Master's thesis

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

Julie Sydow Mo

Transactions in NewSQL

Master's thesis in Computer Science
Supervisor: Svein Erik Bratsberg
January 2022

NTNU
Norwegian University of
Science and Technology

Julie Sydow Mo

Transactions in NewSQL

Master's thesis in Computer Science
Supervisor: Svein Erik Bratsberg
January 2022

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

NTNU
Norwegian University of
Science and Technology

Abstract

The NoSQL movement has enhanced database systems' ability to scale and thereby respond to the increasing demands towards data storage and database systems. Still, many applications are unable to make use of NoSQL systems since they require stronger guarantees towards both transactional isolation and consistency. NewSQL emerged from this development, aiming to combine the scalability of NoSQL with the traditional guarantees of transactions in relational databases towards isolation and consistency.

The goal of this thesis is to create an understanding of NewSQL systems and how they achieve their promised features. A set of systems have been chosen as the basis for this study, namely Google Spanner, CockroachDB, Calvin, and FaunaDB.

As stated by Abadi [13], the concepts of isolation and consistency are used ambiguously throughout literature and documentation. Another contribution of this thesis is therefore the work of describing, structuring, and mapping existing theories and concepts to create an understanding of the fundamentals of distributed database systems.

Sammendrag

NoSQL systemer har forbedret databasers evne til å skalere og har på denne måten svart på markedets økende krav til datalagring og databasesystemer. Likevel er det mange applikasjoner som ikke tar i bruk NoSQL da de behøver sterkere garantier for transaksjoners isolasjon og konsistens. NewSQL har oppstått fra denne utviklingen, og sikter på å levere skalerbarheten til NoSQL kombinert med de tradisjonelle garantiene fra relasjonsdatabaser mot transaksjoners isolasjon og konsistens.

Målet ved denne oppgaven er å skape en forståelse for NewSQL systemer og hvordan de oppnår sine egenskaper. For å oppnå dette har et sett med systemer blitt valgt som grunnlag for denne studien, nemlig Google Spanner, CockroachDB, Calvin og FaunaDB.

Som bemerket av Abadi [13] er begrepene isolasjon og konsistens brukt tvetydig gjennom litteratur og systemdokumentasjon. Et annet bidrag ved dette arbeidet er derfor å beskrive, strukturere og etablere en forståelse for sammenhengen mellom eksisterende teorier og konsepter. Dette for å oppnå en forståelse for de fundamentale konseptene ved distribuerte databasesystemer.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

In today's technology-driven world the amount of data has exploded. Research conducted in 2013 shows that 90% of all data was created over the last two preceding years [23]. Both the exceeding amount of data and the availability demands for it have changed business requirements towards data storage and database systems. Applications must now handle millions of concurrent requests, from users scattered over the entire globe.

Traditional relational database systems have faced challenges coping with these changing requirements, being unable to scale beyond a single server without sacrificing data consistency [44]. Although these systems scale vertically, the scaling is bounded by cost and technology limitations. The focus has therefore shifted towards distributed systems [43], which are systems that reside and operate on multiple servers.

NoSQL systems emerged from this development, addressing the challenges identified with relational database systems. These systems relaxed the traditional ACID guarantees of transactions by introducing the eventual consistency paradigm [32] as well as implementing a distributed architecture, achieving horizontal scalability.

Even though NoSQL systems have earned broad commercial success, many applications have found themselves unable to make use of NoSQL systems since they cannot give up the strong consistency requirements of traditional systems [44]. NewSQL has therefore emerged, aiming to combine the horizontal scalability trait of NoSQL systems with the strong transactional consistency requirements of traditional RDBMSs.

This thesis aims to identify how NewSQL systems achieve horizontally scalable systems that uphold traditional ACID guarantees for transactions. A set of systems have been chosen as a basis for this research, namely Google Spanner, Cockroach DB, Calvin, and FaunaDB.

1.2 Goals and research question

The goal of this thesis is to provide an extensive understanding of NewSQL systems, diving into four specific systems.

The following research questions are to be addressed:

Research question 1: Which protocols and mechanisms are currently in use to facilitate distributed transactions?

Research question 2: Which protocols and mechanisms are currently in use to facilitate demands for availability in distributed systems?

Research question 3: How do the open-source database systems chosen for this thesis make use of protocols and mechanisms to achieve ACID guarantees in horizontally scalable systems?

1.3 Thesis outline

Chapter 2 describe the underlying theoretical concepts needed to understand the concept of transactions in distributed databases.

Chapter 3 provides a general description of NewSQL systems, as defined by Aslett and Will [15]. Additionally, the chapter addresses four different systems and presents their approach to achieving horizontally scalable systems that provide ACID semantics for transactions.

Chapter 4 seeks to compare the systems against one another as well as provide an overview of the different approaches to achieve the features of a distributed, horizontally scalable ACID-compliant database system.

Chapter 5 reason about the development of NewSQL systems over the past decade, as well as giving a brief evaluation of their entry into the market of database systems.

2 Theoretical background

This chapter describes the underlying theoretical concepts for this thesis, delving into transactions, distributed database systems, and concurrency control. The goal is to refresh the reader and establish a knowledge base for further discussions of novel systems. This chapter is based on previous work from the project thesis [41].

2.1 Distributed database systems

A distributed database is a network of logically interrelated databases, and a distributed database management system manages these databases along with providing a transparent interface to its users [43]. The term *distributed database system* includes both of these concepts, and will be used in further discussion of the topic.

Common for distributed systems is a shared-nothing architecture, meaning that no nodes overlap in hardware resources. Data is partitioned into subsets resident to the different databases, such that each database only handles a subset of the data. These traits simplify failure handling and facilitate scalability. However, the downside with this architecture is the need to transfer data over the network, as data is logically integrated but physically distributed. An important topic of distributed database systems is therefore data locality, since the partitioning of data among the nodes determines the cost and complexity of accessing and maintaining the database.

2.2 Transactions

A transaction is defined as a process, or an executing program, that includes one or more operations towards a database [24]. The concept of transactions is first and foremost an abstraction, meant to reduce the complexity of implementing concurrent programs to application programmers [10].

ACID [24] is a set of properties commonly used to define the desired characteristics of transactions, and hence aid application programmers to reason about the state of the database after performing operations towards it.

Atomicity refers to atomic execution. The atomicity property ensures that a transaction is executed in its entirety, implicating that either all operations are executed or none of them are.

Consistency refers to the effect of the transactions on the state of the database. A

transaction shall take the database from one consistent state to another. A further discussion of consistency is conducted in section 2.4.

Isolation refers to interactions between transactions running concurrently. The isolation property guarantee that concurrent transactions are shielded from one another until they commit. A further discussion of isolation is conducted in section 2.3.

Durability refers to the fact that all transactions performed towards the database should be permanent. Durability guarantees that a committed transaction is durable in the sense that if the system crashes or other irregular states occur - the transaction will remain committed.

2.3 Isolation levels

The purpose of isolation is to shield concurrent transactions from each other so that no operation reads or writes temporary data written by a concurrent transaction. Isolation can be implemented in different levels. Perfect isolation [10], called serializability, is defined as the ability of a system to run concurrent transactions in a way that is equivalent to as if they were run successively. Serializability guarantees that executed transactions will take the system to a state consistent with serial execution, but gives no guarantee on the ordering of which the transactions are executed in.

Perfect isolation comes at a significant performance cost [10, 24]. Lower levels of isolation are therefore common when system designers choose to prioritize performance over isolation guarantees.

These reduced levels of isolation lead to the possibility of concurrency anomalies [10]. Anomalies are events that deviate from a serial order when transactions execute concurrently, making the application prone to errors. A set of commonly occurring anomalies are lost-update, dirty-write, dirty-read, non-repeatable read, phantom read, and write skew [10].

Snapshot isolation [10] is an example of a lower isolation level, which is broadly used in commercial DBMSs. A transaction operates towards a snapshot of the database, which is a particular state of the database only containing committed data. The snapshot remains unaltered until the transaction commits or aborts, so all reads are repeatable. If the write set of concurrent transactions is joint, they can detect conflicts and resolve them, avoiding the lost-update anomaly. However, disjoint write sets are prone to write skew anomalies. Phantom read may also occur, dependent on implementation.

2.4 Consistency levels

Consistency refers to a system's ability to ensure that the system upholds a predefined set of invariants, which are statements about the data that must be true [34]. However, the term is ambiguously used dependent on context. The definition presented with ACID is applicable regarding consistency levels, but a later use of the term is introduced in the discussion of the CAP theorem in section 2.5.

As previously described, isolation levels only guarantee correctness for concurrently running transactions. If a system is to uphold correctness for transactions that are not running concurrently, consistency guarantees are required to specify how non-concurrent transactions are to be processed in relation to each other [12].

Consistency levels have historically been addressed in the context of single operations performed by different threads of execution in a shared-memory multiprocessor system [12], and thus will be the basis for illustrating some consistency levels in this section. The levels of consistency are separated by how and when different executing operations can see writes performed by other operations.

The following consistency levels are presented, ordered from strong to weak consistency:

Strict consistency is defined as perfect consistency and entails that all reads reflect all previous writes in real-time, and that the order of writes in sequential order must equal the real-time execution of operations. Additionally, all execution threads must have a global agreement on time.

Linearizability ensures that all writes are ordered globally, and that the ordering of the writes is in fact the same as the real-time execution order. In practice, this is the highest level of concurrency to be implemented in distributed and/or replicated systems [12].

Sequential consistency guarantees that all writes are ordered globally, such that every tread of execution must see all writes in the global order. This order does not make any guarantees on how the global order is achieved, and may therefore differ from real-time execution.

Causal consistency only guarantees a global ordering of writes for related operations where it assumes a causal relation. E.g. if a value is read before a write of that value is performed, the read is thought to cause the write, thereby establishing a causal relationship between the two operations.

Eventual consistency is the weakest of the presented consistency levels, and only offers a guarantee that, in case of a time period in which no write operations are performed, the execution threads will reach an agreement on the value of the last write.

These concepts are easily transferable to the grouping of operations into transactions given two conditions. Firstly, a transaction can only be handled by one thread of execution, and secondly, a thread of execution can only handle one transaction at a time.

2.5 The CAP theorem

Eric Brewer's CAP theorem states that, in case of failures, a distributed database system only can uphold two out of the three properties: consistency, availability, and partition tolerance [31].

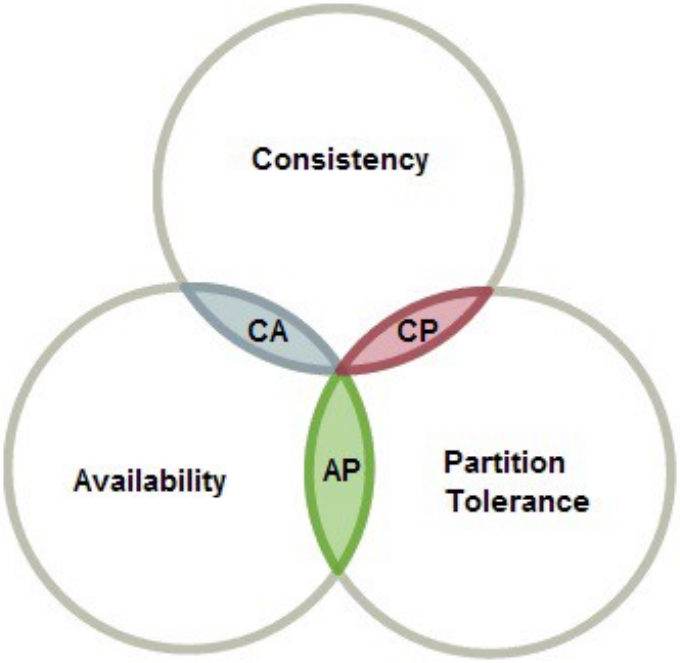


Figure 1: Illustration of the CAP theorem

Source: `https://towardsdatascience.com/cap-theorem-and-distributed-database-management-systems-5c2be977950e`

Consistency

Brewer's definition of consistency, which deviates from the one presented with the traditional ACID properties, states that a total order on all operations must exist such that it appears as if the operations were executed on a single machine.

Availability
Availability refers to the system's ability to handle requests. There should be no state where the system is unavailable, so that all requests will terminate and yield a response for both read and writes. This definition does not declare a time limit on the response time of the system, but only guarantees that a response will be made.

Partition tolerance
Partitioning is an unavoidable scenario of networking, and partition tolerance is, therefore, a desired property. If a partition occurs, no messages will be delivered across the partition and these messages are therefore lost. A partition is always assumed to be temporary.

The consistency requirement and the availability requirement imply that all nodes receiving a request must respond and every response must be consistent. Thus, a fundamental trade-off exists between availability and consistency in the case of a network partition. As web services are expected to be highly available, and therefore tolerant towards faulty nodes in the network, the theorem implicates that a choice must be made on whether to offer availability (AP) or consistency (CP) in the case of network partitions. An AP system will sacrifice consistency for the ability to answer requests, in contrast to a CP system that would become unavailable to uphold its consistency.

2.6 The PACELC theorem

Daniel Abadi's PACELC theorem [7] extends the CAP theorem of Brewer. The PACELC theorem identifies a consistency trade-off during normal operation in addition to the CAP trade-off during failures. This trade-off is due to the availability property. If a system is to be available it implies some degree of replication to make the system adaptable to the possibility of failure, meaning that replication needs to be executed during normal operation. The replication itself constructs a trade-off between consistency and latency depending on the choice of replication method.

Consequently, a system designer must make two fundamental choices regarding consistency. Firstly, the designer must place its system on a scale between strict consistency and high availability for a normal operation setting, thereby choosing between consistency (C) or low latency (L). Secondly, the designer must decide on which abilities are desirable in case of failures in accordance with the traditional CAP theorem.

2.7 Replication

The purpose of replication is to ensure system availability, yield performance gain through reducing response time, and facilitate scalability [43]. A set of decisions must be made regarding the consistency level of the database, where to perform updates, and how to propagate these updates.

Global transactions update copies of a data item resident on different nodes, and the values of these replicated entries might be different at a given point in time. A replicated database is said to be mutually consistent if all replicas of a value are identical [43]. For a strongly consistent replicated DBMS, all of a transaction's writes must be acknowledged by every replica before the transaction is considered committed. More relaxed criteria towards consistency are defined as weak consistency.

Updates can be performed either centralized by a single master copy, or distributed if updates are allowed at any replica. Once updates are performed at any replica, a decision must be made on how to propagate this update to other replicas. These protocols can either be active-active or active-passive [43]. Active-active protocols force updates to be applied to all replicas before a transaction commits. Active-passive protocols on the other hand only update a single node which then distributes the update to other copies after the transaction has committed.

2.8 Isolation in distributed and replicated systems

In a distributed or replicated system, the serializability guarantee must be strengthened in order to avoid the previously mentioned anomalies that occur for lower levels of isolation in non-replicated systems [11]. Two concurrent serializable transactions may take the database to an inconsistent state due to latency in replication. If both transactions update the same data item on different nodes, and the nodes have a latency in propagating the updated data item to each other both transactions will commit, but the updated data item will be inconsistent. Consequently, replication makes the system prone to concurrency bugs despite serially ordered transactions as long as the replication itself is not embedded in the transaction.

Attar, Bernstein, and Goodman [17] investigated correctness in replicated systems and developed the concept of one-copy serializability as an equivalent to serializability for unreplicated systems. One-copy serializability entails that the system behaves as if there is one logical copy of each data item, such that a transaction always interacts with the latest update of the data item in question. A read will therefore

read the most recent write by the closest transaction in serial order, eliminating the problems related to inconsistency caused by replication.

One-copy serializability apparently provides a higher level of isolation than those previously described, yet perfect isolation is defined as serializability. This ambiguity is due to inconsistent use of the term isolation. One-copy serializability is in fact a combination of both guarantees for isolation as well as consistency. Sequential consistency for one-copy serializability is guaranteed when no executing thread can process more than one transaction [12].

2.8.1 Anomalies

Serializability gives no guarantee on which order transactions are to be performed, yielding challenges with the concept of time traveling transactions [11]. These are transactions that arrive at a later time than previously committed transactions, but are executed in a way that makes the database consistent with a serial order in which the latest transaction is performed before previously committed transactions.

Neither serializability for non-replicated systems nor one-copy serializability prevents such time traveling transactions. Non-replicated systems usually offer a time travel guarantee as it is simple to implement this logic on a single-server, making time travel anomalies rare to observe in these systems. Distributed and/or replicated systems are, on the other hand, prone to time travel anomalies [11] such as immortal writes, stale reads, and causal reverse further described in more detail.

Immortal write occurs when a later write time travels to a point in the serial ordered timeline which is earlier than the previous write, causing the later write to be overwritten by the previous one. This anomaly does not violate the serializability guarantee. An immortal write anomaly is rarer to occur if a read is issued before the write rather than a blind write. In multi-master asynchronous replicated systems blind immortal writes can be allowed to achieve conflict resolution in the case of conflicting writes upon synchronization.

Stale read occur when the system time travels a read transaction to a point in the serially ordered timeline earlier than the latest write to the data item in question. This anomaly commonly occurs in asynchronously replicated systems, for example when the read transaction is directed to a different copy than the write before the latest write is propagated to other copies. Stale reads do not violate serializability or even one-copy serializability, as the transaction could be said to time travel to a point in time prior to the write in question.

Causal reverse occurs when a later write caused by an earlier one time travels to a point in the serially ordered timeline prior to the first write. This makes it possible for other transactions to observe the effect without the cause, exposing the application to errors. This anomaly can occur in any distributed system independent of replication methods, and is commonly observed in partitioned database systems.

2.8.2 Correctness guarantees

To achieve a higher level of correctness than one-copy serializability, the system must guarantee that transactions are not allowed to travel in time, i.e. offer a consistency guarantee. This level of correctness is called strict serializability, and meets the one-copy serializability guarantee as well as guaranteeing that if a transaction A commits before transaction B starts, then A appears before B in the serial order of the system. Strict serializability is achieved by systems such as Google Spanner and FaunaDB/Calvin [11].

There exists a variety of systems that have some degree of time travel control, without a global guarantee of strict serializability. One type is **strong session serializability**, which guarantees strict serializability for transactions within a session, but falls back to one-copy serializability for other operations. **Strong write serializability** is another example, where updates or insertions are subject to strict serializability, while read-only transactions are guaranteed one-copy serializability. A third example is **strong partition serializability**, which guarantees strict serializability for transactions that access data within a partition, but otherwise offers a one-copy guarantee. Table 1 summarizes which anomalies that the discussed guarantees are prone to. None of the guarantees are prone to dirty read, non-repeatable read or phantom read.

Guarantee	Write skew	Immortal write	Stale read	Causal reverse
Snapshot isolation	Possible	Possible	Possible	Possible
One copy serializable	Not possible	Possible	Possible	Possible
Strong session serializable	Not possible	Possible	Possible	Possible
Strong write serializable	Not possible	Not possible	Possible	Not possible
Strong partition serializable	Not possible	Not possible	Not possible	Possible
Strict serializable	Not possible	Not possible	Not possible	Not possible

Table 1: Isolation guarantees and anomalies

Source: [11]

2.9 Distributed transactions and consensus

A distributed transaction [22] is a transaction that interferes with data objects distributed over multiple servers. Remembering ACID and the atomicity property of transactions, either all operations of a transaction are to be performed or none of them are. Atomic commit protocols facilitate this property by achieving a joint agreement on whether to commit or abort a transaction across the distributed network [37]. Two phase commit is a broadly used atomic commitment protocol, and will be described in further detail.

Consensus algorithms are also in use to facilitate distributed transactions, but their main role is to achieve fault tolerance through replication [37]. Paxos and Raft are two popular approaches to manage a replicated log across servers [42], and a further description of Raft is conducted.

2.9.1 Two-phase commit

The two-phase commit protocol [22] enables the network of servers to vote on whether to commit or abort a transaction. A node is appointed to the role of coordinator for the distributed transaction. When the coordinator receives a commit from the client, it asks the nodes if they can commit and they respond accordingly.

If all agree to commit, the coordinator sends a "do commit" message (or "do abort" if the answers are not uniform or complete), and the nodes respond with a "have committed" message when they commit. Only until all nodes have responded with "have committed", the coordinator and nodes can safely delete information about the transaction.

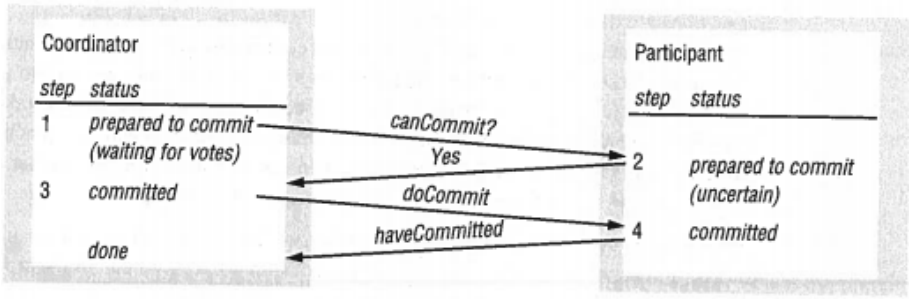


Figure 2: Communication in a two-phase commit protocol

Source: Distributed systems: concepts and design [22]

The problem with 2PC is to ensure that all nodes vote and that they receive all messages sent. Timeouts and resending of messages are used to ensure continuity in the stages of the protocol where the involved nodes are stalled due to waiting for messages. If a node goes down temporarily, or it reaches time out due to a lost message, a node can send a "get decision" request to the coordinator. In this way, the protocol has mechanisms to cope with lost messages and crashing nodes, but it does not cope well with a scenario where the coordinator goes down.

2.9.2 Raft

Raft was developed as an alternative to Paxos, trying to achieve a more understandable consensus algorithm with a better foundation for practical implementations [42]. The purpose of Raft is for a group of replicated servers to agree on both content and order of a log. This log may contain anything, from metadata to operations and transactions.

A Raft group consists of an odd number of several replicated nodes, where nodes can operate as one of three roles: leader, follower, or candidate. During normal operation, one node is chosen as leader and the rest will act as followers. Figure 3 depicts how servers transitions roles in a state schema.

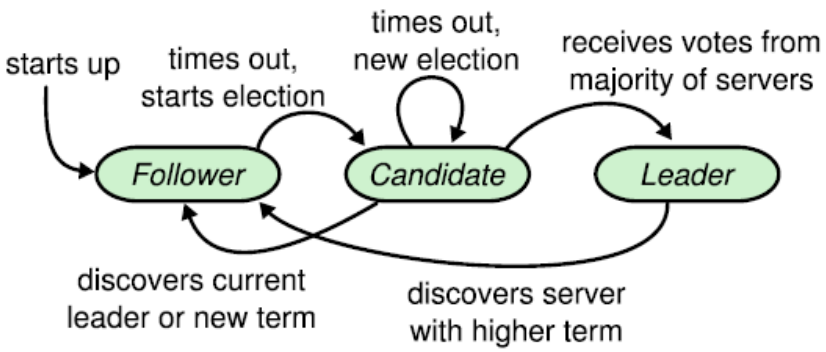


Figure 3: State schema of role transitions in a Raft group

Source: Ongaro and Ousterhout [42]

Time is abstracted into terms, where each term is initiated by a leader election. Each log entry is numbered sequentially by an index and contains the term in which it was issued. Both entry number and term annotation stay constant over time and across logs.

The leader has full responsibility for managing the replicated log, by receiving log entries from clients and propagating these to its followers. It also keeps track of which index each follower is to receive next. The leader issues commit when a majority of followers have received the log entry, by appending its index to all messages sent to followers. The followers will then commit log entries with an index up to the index given by the leader. The leader maintains authority by sending periodical heartbeats to its followers.

Elections are initiated when the current leader fails and is conducted through voting, where followers announce themselves as candidates and propagate their index. A leader will be chosen only if it has an index larger or equal to those who vote for it. This new leader is guaranteed to have committed all previously committed entries from the foregoing term, ensured by the voting process.

In a scenario where a follower is not up to date and it receives a newer log entry than what it currently expects, the follower rejects the update and the leader decrements its index of the follower. The leader then retries with the log entry corresponding to the decremented index. This process is repeated until the leader index is up to date with the follower. Then, the index will increase as the follower receives updates in the right order and get its log up to date with the state of the leader's log. These retries are harmless, since the follower will ignore log entries that are already present in its local log.

In case of follower failure, the described mechanism with retries will take place.

Timeouts are used to handle the failures of candidates during elections. In doing so, Raft guarantees durability for committed transactions as well as eventual consensus for all replicas. Furthermore, Raft guarantees availability and correctness during changes in cluster composition [42].

2.10 Distributed concurrency control

Concurrency control algorithms enforce isolation levels by coordinating the execution of multiple, concurrent transactions to uphold a consistent state of the database [43]. For distributed systems, concurrency control also entails mutual consistency, meaning that copies of a value must converge towards a common, global value for all copies.

The choice of a distributed concurrency control method is dependent on the level of contention and thereby the need for conflict resolution. Some algorithms are better at conflict resolution than others, at the cost of overall performance for transactions.

A selection of different concurrency control algorithms is presented.

Two-phase locking

Two-phase locking (2PL) was the first serializable concurrency protocol [26]. The protocol is divided into two phases, a growing phase and a shrinking phase. All locks needed to execute the transaction must be acquired during the growing phase. After the first lock is released, new locks can no longer be acquired.

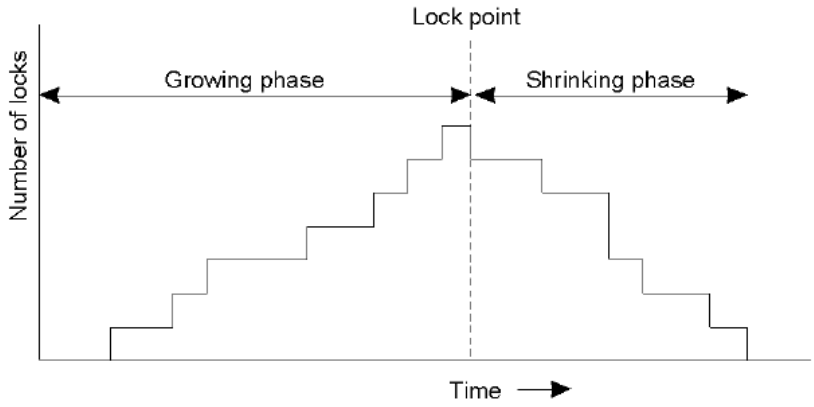


Figure 4: Illustration of lock acquisitions in two-phase locking

Source: `https://medium.com/nixis-institute/concurrency-in-database-management-system-63f9a31752ec`

Locks are either shared or exclusive. Reads are allowed to share locks, but a write needs an exclusive lock to proceed. If a write is not able to acquire the desired lock, the transaction can either wait for the lock to become available or abort and later retry the transaction as a whole. Waiting transactions may cause deadlocks due to cyclic dependencies, creating the need for a deadlock detection algorithm. One approach is to limit the wait to a predetermined time period before it tries again, aborting if this attempt proves to be unsuccessful.

Timestamp and MVCC

Timestamp protocols assign unique timestamps to transactions. A transaction can only read a record that have an equal or lower timestamp than itself. Timestamp-based concurrency control methods select a serialization order through the allocation of timestamps and then execute transactions accordingly [43].

Multi-version concurrency control (MVCC) [19] protocols extends general timestamp protocols. In MVCC, several copies of each record are kept to enable concurrent reads and writes, by allowing reads to access older copies if a write is currently under execution. Thus, transactions can access values at different isolation levels.

Optimistic

Optimistic protocols [35] are optimistic in the sense that they execute transactions concurrently without any prior coordination. The execution of transactions is performed locally, and upon commit, the transactions are validated against committed transactions and other concurrent transactions awaiting validation. Conflicting transactions will be aborted, and those that still prove to be valid after the validation phase are propagated to the database. These protocols impose overhead, both through validation and the execution of later aborted transactions.

Deterministic

Deterministic ordering of transactions is a fairly new field, proposed as an alternative to traditional concurrency control methods [45]. These protocols have a centralized coordinator which predetermines the execution order of transactions to avoid coordination between servers. This yields two advantages. Firstly, deterministic protocols have no need of atomic commit protocols. Secondly, they ease the complexity of replication strategies.

3 State-of-the-art

This section describes the concept of NewSQL systems as defined by Aslett and Will [15], as well as the chosen systems Google Spanner, CockroachDB, Calvin and FaunaDB. The purpose of this section is to provide a thorough introduction to the systems to be able to reason about their approach to achieving highly available, horizontally scalable systems that provide ACID semantics for transactions. This section is partly based on previous work in the project thesis [41].

3.1 NewSQL

NewSQL is a fairly new development in the class of database management systems, with the term first specified by Matthew Aslett in 2011 [15]. The emergence of NewSQL was motivated by the development of NoSQL systems and their prominent scalability strengths. However, NewSQL seeks to address some of the issues that have proven to be decisive for businesses in the choice of taking NoSQL systems into use.

NoSQLs advantage lies is in its sharding middleware layer, distributing data over several nodes to achieve horizontal scalability. However, this trait comes at a cost since NoSQL systems sacrifice stronger consistency and isolation guarantees to achieve it. Along with this, NoSQL has also simplified the concept of transactions, often only amounting to CRUD-operations [22]. Google, a company that adopted NoSQL early, experienced an increasing amount of work for application developers to handle data inconsistency and transactional logic and reasoned their shift back towards transactional support because of this [44].

There are several definitions of the concept of NewSQL systems. One is that NewSQL systems are an implementation with a lock-free concurrency control scheme and a shared-nothing distributed architecture [46]. A broader definition stems from Pavlo and Aslett [44], which defines them as systems that maintain ACID guarantees for transactions as well as providing the scalability property of NoSQL for OLTP read-write workloads.

To conclude, NewSQL systems are characterized by a relational data model and standard SQL implementation, ACID transactions, scalability facilitated by data partitioning in a shared-nothing architecture, and availability through data replication [43]. These systems targets OLTP workloads for enterprise information systems, where transactions are characterised as [44, 33, 43]:

- short-lived,
- only involving a small subset of data using index lookups,
- repetitive with varying input.

Three main categories of NewSQL systems have been identified by Pavlo and Aslett [44]. The first category being novel systems that are based on a new architecture, built ground-up. The second category covers the systems that use a transparent sharding middleware as of the sharding infrastructure that was developed with the NoSQL movement. The third category is the systems that are based on cloud computing services, styled as database-as-a-service (DBaaS) systems.

3.1.1 Common features of NewSQL DBMSs

This section describes some common features of NewSQL DBMSs and points to innovations or challenges they face compared to traditional relational DBMSs or NoSQL systems.

Partitioning or sharding

Most distributed NewSQL DBMSs scale horizontally by partitioning the database into shards or partitions, which are disjoint subsets of the database. Each table of the database is split into multiple fragments based on one or more column values, or other partitioning attributes. Then, related fragments are grouped together and stored at a single node, making this node able to execute any query that accesses the data stored in its partition. The purpose of this is to be able to "send the query to the data", giving nodes the responsibility of executing a transaction self handedly, or to be able to perform a part of a distributed query which is later combined with other partial results to form a final answer in a single node. The advantage of this is reduced network traffic through function shipping, as opposed to sending data to a centralized node for execution.

Furthermore, partitioning facilitates exploiting a common trait of relations between data of a typical OLTP application database, storing related data in the same node. The database schema of such interconnected data can be rearranged to a tree-like structure containing all data related to the root. An example could be a CRM system where all data connected to a specific customer is stored in the same tree and thus in the same node. The goal of this is to minimize transactions that touch upon

data split over different nodes. If a transaction only access data at a single partition there is no uncertainty as to if the transaction is executed correctly across different nodes, which again makes an atomic commitment protocol redundant. Keeping in mind the characteristics of OLTP workloads, this trait might lead to a significant reduction in network traffic.

Another aspect of distributed databases is migration. Migration is the process of moving resources to balance the load on the database without affecting its availability or interrupting ongoing transactions [43]. For NewSQL systems, the process of live migration is a more challenging one as they must uphold ACID guarantees for transactions during migration [25]. Pavlo and Aslett [44] identify two approaches to live migration, the first through the use of virtual partitions, moving virtual partitions across nodes. The second approach is to re-balance by moving tuples through range or hash partitioning.

Concurrency control

As previously described, concurrency control enforces isolation guarantees in a system. The choice of a concurrency control scheme is therefore an important one, along with the choice of having a centralized or decentralized transaction coordination protocol. In general, NewSQL systems based on novel architecture avoid the strict serializable concurrency protocol of two-phase locking, and instead opt for a timestamp ordering protocol. Most of these systems have chosen a decentralized MVCC scheme [15], which allows for read-only transactions not to block writes. Some systems actually do use two-phase locking due to their underlying architecture, but combine it with MVCC. This combination enforces locks upon writes, but allow read-only queries to be non-blocking on writes.

Replication

Replication ensures high availability and durability for databases, and there are two main design decisions to make regarding when to propagate updates to replicas and how to perform this propagation.

The first decision regards how to enforce data consistency between nodes, which can either be strong or weak. Strong consistency entails that a transaction must be propagated to all replicas before the transaction is said to be committed. There are several advantages with strongly consistent replication, but the main disadvantage is that it needs an atomic commitment protocol to ensure consensus between replicas,

which in turn impose overhead.

Secondly, there is the decision of how to perform the replication. There are two ways of doing this, either active-active or active-passive replication. With active-active replication, each replicated node executes the same request in parallel. Active-passive replication does on the other hand process a request at a single node before its result is propagated to other replicas.

All NewSQL systems provide strong data consistency, and most except those that are deterministic use active-passive replication. This is because the use of non-deterministic concurrency control schemes yields the possibility of differently ordered transactions across replicas. Thus, one cannot send queries to each replica to be executed in parallel as it might lead to a divergence in the state of the database.

3.2 Spanner

Spanner was first launched by Google in 2013, as a scalable, multi-version, distributed, and synchronously replicated database [21]. It started as a NoSQL key-value store but later shifted towards a relational model. The described prime motivation behind the shift is that they found it difficult to build OLTP applications on top of a key-value store lacking a strong schema, a powerful query language, transactional support, and consistency in replication [18]. Spanner is now developed into a sharded, geo-replicated, relational NewSQL system.

Spanner facilitates concurrent distributed transactions through a MVCC protocol, assigning timestamps to all transactions. Along with this, Spanner offers what they define as external consistency [18]. External consistency is a further restricted version of strict serializability in terms of real-time ordering and is described as:

> If a transaction A commits before B commits, then A will appear before B in the serial order of the system.

This is opposed to strict serializability where one only guarantee the serial order property if A commits before B starts. The external consistency guarantee is supported by Spanners novel TrueTime API [21]. TrueTime is a distributed clock that enables unique, monotonically increasing timestamps - a guarantee that holds across all servers and timestamps. This enables consistent reads across the entire database, without blocking writes [3].

Distribution and replication is achieved through a synchronous, Paxos-based replication scheme. This scheme is through Spanner's documentation said to facilitate data availability, geographic locality, single database experience, and to ease application development [2].

To achieve a deeper understanding of how Spanner achieves these properties, a description of its implementation and data model, the TrueTime API, and how it enforces concurrency control will be conducted, based on the original Spanner paper of 2012 [21].

3.2.1 Implementation and data model

Implementation

A Spanner deployment is described as a universe, which again is organized in a set of zones. These zones are the unit of administrative deployment, as well as the unit for physical isolation. The number of zones in a universe is dynamic, meaning that zones can be added or removed from a running system. Zones are resident in data centers, and a data center might contain one or more zones.

Each zone has a zonemaster and several spanservers. The zonemaster is responsible for allocating data to spanservers, while the spanservers are responsible for serving data to clients. Location proxies facilitate clients in locating the spanserver responsible for their data. Furthermore, the zones in a deployment are the set of locations across which data can be replicated.

The placement driver handles data movement across zones, communicating with spanservers to perform load balancing or replication tasks. The universe masters purpose is mainly debugging, as it keeps status information about zones in a universe. Figure 5 depicts the described organization of a Spanner server.

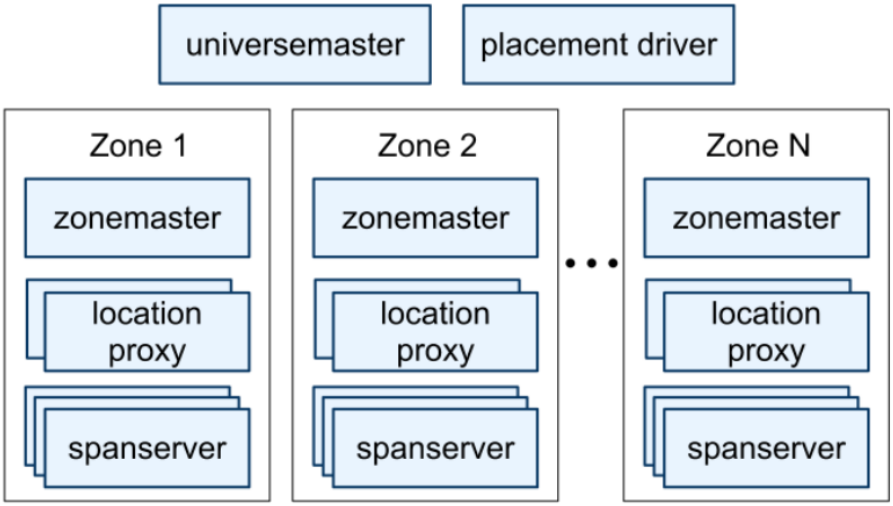


Figure 5: Spanner server organization

Source: Spanner: Google's Globally-Distributed Database [21]

3.2.2 Replication

The software stack of a spanserver is illustrated in Figure 6. Each spanserver is responsible for several instances of data structures called tablets. These tablets are the unit for replication. Each tablet implements a bag of mappings between a key

and a timestamp, forming the basis of the multi-version database. A tablet's state is kept in a distributed filesystem called Colossus, stored in a B-tree like file together with a write-ahead redo log.

To facilitate replication, each spanserver implements a Paxos state machine on top of each tablet as depicted in Figure 6. Thus, Spanner uses a consensus algorithm to achieve agreement on the contents of each log entry across replicas.

A set of replicas is named a Paxos group, which is a replicated state machine. Spanner's implementation of Paxos supports long-lived leaders with time-based leader leases, and at these leaders, each spanserver implements a lock table for concurrency control purposes. Writes must invoke the Paxos protocol at a leader to advance, while reads access directly to a tablet that is sufficiently up-to-date, further described in section 3.2.5. In this way, the Paxos state machines realize a consistently replicated bag of mappings.

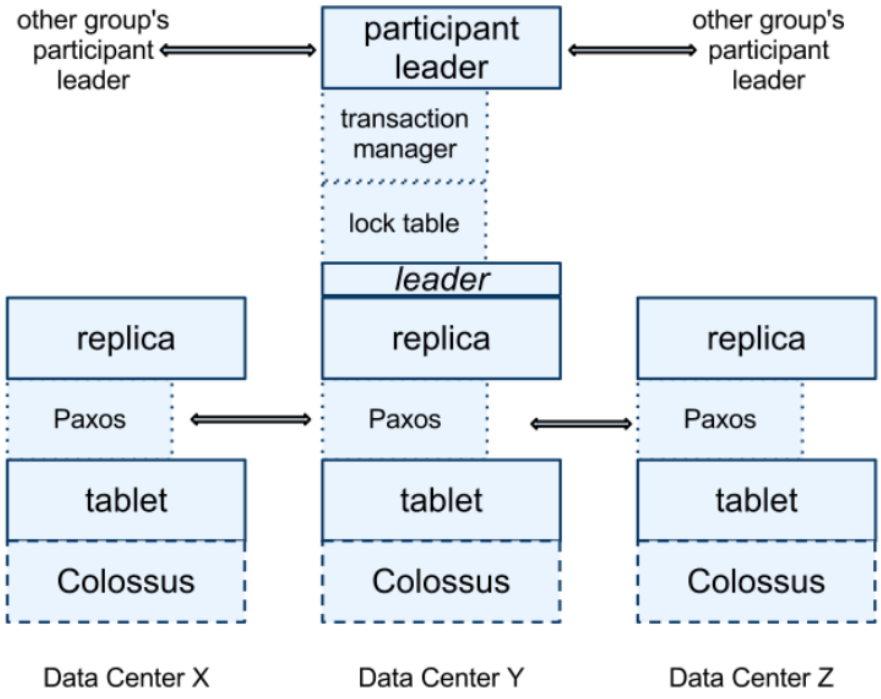


Figure 6: The spanserver software stack

Source: Spanner: Google's Globally-Distributed Database [21]

To handle distributed transactions, in this sense across Paxos groups, each leader replica of a Paxos group appoints a transaction manager at each spanserver. The transaction manager functions as a participant leader from its group, while the others operate as slaves. A coordinator is chosen among the participant leaders of a distributed transaction to perform two-phase commit, achieving consensus for the distributed transaction. Figure 6 illustrates such a scenario. The state of each

transaction manager is stored at the underlying Paxos group, and thereby replicated to the participating slaves. Transactions only involving a single Paxos group will omit the transaction manager, as the lock table along with the Paxos protocol is enough to enforce the desired transactional properties.

3.2.3 Directories, data model, and placement

Spanner's features are as mentioned a data model based on schematized semi-relational tables, a query language comparable to SQL, and general purpose transactions that are further described in section 3.2.5.

Directories are a bucketing abstraction supported by Spanner, containing contiguous keys that share a prefix. Data is partitioned into such directories, and a Spanner tablet might contain several directories. This entails that a tablet, which is the unit for replication, might encapsulate multiple partitions of the row space and not necessarily a contiguous partition, making it possible to co-locate data that does not have a contiguous key range.

The application data model is layered on top of the directories, which again are made up of bucketed key-value mappings. An application may create one or several databases in a universe. These databases are made up of schematized tables, similar to tables of relational databases except for their versioned values.

The data model of Spanner is not entirely relational, since it demands that every table have an ordered set of one or more primary-key columns. These primary-key columns are mapped to the non-primary-key columns in each table, comparable to a key-value store. The purpose of this structure is for the application to be able to control data locality in directories through the allocation of keys.

Databases must be partitioned into hierarchies of tables by the client. The primary table in the hierarchy is defined as a directory table. A directory is formed for each key in a directory table, along with all of the rows in descendant tables that start with the same key in lexicographic order. This interleaving of tables to form directories allows clients to describe the locality relationships that exist between multiple tables - a trait designed to increase the performance in a sharded, distributed database.

3.2.4 TrueTime

As previously mentioned, TrueTime is a time API that enables externally consistent transactions across servers and timestamps. This is achieved through the use of

multiple clock references that are based both on GPS and atomic clocks. The TrueTime API's methods are listed in table 2.

The API uncovers clock uncertainty. TrueTime states time as an interval (TTinterval) made up of two endpoints (TTstamp). This notion represents the uncertainty associated with a time unit. For instance, a call to TT.now() will return a TTinterval which surely encloses the actual time of which the TT.now() was invoked. The last two calls, namely TT.after and TT.before, are convenience wrappers around the TT.now call.

Method	Returns
TT.now()	TTinterval: [earliest, latest].
TT.after(t)	True if t has definitely passed.
TT.before(t)	True if t has definitely not arrived.

Table 2: The TrueTime API
Source: Spanner: Google's Globally-Distributed Database [21]

3.2.5 Concurrency control

Spanner offer externally consistent transactions, lock-free read-only transactions, and non-blocking reads in the past [21]. In doing so, Spanner uses a combination of two-phase locking and MVCC, combined with the API for clock synchronization [44]. Specifically, there are four types of transactions offered, namely read-write, read-only, and snapshot reads with either client-provided timestamp or bound [21].

Read-write transactions use locking, a decision that deviates from the traditional concurrency control methods of distributed systems since it entails overhead. Corbett et al [21] argue that the expense of two-phase commit, despite its performance or availability problems, is not that high as one might expect. They argue that it is better to have application programmers deal with performance problems due to overuse of transactions when it presents itself as a problem, rather than the need to constantly work around the lack of transactions. Along with this, Corbett also states that running two-phase commit over Paxos mitigates the availability problems.

Read-only transactions execute at a system-chosen timestamp without locking, thus not blocking incoming writes. Snapshot reads are reads in the past that execute without locking, either by a client-specified timestamp or a staleness limit. Both read-only transactions and snapshot reads can be processed at any replica that is sufficiently up to date.

Transactions in Spanner depend on several invariants, which are statements about the system's state that must be upheld to guarantee its correctness. These are listed in Table 3.

Invariant	Description
Disjointness	For each Paxos group, each Paxos leader's lease interval is disjoint from every other leader's.
Monotonicity	Within each Paxos group, Spanner assigns timestamps to Paxos writes in monotonically increasing order, even across leaders.
External consistency	If the start of a transaction T2 occurs after the commit of a transaction T1, then the commit timestamp of T2 must be greater than the commit timestamp of T1.

Table 3: Spanner: Invariants facilitating transactions
Source: Spanner: Google's Globally-Distributed Database [21]

The external consistency invariant is dependent on two rules, which must be upheld by the protocol for executing transactions and assigning timestamps. These two rules are the following:

> **Start:** The coordinator leader for a write assigns a commit timestamp no less than the time of the arrival event of the commit request at the coordinator leader.
> **Commit wait:** Ensures that the timestamp of a transaction is less than the absolute commit time of the transaction, thereby ensuring that clients cannot see any data committed by the transaction until TT.after is true.

Read-write transactions

Spanner enforces a pessimistic concurrency control scheme for read-write transactions through the use of two-phase locking. Reads in a read-write transaction are issued to the leader replica of the appropriate Paxos group, obtained through the location proxies. The leader executes read locks before serving the latest read of the data in question. Through this scheme, deadlocks are possible, but Spanner uses a wound and wait scheme to avoid it.

The writes in a transaction are buffered at the client side until commit, thus the reads in the transaction cannot see the effect of the transaction's writes. A write will be assigned a timestamp after all locks have been acquired and before any of them

are released. After all reads are performed and the client has buffered its writes, the client initiates the two-phase commit protocol by choosing a coordinator group and sending a commit message to each participant's leader. The purpose of making the client coordinator for the protocol is to avoid sending data twice.

A non-coordinator-participant leader acquires write locks prior to choosing a prepare timestamp. This timestamp must respect the monotonicity invariant, hence be larger than any timestamp assigned to previous transactions. The leader then logs a prepare record through Paxos, and notify its coordinator of the chosen prepare timestamp.

A coordinator-participant leader chooses a timestamp only after it has received prepare timestamps from its slaves. This timestamp must respect the invariants, and by doing so it must be greater or equal to all prepare timestamps, greater than the time at which the coordinator received its commit message, and greater than any previously assigned timestamps to other transactions. Subsequently, the coordinator logs a commit through Paxos and waits to comply with the commit wait rule. After the given commit timestamp is guaranteed to be in the past, the coordinator sends the commit timestamp to the client and all participating leaders. The leaders then log the transaction's outcome through Paxos. If it is not aborted, all participants apply the commit at the same timestamp before releasing locks.

Read-only transactions

A read-only transaction executes in two phases. First, it is assigned a timestamp, before the transactions reads are executed as snapshot reads at the given timestamp. As mentioned, a read-only transaction can be handled at any replica that is sufficiently up to date. This trait is guaranteed by the monotonicity invariant. Each replica has a value called safe time, which is the maximum timestamp at which the replica is up to date. Hence, a replica can serve a request if the requested timestamp is equal to or less than its safe time.

Assigning a timestamp to a read requires a negotiation phase between all Paxos groups that are involved in the reads. This is facilitated by a scope expression for the transaction, which is a list of the keys that will be read by the transaction. A scope expression is needed for every read-only transaction to facilitate this negotiation between Paxos groups.

If the scope of the reads only touches upon a single Paxos group, then the client issues the read transaction to the group leader which again assigns a timestamp for

the read and executes it. This timestamp can trivially be set to the timestamp of the most recent committed write in the Paxos group if there are no pending writes. If there are read-write transactions not yet committed, the prepare timestamps of the commit protocol ensure that all participants know a lower bound on a prepared, but uncommitted, transaction's timestamp.

The choice of a timestamp becomes a more complicated matter if the scope involves several Paxos groups. Using the most recent, as one would for a single Paxos group scope, will preserve external consistency, but it might lead to blocking if the safe time for all participating Paxos groups has not advanced sufficiently to serve the reads. Therefore, Spanner uses the oldest timestamp that preserves external consistency to reduce the chances of blocking. This requires a round of communication between the group leaders to agree upon a timestamp based on their own most recent committed writes.

3.3 Cockroach DB

Cockroach DB is an open-source NewSQL database system released in 2017. It is a geo-replicated distributed system that scales horizontally and supports SQL semantics as well as ACID transactions, targeting OLTP workloads [1]. The system's design is highly based on Google Spanner, but deviates from it with its implementation of logical clocks. Instead of the TrueTime API, Cockroach makes use of Hybrid Logical Clocks with additional logic to handle cases of inconsistencies due to temporary unsynchronized clocks [47].

As the system is influenced by Spanner, it is based on a transactional and consistent key-value store [1]. Replication is conducted through the Raft consensus algorithm, which guarantees consistency between replicas [47].

Cockroach supports serializable transaction isolation through the use of the Raft consensus algorithm for writes together with a time-based synchronization approach for reads facilitated by a MVCC protocol [1]. Additionally, Cockroach offers support for follower reads, enabling Raft follower nodes to serve client with reads, at the expense of the possibility for stale reads.

The following description of CockroachDB's implementation, transaction model, concurrency control, and how clock synchronization affect isolation and consistency is conducted based on the paper of Taft et. al. [47].

3.3.1 Implementation

A Cockroach deployment is called a cluster. The cluster acts as a single logical application. It can consist of any number of nodes, both resident on the same data center or geographically distributed across data centers. CockroachDB uses a shared-nothing architecture, where each node performs both data storage and computation.

Each node has a layered architecture split into five abstractions, namely the SQL layer, the transactional key-value layer, the distribution layer, the replication layer, and lastly the storage layer. An illustration of a CockroachDB architectural diagram is presented in Figure 7.

The SQL layer is the layer that handles all interactions between users and the database, along with performing query processing converting SQL statements to read/write requests for the underlying key-value store.

The second layer is the transactional key-value layer, which enforces atomicity for changes spanning multiple key-value pairs, as well as facilitate isolation guarantees.

The third layer is a distribution layer, presenting the replicated key value space as a monolithic logical entity ordered by key. In this abstraction, all data is addressable whether it is metadata or user data. The layer is also responsible for routing subsets of queries to the applicable partition.

The fourth layer is a replication layer, and thus enforces the durability property. As mentioned, replication is conducted through the Raft protocol, further described in section 3.3.2.

Lastly, there is the storage layer. Cockroach uses RocksDB [4], which represents a key value store providing writes and range scans to enable SQL execution.

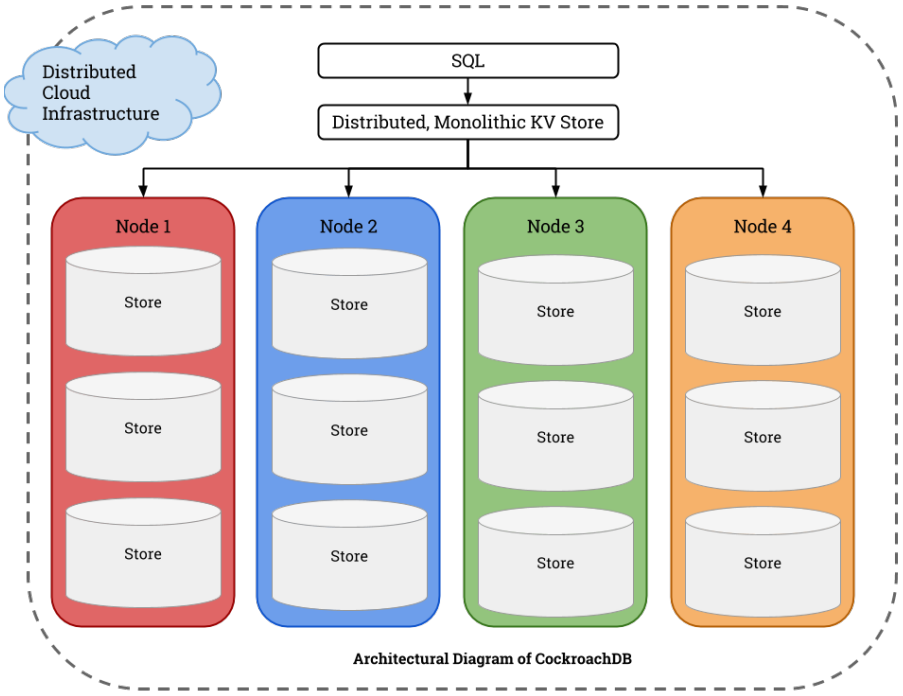


Figure 7: The CockroachDB architectural diagram
Source: `https://thenewstack.io/cockroachdb-unkillable-distributed-sql-database/`

Data model and placement

Keys are range-partitioned to split the data into contiguous ordered chunks called Ranges. These Ranges are stored across the nodes of the cluster, and ordered by the use of a two-level indexing structure maintained by a set of system Ranges. Ranges are load balanced across nodes to reduce hot spots and imbalances in computation

usage.

The data model of Cockroach is relational, as it implements an abstraction of schematized relational tables on top of the underlying KV-store. The SQL layer provides a standard SQL implementation.

Cockroach supports different modes for data placement policies, allowing users to determine policy based on their own workload as well as making trade-offs between performance and fault tolerance. For instance, geo-partitioned replicas allow the tables to be partitioned by access location with Ranges pinned to a specific region. Similar decisions can be made for leaseholders and indexes.

3.3.2 Replication

Each Range is replicated three times and stored on separate nodes. The underlying protocol to achieve consistent replication among these nodes is the Raft consensus algorithm. The replicas of a Range form a Raft group, operating either as leader or follower. The subject for replication is a sequence of low-level edits to be made towards the storage engine, which is represented through a command. Each Raft group member individually performs these commands towards its storage engine in line with the group leader's issued commit messages, as described in section 2.9.2.

In addition to the presented standard implementation of Raft, Cockroach introduces a leaseholder. The leaseholder is the only node that can propose writes to the Raft leader. Moreover, the leaseholder is the only replica allowed to serve consistent reads. In doing so, the leaseholder is able to bypass the Raft protocol entirely, since it knows that it has proposed all writes, as well as that all writes committed locally also have achieved consensus in the group through the protocol. Therefore, it is safe for a leaseholder to serve a read that is committed locally.

Leases for table data are assigned based on epochs. An epoch is dependent on the liveness of the node holding the lease, and the node communicates its liveness through regularly updating a record in a system Range. If a replica detects that the leaseholder is disconnected, a new epoch is initiated where replicas attempt to acquire the lease. To ensure that only one replica holds the lease, lease acquisitions exploit the Raft protocol through a lease acquisition log entry. This initiates a Raft election, and a new leader is chosen from the healthy nodes. Ideally, the responsibility of being Raft leader and leaseholder should be appointed to the same node to minimize write latency. Each lease renewal or leadership transfer will collocate the roles through the election protocol, so divergence is rare and self-corrected.

Ensuring disjoint leases is the foundation for isolation guarantees in Cockroach, further described in section 3.3.6. Meta and system Ranges on the other hand use expiration-based leases to avoid cyclic dependencies since system ranges are required to know if a leaseholder is in fact live or not.

Cockroach treats all membership changes in the cluster in the same way by initiating a load balancing process, where the load is re-balanced across the live nodes. For short-term failures, Raft operates as usual as long as a majority of the nodes are still alive. A leader election is initiated if necessary, and nodes that rejoin the cluster after a temporary failure catch up through the retry mechanism of Raft previously described in section 2.9.2. For permanent failures, Cockroach generates new replicas of under-replicated Ranges on the nodes unaffected by the failure. The liveness of the remaining nodes is established through a peer-to-peer gossip protocol.

3.3.3 Clock synchronization through HLC's

Cockroach defends their choice of not using specialized hardware for clock synchronization on the basis of being able to run on off-the-shelf servers in public or private clouds, needing only software-based clock synchronization services.

Opposed to Spanner, Cockroach depends on the hybrid-logical clock (HLC) scheme. Each node within a cluster maintains a HLC, which provides timestamps. These timestamps are a combination of physical and logical time, composed of the node's own system clock as well as Lamport clocks [36].

A maximum clock offset is defined for HLC's within each deployment, restricting the allowable offset between a HLC's physical time component compared to that of the other HLC's in the cluster. Furthermore, HLC's provide a set of properties listed in Table 4.

3.3.4 Concurrency control

CockroachDB uses a variation of MVCC to provide serializable isolation. Transactions are allowed to span the entire key-space, touching upon multiple nodes while Cockroach still provides ACID guarantees for transactions.

The given isolation level is defined as single key linearizability, which is equivalent to one-copy serializability as described in section 2.8. The single key linearizability property is described as:

Property	Description
Causality tracking	Causality tracking is provided through the logical component in each exchange between nodes. The nodes attach timestamps to each message they send, and use incoming timestamps to update their local clock.
Strict monotonicity	HLCs provide strict monotonicity for timestamps within and across restarts on a single node, through waiting out the maximum offset before starting up and serving any requests. This property ensure that two causally dependent transactions from the same node are given timestamps that reflect their real time ordering.
Self-stabilization	Given sufficient intra-cluster communication, HLC's are able to provide self-stabilization during short-term clock skew fluctuations as the nodes forwards their HLC upon receiving a network message. Thus, HLCs across nodes converge despite diverging physical clocks. This is not a strong guarantee, but it can flatten the curve of synchronization errors in practice.

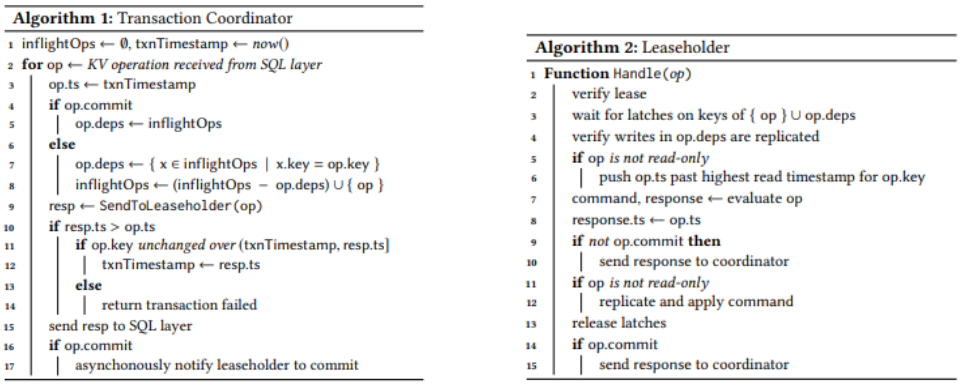
Table 4: The properties of hybrid-logical clocks
Source: CockroachDB: The Resilient Geo-Distributed SQL Database [47]

> Every operation on a given key appears to take place atomically and in some total linear order consistent with the real-time ordering of those operations.

However, regarding isolation and concurrency in distributed systems, one must look to the correctness guarantees that the system offer. Cockroach does not guarantee strict serializability as there is no guarantee that the ordering of transactions touching upon disjoint key sets will match their ordering in real-time. This makes the system prone to causal reverse anomalies, where a transaction caused by another time travels to a point in time prior to its cause.

Under normal operation, Cockroach falls under the category of strong partition serializability, as the possibility of causal reverse anomalies is present [38]. However, clock skew may cause stale reads, which is a violation towards strong partition serializability. Hence, Cockroach places itself somewhere between serializable and strict serializable. This is explained in further detail throughout this chapter.

To understand how Cockroach achieves ACID guarantees, isolation, and perform concurrency control, a further description of its transaction model as well as the borderline causes for clock skew is conducted.



Algorithm 1: Transaction Coordinator

```
1  inflightOps ← ∅, txnTimestamp ← now()
2  for op ← KV operation received from SQL layer
3      op.ts ← txnTimestamp
4      if op.commit
5          op.deps ← inflightOps
6      else
7          op.deps ← { x ∈ inflightOps | x.key = op.key }
8          inflightOps ← (inflightOps − op.deps) ∪ { op }
9      resp ← SendToLeaseholder(op)
10     if resp.ts > op.ts
11         if op.key unchanged over (txnTimestamp, resp.ts]
12             txnTimestamp ← resp.ts
13         else
14             return transaction failed
15     send resp to SQL layer
16     if op.commit
17         asynchonously notify leaseholder to commit
```

(a) Transaction Coordinator

Algorithm 2: Leaseholder

```
1  Function Handle(op)
2      verify lease
3      wait for latches on keys of { op } ∪ op.deps
4      verify writes in op.deps are replicated
5      if op is not read-only
6          push op.ts past highest read timestamp for op.key
7      command, response ← evaluate op
8      response.ts ← op.ts
9      if not op.commit then
10         send response to coordinator
11     if op is not read-only
12         replicate and apply command
13     release latches
14     if op.commit
15         send response to coordinator
```

(b) Leaseholder

Figure 8: CockroachDB: High-level steps of the Transaction Coordinator and Leaseholder algorithm

3.3.5 Transaction model

Each transaction is assigned a gateway node, typically chosen out of proximity. This node functions as a transaction coordinator, and handles the connection between the SQL layer and the lower levels of Cockroach.

The transaction execute in two logical entities, both at the transaction coordinator and at the leaseholder. Pseudo-code for a high-level description of the respective algorithms are presented in Figure 8 to supplement the readers understanding.

Execution at the transaction coordinator
The gateway node receives a series of key value operations, but as SQL requires a response to the current operation, two optimizations are embedded to avoid stalling the transaction while the operations are replicated. These optimizations are write pipelining and parallel commits. Write pipelining allow returning results without waiting for replication of the current operation, and parallel commits allows the commit operation and the write pipeline to replicate in parallel.

As pipelining and parallelization are allowed, the coordinator needs to track operations that may not have completed replication yet, as well as maintain the transaction timestamp since it might need to be moved forward during the course of the transaction.

Write pipelining. Each operation includes the key that is read or updated, as well as metadata implying if the transaction is committing with the current operation. If the transaction is not committing with the current operation and the operation does not overlap an earlier one, it is possible to execute it immediately. In this way, multiple operations on separate keys can be pipelined. However, if the operation

is overlapping, it must wait for the earlier one to be replicated and thus stall the pipelining.

Next, the operation is sent to the leaseholder for execution and the coordinator then awaits a response. A read from another transaction might force the leaseholder to move the timestamp forward, and the coordinator will then need to adjust its own timestamp for them to match. A read refresh must then be performed, to check if they will return the same values with the new timestamp. If not, the transaction fails and may have to be retried.

Parallel commits. A transaction can commit once all of its writes have been replicated, which naively would require two sequential rounds of consensus through e.g. two-phase commit. Instead, Cockroach uses the parallel commit protocol, which uses a staging transaction state to make the true status of the transaction conditional on whether all of its writes have been replicated. Along with the staging status, the transaction must also include a list of the writes that have been performed by that transaction. A transaction is considered committed if its record is in the staging state, and an observer can prove that all of its writes have achieved consensus.

This reduces the number of required consensus rounds down to one as the coordinator can initiate replication of the staging status in parallel with the verification of outstanding writes. If both succeed, the coordinator can acknowledge the transaction as committed to the SQL layer, and asynchronously record the transaction as explicitly committed.

Execution at the leaseholder

Upon receiving an operation from the coordinator the leaseholder checks the validity of its lease. Next, it acquires locks on the keys that the operation touch upon as well as those it depends on. Then, it verifies that dependent operations succeed. If it performs writes it also ensures that the timestamp of the operation does not conflict with other operations. If all checks are in order, the leaseholder evaluates the operation to determine the necessary data modifications, resulting in a command to be executed as well as a response to the coordinator. If this operation is not committing the transaction, then the leaseholder can respond to the coordinator without waiting for replication. Writes are then replicated. After consensus is achieved the commands are applied locally. Finally, the leaseholder release locks and responds to the coordinator if it has not already responded.

Transactional atomicity

Transactions are atomically committed through viewing all of their writes as provisional until commit. Provisional writes are called write intents, stored as regular

MVCC key value pairs with additional metadata indicating its status as an intent and a pointer to a record with additional info about the status of the transaction which performed the write. The state of the transaction can be pending, staging, committed or aborted. The metadata record is able to atomically change the visibility of all intents at once.

A read encountering an intent will look up the transaction record of that intent. If the record says committed, the intent is regarded as a regular value. If it is aborted, the intent is aborted. For pending intents, the reader will block until the transaction is finalized. For staging transactions however, the reader will attempt to abort the transaction by blocking a replication of the transaction. If all writes are already replicated, the transaction is committed and the reader can safely perceive the intent as a regular value.

Concurrency control mechanisms
Cockroach uses a MVCC scheme, where each transaction performs its operations at the commit timestamp resulting in a total ordering of all transactions in the system, equivalent to serializability. In the case of conflicting transactions, this timestamp may be required to move. If so, the transaction will try to verify that its reads remain valid to continue forward with the updated timestamp. The following situations will require such logic:

Write-read conflicts arise when a read runs into an uncommitted intent with a lower timestamp. The reader will wait for the conflicting write to finish before continuing its operations.

Read-write conflicts arise when a write is attempted at a lower timestamp than a performed read. Cockroach forces the writing transaction to advance its timestamp past the timestamp of the earlier read.

Write-write conflicts arise when a write runs into either an intent with a lower timestamp or a committed write at a higher timestamp. In the case of an intent, the write must wait for the earlier transaction to complete. A committed value at a higher timestamp will force the writing transaction to move its timestamp forward. Deadlocks may occur if different transactions have written intents in different orders, a case handled through a distributed deadlock-detection algorithm.

Follower replica serving consistent historical reads
Cockroach allows non-leaseholder replicas to serve snapshot reads with timestamps that are sufficiently in the past. Such requests are specified through a query modifier which specifies the timestamp of the snapshot requested. For a replica to serve such

historical reads safely, the leaseholder must no longer update that specific key, and the follower must be sure that it is up to date with the prefix of the Raft log that affected the snapshot in question. For this purpose, each leaseholder periodically sends a closed timestamp message - a timestamp where no further write operations will be accepted. Closed timestamps along with the Raft indexes allow the replica to evaluate if it is sufficiently up to date to serve the snapshot read request.

3.3.6 Isolation and consistency levels under clock skew

Cockroach satisfy single-key linearizability for reads and writes under normal conditions, and thus ensures that stale reads are impossible as long as the clocks stay within the maximum offset of the deployment. This property is satisfied by monitoring an uncertainty interval for each transaction, under which the causal ordering of two transactions is undetermined. A transaction receives a provisional commit timestamp when it is created, along with an uncertainty interval defining a lower and upper commit timestamp. All values encountered that has a timestamp that fall under the transaction's uncertainty window is treated as past writes, and will cause an uncertainty restart. Under this restart, the lower bound of the provisional commit timestamp of the transaction is moved forward to avoid overlapping. Thus, the operations on each key take place in an order consistent with the real-time ordering of the transactions performing those operations.

However, when the maximum clock offset is violated Cockroach needs additional logic to ensure that the clock skew does not affect transaction isolation. Changes towards a single Range will remain consistent and therefore linearizable as they solely depend on Raft. But as previously described, reads are able to bypass the Raft protocol and thus cause complications if multiple nodes think they hold the lease for a single Range - a scenario which is possible under violation towards the clock offset. This could lead to possible conflicting operations from the leaseholders, resulting in isolation anomalies. To avoid it, Cockroach implements two safeguards detailed in Table 5. Together, these safeguards ensure serializable isolation under circumstances where violations towards the clock offset are present.

Still, violations toward maximum clock offset will cause the possibility of violations towards single-key linearizability between causally dependent transactions. This can happen if the transactions are issued from different gateway nodes who experience clock skew outside the offset bounds. The writes of the first transaction may be outside the uncertainty interval of the second, which would lead to the possibility of reads from the second overlapping the write set of the first - without the second

Safeguard	Description
1.	Range leases have a start and an end timestamp. A leaseholder cannot serve reads at timestamps above its interval, nor writes outside its interval.
2.	Each write to a Range's Raft log contain the sequence number of the Range lease. Upon replication, the sequence number is checked against the active lease. If they do not match, the write is dropped. Thus, only a single leaseholder is able to make changes to a Range at a time.

Table 5: CockroachDB: safeguards for clock skew
Source: CockroachDB: The Resilient Geo-Distributed SQL Database [47]

transaction actually observing the writes from the first. The result is stale reads. Stale reads are only prevented when the offset bounds are respected and thus violate the single-key linearizability property.

3.4 Calvin

Calvin is a transaction processing and replication layer designed to run on a non-transactional storage, transforming it into a shared-nothing, transactional, consistently replicated distributed database system [49]. Calvin facilitates concurrent distributed transactions that uphold ACID-guarantees, as well as supporting synchronous and asynchronous replication for geographically distributed data centers, achieving strong consistency guarantees specifically through a synchronous Paxos-based replication scheme.

Calvin achieves distributed transactions and synchronous replication through a deterministic execution framework, reaching agreement between multiple nodes outside the scope of transaction execution. This trait enables Calvin to omit distributed commit protocols, which again facilitates its scalability properties.

To create an understanding of how Calvin achieve its given properties, a description of its implementation, replication scheme, and concurrency control is conducted. This is based on the Calvin paper of Thompson et. al. [49]. Prior to this, a brief reasoning of how deterministic database systems are able to eliminate distributed commit protocols is given.

Eliminating distributed commit protocols

The main difference between Calvin as a deterministic database system and the previously presented systems is that it is not dependent on distributed commit protocols. The primary need for such agreement protocols in distributed systems is to ensure that a transaction has been successfully executed in its entirety, meaning that all operations at different nodes are in fact executed and have made it to durable storage. If a node is unable to commit its local changes the transaction will be aborted. Events that cause such abortions are classified as either non-deterministic or deterministic. Non-deterministic events are events that are caused by events not tied to the transaction, such as a node failure. Deterministic events on the other hand are caused by transactional logic.

The need for aborting transactions upon non-deterministic events can be eliminated through synchronous replication. If a node goes down during a distributed transaction, a replica can replace it. But this again raises a new issue, as it requires replicas to be actively replicated. The key feature of replication in deterministic databases is the fact that they replicate batches of transaction requests rather than replicating database states. This would not be feasible in a traditional system as

they only guarantee that transactions will be executed in some serial order. Thus, replicating only transaction request would lead to the possibility of diverging states at replicas for traditional systems. Oppositely, deterministic execution would ensure non-divergent behavior if replicas were to apply locks in the same order equivalent to the replicated transaction requests. Hence, deterministic databases allow replicas to be consistent simply by replicating transaction requests.

This partly eliminates the need for an agreement protocol at the end of a distributed transaction due to non-deterministic failures. The reason there is no need to check for failed nodes is that they would safely be replaced by a replica node given that the replica is either actively replicated or able to replay the execution history of the failed node through the replicated transaction request.

Deterministic events can be handled without the need for an agreement protocol at the end of a transaction. This could be done through a communication round, where each node involved in a transaction wait for a one-way message from every other node that could potentially abort the transaction, and then committing after it has received these messages.

3.4.1 Implementation

As mentioned, Calvin is designed to be a scalable, transactional layer on top of a storage system that implements a basic CRUD-interface. Calvin handles partitioning of data across the storage system and its nodes, as well as the scheduling of distributed transactions, replication, and communication across the network of nodes.

The system architecture of Calvin is illustrated in Figure 9. It is split into three main layers of processing, namely a sequencing layer, a scheduling layer, and a storage layer. All layers and their functions are partitioned across a set of shared-nothing nodes, thereby making them horizontally scalable. A node in the Calvin deployment, illustrated by the grey shading in Figure 9, runs one partition of each layer. Nodes in Calvin are organized into replication groups, where each group contains all replicas of a partition.

The sequencing layer handles transactional inputs and orders them into a global transaction sequence. This sequence is the ordering in which replicas will comply with during their local execution. The sequencers main responsibility is therefore also the replication and logging of the transaction input sequence.

The scheduling layer handles the concurrent execution of transactions through a

pool of execution threads. The scheduler adheres to the input sequence from the sequencing layer, using a deterministic locking scheme to achieve it.

The storage layer handles data layout, accessing physical data through a CRUD-interface.

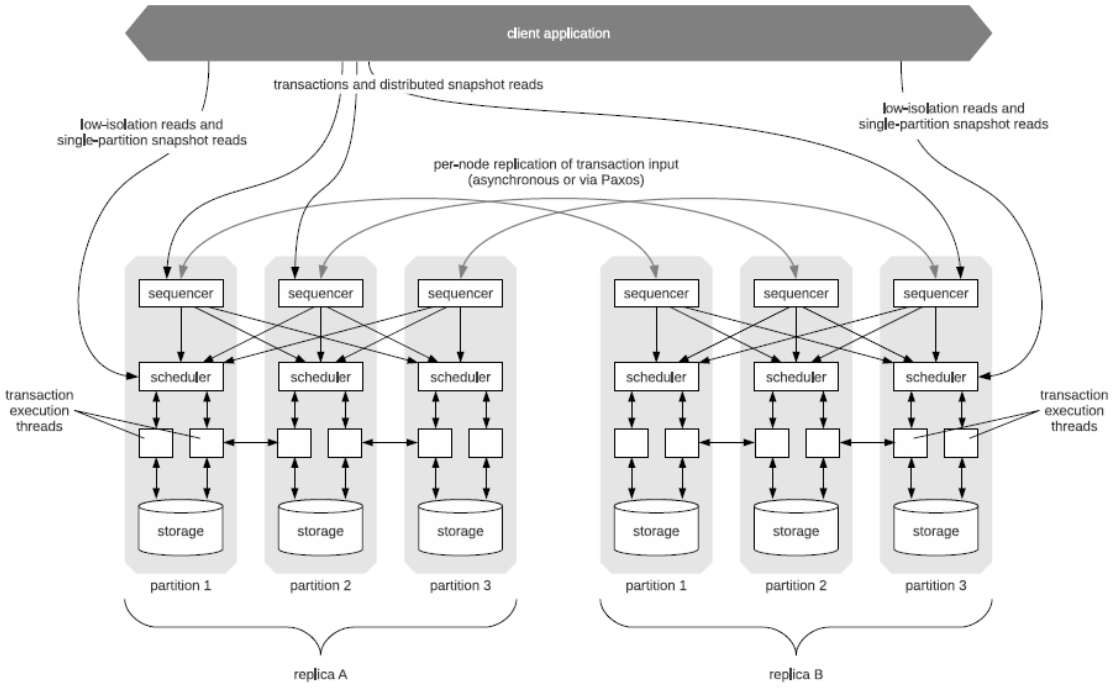


Figure 9: The system architecture of Calvin
Source: Calvin: Fast Distributed Transactions for Partitioned Database Systems [49]

3.4.2 Transactional interface

As depicted in Figure 9, transactions involving a write must go through the sequencing layer. This is also true for reads that require a high level of isolation.

From the client application, there are three main mechanisms for invoking transactions [48]. Namely,

- Built-in stored procedures invoking API calls with transactional semantics.
- Bundling of several API calls to define new built in procedures.
- Optimistic client-side transactions. These require a validation phase and are executed as deterministic stored procedures at commit time.

Additionally, Calvin has later been extended with support for secondary indexes and a smaller selection of SQL statements [48].

3.4.3 Transaction ordering and replication

The sequencing layer is distributed across all replicas, as well as partitioned across every machine at each replica. Each machine's sequencer collects transaction requests from clients, which are then batched together based on a time epoch. Epoch numbers are synchronously incremented across the system once every 10ms. These batched requests are then replicated, as replication in Calvin is done prior to transaction execution.

Calvin supports both asynchronous and synchronous replication. With asynchronous replication, one node is appointed to be master replica, and all requests are directed to sequencers located at the nodes of this master replica. When requests are batched together at the end of an epoch, the sequencer on each master forwards the batch to its slave sequencers within the replication group. This replication scheme has low latency, but at the cost of complex failure handling if the master goes down. A failed master incurs the need for the sequencers at both the master replica and in its replication group to agree on the last valid batch from the failed master, as well as what that batch actually contained. This is due to the fact that each sequencer only receive a partial view of each batch.

For synchronous replication, Calvin uses a Paxos-based scheme for all sequencers within a replication group. They agree on a combined transaction request batch for each epoch.

After replicating the batch, the sequencer sends a message to each scheduler on every partition within its replica. This message contains sequencer ID, the epoch number of the batch, and all transaction inputs that the receiver must participate in. A scheduler will then create its own global transaction ordering by interleaving all sequencers' batches for that epoch in a deterministic manner.

3.4.4 Concurrency control

Since Calvin decouples the transactional component from the physical data, both logging and concurrency control protocols must be entirely logical. This does not raise issues for logging as the state of the database is possible to determine completely based on input. However, logical concurrency control is a bit more complex, since locking key ranges and being robust to phantom updates typically require physical access to data. To overcome these issues, Calvin uses an implementation of virtual resources that can be logically locked in the transactional layer.

To achieve an understanding of how Calvin enforces concurrency control, one must look to the scheduling layer and its deterministic lock manager. The lock manager is partitioned across the scheduling layer. Each node's scheduler is only responsible for locking records that are stored at that node's storage component. The protocol is similar to two-phase locking but holds two additional invariants explained in Table 6.

Invariant	Description
1.	For any pair of transactions that both request an exclusive lock on a record, the one that is placed first in the serial order of the sequencing layer must be the one to first acquire the exclusive lock. Calvin enforces this by serializing lock requests in a single thread. This thread scans the transaction order from the sequencing layer, requesting locks in a serial order.
2.	The lock manager must grant locks in the order in which it receives them, again to comply with the serial order of the sequencing layer.

Table 6: Calvin: invariants for the deterministic locking protocol
Source: Calvin: Fast Distributed Transactions for Partitioned Database Systems [49]

Once a transaction has acquired its locks, it is handed off to a worker thread for execution. This execution has five phases. First, the worker thread performs a **read/write set analysis**. It identifies the reads and writes that are stored locally, as well as the set of active participants. Active participants are nodes at which elements of the write set are stored, as opposed to passive participants which are nodes where only elements of the read set are stored. Next, the worker thread performs **local reads**.

The execution thread then **serve remote reads**. The local reads are delivered to the worker threads on every other active participant in the transaction. Passive participants are omitted from this step since they do not need to execute actual transaction code, and thus need not collect any remote read results. Each worker thread on an actively participating node will then **collect remote read results**, as it needs both local and remote read results to be able to perform its local operations. Lastly, the worker thread **execute transaction logic** by applying all local writes. Remote writes can be ignored as they are the local writes of a counterpart working thread in another node.

Dependent transactions

A special case for deterministic databases are transactions that must perform reads to uncover their full read/write set, so-called dependent transactions. These are

not supported intrinsically by the deterministic transaction manager in Calvin since it requires read/write sets to be known in advance. Instead, Calvin supports the Optimistic Lock Location Prediction (OLLP) scheme to handle such transactions.

The OLLP scheme first performs a reconnaissance step, which is an unreplicated, low-isolation read-only transaction used to uncover the transaction's full read/write set. The transaction is then forwarded to the sequencer along with the resulting read/write set from the reconnaissance query for normal processing. However, there is a possibility that the records read in the reconnaissance step have changed during the time since it was first read, and it is therefore necessary to check the read results upon execution. If the read/write set is found to be no longer valid, the transaction must be retried along with a new reconnaissance operation.

3.4.5 Isolation and consistency levels

Transaction ordering handles much of the isolation and consistency of Calvin. Remembering that serializable isolation requires an ordering of transactions equivalent to some serial order, Calvin's deterministic approach restricts concurrency control to maintain equivalence to a pre-determined order. As long as the concurrency control layer ensures that locks are acquired in a fashion that agrees with the serial execution order, the database state is guaranteed not to diverge [49]. This is achieved through the deterministic lock scheme, where a single thread process lock requests such that locks are requested and granted in the same sequence as the global log. This, combined with synchronous replication of transaction requests allows Calvin to deliver a correctness guarantee of strict serializability, as the system is not prone to any time travel anomaly and maintains the one-copy serializability guarantee.

3.5 FaunaDB

FaunaDB is described as a transactional NoSQL database that upholds the scaling and productivity traits of the NoSQL movement, as well as providing the safety and correctness traits of ACID transactions [30]. Accordingly, FaunaDB falls under the category of NewSQL systems.

Fauna is a globally distributed cloud database-as-a-service, available through APIs. It scales within the cloud and offers pricing models according to use, thereby falling under the category of serverless systems.

As it is highly influenced by Calvin, Fauna adopts a deterministic transaction handling through which it supports strictly serializable, externally consistent transactions. Unlike Spanner and the systems inspired by it, Fauna does not rely on distributed physical clock synchronization nor specialized hardware to maintain transactional consistency.

The description of Fauna falls somewhat short of the others as we have struggled to find sources that can substantiate the claims made about Faunas promised features. The main source of information is Faunas own documentation as well as blog posts from its designers, but these are mainly for commercial use in our opinion. There is little explanation of the promised features except for the Fauna Distributed Transaction Protocol. Nevertheless, we choose to include Fauna in the thesis as we regard it as a substantial contribution to the NewSQL movement, being a commercial database service.

3.5.1 Implementation

A cluster is a concept that refers to the set of nodes that work together to provide a Fauna service. The components of a Fauna cluster are nodes, replicas, and shards. An illustration of a Fauna cluster topology is presented in Figure 10.

Nodes are computers running Fauna, able to run on both public and private cloud services. There is no notion of a master node, as all nodes will perform both storage and retrieval tasks as well as route queries.

A replica is made up of several nodes, and each node belongs to exactly one replica. All replicas contain a full copy of the data within the cluster [40], hereby user data, system data, and the transaction log. There is no redundancy within a replica, as each data item is found on exactly one node. A replica is therefore comparable to a datacenter.

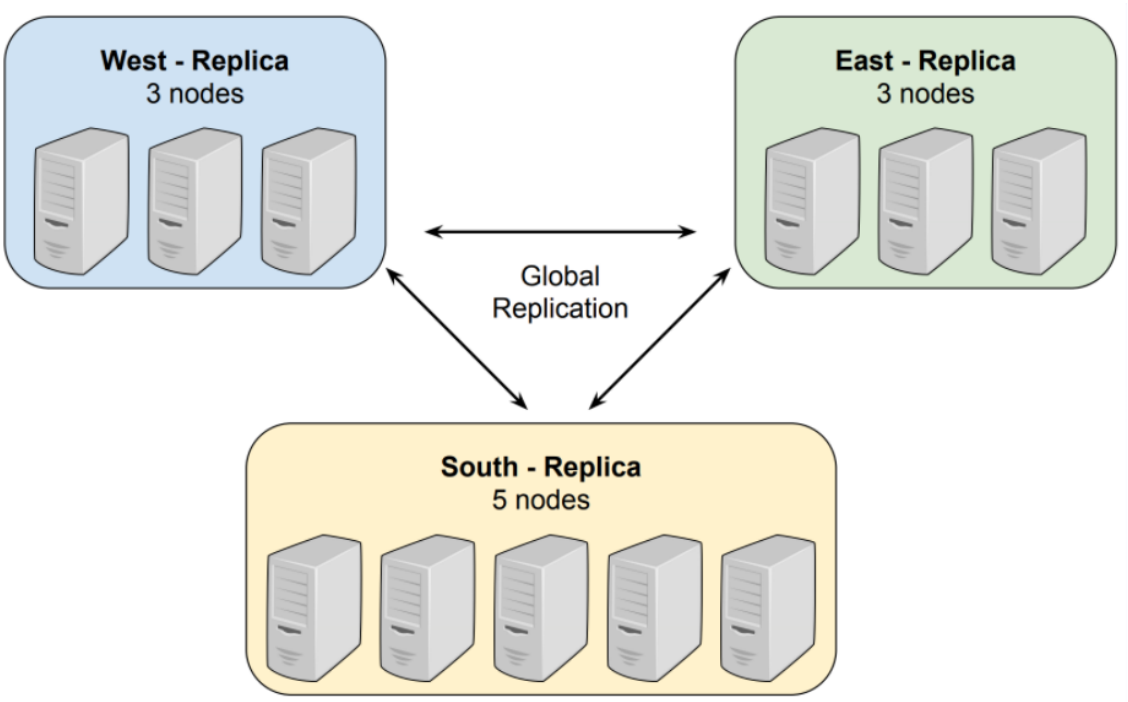


Figure 10: A typical Fauna cluster topology

Source: Fauna blog [39]

Data model

The data model represents the logical layout of a cluster. Fauna implements a semi-structured, schema free object-relational data model. Records are defined as semi-structured documents named instances, and several instances are grouped together in a class. This is comparable to tables, even though Fauna does not require a shared schema within the class. Databases are made up of such classes, and one database might recursively contain other databases. Fauna adopts a NoSQL query language to adhere to the object-relational data model.

Data is partitioned and replicated across the set of nodes, with a group of nodes forming a replica. Each partition contains multiple records, and each record may have several versions as records in Fauna are temporal. An instance is immutable, so a change to an instance will cause Fauna to create a new instance version instead of overwriting the previous one. Each instance is annotated by a transaction identifier, giving the version a contextual placement in the history of the instance. Changes to instances are either create, update or delete events, as a typical CRUD interface of a NoSQL system.

Nodes are transparent to the users, but nevertheless, users are able to modify data locality through the Region Groups infrastructure. Region Groups facilitate the user

in tying the database, storage, and computative services to a geographic region to comply with local legislation such as GDPR [28].

3.5.2 Replication

The replication strategy is presented to be highly influenced by Calvin in the sense that Fauna adopts a master-less architecture with replicas in a cluster performing active-active replication. All nodes can perform any request, inserting transactions into a global log in a similar way as Calvin. Replicas tail this log and apply writes accordingly. Commits to the replicas are said to be semi-synchronous [29], but to our knowledge, the requirements for the level of replication upon an acknowledged commit are not specified through the documentation.

3.5.3 Isolation and consistency levels

FaunaDB delivers strict serializability across multi-key transactions. Furthermore, the system upholds a global consistency guarantee across a geographically distributed system:

> Once a transaction commits, it is guaranteed that any subsequent read-write transaction will read all data that was written by the earlier transaction.

The level of isolation is configurable to the user, spanning from strict serializability to snapshot isolation. Read-write transactions are strictly serializable, and read-only transactions are serializable by default. It is possible to query for strict serializable reads through a linearized query specifier [27]. Apart from this, Fauna offers historic reads through a query specifier. Isolation is enforced through a deterministic concurrency control scheme in the same manner as Calvin.

Fauna enforces its consistency properties through the Fauna Distributed Transaction protocol. This explanation of the Fauna transaction protocol is based on the works of Daniel Abadi and Matt Freels [14].

3.5.4 The Fauna Distributed Transaction protocol

Like Calvin, Fauna batches transactions based on time epochs at each server. These batches are then inserted into a distributed, write-ahead transaction log. The execution engine then ensures that the final result of the transaction processing is equivalent to this log. There is only one transaction log for the entire deployment, so replicas must achieve consensus for how to insert transactions into the log.

Each log entry is associated with a real time, but unlike Spanner based systems, the transaction protocol does not base its execution on real time. Instead, the notion of time in Fauna is entirely dependent on the order of transactions in the distributed log.

Any replica can serve any client request, both read-only and read-write transactions. The server within the replica that receives a request becomes the coordinator for that request. The protocol is based on a speculative scheme since the server receiving the request first executes it locally. In this speculative step, the coordinator chooses a recent timestamp and issues reads to nearby servers within its replica as well as performing local reads. Then, the coordinator completes the transaction logic and buffers the writes locally.

Next, the coordinator inserts its batch of transactions into the distributed log. The Raft protocol is used to achieve consensus between replicas on the order in which batches of transactions are inserted. This step also involves the assignment of a global transaction identifier to the transactions that are inserted, which are assigned relative to their position in the log. The log entry itself contains the transaction identifier, as well as a record of the buffered writes and reads performed by the transaction coordinator. This is the only part of the Fauna protocol that requires consensus across replicas.

All replicas will then independently attempt to commit each transaction in the distributed log. Since the log entry was speculatively executed, the replicas will need to redo the reads of the original transaction execution to check whether any of the reads have become stale. These new reads are issued at the timestamp of the transaction from the global log to ensure the global serializability property. If none of the reads have become stale, the transaction can be executed and the replica append the buffered writes to its core tables. The outline of the Fauna distributed transaction protocol is summed up in Table 7.

Even though the replicas perform this last phase without communicating with each other, they will always come to the same conclusion about stale reads, and thus

always agree on the value of a snapshot collected from the global log. This is due to the fact that there is only one global log, and this log implies a linear order of global operations. The linearity of these operations is enforced by each replica, which again ensures that replicas remain consistent.

Phase	Description
Speculative phase	The coordinator executes the transaction speculatively, buffering writes and performing reads at a chosen timestamp.
Commit phase	The transaction is issued, along with the coordinators transaction batch, to a distributed log where it receives a global timestamp and placement in the global ordering.
Confirmation phase	Each replica perform local checks on the transaction, and issues the buffered writes to the database if these checks do not uncover any changes. Otherwise, the transaction is aborted and restared.

Table 7: The Fauna Distributed Transaction Protocol

Source: Consistency without Clocks: The Fauna Distributed Transaction Protocol [14]

4 Evaluation

Firstly, a recap of the terms isolation, concurrency, and consistency is reasonable. Isolation guarantees refer to the task of shielding concurrent transactions from each other. Concurrency concerns synchronization of concurrent access to the distributed database, such that the integrity of the system is upheld. One way to look at it is that concurrency enforces isolation guarantees. Consistency on the other hand revolves around upholding a predefined set of invariants of the system and is required to specify how non-concurrent transactions are to be processed in relation to each other. CAP consistency requires that a total order in all operations exists, such that it appears as though the operations were executed on a single machine.

As replication makes a system prone to anomalies despite isolation guarantees, there is not much sense to evaluate the systems isolation guarantees by themselves. Instead, the notion of correctness guarantees is a more applicable measure, combining both the isolation and consistency guarantees of the system, as presented with the works of Attar, Bernstein and Goodman [17] in section 2.8.

This section seeks to compare the systems and create an understanding of which design decisions lead to their diverging approaches to achieve the described features. Finally, a summary of the described features is presented to emphasize the different approaches to architecture, the data and query model, concurrency control, replication, and the overall correctness guarantee that the systems provide.

4.1 Spanner vs. Calvin

The two origins of NewSQL systems are fundamentally different in the way they order transactions. Spanner uses TrueTime while Fauna opts for sequencing. Most differences between the two systems, and their descendants, stem from this choice.

In Spanner, locks are obtained within the replicas on all data that are to be written before performing any writes. If all locks are acquired, the transaction proceeds with its writes and is assigned a timestamp after waiting out the uncertainty window of the transaction coordinator. Locks are then released and a commit is issued, so that the transaction receives a timestamp based on the actual time it committed. This timestamp is the means for transaction ordering. Later transactions will see all writes of earlier transactions, enforced by the locking scheme.

Calvin however, opts for a distributed, replicated log for transaction ordering comparable to a write-ahead log for a traditional, non-distributed database. Transactions

are inserted into this global log and ordered through a consensus protocol, thereby ordering transactions outside the scope of execution. Replicas process transactions from their local copy of the log, guaranteeing that the final state is equivalent to as if each transaction in the log was executed one by one.

The choice of transaction ordering method affects the systems approach to replication [8]. For Calvin, the transactional inputs to the log are the sole unit of replication. The deterministic execution framework makes all other cross-replica consensus steps redundant, as the replicated log also ensures that replicas process transactions in a way that guarantees that each replica is equivalent to every other. This is achieved through a preprocessing step where the transaction is pre-executed to remove any non-deterministic operations which could lead to diverging replicas. In this way, replication is performed in advance of the actual transaction execution.

For Spanner, however, transactions are not preprocessed and hence not possible to replicate prior to execution since they are non-deterministic. Spanner opts for replication after transaction execution through a cross-region Paxos protocol.

The two systems also diverge in how they handle a transaction touching upon multiple partitions. To guarantee atomicity and durability, such transactions need a commit procedure to ensure that every partition successfully completed its part. Since servers may fail during any point in the execution timeline, another round of communication is needed between the participating servers - e.g. two-phase commit.

Spanner uses two-phase commit for its distributed transaction handling. In contrast, Calvin omits the two-phase protocol through its deterministic execution only depending on a one-phase commit for inserting transactions into the global log. Calvin does not need to verify that the participating parties have in fact not failed, as they would be able to safely replay their operations through the replicated log reducing the needed rounds of communication down to one.

4.2 The CAP theorem

NewSQL systems positions themselves in relation to the theorem as CP systems, as opposed to NoSQL systems that sacrifice consistency for availability being AP systems.

Some argue that there is little to it to evaluate distributed systems against the CAP theorem today [20, 9, 5]. The reason being that perfect availability is fundamentally not achievable, as no system can guarantee it. So, if one cannot choose availability,

it would make sense to guarantee perfect consistency at the cost of a reduction to the already imperfect availability. Furthermore, as described by Abadi [7], systems that guarantee consistency will only experience reduced availability in the case of network partitions. Even then, the availability might not be compromised to a large extent, as the systems only require a majority of the cluster to be available. Consequently, for a visible reduction in availability to occur, there must both be a network partition and the clients should only be able to connect with the minority partition of the cluster. By following this reasoning, one can argue that it is possible for a distributed system to guarantee consistency while still achieving high availability as these scenarios are relatively rare.

4.3 Consensus in NewSQL systems to achieve CAP consistency

NewSQL systems enforce consistency through the use of consensus protocols as Paxos or Raft. These protocols work by a majority voting mechanism as an update to a data item require a majority of the replicas to acknowledge it. Thus, a minority of replicas can be unavailable without compromising the system's ability to serve requests.

Even though both systems and their descendants use consensus algorithms to enforce consistency, their implementation of it differs. Again, there exists two main approaches. One side is the Spanner-based systems which partition the data into shards, and then perform consensus per shard. The other is the Calvin-based systems, which use a global consensus protocol per database.

Both have disadvantages. Partitioned consensus makes the system prone to causal reverse anomalies, as depicted in Figure 11, for transactions that touch upon multiple shards. Spanner overcomes this problem through their TrueTime API, assigning timestamps that give an indication of "before" and "after" - also for transactions handled at different servers. As server clocks might diverge, transactions wait out an uncertainty window to guarantee that time traveling anomalies are avoided. However, this inflicts latency upon the transaction, and a larger uncertainty window will directly affect the concurrency of the system. Spanner achieves a relatively low uncertainty window through the use of both GPS and atomic clocks, to ensure a minimal clock skew across servers.

A global consensus protocol on the other hand limits scalability, as every transaction requires a consensus protocol and the set of servers must vote on each transaction.

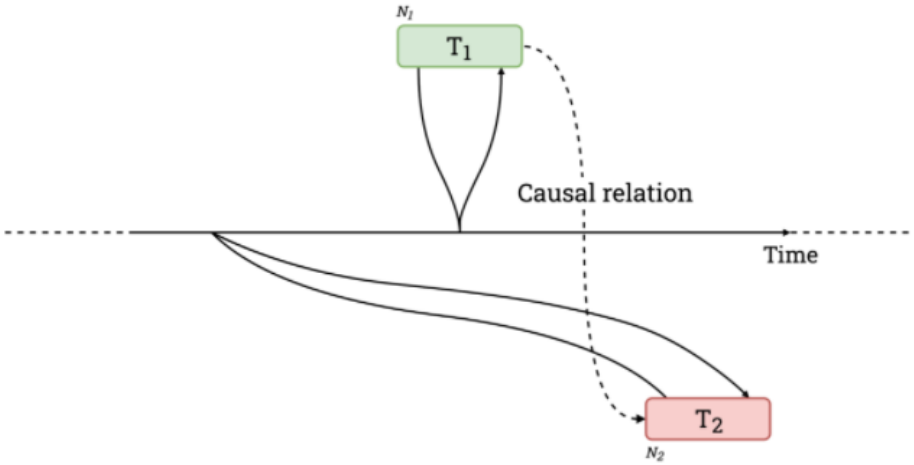


Figure 11: The causal reverse anomaly
Source: https://www.cockroachlabs.com/blog/living-without-atomic-clocks/

As voting requires communication and each server is limited by its ability to process a given number of messages per second, it limits the number of transactions that the system is able to handle per second [9]. Calvin and Fauna reduce the impact of this through the batching of requests, reducing the number of needed consensus rounds by making the servers vote on batches of transactions rather than every transaction independently. Additionally, we would like to point out that the complications caused by partitioned consensus are not present with the global consensus approach as there is no need to order the transactions by a notion of before and after. This is handled in its entirety through the consensus protocol, as the order of transactions is given from it.

As Calvin, Fauna implements a global consensus protocol and is thus able to guarantee consistency of transactions. Cockroach on the other hand implements partitioned consensus as Spanner, and therefore faces the same challenges associated with clock skew to avoid consistency violations. Cockroach does not claim to achieve CAP consistency, as it only implements software-based clock logic, and therefore lacks the support for minimizing and measuring clock uncertainty in the same way as Spanner does.

4.3.1 Anomalies

When clock skew is violated, Cockroach is prone to both stale reads and the causal reverse anomaly [47, 38]. These anomalies occur in Cockroach during the following scenarios.

Causal reverse

A transaction can commit, and a later transaction caused by the first one could write data to a different partition and still receive a timestamp that is earlier than the first transaction. This enables a third transaction to potentially see the write of the later, but not the earlier one - thus observing the effect without the cause.

Stale read
This can happen if two causally dependent transactions are issued from different gateway nodes that experience clock skew outside the offset bounds. The writes of the first transaction may be outside the uncertainty interval of the second, which would lead to the possibility of reads from the second overlapping the write set of the first - without the second transaction actually observing the writes from the first.

4.4 Concurrency control methods

Concurrency control algorithms enforce isolation levels by coordinating the execution of multiple, concurrent transactions to uphold a consistent state of a database [43]. As replication makes a system prone to anomalies despite isolation guarantees, the concept of concurrency control for distributed systems is intertwined with the notion of correctness guarantees - which offer a consistency guarantee in addition to isolation to fully cover which anomalies a system is exposed to.

Spanner uses MVCC and 2PL supported by the TrueTime API to enforce isolation between concurrent transactions. Albeit making the system prone to deadlocks, 2PL is used to implement mutual exclusion for write transactions. Deadlocks are handled through a deadlock detection mechanism. The described level of isolation, or correctness, is external consistency, a further restricted version of strict serializability [21].

Cockroach implements a variation of MVCC to enforce concurrency control, as well as the concept of write intents which facilitates for only needing one round of consensus to commit. Locks are implemented at the leaseholder to secure mutual exclusion for transactions touching upon the same set of keys. The described isolation level is single-key linearizability. Hence, Cockroach does not guarantee a total linear ordering of transactions that touch upon different keys.

Concurrency control in Calvin is enforced by the scheduling layer through a deterministic framework. The deterministic invariant limits the tasks related to concurrency control down to maintaining equivalence to a specific, pre-determined execution order. This is done through a deterministic locking scheme, where a protocol

similar to 2PL is used at each node's scheduler to achieve mutual exclusion. The locking scheme must also adhere to the pre-determined order, a trait that is enforced by serializing lock requests in a single thread. Through the global ordering from the sequencing layer, Calvin avoids time travel anomalies and delivers a correctness guarantee of strict serializability [11].

Fauna upholds a global consistency guarantee, stating that once a transaction is committed, any subsequent transaction will be able to observe its writes. Consistency is enabled through the Fauna Distributed Transaction protocol, which utilizes a deterministic approach to transaction ordering as Calvin.

4.4.1 Correctness guarantees

Consequently, all but Cockroach falls under the category of a strict serializable correctness guarantee. Cockroach falls somewhat short of the others as it is prone to causal reverse anomalies, as well as stale reads if clock synchronization surpasses the maximum skew limit. This is due to their implementation of consistency without specialized hardware clocks and the fact that the transaction model does not wait out maximum clock skew bound before committing a transaction.

4.5 PACELC

As presented by Abadi [7], the PACELC theorem extends the CAP theorem by regarding what trade-offs are made during normal operation. The theorem complies CAP during failure, but else states that the system must make a fundamental choice as to if it prefers consistency or lower latency. This trade-off stems from replication, which is fundamentally required to achieve availability. Hence, the systems are either CP or CA during failures and EL or EC during normal operation. In general, all of the discussed systems choose consistency and are thus CP/EC systems utilizing synchronous replication. Some still offer configurable lower latency for reads, and become CP/EL systems under these settings.

4.6 ACID

In general, atomicity is achieved through the notion of transactions being either fully executed or aborted if one operation should fail. All systems adopt this notion of atomicity and execute transactions in their entirety.

ACID consistency entails that a transaction will take the database from one consistent state to another. Thus, a transaction shall only be committed if it does not violate the invariants of the system. This entails that operations towards the database must always see the result of previously committed transactions on overlapping data to prevent erroneous execution e.g. if a write is performed based on a stale read. All systems guarantee this through their transaction ordering, as detailed in section 4.1.

Isolation is secured in all systems, as they facilitate for serializable execution of concurrent transactions and exclusive access to shared resources through locking. All but Cockroach shield concurrent transactions from each other until commit. For Cockroach, additional logic is implemented to make sure that write intents are handled properly, as described in section 3.3.5.

Durability is achieved through consistent replication, though handled differently by the two groups of systems. Spanner and Cockroach use active-passive replication, while the deterministic ones utilize active-active replication. Still, both deliver strongly consistent replicas.

4.7 Data and query model

To facilitate deterministic processing, Calvin requires preprocessing of transactions to analyze and remove any non-deterministic operations to ensure that execution will not cause divergence in replica states [8]. This limits Calvin's transaction model, as the preprocessor requires the entire transaction to be submitted simultaneously. Spanner, on the other hand, allows for arbitrary client-side interactive transactions which may include external communication.

Consequently, Spanner uses a query model comparable to SQL while Cockroach employs a standard SQL implementation. Calvin and Fauna on the other hand employ a NoSQL query language.

4.8 Summary

Table 8 provides a summary of the given features of the systems discussed throughout this thesis.

Feature	Spanner	Cockroach	Calvin	Fauna
Architecture	Novel	Novel	Middleware layer	Database-as-a-service
Partitioning	Yes	Yes	Yes	Yes
Data model	Schematized semi-relational	Schematized relational	Non-relational, key-value	Semi-structured, schema free object-relational
Query model	SQL comparable	SQL	NoSQL	NoSQL
Concurrency control	MVCC + 2PL	MVCC	Deterministic	Deterministic
Replication	Strong + Passive	Strong + Passive	Strong + Active	Strong + Active
Correctness	Strict serializable	Serializable	Strict serializable	Strict serializable

Table 8: Summary of the system features of the NewSQL systems

5 The future of NewSQL

The research group that first specified the class of NewSQL systems have recently published an evaluation of their initial work, looking back at their original evaluation and comparing it to the actual growth of the NewSQL movement after a decade.

In this report [16], they conclude that the pioneering vendors of NewSQL failed to make an impact on the database market. In hindsight, it is especially telling to look at the survival rate of these early NewSQL vendors compared to the survival rate of the NoSQL systems touched upon in the initial report. Looking closer at the systems from this first generation, most have vanished, some has narrowed their scope, and other remain but has altered their focus. Consequently, it would be correct to conclude that the NewSQL movement has not been able to conquer the market for distributed database systems.

Nevertheless, the NoSQL approach is still a poor fit for many applications, especially to those that utilize transactions spanning multiple partitions. The NoSQL systems force the developer to split the query into multiple transactions, and the developer thereby inherit the burden of reasoning about data placement, atomicity, and isolation [48]. Therefore, NewSQL systems are still relevant. So, why did the first generation fail?

A plausible explanation could be that they competed against a market dominated by highly established vendors with a substantial degree of lock-in for their customers. A change of vendor would inflict costs as well as technological incompatibilities upon the customer, so the risk of choosing a pioneering vendor versus sticking with their established solution could be too great.

Another aspect is that many NoSQL providers have extended their solutions with support for relational schemes, making NoSQL systems adoptable for some workloads similar to what the NewSQL systems target.

Furthermore, Aslett [16] argue that the pioneers of NewSQL were perhaps to early on the scene, being ahead of customer demand for distributed data processing and database services.

However, a new generation of relational database systems have succeeded in capturing a share of the market with notably Spanner and Cockroach in the forefront. Fauna is an established provider of managed serverless database services. Others, though not touched upon in this thesis, have also excelled as TiDB, AWS Amazon Aurora, and Hyperscale - the Azure Database for PostgreSQL.

To understand why this second generation has fared better than its predecessor, one must look to the market. Customer demands for globally distributed databases has excelled compared to a decade ago [16], especially towards as-a-service offerings. In addition to this, the new vendors have adopted several successful features of the NoSQL movement in the ability to focus not only on technical expertise but also developer engagement and ease of use [16]. Lastly, there has been a refinement in the offerings of NewSQL systems. Over the last decade, the NewSQL vendors have had the opportunity to adapt their product according to market feedback and specialize the product towards a market segment.

Aslett [16] argues that the second generation succeeds in highlighting the technical value of utilizing a distributed architecture, as well as benefiting from developer engagement. These benefits materialize in ease of adoption, ease of use and productivity improvement. Arguably an area were the pioneering systems fell short in comparison to the NoSQL vendors.

Although a new line of database services has been fairly established, the term NewSQL is not widely used. Instead, vendors opt for the term "distributed SQL", which is perhaps a more eloquent phrase of their provided services.

One cannot deem the first generation of NewSQL as a complete failure, as that would be to overlook the complexity of what the different vendors were trying to achieve, as well as the lessons learned by the second generation - a successful continuation of a subset of the overall category of NewSQL, namely the as-a-service systems as well as those built based on novel architecture.

5.1 Market

Today, 451 Research [16] estimates that the various NewSQL database providers generated a revenue of $ 587m in 2020, which is only a small proportion of the overall $35.6bn operational database market. In comparison, the NoSQL database providers generated a revenue of $4.9bn in 2020 which is more than eight times the revenue of the NewSQL providers.

On the basis of a survey done by 451Research [6], one could argue that there is still potential of growth for the segment of distributed database offerings. In this survey, they put the question forward as to which of the following statements that best describe the organization's usage of globally distributed databases. Figure 12 show the statement options along with a pie chart of the given answers. Here, 28% report that they in fact use a globally distributed database in production, but more

interestingly 58% of the contributors state that they plan on adopting a globally distributed database within the next twelve months. Additionally, 70% state that they plan on adopting one in the next three years.

Finally, we would like to point out that this survey cover the whole segment of distributed systems. So, the question for the NewSQL systems is if they are able to capture some of this potential growth from the other distributed database vendors, as the NoSQL systems.

The question at hand is if the NewSQL vendors are able to capture some of this potential growth from the other distributed database vendors as the NoSQL systems.

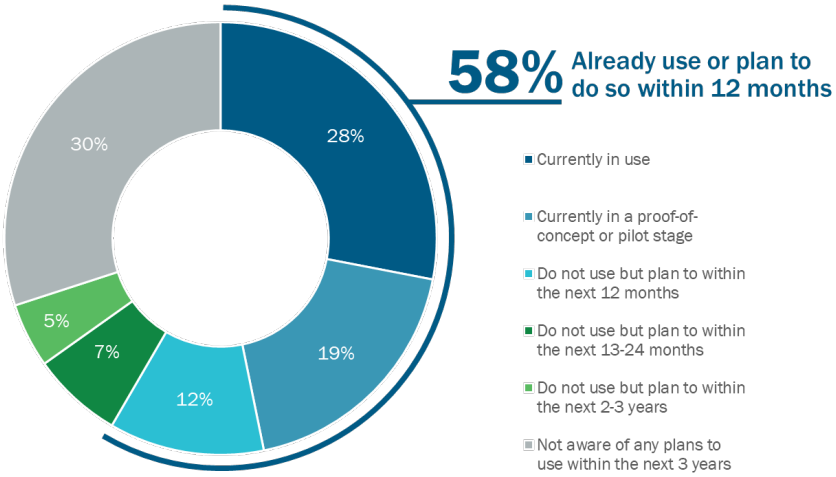


Figure 12: Globally distributed database adoption plans
Source: 451 Research's Voice of the Enterprise: Data & Analytics, Platforms, 2021

6 Conclusion

NewSQL is a category of distributed database systems that offer ACID semantics for transactions all the while they provide horizontal scalability and high availability. Availability is achieved through replication, while the horizontal scalability trait is facilitated by partitioning data across a set of nodes in a shared-nothing architecture. These systems target OLTP workloads, where transactions are repetitive and only touch upon a small subset of the data.

Regarding the fundamental trade-offs in distributed systems, embodied through the PACELC theorem, we have seen that NewSQL systems choose consistency over both availability and latency, as opposed to other vendors of distributed database services that sacrifice consistency for low latency and high availability.

For the systems we have looked into, there are two different approaches to transaction ordering. One side is the Spanner-based systems that order transactions through a timestamp scheme. On the other is the Calvin-based systems, ordering transactions outside the scope of transaction execution. This has consequences for their approach to replication. Spanner and CockroachDB use passive replication of execution results, whereas the Calvin-based systems actively replicate transaction requests.

Consistency is enforced through consensus. Again, we have seen that the choice of transaction ordering method leads to diverging approaches to consensus. Spanner and Cockroach perform consensus per shard, while Calvin and Fauna opt for a global consensus protocol per database. This results in different challenges with respect to design. With the global consensus protocols, one does not need to handle time-traveling anomalies but faces challenges with scalability. Calvin and Fauna resolve the scalability issue through batching of requests, reducing the number of needed consensus rounds. Through partitioned consensus, additional logic is needed as the system is exposed to time-traveling transactions. Spanner handles this through their TrueTime API and thus achieves CAP consistency, as opposed to Cockroach that does not claim to achieve CAP consistency but opts for a single-key linearizability guarantee.

As described, most of the systems deliver a configuration of strict serializability, achieving both one-copy serializability as well as restricting transactions from traveling in time. Albeit not achieving strict serializability as the rest, CockroachDB still delivers a correctness guarantee that is somewhat stronger than serializable, but prone to anomalies under violations towards clock skew.

Although the pioneering systems did not live up to the expectations raised a decade ago, the second generation of NewSQL systems have established themselves among the distributed database vendors today. Even so, the NewSQL, or distributed SQL, vendors only capture a small portion of the overall market for distributed database systems. There are indications that the market for distributed database systems has a potential for growth in the near future, but this does not necessarily entail that NewSQL systems will capture this growth as we have seen that other distributed database vendors expand their existing solutions to make them adoptable to similar workloads as the NewSQL systems target.

Bibliography

[1] Cockroachdb docs. URL https://www.cockroachlabs.com/docs/stable/. Accessed: 2021-11-01.

[2] Replication, . URL https://cloud.google.com/spanner/docs/replication. Accessed: 2021-09-14.

[3] Cloud spanner: Truetime and external consistency; google cloud, . URL https://cloud.google.com/spanner/docs/true-time-external-consistency. Accessed: 2021-09-14.

[4] Rocksdb. https://rocksdb.org/. Accessed: 2021-10-22.

[5] Achieving acid transactions in a globally distributed database, Sep 2017. URL https://fauna.com/blog/acid-transactions-in-a-globally-distributed-database. Accessed: 2021-12-02.

[6] 451 Research group. Research's voice of the enterprise: Data & analytics, 2021.

[7] D. Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42, 2012.

[8] D. Abadi. Distributed consistency at scale: Spanner vs. calvin, Apr 2017. URL http://dbmsmusings.blogspot.com/2017/04/distributed-consistency-at-scale.html. Accessed: 2021-11-30.

[9] D. Abadi. Newsql database systems are failing to guarantee consistency, and i blame spanner, Sept 2018. URL http://dbmsmusings.blogspot.com/2018/09/newsql-database-systems-are-failing-to.html. Accessed: 2021-11-30.

[10] D. J. Abadi. Demystifying database systems, part 1: An introduction to transaction isolation levels, May 2019. URL https://fauna.com/blog/introduction-to-transaction-isolation-levels. Accessed: 2021-11-07.

[11] D. J. Abadi. Demystifying database systems, part 2: Correctness anomalies under serializable isolation, Jun 2019. URL https://fauna.com/blog/demystifying-database-systems-correctness-anomalies-under-serializable-isolation. Accessed: 2020-10-03.

[12] D. J. Abadi. Demystifying database systems, part 3: Introduction to consistency levels, Jul 2019. URL https://fauna.com/blog/demystifying-database-systems-introduction-to-consistency-levels. Accessed: 2020-10-03.

[13] D. J. Abadi. Demystifying database systems, part 4: Isolation levels vs. consistency levels, Aug 2019. URL https://fauna.com/blog/demystifying-database-systems-part-4-isolation-levels-vs-consistency-levels. Accessed: 2020-10-04.

[14] D. J. Abadi and M. Freels. Consistency without clocks: The fauna distributed transaction protocol, Oct 2018. URL https://fauna.com/blog/consistency-without-clocks-faunadb-transaction-protocol. Acessed: 2021-10-24.

[15] M. Aslett. How will the database incumbents respond to nosql and newsql, 2011.

[16] M. Aslett. Ten years of newsql: Back to the future of distributed relational databases, Jun 2021. URL https://www.spglobal.com/marketintelligence/en/news-insights/blog/ten-years-of-newsql-back-to-the-future-of-distributed-relational-databases. Accessed: 2021-10-23.

[17] R. Attar, P. A. Bernstein, and N. Goodman. Site initialization, recovery, and backup in a distributed database system. *IEEE Transactions on Software Engineering*, (6):645–650, 1984.

[18] D. F. Bacon, N. Bales, N. Bruno, B. F. Cooper, A. Dickinson, A. Fikes, C. Fraser, A. Gubarev, M. Joshi, E. Kogan, et al. Spanner: Becoming a sql system. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 331–343, 2017.

[19] P. A. Bernstein and N. Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983.

[20] E. Brewer. Spanner, truetime and the cap theorem. 2017.

[21] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31 (3):1–22, 2013.

[22] G. F. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems: concepts and design*. pearson education, 2005.

[23] Dragland. Big data – for better or worse, May 2013. URL https://www.sintef.no/en/latest-news/big-data-for-better-or-worse/.

[24] R. Elmasri. *Fundamentals of database systems*. Pearson Education Limited, 7 edition, 2016.

[25] A. J. Elmore, V. Arora, R. Taft, A. Pavlo, D. Agrawal, and A. El Abbadi. Squall: Fine-grained live reconfiguration for partitioned main memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 299–313, 2015.

[26] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.

[27] Fauna, Inc. Isolation levels, Nov 2021. URL https://docs.fauna.com/fauna/current/learn/understanding/isolation_levels. Accessed: 2021-11-17.

[28] Fauna, Inc. Region groups, Nov 2021. URL https://docs.fauna.com/fauna/current/learn/understanding/region_groups. Accessed: 2021-11-17.

[29] Fauna, Inc. Postgres vs. fauna, Nov 2021. URL https://docs.fauna.com/fauna/current/comparisons/compare-faunadb-vs-postgres#replication-and-high-availabilitys. Accessed: 2021-11-17.

[30] M. Freels. Faunadb: An architectural overview, 2018.

[31] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, 33(2):51–59, 2002.

[32] J. Han, E. Haihong, G. Le, and J. Du. Survey on nosql database. In *2011 6th international conference on pervasive computing and applications*, pages 363–366. IEEE, 2011.

[33] R. Harding, D. Van Aken, A. Pavlo, and M. Stonebraker. An evaluation of distributed concurrency control. *Proceedings of the VLDB Endowment*, 10(5): 553–564, 2017.

[34] M. Kleppmann. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. " O'Reilly Media, Inc.", 2017.

[35] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[36] L. Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, pages 179–196. 2019.

[37] S. Maiyya, F. Nawab, D. Agrawal, and A. E. Abbadi. Unifying consensus and atomic commitment for effective cloud data management. *Proceedings of the VLDB Endowment*, 12(5):611–623, 2019.

[38] A. Matei. Cockroachdb's consistency model, Jan 2019. URL https://www.cockroachlabs.com/blog/consistency-model/. Accessed: 2021-11-07.

[39] J. Miller and S. Schrader. Introduction to fauna clusters, June 2018. URL https://fauna.com/blog/introduction-to-faunadb-clusters. Accessed: 2021-11-22.

[40] J. Miller and S. Schrader. The database that stays alive even when you issue a command to remove the last replica, June 2018. URL https://fauna.com/blog/fauna-topology-operations. Accessed: 2021-11-22.

[41] J. S. Mo. *Distributed Transactions in NewSQL*. Project thesis, NTNU - Norwegian University of Science and Technology, Dec 2020.

[42] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319, 2014.

[43] M. T. Özsu and P. Valduriez. *Principles of distributed database systems*. Springer, 4 edition, 2020.

[44] A. Pavlo and M. Aslett. What's really new with newsql? *ACM Sigmod Record*, 45(2):45–55, 2016.

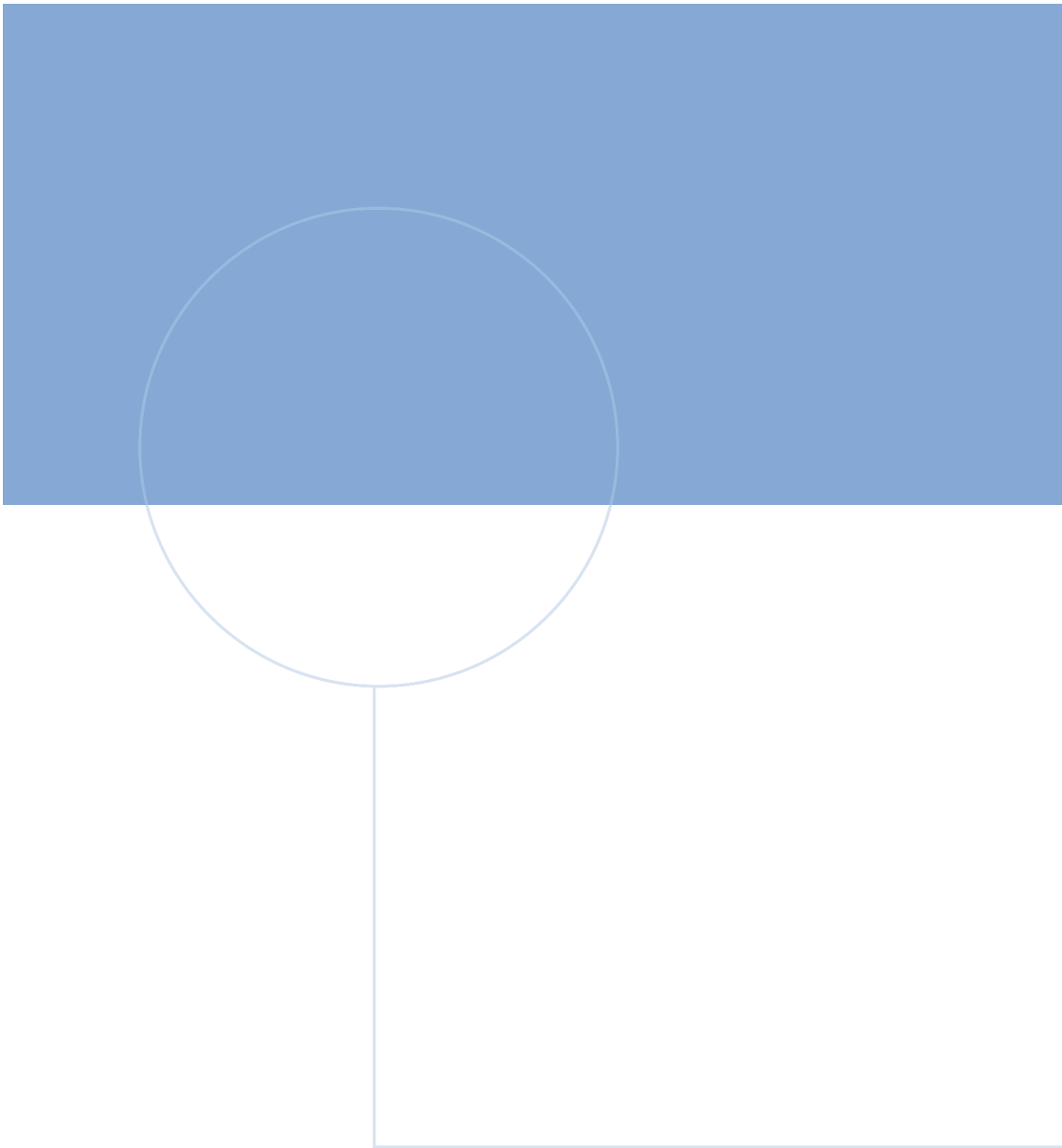
[45] K. Ren, A. Thomson, and D. J. Abadi. An evaluation of the advantages and disadvantages of deterministic database systems. *Proceedings of the VLDB Endowment*, 7(10):821–832, 2014.

[46] M. Stonebraker. Newsql: An alternative to nosql and old sql for new oltp apps. *Communications of the ACM. Retrieved*, pages 07–06, 2012.

[47] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, et al. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1493–1509, 2020.

[48] A. Thomson and D. J. Abadi. Modularity and scalability in calvin. *IEEE Data Eng. Bull.*, 36(2):48–55, 2013.

[49] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12, 2012.



NTNU

Norwegian University of
Science and Technology