MASTER'S THESIS 2022

Utilizing highly synchronized clocks in distributed databases

Jacob Gunnarsson, Fabian Lindfors

ISSN 1650-2884
LU-CS-EX: 2022-37

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2022-37

Utilizing highly synchronized clocks in distributed databases

Att utnyttja högsynkroniserade klockor i distribuerade databaser

Jacob Gunnarsson, Fabian Lindfors

Utilizing highly synchronized clocks in distributed databases

(a case study on CockroachDB)

Jacob Gunnarsson
ja7180gu-s@student.lu.se

Fabian Lindfors
fa5518li-s@student.lu.se

June 20, 2022

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Flavius Gruian, flavius.gruian@cs.lth.se

Examiner: Emma Söderberg, emma.soderberg@cs.lth.se

Abstract

Relying on clocks in distributed systems has long been seen as a convenient way of ordering events but also a challenge because of the inevitable clock skew. In recent years, the availability of highly synchronized clocks has improved which enables all new innovations and system designs.

In this thesis, we investigate how the distributed database CockroachDB can be adapted to utilize highly synchronized clocks. We implement our findings and evaluate the performance impact of our modifications using a custom benchmark designed to emulate a workload with long reads across multiple nodes with contending writes.

Our results show significant performance improvements for our workload, with median latency being reduced by up to 47% for reads and 43% for writes. We conclude that our changes might make worthwhile additions to CockroachDB but further research will be necessary to understand if they will be benefitial for production workloads.

Keywords: distributed systems, databases, consistency, highly synchronized clocks

Acknowledgements

We would like to sincerely thank Erik Grinaker at Cockroach Labs for his assistance throughout the thesis. Without his deep knowledge of CockroachDB and distributed systems coupled with his eagerness to help, this project would not have been within our reach. We would also like to thank our supervisors, Flavis Gruian and Jörn Janneck, for their thorough feedback and guidance.

Contents

1 Introduction **7**
1.1 Research questions 8
1.2 Scientific contribution 8
1.3 Individual contributions 9

2 Background **11**
2.1 Consistency 11
2.2 Atomic clocks and GPS synchronization 12
2.3 Spanner and TrueTime 13
2.4 Amazon Time Sync Service and ClockBound 13
2.5 CockroachDB 14
2.5.1 Necessary uncertainty restarts 15
2.5.2 Unnecessary uncertainty restarts 16

3 Approach **19**
3.1 Literature study 19
3.2 Method 20
3.3 Implementation 21
3.3.1 TrueClock 21
3.3.2 CockroachDB 21
3.3.3 Verification of implementation 23
3.4 Experimental setup 23
3.4.1 Benchmark evaluation 23
3.4.2 Benchmark design 24
3.4.3 Benchmark setup 27
3.4.4 Measuring timestamp bounds sizes 28

4 Experimental validation **29**
4.1 Read-heavy workload 29

4.2 Write-heavy workload 32
4.3 Timestamp bounds sizes 35

5 Discussion **37**
5.1 Interpreting results 37
5.2 Limitations and costs 39
5.3 Validity threats 40
5.4 Future work 42

6 Conclusion **45**

References **47**

Appendix A Benchmark database schema **53**

Appendix B Timestamp bounds size test setup **55**

Appendix C Read-heavy benchmark results **59**

Appendix D Write-heavy benchmark results **63**

Appendix E Timestamp bounds size test results **67**

Chapter 1

Introduction

The idea that you can not trust clocks has a long history in distributed systems. As Lamport established all the way back in 1978, ordering events is a common problem for distributed systems and one which could be fixed with perfectly synchronized clocks, if they did exist. In practice, clocks can not be perfectly synchronized which has led to many innovations to counter their imprecision. To fix this, Lamport proposed the *Lamport clock*, a form of logical clock which can be used to establish ordering in some cases without the need for real clocks. His invention later earned him a Turing Award and lead to the development of *Vector clocks* about a decade later, designed to strengthen the guarantees offered by Lamport clocks. Innovation continues to this day, for example with the *Hybrid Logical Clock* introduced in 2014 which combines a wall clock with a logical clock. This kind of clock allows timestamps to be ordered internally while also allowing values to be queried by their timestamps, for example to find all rows in a database created on a specific day.

What these innovations all have in common is that they either replace the need for physical clocks or augment them to hide their inherent uncertainty. In recent years, the industry has started approaching the problem from the other end by improving clock synchronization enough to make them dependable. This was the approach Google chose for their globally distributed database Spanner. They installed specialized hardware and software in their own datacenters to dramatically lower the clock skew and also to calculate error bounds on all clock readings. Thanks to this, Spanner achieved the strongest level of consistency with high transactional performance, using only physical clocks.

As Spanner relies on specialized hardware, its innovations are out of reach for commodity databases, but in recent years this has started to change. AWS introduced their Time Sync Service which puts highly synchronized clocks in the hands of all applications deployed to AWS. The Open Compute Project has also been working on establishing standards and sharing technology around clock synchronization. As part of this project, Facebook has for

example open-sourced the design of a plug-in card for precise clock synchronization, built around an onboard GPS receiver and atomic clock. One database that stands to gain from these developments is CockroachDB, a distributed relational database that provides strong consistency guarantees. CockroachDB was conceived by former Google engineers and largely inspired by Spanner but with one crucial difference, it was designed to run on commodity hardware and hence could not rely on highly synchronized clocks. Instead, it uses regular clocks but takes measures to retry transactions where clock skew may impact consistency.

1.1 Research questions

For this thesis, we wanted to investigate how distributed databases can be adapted to utilize highly synchronized clocks and what impact it has on performance. In particular, we conducted the thesis as a case study on CockroachDB because it's source-available, allowing us to modify it freely, and because weakly synchronized clocks play a central role in its design. On top of investigating how CockroachDB can be adapted, we also wanted to implement these changes and see what effect they have on performance.

In summary, we aimed to answer the following research questions:

Q1 How can CockroachDB be adapted to utilize high-precision clock synchronization?

Q2 What effect does this have on CockroachDB's transactional performance?

1.2 Scientific contribution

The belief that clocks can not be relied upon is quickly becoming outdated as highly synchronized clocks become more commonplace, and we hope our work can help in the building of next-generation distributed systems. In particular, we believe our thesis contributes to this young research area in two ways.

The first is our literature study which investigated the current state of distributed databases and clocks, and resulted in two proposed ways of adapting CockroachDB for highly synchronized clocks. The study was conducted with a particular focus on the industry and what is achievable in production systems, which we hope can serve as a useful reference for future distributed systems development. Even though this thesis was conducted as a case study, we believe our learnings are applicable to other databases and distributed systems, both existing and yet to be invented.

The second way is the data from our experiments. During our research, we could not find any good data on how well Amazon Time Sync Service and ClockBound performs, so we built and ran our own test. We hope the data from this experiment can help inform researchers and developers what level of clock synchronization is available today for production systems. The biggest contribution of this thesis though is the benchmark we performed on CockroachDB modified to utilize highly synchronized clocks. We hope the data from our benchmarks can offer valuable insight into when highly synchronized clocks are useful and what can be gained from designing distributed systems around them.

1.3 Individual contributions

The authors have collaborated on all aspects of this thesis, including writing, coding, and reviewing literature. Some parts of the work was split up with each author focusing on different things, but in those cases the work was always discussed between us and reviewed. In general, Fabian placed a larger focus on implementation and coding, for example modifying CockroachDB, porting ClockBound, implementing the custom benchmark and creating the timestamp bounds size experiment. Jacob, on the other hand, stayed more focused on benchmarking, including researching and evaluating potential benchmarks, running and debugging benchmarks and analyzing results.

Chapter 2

Background

In this chapter, we present the findings of our literature study, on which this thesis is based. We start with the concept of consistency in distributed databases and how clocks could be used to achieve it if they were perfectly synchronized. After that we briefly explore how clock synchronization can be improved with specialized hardware before moving onto Spanner, an example of a proprietary database built to use such hardware in order to achieve the strongest level of consistency. From there, we get to Amazon Time Sync Service which enables precise clock synchronization for all services running on AWS. This opens the possibility for databases designed to run on commodity servers to rely on highly synchronized clocks, just like Spanner. Finally, we take a close look at one such database, CockroachDB, which is the subject of our case study. We explore the techniques used by CockroachDB to achieve strong consistency using only weakly synchronized clocks, how clock synchronization affects performance and the downsides of their approach.

2.1 Consistency

In a distributed system, data is often replicated across several nodes for reliability and scalability. This has implications for the consistency of the data. During writes, the update may reach different nodes at different times causing them to have different views of the current value. Any reads that occur simultaneously may get inconsistent values depending on which node serves the read. Distributed databases can offer different guarantees for consistency where one of the weaker guarantees is *eventual consistency*, which is used by distributed databases like Cassandra [9] and Amazon DynamoDB [33]. With eventual consistency, all nodes are guaranteed to eventually serve the same value for reads if all writes stop [21], but there are no guarantees for how long convergence takes. The weak guarantees provided by eventual consistency places a larger burden on application developers as they need to design

applications that can deal with inconsistencies [7].

As an example of this, let's say we are building an application where users can make posts that are either public or private. The visibility of a post is determined based on a setting on the user profile, which is read when the post is created. If our application is backed by a database that offers *eventual consistency* the user risks seeing some unexpected behaviour. For example, if a user wants to make a private post, they'll first update their settings and then write and create their post. Our application will check the current setting when creating the post, but because of eventual consistency, there is no guarantee that we'll actually observe the previous change and the post might end up inadvertently public.

To avoid having to design around situations like these we need stronger consistency guarantees, and the strongest possible one is *linearizability*. If a system guarantees linearizability, reading from two different nodes will always yield the same, most recent value. From the perspective of the application, the system appears to have only a single copy of the data and when performing a write, there must be some point in time during the write at which the update is atomically applied. Without this guarantee, values could flip back and forth if they are read at the same time as a write runs, which can happen in an eventually consistent system [21].

To achieve linearizability, a total ordering must be established for all operations in order to determine which operation precedes another and to know which is the most recent write for a value. One way to achieve that is to use a real-time clock to assign timestamps to operations [21]. This requires perfectly synchronized clocks across all nodes though which is not attainable, hence any distributed system that relies on synchronized clocks also needs to handle clock skew between nodes [21]. The amount of clock skew depends on a lot of different factors [21] and for systems running in public clouds, measurements show that when sampling clock skew the 99th percentile of samples can reach an average value of 60 milliseconds across three nodes [3].

2.2 Atomic clocks and GPS synchronization

Although perfectly synchronized clocks are impossible to achieve, low clock skews are attainable using specialized hardware and software. One way to achieve low clock skew is to utilize the Global Positioning System, more commonly knows as GPS [36]. As the name indicates, the main purpose of GPS is to localize devices across the world which is achieved by measuring the distance from a device to a number of GPS satellites. All GPS satellites contain highly synchronized clocks and by reading the current time from at least three satellites, a device can calculate the distance to each one using the speed of light and known signal delays. Although the positioning feature of GPS is highly useful on its own, GPS also has the added benefit of providing a global, highly accurate clock synchronization source for distributed systems [36].

In the periods between synchronizations, clock skew will grow as the local clocks at each node do not run at the same rate. This *clock drift* puts a lower limit on the attainable skew [21]. The quartz clock in a computer runs at a constant rate but the rate differs between clocks and is sensitive to environmental factors, where even small differences can accumulate to large

drifts [35]. An alternative is to utilize atomic clocks which run at much more consistent rates but are larger and more expensive than quartz clocks [31].

2.3 Spanner and TrueTime

One company which has employed atomic clocks and GPS synchronization to manage clock skew is Google. By installing the specialized hardware in their datacenters they were able to lower the clock skew enough to build Spanner, a distributed SQL database which achieves linearizability using real-time clocks [13]. This is made possible by a novel API called TrueTime which generates timestamps with guaranteed bounds on clock skew. The specialized hardware keeps the bounded clock skew below 7 ms with an average of 4 ms in production [13].

The TrueTime API consists of a single basic operation, $now()$, which returns two timestamps that form an interval. The actual time is guaranteed to lie within that time interval. Thanks to this invariant, it is possible to determine a causal ordering by comparing time intervals. Given two intervals, $T_1 = \left[t_1^{min}, t_1^{max}\right]$ and $T_2 = \left[t_2^{min}, t_2^{max}\right]$, we can determine that T_2 occurred after T_1 if $t_2^{min} > t_1^{max}$ and vice versa [13].

Spanner utilizes these bounded timestamps to establish an ordering for its transactions. Every transaction is assigned a bounded timestamp when it has acquired all the locks it needs to operate. After performing all operations, the leader uses a method which Spanner refer to as commit-wait [13]. It must wait until t^{max} of the transaction's timestamp has passed before committing and releasing its locks. As t^{max} provides an upper bound for timestamps generated across all nodes, it is guaranteed that the changes made by the transaction will be visible across the entire cluster, achieving linearizability [21]. This also means that the latency for write transactions is directly affected by the clock skew, which makes highly synchronized clocks crucial to achieving good performance [13].

Although Spanner uses locks for read-write transactions, it can perform strongly consistent read-only transactions without any locking. This is made possible using snapshot reads and TrueTime. When a read transaction is started, it is assigned a timestamp as the current t^{max} and the read is then performed on all nodes that contain data of interest. Every node will return data as a snapshot at time t^{max}. Writes committed after that will not be considered. This ensures strong consistency for read-only transactions without locking [13].

2.4 Amazon Time Sync Service and ClockBound

TrueTime is internal to Google and only usable by Google's own cloud services. This means that commodity databases can not make use of the highly synchronized clocks and bounded timestamps. AWS has released their own service to provide highly synchronized clocks across their datacenters, called Amazon Time Sync Service. This service is implemented similarly to TrueTime with redundant GPS synchronized atomic clocks in datacenters [13]. The main

difference is that AWS allows access to these highly synchronized clocks on all EC2 virtual machines [4].

Amazon Time Sync Service does not on its own provide an alternative for TrueTime as it only provides synchronized clock readings and not bounded timestamps [4] [13]. AWS has released a complementary open-source tool called ClockBound which implements the same API as TrueTime [5], allowing applications to generate bounded timestamps backed by Time Sync Service. ClockBound runs as a daemon on each server which applications can communicate with using a custom protocol over UNIX datagram sockets. The timestamps are 64-bit unsigned integers representing the number of nanoseconds since January 2st 1970 (the Unix epoch) [34].

ClockBound works by continuously tracking the maximum possible clock skew which the system clock may currently see [34]. This is made possible by Chrony, a tool for synchronizing the system clock against external sources, such as NTP servers or reference clocks. Chrony not only synchronizes the system clock but also tracks metadata, for example estimates of clock skew and network delays [10]. ClockBound uses this metadata, combined with an estimate for the local clock's drift, to form error bounds for the system clock and based on that generates bounded timestamps [34]. Because ClockBound only requires Chrony to operate and Chrony can use many synchronization sources, it's possible to run ClockBound without Amazon Time Sync Service. With this setup, it would still be possible to generate bounded timestamps but without good synchronization, the bounds will be greater. Unfortunately, we were not able to find any numbers on what level of clock skew to expect when using Amazon Time Sync Service. Because of this, we opted to run our own tests as described in section 3.4.4 with the results presented in section 4.3.

2.5 CockroachDB

Unlike Spanner, CockroachDB does not rely on any specialized hardware. This allows it to run on commodity servers in public as well as private clouds which usually relies on software-level clock synchronization, such as the Network Time Protocol (NTP) [14].

CockroachDB's reliance on wall time without specialized hardware means it must be able to handle a large amount of clock skew. The commit-wait method Spanner uses to achieve strong consistency can only obtain reasonable performance with tightly synchronized clocks and guaranteed bounds on skew, as provided by TrueTime [13]. To enable strong consistency, CockroachDB instead detects when a transaction can not order itself among other transactions that recently occurred. These conflicts are detected by using a static *max_offset* value which is a pessimistic upper bound on the clock skew any node might see. The default value for *max_offset* is 500 milliseconds but can be manually set by the operator [14]. As long as the actual clock skew does not exceed *max_offset*, CockroachDB offers linearizability for reads and writes which operate on the same keys [14]. This is also called causal consistency and offers slightly lower guarantees than full linearizability, which implies a total ordering of all operations across the database. With causal consistency, if we have two transactions with non-overlapping read and write sets then a third observer could see their writes in either order. According to Kleppman though, most systems only require causal consistency and a global ordering is not necessary [21].

Each value stored in the database is associated with the commit timestamp of the last transaction that wrote the value. During execution of a transaction, each value read is checked against the transaction's commit timestamp. If the value timestamp is later than the transaction's, then there is a risk that the transaction that wrote the value actually precedes the current transaction in real-time but clock skew gave it a later timestamp. As *max_offset* sets an upper bound on the clock skew, CockroachDB checks for $T_t < T_v < T_t + max_offset$ where T_t is the current transaction's timestamp and T_v is the value timestamp. If T_v falls within this interval, known as the *uncertainty interval*, an *uncertainty restart* is initiated, forcing the current transaction to restart with a new timestamp ahead of the one encountered [14]. The implication of this is that all values whose timestamps reside within the uncertainty interval are treated as past writes. Uncertainty restarts can be split into two categories and we will look at both in detail: *necessary restarts*, which occur due to actual clock skew between nodes, and unnecessary restarts, which occur due to latencies in the system.

2.5.1 Necessary uncertainty restarts

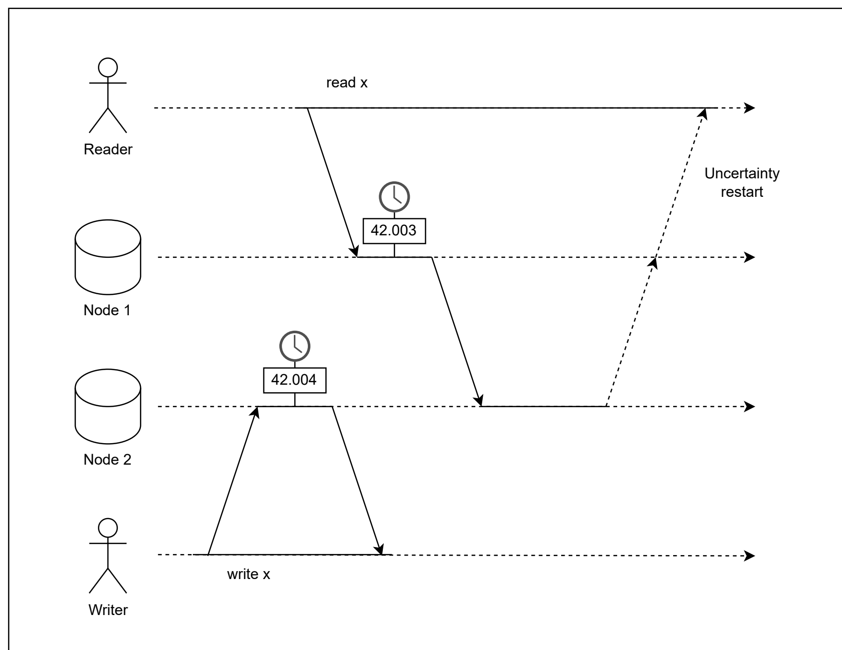

Figure 2.1: Clock skew causing uncertainty restart

Figure 2.1 depicts a very simplified example of how a necessary uncertainty restart may occur in the face of clock skew. Initially, a write to key x is issued to node 2, causing it to start a transaction and assign it the provisional timestamp 42.004 based on node 2's local clock. Every key in CockroachDB has a single leaseholder node which is the only node that can read or write to that key. In this case, the leaseholder node for key x is node 2 so as the write arrives, it attempts to write key x and successfully commits, pushing the timestamp when x was last modified to 42.004. Moments later, a read for key x is issued to node 1 which starts a transaction with read timestamp 42.003 from node 1's clock. Even though this operation happens after the previous write, it receives an earlier timestamp because of clock skew between node 1 and 2. Next, node 1 finds out that node 2 is the leaseholder for x and sends

a read request. When the value x is read from node 2, CockroachDB detects that its write timestamp is later than the current read timestamp and falls within the uncertainty interval. Since CockroachDB can't tell if the write timestamp is later because it actually occurred later in real-time or if it occurred earlier in real-time but got a later timestamp because of clock skew, the read must be retried with a higher timestamp to maintain consistency. We call these necessary restarts as consistency will be broken if the transaction is not restarted.

An important aspect of necessary restarts is that they are only dependent on the actual clock skew and not the size of the uncertainty intervals, as defined by *max_offset*. This is the case because a larger amount of clock skew increases the risk for a transaction to get a later timestamp than another transaction which occurred after the first in real-time. The effect of clock skew on uncertainty restarts has been studied previously by Geng et al. when they evaluated a novel clock synchronization method. In their study, they showed that lowering the actual clock skew, without changing *max_offset*, reduced the amount of uncertainty restarts [15].

2.5.2 Unnecessary uncertainty restarts

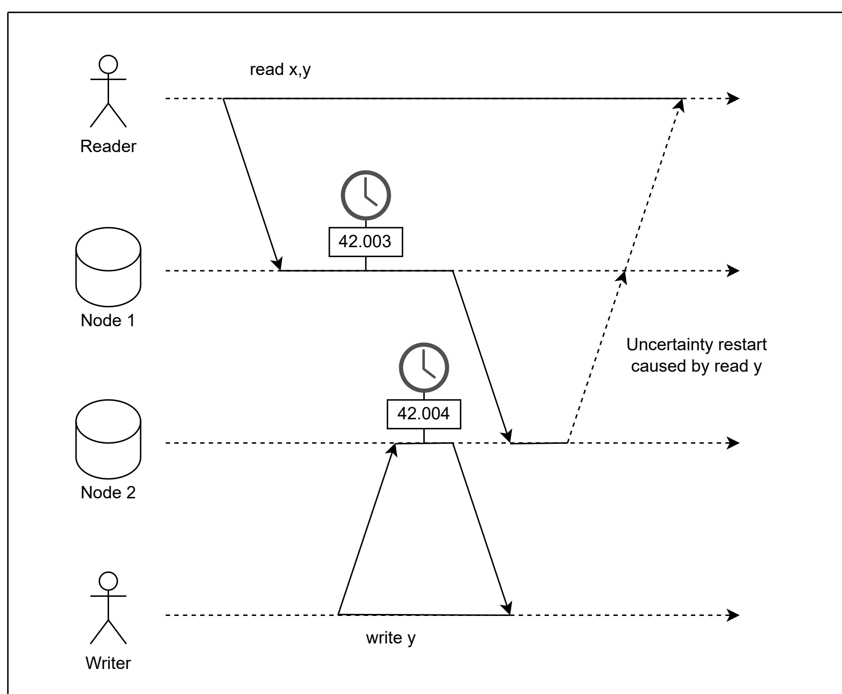

Figure 2.2: Write causing unnecessary uncertainty restart

Clock skew is not the only thing that can cause restarts, different forms of latency across the system can also cause writes to be placed inside the uncertainty interval of a read, which restarts the read. We call these unnecessary uncertainty restarts as the ordering of the timestamps matches the real-time ordering and hence no restart is necessary to ensure consistency.

Figure 2.2 shows an example of such a situation. In this scenario, node 1 is leaseholder for the range containing x and node 2 for key y. A read for keys x and y is issued to node 1. Node 1 initiates the transaction, assigning it a timestamp of 42.003 along with an uncertainty interval. As node 1 is leaseholder for the range containing x it fetches that value. It also sees

that node 2 is the leaseholder for the range containing key y and sends a request to read from it, which takes some time because of network latency. Later in real time, a write to key y is issued to node 2. As it is the leaseholder for that range, it successfully writes to key y with the timestamp 42.004 and commits momentarily before the other read request arrives. When the read request arrives, it detects that a write has occurred for that key inside its uncertainty interval and hence can not determine if the write actually happened before the read started but was assigned a later timestamp because of clock skew. Because of this ambiguity, the read must be restarted to not break consistency. If the *max_offset* had been low enough for the write to fall outside the uncertainty interval, then the read could be certain that the write actually started after itself, making it safe to ignore. These unnecessary restarts can be caused by any latency impacting a read operation, including if the read operations needs to read a lot of data which spans multiple nodes.

Chapter 3

Approach

In this chapter, we walk through how we conducted this thesis. We start by detailing our literature study in section 3.1, which leads us to the method, presented in section 3.2, used to adapt CockroachDB for highly synchronized clocks. In section 3.3, we detail the actual changes made to CockroachDB and how we verified their correctness. Finally, in section 3.4 we explain the experimental setup we used to measure the impact of our changes, which included both a benchmark of database performance and a test of the level of clock synchronization provided by Amazon Time Sync Service and ClockBound.

3.1 Literature study

To start this thesis and lay the groundwork for answering **Q1**, we conducted a literature study. A central part of this was the book *Designing Data-Intensive Applications* by Martin Kleppman [21], which we used as a reference on transactions, consistency, clocks and other relevant concepts. We also followed the books citations to dive deeper on these topics. On top of this, we performed keyword searches on Google Scholar based on what we noted from Kleppmans books. This proved especially helpful to find papers and books on topics that Kleppman does not offer as much detail on, such as clocks and clock synchronization.

Another central part of our literature study was the help of Erik Grinaker, an engineer at Cockroach Labs, the company behind CockroachDB. He helped us make sense of CockroachDB's documentation and design. He also pointed us towards other relevant things, such as previous studies on CockroachDB and clock synchronization.

3.2 Method

As mentioned in chapter 2, CockroachDB and Spanner use different transactional models to achieve strong consistency. Spanner uses a commit-wait approach, meaning it waits out the uncertainty for all writes. Since Spanner has access to highly accurate atomic clocks and bounded timestamps, it can keep the time it needs to wait low. With Amazon Time Sync Service [4] and ClockBound [5], tight, bounded timestamps are made available to commodity databases like CockroachDB, meaning it would be feasible to change CockroachDB to use a commit-wait model as Spanner does, removing the need for uncertainty restarts. This would improve consistency, making all operations linearizable rather than just ones with overlapping read and write sets. It also has the potential of improving performance, especially for read-heavy workloads where uncertainty restarts can cause increased latency for transactions. The drawback would be potentially increased latency for write operations as each operation would have to wait out any clock uncertainty. This also means that the write latency would be directly affected by any variation in clock skew.

Another approach would be to improve the precision of the uncertainty intervals that CockroachDB uses to detect potential consistency issues. In section 2.5, we mentioned that CockroachDB uses a statically configured value *max_offset* for the upper bound on the uncertainty interval. Since the value is static, it must be set to handle the worst case clock skew at any point, which makes the uncertainty intervals much larger than the average clock skew [14]. It also places a burden on the operator to understand the level of clock synchronization achievable and select the lowest possible *max_offset* based on that, unless they decide to not optimize and use the default value. With ClockBound, we can get bounded timestamps that provide an upper bound on any clock skew and which are dynamically recalculated as the level of clock synchronization changes. By replacing the static *max_offset* with a dynamic value from ClockBound, we can potentially make the uncertainty intervals shorter on average without any manual configuration by the operator. As the sizes of the bounded timestamps should always be lower than the actual clock skew at any point in time, consistency will also be maintained. Combining this with highly synchronized clocks, like Amazon Time Sync Service, the uncertainty intervals could potentially become very short. Smaller uncertainty intervals have potential to improve performance by reducing the risk of unnecessary uncertainty restarts, as we saw in section 2.5. As we also saw in figure 2.2 unnecessary restarts are especially relevant for workloads with lengthy reads across multiple nodes along with contending writes. An uncertainty restart causes increased latency for restarted read operations, which naturally has a greater impact on performance when reads are already lengthy [20]. Necessary restarts however will not be affected though as they are a result of actual clock skew, not the size of uncertainty intervals.

After concluding our literature study, we met with Erik Grinaker from Cockroach Labs. Together with him, we discussed the two previously mentioned methods for adapting CockroachDB to better utilize highly synchronized clocks. The commit-wait approach ended up being discarded as the changes required were determined as too big for this project. We instead settled on attempting to make the uncertainty intervals dynamic and reduce their length, hypothesising that tightening the uncertainty intervals would improve transactional performance. To test our hypothesis, we forked and altered CockroachDB to make use of Amazon Time Sync Service and ClockBound to generate bounded timestamps. These

bounded timestamps were then used to form the uncertainty interval, replacing the static *max_offset* setting.

3.3 Implementation

Here we detail the implementation work we did to adapt CockroachDB as described in section 3.2. We start by discussing our efforts to integrate ClockBound in section 3.3.1, which ended with us porting ClockBound from a daemon to a library. Next, in section 3.3.2, we detail how we modified the CockroachDB source code to make use our ClockBound port to form dynamic uncertainty intervals.

3.3.1 TrueClock

Our initial plan was to use AWS ClockBound to generated bounded timestamps. ClockBound uses a client-server model with a daemon running in the background that the application communicates with over UNIX datagram sockets and a custom protocol [34]. After implementing a client library in Go and integrating it with CockroachDB, we noticed that the added latency from performing a datagram request had a significant impact on the latency of database operations. Our testing showed that retrieving a single bounded timestamp from ClockBound took around 50 µs, compared to around 60 ns for a system clock reading that standard CockroachDB uses. Given that a single operation in CockroachDB results in multiple clock readings, the added latency became significant.

To fix this issue, we ported the open-source ClockBound daemon to a Go library which allowed us to include it directly in CockroachDB as it is also written in Go. This in turn removed the need for a datagram request for each clock reading. We ended up calling the library *TrueClock* and released it as an open-source project on GitHub [30]. Just like ClockBound, TrueClock uses a background thread to periodically retrieve clock synchronization data from Chrony which is stored as a shared variable protected by a mutex. When the library is called to make a clock reading, it reads the system clock and uses the latest stored data from Chrony to determine the maximum possible skew and drift which are added together to form error bounds for the system clock reading. By integrating it as a library, we were able to bring the latency of a single clock reading down to 250 ns, low enough to not have a significant impact on database operations while yielding the same results as ClockBound.

3.3.2 CockroachDB

In CockroachDB, forming the uncertainty interval for a transaction involves multiple different parts of the codebase. It starts with a transaction being created, at which point a clock reading is performed to determine the current time. The current timestamp is stored as the transaction's read timestamp and then another timestamp is created by adding *max_offset* to the read timestamp. This new timestamp is called the *global uncertainty limit* and forms the upper bound of the transaction's uncertainty interval. Clock readings are handled by a singleton structure called *hlc.Clock*, which returns a timestamp stored as a different structure called *hlc.Timestamp*. To implement our changes, we needed to update this flow to instead make use

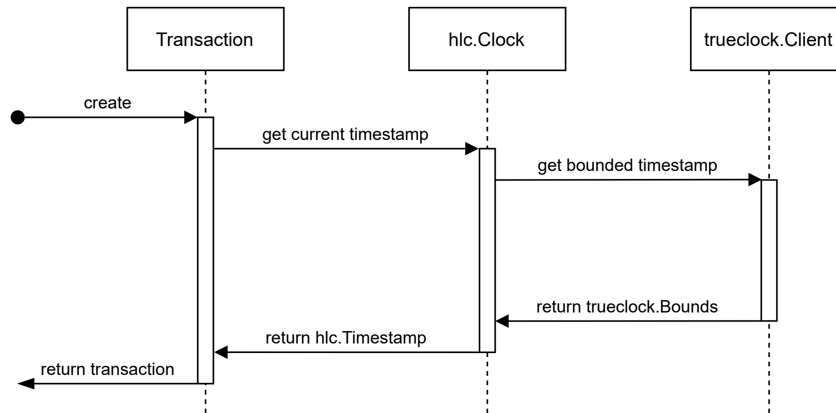

Figure 3.1: Interaction with TrueClock when a transaction is created

of TrueClock when forming the uncertainty interval for a transaction. Figure 3.1 shows a sequence diagram describing the transaction creation process after our modifications.

The first change we had to do was update the *hlc.Timestamp* structure to store not just a single clock reading, but a bounded timestamp with a lower and upper bound. Many structures in CockroachDB, including timestamps, are built using the serialization format Protocol Buffers (Protobuf), allowing them to be exchanged between nodes over a network. To add support for bounded timestamps, we added a new field to the Protobuf definition for *hlc.Timestamp* called *WallTimeUpperBound*, serving as an upper bound on the current time. The existing *WallTime* field was reused to be the lower bound. CockroachDB also has another structure, *hlc.LegacyTimestamp*, which contains the same fields as a regular timestamp but uses a different encoding. The two different timestamp types can be converted between each other so we also added the new field to the legacy model and updated the conversion code.

Next, we needed to update *hlc.Clock*, which normally serves as a wrapper around the system clock, to instead generate bounded timestamps from TrueClock. We updated *hlc.Clock* to store an instance of *trueclock.Client*, as provided by our TrueClock library described in section 3.3.1. This client was then used to retrieve a bounded timestamp in the shape of a structure called *trueclock.Bounds*, containing an upper and lower bound timestamp. From this, we generate and return a *hlc.Timestamp*. For these timestamps, *WallTime* was set to the lower bound and *WallTimeUpperBound* to the upper bound.

The final part of the changes consisted of updating the transaction creation to use the timestamp bounds rather than the static *max_offset* to form uncertainty intervals. If *now* is the current timestamp, then the current implementation uses *global uncertainty limit* = *now* + *max_offset* and *read timestamp* = *now*. To make use of our new bounded timestamp, which consists of two timestamps *earliest* and *latest*, we updated the code to use *global uncertainty limit* = *latest* and *read timestamp* = *earliest*.

Our fork of CockroachDB with the modifications described above is available on GitHub [29]. This also includes our custom benchmark described in section 3.4.2, which is implemented in CockroachDB's workload tool.

3.3.3 Verification of implementation

By changing the way CockroachDB sets its uncertainty intervals, we risk breaking consistency if our changes are incorrect or if our port of ClockBound does not calculate accurate error bounds. To ensure our changes were correct and did not affect consistency, we ran the automated Jepsen tests that CockroachDB uses internally with the help of Erik Grinaker [18]. Jepsen is an open-source testing framework for verifying different aspects of distributed systems, including consistency guarantees [17]. Normally, Jepsen will try to take control over the system clock in order to simulate clock skew, which it calls a nemsis. In our case, that would remove the ability for TrueClock and Chrony to function and as we were also interested in the correctness of our TrueClock bounds, we opted to disable the nemesis. After running the tests, Jepsen reported that consistency was maintained which verified that our changes were correct.

3.4 Experimental setup

In this section, we detail how we evaluated the performance impact of our changes. In section 3.4.1, we detail the work we did in evaluating existing benchmarks and how it led us to design our own benchmark. The design of our own benchmark is then laid out in section 3.4.2 followed by a description of how the benchmark was run in section 3.4.3. Finally, in section 3.4.4, we describe another smaller experiment we performed to determine what size to expect for our uncertainty intervals.

3.4.1 Benchmark evaluation

Our initial approach to evaluating the impact of our modifications was to find an established benchmark designed to emulate a real life workload. The first benchmark we found which we thought would be suitable was TPC-C, the most widely used online transaction processing (OLTP) benchmark in both industry and academia [37]. The benchmark itself consists of a set of operations designed to replicate a complex OLTP environment [12]. It simulates a wholesale supplier where the workload consists of managing, selling and distributing a product or service. The transactions within the benchmark are both reads and writes of varying complexity. They involve everything from customers making orders, checking the status of already existing ones, entering payments from customers, processing orders for delivery and examining stock levels to identify potential supply shortages. The official measurement of performance is done in transactions per minute (tpmC) [12]. The fact that TPC-C has a diverse set of transactions with risk of contention led us to believe it could suffer from unnecessary uncertainty restarts. This together with the fact that CockroachDB has a full TPC-C implementation built into its workload tool, and publishes guides on how to run a full benchmark on AWS, made it a good choice for us [24]. We ran a small 30-minute test of TPC-C against a standard CockroachDB cluster to evaluate the benchmark and saw around 350 restarts. We deemed this as too few for our changes to have any significant impact on performance which we confirmed by running a small test of TPC-C against a cluster with our modifications. This test showed no impact on performance from our changes, neither positive or negative. From this we concluded that the workload TPC-C emulates does not

represent the sort that is most affected by unnecessary uncertainty restarts, the kind our changes would be most valuable for.

We also had a look at and tested Yahoo Cloud Serving Benchmark (YCSB), another widely used database benchmarking tool [11]. YCSB takes a different approach than TPC-C. Instead of trying to emulate a large, realistic workload, it consists of multiple smaller workloads which are artificially designed to test one aspect of a database. Out of the available workloads, we determined that two were of particular interest. The first one was the "Read mostly workload", which consists of a set of records which are read and written to at a ratio of 95% reads and 5% writes [8]. These operations carry a risk of read-write contention meaning there is a risk for uncertainty restarts. However, as these read operations are short, reading only single records, the probability of a conflict is small. The second one was the "Read-modify-write workload", where a record is read, modified and then written back [8]. This workload also runs the risk of read-write contention and the transactions also run for longer as the value needs to be sent back to the client and modified. We hypothesised that these workloads could be susceptible to unnecessary restarts but after running some small exploratory tests, we did not see a significant number occurring. Just like for TPC-C, we also ran small tests comparing standard CockroachDB with our modified version and saw no positive or negative difference in performance. Based on this we concluded that neither TPC-C nor YCSB represented the sort of workloads that were susceptible to unnecessary restarts.

After evaluating TPC-C and YCSB, we switched approach to instead determine what sort of workload might gain most from our changes and work from that. As mentioned in section 3.2, a workload with long-running reads across multiple nodes and contending writes should be highly susceptible to unnecessary uncertainty restarts. We decided to design our own specialized benchmark to emulates this sort of workload, akin to the YCSB approach. This introduces questions surrounding the validity of our results, which we discuss in section 5.3.

3.4.2 Benchmark design

The benchmark we designed emulates a simple social media application with posts and likes. We chose this setup for simplicity and to tie back to the example given in section 2.1, although it does not necessarily represent the kind of application where consistency is key. The most important property of our setup is the form of workload it represents, which is not bound to any application in particular.

Figure 3.2: Overview of the benchmark database

As depicted in figure 3.2, our benchmark uses two tables, posts and likes, where each post has a title and each like belongs to a post. The full code for the database schema is available in appendix A. At the start of the benchmark, 100 posts are generated with random titles and 1000 likes are created, randomly assigned to posts with a uniform distribution. For the

actual benchmark, there are K workers running in parallel with each worker performing one operation at a time. The two possible operations are *read* and *like* and when a worker is going to run a query, it randomly selects one of them with p being the probability that it will be a read. K and p are the two parameters we control where K lets us vary the contention on our data and p the balance between reads and writes.

Before running our benchmarks, we performed some tests and tried different number of workers, K, to find a reasonable interval for our benchmarks. We looked for values which let us try a nice variation of latencies whilst keeping them reasonable, no larger than a few seconds. Based on this testing, we settled on starting at 50 workers and going up to 500 workers at steps of 50. For p, the balance between reads and writes, we decided to use two different values to represent a read-heavy and a write-heavy workload. The reason we decided this is that uncertainty restarts are a result of read-write contention and hence we wanted to see what happens when the balance between the two changes. For the read-heavy workload we used 95% reads and 5% writes, and for the write-heavy workload we used 50% reads and 50% writes. These percentages were selected as they are the same used by the industry standard YCSB benchmark [11]. We ran our benchmark for every combination of K and p. Each test started with a 1-minute ramp up to give CockroachDB time to initialize, followed by 5 minutes of benchmarking where data was collected. This resulted in a total of 20 benchmarks.

```
1 SELECT posts.id, posts.title, COUNT(*)
2 FROM posts
3 LEFT JOIN likes ON posts.id = likes.post_id
4 GROUP BY posts.id
5 ORDER BY COUNT(*) DESC
```

Listing 3.1: Read query

```
1 INSERT INTO likes(post_id)
2 VALUES ($1)
```

Listing 3.2: Like query

The read operation represents a user loading the front page of the application which shows all 100 posts sorted by the number of likes. The query for this is shown in listing 3.1 and requires reading all posts and all likes. As the query performs a join using the post ID, we added an index for the *post_id* column in the *likes* table. The like operation adds a new like to a randomly selected post and its query is shown in listing 3.2. These are the only two queries performed during our benchmark, meaning that posts and likes are never removed or altered after creation.

CockroachDB stores all data in ranges and attempts to keep each range below 512 megabytes. If a range grows larger than this limit, it is automatically split into two new ranges. This mechanism becomes a problem for our benchmark since our data is small and will not be automatically split, instead ending up in a single range per table. As only the leaseholder for a range can serve reads and writes, having only a single range would utilize only a single node, leaving unused capacity in the cluster. A solution suggested by CockroachDB, which is depicted in figure 3.3, is to manually split the tables consisting of single ranges evenly into three ranges and then assign a lease to every node, spreading the load evenly across the

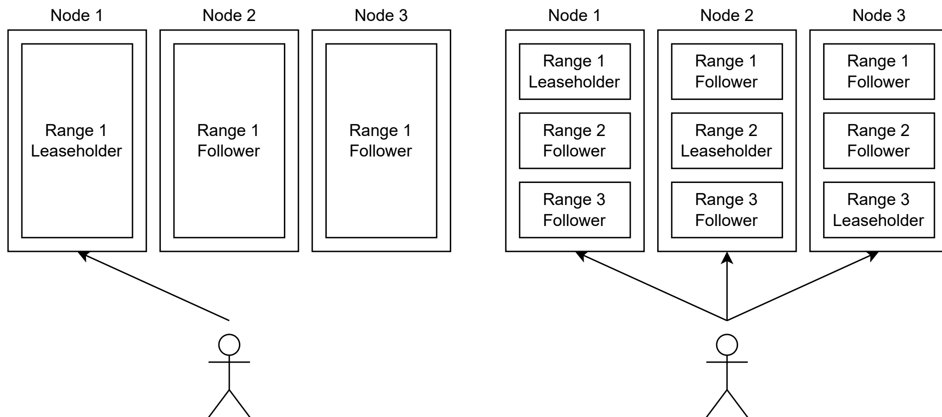

Figure 3.3: Before and after splitting ranges and leases

cluster. This approximates how CockroachDB would attempt to distribute a larger data set in a production setting [27].

It is noteworthy though, in our benchmark the ID for a like entry is based on an integer which is incremented for every insert being done. Our initial ranges are split into intervals containing the entries with ID's 0-333, 334-666 and 667-1000 respectively. But as new likes are inserted, their ID's continue to increment beyond the predefined ranges. CockroachDB will then create a new range for all the new likes, skewing the workload of the write operation slightly as the node being the leaseholder of the new range will be the one in charge of proposing new inserts. It is not the case for the read operation though. When we perform the read query, a scan is performed on an index table rather than the actual likes table. This table is based on the post's ID which is always of fixed size. As we have split this index on the post ID the workload will always be evenly divided.

In terms of measurements, we were most interested in latency as uncertainty restarts will increase the latency of reads. We decided to build our benchmark on top of CockroachDB's workload generator [22] which automatically handles collection of metrics. These metrics include throughput, measured as operations per second, as well as the median, 95th and 99th percentile latency. We will refer to these as the p50, p95 and p99 latencies from now on. We chose to measure these as, according to Kleppman, percentiles are a better choice than the arithmetic mean as they better reflect what the end user experiences. These latency percentiles are also commonly used to define *service level agreements*, making them particular interesting for some service providers [21]. Because of the way our benchmark was designed, with parallel workers making sequential requests, the throughput and latency were directly connected. We still opted to measure the throughput separately though to better visualise how the cluster utilization changed with the number of workers.

We also wanted to find out whether our changes actually reduced the number of uncertainty restarts. CockroachDB stores a number of different metrics per node and exposes them as an HTTP endpoint at */_status/vars*. These metrics include the number of uncertainty restarts that has occurred at a specific node. To get the total amount of uncertainty restarts, we fetched these metrics from each node using the endpoint and added them up at the end of the benchmark [23].

The benchmark was built on top of CockroachDB built-in workload tool, which includes support for a number of benchmarks and provides a convenient framework to implement new ones [22]. The source code for our benchmark is available as part of our CockroachDB fork on GitHub [29].

3.4.3 Benchmark setup

AWS instance size	m5d.xlarge
vCPUs	4
Memory	16 GB
Storage	150 GB NVMe SSD
Network bandwidth	Up to 10 Gbps
Operating system	Ubuntu 20.04 LTS
Extra software	Chrony 3.5 configured to use Amazon Time Sync Service

Table 3.1: Benchmark node specifications

To run our benchmark, we deployed two CockroachDB clusters to AWS. One of the two clusters ran a standard deployment of CockroachDB 21.2.4, with a precompiled binary retrieved from their website [26]. The cluster used all the default settings which includes a *max_offset* of 500 ms. The second cluster ran our forked version of CockroachDB 21.2.4 with the same settings as the standard cluster. We built our fork using the provided Docker image which is published on Docker Hub and encapsulates all dependencies needed for compilation [1]. Both setups were configured to use Amazon Time Sync Service, meaning they both had highly synchronized clocks and hence should both have a reduced risk of necessary uncertainty restarts compared to a cluster with standard clock synchronization. For this experiment, we were specifically interested in testing the effects of our modifications to reduce the size of uncertainty intervals, which should only affect unnecessary restarts.

For our experiments, we aimed to set up clusters which were as close to a production ready setup as possible. To achieve this, we used the recommendations published by CockroachDB in their production checklist [25]. As recommended by the checklist for a high-availability setup, we deployed three nodes across three availability zones in the eu-north-1 (Stockholm) region. CockroachDB supports more than three nodes per cluster, but we opted to use three because of cost restrictions. We used m5d.xlarge EC2 instances which include local SSD physical disks rather than slower network-attached disks that other instance sizes rely on [6]. These instances are the minimum ones recommended by CockroachDB for production AWS deployments, and we decided against using more powerful nodes because of cost restrictions [25]. Table 3.1 shows more detailed hardware and software specifications for the nodes used. The benchmark code itself was run from a fourth server in one of the availability zones, with the same specifications as the cluster nodes.

Previous benchmarks have shown that performance varies between AWS EC2 instances of the same type [19]. Since this could affect our result, we ran the benchmarks against both regular CockroachDB and our modified version in sequence on the same instances. EC2 instance performance normally does not fluctuate too much when running for a long period of time. There are however occasions where performance degradation may appear due to

AWS underlying resource sharing mechanism. This can happen, for instance, when another virtual machine is created on the same physical device. Such degradations may stay between 1 and 2 minutes [19]. To detect such degradations, we looked for outliers in the data and looked more closely at the measurements to see if there were any shorter dips in performance during the test. No such degradations were found.

3.4.4 Measuring timestamp bounds sizes

As mentioned in section 2.4, we were not able to find any numbers on what clock skew to expect from Amazon Time Sync Service and ClockBound. As the size of the bounded timestamps would directly affect our results, we opted to run our own tests separately from our benchmark of CockroachDB. The test was constructed as a script written in Go which used our port of ClockBound, called TrueClock, which is further described in section 3.3.1. The script made a clock reading every 250 ms and saved the size of the bounded timestamp. As TrueClock synchronizes with Chrony once a second, we chose 250 ms to ensure readings were made across the entire synchronization interval. The results were then split into 5 minute buckets and for each bucket the script calculated the 50th, 95th and 99th percentile bounds length. The script ran for a total of 12 hours on a t3.medium EC2 instance, resulting in 144 measurements over 172 800 clock readings. The full code for the test script and the specification for the test server are shown in appendix B.

Chapter 4

Experimental validation

In this chapter, we start of by presenting the results from performing our benchmark as described in section 3.4.2. The first results are based on running the workload with 95% reads and 5% writes, then we present results based on 50% reads and 50% writes. Finally, we show measurements on the bounds sizes sampled from TrueClock during a 12-hour test.

4.1 Read-heavy workload

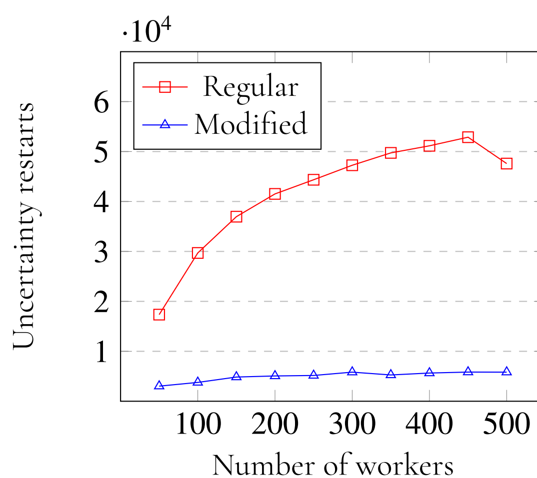

Figure 4.1: Uncertainty restarts

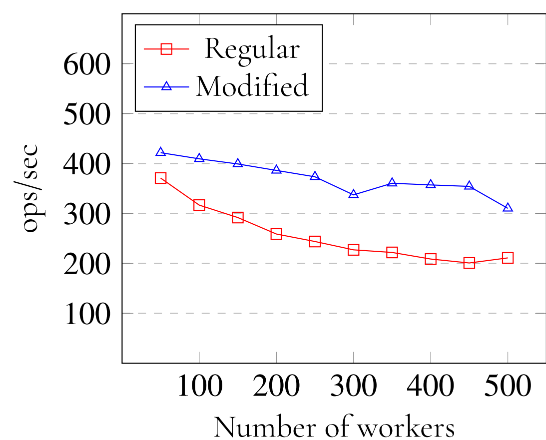

Figure 4.2: Throughput

Figure 4.1 shows the number of uncertainty restarts that occurred during our read-heavy benchmarks (95% reads, 5% writes). We see that the modified version of CockroachDB greatly

reduced the number of uncertainty restarts compared to the regular version. For our modified version, the number of restarts grew slightly as the number of workers increased, but eventually reached a stable level from about 300 workers and up. For the regular version, the restarts increased rapidly as the number of workers increased, and then the growth slowed down a bit after 200 workers. The number of restarts kept growing with each increase in workers, except for the last benchmark at 500 workers. Here the number of restarts dropped slightly, breaking the previous trend. There was no similar drop at 500 workers for the modified version.

In figure 4.2, we show the throughput, measured as the total number of queries per second, by the number of workers. From this we see that the highest throughput was achieved with the lowest number of workers, 50, for both the regular and modified version. As the number of workers increased, the throughput decreased. The modified version of CockroachDB consistently achieved better throughput than the regular version, with the gap growing as the number of workers increased. The exception to this were the tests at 300 and 500 workers respectively, where the gap closed a bit. The modified version however still performed better.

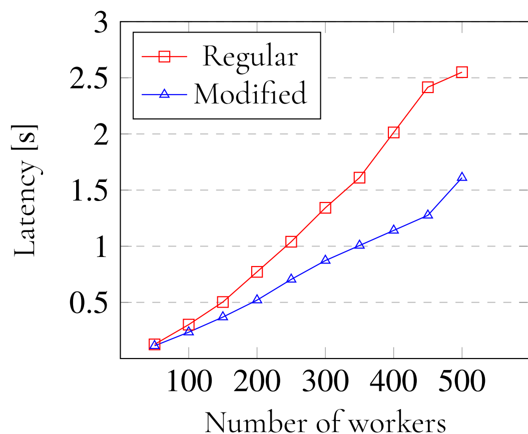

Figure 4.3: p50 latency of read operations

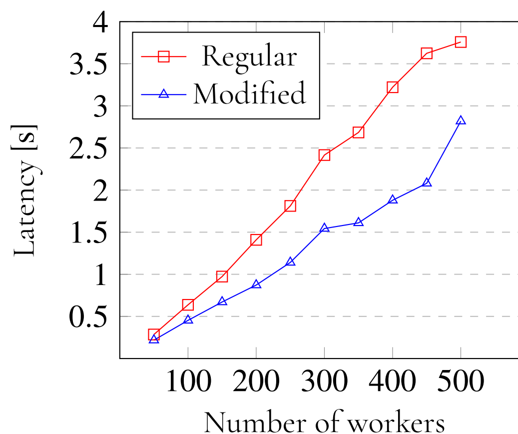

Figure 4.4: p95 latency of read operations

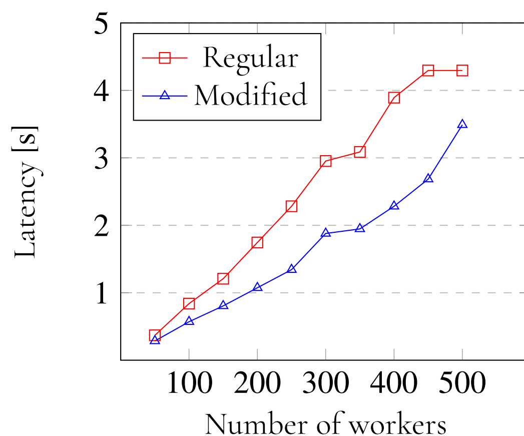

Figure 4.5: p99 latency for read operations

Figures 4.3 through 4.5 show the 50th, 95th and 99th percentile latencies measured for the

read operations. Looking at the p50 latencies, in figure 4.3, we see that the latencies were roughly the same between the regular and modified version at the lowest number of workers. As the number of workers increased, the latencies grew at a linear pace, but the regular version grew faster than the modified version. The difference topped out at 450 workers where the modified version was roughly 47% faster than the regular version. The p95 and p99 latencies, figures 4.4 and 4.5, show a similar pattern with the latencies growing linearly and the modified version being consistently faster, reaching the greatest improvement at 450 workers. Here the modified version was 43% and 37% faster for the p95 and p99 latencies respectively.

Similarly to the uncertainty restarts and throughput, the trends broke slightly when the number of workers reached 500. In the case of the p50 latencies, the modified version jumped up whilst the regular version grew less, causing the gap between the two to shrink. The p95 and p99 figures show a similar jump but more pronounced with the modified version's p99 latency seeing an even greater increase. The modified version stayed faster than the regular version but comparing 500 workers to 450, the p50 latency improvement decreased from 47% to 37% and the p99 from 37% to 19%. We also see some skews in the p95 and p99 tables at 300 workers where the latencies are slightly greater than the trend.

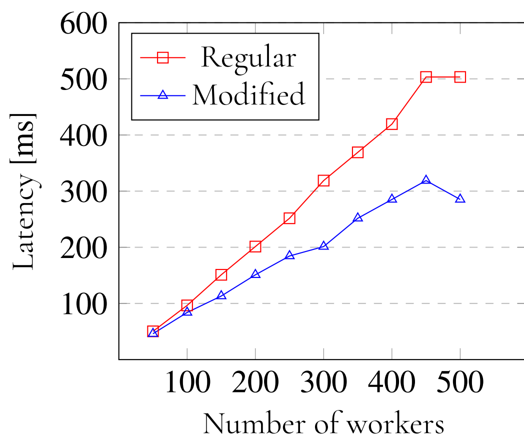

Figure 4.6: p50 latency of like operations

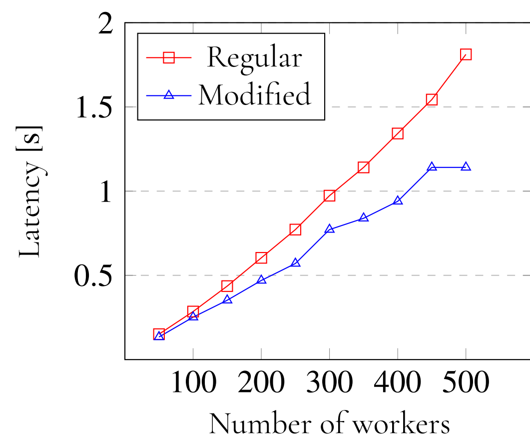

Figure 4.7: p95 latency of like operations

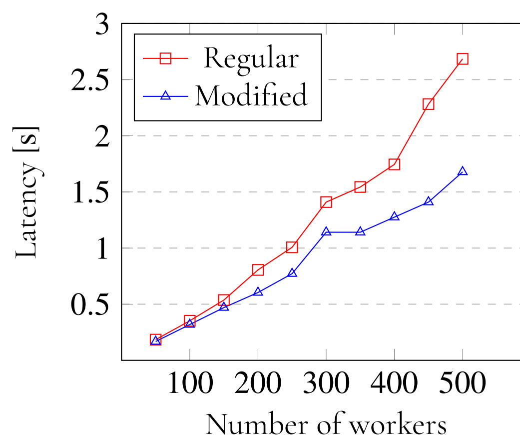

Figure 4.8: p99 latency of like operations

The latency measurements for the like operations, described in listing 3.2, are shown in figure 4.6, 4.7 and 4.8. These show similar trends to the read operations where the latencies were roughly equal when starting at 50 workers and then increased linearly as the number of workers grew. The modified version was consistently faster than the regular version, just like the read operations, and the difference grew as the number of workers increased. The improvements were generally not as large as for the read operations but topped out at a 37% improvement for the p50 latency at 450 workers. We can also see similar skews in the graphs at 300 and 500 workers respectively.

The full results for the read-heavy benchmark are available in appendix A.

4.2 Write-heavy workload

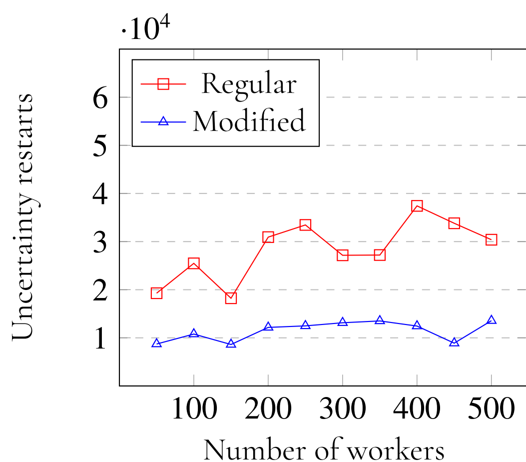

Figure 4.9: Uncertainty restarts

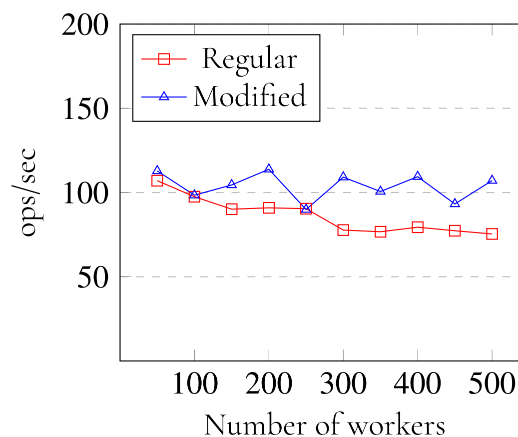

Figure 4.10: Throughput

Figures 4.9 and 4.10 shows the uncertainty restarts and total throughput for our write-heavy benchmark (50% reads, 50% writes). Compared to the results for the read-heavy benchmark in section 4.1, the modified version still yielded fewer uncertainty restarts, but the difference is much smaller. Regular CockroachDB generated much fewer restarts while the modified version generated more. We also do not see the same clear trend from the read-heavy benchmark with the regular version seeing increased restarts as the number of workers increased. The number of restarts does seem to increase, but the trend is not as clear and the measurements are noisy.

In figure 4.10 we see that the modified version offered improved throughput, just as in the read-heavy benchmark, but the difference is not as great and occasionally throughput dips to the same level as the regular version.

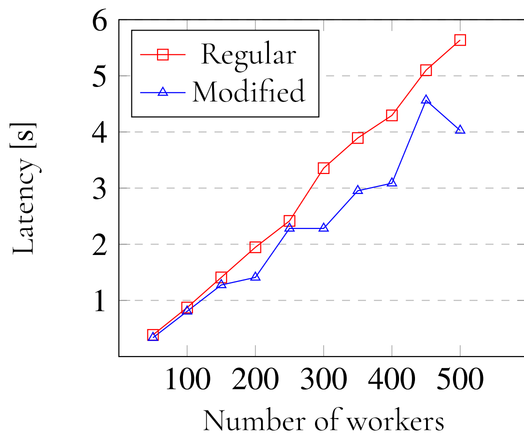

Figure 4.11: p50 latency of read operations

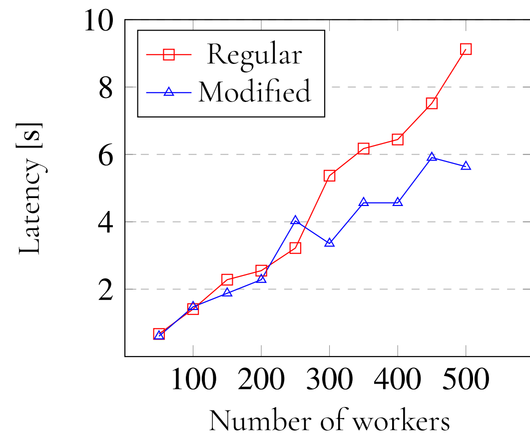

Figure 4.12: p95 latency of read operations

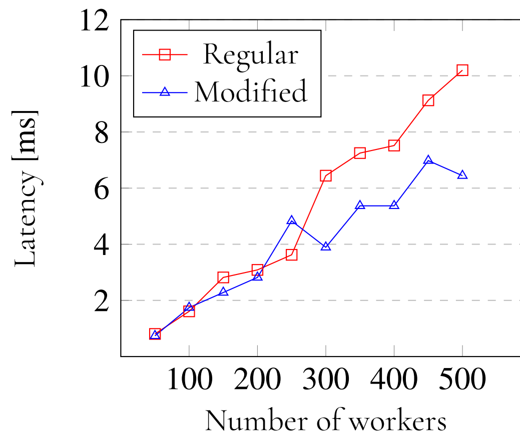

Figure 4.13: p99 latency for read operations

Figures 4.11, 4.12 and 4.13 show the latency results from our write-heavy benchmarks. Here we see a similar trend as in the read-heavy benchmarks with the latency growing linearly with the number of workers. The result differed a bit from the read-heavy benchmark though. In this case, the modified version generally achieved lower latencies, particularly as the workers grew, but the difference was smaller. For lower number of workers, there was no discernible difference between the two.

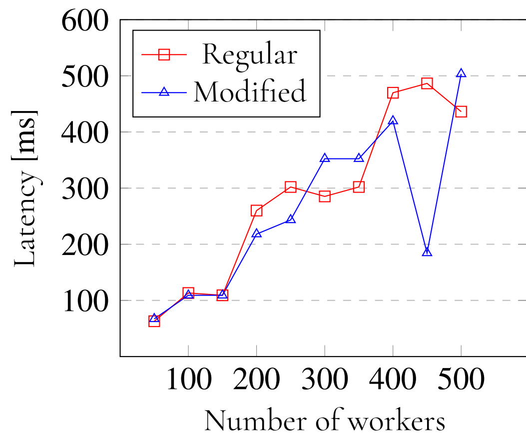

Figure 4.14: p50 latency of like operations

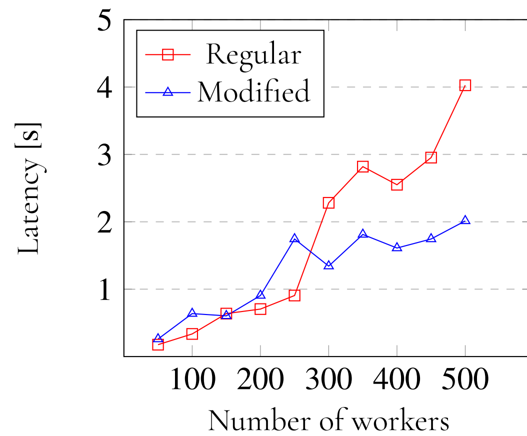

Figure 4.15: p95 latency of like operations

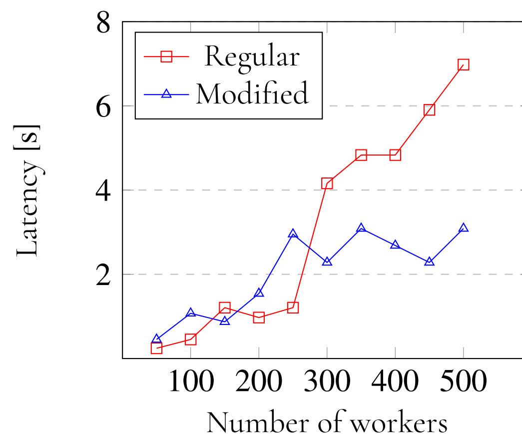

Figure 4.16: p99 latency for like operations

The like operations showed a greater variance in the latency measurements, as can be seen in figures 4.14 through 4.16. In terms of the p50 latency, the regular and modified versions showed similar latencies which grew with the number of workers. The exception is an outlier value at 450 workers, where the modified version's latency dipped dramatically. The p95 and p99 latencies show a bigger difference between the two. Here they stayed relatively equivalent in latency until around 300 workers where the regular version's latency increased rapidly, making the modified version the faster of the two.

The full results for the write-heavy benchmark are available in appendix D.

4.3 Timestamp bounds sizes

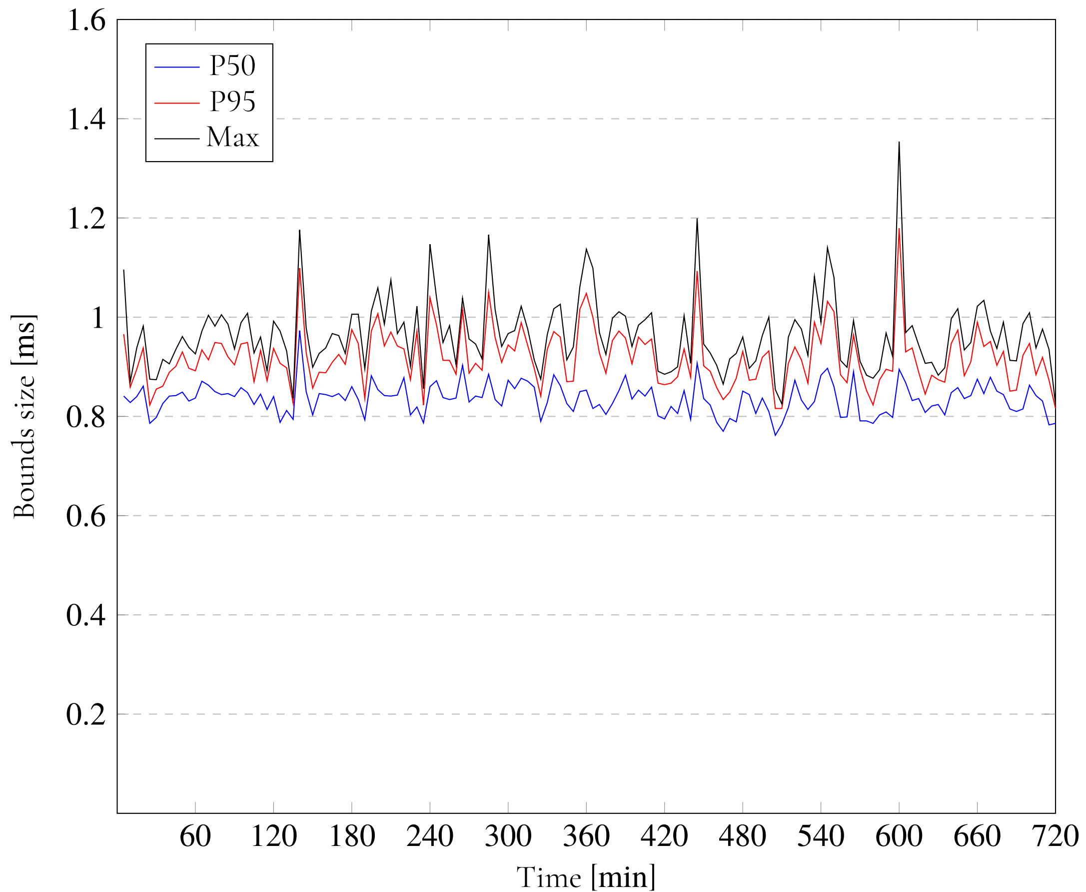

Figure 4.17: Size of bounds generated by TrueClock

Figure 4.17 shows the median (P50), 95th percentile (P95) and maximum size of the bounds generated during our 12-hour test. We see that the size stays stable throughout the test with short spikes at irregular intervals. The median size is slightly above 0.8 ms with the maximum size for each measurement varying between 0.9 ms and 1 ms. During the spikes, the maximum measurements mostly stayed under 1.2 ms with the largest spike, at around 600 minutes in, reaching almost 1.4 ms. The results also included the 99th percentile, but we opted not to include it in the diagram as it stayed very close to the maximum values, making the diagram hard to read. The full results are available in appendix E.

Chapter 5

Discussion

In this chapter, we discuss the results from our experiments, analyze them in relation to our hypothesis, and take a look at some of the limitations and costs of our changes. We also discuss the validity of our results from different perspectives, with a particular focus on our custom benchmark, and then finish by presenting some relevant future work.

5.1 Interpreting results

Looking at the throughput for both the read-heavy (4.1) and write-heavy workloads (4.2), we see that the highest throughput was achieved at 50 workers and that it did not improve with an increase in workers. This indicates that our workload saturated the database capacity which explains the linear relationship between latency and number of workers that we saw in all tests. This also meant that by varying the number of workers, we could vary the length of the reads as well as the contention on the data. In section 3.2, we hypothesised that workloads with longer reads over multiple nodes along with heavier contention would see larger gains from shorter uncertainty intervals, which our data confirms. For the read-heavy workload in section 4.1, we see that the latency difference between the regular and modified versions grew with the number of workers. The results for the write-heavy workload, shown in section 4.2, reveal a similar although less clear trend. In general, the improvements were smaller for the write-heavy workload compared to the read-heavy, including the number of uncertainty restarts. We attribute this to the fact that only reads are restarted because of uncertainty and a smaller number of reads hence results in fewer restarts. This in turn reduces the latency improvements for reads which indirectly also affects the performance of writes, which we discuss in more detail later.

Our hypothesis that reducing the uncertainty interval would improve transactional perfor-

mance is based on the assumption that the static *max_offset* which CockroachDB uses causes unnecessary uncertainty restarts, as described in section 2.5, which results in increased latency. By switching to dynamic and significantly shorter uncertainty intervals, which we have shown in section 4.3, we believed that the number of uncertainty restarts would decrease but the effect size is dependent on how short we can get our dynamic intervals and how stable their sizes are. To better understand this, we ran a separate test for the bounds sizes generated by ClockBound and Amazon Time Sync Service over a 12-hour period. The results in section 4.3 show that they stay very stable, normally below 1 ms in size. There are occasional spikes but they are short and do not deviate drastically from the baseline. As the sizes of these bounds directly translate to the size of our uncertainty intervals, we managed to reduce the interval size by a factor of more than 500, compared to the standard *max_offset* of 500 ms. Even when the bounds sizes spike, in our case to a maximum of 1.4 ms, our dynamic intervals are still more than 350x shorter than standard CockroachDB's.

Jumping back to the benchmark results, it seems that our shortened uncertainty intervals did in fact reduce the number of restarts, particularly for the read-heavy workload. The number of restarts were reduced by up to 9x, with the difference growing as the number of workers increased. We believe the increase was caused by more workers causing more contention, in turn heightening the risk of a read-write conflict. The number of restarts for our modified version did not increase significantly with more contention though. This can be explained by the length of the reads greatly exceeding the length of the uncertainty intervals, which are around 1 ms. This places the vast majority of collisions outside the uncertainty interval and hence they do not cause restarts.

Of course, it's possible for read-write collisions to fall outside the uncertainty interval for the regular version as well if the operation takes longer than 500 ms. We believe we are seeing the effect of this in figure 4.1, where the growth in the number of uncertainty restarts slows down as the workers and latency increases. If we had used a higher *max_offset*, we believe the regular version would have seen even more uncertainty restarts as fewer collisions would have fallen outside the uncertainty interval. Interestingly, we can not see the trend from the uncertainty restarts in the latency numbers, instead the difference there grows even larger as the number of workers increases. We believe this is a result of a restart having an effect on latency that is directly proportional to how slow the restarted query is, so even a lower amount of restarts can have a growing impact on latency. If our speculations are correct, we should eventually see the number of uncertainty restarts taper off to a constant number, but our tests were not large enough to show that.

In section 2.5, we established that there are two different kinds of uncertainty restarts: necessary and unnecessary. The necessary restarts are the ones caused by clock skew which results in the ordering of timestamps not matching the real-time ordering. In those cases, a restart must be performed to maintain consistency. In our experiments, we were not interested in testing necessary restarts and hence we configured both clusters to use highly synchronized clocks as provided by Amazon Time Sync Service. This means that both the standard and modified version of CockroachDB should have encountered roughly an equal number of necessary restarts during our experiment. With this in mind, we can conclude that the great difference in restarts we saw throughout our results was composed entirely of unnecessary restarts. These were the restarts we were interested in testing as they are dependent on the size of the uncertainty intervals, which we aimed to shorten. Another interesting observation

is that our modified version still saw a few thousand restarts each test. Some of these could still be unnecessary restarts, but they might also be necessary restarts as we still have some level of clock skew.

A very interesting outlier in our read-heavy results is the clear drop in performance for the modified version at 500 workers. We see it in the throughput and latency measurements but interestingly, the performance of the regular version improves slightly compared to its previous trend. It is hard to pinpoint the reason for this outlier but the sudden decrease in uncertainty restarts for the regular version at 500 workers might offer a clue. One explanation is that our tests ran into some bottleneck in CockroachDB that lay outside the path that risks causing uncertainty restarts. With that being the main bottleneck, it could reduce the contention that caused the restarts, in effect reducing them and the added latency they bring. The same bottleneck probably affected the modified version as well but here it would not have helped in reducing restarts as the modified version already sees very few. Unfortunately, our benchmarks only went up to 500 workers, so we aren't able to determine if the outlier is part of a greater trend, which would have helped confirm our explanation.

We hypothesised that our modified version would reduce read latency, but we did not expect it to also reduce write latency as only read operations are affected by uncertainty restarts. Our results show a similar decrease in write latency, especially for the read-heavy benchmark, which we have concluded is most likely caused by CockroachDB's usage of latches. Latches are a form of mutex which are used to achieve serialization guarantees by allowing multiple concurrent reads, but having reads and writes block each other [28]. Because we have a high degree of contention between our reads and writes, writes risk having to wait on latches held by concurrent read operations. When forming latches for read operations, CockroachDB uses the uncertainty interval as an upper limit on the length of the latch duration. This means that the read will either hold the latch on each node until it is complete, or it has taken more time than the uncertainty interval [16]. Given that we have drastically reduced the size of the uncertainty intervals to around 1 ms, we may have reduced them enough to allow many more reads to release their latches earlier than before. Shortening latch durations should reduce both the risk for contention and the average time that writes have to wait on reads. This is what we believe caused our modified version to improve performance of writes as well as reads.

5.2 Limitations and costs

An important consideration for the viability of building distributed systems around highly synchronized clocks are their availability. AWS provides them for free through Amazon Time Sync Service but the two other major cloud platforms, Google Cloud and Azure, do not have equivalent services. For a database like CockroachDB, which is designed to run on a wide array of setups, it's not currently reasonable to require access to highly synchronized clocks but it could be exposed as a configuration option or perhaps be automatically detected. This way, CockroachDB could continue using *max_offset* as it does right now by default, but switch to dynamically calculated uncertainty intervals when possible.

One nice aspect of Amazon Time Sync Service is that it's entirely free [4]. There is no guarantee that this will be the case for other cloud providers though if they introduce a similar

service, or that AWS will not start charging for the service eventually. Even if there was a cost associated with accessing highly synchronized clocks, the performance benefits might make it economical but such a cost-benefit analysis will probably have to be made on a case-by-case basis.

Another important consideration when relying on a service like Amazon Time Sync Service are the added failure modes. In our case, if the service were to fail, it would not cause CockroachDB to immediately fail. Instead, the database would have to rely on the system clock which will drift as it can not be synced against the reference. This would result in the uncertainty intervals gradually increasing in size as the amount of time since last synchronization increases, which ClockBound accounts for. A possible way to mitigate this is to place some upper limit on the size of the uncertainty intervals, after which the node will shut itself down. This would be useful if the problem with Amazon Time Sync Service is limited to a single availability zone as CockroachDB can handle the failure of one availability zone when deployed in a high-availability setup. CockroachDB already has a mechanism where a node will shut itself down if it detects clock skew approaching the *max_offset*, this is a mitigation to avoid losing consistency in the face of degraded clock synchronization. In our case, consistency should never be lost even if clock synchronization is temporarily degraded as ClockBound will continue producing valid, albeit growing, timestamp bounds. Because of this, it might be preferable to let the uncertainty intervals grow and accept any performance degradation that causes. What we have not been able to find is any data on how reliable Amazon Time Sync Service is or what failure modes it has, for example if it's redundantly deployed across availability zones. We believe such information will be needed going forward to fully understand the reliability implications of adapting highly synchronized clocks.

As ClockBound depends on the synchronization metadata provided by Chrony to calculate error bounds, the correctness is dependent on the accuracy of the data provided by Chrony. Chrony calculates these values based on measurements and assumptions about various delays between itself and the reference clock [32]. We have not been able to find much research into how accurate the numbers are, although Facebook claims that they might not always be reliable [32]. The question remains if this will affect the correctness of the bounds generated by ClockBound and break consistency, or if the differences are small enough to be ignored. The Jepsen tests we ran, described in section 3.3.3, indicates that they are correct, but that was only for the limited time and scope of the tests. More public research would be welcome to shed light on the challenges around establishing clock errors bounds as it will be critical to fully utilize highly synchronized clocks. The fact that Spanner has been in production for at least a decade though indicates that calculating accurate error bounds can be done reliably.

5.3 Validity threats

Our decision to design our own benchmark, as described in section 3.4.1, introduces many questions in terms of validity. The most significant is the connection to the real world and how well the benchmark will reflect the performance in a production system. As we have not designed our benchmark around some realistic use case, like TPC-C, we can not say how realistic our performance gains are in relation to a real workload. We also can not say that our non-improvements on TPC-C will apply to all real workloads. Even though TPC-C is

built to be realistic, it only represents a specific application with a narrow use case.

Our approach was instead more akin to the one taken by YCSB, with an artificial workload designed to test a specific aspect of a database. With this in mind, we note that our results need to be placed in context and only represents a single, narrow aspect of database performance. Even a real life workload that does contain long reads and contending writes is bound to be more diverse than our benchmark, meaning the same level of performance improvements that we have seen should not be expected in a production setting. We have not been able to find public numbers on how common uncertainty restarts are but Erik Grinaker was able to provide us with some internal metrics from their managed CockroachDB service. These numbers show that during the period 2022-04-23 to 2022-05-23, on average 0.0152% of transactions were restarted across all clusters. The worst affected cluster saw an average of 0.583%, topping out at 0.933% during the worst measured hour [16]. For comparison, our read-heavy test with 450 workers, the one which saw the largest performance improvements, had 58.2% of transactions restarted for the standard CockroachDB cluster. Worth noting is that their managed databases use a *max_offset* of 250 ms, compared to the 500 ms we used for our standard cluster [16]. The great difference in restart susceptibility tells us that performance gains, if any, for production workloads are bound to be much smaller than the ones measured during our experiments. More experiments will be necessary though to fully judge how our changes will affect actual performance in terms of latency and throughput.

One positive result from our evaluation of both TPC-C and YCSB is that we saw no performance degradation caused by our changes, as noted in section 3.4.1. This was expected as uncertainty intervals are only used to determine when restarts are needed, so shortening them should not negatively affect performance. Our dynamic intervals could theoretically grow past the standard 500 ms length if the clock synchronization is bad enough. Given that the default value is already a pessimistic upper bound on clock skew though, we can expect our dynamically calculated bounds to stay below that even when clocks are not highly synchronized. If the clock synchronization were to fail, the intervals risk growing unbounded like described in section 5.2, but this can be avoided with the discussed mitigations. The only way we see that our changes could directly degrade performance is the added latency of a clock reading. As mentioned in section 3.3, we ran into issues with the standard ClockBound daemon as the clock readings were made over a datagram socket, which made each reading much slower. We noticed this issue after seeing a degradation in performance when testing TPC-C and YCSB. The issue was fixed by porting ClockBound to a Go library, removing the need to send a datagram request. This dramatically lowered the latency of a clock reading, but it was still slower than a normal system clock reading that standard CockroachDB uses. Judging by the fact that the performance degradation we saw for TPC-C and YCSB disappeared though, we believe the added latency in our implementation to be negligible.

We believe the benchmark introduces the largest threats to the validity of our results, but there are other threats that should be discussed as well. As we have seen in the results, especially for the write-heavy workload, there are some data points that are noisy, making the trends harder to observe. According to their own guides, CockroachDB recommends running other benchmarks for at least 30 minutes, but we only ran every individual benchmark for 5 minutes due to cost concerns [24]. Running the benchmark for a longer duration or several times with aggregated results could reduce the noise.

As previously mentioned, we believe the database was fully saturated throughout all tests based on the throughput numbers. Although this gave us good control over the latency it also meant we could not vary the latency and contention separately. A better setup would be to have separate parameters to vary the length of the reads, for example data size, and another for contention, like the number of workers. With our setup, we are not able to tell if the improvements are caused by an increase in the length of the read operations, an increase in contention or both.

We also only ran the benchmark on the simplest setup recommended by CockroachDB, consisting of only three nodes within a single region. It's possible that our results does not translate to clusters with more nodes or clusters that are distributed across multiple regions. What these setups have in common though is that they imply increased latency, meaning our results might actually become more significant in these setups. Because of cost and time restrictions, experimenting with these setups is outside the scope of this thesis, but we hope more research will be conducted in the future.

5.4 Future work

As mentioned earlier, the benchmark we built represents the form of workload which we thought stood to gain most from our changes, but these gains might not appear for other workloads. We believe one of the most interesting avenues for future research is what effect highly synchronized clocks can have on other workloads. In particular, we would be interested to see numbers from the industry on how shorter and dynamic uncertainty intervals impacts performance in production. This thesis has also been conducted as a case study on CockroachDB but highly synchronized clocks could potentially be useful to a wide variety of distributed systems in different ways. With AWS providing free availability to highly synchronized clocks on their EC2 instances, we hope to see more case studies conducted as highly synchronized clocks are increasingly commoditized.

Throughout this project, a limiting factor has been the cost of running the benchmarks on AWS, and we hope to see more studies performed on the subject with better funding. One way in which costs limited our experiments was the length of the benchmarks. We noticed that some of our results were noisy, which could be improved by running longer benchmarks. Cost also limited us to a three node cluster and we believe it would be worth researching how well these changes scale to a larger number of nodes. As latency increases due to extra roundtrips between more nodes, so does the probability of uncertainty restarts occurring. On top of that, to reduce the number of uncertainty restarts, CockroachDB implements an optimization. When a transaction encounters a timestamp within the uncertainty interval, it chooses the maximum value of that timestamp or the current node's time. This is to make sure that all data on that node is treated as past operations and resulting in each node causing a maximum of one uncertainty restart per transaction [2]. The number of uncertainty restarts for a transaction is thus limited by the number of nodes in the cluster and a cluster with more nodes should theoretically risk encountering more restarts.

When investigating possible ways to adapt CockroachDB, we chose between two possible approaches. Reducing the uncertainty intervals, which is what we did, or implement commit-wait, similar to Spanner. We opted to not implement commit-wait as the scope of that work

was too great for this thesis, but we would be very interested to see future work on how that could be implemented in CockroachDB and what effect it would have on performance. The commit-wait approach would eliminate the need for uncertainty restarts, which might further reduce latency, but will also require adding some delay to write operations proportional to the clock skew. As we saw in our results, reducing uncertainty restarts also improves the performance of write operations, so any potential added latency from commit-wait might be balanced out.

Chapter 6

Conclusion

With this thesis, we aimed to investigate how distributed databases can be adapted to utilize highly synchronized clocks. We conducted the project as a case study on CockroachDB, a distributed database which relies on clocks but does not expect them to be highly synchronized. With our work, we wanted to answer two research questions:

Q1 How can CockroachDB be adapted to utilize high-precision clock synchronization?

Q2 What effect does this have on CockroachDB's transactional performance?

In answering **Q1**, we arrived at two approaches. One where CockroachDB would be modified to replicate Spanner's transactional model where each write transaction will wait for an amount of time which matches the current clock skew, removing any uncertainty around timestamp ordering. This model has the benefit of enabling strongly consistent reads without locks, and it also enables global linearizability, compared to the slightly weaker causal consistency offered by CockroachDB today. The work of implementing was deemed too large for this project and was instead left to future research. Instead, we implemented our second proposal which implied making CockroachDB's uncertainty intervals shorter and dynamic. These uncertainty intervals are used to determine when two transactions can not be ordered by their timestamps alone because of clock skew. Unfortunately, they also risk causing unnecessary transaction restarts and added latency. In order to make the uncertainty intervals dynamic, we used a tool developed by AWS called ClockBound to calculate error bounds on clock readings. We then combined this with highly synchronized clocks provided by Amazon Time Sync Service, which we hoped would dramatically shorten the uncertainty intervals and in turn reduce the number of unnecessary restarts and improve performance.

To test our modifications and answer **Q2**, we performed two different experiments. The first was designed to determine how short we could get our uncertainty intervals when using Amazon Time Sync Service and ClockBound, as well as how stable their length would be.

The results from this test showed that the sizes stayed very stable over a 12-hour period, making our uncertainty intervals around 500x shorter than the default, static 500 ms used by standard CockroachDB.

The second experiment was designed to measure the performance implications of our changes and the shorter uncertainty intervals. We constructed a benchmark which emulated the kind of workload which we believed would be most affected by unnecessary restarts, that is workloads with long reads over multiple nodes with contending writes. After running the benchmark in both a read-heavy and write-heavy setup and comparing it against standard CockroachDB, we found that our modifications dramatically lowered the number of uncertainty restarts. This in turn significantly improved performance, especially for the read-heavy benchmark, with median latencies seeing up to a 47% improvement for read operations and up to 37% for write operations. These improvements also grew with the level of contention and the length of the reads. As our benchmark was specifically focused on unnecessary restarts, we do not expect equivalent performance improvements for production workloads and further research will be needed to better understand the performance implications of our modifications. We do conclude though that our changes should never result in reduced performance.

As a result of our work, we believe better support for highly synchronized clocks has potential to be a worthwhile addition to CockroachDB. We also believe that highly synchronized clocks will play an important role in distributed systems going forward and deserves further investment and research, especially on their reliability and availability which are the biggest roadblocks to adoption.

References

[1] CockroachDB Docker image. https://hub.docker.com/layers/builder/cockroachdb/builder/20210714-130445/images/sha256-5b1fbd5c5d4d94cee2be6897dc76bcd59c5180773ce73df6a12fe6ea294f96a3. Accessed: 2022-05-11.

[2] Design. https://github.com/cockroachdb/cockroach/blob/master/docs/design.md, October 2021. Accessed: 2022-04-19.

[3] Aarush Ahuja, Vanita Jain, and Dharmender Saini. *Measuring Clock Reliability in Cloud Virtual Machines*, pages 87–98. Springer International Publishing, Cham, 2021.

[4] Amazon. Introducing the Amazon Time Sync Service. https://aws.amazon.com/about-aws/whats-new/2017/11/introducing-the-amazon-time-sync-service/, Nov 2017. Accessed: 2022-01-03.

[5] Amazon. Amazon Time Sync Service now makes it easier to generate and compare timestamps. https://aws.amazon.com/about-aws/whats-new/2021/11/amazon-time-sync-service-generate-compare-timestamps/, Nov 2021. Accessed: 2022-01-03.

[6] AWS. Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/. Accessed: 2022-02-01.

[7] Peter Bailis and Ali Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *Commun. ACM*, 56(5):55–63, may 2013.

[8] brianfrankcooper. Core workloads. https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads, 2020. Accessed: 2021-05-03.

[9] Apache Cassandra. Guarantees. https://cassandra.apache.org/doc/latest/cassandra/architecture/guarantees.html. Accessed: 2022-01-31.

[10] chrony. chrony. https://chrony.tuxfamily.org, 2021. Accessed: 2022-05-02.

[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. SoCC '10, pages 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[12] Transcation Processing Performance Council. TPC benchmark C. `https://tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf`, February 2010. Accessed: 2022-02-22.

[13] James C. Corbett et al. Spanner: Google's Globally-Distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 261–264, Hollywood, CA, October 2012. USENIX Association.

[14] Rebecca Taft et al. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1493–1509, New York, NY, USA, 2020. Association for Computing Machinery.

[15] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 81–94, Renton, WA, April 2018. USENIX Association.

[16] Erik Grinaker. Personal communication.

[17] Jepsen. Jepsen. `https://github.com/jepsen-io/jepsen`, 2022. Accessed: 2021-02-08.

[18] Jepsen. Jepsen tests for cockroachdb. `https://github.com/jepsen-io/jepsen/tree/main/cockroachdb`, 2022. Accessed: 2021-02-08.

[19] Dejun Jiang, Guillaume Pierre, and Chi-Hung Chi. Ec2 performance analysis for resource provisioning of service-oriented applications. volume 6275, pages 197–207, 01 2009.

[20] Spencer Kimball and Irfan Sharif. Living Without Atomic Clocks. `https://www.cockroachlabs.com/blog/living-without-atomic-clocks/`, Apr 2021. Accessed: 2022-01-03.

[21] Martin Kleppmann. *Designing Data-intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*, pages 14–16, 221–226, 287–299, 324–329, 321–351. O'Reilly Media, 2017.

[22] Cockroach Labs. cockroach workload. `https://www.cockroachlabs.com/docs/stable/cockroach-workload.html`. Accessed: 2022-03-15.

[23] Cockroach Labs. Monitor CockroachDB with Prometheus. `https://www.cockroachlabs.com/docs/v21.2/monitoring-and-alerting#prometheus-endpoint`. Accessed: 2022-05-02.

[24] Cockroach Labs. Performance Benchmarking with TPC-C. `https://www.cockroachlabs.com/docs/v21.2/performance-benchmarking-with-tpcc-large`. Accessed: 2022-02-01.

[25] Cockroach Labs. Production Checklist. https://www.cockroachlabs.com/docs/v21.2/recommended-production-settings. Accessed: 2022-02-01.

[26] Cockroach Labs. Releases. https://www.cockroachlabs.com/docs/releases/index.html. Accessed: 2022-05-11.

[27] Cockroach Labs. Split at. https://www.cockroachlabs.com/docs/stable/split-at.html. Accessed: 2022-04-07.

[28] Cockroach Labs. Transaction Layer. https://www.cockroachlabs.com/docs/stable/architecture/transaction-layer.html. Accessed: 2022-04-11.

[29] Fabian Lindfors. Cockroach. https://github.com/fabianlindfors/cockroach, 2022. Accessed: 2022-05-11.

[30] Fabian Lindfors. Trueclock. https://github.com/fabianlindfors/trueclock, 2022. Accessed: 2022-03-16.

[31] Dennis D. McCarthy and P. Kenneth Seidelmann. *Time: From Earth Rotation to Atomic Physics*. Cambridge University Press, 2018.

[32] Oleg Obleukhov. Building a more accurate time service at Facebook scale. https://engineering.fb.com/2020/03/18/production-engineering/ntp-service/, March 2020. Accessed: 2022-05-16.

[33] Amazon Web Services. Read Consistency. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadConsistency.html. Accessed: 2022-01-31.

[34] Amazon Web Services. Clockbound. https://github.com/aws/clock-bound, 2021. Accessed: 2021-01-19.

[35] Pradeep K. Sinha. *Distributed Operating Systems: Concepts and Design*, pages 283–292. PHI Learning Pvt. Ltd., 1996.

[36] Bernhard Sterzbach. Gps-based clock synchronization in a mobile, distributed real-time system. *Real-Time Systems*, 12:63–75, 2004.

[37] Pınar Tözün, Ippokratis Pandis, Cansu Kaynak, Djordje Jevdjic, and Anastasia Ailamaki. From A to E: Analyzing TPC's OLTP Benchmarks: The Obsolete, the Ubiquitous, the Unexplored. EDBT '13, pages 17–28, New York, NY, USA, 2013. Association for Computing Machinery.

Appendices

Appendix A

Benchmark database schema

The following listing shows the code used to set up the database schema for our benchmark.

```
CREATE TABLE posts(
    id INTEGER PRIMARY KEY, title TEXT
);

CREATE TABLE likes(
    id SERIAL PRIMARY KEY,
    post_id INTEGER,
    FOREIGN KEY (post_id) REFERENCES posts (id),
    INDEX (post_id)
);
```

Listing A.1: Code for setting up database schema

Appendix B

Timestamp bounds size test setup

Listing B.1 shows the Go code used to conduct our test for the size of timestamp bounds generated by TrueClock and Amazon Time Sync Service. The specifications for the server used to run the test are shown in table B.1.

```
1  package main
2
3  import (
4      "fmt"
5      "math"
6      "sort"
7      "time"
8
9      "github.com/fabianlindfors/trueclock"
10 )
11
12 const TestDuration = 12 * time.Hour
13 const BucketDuration = 5 * time.Minute
14 const TimeBetweenMeasurements = 250 * time.Millisecond
15
16 type Measurement struct {
17     Mean   float64
18     Median float64
19     P95    float64
20     P99    float64
21     Max    int64
22 }
23
24 func main() {
25     clock, err := trueclock.New()
```

```
26      if err != nil {
27          panic(err)
28      }
29
30      startTime := time.Now()
31      endTime := startTime.Add(TestDuration)
32      bucketNum := 1
33
34      values := make([]int64, 0)
35
36      ticker := time.NewTicker(TimeBetweenMeasurements)
37
38      for {
39          <-ticker.C
40
41          bounds := clock.Now()
42          diff := bounds.Latest.Sub(bounds.Earliest)
43
44          values = append(values, diff.Microseconds())
45
46          now := time.Now()
47          if now.After(startTime.Add(time.Duration(bucketNum) *
        BucketDuration)) {
48              measurement := calculateMeasurement(values)
49              sinceStart := now.Sub(startTime)
50              fmt.Printf(
51                  "[%s]\t%d\t%v\t%v\t%v\t%v\t%d\n",
52                  sinceStart,
53                  bucketNum,
54                  measurement.Mean,
55                  measurement.Median,
56                  measurement.P95,
57                  measurement.P99,
58                  measurement.Max,
59              )
60
61              values = make([]int64, 0)
62              bucketNum += 1
63          }
64
65          if now.After(endTime) {
66              return
67          }
68      }
69  }
70
71  func calculateMeasurement(values []int64) Measurement {
72      total := int64(0)
73      max := int64(0)
74      for _, value := range values {
75          total += value
76
77          if value > max {
78              max = value
79          }
80      }
```

```
81
82     mean := float64(total) / float64(len(values))
83
84     sort.Slice(values, func(i, j int) bool { return values[i] <
       values[j] })
85
86     return Measurement{
87         Mean:   mean,
88         Median: calculatePercentile(values, 0.5),
89         P95:    calculatePercentile(values, 0.95),
90         P99:    calculatePercentile(values, 0.99),
91         Max:    max,
92     }
93 }
94
95 // Ported from https://code.activestate.com/recipes/511478-finding-
       the-percentile-of-the-values/
96 // Author: Wai Yip Tung
97 // License: Python Software Foundation License (PSF)
98 func calculatePercentile(sortedValues []int64, percentile float64)
       float64 {
99     index := float64(len(sortedValues)-1) * percentile
100
101    floor := math.Floor(index)
102    ceil := math.Ceil(index)
103
104    if floor == ceil {
105        return float64(sortedValues[int(index)])
106    }
107
108    d0 := float64(sortedValues[int(floor)]) * (ceil - index)
109    d1 := float64(sortedValues[int(ceil)]) * (index - floor)
110    return d0 + d1
111 }
```

Listing B.1: Script used to measure sizes of bounds generated by TrueClock

AWS instance size	t3.large
vCPUs	2
Memory	8 GB
Storage	30 GB network-attached EBS volume
Network bandwidth	Up to 5 Gbps
Operating system	Ubuntu 20.04 LTS
Extra software	Chrony 3.5 configured to use Amazon Time Sync Service

Table B.1: Server specifications for the timestamp bounds size test

Appendix C

Read-heavy benchmark results

Table C.1 through C.5 shows the full results of our read-heavy benchmark, on which the diagrams in 4.1 are based.

Workers	Regular	Modified
50	17344	3023
100	29694	3768
150	36959	4846
200	41519	5063
250	44351	5159
300	47256	5825
350	49731	5262
400	51150	5647
450	52875	5840
500	47594	5826

Table C.1: Uncertainty restarts

Workers	Regular	Modified
50	370.8	421.7
100	316.7	409.4
150	291.8	399.1
200	258.8	386.2
250	243.8	373.4
300	227.1	337.2
350	221.9	360.5
400	208.7	357.1
450	200.8	354.2
500	211.0	310.2

Table C.2: Throughput (ops/sec)

	Read operations		Like operations	
Workers	Regular	Modified	Regular	Modified
50	125.8	113.2	50.3	46.1
100	302.0	234.9	96.5	83.9
150	503.3	369.1	151.0	113.2
200	771.8	520.1	201.3	151.0
250	1040.2	704.6	251.7	184.5
300	1342.2	872.4	318.8	201.3
350	1610.6	1006.6	369.1	251.7
400	2013.3	1140.9	419.4	285.2
450	2415.9	1275.1	503.3	318.8
500	2550.1	1610.6	503.3	285.2

Table C.3: P50 latency (ms)

	Read operations		Like operations	
Workers	Regular	Modified	Regular	Modified
50	285.2	218.1	151.0	134.2
100	637.5	453.0	285.2	251.7
150	973.1	671.1	436.2	352.3
200	1409.3	872.4	604.0	469.8
250	1811.9	1140.9	771.8	570.4
300	2415.9	1543.5	973.1	771.8
350	2684.4	1610.6	1140.9	838.9
400	3221.2	1879.0	1342.2	939.5
450	3623.9	2080.4	1543.5	1140.9
500	3758.1	2818.6	1811.9	1140.9

Table C.4: P95 latency (ms)

	Read operations		Like operations	
Workers	Regular	Modified	Regular	Modified
50	369.1	285.2	184.5	167.8
100	838.9	570.4	352.3	318.8
150	1208.0	805.3	536.9	469.8
200	1744.8	1073.7	805.3	604.0
250	2281.7	1342.2	1006.6	771.8
300	2952.8	1879.0	1409.3	1140.9
350	3087.0	1946.2	1543.5	1140.9
400	3892.3	2281.7	1744.8	1275.1
450	4295.0	2684.4	2281.7	1409.3
500	4295.0	3489.7	2684.4	1677.7

Table C.5: P99 latency (ms)

Appendix D

Write-heavy benchmark results

Table D.1 through D.5 shows the full results of our write-heavy benchmark, on which the diagrams in 4.2 are based.

Workers	Regular	Modified
50	19272	8715
100	25476	10783
150	18227	8615
200	30926	12166
250	33457	12485
300	27152	13132
350	27204	13509
400	37418	12448
450	33806	8893
500	30387	13565

Table D.1: Uncertainty restarts

Workers	Regular	Modified
50	214.0	226.6
100	193.6	195.6
150	180.5	209.7
200	181.3	228.7
250	181.5	181.0
300	154.8	220.1
350	154.6	200.1
400	158.5	218.1
450	154.7	187.2
500	150.0	215.4

Table D.2: Throughput (ops/sec)

	Read operations		Like operations	
Workers	Regular	Modified	Regular	Modified
50	385.9	335.5	62.9	67.1
100	872.4	805.3	113.2	109.1
150	1409.3	1275.1	109.1	109.1
200	1946.2	1409.3	260.0	218.1
250	2415.9	2281.7	302.0	243.3
300	3355.4	2281.7	285.2	352.3
350	3892.3	2952.8	302.0	352.3
400	4295.0	3087.0	469.8	419.4
450	5100.3	4563.4	486.5	184.5
500	5637.1	4026.5	436.2	503.3

Table D.3: P50 latency (ms)

	Read operations		Like operations	
Workers	Regular	Modified	Regular	Modified
50	671.1	604.0	176.2	260.0
100	1409.3	1476.4	335.5	637.5
150	2281.7	1879.0	637.5	604.0
200	2550.1	2281.7	704.6	906.0
250	3221.2	4026.5	906.0	1744.8
300	5368.7	3355.4	2281.7	1342.2
350	6174.0	4563.4	2818.6	1811.9
400	6442.5	4563.4	2550.1	1610.6
450	7516.2	5905.6	2952.8	1744.8
500	9126.8	5637.1	4026.5	2013.3

Table D.4: P95 latency (ms)

	Read operations		Like operations	
Workers	Regular	Modified	Regular	Modified
50	805.3	738.2	243.3	453.0
100	1610.6	1744.8	453.0	1073.7
150	2818.6	2281.7	1208.0	872.4
200	3087.0	2818.6	973.1	1543.5
250	3623.9	4831.8	1208.0	2952.8
300	6442.5	3892.3	4160.7	2281.7
350	7247.8	5368.7	4831.8	3087.0
400	7516.2	5368.7	4831.8	2684.4
450	9126.8	6979.3	5905.6	2281.7
500	10200.5	6442.5	6979.3	3087.0

Table D.5: P99 latency (ms)

Appendix E

Timestamp bounds size test results

The following table, E.1, contains the full data collected during our test of the timestamp bounds sizes generated by TrueClock and Amazon Time Sync Service. The data was used for the diagram in 4.3.

Time [min]	P50 [ms]	P95 [ms]	P99 [ms]	Max [ms]
5	841.00	966.30	1070.06	1096
10	828.00	859.00	864.00	870
15	840.00	895.05	926.02	938
20	861.00	938.05	966.00	982
25	786.00	823.05	862.03	875
30	798.00	855.05	871.01	874
35	826.00	861.00	902.03	915
40	841.00	889.00	898.01	906
45	842.00	901.05	926.00	935
50	849.00	930.05	954.00	961
55	831.00	897.05	926.03	939
60	837.00	892.10	918.00	926
65	871.00	934.00	963.00	972
70	863.00	914.05	990.03	1004
75	850.00	949.00	969.03	982
80	844.00	947.05	992.03	1005
85	846.00	920.00	972.03	986
90	840.00	904.05	926.01	936
95	858.00	946.05	973.03	989
100	848.00	949.05	995.03	1008

105	824.00	870.10	914.03	928
110	845.00	933.00	953.00	960
115	814.00	872.00	883.00	892
120	840.00	937.00	972.04	992
125	788.00	907.00	954.04	972
130	812.00	898.05	925.01	932
135	794.50	827.00	833.01	837
140	973.00	1099.00	1148.06	1176
145	850.00	929.00	961.03	979
150	803.00	857.00	887.02	899
155	846.00	889.00	910.04	927
160	844.00	888.00	925.03	938
165	839.50	909.05	949.03	967
170	846.00	925.00	947.03	963
175	832.00	905.00	918.00	927
180	860.00	975.00	993.00	1006
185	834.00	947.15	990.03	1006
190	793.00	836.10	883.03	896
195	882.00	972.00	1001.01	1012
200	854.00	1007.00	1048.02	1059
205	842.00	942.00	970.03	986
210	841.00	970.20	1053.05	1075
215	843.00	942.00	959.01	967
220	878.00	936.05	975.03	990
225	803.00	874.05	891.00	901
230	819.00	971.05	1011.01	1022
235	787.00	823.00	843.02	856
240	860.00	1039.20	1125.05	1147
245	872.00	985.00	1026.04	1040
250	838.00	913.00	935.03	949
255	834.00	913.00	964.04	983
260	837.00	885.00	897.00	905
265	903.00	1015.00	1030.01	1038
270	829.00	887.15	942.03	956
275	841.00	907.00	931.03	946
280	838.00	893.00	910.02	916
285	885.00	1050.25	1142.05	1166
290	834.00	959.05	999.02	1015
295	821.00	909.00	931.01	941
300	873.00	944.05	959.02	967
305	856.00	932.05	962.01	973
310	877.50	989.00	1003.04	1022
315	871.00	943.00	963.02	976
320	859.00	897.00	908.01	914

325	790.00	841.00	863.03	875
330	827.50	935.00	948.01	965
335	884.50	971.05	1001.03	1017
340	862.00	960.00	1004.05	1026
345	826.00	870.00	896.03	914
350	810.00	871.15	926.03	940
355	849.50	1016.05	1052.01	1061
360	853.00	1048.05	1111.05	1137
365	816.00	1000.15	1074.05	1099
370	824.00	928.00	951.03	969
375	804.00	887.10	908.04	925
380	826.00	953.05	988.00	998
385	852.00	972.10	1003.01	1011
390	883.00	958.00	988.01	1002
395	835.00	906.00	932.00	941
400	853.00	960.00	977.00	984
405	841.00	945.15	977.00	995
410	859.00	956.05	989.04	1009
415	801.00	867.10	884.02	891
420	795.00	864.00	880.00	885
425	820.00	867.00	883.01	890
430	806.00	880.00	891.01	900
435	852.00	936.00	986.04	1003
440	794.00	879.00	893.03	907
445	907.50	1093.05	1164.07	1199
450	836.00	902.00	932.03	946
455	823.00	891.00	913.01	928
460	788.00	858.00	890.03	903
465	770.00	834.00	859.01	865
470	796.00	849.00	895.05	916
475	789.50	877.00	916.02	927
480	851.00	931.00	948.01	960
485	844.00	873.05	887.01	897
490	806.00	875.10	897.06	912
495	837.00	919.10	948.03	963
500	810.00	932.15	987.03	1000
505	762.00	816.00	842.02	854
510	784.00	816.00	823.00	825
515	818.00	907.05	942.03	960
520	873.00	940.05	973.02	995
525	833.00	914.05	963.03	976
530	814.00	867.10	911.03	923
535	830.00	990.20	1064.03	1082
540	883.00	947.00	976.03	990

545	897.00	1032.20	1118.05	1140
550	860.00	1011.05	1059.05	1080
555	798.00	884.05	904.01	913
560	799.00	868.00	893.00	899
565	890.00	964.00	982.01	993
570	791.00	895.00	907.00	911
575	791.00	852.00	876.01	883
580	786.00	823.05	865.02	877
585	803.00	874.00	889.01	894
590	809.00	895.00	951.04	968
595	798.00	891.00	911.02	921
600	895.00	1179.35	1319.07	1354
605	868.00	930.05	954.03	969
610	832.00	938.10	970.02	983
615	836.00	889.05	931.03	944
620	808.00	845.05	895.02	907
625	821.00	883.00	903.01	909
630	824.00	874.00	881.01	883
635	803.00	869.10	893.00	898
640	848.00	947.05	979.03	997
645	858.00	974.00	1003.03	1017
650	835.50	882.00	921.03	934
655	842.00	910.00	937.01	949
660	875.00	990.00	1013.02	1022
665	846.00	941.05	1004.06	1034
670	879.00	951.00	963.01	972
675	851.00	903.00	922.03	937
680	844.00	931.00	972.04	990
685	815.00	851.15	900.03	913
690	810.00	853.05	900.03	912
695	815.00	923.00	974.03	987
700	862.50	947.10	992.04	1009
705	842.00	884.00	926.02	938
710	831.00	919.10	964.02	976
715	783.00	873.00	922.03	935
720	786.00	816.00	821.00	825

Table E.1: Results from timestamp bounds size test

EXAMENSARBETE Utilizing highly synchronized clocks in distributed databases
STUDENTER Jacob Gunnarsson, Fabian Lindfors
HANDLEDARE Flavius Gruian (LTH)
EXAMINATOR Emma Söderberg (LTH)

Kan högsynkroniserade klockor förbättra prestandan hos distribuerade databaser?

POPULÄRVETENSKAPLIG SAMMANFATTNING **Jacob Gunnarsson, Fabian Lindfors**

Klockor har länge spelat en central roll i många distribuerade system men svårigheterna med att hålla dem synkroniserade har begränsat deras användbarhet. På senare år har dock högsynkroniserade klockor blivit mer tillgängliga och vårt arbete har utforskat hur dessa kan utnyttjas för att förbättra prestandan hos distribuerade databaser.

I distribuerade databaser sprids kopior av all data ut över flera servrar för att förbättra prestanda och bättre kunna hantera störningar. Detta skapar svårigheter när data ska ändras då även kopiorna måste ändras på flera olika ställen. Om data läses från en kopia samtidigt som originalet ändras finns det risk att läsningen ger gammal information efter att ändringen har gjorts. Vissa databaser garanterar dock att detta inte kan hända och att när en ändring har gjorts så syns den för alla framtida läsningar, en egenskap som kallas *konsistens*.

För detta krävs det att databasen kan avgöra i vilken ordning läsningar och ändringar görs även om de hanteras av olika servrar. Ett enkelt sätt att åstadkomma detta är att förknippa en händelse i databasen med en tidsstämpel från serverns klocka. Svårigheter med att hålla servrarnas klockor synkroniserade har dock länge varit begränsande. På senare år har det skett en utveckling i tillgängligheten på bättre klocksynkronisering. AWS har exempelvis introducerat en gratis tjänst som ger tillgång till högsynkroniserade klockor i sina datacenter.

I detta arbete modifierade vi databasen CockroachDB för att utnyttja dessa högsynkroniserade klockor på AWS. CockroachDB använder redan klockor idag men kräver inte god synkronisering vilket leder till att vissa operationer måste startas om i onödan. Genom att anpassa databasen till bättre klockor hoppades vi minska dessa omstarter och på så vis förbättra prestanda.

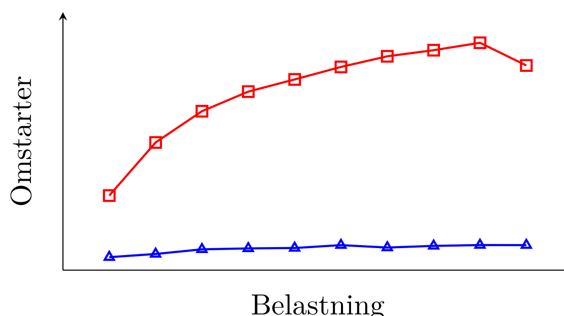

Figure 1: Omstarter för vanliga CockroachDB (röd) och modifierade CockroachDB (blå)

För att se inverkan av våra ändringar byggde och körde vi ett test anpassat för dessa omstarter. Våra ändringar resulterade i 9 gånger färre omstarter vilket gav upp till 1,9 gånger snabbare läsningar och 1,5 gånger snabbare skrivningar.