Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Orhan Henrik Hirsch

Scalability of NewSQL Databases in a Cloud Environment

Master's thesis in Computer Science
Supervisor: Svein Erik Bratsberg
June 2020

NTNU
Norwegian University of Science and Technology

Orhan Henrik Hirsch

Scalability of NewSQL Databases in a Cloud Environment

Master’s thesis in Computer Science
Supervisor: Svein Erik Bratsberg
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

In recent years, there has been an increased demand for NewSQL databases, which are systems that both scale horizontally and can guarantee transaction isolation and consistency. NewSQL systems are quite new, and little research exists about the difference between their architecture and performance. In this thesis, we investigate the open source NewSQL systems CockroachDB, TiDB and YugabyteDB in depth, and perform several evaluations of their performance characteristics.

To evaluate these databases, we have built a novel automated testing approach, which can create and evaluate clusters without any manual intervention. During our research, we used this system to automatically test over 200 separate clusters using more than 1,500 virtual machines in a cloud, which speaks to the success of this approach.

During our evaluation, we found that the performance of the databases is comparable, but that they have different trade-offs. CockroachDB and serializable YugabyteDB provide higher levels of transaction isolation than TiDB and snapshot YugabyteDB, but for write operations, they have lower throughput and higher latency. However, CockroachDB outperforms the other systems in reads, which means that the transaction isolation trade-off only affects write operations.

Sammendrag

De siste årene har det vært en økt etterspørsel etter NewSQL databaser, som er systemer som både skalerer godt horisontalt, men også kan garantere isolasjon og konsistens mellom transaksjoner. NewSQL er et ganske nytt område, og det finnes lite forskning om forskjellene i databasenes ytelse og arkitektur. Denne oppgaven undersøker tre NewSQL systemer i dybden og evaluerer deres ytelse. Disse systemene er CockroachDB, TiDB og YugabyteDB, og har alle åpen kildekode.

For å evaluere disse databasene har vi bygget en ny automatisk testmetode som kan opprette og evaluere databaseclustere uten manuelle handlinger. Gjennom vårt forskningsarbeid har vi brukt denne metoden til å automatisk evaluere over 200 clustere ved bruk av over 1,500 virtuelle maskiner i en sky. Størrelsen på disse tallene viser at denne testmetoden har vært vellykket.

Gjennom evalueringen av ovennevnte databasesystemer fant vi at ytelsen deres er sammenlignbar, men at systemene har ulike kompromisser. CockroachDB og serialiserbar YugabyteDB har sterkere transaksjonsgarantier enn TiDB og øyeblikksbilde YugabyteDB, men for skriveoperasjoner har de lavere datagjennomstrømming og høyere forsinkelse. For leseoperasjoner oppnår derimot CockroachDB bedre resultater enn de andre systemene, hvilket betyr at kompromisset kun påvirker ytelsen til skriveoperasjoner.

Preface

This thesis is written during the spring of 2020 for the Department of Computer Science at Norwegian University of Science and Technology, and is the final work for a Master of Science degree. The research was conducted by Orhan Henrik Hirsch, and was supervised by Professor Svein Erik Bratsberg.

I am very thankful to Svein Erik Bratsberg for his valuable input during this work, and that he has given me the opportunity to shape my own project.

Finally, I would like to thank Hetzner Cloud for a generous research grant that allowed me to perform this research on their cloud platform.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

With the increasing volume of data that is being generated and processed in the last 20 years, new database systems that scale better than traditional systems have emerged [17]. Many of these systems sacrifice consistency and isolation guarantees to be able to scale well, and such systems are often categorized as NoSQL databases. Although these systems provide much better scalability than traditional database systems, their reduced guarantees mean that it's the application's responsibility to handle inconsistencies. For some applications, like financial systems, this requirement makes these databases infeasible to use [17].

As a response to these challenges, much research has been done into a new class of database systems that can scale well while also guaranteeing isolation and consistency for transactions. These databases are usually referred to as NewSQL, and offer an SQL interface with traditional ACID guarantees, while also being failure tolerant and scalable. Many of these systems are inspired by the architecture of Google's Percolator [18] and Spanner [7], while another novel architecture was introduced by Calvin [22].

Three popular open source NewSQL databases are CockroachDB [6], TiDB [23] and YugabyteDB [26]. These are inspired by Google Spanner and Google Percolator, and take different approaches to scalability and transaction handling. The three databases are all built to work well in a cloud environment, and are highly focused on providing horizontal scalability, which means that applications using these databases are no longer limited by the performance of a single machine. However, there exists no research that compares the three systems in depth, and it is unclear what the performance trade-offs of each system are. In this thesis, we want to address this issue by investigating these databases further to uncover their differences in architecture and performance.

To compare these database systems in a fair way, we need to run different evaluations on several cluster configurations. This is a time consuming and error prone task, as each evaluation requires a cluster to be configured across multiple machines. To efficiently do this for the aforementioned systems in a cloud environment, we need to automate the processes for starting clusters and running evaluations. Another advantage of automating the evaluations is that it enables easier reproduction of test results.

1.2 Research Goals

1. Compare the performance of CockroachDB, YugabyteDB and TiDB in a cloud environment.
2. Investigate how well each database scales vertically and horizontally.
3. Create an automated system for running database benchmarks in a cloud environment.

1.3 Thesis Structure

- **Chapter 1 – Introduction** describes the background for this thesis and outlines our research goals.
- **Chapter 2 – Theoretical Background** presents the background knowledge that we find relevant for this thesis. We also describe in depth the database systems we plan to evaluate.
- **Chapter 3 – Implementation** thoroughly describes our automated benchmarking system, and outlines which workloads we are evaluating the database systems with.
- **Chapter 4 – Results and Discussion** presents and discusses the results of our benchmarks, and compares the performances of the aforementioned database systems.
- **Chapter 5 – Conclusion and Future Work** concludes our work, and presents our thoughts on possible improvements to our benchmarking system and database comparisons.

Chapter 2

Theoretical Background

This chapter describes the theoretical background that supports this thesis. First, we describe some useful concepts for understanding distributed databases, and next describe the relevant databases for this thesis in depth. As mentioned in Section 1.1, the NewSQL databases we cover are CockroachDB, TiDB and YugabyteDB. For each database, we describe how their architecture and storage engines are designed, as well as how they handle transactions and geo-replication. This chapter is based on work from our specialization project [13].

2.1 CAP Theorem

The CAP Theorem [9], created by Eric Brewer, states that a distributed system can not provide both consistency, availability and partition tolerance at the same time. The implication of this is that any distributed system must choose at most two of these properties. Since networks are unreliable, any distributed system must support partition tolerance, and thus, distributed systems on a network can choose to be either consistent (CP) or available (AP). A CP system is a system that stays fully consistent during a network partition, but will not be fully available. Usually, this means that the smaller partition or sometimes the whole database becomes unavailable during a partition. An AP system chooses to always stay available at the cost of consistency. During a network partition, different partitions may not be in states that are compatible, and in this case, one of the states needs to be chosen while the other is discarded.

The CAP Theorem states a very simple fact about the trade-offs in distributed systems during partitions, but does not describe trade-offs that can be made while the network is healthy. To address this, Daniel J. Abadi created a new theorem called PACELC [1]. The theorem states that during a partition, a system must choose either availability or consistency, but otherwise, it must prioritize either low latency or consistency. This implies that in order to achieve the lowest latency while the network operates normally, a distributed database must sacrifice some consistency guarantees.

2.2 ACID

ACID are four common properties of transaction that many relational databases guarantee, and can make it easier for developers to reason about how transactions affect the system's state. We have described each property below [16, 4].

Atomicity

A transaction being atomic means that either the whole transaction is executed or it is not executed at all, i.e. no transaction can be only partially complete.

Consistency

Consistency means that transactions starting at the same time see the same state. If different transactions see different states at the same time, the states are said to be inconsistent. For example, if a database does not provide consistency and reads are initiated in two different regions, they may see two different versions of the same data keys. In addition to this definition of consistency, transactions must be visible in the order that they were executed. For example, if transaction B reads data that transaction A wrote, and transaction C sees the writes that B performed, C must also see the writes that A performed.

Isolation

Isolation between transactions means that transactions do not see effects from other transactions that are in progress. Perfect isolation essentially means that it appears to transactions as if they are running sequentially rather than concurrently. Isolation is very helpful to developers as they do not need to consider every way a set of transactions may interact if running concurrently. There are many different levels of isolation that are used in databases, and in general, the more strict isolation levels come at a significant reduction in performance [2]. We have covered some different isolation levels in Section 2.3.

Durability

Durability is the property that, when a transaction is acknowledged as committed, it will stay that way forever. No transaction can be rolled back after it is committed, which means that a user can trust the system to store their data. In order to achieve durability, databases need to store a record of each transaction to a non-volatile storage medium like a hard disk before acknowledging it.

2.3 Isolation levels

Because perfect isolation, also known as *strict serializability*, has a significant performance impact [2], many lower levels of isolation are used in common databases. In order to understand the differences between isolation levels, a set of common anomalies have been defined and are listed below [2]. Anomalies are events that occur where a transaction can observe that it is not the only one running, i.e. not running sequentially. Strict serializability does not allow any of these anomalies to occur.

- **Lost update:** A lost update occurs if two different transactions read the same key at the same time followed by a write. If both transactions e.g. decrement a field at the same time, they may overwrite each other without realizing, resulting in both transaction committing but the value only being decremented once. This is is an isolation anomaly, as a sequential execution would result in a different state than this concurrent execution.
- **Dirty write:** Dirty writes occur when a transaction reads a value that another transaction has written but not yet committed. If the second transaction then performs some action based on the first, and the first aborts while the second commits, the state of the database can become inconsistent. Taking the same example as above, if transaction A decrements a value, and B decrements it afterwards, but then transaction A aborts and B commits, you now have one committed transaction but the value is decremented by two. A sequential execution of transactions in this case would also result in a different state, as the second transaction would never see the aborted first transaction.
- **Dirty read:** Two transactions, A and B, are being executed concurrently, and A has written to key `K1` and `K2`. If B now reads key `K1` and sees the write that A performed, but then reads key `K2` and does not see the write that A performed, the anomaly is considered a dirty read. Since B only sees a part of the operations that A has per-

formed, and thus can neither be considered to be executed before nor after A. The result is different from a sequential execution, as either A or B would need to be performed first in a sequential execution.

- **Non-repeatable read:** If a transaction reads the same record twice, but the value differs from the first read to the second, then the transaction has observed the result of another transaction while running. This is different from a sequential execution, as there should not be anything else than the running transaction that changes data.
- **Phantom read:** Phantom reads are similar to non-repeatable reads, where reads return different values. However, for phantom reads, every key returns the same value, but a scan returns a different *set* of values. This can happen if another transaction inserts a row that matches a scan performed by the transaction.
- **Write skew:** Write skew occurs when two transactions read the same keys, but then write different keys based on this data. For example, if two transactions both read the keys `K1=5` and `K2=4`, and the first transaction sets key `K1` equal to key `K2`, i.e. 4, while the second transaction sets key `K2` equal to key `K1`, i.e. 5. The intention of both transactions is for both keys to have an equal value, but a write skew means that these two values are now swapped, while any sequential execution would mean that the two values were equal.

These are six of the most commonly discussed anomalies when talking about isolation levels. The SQL standard, however, only mentions the dirty read, non-repeatable read and phantom read anomalies when defining isolation levels. This means that the SQL isolation level of a database does not define whether or not the database prevents the three other anomalies described above [3]. The isolation levels defined by the SQL standard are `READ UNCOMMITTED`, `READ COMMITTED`, `REPEATABLE READ` and `SERIALIZABLE`. Another common isolation level that is not defined in the SQL standard is Snapshot Isolation, which provides a snapshot of the database that the transaction can read from, and only allows the write skew anomaly. In this thesis, the most relevant isolation levels are Snapshot Isolation and serializability. There are different levels of serializability as well, allowing for different types of smaller anomalies, but those are out of the scope of this thesis.

2.4 Distributed concurrency control

Concurrency control is required to allow multiple transactions to run at the same time while keeping the system state consistent [16]. Concurrency control is a well researched area and is needed in any data store that allows concurrent transactions to occur. However, in a distributed system, the choice of concurrency control becomes even more important, as each thread no longer runs on the same machine, but rather on a network with significant network latencies. Each strategy has its benefits, and depending on how the workload looks, different strategies are best. The most important factor for which concurrency control method should be used is how often there are conflicts between transactions, as some methods are good at handling conflicts, but at the cost of every transaction's performance [10]. Below, we have described four common concurrency control algorithms [10].

Two-Phase Locking

Two-Phase Locking (2PL) was the first method that was proven to be serializable. Two-Phase Locking introduces two stages of locking in a transaction's lifecycle: lock acquisition and lock releasing. A transaction can request new locks as they are needed during its execution, and when locks are no longer needed they can be released. One caveat, however, is that after any lock is released, no more locks are allowed to be acquired, which is why the algorithm is said to have two phases.

There are two types of locks in 2PL, one for reads and one for writes. Read locks can be shared with other read locks, while write locks can not be shared with other read or write locks, i.e. a write lock needs exclusive access to a data item. If a write lock is requested for a data item that has other locks, the transaction needs to wait. This waiting can cause deadlocks, since transactions may have a cyclic waiting dependency. This needs to be handled by killing one of the transactions. Another approach to the deadlock problem is to kill any transaction if it needs to wait for a lock, and instead retry the whole transaction later. This might be more costly if complete retries are expensive, but it solves the deadlock problem without a complex algorithm.

Optimistic

Optimistic concurrency control systems perform transactions without any locks. However, right before a transaction commits, it performs a check to see if any of the data it has read has been modified by another transaction. If this is the case, the transaction needs to be retried. Optimistic concurrency control can perform well if most operations do not conflict, but if many conflicts happen, the performance suffers greatly as retries become common.

Multi-version concurrency control

Multi-version concurrency control (MVCC) stores multiple versions of each key. A transaction gets assigned a version or timestamp when it starts, and only read keys that have a lower or equal version number to this. This makes it relatively easy to implement snapshot isolation, but there are still some challenges. For one, old keys need to be cleaned periodically so that the database does not fill up with stale data, and second, assigning monotonically increasing versions or timestamps becomes complicated if the system is distributed.

Deterministic

In a deterministic concurrency control system, transactions need to be sent in full to the database before execution, i.e. no interactive transactions are allowed. All transactions are then analyzed and assigned to batches by a scheduler. This scheduler guarantees that there will be no conflicts between transactions within a batch, and the actual execution of transactions can therefore be done with little overhead. One disadvantage of this method is that the scheduler can become a bottleneck in the system, and that a latency is introduced, as each transaction needs to be batched before it is executed.

2.5 Distributed transactions and consensus

In a distributed system, distributed consensus must be implemented to maintain consistency. This means that nodes must agree on an ordering of transactions. However, all transactions must not necessarily be ordered, but the transactions that are in conflict must have the same order on all nodes [16].

Distributed consensus can be implemented in many different ways, but two common algorithms are Two-Phase Commit and Raft. These two algorithms are described further in Section 2.6 and Section 2.7.

2.6 Two-Phase Commit

Two-Phase commit (2PC), not to be confused with Two-Phase locking (2PL), is used in replicated databases to ensure that the different replicas agree on whether a transaction should be committed or aborted [16]. This synchronization is important, as otherwise the states on replicas will diverge and become inconsistent.

The Two-Phase commit protocol is divided into the prepare phase and the commit phase. In the prepare phase, a coordinator will gather votes from all replicas on whether they want to commit or abort. After all votes are collected the commit phase starts, and the coordinator inspects the votes. If any replica voted to abort, the transaction will be aborted, and otherwise, the transaction is committed. The final step is that the coordinator broadcasts its decision to all replicas so that they know whether the transaction was aborted or committed. This protocol ensures that all replicas agree about the state of all transactions and ensures that no inconsistencies arise.

2.7 Raft

Raft [15] is a distributed consensus algorithm that was developed to be a simpler and more understandable alternative to Paxos. The algorithm allows an odd set of nodes to agree on the content and order of a log, called the Raft log. This log can contain anything, but in the case of a database it will usually contain records similar to a traditional write-ahead log used by non-distributed databases.

Raft works by periodically electing a leader for a Raft group. Each group should also have an odd number of nodes, so that a majority vote is always guaranteed. Leaders are elected for a configurable interval, and are usually reelected after each interval unless they become unavailable. If a leader becomes unavailable, other nodes will announce an election and try to take the leader role after a random delay. For a new node to become the leader, it needs to receive a majority vote from the group.

New log entries can only be added by the Raft group leader. The leader then needs to initiate a vote and get a majority of votes before the write is considered as committed. Since only a majority of votes is required, some nodes may fall behind the leader on logs. In case a leader fails and a new leader must be elected, nodes need to announce what their latest log entry number is. A node will never vote for another node that has an older log than itself, but should always vote for a node that has a newer log. This mechanism combined with that the majority of nodes have the newest log ensures that a leader can never be elected with an out of date log.

When implementing an SQL database on top of Raft, both reads and writes should add a log entry to Raft. If no log entry is added, reads can become inconsistent if reading from followers or reading from a leader during a leadership change. However, as this adds a significant overhead to reads, many databases use a mechanism called leadership lease to avoid adding logs for reads [21, 19, 6]. A leadership lease is held by the group leader for a given period of time, and within this time period, no other node can be elected as the leader. This guarantees that a read served by the leader will always have the latest data, as long as the leader still has its lease. Followers can still not serve reads with this mechanism, however, as they may still have stale data. Leadership leases solve a bottleneck for reading data, but can also introduce some period of unavailability if a leader crashes and its lease is too long. For this reason, lease durations are usually quite short.

2.8 NewSQL

NewSQL databases are a relatively new class of databases that combine horizontal scalability with the guarantees that traditional relational databases provide. Our description of NewSQL databases is based on a survey by Andrew Pavlo and Matthew Aslett of NewSQL systems [17]. NewSQL databases were inspired by the growth in the popularity of NoSQL databases because of their scalability. A problem with NoSQL databases, however, is that they usually provide much weaker consistency and isolation, and often have no transaction support. This makes it much more difficult for application developers to reason about database states, and it also means that additional logic needs to be created in applications to handle inconsistencies. There are also many domains for which weaker guarantees are not feasible to use, e.g. financial applications.

There are two main differences between a NewSQL system and a traditional SQL database. First, a NewSQL system scales horizontally, meaning that it can handle increased load by adding nodes to a cluster. Non-distributed systems can only scale vertically, meaning increasing hardware resources of a node, and this means that there are hard limits on how much a system can scale. Second, NewSQL systems usually support replication, meaning that all state is stored on more than one node. This means that the crash of any single node does not imply downtime for the system, which makes NewSQL systems more resilient to failure than traditional systems.

NewSQL systems often have some commonalities in how they are architected. For one, databases are usually sharded, so that different nodes can be responsible for different sets of data. Sharding is what enables horizontal scalability, as without sharding, every node would need to store all data, and one would quickly be limited by the least powerful node. The sharding strategy is also very important for performance, as operations that touch multiple shards are much more expensive than single-shard operations. Another similarity of NewSQL databases is mentioned in the previous paragraph: most systems replicate each shard to multiple nodes for failure tolerance and sometimes for increased read performance. Replication strategies are usually either synchronous or asynchronous, where the difference lies in whether or not replicas need to acknowledge writes before the leader commits. Synchronous replication is required for strong consistency guarantees, while asynchronous replication is more performant. This is an instance of the PACELC theorem described in Section 2.1, as either latency or consistency must be chosen by a system during normal network conditions.

2.9 Percolator

In 2010, Google published a distributed transaction system called Percolator [18]. It supports concurrent transactions and implements multi-version concurrency control. Each transaction sees a consistent state of the database with MVCC, and it provides snapshot isolation between transactions. Percolator is not a NewSQL system, as it only supports simple key-value operations, but its design has been used in TiDB to implement a distributed SQL database. TiDB is discussed further in Section 2.13.

Percolator uses a timestamp "oracle" in order to get a unique and monotonically increasing version number for each transaction. The oracle is essentially is a centralized system that all transactions must go through in order to get a version number for their reads and writes. This means that there is no dependence on clocks being synchronized, but it can also be a performance bottleneck if the cluster outgrows the central timestamp oracle.

2.10 Spanner

Spanner [7] is a NewSQL system that was published by Google in 2013. Spanner enables concurrent distributed transactions by using MVCC, and it uses timestamps to assign unique version numbers to transactions. However, to overcome the issue of unsynchronized clocks, Spanner introduces the "TrueTime API", which is an accurate clock that can return the current time along with an uncertainty. Spanner uses this timestamp as the transaction's version number, but also adds additional logic to handle the uncertainty of the timestamp. For example, if reading a key that may have been written either before or after the transaction, the transaction needs to be retried with a new timestamp. With this system, Spanner is able to provide a high level of consistency called external consistency, which is stricter than serializability, but not the strictest form.

The TrueTime API is based on atomic clocks and GPS clocks to get accurate results, and it can usually keep the uncertainty below 10ms. A lower uncertainty is always beneficial for performance, as the probability of a transaction needing to retry is lower.

Spanner uses a key-value interface for the underlying data store, and handles replication with the Paxos algorithm. As mentioned, concurrency control is handled with MVCC, and transactions use a two-phase commit protocol to ensure consensus.

2.11 Calvin

Calvin [22] is a database that utilizes a deterministic form of concurrency control. This means that all transactions are placed into batches where they won't have conflicts, which can speed up transaction processing significantly. Calvin does this by having one centralized scheduler that accepts all transactions and batches them.

Even though this approach can lead to a significant speed increase, it also has some disadvantages. For one, the scheduler may become a bottleneck at a certain scale. Second, so-called dependent transactions are challenging to execute. Since the scheduler needs to know all keys that are accessed by the transaction, operations that e.g. fetch results based on a query like a foreign key lookup, are challenging to execute. Calvin suggests that in this case, a read-only query should first be executed to figure out the actual keys, and the keys should then be used in the actual transaction. In order to avoid inconsistencies, the real transaction must then check that the values that were prefetched did not change in the meantime. In case the values have changed, the transaction needs to be retried. However, the authors argue that this is not a significant issue in most systems, as data like foreign keys rarely changes in most applications.

2.12 CockroachDB

CockroachDB is a NewSQL database developed by Cockroach Labs, and was first released in 2017. CockroachDB's design was inspired by Google Spanner, but one significant difference compared to Spanner is that it does not use any specialized hardware clocks. Instead, Hybrid Logical Clocks are used, which can emulate the features of the TrueTime API. However, this means that the uncertainty in CockroachDB is much larger than in Spanner, and in case of clock anomalies, there is a chance of introducing inconsistencies into the database. To mitigate damage, CockroachDB has mechanisms in place to stop the system in case of unsynchronized clocks. CockroachDB supports serializable transaction isolation, but not the strongest form of it.

Cockroach Labs also sells Cockroach Cloud, where they sell a managed offering of CockroachDB clusters. CockroachDB is licensed so that it can be used for any purpose other than selling “CockroachDB as a service”. All source code is also released freely three years after release, meaning that three year old code can be hosted and sold as a service [14]. In addition to this, there is an enterprise version of CockroachDB that adds some extra features.

Our description of CockroachDB is based on their official online documentation [6]. In the sections below, we will describe the architecture and storage engine of CockroachDB, as well as how it handles transactions and geo-replication.

2.12.1 Architecture

CockroachDB is a homogenous cluster of nodes, meaning that all nodes run the same software and there is no special master role in the cluster. Thus, all nodes receive requests from a client and handle transactions without requiring the client to have knowledge of data placement.

The CockroachDB software is split into five layers that build on top of each other. All layers run in the same program, but the layers are useful for reasoning about the system. The five layers are listed below:

1. **SQL layer:** The SQL layer is the layer that clients communicate with. It receives SQL queries and translates these into key-value operations that are sent to the next layer. Thus, no other layers are aware of SQL, and instead operate with key-value data.
2. **Transactional layer:** This layer coordinates transactions and provides ACID support to the database. Concurrency control is handled at this layer.
3. **Distribution layer:** The distribution layer abstracts the distributed nature of the key-value space to higher layers. Higher layers do not need to be aware of the placement of data. Instead, requests are routed to the correct peers in this layer.
4. **Replication layer:** The replication layer replicates data for each shard, making sure that each shard is consistent and that peers are in sync. This is accomplished by using the Raft algorithm.
5. **Storage layer:** The storage layer is the layer that reads and writes from disk. As will be discussed later, each node may be a member of

many Raft groups, i.e. shards, but all data on one node is handled by a single storage layer.

2.12.2 Storage engine

In order to better compare the different NewSQL databases, we have defined the storage engine as everything below the SQL layer. Thus, for CockroachDB, what we call the storage engine consists of the transactional, distribution, replication and storage layers.

CockroachDB shards data into “ranges” that are each about 64MB large. Since shards are quite small, any node will usually host many different shards. All shards are replicated to a configurable number of nodes, but the default is three. Replication is handled synchronously to enable consistency, and is performed with the Raft algorithm. CockroachDB also implements leader leases with Raft, as described in Section 2.7, for increased read performance on each shard.

CockroachDB’s storage layer is built on top of RocksDB, a key-value storage system that persistently stores data on disks and has high performance. RocksDB supports standard key-value operations, and is also highly configurable for special needs. CockroachDB uses RocksDB as an MVCC store, where each key can have multiple timestamped versions. In order to get the latest version of a key, a scan in reverse order can be performed with a prefix, where the first result is the latest version. RocksDB is highly dependent on bloom filters to figure out quickly in which blocks it should search for a certain key, but this does not work out of the box with prefix searches as bloom filters are built with full keys. However, RocksDB also supports prefix bloom filters, where a user-defined prefix of keys can be used when generating bloom filters. This enables the use of MVCC with RocksDB without any significant performance loss.

Another special feature in RocksDB that is utilized by CockroachDB is snapshots. RocksDB can generate consistent snapshots of the database state without blocking other operations. When a new replica joins a Raft group, a RocksDB snapshot can be generated and transmitted to the new node so that it can catch up with the state quickly. Additionally, some optimizations for data ingestion are used so that multiple compactions do not need to be performed on the new node.

2.12.3 Transaction handling

As mentioned earlier, CockroachDB uses timestamps to order transactions, based on Hybrid Logical Clocks (HLCs). Any communications between nodes also include their current HLC timestamp, in order to quickly identify any anomalies in clock values. If clock offsets are larger than the maximum offset, 500ms by default, inconsistencies may occur, so CockroachDB instantly shuts down the node if this is detected.

Transactions in CockroachDB are tracked by having a record for each transaction stored in the database. This record is used as a source of truth for the transaction's state, and can be queried by other transactions to see whether or not the data it wrote is committed. The node that initiates a transaction is called the transaction coordinator, and it must periodically update the transaction record while running in order to signal that it has not crashed. If a transaction is marked as in progress but has not been updated after some time threshold, it is considered by all other transactions as aborted and can be deleted.

In order to have a distinction between committed values and uncommitted values, CockroachDB uses something called write intents, which are essentially temporary writes. Any data written by a transaction is initially created as a write intent. A write intent looks similar to normal data values, but it also contains a pointer to the transaction's transaction record. Once a transaction commits, the transaction record is marked as committed, and all write intents are turned into normal MVCC values asynchronously. However, since this happens asynchronously, other transactions may encounter a write intent that belongs to a committed transaction, and the value should therefore be considered as a normal value. Any reads that encounter a write intent must look up the status of the transaction through the transaction record pointer in the intent. If a transaction is aborted, the intent can be deleted, and if the transaction is committed, the intent can be converted into a normal value. However, if the transaction is still in progress, the transaction that tried to read the value must wait and is added to a wait queue. This system of write intents and transaction records enables atomic commits, as the transaction record is always the source of truth for a transaction's status. Changing the record changes the status of the whole transaction, and is a single-shard operation since it is a single data item, and can thus be performed atomically.

The wait queue that transactions are added to is stored in a single CockroachDB shard. This makes it easier to detect deadlocks, as all of this data is stored locally on the same hardware. Each transaction in the wait queue registers with the ID of the transaction it is waiting for, and whenever a transaction is completed, the queue is notified and can resume any transactions that were blocked by the now completed transaction. Deadlocks in the waiting queue are handled by randomly killing one of the transactions that are in the deadlock loop.

Before creating a write intent, a transaction must check the existing versions of the key. If a write intent for the key already exists, the transaction must check whether it is still in progress, and if so, the transaction must wait. Next, the transaction must check if the existing MVCC value has a higher timestamp version than the current transaction's timestamp, and if so, the transaction must be retried with a new timestamp. The last thing that must be checked to ensure isolation and consistency is that the most recent read must have occurred with a lower timestamp than the current transaction. Every time a key is read, the timestamp of the reading transaction is stored in order to guarantee that the result of the read at that timestamp can not change later on. In case a write transaction wants to write to a key that has a more recent read, CockroachDB tries to automatically push the timestamp of the writing transaction to a higher value. However, as the transaction's timestamp has changed, the transaction must check that none of its previous reads have become stale as a result of the new timestamp, and this process is called a read refresh.

As CockroachDB's transaction ordering depends on synchronized clocks, and because timestamps come with some uncertainty, each timestamp has some overlap with other timestamps where it is impossible to determine which one came first. This results in some issues when dealing with MVCC, as an absolute ordering is required for consistency. To work around this, CockroachDB uses a predefined maximum clock uncertainty, and tries to push timestamps beyond this uncertainty if there is a conflict, or retry the transaction altogether. For example, if a read is performed and the MVCC timestamp is very close to the transaction's timestamp, the transaction's timestamp needs to be pushed to a later point where it is no longer uncertain what happened first. When pushing timestamps like this, as mentioned earlier, a read refresh needs to occur to ensure that no previous reads are now stale. In case some keys are both written and read very frequently, this feature can slow down operations, but it also prevents anomalies from occurring.

2.12.4 Geo-replication

CockroachDB supports geo-replicating workloads, and allows configuration of data placement so that data availability and latency can be optimized for the requirements of the database users. However, the geo-replication features are only available under the enterprise license, meaning that they cost money and are not part of the open source offering.

The main way that users can control geo-replication in CockroachDB is to set a partitioning key on each table, which will determine where a table row is placed. Additionally, partition keys can be assigned to regions so that data is faster to access in a certain region. There are many different strategies for data placement, and one of these is to place all replicas for a set of partition keys in one region, which allows for fast reads and writes in that region. However, this can come at the cost of failure tolerance, as downtime in this region will make the affected rows unavailable in all regions. Another strategy is to place replicas in adjacent regions close to where they are most needed in order to minimize latency while still being failure tolerant.

CockroachDB also allows requesting the placement of the Raft group leader in a certain region. Since CockroachDB implements leaseholder reads, this means that reads can be served with very low latency in this region, while writes still require cross-region communication. This can be a good trade-off for certain applications if the read speeds are more important than write speeds. Another approach that is possible with CockroachDB is to enable follower reads, which enables low latency reads in all regions that have replicas. This, however, does not guarantee that reads return the most recent data and may result in inconsistencies if doing writes based on these reads. If read-only transactions can tolerate old data, though, and they require very low latency, it may be a suitable feature for the application.

2.13 TiDB

TiDB is a NewSQL database that is built on top of the key-value database TiKV, and was first released in 2017. Both TiDB and TiKV are developed by PingCAP under the Apache 2 open source license. For the purposes of this section, we count TiKV as a part of TiDB, even though it can run independently of TiDB. The design of TiKV was largely inspired by Percolator, described in Section 2.9. This means that TiDB uses a centrally assigned version number for ordering transactions rather than timestamps with uncertainty. The Percolator design also means that TiDB only supports Snapshot Isolation, which is weaker than serializability that CockroachDB and YugabyteDB offer.

Below, we describe TiDB and the relevant parts of TiKV in depth to give insight into how the database works. Our descriptions are based on TiDB's and TiKV's online documentation [23, 24]. First, we describe the architecture of a cluster and the software, followed by how the storage engine TiKV works and handles transactions. Finally, we include a short section about how TiDB works in a geo-replicated scenario.

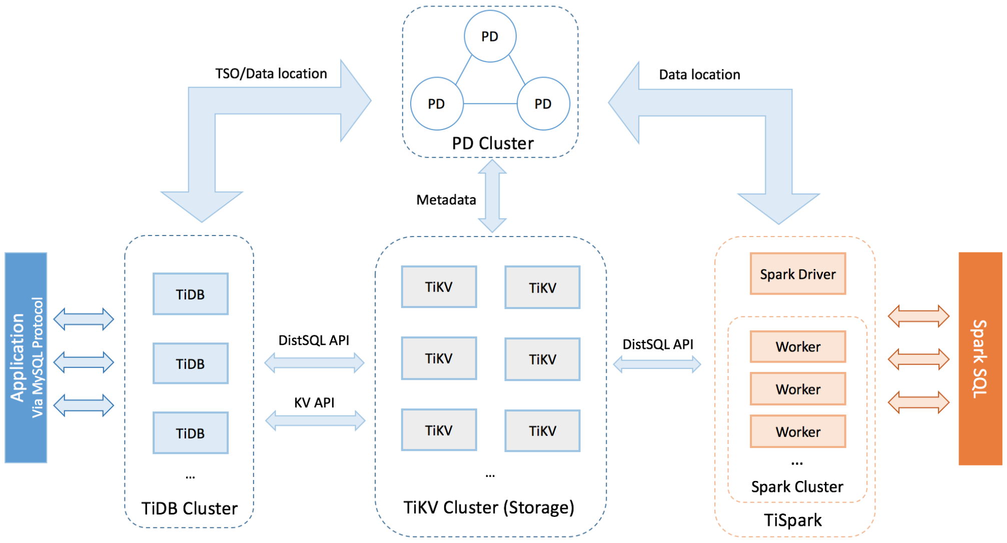

Figure 2.1: TiDB's architecture

https://pingcap.com/docs/stable/architecture/

2.13.1 Architecture

A TiDB cluster consists of three main parts: the timestamp oracle, called the Placement Driver (PD), the key-value store, called TiKV, and the SQL processing system, called TiDB. These three parts interact to form a NewSQL system that can handle SQL transactions. Each of the parts supports replication and is usually run on separate machines. A fourth part that may also be included in a cluster for Online Analytical Processing (OLAP) queries is TiSpark. The four components are illustrated in figure 2.1. The TiSpark component is not discussed in this chapter, as it is not related to NewSQL. As can be seen in the figure, the three components of the NewSQL system, PD, TiKV and TiDB must all communicate together to execute SQL queries for clients.

All SQL clients communicate with TiDB, either directly or through a load balancer. TiDB is responsible for translating user queries into key-value operations that can be handled by TiKV. TiDB is a stateless service, which means it can be scaled up and down quickly depending on demand. All data in the system is stored in TiKV and the Placement Driver.

The Placement Driver acts as a kind of master for the cluster, and is a single shard replicated system. PD is replicated with Raft, and only the leader performs actions on behalf of the PD cluster. The PD keeps track of the different shards in TiKV and knows which TiKV nodes are responsible for which data. In addition to this, the Placement Driver acts as a timestamp oracle for the cluster and assigns timestamps to all transactions for ordering purposes. As this is done by a single system, there is no requirement for all clocks to be synchronized. The Placement Driver is replicated on a number of nodes, but they act as one system, which means that adding nodes does not increase PD's performance, and extra nodes only help by adding more failure tolerance. In order to ensure that no timestamp is assigned twice if a leader crashes, the leader must reserve a block of timestamps by committing to the Raft log before assigning timestamps, which ensures that the next leader can not assign any timestamps in the same block.

TiKV is the key-value interface of TiDB, and it also has full transaction support. The design of TiKV is further described in the storage engine section below. TiKV mainly responds to requests from TiDB, but it can also communicate with clients that wish to use a key-value interface instead of an SQL interface. TiKV nodes also communicate with the PD cluster, for example in order to know what shards they should be serving. Since shards are spread out evenly between TiKV nodes, adding TiKV nodes to the cluster will increase performance for most workloads, as there are more nodes that can share the work.

2.13.2 Storage engine

As described in the previous section, TiKV is the storage engine of TiDB. TiKV was inspired by Google Percolator, and has support for transactions with snapshot isolation.

TiKV uses Raft for synchronously replicating data, and by default replicates data to three nodes. The key-value space is range-partitioned into "regions", where each region is usually small, which allows for fast movement of regions between nodes. Since regions are small, each node in the cluster usually serves many of them. An illustration of how Raft groups, regions and nodes interact can be seen in Figure 2.2. The figure shows a cluster where data is sharded into three regions that are distributed among four nodes. No node stores more than one replica of each group, as this would reduce redundancy. When using TiDB as a NewSQL system, the clients on the top of the figure would be TiDB nodes that have translated client SQL requests into key-value requests.

For storing key-value data to disk, TiKV uses RocksDB. This choice was made based on the high performance and maturity of the technology [24]. Similar to CockroachDB, TiKV also uses RocksDB as an MVCC store and therefore requires prefix bloom filter support. Another RocksDB feature that TiKV utilizes is the multi column family support, which essentially means that there are multiple databases stored on the RocksDB instance, and there is support for atomic writes across these. How this is used is

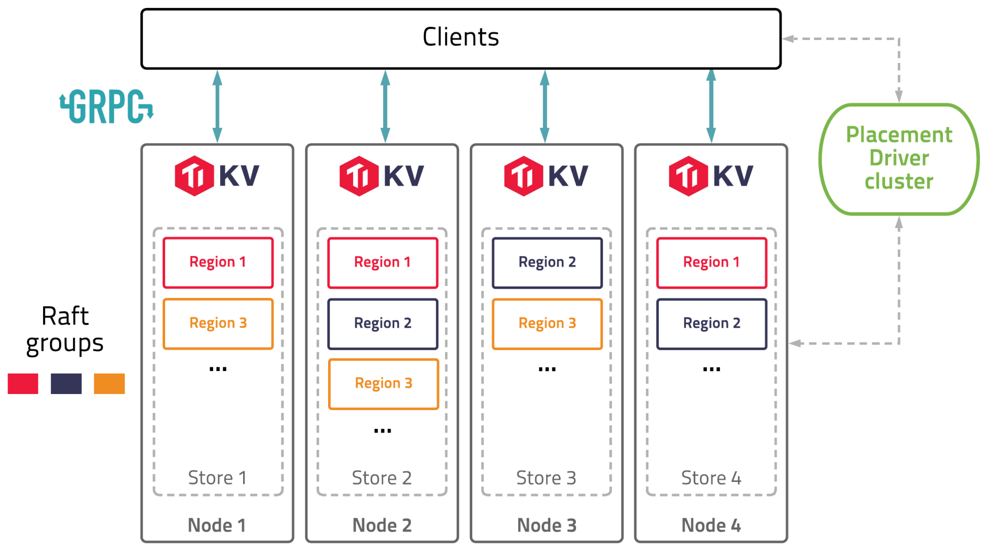

Figure 2.2: TiKV sharding and replication.

https://tikv.org/docs/3.0/concepts/overview/

described further in the next section.

2.13.3 Transaction handling

All transaction handling in TiDB happens inside TiKV, as TiDB is a stateless layer in the database. TiKV implements Percolator's concurrency control mechanism and therefore supports snapshot isolation between transactions. However, if one needs to avoid the write skew anomaly, a special SQL syntax can be used: `SELECT .. FOR UPDATE`. As in Percolator, transaction ordering in TiKV depends on the timestamps that are assigned by a central timestamp oracle, which in TiKV's case is the Placement Driver.

Each key in TiKV has three separate columns that contain the data, lock and write values. To support these three separate values, TiKV uses one RocksDB column family for each value type. The same key may have a different value in each column family, enabling the multi-value functionality required by the Percolator model. The data column of each key contains multiple versions of the data, but does not contain any information about whether or not this data is committed. The lock column stores a single lock for the row, and finally the write column contains commits for the key, where each commit is stored with a timestamp and points to a value in the data column. The timestamps of MVCC keys are encoded in such a way that the latest version will always be the first version to appear in a scan operation.

When a transaction in TiKV starts, it requests a timestamp from the timestamp oracle. For a write transaction, the next step is to acquire locks on all of the data items it wants to write to. If any key has a value with a higher timestamp than the current transaction, or a lock already exists, the transaction needs to release its locks and retry with a later timestamp. The first lock to be created by a transaction is assigned as the primary lock, and all other locks contain a pointer to the primary lock. After each lock is created, a data value is also added to the data column of the key. After all of the writes have been performed in this way, the transaction can commit by updating the primary lock and at the same time create an entry in the corresponding write column. The primary lock is the source of truth for the transaction status, and can be used by other transactions if they encounter a secondary lock. After the primary lock has been deleted, all of the secondary locks can be converted into write values, but this is done asynchronously.

When a transaction wants to read values, it uses the timestamp that it received from the timestamp oracle. It must then first check if there is a lock that has a timestamp lower than the current transaction. If there is no lock, it can fetch the newest version from the write column that has a lower timestamp than the current transaction, and then read the value that the write points to. However, if there is a lock on the key, the read transaction must check the state of the writing transaction. This is done by looking up the primary lock, and if it is still active the read transaction needs to wait or be retried. If the primary lock does not exist, it must be determined if the transaction was aborted or committed. This can be done by checking if there is a value in the write column that corresponds to the primary lock. If there is a write, the transaction is considered committed, and if not, it is considered aborted. This mechanism for handling secondary locks ensures that commits are atomic, as the action of changing the primary lock guarantees that the whole transaction is either committed or aborted.

2.13.4 Geo-replication

TiDB has some support for geo-replication, but a large issue with its design is that TiDB is highly dependent on the centralized Placement Driver. Only one replica of the PD can hand out timestamps to transactions, and any transaction that is started from a region different from where the PD leader is will get a high latency because of this. However, having PD replicated to multiple regions will increase the failure tolerance of the system, as replicas in other regions can take over in case of region outages.

2.14 YugabyteDB

YugabyteDB, developed by Yugabyte, is an open source NewSQL database that is inspired by Google Spanner. YugabyteDB is licensed under the Apache 2 open source license, and the company behind it also sells a managed cloud version of the database. Like CockroachDB, YugabyteDB has no specialized hardware clocks for the TrueTime protocol, but instead they use Hybrid Logical Clocks. The database supports both snapshot isolation and serializability and allows the user to choose which level of transaction isolation they require. YugabyteDB is quite new, with its SQL interface becoming production-ready in late 2019.

The sections below describe YugabyteDB in depth in the same format as the descriptions of CockroachDB and TiDB: architecture, storage engine, transaction handling and geo-replication. Our descriptions are based on the YugabyteDB online documentation [26].

2.14.1 Architecture

A YugabyteDB cluster consists of nodes with two separate roles, the master nodes and the storage instances, called tablet servers (TServer). The master nodes are replicated with Raft, and act as a single node, i.e. no sharding. The TServers are also replicated with Raft, but also sharded in order to support horizontal scalability.

The master node in YugabyteDB is responsible for the placement of shards and stores metadata for the whole system. However, unlike in TiDB, the master nodes are not involved in every transaction, as they only control the placement and movement of data. YugabyteDB instead relies on Hybrid Logical Clocks to order transactions.

TServers in YugabyteDB are sharded by hash of the primary key, and they are also replicated to a configurable number of nodes, usually three. TServers receive SQL requests from SQL clients and then translate these to key-value operations that are handled by the document layer, called DocDB. Any TServer can process queries, and clients do not need to be aware of data placement. In addition to SQL clients, YugabyteDB also supports the Redis and Cassandra query interfaces, and they all use the same underlying key-value storage to fulfill queries.

2.14.2 Storage engine

YugabyteDB's storage engine is called DocDB, and transactions are handled in this layer. DocDB is a key-value database and is the underlying storage for both the SQL, Redis and Cassandra interfaces of YugabyteDB,

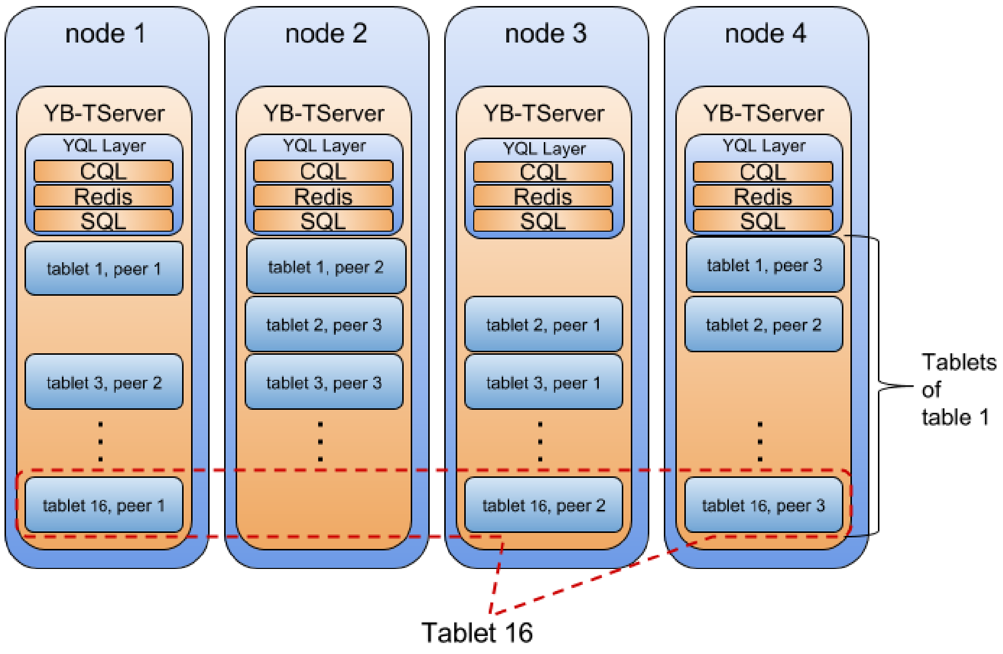

Figure 2.3: Sharding and replication in Yugabyte Tablet Server

https://docs.yugabyte.com/latest/architecture/concepts/yb-tserver/

but we will only focus on the features that enable SQL support in this section.

DocDB is a sharded system, and each shard is called a tablet, which is why storage nodes in YugabyteDB are called Tablet Servers (TServers). Each tablet is small in size and is replicated synchronously with Raft to ensure consistency, and is usually replicated to three nodes. DocDB is designed so that different SQL tables are never assigned to the same tablet, but a table may consist of multiple tablets. An illustration of the sharding of DocDB is shown in Figure 2.3, where there is one table consisting of 16 tablets, spread out among four nodes.

For Raft reads, DocDB implements leader leases as described in Section 2.7. This greatly improves the read performance of DocDB, while still ensuring consistency. DocDB also supports follower reads, but this does not guarantee that clients receive the most recent data.

The underlying storage system for DocDB is RocksDB, which is used as an MVCC store. DocDB has chosen to use one instance of RocksDB for each tablet, meaning that each TServer will run many independent instances of RocksDB. The reason for doing this is that copying a tablet to another node is very simple, as the raw SSTable files from disk can simply be copied to another node. Additionally, table deletions mean that the RocksDB instance can simply be deleted, instead of needing to create tombstone records and waiting for compactions to free up space. DocDB uses RocksDB as an MVCC store with a timestamp version for each key, and therefore also utilizes RocksDB's support for prefix bloom filters.

2.14.3 Transaction handling

YugabyteDB, like TiDB and CockroachDB uses MVCC for their concurrency control. Each key has a timestamp attached so that there can be many versions of the same key. In order to ensure proper isolation, transactions do not write normal records, but rather what is called a provisional record. Provisional records are marked by having a special prefix on the key, but they are always stored in the same tablet that the actual record would be stored on, in order to enable an atomic replacement of the provisional record.

Transactions in YugabyteDB, or more specifically in DocDB, are tracked in a transaction table. Any provisional record that is created by a transaction points to the corresponding record in the transaction table, which ensures that the status of a provisional record can always be looked up by other transactions. Any change to the transaction record acts as an atomic action on the whole transaction, and this enables atomicity in YugabyteDB. Whenever a transaction commits, it will update the transaction record first, and then convert all provisional records into normal records asynchronously. However, clients receive a result once the transaction commits, and do not need to wait for the asynchronous cleanup. The TServer that initiates a transaction becomes the manager for that transaction and is responsible for coordinating the execution of the transaction and returning results to the client.

When a transaction wants to write data, it first needs to acquire a lock for the relevant data. However, locks are not stored explicitly but rather, the provisional records are considered locks [20]. Additionally, the Raft leader will keep all of these locks in memory for fast access. If a transaction wants to write data but a provisional record already exists, one of the transactions must be aborted based on priority. In order to abort the transaction that has the lock, the provisional record of that transaction can be removed. However, this also means that all transactions need to check that all their provisional records still exist before committing, to ensure that the transaction wasn't aborted due to a conflict.

When a transaction reads data, it uses its own timestamp to choose the correct MVCC version to read. The highest version that is lower than the transaction's timestamp is always chosen. However, because HLCs have some uncertainty, it can not always be determined if the key was written before or after the current transaction. In this case, the whole transaction is aborted and retried with a later timestamp in order to ensure consistent reads. If a provisional record is encountered by a read, the transaction status needs to be looked up from the transaction table. If the transaction is committed, the value is considered as a normal value, while if the transaction is aborted the value is ignored. If, however, the transaction is still in progress, the transaction needs to abort and be retried with a later timestamp.

Figure 2.4 shows the write path of a transaction that involves writes to multiple tablets. First, a client sends a request to a TServer (1). In the figure, this is a key-value request, but it could also be a SQL request when using YugabyteDB as a NewSQL system. This TServer then becomes the transaction manager and creates a record in the transaction table (2). Next, the transaction manager creates provisional records by contacting the Raft leader of each tablet that is affected (3). Finally, a commit is performed by updating the transaction record (4) and the client receives a response (5). After the transaction is committed, the provisional records are asynchronously turned into normal records (6). This figure also shows that each tablet has two followers, which is a typical deployment, i.e. three replicas for all data.

2.14.4 Geo-replication

YugabyteDB currently has no explicit support for geo-replication, but they have some partitioning features on their roadmap[1]. A YugabyteDB cluster can currently be run in different regions, but operations may have high latency, and the user has no control over data placement.

YugabyteDB also has support for follower reads, which may be helpful for some applications that require high availability and can sacrifice some guarantees by allowing reads to get stale data.

[1]https://github.com/yugabyte/yugabyte-db/issues/1958

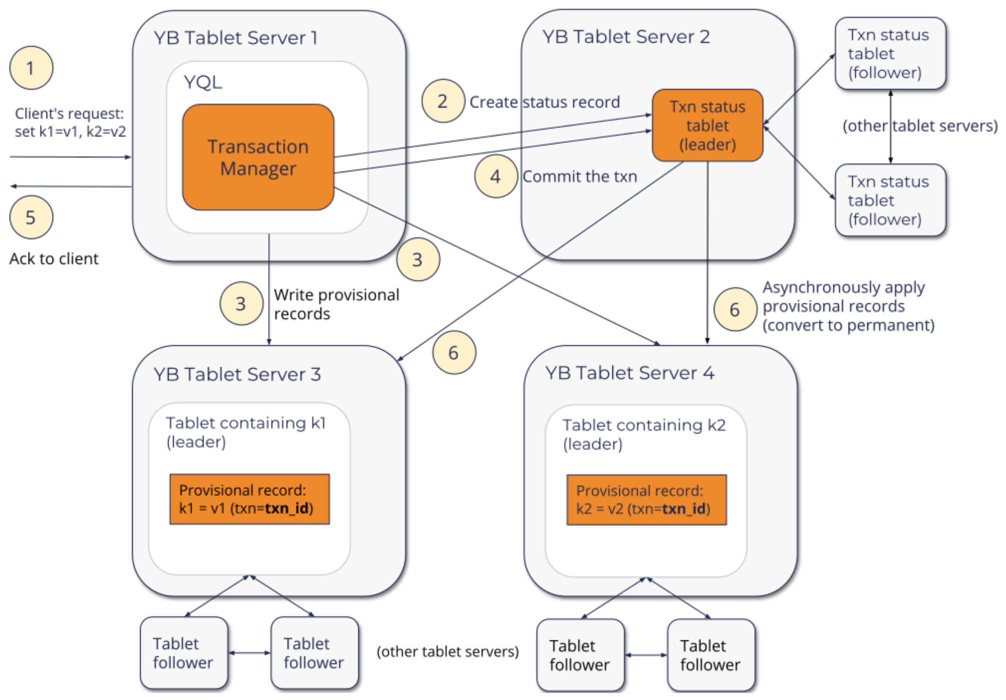
YB Tablet Server 1
YQL
Transaction Manager
1
Client's request: set k1=v1, k2=v2
5
Ack to client
YB Tablet Server 2
2
Create status record
Txn status tablet (leader)
4
Commit the txn
Txn status tablet (follower)
(other tablet servers)
Txn status tablet (follower)
3
3
Write provisional records
6
6
Asynchronously apply provisional records (convert to permanent)
YB Tablet Server 3
Tablet containing k1 (leader)
Provisional record: k1 = v1 (txn=txn_id)
YB Tablet Server 4
Tablet containing k2 (leader)
Provisional record: k2 = v2 (txn=txn_id)
Tablet follower
Tablet follower
(other tablet servers)
Tablet follower
Tablet follower

Figure 2.4: Write Path in a YugabyteDB transaction

https://docs.yugabyte.com/latest/architecture/transactions/transactional-io-path/

Chapter 3

Implementation

The goals of this thesis, as described in Section 1.2, require us to test multiple databases in many different configurations. We need to vary both cluster and instance sizes, and each configuration should run several different workloads. Because of the variance in cloud instance performance, described in Section 3.3.1, we also need to repeat each configuration multiple times to get an average value. These requirements made us realize early on that we needed to automate as much of the testing as possible. The result of our work is an extendable system that enables automatic execution of benchmarks by defining which databases and configurations one wants to test. This system also makes it easy to reproduce benchmarks that others have run and published.

3.1 Benchmarking system

Our system consists of three main parts: the benchmarking software running on client machines, the database clusters that are being tested, and a control script that creates clusters and coordinates execution of benchmarks. The three parts are described in detail below, and finally we illustrate how the different parts interact.

3.1.1 Benchmarking software

For executing benchmarks, we considered multiple options that could all run standard database benchmarks. Specifically, we investigated *oltpbenchmark* [8], *YCSB*[1], *go-ycsb*[2] and *go-tpc*[3]. Out of these four tools, the only

[1]https://github.com/brianfrankcooper/YCSB
[2]https://github.com/pingcap/go-ycsb
[3]https://github.com/pingcap/go-tpc

one to support more than one type of workload was oltpbenchmark. For this reason, combined with the advantages we describe below, we chose to base our system on oltpbenchmark.

Oltpbenchmark [8] was created because of the lack of standardized tooling for running benchmarks. Many benchmarks were hard to reproduce, and the authors hypothesized that their standardized tool might help. A benchmark is defined by configuring a database connection and workload settings in an XML file, which can then be published alongside the results of benchmarks for easy reproduction. Oltpbenchmark supports 15 different benchmarks and any database that has a Java Database Connectivity (JDBC) driver.

3.1.2 Database clusters

Starting a distributed database cluster to run benchmarks against is no small task. Servers first need to be started and configured, and the database software needs to be installed. Finally, the different servers need to be made aware of each other so that they can communicate and create a cluster. In many cases, servers also have different roles in the cluster which need to be configured. All of this is a lot of work to perform manually, and the many steps involved also makes it error-prone when trying to reproduce a benchmark.

We decided to automate the cluster setup process to make benchmarks faster to execute and easier to reproduce. To accomplish this, we use a combination of Packer [11], Terraform [12] and cloud-init [5]. Packer handles build time configuration of servers, Terraform communicates with a cloud API to start servers, and cloud-init handles run time configuration of servers. These three components are described in detail below.

Packer

Packer is used to create snapshots of a fully configured server that can be used when starting new servers. A snapshot, often called an image, is created by starting a clean server, running some configuration commands based on a user-defined manifest, and finally requesting for the cloud service to generate a snapshot based on this server. The building of this snapshot can be seen as the build time configuration of databases in our system. However, this configuration alone is not enough to start a database cluster, since IPs of the servers that need to communicate in the cluster are not known at this point. To address this issue, there is a separate run time configuration that is described in the cloud-init section below.

Using Packer for configuring servers has multiple advantages. First, it speeds up cluster setup significantly, as servers are preconfigured with software preinstalled when started. Second, allowing the configuration of servers to be defined as code makes it easier to reason about system state and to reproduce tests. However, one disadvantage of using Packer is that changes to the configuration mean that a new snapshot needs to be created and servers need to be recreated based on this snapshot. This is not a large issue for our system though, as servers only live for the duration of a single benchmark.

Terraform

Terraform is used to start the different servers that make up a database cluster, and it communicates with a cloud API to create these servers and connect them to a network. Terraform has its own configuration language where one can define any cloud resources that should be created and the relation between them. When starting servers with Terraform, an image created by Packer can be chosen as the base for the server, meaning the server will get an exact copy of the configuration generated in the Packer build.

Terraform supports all of the largest and many small cloud providers. However, since all clouds have different offerings, a Terraform file written for one cloud will not work for another cloud without some small adjustments. We only provide configuration files for the cloud that we used in our work, but the modifications needed to run the benchmarks on another cloud are minor.

Cloud-init

As mentioned earlier, the build time configuration that Packer performs is not enough to start a cluster, as servers need to be made aware of their role in the cluster and their peers. For this purpose, we use cloud-init [5], which runs a series of tasks the first time a server boots. These tasks can involve installing software, configuring networks, running scripts and changing files. In our case, we use cloud-init to write a few configuration files and start the required services for the server's role in the cluster. More specifically, the files written include cluster configurations with the network addresses of peers, and the services started can be e.g. the master node service. In practice, cloud-init works by passing a configuration string to the cloud API when the server is being created. This is handled by Terraform, as Terraform is the component in our system that handles communications with the cloud API.

3.1.3 Control script

The components described in the previous section, Packer, Terraform and cloud-init, together enable the creation of a cluster without manual work. However, to run complete tests, there is still some manual work involved, since each step needs to be initiated manually: starting the cluster, starting the benchmarks and stopping the cluster. In order to eliminate manual work altogether, we have automated the flow of these operations in our system. This also enables a user to easily run multiple benchmarks concurrently. Automating the flow of operations is performed by a Python script that takes as input the cluster configurations and workloads that should be tested, and then coordinates with Terraform and oltpbenchmark to execute benchmarks.

The control script starts by creating all the required clusters concurrently with Terraform. Each cluster also contains a client machine that has oltpbenchmark installed, which removes the risk of high WAN latencies affecting performance, while also ensuring isolation between benchmarks as each benchmark has a separate client. Once each cluster has successfully been created and cloud-init has run, the script logs into the client machine through SSH, uploads the benchmark workload XML files, and runs the benchmarks sequentially. After all benchmarks are completed, the result files are downloaded for later analysis. Finally, the cluster is shut down by using Terraform, and the benchmark is considered as completed.

3.1.4 System overview

The different moving parts of our benchmarking system are illustrated in Figure 3.1. The control script first communicates with the cloud API through Terraform to start all the necessary servers. After all servers are started, the control script communicates with the clients to start and keep track of benchmarks. The benchmark client runs oltpbenchmark, and it communicates with the cluster through a load balancer to ensure even load between cluster nodes. Finally, the cluster is configured behind the load balancer. The figure shows a cluster with three identical instances, but this configuration differs depending on which database is running. The control script can also communicate with the cloud API to start multiple independent configurations of the cluster, load balancer and client. This enables faster collection of results, since different benchmarks run in parallel instead of sequentially.

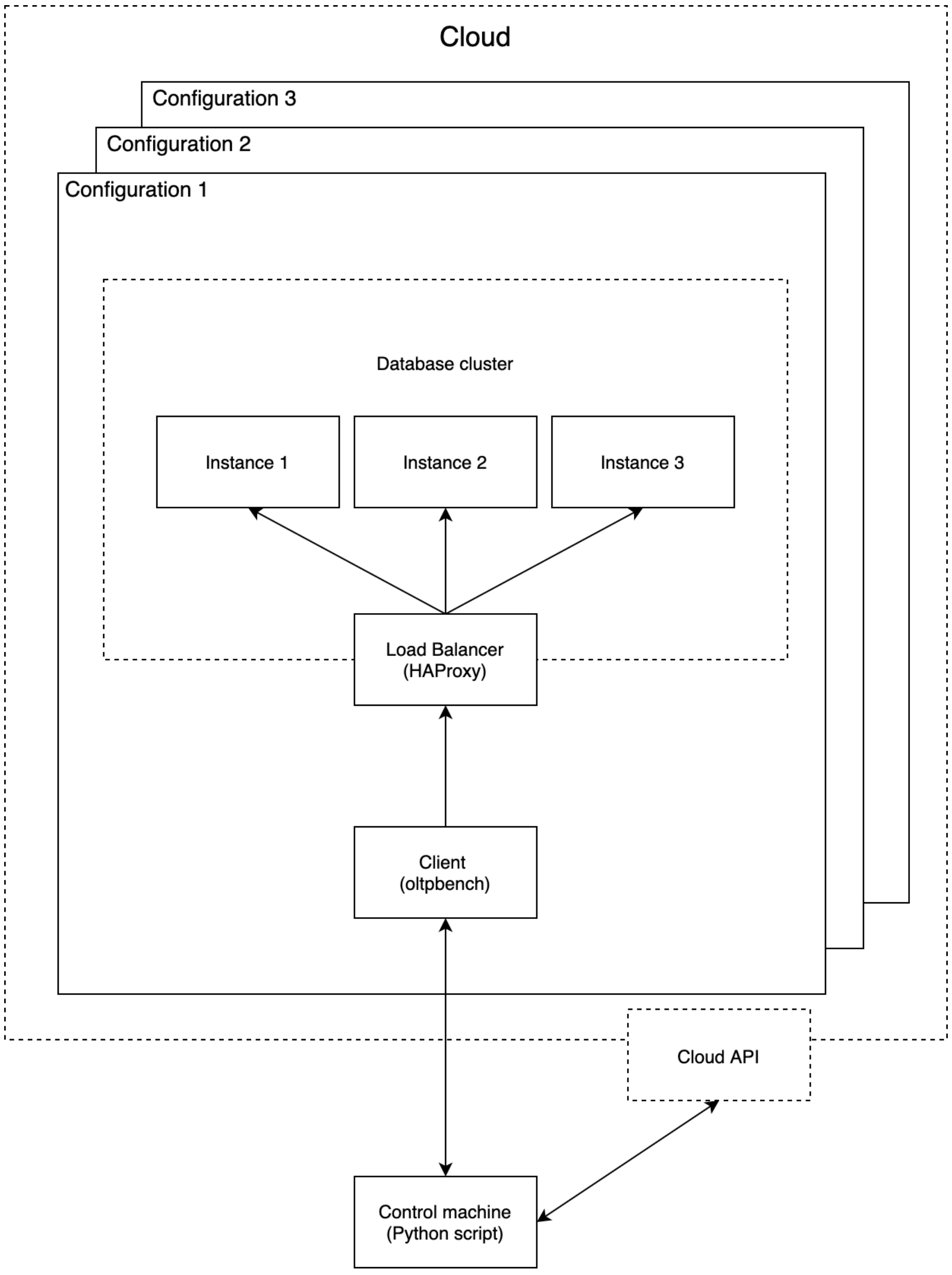

Figure 3.1: An overview of the different parts of a benchmark that is being executed with our system.

3.2 Benchmarks

For this thesis, we decided to test the databases with two different benchmarks that are commonly used to test databases, as it allows us to see how the databases perform in different scenarios. The choices of benchmarks are described in the two sections below.

3.2.1 YCSB

Yahoo! Cloud Serving Benchmark (YCSB) is a popular benchmarking tool created to test a multitude of databases. YCSB operates on a single user table and only performs simple key-value operations on this table. This means that YCSB supports most databases, but does not test any relational features of the database. However, it can still be a useful benchmark for relational systems, as it is able to show the raw performance of each system. YCSB defines four operations that can be performed on the user table: read a record, insert a record, update a record, and scan n records. These operations are combined into six standard workloads that are commonly used in benchmarks. The workloads are named A through F and are described below:

- **Workload A – Update heavy:** This workload has a 50% reads and 50% updates.
- **Workload B – Read mostly:** This workload is another variant of workload A, but with a different balance: 95% reads and 5% updates.
- **Workload C – Read only:** This is the simplest workload with only reads.
- **Workload D – Read latest:** This workload inserts new rows while reading mostly new records. There are 95% reads and 5% inserts.
- **Workload E – Short ranges:** Short ranges are scanned in this workload instead of fetching individual rows. There are 95% scans and 5% inserts in this workload.
- **Workload F – Read-modify-write:** Clients retrieve rows, modify them and then write this change to the database. There are 50% reads and 50% read-modify-write operations in this workload.

3.2.2 TPC-C

TPC-C is a benchmark meant to measure the performance of online transaction processing (OLTP) databases [25]. TPC-C is an industry standard benchmark and is defined by the Transaction Processing Performance

Council. The benchmark simulates the activity of a wholesale supplier that has a number of warehouses. Each warehouse is linked to ten districts, and each district is linked to three thousand customers. The benchmark has five different transaction types that access a total of nine different tables. This benchmark tests a wider set of functionality of the databases we are testing, as tables include additional constraints such as foreign keys and unique columns. The performance of TPC-C is measured in the number of "new order" transactions that are executed per minute.

3.3 Choice of cloud provider

For our research, we considered different cloud providers to run our benchmarks on. Some of the large providers like Amazon Web Services, Google Cloud Platform and Microsoft Azure are often the go-to platforms when choosing a public cloud. These clouds provide numerous advantages by being very large and having a lot of managed services. There are many other providers in the cloud market, and they usually have fewer services but much cheaper pricing. One of these clouds is Hetzner Cloud, and it only offers Virtual Machines, networking and disks. However, all we need for our running our benchmarks are virtual servers and disks. Additionally, there is a larger incentive to run your own NewSQL database if using a smaller cloud, as they usually do not offer a managed database service.

After talking to Hetzner Cloud, we received a generous grant that would allow us to perform our research on the platform. This, in addition to wanting to use a platform that did not have an existing database offering, was the basis for our decision to run on Hetzner Cloud.

3.3.1 Variable server performance

In a cloud environment one usually shares hardware with multiple other tenants. This causes performance to often vary a lot, which is often referred to as "noisy neighbours", as one can notice other tenants (neighbours) running on the same machine. Hetzner Cloud has an offering where one can get dedicated virtual CPUs, which reserves a number of hyperthreads on the CPU to only your virtual server. This works well and yields very stable CPU performance. However, Hetzner Cloud does not provide any guarantees for memory or disk performance. The disks are very performant, but the performance is highly variable, both between different servers and even on the same server over time.

As the performance of virtual machines is variable, the result of benchmarks will also vary between runs, which means that we need to gather multiple samples so that we get good average performance results. However, gathering results from multiple runs is simple with the automated system as we described in Section 3.1.

3.4 Database specifics

For our tests we decided to test three open source NewSQL databases: CockroachDB, TiDB and YugabyteDB. Each of the databases needed to be configured differently, and we wrote separate Packer and Terraform manifests for each database. For all of the databases we used the default replication level of three. We have described specifics of how we configured and ran each database in the following sections.

3.4.1 CockroachDB

CockroachDB, as described in Section 2.12, runs as a homogenous cluster without any dedicated master nodes. We configured CockroachDB to run on Ubuntu 18.04, as suggested by their documentation. We configured network time protocol (NTP) on the servers and installed CockroachDB v20.1.0. Further, we set up a HAProxy instance between the client and the cluster that all requests were routed through. Load balancing is recommended in CockroachDB's documentation as it ensures that requests can be balanced evenly between nodes.

Since clusters in CockroachDB are homogenous, there are only two cluster variables to tune: the number of nodes in the cluster and the hardware resources of these nodes. CockroachDB supports the Postgres wire protocol, and we therefore used the Postgres JDBC driver to run benchmarks against CockroachDB. However, some issues arose when CockroachDB was returning retryable errors that oltpbench did not support. We had to modify oltpbench slightly in order to retry when receiving one of these errors.

3.4.2 TiDB

TiDB, described in Section 2.13, has three different components in a cluster: the placement driver which acts as a kind of master, TiKV which stores key-value data and TiDB which handles SQL execution and translation to key-value operations. As recommended by TiDB's documentation, we configured the cluster to run on CentOS 7. We also configured some system limits on the operating system, following the instructions from TiDB's documentation [23]. Finally, we installed TiDB v3.0 on the servers.

When starting a cluster, placement drivers need to be started first and communicate directly with each other. Next, TiKV servers are started and communicate with placement drivers for synchronization. Finally, TiDB servers are started and also communicate with placement drivers. Placement drivers act as masters of the cluster and enable the other components to communicate. We also configured a HAProxy load balancer between the client and the TiDB instances to spread the load evenly between the servers. Balancing of TiKV is handled internally by TiDB, as different TiKV instances are responsible for different sets of shards.

TiDB has many parts that can be tuned when testing performance. We decided to use the same instance size for all components for simplicity, but varied which instance type we used. Additionally, we varied the number of TiKV and TiDB nodes.

TiDB supports the MySQL wire protocol, and we were able to use the MySQL JDBC driver to run benchmarks against TiDB. TiDB worked without modifications with all of the benchmarks we performed against it.

3.4.3 YugabyteDB

YugabyteDB is described in Section 2.14. A cluster consists of two parts: master nodes and TServers, and the master software is usually run on the same node as a TServer. We used CentOS 7 as the operating system for this cluster, as recommended by YugabyteDB's documentation. Next, we configured NTP and some limits on the operating system, and installed YugabyteDB v2.1.6.0.

To start a cluster, we first started the master nodes and allowed them to communicate directly. Afterwards, all the TServer processes were started and pointed to the master nodes for registration. Since any TServer can receive client requests, we configured HAProxy between the client and TServers to balance the load. For benchmarks we varied the number of TServers and the instance sizes of these.

YugabyteDB supports the Postgres wire protocol, and we could therefore use the Postgres JDBC driver to run benchmarks. We initially had some issues with benchmarks getting stuck on table creation, but solved this by adding retries in case of failure. YugabyteDB also returned some retryable errors that oltpbenchmark did not recognize. We therefore implemented some additional logic to recognize and retry these transaction failures.

3.5 Final test configuration

As described above, we are testing CockroachDB, TiDB and YugabyteDB. YugabyteDB also supports two different isolation levels, serializable and snapshot, and we test both of these settings separately, meaning we test a total of four databases. Because of the performance variance described in Section 3.3.1, all of our tests are run on five separate clusters to get a good average performance value.

Each database configuration is tested with seven different workloads, six of which are YCSB workloads and the last being TPC-C. The workloads are described in depth in Section 3.2. Each YCSB benchmark uses a table of 100,000 records, while the TPC-C benchmark uses 2 warehouses. All workloads are run for 5 minutes each, and we use the average performance for comparisons.

To analyze the vertical scalability of each database, we test them with four different instance sizes that are available on Hetzner Cloud. These have a different number of CPU cores and sizes of memory and disk, but they all have the same disk performance and no guaranteed disk speeds. We believe this is the largest factor of variance in tests. The instances sizes we used are listed below.

- 2 virtual CPUs, 8 GB RAM, 80 GB NVMe disk
- 4 virtual CPUs, 16 GB RAM, 160 GB NVMe disk
- 8 virtual CPUs, 32 GB RAM, 240 GB NVMe disk
- 16 virtual CPUs, 64 GB RAM, 360 GB NVMe disk

In addition to evaluating vertical scalability, we investigate horizontal scalability. To do this, we test each database with various cluster sizes between 3 and 12. For all of these tests, we use the instance size of 4 CPUs and 16GB RAM.

Chapter 4

Results and Discussion

This chapter presents and discusses the results of running our benchmarking system on various configurations of CockroachDB, TiDB and YugabyteDB, as described in Section 3.5. We run both YCSB and TPC-C on different instance sizes and on clusters with different numbers of nodes. In total, we have run more than 200 different clusters using over 1,500 virtual machines for these evaluations.

We start by presenting the YCSB results for each database separately to analyze what configurations they each work best with. Next, we present some comparisons for both YCSB and TPC-C that will show the strengths and weaknesses of each database, and finally, we compare the scalability numbers of each database.

4.1 CockroachDB

Instance size

To investigate CockroachDB's vertical scalability, we analyze the performance of different instance sizes using clusters of three nodes. The experiment was conducted by taking the average of five samples as mentioned previously. The results of the test can be seen in Figure 4.1, where all of the YCSB benchmarks are shown for four different instance sizes, each being twice as powerful as the previous. In order to see how much performance per hardware the different configurations yield, Figure 4.2 normalizes the performance number by dividing by the number of CPU cores. In an ideal world, normalized performance should stay constant, meaning that all types of queries scale perfectly with the provided resources, but in reality this can not be expected.

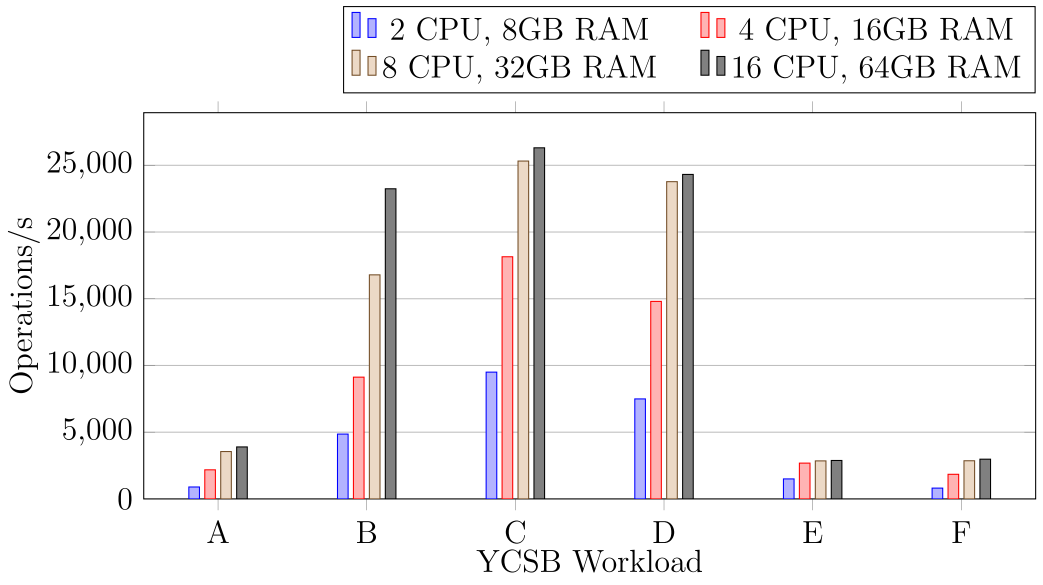

Figure 4.1: YCSB performance for three-node CockroachDB clusters with different instance sizes.

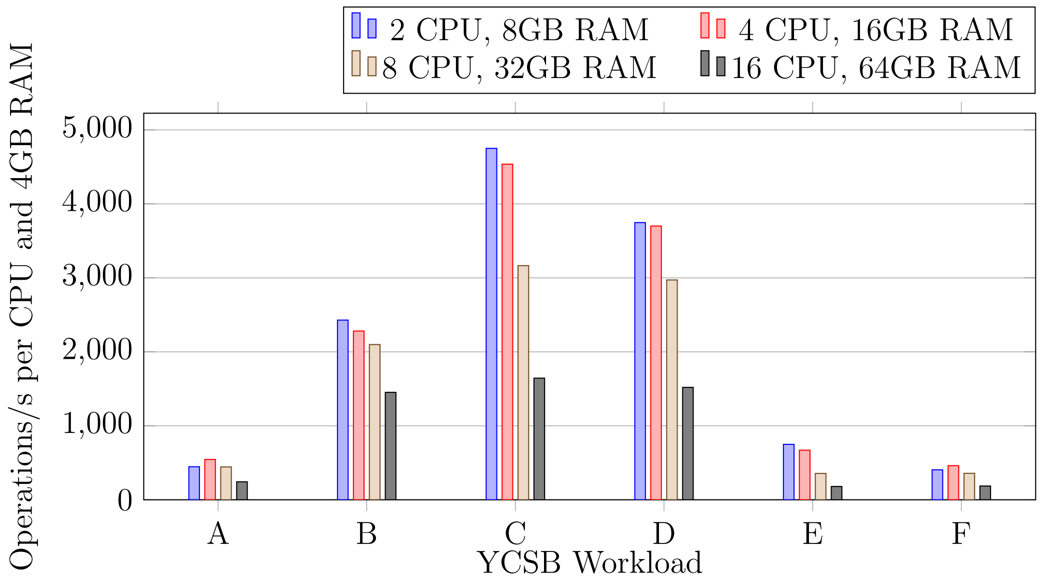

Figure 4.2: YCSB performance per CPU and 4GB RAM for three-node CockroachDB clusters with different instance sizes.

Cluster size

Next, we conduct an experiment to investigate how well CockroachDB's performance scales when adding nodes. We used nodes with 4 CPUs and 16GB of RAM and set up clusters of sizes 3, 6, 9 and 12. Figure 4.3 shows the results of these tests, and Figure 4.4 shows a normalized performance score per node.

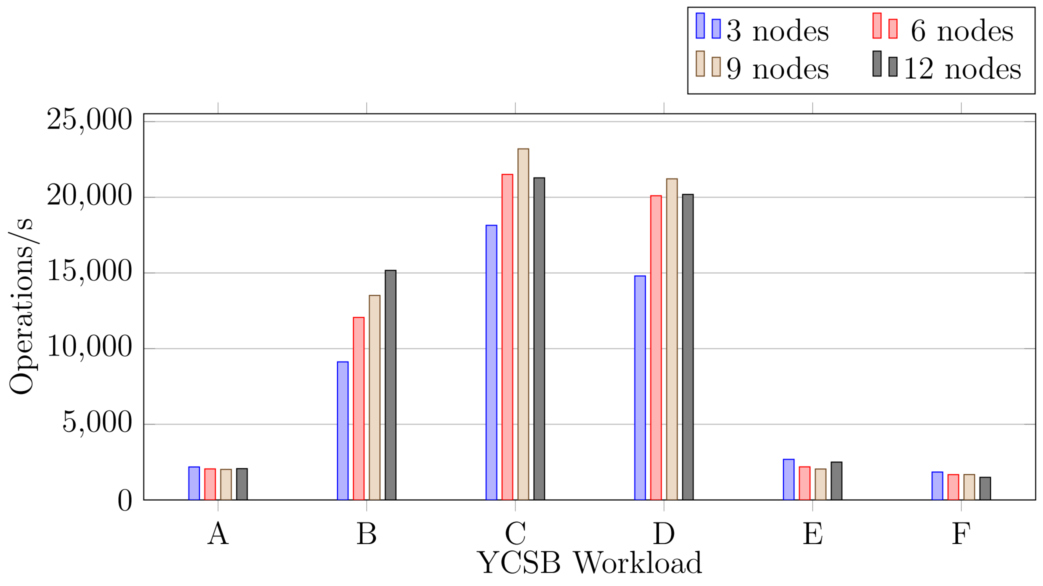

Figure 4.3: YCSB performance for CockroachDB clusters with different numbers of nodes, using 4 CPU 16GB RAM nodes.

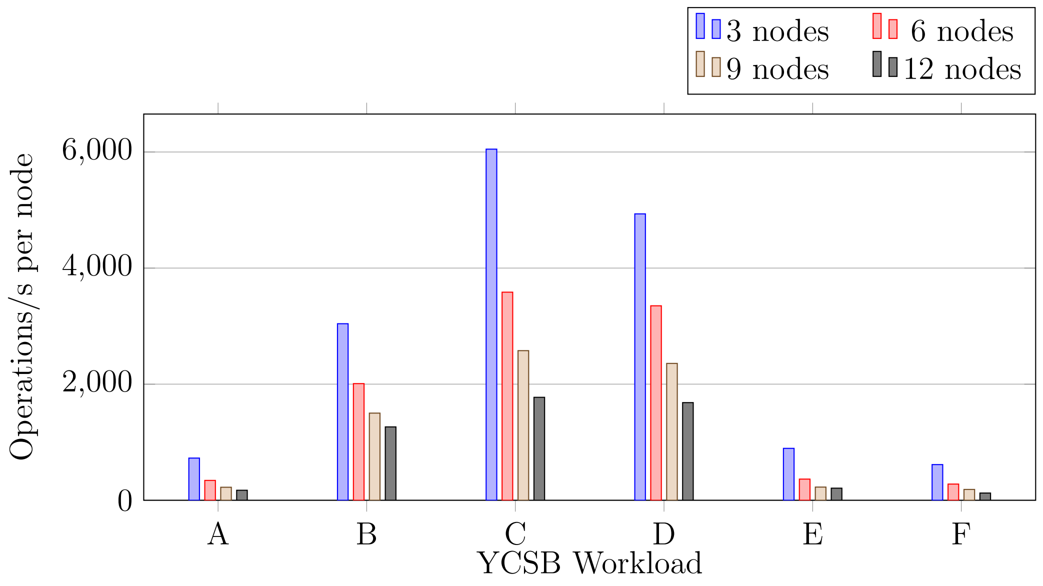

Figure 4.4: YCSB performance per node for CockroachDB clusters with different numbers of nodes, using 4 CPU 16GB RAM nodes.

Evaluation

Based on the presented figures, we can see that CockroachDB scales better vertically than it does horizontally for the workloads we ran. However, increasing from 8 to 16 CPUs seems to have almost no effect except for workload B. CockroachDB's per core performance is best with the smallest instance sizes, and this may be because the disk performance on all instances is the same, meaning that smaller instances get higher disk performance per cost. Finally, we can see that workload E scales very poorly, and increasing hardware resources has little effect on performance. We believe this may also be caused by scan operations being I/O-heavy compared to other operations.

CockroachDB's horizontal scalability is not very promising in the tests we ran. The write-heavy workloads A and F do not increase performance at all, and doubling the number of nodes increases performance by less than 50% for the other workloads. One surprising result is that 12 node clusters do not increase performance at all over 9 node clusters for workloads C, D and F. We believe that this is caused by the fact that our test datasets are only 100,000 rows, which might not be enough to balance the data evenly between nodes. We believe that there would be a larger difference between the performance of cluster sizes if using a larger test dataset or different workload types.

4.2 TiDB

TiDB clusters contain nodes with three different roles: TiDB, TiKV and PD. Each of these can be scaled independently, but the main performance differences should come from scaling TiDB and TiKV, as PD acts as a single node, independent of replication factor.

To find which cluster configurations to experiment with, we started by investigating the performance effect of PD. TiDB's documentation mentions that clusters can use any odd number of PDs, but that three is the minimum for failure tolerance. However, for our tests, we do not need failure tolerance, meaning that if there is no performance difference between one and three PDs, we can use smaller clusters and get the same results. The documentation also mentions that it is possible to run PD on either a separate node or on a shared node with TiDB. To investigate the effect of these cluster configurations, we performed an experiment with four different configurations listed below. Our hypothesis is that the performance is the same for all configurations, and that we can therefore use the simplest configuration in the rest of our tests.

1. 3 PD, 1 TiDB, 3 TiKV
2. 1 PD, 1 TiDB, 3 TiKV
3. 3 PD, 1 TiDB, 3 TiKV (TiDB running on same node as a PD)
4. 1 PD, 1 TiDB, 3 TiKV (TiDB running on same node as a PD)

The result of these four configurations with standard deviations can be seen in Figure 4.5, and they show that our hypothesis was correct. As the performance is not significantly different between the configurations, we can use the configuration with one PD and one TiDB on a shared node. All further tests of TiDB will use this setup, and when using multiple TiDB nodes, PD will be running on only one of the TiDB nodes.

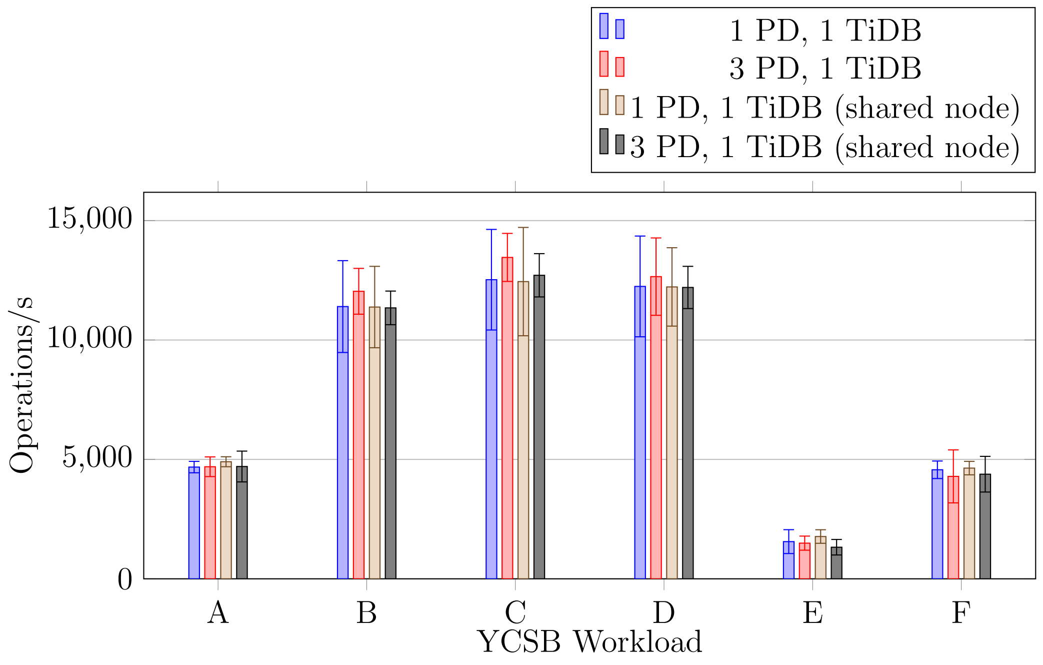

Figure 4.5: TiDB clusters with different configurations for PD and TiDB. Each cluster has three TiKV nodes and uses 4 CPU 16GB RAM nodes.

Instance size

Next, we investigated the vertical scalability of TiDB. We used TiDB's minimal configuration, as described above, and tested the cluster with four different instance sizes. It is also possible to run TiDB and TiKV on different instance sizes, but we chose to use uniform clusters in these tests to make the analysis simpler. The results of the experiment can be seen in Figure 4.6. Figure 4.7 shows the performance per CPU core and 4GB of RAM, giving a picture of the performance per hardware resource when scaling.

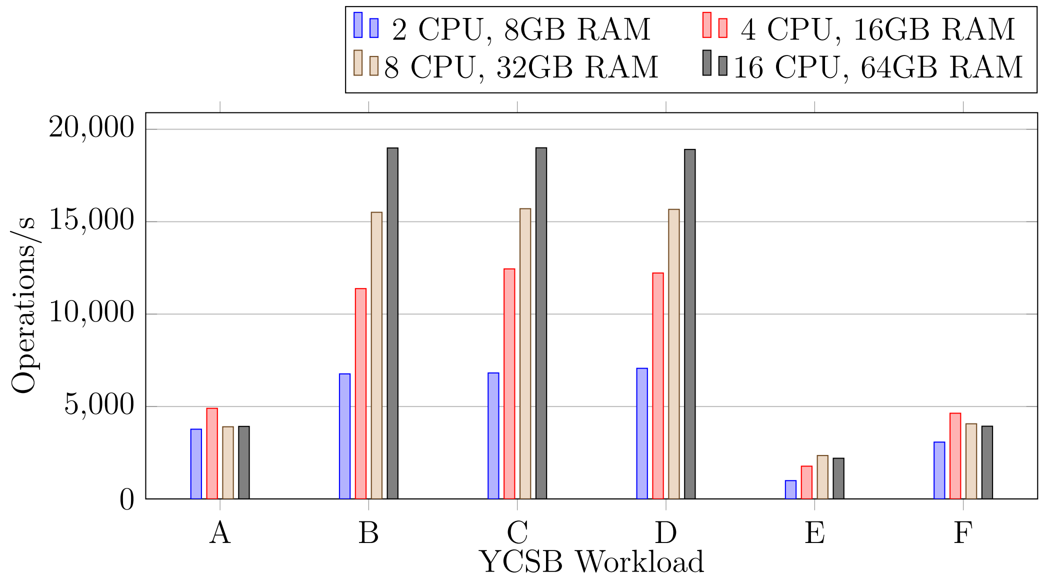

Figure 4.6: YCSB performance for four-node TiDB clusters with different instance sizes. The cluster consists of three TiKV nodes and one shared TiDB and PD node.

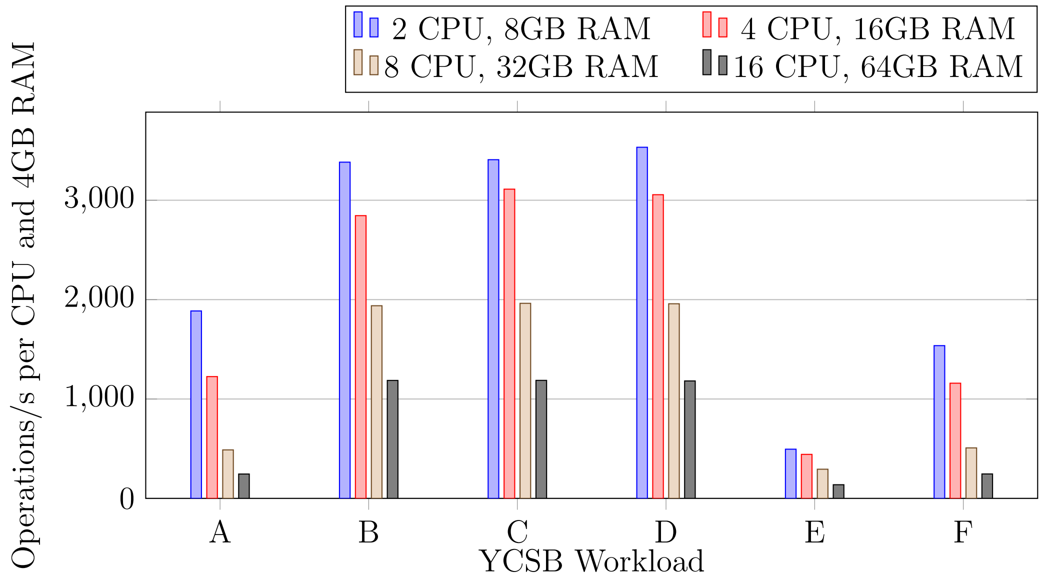

Figure 4.7: YCSB performance per CPU and 4GB RAM for four-node TiDB clusters with different instance sizes. The cluster consists of three TiKV nodes and one shared TiDB and PD node.

Cluster size

As TiDB clusters can be scaled by both varying the number of TiDB and TiKV nodes, we decided to test a set of different combinations listed below. Note that all of these configurations have one instance of PD running on one of the TiDB nodes.

- 3 TiKV, 1 TiDB (4 nodes in total)
- 3 TiKV, 3 TiDB (6 nodes in total)
- 3 TiKV, 6 TiDB (9 nodes in total)
- 6 TiKV, 1 TiDB (7 nodes in total)
- 6 TiKV, 3 TiDB (9 nodes in total)
- 6 TiKV, 6 TiDB (12 nodes in total)

These clusters were all run on 4 CPU, 16GB RAM nodes, and the results of our tests can be seen in Figure 4.8. We have also normalized these results to show how much performance per node the clusters give in Figure 4.9.

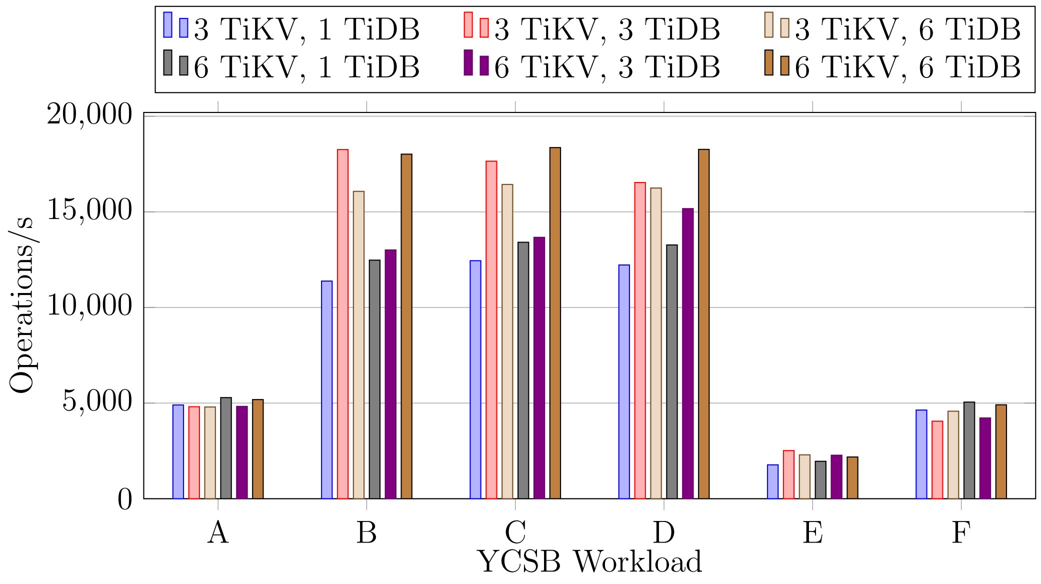

Figure 4.8: YCSB performance for TiDB clusters with different numbers of TiDB and TiKV nodes, using 4 CPU 16GB RAM nodes. All clusters use one PD running on a TiDB node.

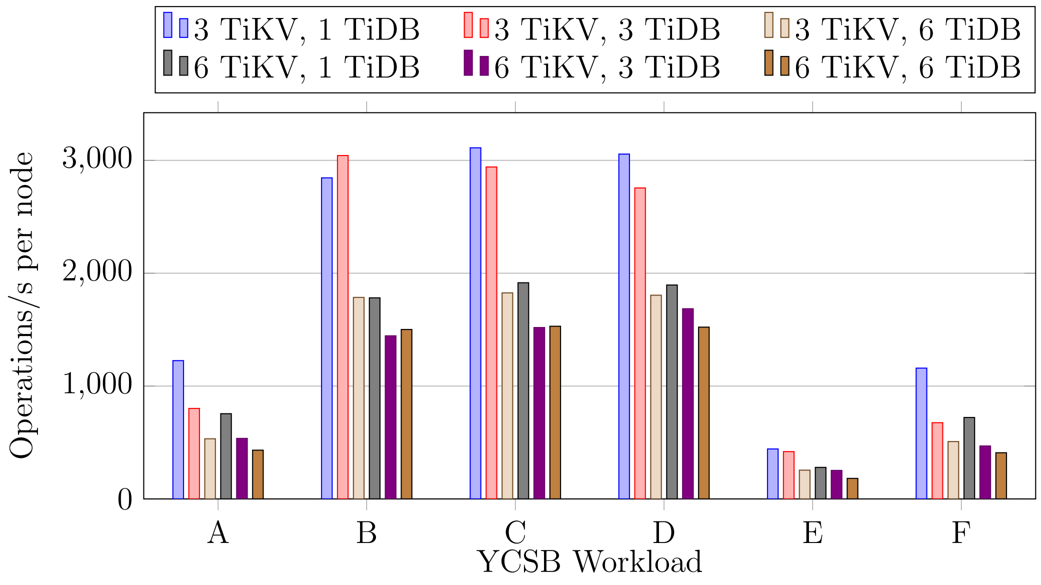

Figure 4.9: YCSB performance per node for TiDB clusters with different numbers of TiDB and TiKV nodes, using 4 CPU 16GB RAM nodes. All clusters use one PD running on a TiDB node.

Evaluation

TiDB's performance for workloads B, C and D scales consistently when vertically scaling a cluster. However, the write-heavy workloads A and F do not scale particularly well, and the performance on the scan workload E does not increase by using the largest instance size. We believe that workloads A and F may scale poorly because of contention to some keys, in addition to the disk performance being the same on all instances. Looking at the workloads that do scale, they scale best for the two smallest instance sizes and lose more performance when scaling beyond that. Again, we believe this may have to do with constant disk performance, as increasing cores and memory are not the only qualities that affect performance.

TiDB's horizontal scalability is a bit harder to analyze because of the many combinations of configurations. Looking at Figure 4.9, we can see that the two smallest clusters have the best performance per node. They have almost identical per-node performance in workloads B, C and D and E, while the smallest cluster is best for write heavy workloads A and F. This is likely because all disk operations are handled by TiKV, while TiDB is only a stateless query layer on top of that, and thus, adding more TiDB instances does not increase write performance. However, for reads and scans, the increased capacity in the query layer seems to increase performance significantly.

Another interesting finding is that when using 3 TiKV nodes, increasing from 3 to 6 TiDB nodes does not have any positive impact on the results. This indicates that at this point, TiKV is the bottleneck, and any extra TiDB capacity will be unused. Finally, when using 6 TiDB nodes and going from 3 to 6 TiKV nodes, the throughput of the cluster is actually reduced. We have no good explanation for this, but believe it is largely caused by highly variable cluster performance, and that a larger sample size would remove this anomaly.

4.3 YugabyteDB

YugabyteDB supports both serializable isolation like CockroachDB and snapshot isolation like TiDB. The two isolation levels perform significantly different, so we have chosen to present both results in separate sections below.

4.3.1 Snapshot isolation

Instance size

We performed instance size tests with four different hardware configurations, using a minimal cluster of three nodes. The results of the experiment can be seen in Figure 4.10 and the normalized performance can be seen in Figure 4.11.

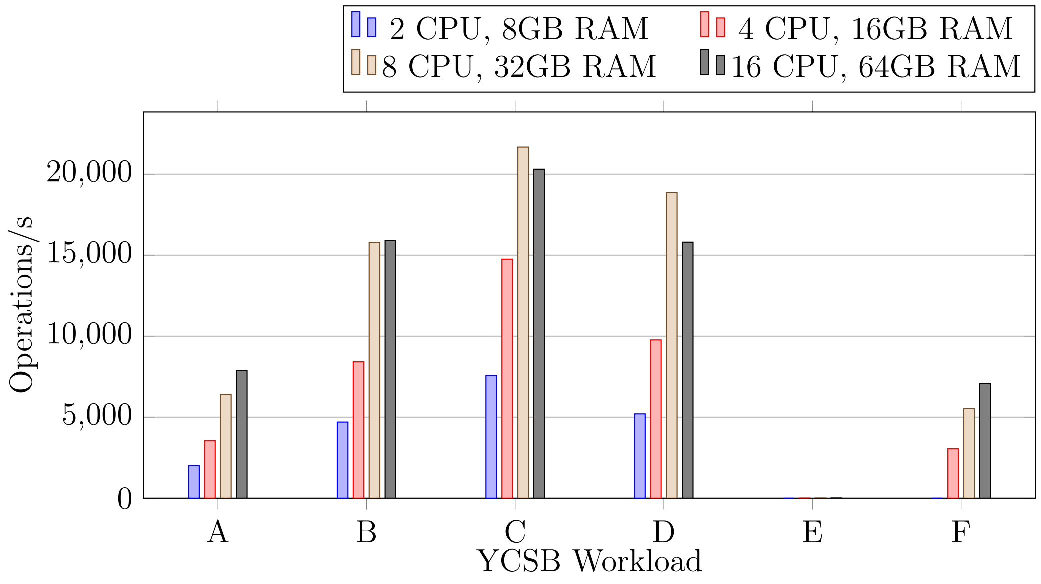

Figure 4.10: YCSB performance for three-node YugabyteDB clusters with snapshot isolation on different instance sizes.

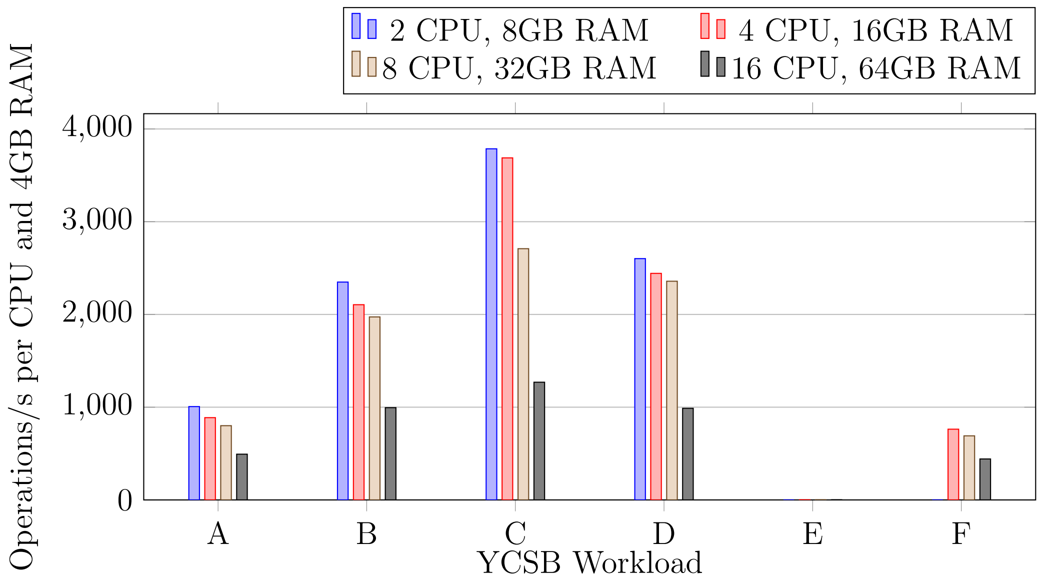

Figure 4.11: YCSB performance per CPU and 4GB RAM for three-node YugabyteDB clusters with snapshot isolation on different instance sizes.

Cluster size

To get a picture of the horizontal scalability of YugabyteDB, we test clusters of sizes 3, 6, 9 and 12 nodes. Each of the clusters TServers running on all nodes, and three masters running on the same nodes as TServers. For all cluster sizes, we use nodes with 4 CPU and 16GB of RAM. The results can be seen in Figure 4.12, and Figure 4.13 shows the per node performance numbers.

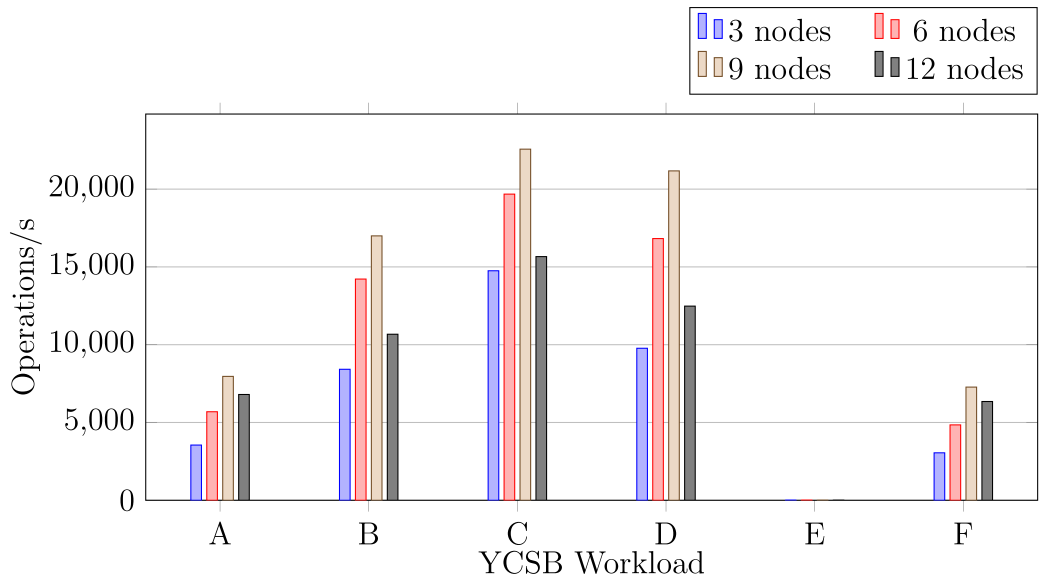

Figure 4.12: YCSB performance for YugabyteDB clusters with snapshot isolation using different numbers of 4 CPU 16GB RAM nodes.

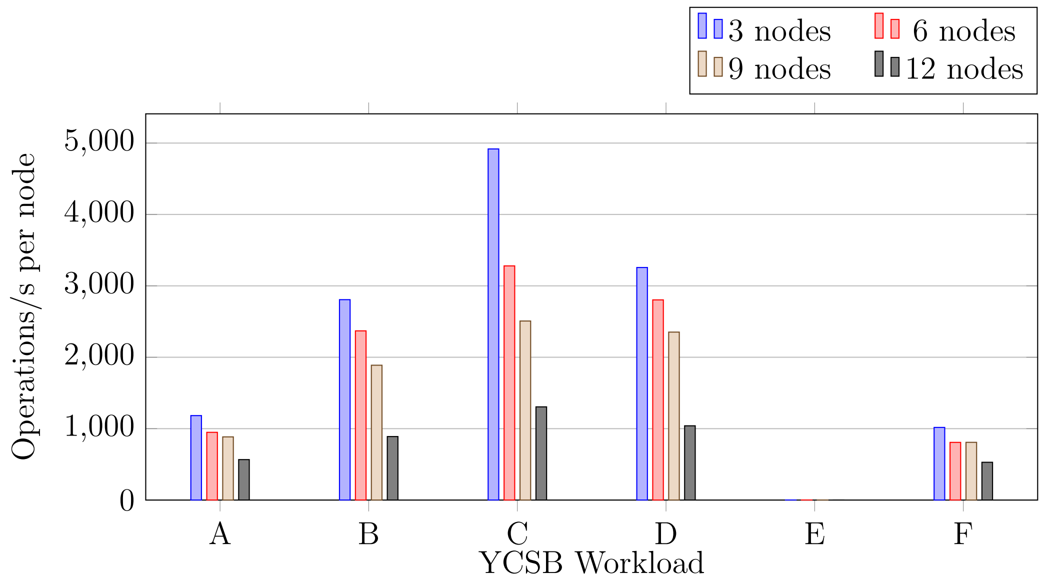

Figure 4.13: YCSB performance per node for YugabyteDB clusters with snapshot isolation using different numbers of 4 CPU 16GB RAM nodes.

Evaluation

The first thing that is very noticeable in these results is the performance of workload E. Workload E achieves only between 5 and 10 operations per second, with a latency of over 30 seconds. One can therefore conclude that workload E is not compatible with YugabyteDB's snapshot isolation. One explanation for this performance is that YugabyteDB partitions data with hash partitioning rather than range partitioning. This makes scans more expensive, as many different shards and nodes need to communicate in order to serve a query. However, there is likely some other factor involved in this slow performance as well, as the number of nodes is low and the communication overhead is therefore somewhat limited.

One more issue with YugabyteDB with snapshot isolation is that workload F fails for the smallest instance size, but scales well for the three other instance sizes. We have no good explanation for this failure, but believe it may be caused by many conflicts that result in unrecoverable transaction errors. This might be fixed by some modifications to oltpbenchmark's error handling, but this was not prioritized in this thesis.

Other than the issue with workload E, the database scales quite well for the rest of the workloads with the three smaller instance sizes. However, when going from 8 to 16 CPUs, there is no performance gain for workloads B, C and D. We suspect this has to do with the disk performance, as mentioned in the other databases. Overall, the three smallest instances have similar per-hardware performance, and YugabyteDB with snapshot isolation therefore seems to scale very well on these instances.

The horizontal scalability of YugabyteDB with snapshot isolation is similar to its vertical scalability, where it scales well up until a certain point. The largest cluster of 12 nodes even performs worse than the cluster of 6 nodes. This result combined with the vertical scalability leads us to believe, as mentioned earlier, that the dataset for these benchmarks might not be large enough to see the benefits of more powerful configurations. A small dataset is harder to distribute, and it will also introduce contention when operations are very fast.

4.3.2 Serializable isolation

Serializable isolation is the strongest isolation level that YugabyteDB provides. We expect this to have significantly lower performance than snapshot isolation, as it needs to prevent more anomalies and therefore do more checks during transaction runtime.

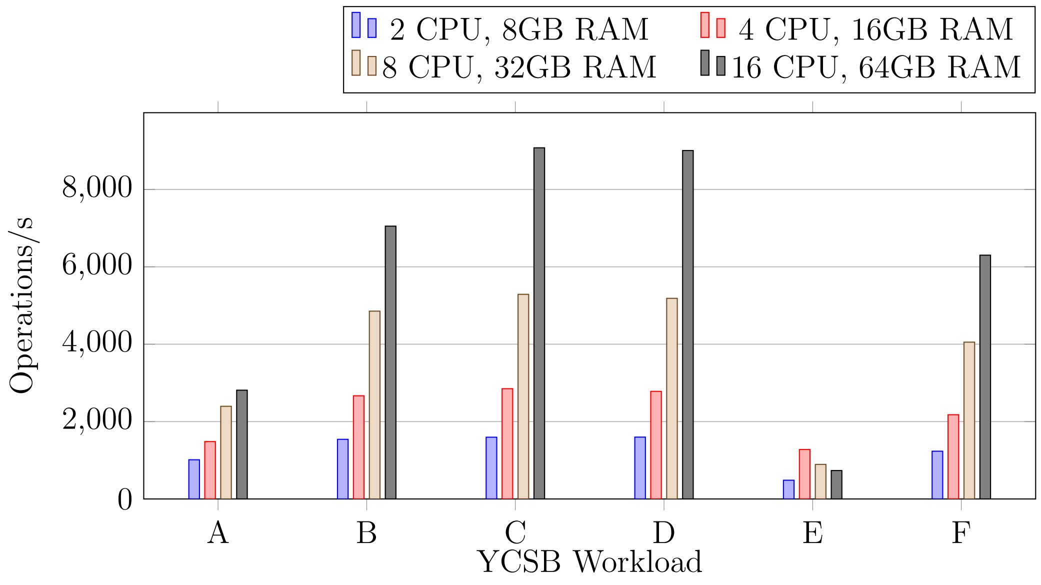

Figure 4.14: YCSB performance for three-node YugabyteDB clusters with serializable isolation on different instance sizes.

Instance size

For instance size tests, we use clusters with three nodes, each having the master and TServer software running, on four different instance sizes. The results can be seen in Figure 4.14. We also show a normalized version of this graph in Figure 4.15, where the performance per hardware resource is shown.

Cluster size

Varying cluster sizes with YugabyteDB serializable isolation were also tested, using 3, 6, 9 and 12 nodes. All of these clusters also had three master processes running on three of the TServer nodes. Figure 4.16 shows the performance results, while Figure 4.17 shows normalized numbers with per node performance.

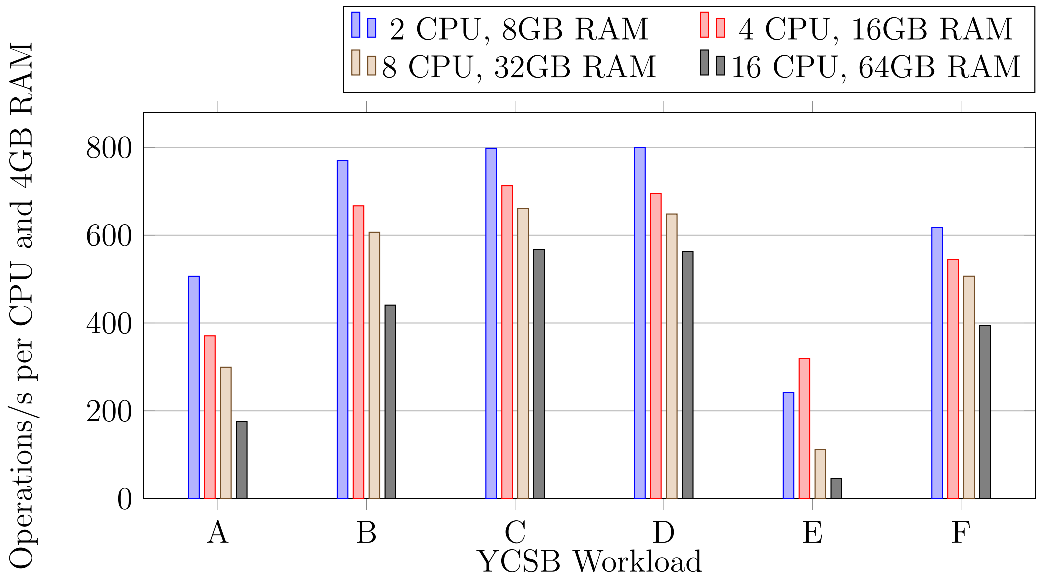

Figure 4.15: YCSB performance per CPU and 4GB RAM for three-node YugabyteDB clusters with serializable isolation on different instance sizes.

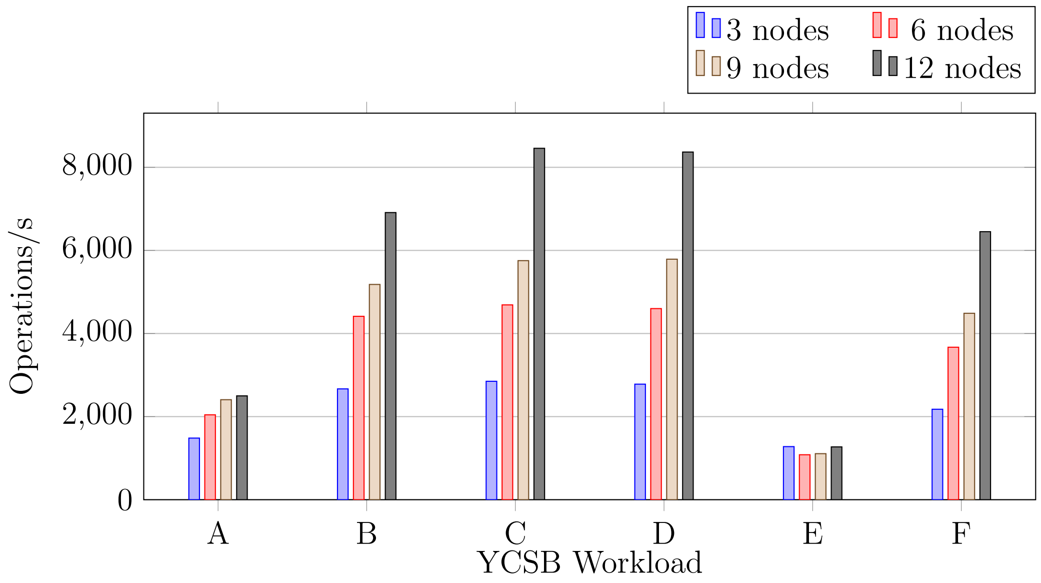

Figure 4.16: YCSB performance for YugabyteDB clusters with serializable isolation using different numbers of 4 CPU 16GB RAM nodes.

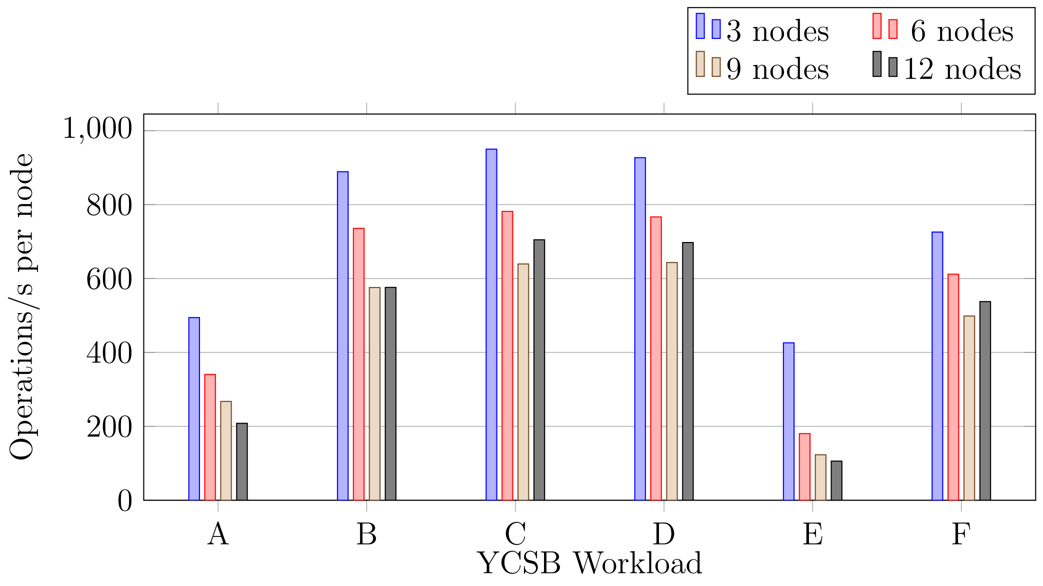

Figure 4.17: YCSB performance per node for YugabyteDB clusters with serializable isolation using different numbers of 4 CPU 16GB RAM nodes.

Evaluation

YugabyteDB with serializable isolation scales quite well for all benchmarks except E. However, unlike snapshot isolation, serializable isolation is able to handle workload E with good results. The per core performance of YugabyteDB with serializable isolation is reduced steadily with the number of nodes, which is expected as overhead grows when more nodes are introduced in a cluster.

The horizontal scalability is similar to the vertical scalability, with all workloads scaling well except for the scan workload E. There are no issues when scaling to the largest instance or cluster sizes with this database, but we believe that this is caused by the absolute performance numbers being significantly lower than for other databases. As absolute numbers are lower, the issues with a small workload size are not as apparent, since fewer operations per second mean that a smaller subset of data is accessed.

4.4 Comparison

YCSB

To compare the performance of all four databases, we analyze the per node performance of a minimal cluster where nodes have 4 CPUs and 16GB of RAM for each database. The results are shown in Figure 4.18. The graph shows that CockroachDB performs best on read-heavy workloads B, C, D and E, while TiDB performs best on write-heavy workloads A and F. On average, YugabyteDB with serializable isolation performs worst of the four databases we tested. Another observation, as mentioned earlier, is that YugabyteDB with snapshot isolation is almost unable to handle workload E.

If we divide the databases by their isolation level, we can see that for serializable isolation, CockroachDB is a clear winner over YugabyteDB in these tests. For snapshot isolation, however, there is no clear winner between TiDB and YugabyteDB in general. The two big differences between TiDB and YugabyteDB are seen in workloads C and E. On workload C, YugabyteDB significantly outperforms TiDB, while on workload E, YugabyteDB has almost zero performance.

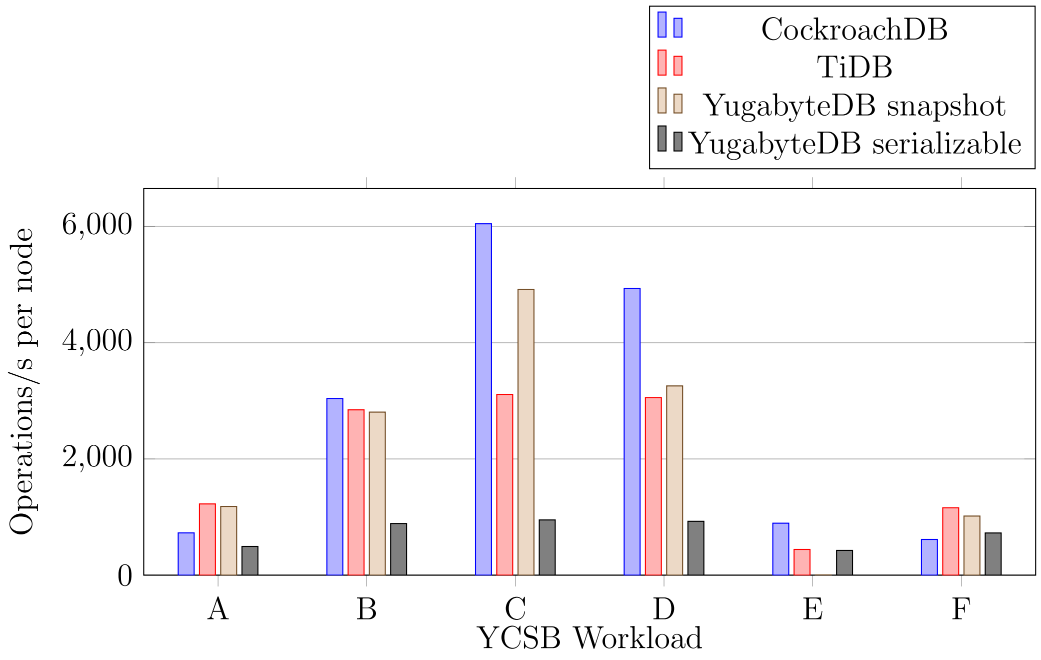

Figure 4.18: YCSB performance per node for minimal CockroachDB, TiDB and YugabyteDB clusters using 4 CPU 16GB RAM nodes.

Using the same benchmark as above, Figure 4.19 shows the average latency of each database. The graph is cut off on the y-axis because of YugabyteDB's snapshot isolation performance with workload E. The graph shows that in general, the databases with snapshot isolation have lower latencies than the databases with serializable isolation. This shows that there is a significant cost to the stronger isolation. When looking at CockroachDB in specific, one can see that the largest cost occurs for the write-heavy workloads A and F, while on read-heavy workloads, CockroachDB is comparable to the databases with snapshot isolation. This figure also shows that the latency of workload E for YugabyteDB, which is hash partitioned, is much higher than CockroachDB and TiDB, which are both range partitioned.

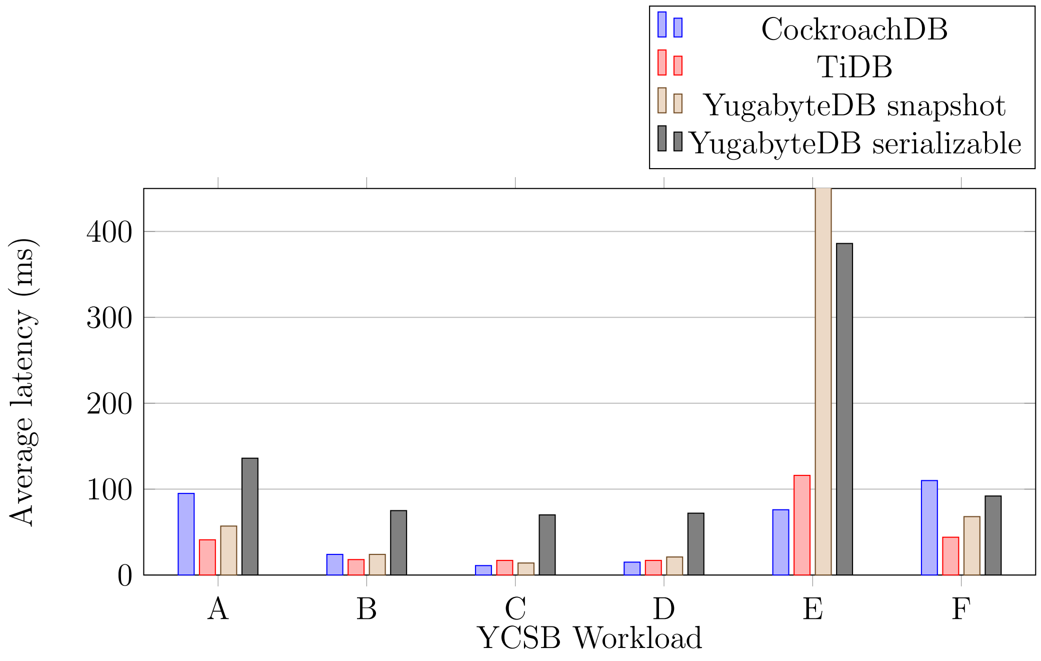

Figure 4.19: YCSB latencies for minimal CockroachDB, TiDB and YugabyteDB clusters using 4 CPU 16GB RAM nodes. The y-axis is cropped because YugabyteDB with snapshot isolation on workload E has a latency of over 30 seconds.

TPC-C

We also evaluated all of the databases with the TPC-C benchmark in order to get more data points to compare. However, there were some issues with Yugabyte where it would get stuck on table creation, and we were not able to resolve these issues. This means that we only have TPC-C numbers for CockroachDB and TiDB. Nevertheless, we present these results here for an additional data point in the comparison between the two. The results are shown in Figure 4.20, and we can see that the per node performance results are comparable. TiDB outperforms CockroachDB on a smaller instance size, while the opposite is true for the larger instance size. This figure also shows that the TPC-C benchmark's vertical scalability is not very good with these databases, as the performance of smaller and larger nodes is almost identical.

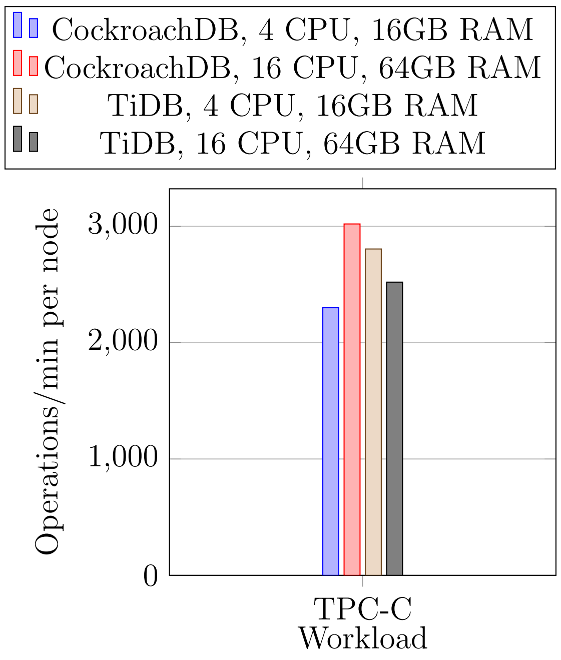

Figure 4.20: TPC-C performance per node for minimal CockroachDB and TiDB clusters using both 4 CPU 16GB RAM and 16 CPU 64GB RAM nodes. YugabyteDB is excluded because it was not able to run the TPC-C workload.

Scalability

Finally, we illustrate the scalability of each of the systems by finding the average performance per node scaled to the smallest cluster size. This means that the smallest cluster size has a performance per node of 1, while larger clusters have a number that reflects how much the performance per cluster is reduced. The smallest cluster for TiDB is four nodes, it is three for the other databases. The results of this are shown in Figure 4.21. The figure shows that YugabyteDB appears to maintain the highest per node performance while scaling, while CockroachDB is worst at scaling horizontally. However, it should be noted again that the absolute values for CockroachDB are higher, and that the benchmark datasets are likely too small, effectively punishing higher performing clusters.

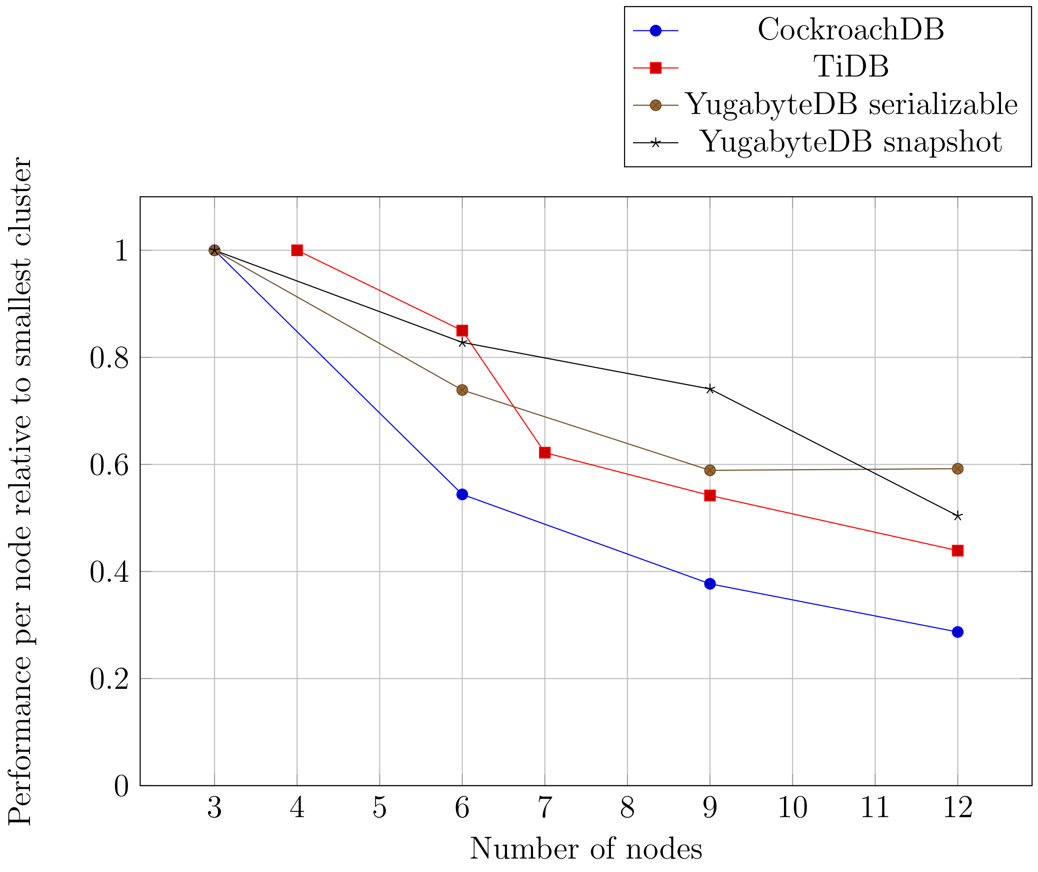

Figure 4.21: Comparison of performance per node for CockroachDB, TiDB and YugabyteDB when scaling horizontally. Performances are normalized to illustrate the cost of scaling by defining that minimal cluster of each database has a performance per node of 1.

4.5 Discussion

YugabyteDB started as a snapshot isolation database, and later added serializable isolation [26]. Analyzing the performance of the two isolation levels should show the "cost" of stronger isolation, and it appears that serializable isolation inherently can provide a much lower throughput than snapshot isolation. However, when also looking at CockroachDB and TiDB, it is apparent that CockroachDB, a database built from the start for serializable isolation, is able to compete quite well with snapshot isolation databases. The throughput of CockroachDB is competitive with the snapshot isolation databases on workloads B through E, but is slightly lower on write-heavy workloads A and F. This is to be expected, as write transactions are the ones that make isolation harder, as read transactions do not block each other.

As mentioned in the previous section, when comparing latencies of the database categories, there is a significant increase in latency for stronger isolation levels when there are many writes. This is likely caused by more write transactions needing to wait for read transactions. In CockroachDB, for example, write skew is avoided by storing the latest read timestamp for keys, and ensuring that write intents have a higher timestamp than this. Write transactions may need to wait longer and push their timestamps when writing to frequently read keys, causing a higher latency. This scenario is avoided by databases with snapshot isolation, as they can allow write skew to occur.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

We have successfully compared the performance of CockroachDB, TiDB and YugabyteDB with the standardized YCSB and TPC-C workloads, although the latter failed to run on YugabyteDB. During our work on this thesis, we have tested over 200 separate clusters using over 1,500 virtual machines in a cloud environment. The volume of tests that we were able to perform without manual intervention speaks to the success of our novel automated testing approach.

For the YCSB workload, the databases had generally comparable performance, but YugabyteDB with serializable isolation was an outlier with worse performance. Thus, for applications requiring serializable isolation, CockroachDB is the best performing option. However, if isolation requirements are more flexible, CockroachDB, TiDB and YugabyteDB with snapshot isolation are all good options. The two snapshot isolation databases outperform CockroachDB in throughput and latency for write-heavy workloads, but have lower throughput for read-heavy workloads.

Out of the databases we have evaluated, YugabyteDB is the newest one, with their SQL interface being officially released two years after CockroachDB and TiDB. Their recent release indicates their implementation may not be as mature as other offerings, which we also noticed during our evaluation. First, YugabyteDB with snapshot isolation struggles with handling the scan workload, having less than ten operations per second and latencies of over 30 seconds. Second, YugabyteDB with either isolation level was not able to run the TPC-C workload without failures. Furthermore, YugabyteDB does not perform well with serializable isolation, as mentioned previously. We hypothesize that this is because the database was originally created with only snapshot isolation, and that serializable isolation was added later, meaning that it might not be as optimized yet. In summary, YugabyteDB's performance with snapshot isolation shows potential, but there is currently no good reason to choose YugabyteDB if serializable isolation is a requirement.

The scalability numbers of all the databases show that the performance per node quickly deteriorates when scaling horizontally with the YCSB workloads. This can largely can be attributed to the overhead of network communications and the cost of transaction conflicts that span between nodes. However, we also hypothesize that these numbers would be better with workloads that are larger and therefore easier to distribute.

Overall, we conclude that CockroachDB and TiDB are currently better choices than YugabyteDB as NewSQL databases. It should be noted that YugabyteDB also supports other protocols which can make it a more attractive offering, but that is out of the scope of this research. Between CockroachDB and TiDB, aside from the isolation differences, they both have strengths and weaknesses. The most important performance difference that can be seen in the benchmarks is that CockroachDB has worse throughput and latency on write workloads, while TiDB has lower throughput but similar latency for read workloads. Thus, the choice between CockroachDB and TiDB will highly depend on both the isolation and workload requirements of an application.

5.2 Future Work

In this section we present some work that we believe are worthwhile efforts to expand on this thesis. First, we would like to expand our benchmarking system to support multiple clouds without needing to rewrite Terraform manifests. This would make it easier to evaluate databases across even more hardware configurations, and would also allow a user to compare the performance of different clouds. Additionally, in a more configurable cloud, one might be able to tune CPU, RAM and disk independently, which will make it easier to analyze the effect each resource has on performance.

As mentioned previously, we hypothesize that some of our performance results may have been lowered because of the structure and size of our workloads. To investigate this further, we would like to try more workloads and increase the size of datasets. This could result in a better evaluation for high performing clusters, making the comparisons fairer across cluster sizes.

Another analysis that we hypothesize can help gain insight into the databases' performance is to analyze the results over time while performing changes to a cluster. For instance, one could add or remove different types of nodes from the cluster and see how this affects performance. One would expect that adding a node will increase performance, but in order to synchronize data to that node, some query capacity must be sacrificed until the node is up to date. Showing these trade-offs visually and comparing how different systems handle scaling during runtime can enable a deeper understanding of these NewSQL systems.

References

[1] Daniel Abadi. “Consistency tradeoffs in modern distributed database system design: CAP is only part of the story”. In: *Computer* 45.2 (2012), pp. 37–42.

[2] Daniel J Abadi. *Demystifying Database Systems, Part 1: An Introduction to Transaction Isolation Levels.* 2019. URL: `https://fauna.com/blog/introduction-to-transaction-isolation-levels` (visited on 12/07/2019).

[3] Daniel J Abadi. *Demystifying Database Systems, Part 2: Correctness Anomalies Under Serializable Isolation.* 2019. URL: `https://fauna.com/blog/demystifying-database-systems-correctness-anomalies-under-serializable-isolation` (visited on 12/07/2019).

[4] Daniel J Abadi. *Demystifying Database Systems, Part 4: Isolation levels vs. Consistency levels.* 2019. URL: `https://fauna.com/blog/demystifying-database-systems-part-4-isolation-levels-vs-consistency-levels` (visited on 12/07/2019).

[5] *Cloud-init: The standard for customising cloud instances.* URL: `https://cloud-init.io/` (visited on 05/16/2020).

[6] *CockroachDB Docs.* URL: `https://www.cockroachlabs.com/docs/stable/` (visited on 11/01/2019).

[7] James C Corbett et al. “Spanner: Google’s globally distributed database”. In: *ACM Transactions on Computer Systems (TOCS)* 31.3 (2013), p. 8.

[8] Djellel Eddine Difallah et al. “Oltp-bench: An extensible testbed for benchmarking relational databases”. In: *Proceedings of the VLDB Endowment* 7.4 (2013), pp. 277–288.

[9] Seth Gilbert and Nancy Lynch. “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services”. In: *Acm Sigact News* 33.2 (2002), pp. 51–59.

[10] Rachael Harding et al. “An evaluation of distributed concurrency control”. In: *Proceedings of the VLDB Endowment* 10.5 (2017), pp. 553–564.

[11] *HashiCorp Packer.* URL: `https://www.packer.io/` (visited on 05/16/2020).

[12] *HashiCorp Terraform.* URL: `https://www.terraform.io/` (visited on 05/16/2020).

[13] Orhan Henrik Hirsch. *Distributed SQL databases for the cloud.* Project report in TDT4506. Department of Computer Science, NTNU – Norwegian University of Science and Technology, Dec. 2019.

[14] Peter Mattis, Ben Darnell, and Spencer Kimball. *Why We're Relicensing CockroachDB.* 2019. URL: `https://www.cockroachlabs.com/blog/oss-relicensing-cockroachdb/` (visited on 12/01/2019).

[15] Diego Ongaro and John Ousterhout. "In search of an understandable consensus algorithm". In: *2014 USENIX Annual Technical Conference (USENIX ATC 14).* 2014, pp. 305–319.

[16] M Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems.* Springer Science & Business Media, 2011.

[17] Andrew Pavlo and Matthew Aslett. "What's really new with NewSQL?" In: *ACM Sigmod Record* 45.2 (2016), pp. 45–55.

[18] Daniel Peng and Frank Dabek. "Large-scale incremental processing using distributed transactions and notifications". In: (2010).

[19] Karthik Ranganathan. *Low Latency Reads in Geo-Distributed SQL with Raft Leader Leases.* 2019. URL: `https://blog.yugabyte.com/low-latency-reads-in-geo-distributed-sql-with-raft-leader-leases/` (visited on 12/07/2019).

[20] Karthik Ranganathan. *Yes We Can! Distributed ACID Transactions with High Performance.* 2018. URL: `https://blog.yugabyte.com/yes-we-can-distributed-acid-transactions-with-high-performance/` (visited on 12/01/2019).

[21] Siddon Tang. *How TiKV Uses "Lease Read" to Guarantee High Performances, Strong Consistency and Linearizability.* 2018. URL: `https://pingcap.com/blog/lease-read/` (visited on 12/07/2019).

[22] Alexander Thomson et al. "Calvin: fast distributed transactions for partitioned database systems". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data.* ACM. 2012, pp. 1–12.

[23] *TiDB Docs.* URL: `https://pingcap.com/docs/stable/` (visited on 11/01/2019).

[24] *TiKV Docs.* URL: `https://tikv.org/docs/` (visited on 11/01/2019).

[25] *Transaction Processing Performance Council Benchmark C.* URL: `http://www.tpc.org/tpcc/` (visited on 05/15/2020).

[26] *YugabyteDB Docs.* URL: `https://docs.yugabyte.com/latest/` (visited on 11/01/2019).