Report - Project 1st stage - Group 5

Andrey Bortnikov[ist1108489]andrey.bortnikov@tecnico.ulisboa.pt,
Bernardo Santos[ist199185]bernardopaduasantos@tecnico.ulisboa.pt, and
Gonçalo Carmo[ist199228]goncalo.n.carmo@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisbon, Portugal

Abstract. YugabyteDB is a open-source, distributed database that claims to offer consistent, reliable, high permorfance and scalability transactional services. The use of benchmarks allows the verification of the claims made by the database developers and finding possible bottlenecks in the stages of a pipeline which was empirically confirmed through using Scalability and Sysbench benchmarks.

Keywords: Scalability · Distributed · Benchmarks · Pipeline

1 Introduction

YugabyteDB is a distributed, open-source database management system designed for high-performance, scalability, and resilience in cloud-native and data-intensive applications. The system combines the SQL interface with NoSQL databases and is particularly well-suited for applications requiring low-latency, high-availability, and geo-distribution.

Our system choice was based on the amount and quality of the documentation provided by official sources, the system already being containerized and matching all the necessary requirements.

Github commit id to be evaluated: commit 6575e32d9d82427630b38cc63755c89660ca07ae

2 System description

YugabyteDB represents a distributed SQL database system with a primary goal of maintaining strong transactional consistency even in the face of failures, adhering to the principles of ACID compliance. In terms of the CAP Theorem, YugabyteDB can be categorized as a Consistent/Partition Tolerant (CP) database. YugabyteDB consists of two key components: a storage engine referred to as DocDB and the Yugabyte Query Layer.

2.1 DocDB

The storage engine in YugabyteDB is a specialized version of RocksDB, enriched with sharding and load balancing algorithms to efficiently manage data. Furthermore, data replication among nodes is overseen by the Raft consensus algorithm.

There is also a Distributed Transaction Manager and Multiversion Concurrency Control system to facilitate and manage distributed transactions.

This storage engine also uses a Hybrid Logical Clock, a timekeeping mechanism that combines roughly synchronized physical clocks with Lamport clocks to keep track of causal relationships among events. It ensures accurate sequencing of operations in a distributed environment.

The DocDB layer operates behind the scenes to manage data storage and consistency within the system, therefore users don't interact with it.

2.2 Yugabyte query layer

Yugabyte query layer separates the query process from the storage management. Currently there are two APIs that can access the database:

YSQL: This API, designed to be PostgreSQL code-compatible, is accessible through standard PostgreSQL drivers, utilizing native protocols. It effectively replaces the storage engine with calls to the query layer while retaining compatibility with PostgreSQL features.

YCQL: YCQL adopts a Cassandra-like API which is implemented in C++. Standard Cassandra drivers can be used to interact with YCQL. YCQL enhances the core Cassandra components with additional capabilities such as transactional consistency, native support for JSON data types, and the ability to create secondary indexes on tables.

2.3 Clustering

YugabyteDB's clustering mechanism involves sharding data into tablets, replicating these tablets across multiple Tablet Servers using the Raft consensus algorithm, and efficiently balancing the query load across nodes.

Tablet Servers are responsible for managing and serving data. Each Tablet Server hosts one or more tablets, which are the basic unit of data distribution in YugabyteDB. These tablets contain ranges of data.

The Raft consensus algorithm is a distributed consensus algorithm designed to ensure that a distributed system of nodes can agree on a single, consistent state, even in the presence of network failures and node failures. Therefore granting YugabyteDB consistency and durability across replicas.

2.4 Chosen workload

The workload used to test the system was two types of benchmarks. A horizontal scalabiility benchmark to test the effect of increasing threads on the system performance and sysbench to test the effect of increasing the number of nodes and replication factor (which also entails node increase). The number of threads tested went from 1 to 256 in order to simulate an increasing scale in usage of the system, analagous to real rising system usage The replication factor used went from 1 to 9, but only odd numbers, to ensure a majority consensus can be achieved with the RAFT algorythm, so that performance could be tested with an increase in Master Servers, accompanying the number of nodes. The number of nodes used went from 1 to 13 in order to test performance with a cluster with more nodes than Master Servers so that the difference in the performance in relation to a cluster with the same number of nodes and Master Servers can be assessed.

All the metrics were tested in a local cluster network, since, after experimenting with Goocle Cloud Services the obtained results were very similar.

2.5 Pipeline Stages

A YugaByteDB request can be decomposed into a pipeline of four stages. Firstly, a client request. Secondly, the leader (elected master node through RAFT algorithm) receives the request and distributes requests across the various tablet servers, being responsible for balancing the distributed load. The third stage represents the tablet servers that receive the requests and execute them. This execution can include data retrieval from local storage or coordination with other tablet servers. At last, the leader aggregates the tablet servers' results and send the final result to the client

3 Results

3.1 Scalability and performance

The properties being tested in this section are throughput and latency regarding several factors, such as number of nodes, system's replication factor and number of threads. The first test done was the effect in throughput by increasing nodes, regarding read operations.

Even though there is a small amount of oscillation, for each fixed value of replication factor the throughput is stable even with an increasing number of nodes (Fig. 1), which denotes its consistency. It is also possible to notice that, with increasing the replication factor, the average throughput decreases (Fig. 1 & 2). In fact, when testing read operations in relation with replication factor (rf) it is possible to verify that latency increases and throughput decreases(Fig. 2). A possible explanation is due to the fact that increasing replication factor, increases the time to elect the leader among the master nodes.

In regard to testing write operations by increasing threads, it is possible to verify that increasing threads has a huge impact on throughput (Fig. 3a). It occurs due to the fact that, as threads run in parallel, the amount of transactions will increase and, although latency increases, this portion is not equivalent to the number of transactions' increase which results in a rise of throughput (Fig. 3a & 3b). There's a visible exception when studying the values of latency and throughput for 256 threads as the latency presents a huge rise. This factor is related to the system's inability to support these 256 threads in parallel, make it under heavy load and affecting the performance (Fig. 3a & 3b).

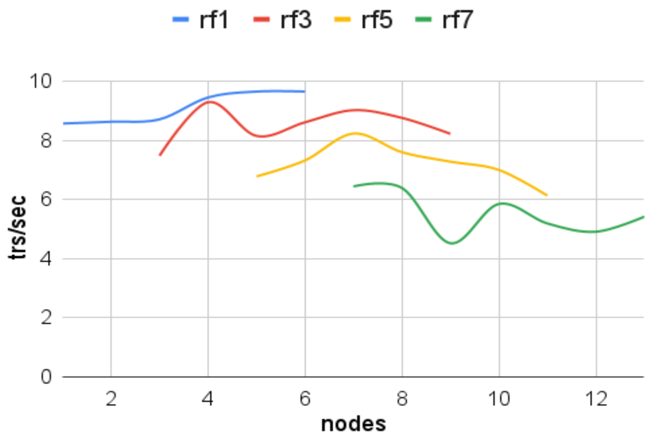
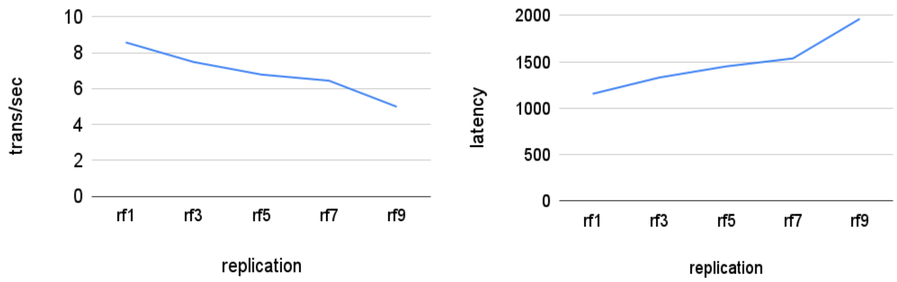


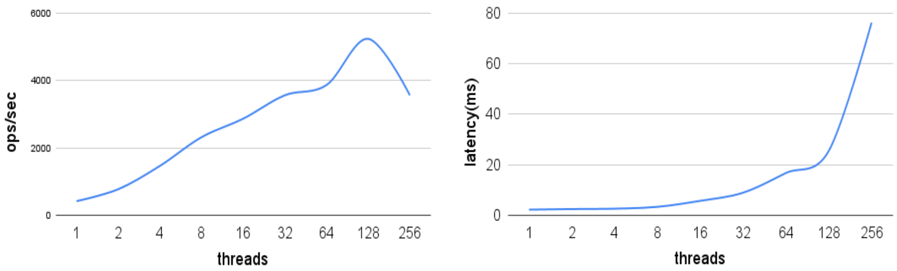
Fig. 1. Throughput in relation to number of nodes



(a) Throughput and replication factor

(b) Latency and replication factor

Fig. 2. Throughput and latency in relation to replication factor



(a) Throughput and threads

(b) Latency and threads

Fig. 3. Throughput and latency in relation to number of threads

At last, given the Universal Scalability Law

$$X(N) = \frac{\lambda N}{1 + \sigma(N-1) + kN(N-1)}$$

Bearing in mind the values obtained in the plot that relates throughput with threads (Fig. 3), it was possible to calculate lambda, sigma and k with the help of a jar file given in class, where $\lambda = 545.4865491216 \approx 545.49$, $\sigma = 0.1305164127$ ≈ 0.13 and k $= 0.0004571459 \approx 0.00046$, having that:

$$X(N) = \frac{545.49N}{1 + 0.13(N-1) + 0.00046N(N-1)}$$

3.2 Pipeline bottleneck

Both stage two and three can be a bottleneck depending on the system configuration and workload.

The leader can become a bottleneck if it is not able to validate requests and replicate them to the other nodes in the cluster quickly enough. This can be caused by a number of factors, such as a high CPU utilization on the leader machine, a slow network connection between the leader and the other nodes in the cluster or a very high replication factor as the leader election takes longer, increasing latency.

The tablet servers can become a bottleneck if they are not able to execute requests quickly enough. This can be caused by a number of factors, such as a high CPU utilization on the executor machines, a slow disk I/O performance or data storage under heavy load.

Techniques such as query optimization, indexing, and proper resource allocation can help alleviate bottlenecks and improve overall system performance.

4 Conclusion

To summarize, according to the YugabyteDB website, the experimental values matched with the expected results, such as transactions consistency regardless of the number of tservers (nodes). Latency, as expected, increases proportionally with the increase of replication factor. Moreover, by increasing parallel threads, latency increase is overshadowed by the transactions increase, until the system no longer supports the load.

Regarding Universal Scalability Law, it is possible to conclude that the serial (σ) and crosstalk (k) portions of the work are reduced which is important on scalable applications.

In general, YugabyteDb is an excelent solution to address demanding data needs if you are looking for any application that requires resilience, performance, and seamless scaling.

5 References

1. YB-Master service, https://docs.yugabyte.com/preview/architecture/concepts/yb-master/, last accessed 2023/10/11
2. Manual software installation, https://docs.yugabyte.com/preview/deploy/manual-deployment/install-software/, last accessed 2023/10/09
3. yb-ctl - command line tool for administering local YugabyteDB clusters, https://docs.yugabyte.com/preview/admin/yb-ctl/, last accessed 2023/10/09
4. Benchmark scaling YSQL queries, https://docs.yugabyte.com/preview/benchmark/scalability/scaling-queries-ysql/, last accessed 2023/10/10
5. Benchmark YSQL performance using sysbench, https://docs.yugabyte.com/preview/benchmark/sysbench-ysql/, last accessed 2023/10/10
6. YugabyteDB Quick start for Docker, https://docs.yugabyte.com/preview/quick-start/docker/, last accessed 2023/09/22
7. yb-docker-ctl - command line tool for administering local Docker-based clusters, https://docs.yugabyte.com/preview/admin/yb-docker-ctl/, last accessed 2023/10/08
8. YugabyteDB, https://en.wikipedia.org/w/index.php?title=YugabyteDB&oldid=1173416476, last accessed 2023/10/11
9. Raft (algorithm), https://en.wikipedia.org/w/index.php?title=Raft_(algorithm)&oldid=1165982346, last accessed 2023/10/11
10. Cluster topology, https://docs.yugabyte.com/preview/yugabyte-cloud/cloud-basics/create-clusters-topology/, last accessed 2023/10/10