, automatically handling the distribution of data across multiple servers. Figure 3 shows the read latencies on DynamoDB on two different workloads of YCSB [10]. The DynamoDB read latencies do not vary much with increased throughput, Figure 4 shows the write latencies of two YCSB workloads at p50 and p99. The write latencies too vary little with the throughput of the workload.

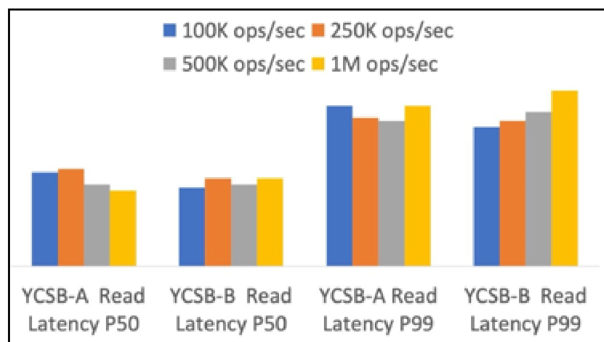

Figure 3: Summary of YCSB read latencies on DynamoDB [2]

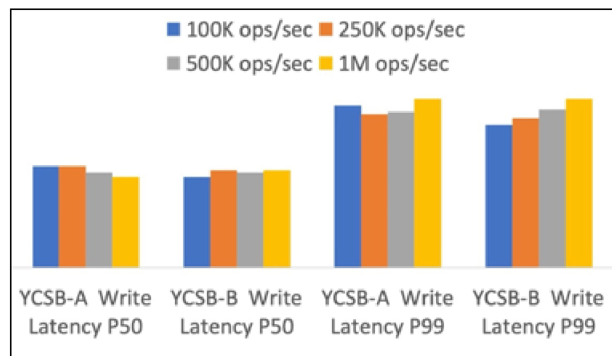

Figure 4: Summary of YCSB write latencies on DynamoDB [2]

1.3 CockroachDB

A. Architecture

CockroachDB (CRDB) utilizes a standard shared - nothing architecture where each node is responsible for both data storage and computation. [3] The cluster can consist of any number of nodes located in the same datacenter or spread globally, giving clients the flexibility to connect to any node within the cluster. Internally, CRDB is structured in layers, each serving a specific purpose.

1) SQL Layer: The topmost SQL layer serves as the primary interface for user interactions with the database. It comprises the parser, optimizer, and SQL execution engine, which convert high - level SQL statements into low - level read and write requests for the underlying key - value (KV) store. Typically, the SQL layer operates without knowledge of data partitioning or distribution, as the subsequent layers present the illusion of a single, unified KV store.
2) Transactional KV Layer: Requests from the SQL layer are directed to the Transactional KV layer, ensuring atomicity for changes involving multiple KV pairs. This layer also plays a significant role in providing CRDB's isolation guarantees.
3) Distribution Layer: This layer provides the notion of a monolithic logical key space organized by key, where all data (system data and user data) is addressable. CRDB employs range - partitioning on the keys to segment the data into contiguous, ordered chunks (Ranges) of about
4) 64 MiB in size, distributed across the cluster. These Ranges are carefully indexed in a two - level structure within a cache of system Ranges for swift key lookups. The Distribution layer determines which Ranges handle subsets of each query and routes them accordingly. Dynamically adjusting their size, Ranges split when they grow too large and merge when they shrink, while loadbased splitting helps alleviate hotspots and CPU usage imbalances.
5) Replication Layer: Each Range is replicated three times by default, with each replica stored on a different node. The Replication layer ensures the durability of modifications via consensus - based replication.
6) Storage: The bottom - most Storage layer represents a local disk - backed KV store, facilitating efficient writes and range scans to support high - performance SQL execution.

Paper ID: SR24426081530 DOI: https://dx.doi.org/10.21275/SR24426081530

B. Consistency Model

The consensus model in CockroachDB is based on the Raft consensus algorithm, which is a widely - used distributed consensus protocol and provides strong consistency.

C. Scalability and Performance

CockroachDB is a highly - scalable, consistently replicated, and transactional database that is specifically designed to operate on cloud platforms with exceptional fault tolerance. This powerful datastore offers seamless horizontal scaling while maintaining strong consistency across multiple nodes, making it an ideal solution for businesses requiring reliable, high - performance data storage capabilities that can adapt to ever - changing workload demands [11]. The Figure 5 shows the latency and throughput performance of CockroachDb [12]. It shows that the p95 throughput scales linearly with number of nodes in the cluster. Also, with increasing number of nodes, there is little variance in p50 and p95 latencies.

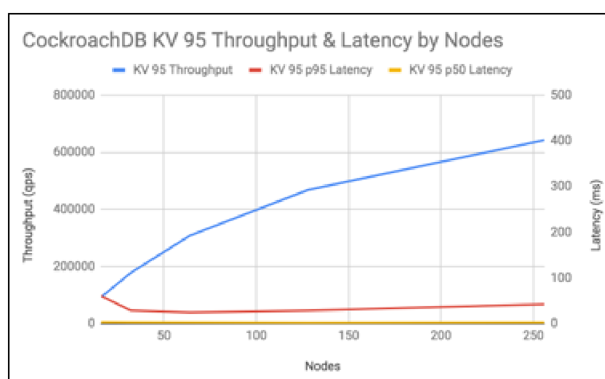

Figure 5: CockroachDB Throughput and Latency with number of nodes [12]

2. Comparative Analysis

Performance and Scalability Analysis:

DynamoDB is known for its low - latency performance and scalability, especially for read - heavy and write - heavy workloads. It offers consistent single - digit millisecond latency for both read and write operations, making it suitable for real - time applications. DynamoDB's performance scales automatically with workload demand, thanks to its managed infrastructure, which dynamically adjusts resources based on usage patterns. Benchmarking results often highlight DynamoDB's ability to handle millions of requests per second with minimal latency fluctuations.

Cassandra is designed for high throughput and linear scalability, making it suitable for large - scale distributed applications. It offers tunable consistency levels, allowing users to balance between consistency and availability based on their application requirements. Cassandra can handle a high volume of read and write operations across multiple nodes, with benchmarking results showcasing its ability to sustain tens of thousands of transactions per second. Performance benchmarks often highlight Cassandra's ability to maintain lowlatency responses even under heavy load and in geographically distributed deployments.

CockroachDB aims to provide distributed SQL with ACID transactions while maintaining high availability and scalability. It offers strong consistency guarantees and multi - active availability, allowing multiple nodes to handle read and write requests simultaneously. Benchmarking results demonstrate CockroachDB's ability to handle large - scale transactions with low - latency responses, making it suitable for OLTP (Online Transaction Processing) workloads. CockroachDB's performance scales linearly with the number of nodes in the cluster, allowing it to handle increasing workload demands by adding more nodes.

Suitable Use Cases:

DynamoDB's low - latency performance makes it well - suited for real - time applications such as gaming leaderboards, live chat, and real - time analytics. Applications that require seamless scalability to handle fluctuating workloads, such as ecommerce platforms, social media applications, and IoT (Internet of Things) data ingestion. DynamoDB integrates well with serverless architectures on AWS, such as AWS Lambda, making it a preferred choice for serverless applications with variable traffic patterns.

Cassandra excels in distributed environments where data needs to be replicated across multiple nodes and data centers, making it suitable for global - scale applications like content delivery networks (CDNs), messaging platforms, and distributed sensor networks. Applications that require high write throughput and low - latency writes, such as time - series data storage, logging, and financial transaction processing. Cassandra's ability to handle large volumes of data and perform ad - hoc queries makes it suitable for analytics and reporting applications that require fast data retrieval and analysis.

CockroachDB's support for ACID transactions and distributed SQL makes it suitable for transactional applications such as e - commerce platforms, financial systems, and order management systems. CockroachDB's ability to replicate data across multiple geographically distributed clusters makes it suitable for global deployments where data sovereignty and low - latency access are critical. Applications that require multiactive availability and can benefit from distributing read and write traffic across multiple nodes, such as content management systems (CMS), collaboration platforms, and multiplayer online games.

3. Conclusion

In summary, distributed key - value stores provide a scalable and fault - tolerant solution for managing and accessing data in a distributed environment. They are well - suited for applications that require high availability, low - latency access to data, and the ability to scale horizontally as the workload increases.

References

[1] Lakshman and P. Malik, "Cassandra - a decentralized structured storage system, " in *In Proceedings of ACM SIGOPS symposium on Operating Systems Review*, 2010, pp.35–40.

[2] M. Elhemali, N. Gallagher, B. Tang, N. Gordon, H. Huang, H. Chen, J. Idziorek, M. Wang, R. Krog, Z. Zhu *et al.,* "Amazon {DynamoDB}: A scalable, predictably performant, and fully managed {NoSQL} database

Paper ID: SR24426081530 DOI: https://dx.doi.org/10.21275/SR24426081530

service, ” in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp.1037–1048.
[3] Cockroachdb: The resilient geo - distributed sql database. [Online]. Available: https: //dl. acm. org/doi/pdf/10.1145/3318464.3386134
[4] DataStax. Introduction to apache cassandra. [Online].
[5] Available: https: //www.odbms. org/wp - content/uploads/2014/06/WPIntroToCassandra. pdf
[6] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy, “Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web, ” in *In ACM Symposium on Theory of Computing*, 1997, pp.654–663.
[7] Leveldb. [Online]. Available: https: //github. com/google/leveldb
[8] Rocksdb. [Online]. Available: https: //github. com/facebook/rocksdb
[9] Netflix. Benchmarking cassandra scalability on aws over a million writes per second. [Online]. Available: https: //netflixtechblog. com/benchmarking - cassandra - scalabilityon - aws - over - a - million - writes - per - second - 39f45f066c9e [9] L. Lamport, “Paxos made simple, ” in *ACM Sigact News*, vol.32 (4), 2001, pp.18–25.
[10] F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with ycsb, ” in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp.143–154.
[11] Doordash. How we scaled new verticals fulfillment backend with cockroachdb. [Online]. Available: https: //doordash. engineering/2023/02/07/how - we - scaled - newverticals - fulfillment - backend - with - cockroachdb
[12] Cockroachdb: Benchmarking overview. [Online]. Available: https: //www.cockroachlabs. com/docs/stable/performance