Benchmarking Distributed Key-Value Stores

Divyaank Tiwari
115765038

Vishnutej Reddy
115934115

Sai Yashwanth Brahmadevara
115752122

Abstract

Distributed Key-Value Stores or Caches comprise the heart of the modern Distributed System when it comes to extracting the last bit of performance. These systems have become essential components in building scalable, reliable, and high-performance distributed applications. In the last decade or so, many new Key-Value stores have been developed like Apache Ignite, TiKV, etcd that fundamentally serve the same purpose, but are used for different kinds of applications. The existing studies that benchmark key-value stores don't seem to cover these new systems or atleast all these systems together. Thus, our project aims to fill this gap in literature and we aim to benchmark these new systems.

Problem

Developers leverage distributed key-value storage systems to improve the performance of their distributed systems, by eliminating costly database lookups. However, when it comes to the choice of a key-value storage solution, they often restrict themselves to a handful of classic systems like Cassandra [11]. The root-cause of this problem seems to be the absence of detailed benchmarking studies with respect to performance and distributed systems properties like consistency, scalability, tolerance that limits the knowledge about upcoming key-value storage systems. We believe that with the explosion of new key-value stores in the last decade, falling back on legacy systems might prove to be a suboptimal decision.

Existing Solutions

Shankar et al[1] study popular web-scale and cloud serving workloads, to identify different application-specific aspects, including commonly occurring data request distributions, update patterns, and environmental factors, that affect the performance of high-performance key-value stores that can operate well with `RAM+SSD' hybrid storage architecture. They go on to propose a micro-benchmark suite that can be used to study high-performance, hybrid key-value stores on modern clusters, from the perspectives of both the application and the key-value store. Sen et al [2] perform comprehensive benchmarking of the Apache Accumulo[3] a Hadoop-based Distributed Key-Value Store, their paper documents the workloads that they have used as well as the details about their experimental setups. Boza et al[4] propose a novel, trace replay model suitable for key-value stores and describe KV-replay, an open-source replayer that implements this model. They go on to show that KV-replay is as fast as YCSB [12], accurate in reproducing inter arrivals and burstiness, and useful, as it allows for the evaluation of key-value stores under real workloads. The study by Cao et al[5] reveals a discrepancy between the workloads generated by the widely utilized YCSB key-value benchmark and the actual workloads collected, primarily stemming from the neglect of key-space localities in YCSB-triggered workloads for underlying storage systems. To rectify this issue, the survey suggests a key-range based modeling approach and introduces a benchmark specifically designed to more accurately emulate the workloads encountered in real-world key-value stores. This proposed benchmark has the capability to synthetically generate more precise key-value queries, offering a representation of the read and write operations performed by key-value stores on the underlying storage system.

Shortcomings of existing solutions

Majority of the studies mentioned earlier focus mostly on popular key-value stores. With the emergence of new key-value stores that are now written in modern languages like Golang and Rust, it is imperative to have a comparative analysis between the old and new systems along the dimensions of performance as well as important properties like consistency, scalability, tolerance, replication etc.

Our Solution

We aim to perform a detailed comparative study by benchmarking key-value stores like Apache Ignite, TiKV, and etcd against a tried-and-tested, widely used store like Cassandra. We aim to compare these systems with respect to performance as well as their distributed systems properties. For this, we aim to use the YCSB benchmark. As explained later in detail, YCSB has an array of workloads, each of which has a different proportion and distribution of database operations. This allows for a thorough evaluation of the datastore under test and given its extensible nature, the proportion and distribution of operations can be changed according to the use case.

Overview of benchmarked Key-Value stores

Here we give a brief overview of the systems that we are going to be benchmarking as a part of our study:

1. **Apache Ignite:** Ignite's[8] in-memory computing capabilities ensure unparalleled data access and processing speeds by keeping data in RAM, making it ideal for high-performance computing and real-time analytics. It supports SQL queries, which enhances its usability and integration with existing tools and applications. Ignite follows a 2-Phase Commit Protocol [24] for its Transaction Processing and it uses optimizations like Write-Ahead-Logging and Checkpointing [25] for handling native persistence of data. Ignite's ability to offer durable disk persistence complements its in-memory nature, ensuring data resilience and providing a reliable recovery mechanism, thereby serving as both a volatile and persistent store for critical business applications.
2. **TiKV:** TiKV's[9] distributed architecture enables it to achieve high availability and scalability by spreading data across multiple nodes. It distinguishes itself with transactional support that adheres to ACID compliance. Furthermore, its design as a key-value store, and communication via the gRPC protocol facilitating efficient and language-agnostic service interactions.
3. **etcd:** etcd's[10] foundation on the Raft consensus algorithm ensures strong consistency and reliability, making it indispensable for distributed systems requiring fault tolerance and high availability. Its simple clustering model facilitates easy management and scalability of services, while support for atomic transactions allows for complex operations to be performed reliably across multiple keys. Additionally, etcd's critical role as the backbone for Kubernetes, managing cluster state and configuration, highlights its importance in modern infrastructure, where secure, scalable, and consistent data storage is paramount.
4. **Cassandra:** Its [11] decentralized architecture ensures there's no single point of failure, making it exceptionally reliable for critical applications that require continuous uptime. Its support for flexible data storage allows it to accommodate structured, semi-structured, and unstructured data, catering to a wide range of use cases from simple to complex data models. Furthermore, Cassandra's tunable consistency levels enable developers to optimize for either consistency or performance as needed, while its built-in support for time series data makes it an ideal choice for IoT, monitoring, and real-time analytics applications, underscoring its versatility and importance in handling diverse data requirements efficiently.

Experimental Methodology

The experimental methodology aims to benchmark distributed systems for measuring their throughput, latency, and scalability. This study employs various benchmarking tools and techniques to comprehensively evaluate the performance of distributed systems under different conditions. The following subsections provide the details of environment setup, metrics which we aim to measure and benchmarking tools which are used.

Benchmarking Metrics & Tools

The performance evaluation of the distributed systems are evaluated on metrics: Throughput, Latency, and Scalability,

- Throughput measures the rate of successful requests processing per unit time. Use the **operationcount** and **recordcount** parameters to define the total number of operations and records.

- Latency access the time taken for a message to travel through the system. YCSB reports average, minimum, maximum, and percentile latencies.

- Scalability evaluates the system's ability to handle increased workloads without significant performance degradation. Vary the **recordcount** and **operationcount**, and run YCSB on different cluster sizes to test system scalability.

Experimental Setup

We are leveraging CloudLab [13] for benchmarking our distributed Key-Value stores. CloudLab is a "virtual lab" for experiments on networking, cloud computing, and distributed systems. It allows experimenters to set up real (not simulated!) end hosts and links at one or more CloudLab host sites located around the United States. Experimenters can then log in to the hosts associated with their experiment and install software, run applications to generate traffic, and take network measurements.

We have designed our benchmarking setup to closely resemble a real-world distributed application used in production settings. Such an application usually consists of multiple instances running across multiple servers. On a particular server the application instances are backed by one or more instances of a decentralized Key-Value store. This store acts as a cache for the application data which is persisted and fetched, usually from an RDBMS database like MySQL[21], Postgres[22], Microsoft SQL Server[23] etc. To mimic this setting we use a single CloudLab server to perform our benchmarking experiments. Depending on our experiment, we instantiate either a single-node or multi-node cluster of the Key-Value store under a test. Once the cluster setup is complete, we load the data and run our benchmarks using a YCSB client, which has a binding for each of our Key-Value stores. We elaborate upon YCSB in the next section.

Although CloudLab offers access to various classes of specialized hardware, we have opted for standard commodity-grade machines for all our VMs. The primary reason for this is that beefier servers usually are difficult to reserve in CloudLab, and even if we are able to do so, it's very difficult to have reservations for longer periods of time due to high demand for CloudLab compute resources. For more details about our server specifications, please refer to our GitHub repository[20].

YCSB Benchmark

YCSB (Yahoo! Cloud Serving Benchmark) is a versatile benchmarking tool designed to assess various distributed database systems such as HBase, Cassandra, Riak, MongoDB, and others. It offers a core package featuring five predefined benchmark workloads labeled A to E. Each record in a typical YCSB workload consists of a row key and 10 columns, with each column containing 100 bytes of random ASCII text.

The YCSB benchmark comprises two main components: a workload generator and a workload configurator. Within the workload generator, there exists a fundamental core workload class responsible for initiating operations. These operations, including inserts, updates, or reads, can be tailored through the workload configurators, allowing for the utilization of various statistical distributions across target databases. By adjusting different database configuration parameters, the benchmarks can be executed to evaluate throughput and latency performance. Additionally, the core workload class can be expanded to manipulate the environment and initiate operations that reveal further performance attributes.

The workload client selects operations (such as insert, update, read, or scan) and determines which record to access or modify, the number of records to scan, and other factors randomly. These decisions are influenced by various random distributions, including uniform, Zipfian, latest, or multinomial. In a uniform distribution, items are chosen randomly with equal probability. However, in a Zipfian distribution, certain records are more probable to be selected than others.

Currently, YCSB has dedicated workloads for Cassandra and Ignite[12], as well as TiKV and etcd[17]. YCSB provides two benchmark tiers for evaluating the performance and scalability of cloud serving systems. The Performance tier of the benchmark focuses on the latency of requests. The Scaling tier of the benchmark focuses on scalability and elasticity.

Workload	Operations
A - Update Heavy	Read : 50% Update : 50%
B - Read Heavy	Read : 95% Update : 5%
C - Read only	Read : 100%
D - Read latest	Read : 95% Insert : 5%
E - Short ranges	Scan : 95% Insert : 5%
F - Read Modify Write	Read: 50% Read-Modify-Write: 50%

Table 1. The table provides various YCSB workloads[15] with the proportion of operations which gives us insights of the nature of workloads.

Throughput & Latency Evaluation : The workloads of YCSB could be used for measuring the read/update latency of executions of requests and throughput.

Scalability Evaluation : Scalability evaluation could be done by considering combinations of changing load and system capacity between subsequent workloads. Testing by increasing the instances along with proportional increase of load could also be used for evaluating scalability.

Capturing Metrics

CloudLab's monitoring and logging features provide detailed insights into system performance during YCSB benchmarks, tracking metrics like CPU, memory, and disk usage. These tools help identify performance bottlenecks and system behavior under load, ensuring accurate benchmark results. Additionally, CloudLab's monitoring and logging tools can help in comparing the performance impact of different configuration settings, identifying optimal configurations for each key-value store when running YCSB benchmarks. In addition to these metrics, YCSB itself exposes many metrics post its workload execution like total runtime, operation latency across different granularities (min/max/average/95th Percentile/99th Percentile) as well as metrics like proportion of time spent in Garbage collection for JVM based Key-Value stores like Apache Ignite and Apache Cassandra. Due to the nature of our experiments, we chose to go ahead with YCSB metrics.

Analysis

We have used existing 6 YCSB workloads to benchmark the mentioned key value store systems for measuring the latencies of operations in the workloads. The graphs represent the data collected from the average of 3 runs of each experiment, for each Key-Value store and cluster configuration. In the next few sections, we first compare the performance of each system with all workload by varying the number of nodes in the cluster. Finally, we compare the performance among the systems and try to make some inferences.

For all the YCSB workloads, we went ahead with the default parameters: operationcount=100000, recordcount=100000, thread count=4.

Apache Ignite

YCSB enables native persistence mode for Ignite during the initial setup. Native persistence mode enables data to be stored persistently on disk, ensuring durability even in the event of node failures or restarts. This feature allows Ignite to efficiently manage larger datasets that exceed available memory capacity by offloading data to disk while still providing fast access to frequently accessed data in memory. When we execute the constituent YCSB workloads and observe the overall workload runtime, we found the pattern depicted in Fig 1. Note that YCSB doesn’t return the runtime for a single-node Ignite cluster. This pattern shows that when we increase the number of nodes in the cluster, the runtime increases from 3-Node setup to 5-Node setup and then starting from 7-Node it starts decreasing.

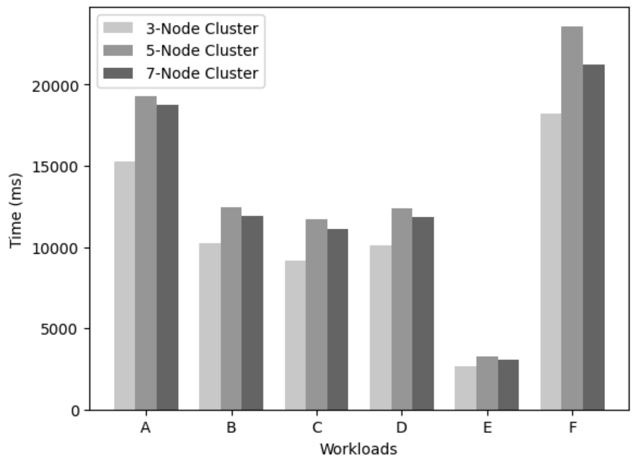

Fig 1: Overall Apache Ignite Runtime for YCSB workloads

To see whether this pattern persists, we ran the experiments with a 9-Node cluster as well, and obtained the results as shown in Fig 2. We see that in this case, the runtime is almost equal to or lesser than the 7-Node cluster, thus depicting that as the number of nodes in the cluster increases, the runtime starts dropping, with the 5-Node setup being the global maxima.

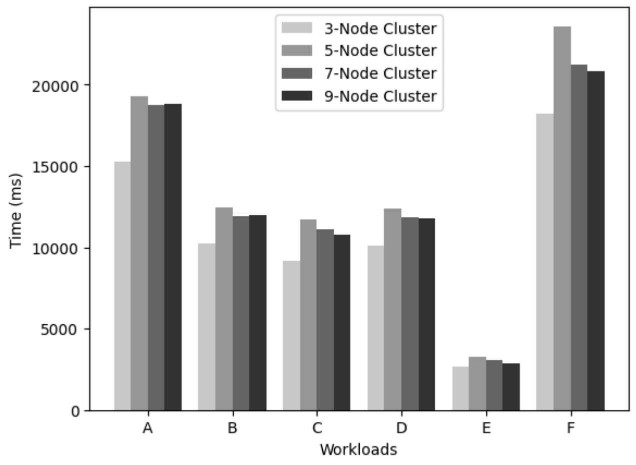

Fig 2: Continuous decrease in runtime for 9-Node setup

Next, we compare the Read, Update and Insert latencies for a 1, 3, 5 and 7-Node Ignite cluster. As seen in Figure 3, only workloads A, B, C, D and F report the Read Latencies and in case of F, we don’t get the corresponding latency for a single node cluster. As observed in the overall runtime case, the runtime increases from the single node cluster all the way up to the 5-Node cluster, then it starts decreasing from the 7-Node cluster. The only exception to this trend occurs for Workload F, which has a very high latency for the 3-Node Cluster. We observed this behavior during multiple iterations of this experiment but were unable to reason about its occurrence. However, we note that just like other

Workloads, even in this case we see a drop in the latency as the cluster size increases.

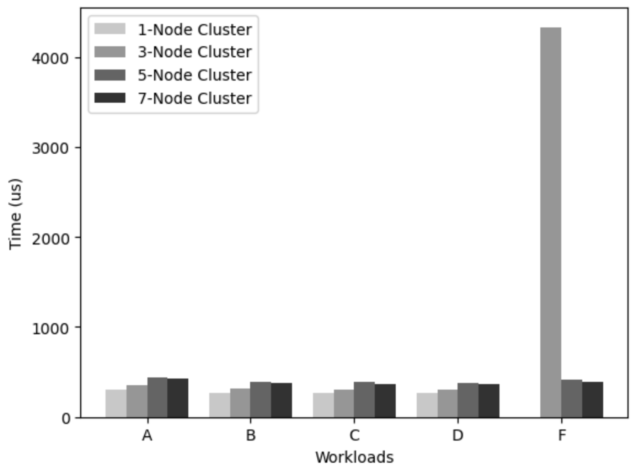

Fig 3: Read Latency for Ignite Clusters

An analogous trend holds true for Update and Insert Latencies as well, as shown in Figures 4 and 5 respectively. As far as the Read-Modify-Write latency is concerned (i.e Workload F), a similar trend holds but we haven't depicted it here due to space limitations.

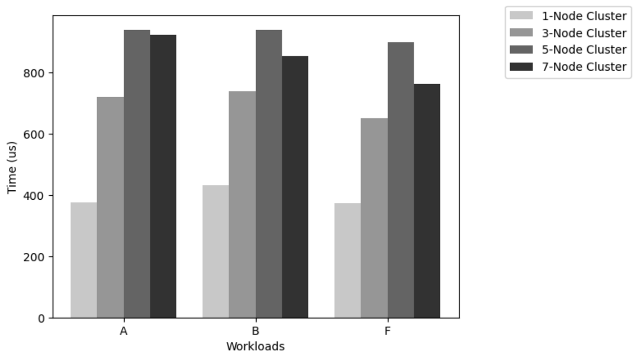

Fig 4: Update Latency for Ignite Clusters

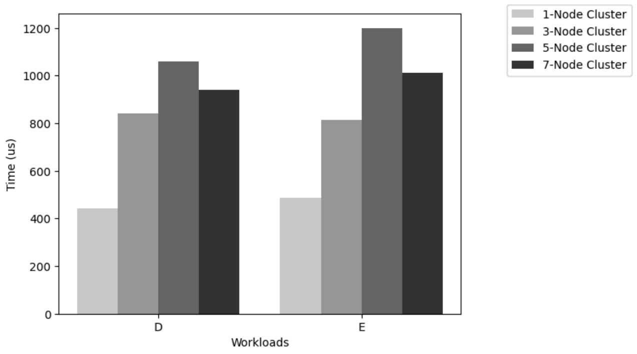

Fig 5: Insert Latency for Ignite Clusters

We hypothesize the following reason for the trend of decrease in operation runtime for Apache Ignite. Increasing the cluster size from 1 to 3 to 5 nodes initially raises resource utilization, potentially causing contention and longer runtimes with higher latencies. However, beyond a certain threshold, like 7 nodes, additional resources may alleviate contention, balancing data distribution for more efficient access and lower latencies across read, update, insert, and read-modify-write operations. Larger clusters offer enhanced parallelism and fault tolerance, leading to higher throughput, lower latencies, and improved reliability. Additionally, performance gains may stem from system warm-up effects as caches populate and components optimize over time.

Apache Cassandra

For Apache Cassandra, the total runtime for all workloads across all cluster sizes is shown in Fig 6. Unlike Ignite, there doesn't exist a clear cut trend with regards to the decrease in runtime with cluster size. We can infer that the maxima with regards to the run time occurs when the cluster has a size of around 3 or 5. For Workloads A, C, D there is a drop in the runtime for the 7 node cluster, while for the read-heavy Workload B, it shows the highest runtime. For Workloads E and F, there is a slight increase in the runtime for the 7-Node cluster compared to the 5-Node case. Thus, for Cassandra, we can't confidently say that increasing the cluster size makes operations more performant.

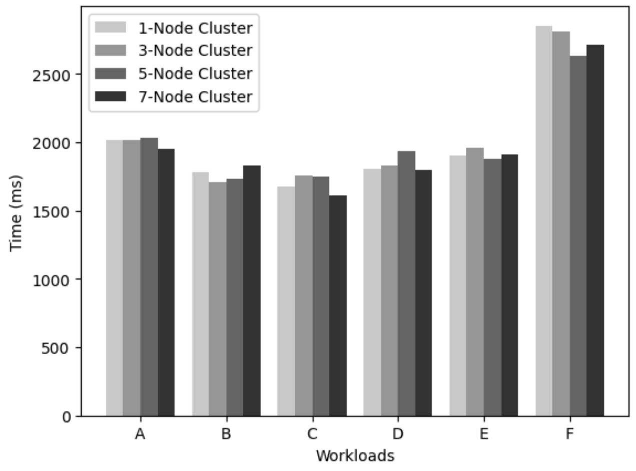

Fig 6: Overall Cassandra Runtime for YCSB workloads

Figure 7 shows the trends for the Read Latency of Cassandra for different workloads and cluster sizes. Here for all workloads except B, we do see the trend that increasing cluster size does lead to better performance. For the update latencies depicted in Figure 8, we see that for workloads A and F, the latency remains mostly constant irrespective of the cluster size. For workload B, we see that there is a latency drop from 1-Node to 3-Node cluster then a rise for 5-Node followed by yet another drop. Thus, it becomes difficult to reason about Cassandra for this particular workload.

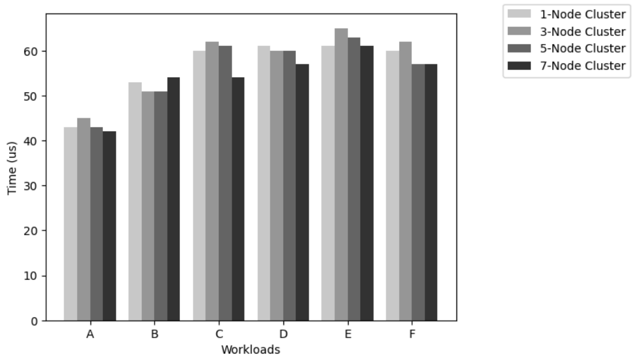

Fig 7: Read Latency for Cassandra Clusters

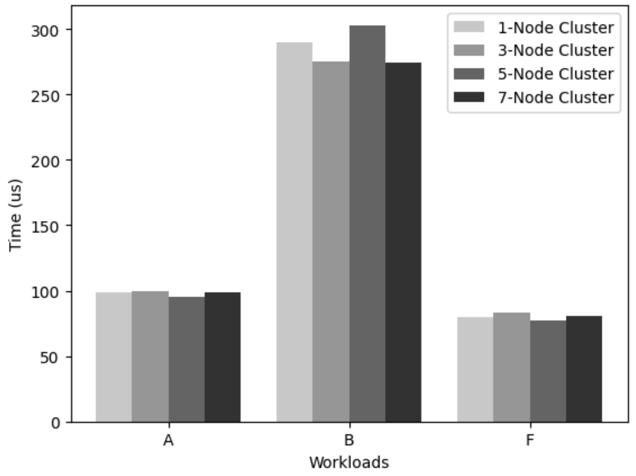

Fig 8: Update Latency for Cassandra Clusters

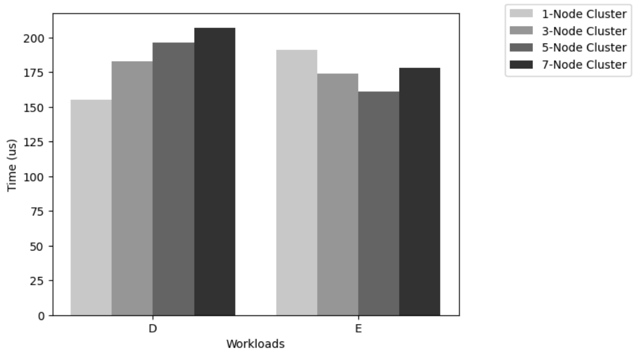

Fig 9: Insert Latency for Cassandra Clusters

With regards to the insert latencies depicted in Fig 9, for Workload D, there seems to be a steady increase in the latency as we increase the cluster size. In contrast, for Workload E, there is a decrease in latency from 1-Node to 5-Node cluster, followed by an increase for the 7-Node cluster. We find somewhat similar behavior when we measure the Read-Modify-Write latency in Workload F, where increasing the cluster size doesn't lead to better performance. Thus, from our Cassandra benchmarking experiments, we conclude that for Read heavy workloads Cassndra's performance scales linearly with the cluster size, while for any other workload, a more thorough benchmarking needs to be performed.

TiKV

We first observe that the workloads A and F have their total time significantly larger than workloads B,C,D,E because A and F have significant(~50%) write operations and the remaining are read heavy workloads. In Fig 10A, 10B, 10C, we can observe that there is a jump in total time taken from 1-Node cluster to 3-Node cluster for all the workloads especially for workload A and F. These workloads have significant write operations and in order to perform consensus for 3-Node cluster, the total time taken to complete the workload took a jump from 1-Node to 3-Node cluster. There is only a slight increase in other workloads because they are read heavy workloads with fewer write operations. As we increase the number of nodes in a cluster the read operation should tend to decrease as the operations would fetch the data using nodes and the write operation should tend to increase as to perform the consensus. We see that average read latency is reduced as we increase the number of nodes. Since we are using 4 threads to perform all operations in workloads, we see a slight fall from 3-Node to 5-Node for few workloads in average read latency. Similarly the average update latency also shows the expected behavior for the initial jump.

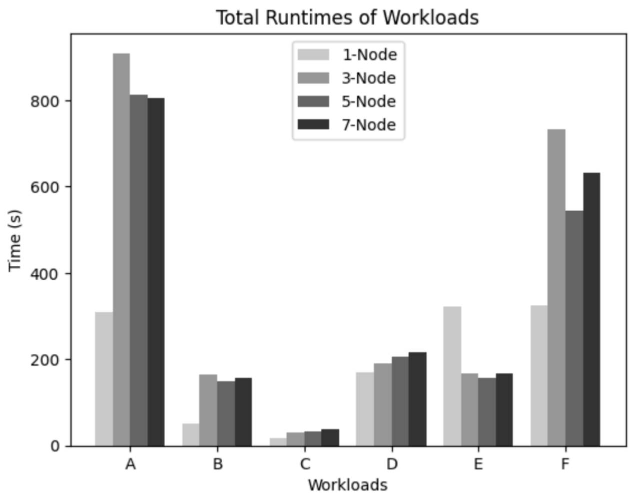

Fig 10A: The above graph shows the total time in seconds for completion of YCSB workloads for 1, 3, 5, 7 Nodes cluster.

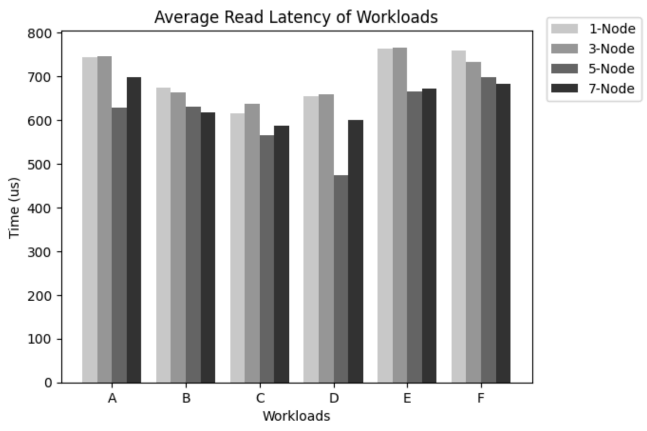

Fig 10B: The above graph shows the average read latency in microseconds taken in YCSB workloads for respective clusters varying with nodes.

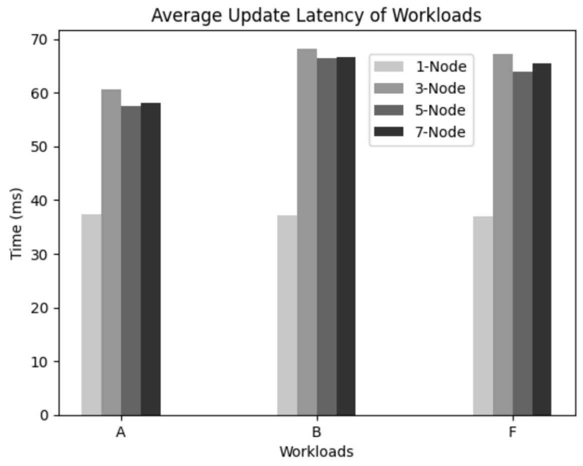

Fig 10C: The above graph shows the average update latency in microseconds taken in YCSB workloads for respective clusters varying with nodes.

etcd

Similar tests were performed on etcd to measure the time/latencies taken by YCSB workloads. In Fig 11, it can be seen that the total time taken is increased from 1-Node to 3-Node cluster and then slightly reduced or almost remained the same in the next clusters. We could observe a similar trend regarding the consensus and the workloads B, C, D and E considerably took less time because they are read-heavy workloads. Fig 12 and Fig 13, represents the average read and average update latencies. The read latency of workload C is because the workload has only read operations whereas the other workloads have at least some write operations making the read latency to increase.

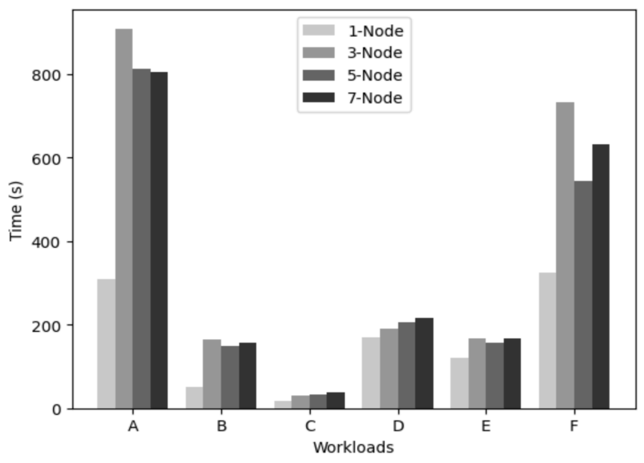

Fig 11: Total time in seconds taken to complete the YCSB workloads for etcd clusters.

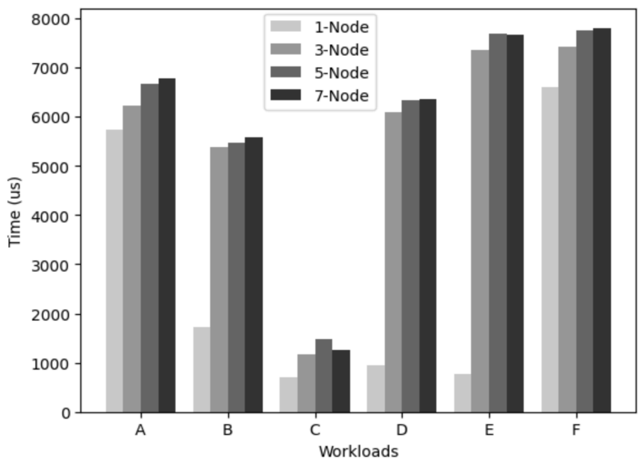

Fig 12: Read latency in micro seconds to complete the YCSB workloads for etcd clusters.

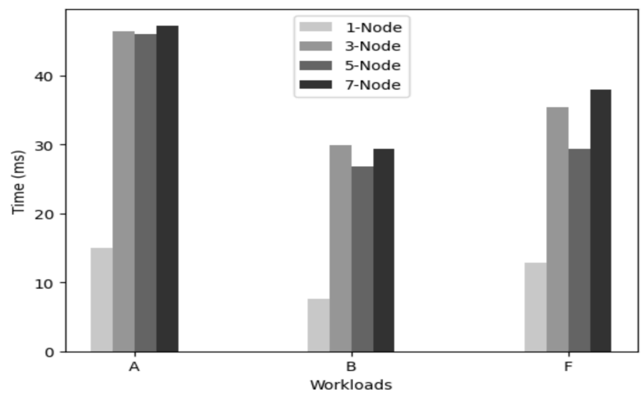

Fig 13: Update latency in milliseconds to complete the YCSB workloads for etcd clusters.

Comparison Across Systems

Despite our initial hypothesis that one could find common performance characteristics for disparate Key-Value stores, given that they do logically the same work just in different contexts, we discovered that it is very difficult to have a thorough, apples-to-apples comparison between the systems under test.

A pivotal point to note is that TiKV and etcd are taking significantly more total time to complete the workloads than Ignite and Cassandra because the TiKV and etcd offer strong consistency. Cassandra is eventually consistent and doesn't offer SQL-based querying capacity, which is in contrast to Ignite which is strongly consistent and has support for distributed transactions and SQL-interface.

Fig 14 shows the total time taken by clusters and plotted with two y-axis for better representation. As discussed above, TiKV and etcd took significantly more time. TiKV is built on Rocks DB which has LSM tree based storage which could efficiently handle large data with low latency. etcd is built on Level DB and leverages Raft for State Machine Replication. We can observe that TiKV took slightly less time than etcd. Ignite is equivalent to etcd in this case which is to be expected given their strong consistency guarantees. Cassandra's eventually consistent model helps it perform better than all 3 key-value stores.

Next in Fig 15, we can observe the average read latency of the 3-Node cluster of YCSB workload C. To get the best comparison, we compared workload C which has only read operations. Data sharding is done in TiKV in the data distribution process and offers less read latency than etcd.
etcd could be usually used for storing the configurations and performs better in low workloads. Just like earlier, Cassandra ends up outperforming other Key-Value stores in this case as well, with Ignite coming in a close second. We see roughly similar trends for Update, Insert, Read-Modify-Write latencies as well; however in the interest of brevity, we haven't included their results.

With regards to ease of installation for our experiments, we found Ignite followed by Cassandra, followed by TiKV and etcd to be the easiest to set up. Although not related to the performance itself, while benchmarking we experienced significant delays while loading YCSB benchmarks for etcd, which discouraged us from performing more ambitious experiments.

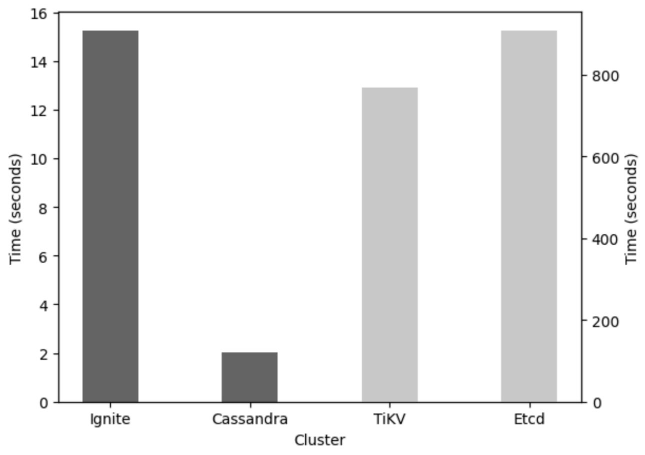

Fig 14: The above graph depicts the total time taken for the 3-Node clusters of , teach system in seconds to complete the YCSB workload A. The left two bars are scaled to the left y-axis and the right two bars are scaled to the right y-axis since there is a significant difference between them. For example, Ignite took around 15 sec and TiKV took around 750 sec.

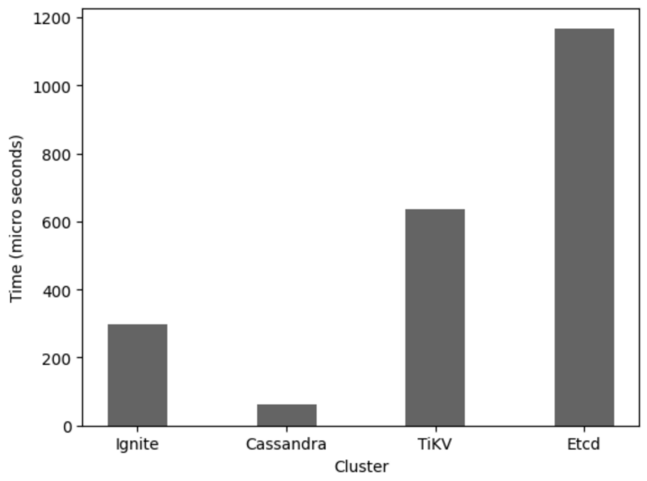

Fig 15: The above graph depicts the average read latency in microseconds for 3-Node clusters of systems for YCSB workload C.

Conclusion

We conclude that benchmarking of distributed key-value stores using YCSB workloads provides valuable insights into the performance and suitability of Apache Ignite, Cassandra, TiKV, and etcd for various use cases. Owing to their consistency models primarily, we recommend Apache Cassandra followed by Apache Ignite for general use cases. For applications in need of stronger consistency guarantees, TiKV and etcd should be the preferred choice. All in all, our evaluations have been primarily based on performance metrics and thus, for any particular use case a more thorough application specific evaluation must be undertaken.

Future Work

KVBench concerned itself primarily with performance benchmarking. In the future, for testing distributed system properties like fault tolerance, we aim to use Chaos Monkey [14] that is based on the principles of Chaos Engineering, which is a discipline that involves experimenting on a software system or a distributed system in order to build confidence in its resilience, reliability, and ability to withstand turbulent conditions. The core idea is to intentionally inject controlled and measured amounts of chaos into a system to uncover weaknesses and vulnerabilities that might not be apparent under normal operating conditions. Our current benchmarking setup consists of a single server of sufficient capacity. To further test the resilience of the Key-Value stores under test, we aim to set up a geo-distributed cluster of servers, spread across Utah, Wisconsin and South Carolina, the regions where CloudLab servers are located.

REFERENCES

[1] D. Shankar, X. Lu, M. Wasi-ur-Rahman, N. Islam, and D. K. Panda, ‘Benchmarking key-value stores on high-performance storage and interconnects for web-scale workloads’, in *2015 IEEE International Conference on Big Data (Big Data)*, 2015, pp. 539–544.

[2]https://accumulo.apache.org/papers/accumulo-benchmarking-2.1.pdf

[3] https://accumulo.apache.org/

[4] E. F. Boza, C. San-Lucas, C. L. Abad, and J. A. Viteri, ‘Benchmarking Key-Value Stores via Trace Replay’, in 2017 IEEE International Conference on Cloud Engineering (IC2E), 2017, pp. 183–189.

[5] Z. Cao, S. Dong, S. Vemuri, and D. H. C. Du, ‘Characterizing, modeling, and benchmarking RocksDB key-value workloads at facebook’, in Proceedings of the 18th USENIX Conference on File and Storage Technologies, Santa Clara, CA, USA, 2020, pp. 209–224.

[6] Sysbench - https://github.com/akopytov/sysbench

[7] TPC-C benchmarking

[8] https://ignite.apache.org/

[9] https://tikv.org/

[10] https://etcd.io/

[11] https://cassandra.apache.org/_/index.html

[12] https://github.com/brianfrankcooper/YCSB

[13] https://www.cloudlab.us/

[14] https://github.com/Netflix/chaosmonkey

[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, Russell Sears ’Benchmarking Cloud Serving Systems with YCSB’

[16] Jo ̈rn Kuhlenkamp, Markus Klems, Oliver Ro ̈ss, ‘Benchmarking Scalability and Elasticity of Distributed Database Systems’

[17] https://github.com/pingcap/go-ycsb

[18] https://docs.pingcap.com/tidb/stable/quick-start-with-tidb

[19]https://github.com/brianfrankcooper/YCSB/wiki/Running-a-Workload

[20] https://github.com/divyaankt/KVBench

[21] https://www.mysql.com/

[22] https://www.postgresql.org/

[23] https://www.microsoft.com/en-us/sql-server/

[24]https://www.gridgain.com/resources/blog/apache-ignite-transactions-architecture-two-phase-commit-protocol

[25]https://www.gridgain.com/resources/blog/apache-ignite-transactions-architecture-ignite-persistence-transaction-handling

Appendix

GitHub Source Code: https://github.com/divyaankt/KVBench