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 systems· Benchmarks · Pipeline · Containerized applications · Docker

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 1a6e31bdd6a493dc821f8f3a5a48147a6e70f739

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

The 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.

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 scalability benchmark to test INSERT workload on the system and sysbench to test READ workload to follow the effect of increasing 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 the usage of the system, analogous 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 algorithm, so that performance could be tested with an increase in Master Servers, accompanying the number of nodes(TServers). The number of nodes used went from 1 to 13 in order to test performance on a cluster, so the difference in the performance in relation to a cluster with a different number of nodes but the same replication factor can be assessed.

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

2.5 Pipeline Stages

The YugaByteDB pipeline starts by receiving a client request. Then, the leader (elected master node through the RAFT algorithm) receives the request.

If the request is a write request, it is forwarded by the leader to the appropriate shard tablet server, the server validates the data and writes it in the shard's local storage, and then, the tablet server replicates the data to the other replicated servers and then waits for the other replicas to confirm the replication success. Finally, success is returned to the leader and then sent to the client.

If the request is a read request, the leader distributes the read as necessary to the tablet servers. The servers then read the necessary data and send it back to the leader who then aggregates all the read data and sends it back to the client.

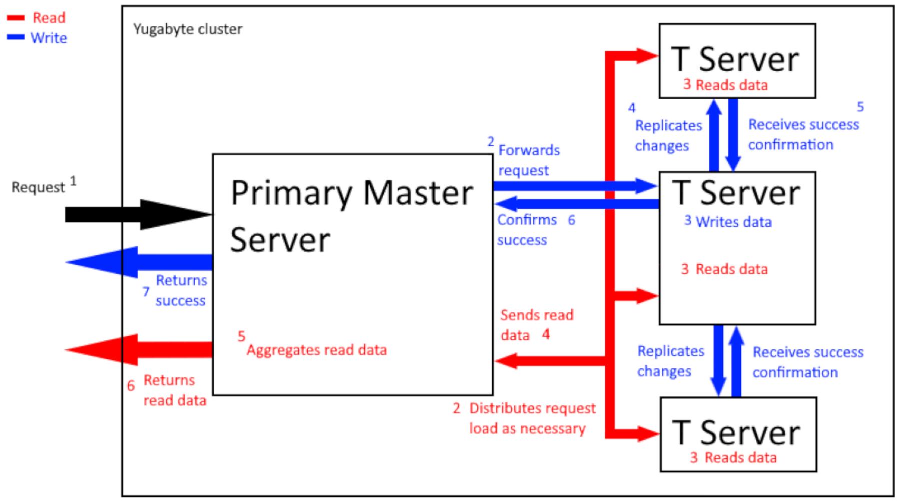
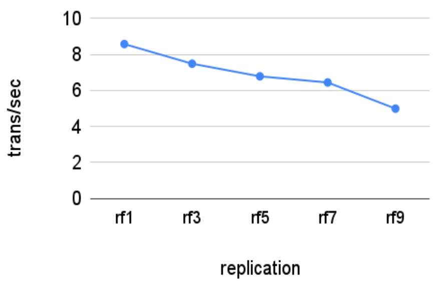


Fig. 1. Enter Caption

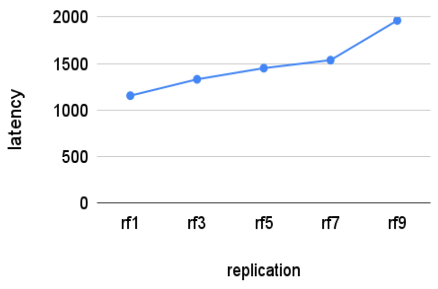
3 Results

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 and latency by increasing replication factor, regarding read operations. It is possible to notice that, with increasing the replication factor, the average throughput decreases (Fig. 2a) and latency increases (Fig. 2b). A possible explanation is due to the fact that increasing replication factor increases the time to respond because of possible leader election and more time costs for an internal communication between the nodes.



(a) Throughput and replication factor



(b) Latency and replication factor

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

In order to evaluate Yugabyte's horizontal scalability, by fixing replication factor as 3 (recommended by yugabyteDB website), the effect in throughput by increasing the number of nodes (TServers) was studied.

Given the Universal Scalability Law, bearing in mind the values obtained in the plot that relates throughput with number of nodes (Fig. 3), it was possible to calculate lambda, sigma and k with the help of a jar file given in class, where $\lambda = 3.9666278896 \approx 3.97$, $\sigma = 0.1963836829 \approx 0.20$ and k $= 0.0236547117 \approx 0.024$, having that:

$$X(N) = \frac{3.97N}{1 + 0.20(N-1) + 0.024(N-1)}$$

It is possible to conclude that YugabyteDB is horizontally scalable in terms of consistency and fault tolerance as data and workload distributed between multiple nodes. However, it is not scalable in terms of performance. Relatively same results were obtained with both sysbench and scalability benchmarks. The inversion in the USL line can be explained due to the fact that the system have no more resources to handle the increase of nodes.

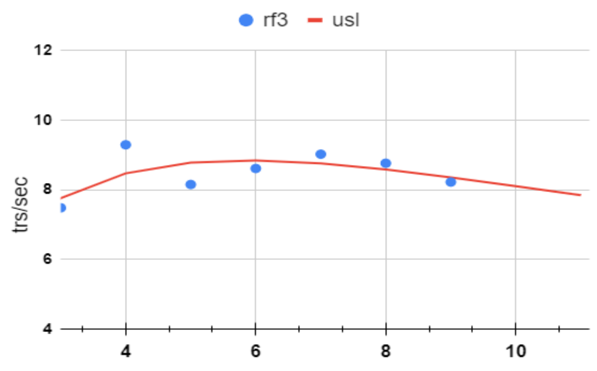


Fig. 3. Throughput in relation to number of nodes and respective USL

3.2 Pipeline bottleneck

Both the client and the servers can be a bottleneck depending on the system configuration and workload.

The client can be a bottleneck since the machine we were testing on was not capable of handling more threads.

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. It can be caused due to 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 the expected results, such as transaction consistency and horizontal scalability regarding the number of TServers (nodes). Latency, as expected, increases proportionally with the increase of the 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 in scalable applications.

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

5 References

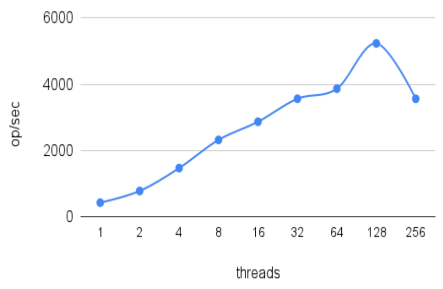
1. YB-Master service, https://docs.yugabyte.com/preview/architecture/concepts/yb-master/, last accessed 2023/10/11
2. Manual software installation, https://docs.yugabyte.com/preview/deploy/manual-deployment/install-software/, last accessed 2023/10/09
3. yb-ctl - command line tool for administering local 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

A Appendix

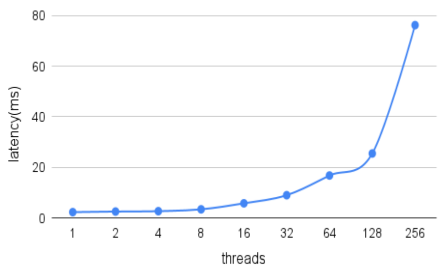
All of the values were experimented and tested on a system with the following characteristics:

1. 11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz 1.80 GHz
2. 32,0 GB RAM
3. 64-bit Windows 11

Additionally, the effect of increasing threads in throughput and latency in write operations was tested. It is possible to verify that increasing threads has a huge impact on throughput (Fig. 4a). It occurs since, 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. 4a & 4b). 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



(a) Throughput and threads



(b) Latency and threads

Fig. 4. Throughput and latency in relation to number of threads

threads in parallel, make it under heavy load and affecting the performance (Fig. 4a & 4b).

Due to a matter of time, the USL was tested for a fixed number of threads. We will improve it on stage 2.