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

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.

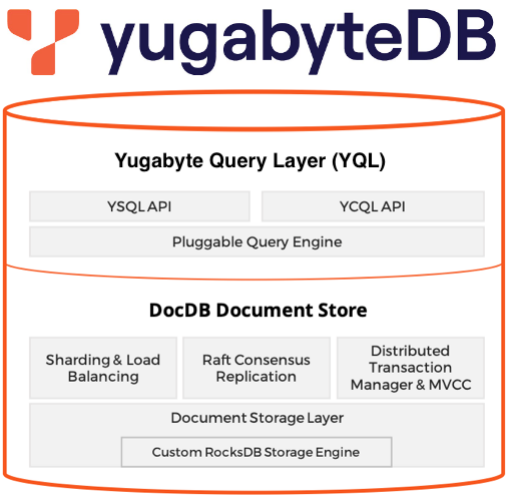

Fig. 1. YugabyteDB architecture

2.1 DocDB

DocDB serves as a distributed document store characterized by robust write consistency, exceptional fault tolerance, automatic sharding and load balancing, and the ability to intelligently place data based on zones, regions, and cloud considerations. It also offers adjustable read consistency.

Data is organized within tables in DocDB. Typically, a DocDB table is partitioned into multiple transparently sharded tablets.

Each tablet, containing user data, undergoes replication with a defined replication factor using the Raft consensus algorithm. Replication occurs at the tablet level, ensuring that even in the face of system failures, single-row linearizability is maintained.

To persist data, DocDB employs a log-structured storage approach that caters to both row- and document-oriented data. This approach includes several optimizations to efficiently manage constantly growing datasets.

DocDB offers support for both single-row and multi-row transactions. This capability enables concurrent modifications of multiple keys while preserving the critical ACID (Atomicity, Consistency, Isolation, Durability) properties.

2.2 Yugabyte query layer

The YugabyteDB Query Layer, often referred to as YQL, serves as the primary interface for applications, which communicate with it through client drivers. This layer is responsible for handling API-specific tasks like query and command compilation, in addition to managing runtime functions such as data type representations and built-in operations. YQL is intentionally designed with extensibility in mind, making it possible to incorporate new APIs. Presently, YQL offers support for two distributed SQL APIs: YSQL and YCQL.

YSQL YSQL, short for Yugabyte Structured Query Language is a ANSI SQL API that offers comprehensive support for traditional relational modeling features. This includes essential aspects like referential integrity, which is maintained via foreign key constraints (linking child tables to primary keys in their parent tables), as well as support for joins, partial indexes, triggers, and stored procedures. YSQL seamlessly extends the well-known transactional concepts into the architecture of the YugabyteDB Database.

The key components of YSQL encompass the Data Definition Language (DDL), Data Manipulation Language (DML), Data Control Language (DCL), built-in SQL functions, and the PL/pgSQL procedural language designed for stored procedures

YCQL The Yugabyte Cloud Query Language (YCQL) is a semi-relational SQL API tailored for internet-scale OLTP (Online Transaction Processing) and HTAP (Hybrid Transaction/Analytical Processing) applications that demand high-speed data ingestion and lightning-fast queries. YCQL offers robust support for strongly consistent secondary indexes, a native JSON column type, and distributed transactions. Its origins can be traced back to the Cassandra Query Language (CQL).

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.

Sharding This technique entails the dispersion of data across numerous nodes within a cluster, with each node handling a distinct data segment. This division of data into smaller segments and its distribution among multiple nodes enables YugabyteDB to achieve parallel processing and efficient load distribution. In the event of a node failure, the sharded design of YugabyteDB ensures that the remaining nodes can assume the role of serving the data, thereby preserving availability.

RAFT Raft serves as a consensus algorithm enabling participants in a distributed system to reach a consensus on a series of values, even when faced with failures. More precisely, it has emerged as the preferred protocol for constructing robust, strongly-consistent distributed systems.

Yugabyte employs the Raft consensus algorithm for both leader selection and data replication. Rather than utilizing a single Raft group for the entire dataset across the cluster, Yugabyte implements Raft replication on a per-shard basis, with each shard possessing its dedicated Raft group.

Nodes Each node contains a Tablet Server (TServer) and, depending of the replication factor (rf), it may have a Master Server.

The TServers oversee the input and output (I/O) operations for end-user requests within a YugabyteDB cluster. Tables are divided (sharded) into tablets, with each tablet consisting of one or more tablet peers, contingent upon the replication factor. Each TServer accommodates one or more tablet peers.

The YB-Master manages system metadata and crucial records, including table details and the locations of their tablets, user accounts, roles with associated permissions, and more. Moreover, the YB-Master takes charge of orchestrating tasks like load balancing and initiating the re-replication of data that is under-replicated. It also handles various administrative tasks, such as creating, modifying, and deleting tables.

Notably, the YB-Master is designed for high availability. It forms a Raft group with its peers, ensuring data integrity and consistency, and it operates independently of the critical path for user table input-output operations.

2.4 Chosen workload

The workload used to test the system was a horizontal scalability benchmark to test INSERT workload (write operations) on the system to follow the effect of increasing number of nodes, RAM, processor cores, max user processes, max locked memory and replication factor (which also entails node increase). In this sense, each VM represents a different node, which means that in order to test the increase of nodes, it was necessary to create a new VM instance in Google Cloud. It is possible to change each VM RAM and vCPUs meaning that to test those two factors, each VM was configured and changed accordingly, except for the one that was running the benchmark. To test replication factor, it is only necessary to change the cluster's configuration. To test max user processors and max locked memory, it is necessary to change the Linux configuration properties in each VM with ulimit.

The chosen cluster was an official Yugabyte cluster called Yugabyted. It is only necessary to install locally on each VM Yugabyte and start the cluster in a main node and start in other instances with a flag (–join).

For consistency, the benchmark was ran on a separate Google Cloud VM with 4 vCPUs (2 cores) and 32GB of RAM and its number of write threads was dependant from the number of nodes used. For testing the 6 factors, when testing the impact of increasing nodes, by each node there were 100 threads added. When testing replication factor, as the higher tested value was a factor of 7, meaning that we used a cluster of 7 nodes, the number of write threads in the benchmark was 700. For the remaining factors, a cluster of 3 nodes was used, but instead of using 300 threads, the system was pushed closer to the limit with a healthy and safe number of 400 write threads.

2.5 Pipeline Stages

Write If a client wants to update a row in tablet3 and, and the initial destination for this request is yb-tserver2, it is automatically rerouted to the current master-leader, yb-master3, to obtain the location of tablet3's leader node. This location is then stored in the client driver's cache to eliminate the need for future interactions with the master-leader. This location is then stored in the client driver's cache to eliminate the need for future interactions with the master-leader. The actual update request is then directed to the node hosting the tablet3-leader, which in this case is yb-tserver3. The tablet3-leader adds the update to its own Raft log and replicates it to the Raft logs of the two follower nodes. Once this occurs, the tablet3-leader marks the update as committed in its own Raft log, applies the update to the memtable of its locally stored RocksDB-based data, and sends an acknowledgment to the client, confirming the successful update.

Read With the consistency task already addressed during the write phase, YugabyteDB's Raft implementation guarantees the ability to fulfill read requests with low latency by eliminating the need for quorum-based consultations with other replicas. Furthermore, it offers the application client the flexibility to select whether to read from the leader (for strong consistency) or from the followers (for timeline-consistent reads).

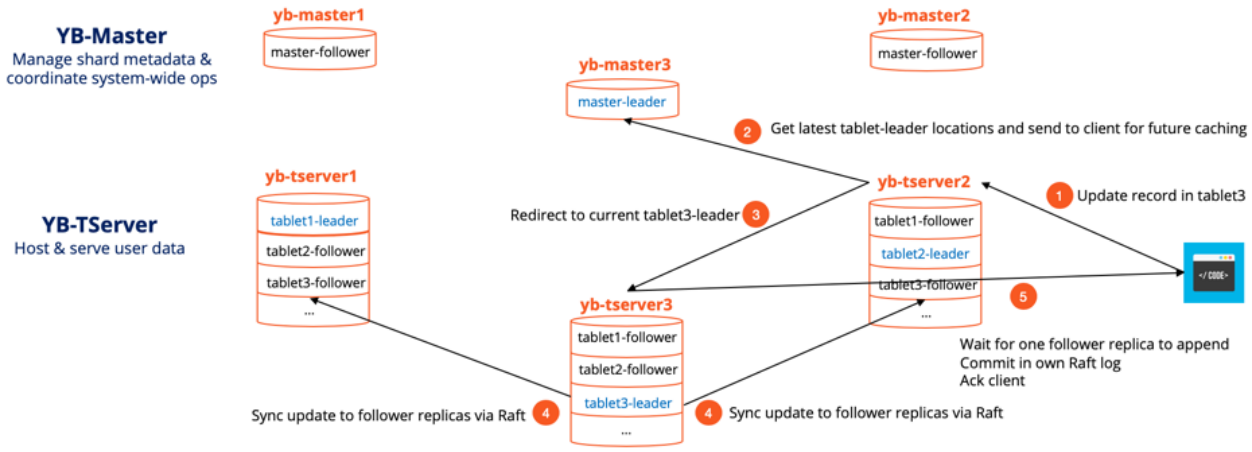

Fig. 2. Single-row write

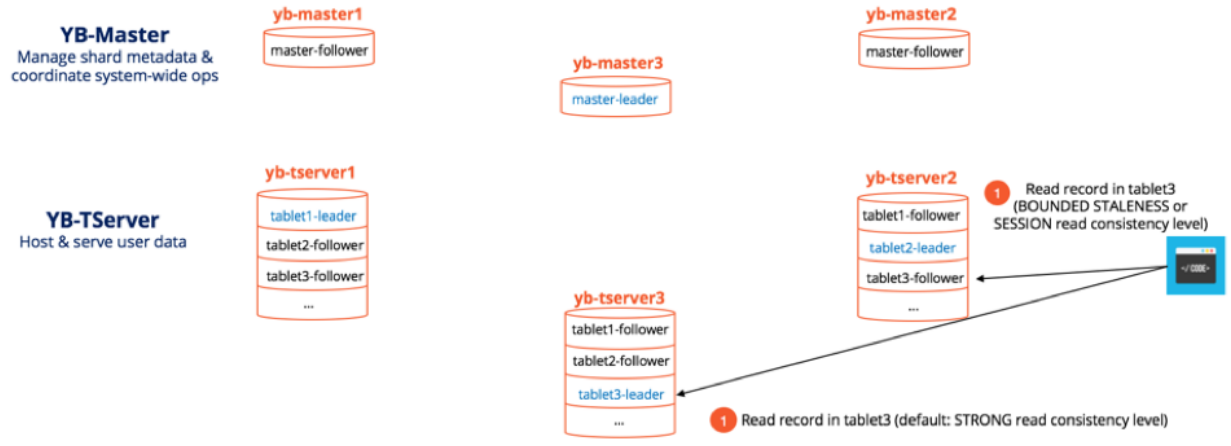

Fig. 3. Quorumless read

3 Results

3.1 Scalability and performance

Individual factor tests In order to benchmark YugabyteDB regarding scalability and performance there were six factors tested, more precisely, number of nodes, RAM, number of processor cores, replication factor, max locked memory and max user processes. Max locked memory refers to memory pages that are pinned in RAM and cannot be swapped out to disk. Max user processes determines the maximum number of user processes that can connect to YugabyteDB. Due to Google Cloud vCPUs limitation to 32 for all regions, factors like number of processor cores, number of nodes and replication factor have fewer number of observations.

Firstly, the throughput was tested regarding the increase of number of nodes and threads proportionally (Fig. 4) in multiple Google Cloud VMs with 16GB RAM, 2VCPUs (1 processor core), replication factor of 3. For each added node, the number of write threads was also increased by 100, where the starting point was 3 nodes and 300 write threads. Due to the aforementioned limitation, there was a maximum number of nodes of 14. In the appendix (Fig. 10) it is possible to observe an USL test done with fixed 400 write threads, 4 vCPUs (2 cores), replication factor of 3 and 32GB RAM. It is possible to conclude that when the cluster has 7 nodes, the client became a bottleneck since the performance didn't increase which is explained by the fact that 6 nodes are enough to handle and answer the threads, meaning that adding a node will just overwhelm more resources.

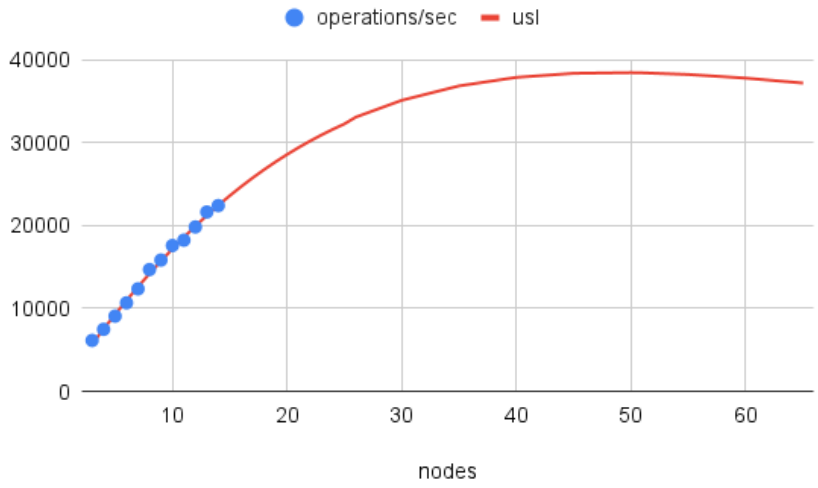

Fig. 4. Nodes in relation to throughput

Given the Universal Scalability Law, bearing in mind the values obtained in the plot that relates throughput with number of nodes (Fig. 4), it was possible to calculate lambda, sigma and k with the help of a jar file given in class, where $\lambda = 1949.7470062964 \approx 1949{,}75$, $\sigma = 0.0104283094 \approx 0{,}10$ and k = $0.0004187466 \approx 0{,}00042$, having that:

$$X(N) = \frac{1949,75N}{1 + 0,10(N-1) + 0,00042N(N-1)}$$

It is possible to conclude that YugabyteDB is horizontally scaling in terms of consistency, fault tolerance and performance. According to the USL it is expected that the higher number of nodes for a replication factor of 3 and 100 write threads per node is 50 nodes.

Then, the behaviour of increasing RAM was tested (Fig. 5) with 400 write threads. As increasing RAM may also affect number of cores, the number of cores was fixed as 4 (8 vCPUs) for 3 nodes with replication factor of 3, allowing the RAM to vary between 4GB to 64GB.

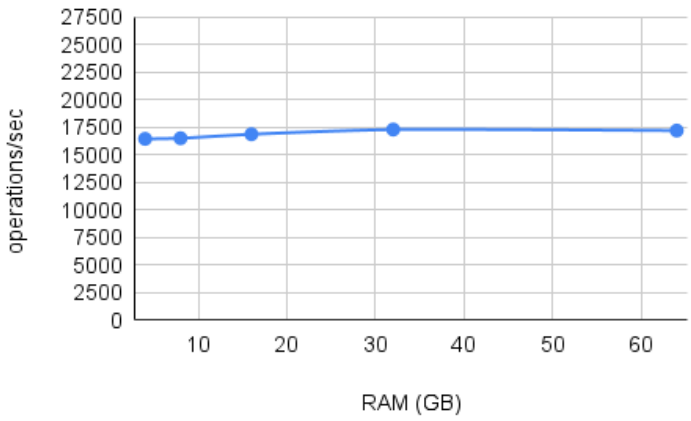

Fig. 5. RAM in relation to throughput

It is possible to verify and there is a slight increase of performance when increasing RAM. This happens due to the fact that with more RAM, data can be kept in memory for longer, reducing the frequency and intensity of disk reads and writes and minimizing contention for disk resources, resulting in an improved throughput. Moreover, not only RAM can be used to cache frequently accessed data and indexes, which reduces the need to fetch data from slower storage devices but also allows YugabyteDB to allocate larger memory pools for query processing, enabling it to handle more complex queries and larger datasets without swapping data to disk.

The number of processor cores was also a factor tested (Fig. 6). This factor was highly limited by Google Cloud. Therefore, the best way to test the variance of cores were running a 3 node cluster with replication factor of 3, where each VM had 16GB RAM. The benchmark ran 400 write threads.

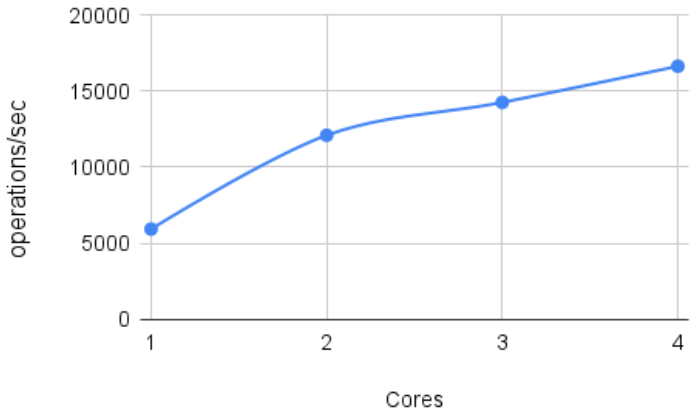

Fig. 6. Cores in relation to throughput

It is possible to verify that cores have a lot of impact in Yugabyte's performance. By increasing processor cores, it is possible to strengthen parallel processing, where YugabyteDB can distribute workloads across the cores, enabling simultaneous execution of operations. Additionally, increasing processor cores alleviates the processes concurrency, reducing latency and boosting throughput.

The following factor tested was the increase of replication factor (Fig. 7), for fixed 7 nodes, 4 vCPUS (2 cores), 32GB RAM and 400 write threads.

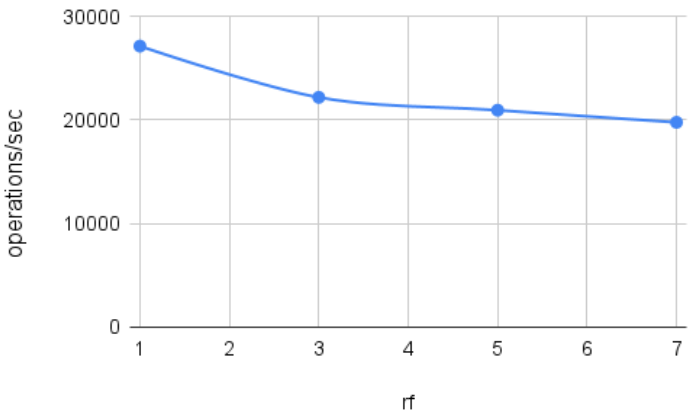

Fig. 7. Replication Factor in relation to throughput

It is possible to verify that increasing replication factor means decreasing performance since each write operation needs to be replicated to more nodes, resulting in increased network traffic, leading to higher latency. In addition, more resources are dedicated to maintaining replicas. YugabyteDB uses the RAFT consensus algorhitm as descreibed in System description section. As the replication factor increases, the consensus process involves more nodes, requiring more communication and coordination, impacting overall throughput. It is important to refer that, although increasing replication factor from 1 to 3 decreases throughput, this reduction is a trade-off for fault tolerance and data durability.

Regarding max user processes testing (Fig. 8), there were 3 nodes with replication factor of 3 where each VM instance had 32GB RAM and 4 vCPUs and the benchmark ran 400 write threads.

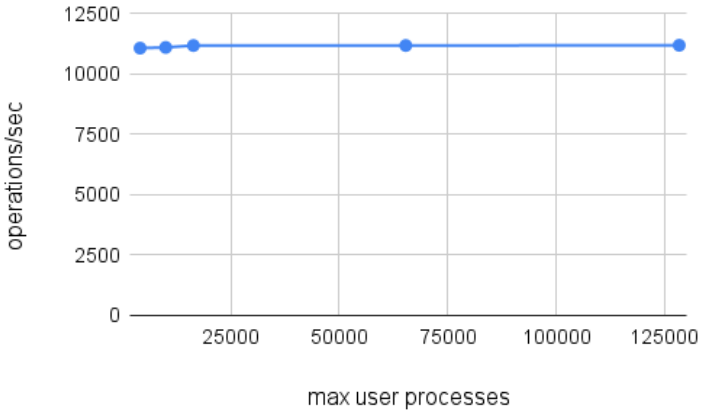

Fig. 8. Max user processes in relation to throughput

it is possible to verify that there is a minimal increase in throughput. On the one hand, by having a higher limit means that there is less process concurrency and it enables more parallelism increasing the throughput. On the other hand, this rise is not very considerable since limiting max user processes also means that the system can prioritize database operations and reduce the impact of other background processes or unrelated tasks.

The last factor tested independently was max locked memory (Fig. 9) for 3 fixed nodes, 4 vCPUS (2 cores), 32GB RAM and 400 write threads.

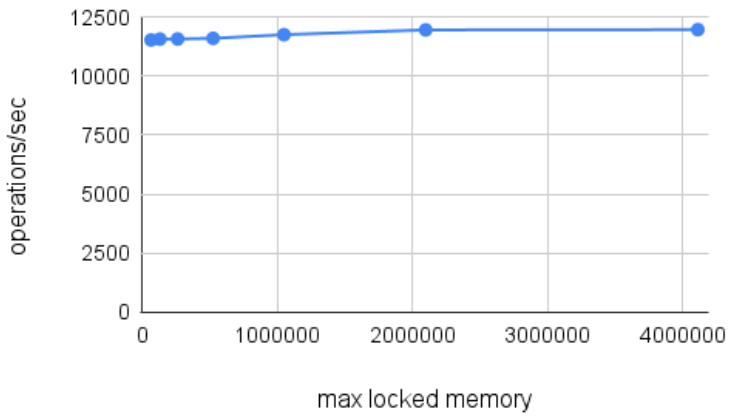

Fig. 9. Max locked memory in relation to throughput

It is possible to verify that there is a slight increase in throughput. This is related to YugabyteDB's necessity to access memory pages, since increasing the maximum locked memory allows YugabyteDB to keep more frequently accessed data in physical memory, reducing page faults and improving throughput.

2^{k-p} Fractional Factorial Designs After analyzing all six factors independently, it is now possible to start computing the experimental table. As number of nodes, number of processor cores and replication factor were the most impactful ones, they will be the independent factors from the table. The remaining three factors (RAM, max user processes and max locked memory) will be co-founded so as to decrease the number of necessary experiments. For each factor, it was defined the low and high level:

- nodes: lower = 3 nodes, higher = 7 nodes;
- processor cores: lower = 1, higher = 2;
- replication factor: lower = 1, higher = 3;
- RAM: lower 2GB, higher = 16GB;
- max user processes: lower = 4096, higher = 7842;
- max locked memory: lower = 64KB, higher = 253648KB

It is important to bear in mind that these lower and higher cases were highly conditioned due to factors dependence regarding the Google Cloud limitation.

In order to fill the table the following rules were adopted:

1. RAM column is given by Nodes x Processor Cores (AxB);
2. Max Locked Memory column is given by Processor Cores x Replication factor (B x C)
3. Max User Processes column is given by Nodes x Processor Cores x Replication factor (A x B x C)

Having the following table:

Experiment	Nodes	Processor Cores	Replication factor	RAM	Max Locked Memory	Max User Processes
1	-1	-1	-1	1	1	-1
2	1	-1	-1	-1	1	1
3	-1	1	-1	-1	-1	1
4	1	1	-1	1	-1	-1
5	-1	-1	1	1	-1	1
6	1	-1	1	-1	-1	-1
7	-1	1	1	-1	1	-1
8	1	1	1	1	1	1

The effect of each factor is calculated by:

$$\begin{aligned}
q_A &= \frac{-y1 + y2 - y3 + y4 - y5 + y6 - y7 + y8}{8} \\
&= \frac{-7099 + 14958 - 13665 + 18691 - 6316 + 11435 - 11471 + 15147}{8} = 2710 \\
q_B &= \frac{-y1 - y2 + y3 + y4 - y5 - y6 + y7 + y8}{8} \\
&= \frac{-7099 - 14958 + 13665 + 18691 - 6316 - 11435 + 11471 + 15147}{8} = 2395.75 \\
q_C &= \frac{-y1 - y2 - y3 - y4 + y5 + y6 + y7 + y8}{8} \\
&= \frac{-7099 - 14958 - 13665 - 18691 + 6316 + 11435 + 11471 + 15147}{8} = -1255.5 \\
q_D = q_{AB} &= \frac{y1 - y2 - y3 + y4 + y5 - y6 - y7 + y8}{8} \\
&= \frac{7099 - 14958 - 13665 + 18691 + 6316 - 11435 - 11471 + 15147}{8} = -534.3 \\
q_E = q_{BC} &= \frac{y1 + y2 - y3 - y4 - y5 - y6 + y7 + y8}{8} \\
&= \frac{7099 + 14958 - 13665 - 18691 - 6316 - 11435 + 11471 + 15147}{8} = -179 \\
q_F = q_{ABC} &= \frac{-y1 + y2 + y3 - y4 + y5 - y6 - y7 + y8}{8} \\
&= \frac{-7099 + 14958 + 13665 - 18691 + 6316 - 11435 - 11471 + 15147}{8} = 173.75
\end{aligned}$$

The total variation of each factor is given by:

$$SS_A = 2^3 \times q_A^2 = 58752800$$

$$SS_B = 2^3 \times q_B^2 = 45916944.5$$

$$SS_C = 2^3 \times q_C^2 = 12610242$$

$$SS_D = SS_{AB} = 2^3 \times q_D^2 = 2283811.92$$

$$SS_E = SS_{BC} = 2^3 \times q_E^2 = 256328$$

$$SS_F = SS_{ABC} = 2^3 \times q_F^2 = 241512.5$$

$$SS_T = SS_A + SS_B + SS_C + SS_D + SS_E + SS_F = 120061638.9$$

And the respective percentages of effect are:

$$\text{Effect of Nodes (A)} = \frac{SS_A}{SS_T} \times 100 \approx 48,94\%$$

$$\text{Effect of Processor Cores (B)} = \frac{SS_B}{SS_T} \times 100 \approx 38,24\%$$

$$\text{Effect of Replication Factor (C)} = \frac{SS_C}{SS_T} \times 100 \approx 10,50\%$$

$$\text{Effect of RAM (D)} = \frac{SS_D}{SS_T} \times 100 \approx 1,90\%$$

$$\text{Effect of Max Locked Memory (E)} = \frac{SS_E}{SS_T} \times 100 \approx 0,21\%$$

$$\text{Effect of Max User Processes (F)} = \frac{SS_F}{SS_T} \times 100 \approx 0,20\%$$

The results obtained in the experimental table match with the study done on the six factors individually. Increasing nodes and processor cores boost overall system's performance, with replication factor having a fewer but also impactful role.

3.2 Pipeline bottleneck

Both the masters and tablet servers 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. A way to mitigate this bottlenck is to vertically scale system's resources.

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. A way to solve this bottleneck is to vertically scaling the system, allocating more resources. Since we studied that RAM does not have so much impact, a good way to solve the bottleneck is to focus on scaling processor cores. In addition, as studied, horizontal scaling such as adding nodes can prevent this bottleneck.

Another possible bottleneck is the network bandwidth since Yugabyte relies on a network to communicate between nodes. A possible way to mitigate this bottleneck is to deploy nodes in the same region or with faster network connection.

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, decreasing throughput, which is a trade-off for fault tolerance. Moreover, by increasing number of processor cores, throughput increases substantially, making it possible to conclude that YugabyteDB is also vertically scalable.

Regarding Universal Scalability Law, it is possible to conclude that the serial (σ) and crosstalk (k) portions of the work are reduced which denotes that yugabyteDB is a scalable system.

Regarding 2^{k-p} Fractional Factorial Designs, results completely matched the tests done for individual factors, where number of nodes, number of processor cores and replication factor are the major factors to bear in mind. Although RAM, max user processes and max locked memory may also increase performance, they only have major impact in some rare and specific circumstances.

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. Benchmark scaling YSQL queries, https://docs.yugabyte.com/preview/benchmark/scalability/scaling-queries-ysql/, last accessed 2023/10/20
2. Cluster topology, https://docs.yugabyte.com/preview/yugabyte-cloud/cloud-basics/create-clusters-topology/, last accessed 2023/10/10
3. How Does the Raft Consensus-Based Replication Protocol Work in YugabyteDB?, https://www.yugabyte.com/blog/how-does-the-raft-consensus-based-replication-protocol-work-in-yugabyte-db/, last accessed 2023/10/26
4. Manual software installation, https://docs.yugabyte.com/preview/deploy/manual-deployment/install-software/, last accessed 2023/10/23

5. Overview of YugabyteDB Query Layer, https://docs.yugabyte.com/preview/architecture/query-layer/overview/, last accessed 2023/10/25
6. Raft (algorithm), https://en.wikipedia.org/w/index.php?title=Raft_(algorithm)&oldid=1165982346, last accessed 2023/10/11
7. Raft Protocol: What is the Raft Consensus Algorithm?, https://www.yugabyte.com/tech/raft-consensus-algorithm/, last accessed 2023/10/24
8. System configuration, https://docs.yugabyte.com/preview/deploy/manual-deployment/system-config/, last accessed 2023/10/23
9. YSQL - Yugabyte SQL for distributed databases, https://docs.yugabyte.com/preview/api/ysql/, last accessed 2023/10/25
10. Yugabyte Cloud Query Language (YCQL), https://docs.yugabyte.com/preview/api/ycql/, last accessed 2023/10/25
11. YugabyteDB, https://en.wikipedia.org/w/index.php?title=YugabyteDB&oldid=1173416476, last accessed 2023/10/11
12. yugabyted reference, https://docs.yugabyte.com/preview/reference/configuration/yugabyted/, last accessed 2023/10/24
13. yb-ctl - command line tool for administering local YugabyteDB clusters, https://docs.yugabyte.com/preview/admin/yb-ctl/, last accessed 2023/10/24
14. YB-Master service, https://docs.yugabyte.com/preview/architecture/concepts/yb-master/, last accessed 2023/10/25
15. YB-TServer service, https://docs.yugabyte.com/preview/architecture/concepts/yb-tserver/, last accessed 2023/10/25

A Appendix

As explained in results section, a fixed number of threads USL was plotted. This allowed us to conclude that 400 write threads for 3 nodes are a safe number of threads, even thought, edging to the limit and the necessity for scaling threads proportionally.

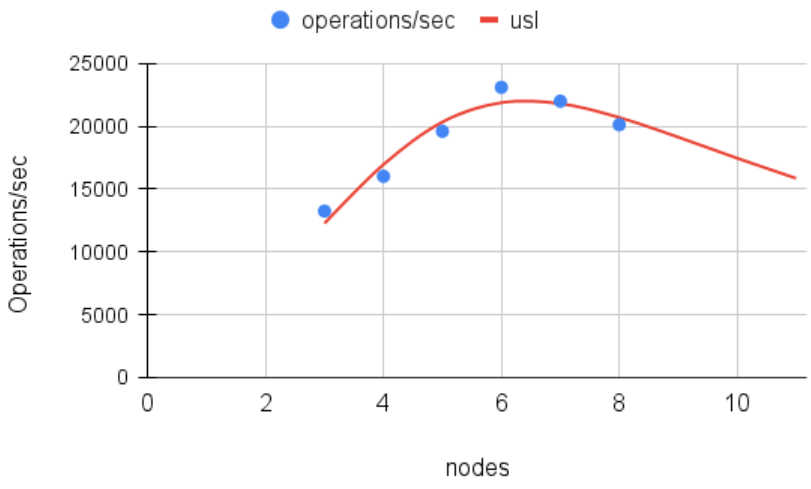

Fig. 10. Nodes in relation to throughput and USL