cockroachlabs.com

Parallel Commits: An atomic commit protocol for globally distributed transactions

35–44 minutes

Distributed ACID transactions form the beating heart of CockroachDB. They allow users to manipulate any and all of their data transactionally, no matter where it physically resides.

Distributed transactions are so important to CockroachDB's goal to "Make Data Easy" that we spend a lot of time thinking about how to make them as fast as possible. Specifically, CockroachDB specializes in globally distributed deployments, so we put a lot of effort into optimizing CockroachDB's transaction protocol for clusters with high inter-node latencies.

Earlier this year, we published a blog post about a new feature called Transactional Pipelining in our 2.1 release. The feature reworked CockroachDB's transaction protocol to pipeline writes within interactive transactions in order to reduce their end-to-end latency. Customers saw a marked improvement in the speed of their transactions when this feature was released.

We closed out that discussion with a preview of how we planned to continue the work of speeding up global transactions. To do so, we needed to take a hard look at the atomic commit protocol we used when

committing transactions. We found that it was overly rigid and hindered performance when run over a collection of individually replicated consensus groups (i.e. `ranges` in CockroachDB). So we went back to the drawing board and developed a new atomic commit protocol optimized for globally distributed transactions running in a system of partitioned consensus groups.

This post will explore the new atomic commit protocol, which we call Parallel Commits. Parallel Commits will be part of CockroachDB's upcoming 19.2 release. The feature promises to halve the latency of distributed transactions by performing all consensus round trips required to commit a transaction concurrently.

The post will pick up where the previous post about transactional pipelining left off. Readers who have read that blog post will be able to jump right into this one. Readers who haven't may want to skim through that post to get a feel for the cluster topology assumptions we will continue to make here.

What is transaction atomicity?

Atomicity is likely the first property that people think of when they hear the word "transaction". At its core, a transaction is simply a collection of operations, and it is said to be atomic if all of its composite operations succeed or fail together as a single unit. In other words, atomicity ensures that a transaction is "all or nothing" - it either commits or rolls back instantaneously.

Databases typically ensure atomicity through a delicate dance of locking, indirection, and coordination between conflicting transactions. The details of how this all fits together is specific to each database

implementation, but they each share common themes.

For example, in PostgreSQL, individual tuples (i.e. versioned rows) each contain the ID of the transaction that wrote it. When a transaction is still in progress, no other transaction should consider these provisional tuples to be "visible", as doing so would be a violation of isolation at even the weakest isolation levels. PostgreSQL enforces this by maintaining each transaction's disposition in a single place, the **commit log** ("clog"). When a transaction finds a tuple from a possibly live conflicting transaction, it checks the commit log to determine whether the tuple is visible or not.

When a transaction commits, its changes should all suddenly become visible at once. How is this accomplished? In PostgreSQL, the trick is to write the fact that the transaction committed to the **commit log**. This ensures that from that point on, any conflicting transaction that runs into one of its tuples will observe the transaction as committed when it checks the commit log. PostgreSQL then updates each tuple with a "hint bit" to inform conflicting transactions that the tuple is committed and that no commit log lookup is necessary. This last part is strictly an optimization and not necessary for atomicity.

When we think about this dance carefully, we realize that the challenge of atomicity isn't one of performing multiple changes simultaneously. Instead, it's fundamentally a game of managing "visibility" of the transaction's operations such that all of a transaction's operations appear to be committed (or rolled back) instantaneously to all observers.

Atomicity in CockroachDB

In a distributed database like CockroachDB, things are fairly similar. The only real difference is that a transaction may perform writes across a

series of different machines. We've seen in past blog posts how CockroachDB handles this with an ID-addressable **transaction record** and **write intents** that contain their transaction's ID. This is discussed in the CockroachDB architecture documentation and in this blog post.

Just like in PostgreSQL, any transaction that stumbles upon a write intent is forced to look up the corresponding transaction record to determine the intent's visibility. Also similar is the fact that a transaction commit simply flips a bit on its transaction record to mark the transaction as committed. The only difference here is that unlike PostgreSQL's commit log, these transaction records are distributed throughout a cluster instead of centralized to improve locality of these transaction record accesses. Finally, after the transaction commit completes, each of the write intents is "cleaned up" asynchronously to avoid unnecessary transaction record lookups. Like hint bits in PostgreSQL, this is a performance optimization and not strictly necessary for atomicity.

Let's take a look at the timeline of a transaction in CockroachDB to get a feel for how this behaves over time. For the sake of simplicity, we'll focus on a write-only transaction. Of course, CockroachDB supports generalized read-write SQL transactions, and the handling of read-only statements within a read-write transaction is straightforward due to multi-version concurrency control (MVCC).

Below, we see that CockroachDB navigates SQL transactions by incrementally laying down write intents as each mutation statement is issued. The transaction also writes its transaction record with a PENDING status at the same time as writing its first intent. We say that each of these intent writes is "pipelined" because we don't wait for them to succeed before responding to the SQL client and informing it to issue more statements.

Eventually, the SQL client issues a COMMIT statement. At this point, CockroachDB waits for all of its intent writes to complete the process of durable replication. Once this is complete, the transaction modifies its transaction record to mark it as committed. This change is also replicated for durability, and the transaction is considered to be committed as soon as this operation completes. The success of the COMMIT statement is then acknowledged to the SQL client.

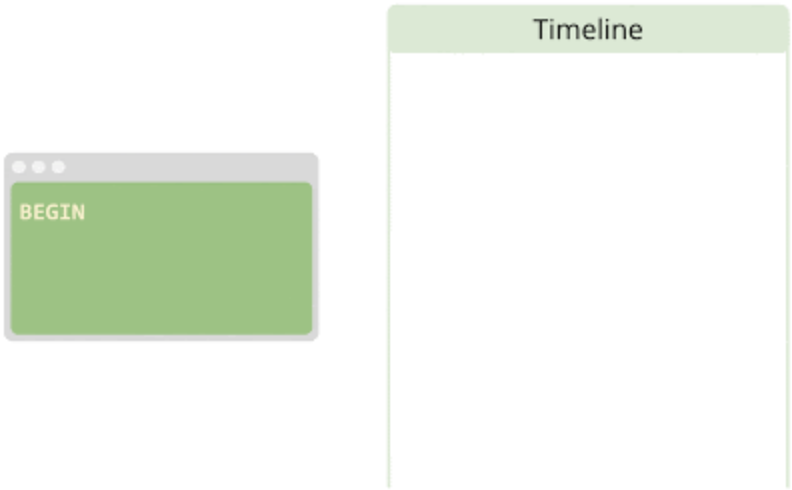

Figure 1: Transaction Timeline, Without Parallel Commits

Asynchronously, after the client has been acknowledged, the transaction goes through and resolves each of its intents. Once this has finished, it can delete its transaction record entirely.

Or at least, this is how transaction atomicity has worked in CockroachDB up until version 19.2.

The problem with two-phase commit

Those familiar with distributed systems may recognize this multi-step process as a variation of the two-phase commit protocol (2PC). Specifically, the process of writing intents is analogous to the “prepare”

phase of 2PC and the process of marking the transaction record as committed is analogous to the "commit" phase of 2PC.

One commonly cited problem with two-phase commit is that it is "blocking". After each participant has voted to commit but before the transaction coordinator has made its final decision, the system enters a fragile state. If the transaction coordinator were to crash, then it would be impossible for others to know the final outcome of the transaction. This would stall the transaction itself and any that conflicted with it. To avoid this issue, systems like CockroachDB run 2PC on top of a consensus protocol, ensuring that transaction state is just as highly available and resilient to failure as the rest of the system. Specifically, in CockroachDB this means that a transaction's **transaction record** is replicated in the same way that its **write intents** are.

As we demonstrated in the previous blog post, the major downside to this strategy, when run in a partitioned-consensus system like CockroachDB, is that it requires two sequential rounds of consensus writes. For example, if a transaction writes to two ranges, the writes to each range will achieve consensus concurrently. However, the transaction must wait for both of these writes to succeed before marking its transaction record as committed. This write to the transaction record then requires a second round of consensus. This is why the latency model we derived in the previous post showed the latency of a transaction to be twice the latency of a single consensus write.

So we see that this structure of layering two-phase commit on top of a system of disjoint consensus groups (i.e. ranges) is responsible for increased transaction latency, which was something we wanted to get rid of.

Proposed solutions from academia

A number of academic groups have taken notice of this problem and have proposed unique solutions to addressing it. The motivating example for much of this work has been the original Google Spanner paper from 2012, which also served as an inspiration for CockroachDB.

In 2013, Kraska et al. presented **MDCC: Multi-Data Center Consistency**, which achieves atomic transaction semantics by replicating update **options** to all storage nodes instead of replicating the update values directly. In MDCC, as soon as the system's application server learns the options for all the records in a transaction, it is able to consider the transaction committed and asynchronously notify the storage nodes to execute the outstanding options. The protocol is then optimized by allowing clients to propose Paxos writes without communicating with a leader and by permitting updates to commute with one another. These two optimizations are made within the framework of Generalized Paxos.

Later in 2013, Mahmoud et al. published Low-Latency Multi-Datacenter Databases using Replicated Commit. The paper suggested running Two-Phase Commit multiple times in different data centers while using Paxos to reach consensus among each data center as to whether a transaction should commit. This strategy, called Replicated Commit, avoids several inter-data center communication trips by placing an independent transaction coordinator in each data center.

In 2015, Zhang et al. approached the same problem from yet a different angle in TAPIR: Building Consistent Transactions with Inconsistent Replication. The project focused on reducing coordination through the use of a new transaction protocol layered on top of a replication scheme

that on its own provides no consistency. The proposed replication technique, called inconsistent replication, provides fault-tolerance without ordering, which allows it to avoid synchronous cross-replica coordination or designated leaders. The transaction protocol is then made aware of the limited guarantees of the replication layer, allowing the system to commit a distributed read-write transaction in a single round-trip.

Finally, in 2018, Yan et al. introduced Carousel: Low-Latency Transaction Processing for Globally-Distributed Data. The system reduced the number of sequential wide-area network round-trips required to commit a transaction and replicate its results while maintaining serializability. It did so by targeting 2-round Fixed-set Interactive transactions, where read and write keys must be known at the start of the transaction, but where reads in the first round can influence writes in the second.

Each of these projects reduced the latency of transactions running within a system of replication groups in different ways, and each required different trade-offs to its system's transaction model to do so. Ultimately, the projects confirmed that the slowdown inherent with running two-phase commit on top of strong replication can be avoided by rethinking how transactions and replication fit together. However, none of them provided a clear blueprint for how an existing system like CockroachDB could address the problem.

For that, we had to create our own solution.

Parallel Commits

Parallel Commits is CockroachDB's approach to solving this

performance problem. The goal of the new atomic commit protocol is to reduce the latency of transactions down to only a single round-trip of distributed consensus. To accomplish this goal, we had to throw away two-phase commit and rework how transactions in CockroachDB arrive at a committed state.

Before discussing how Parallel Commits works, let's consider the requirements that we needed to maintain while adjusting our commit protocol.

The first was that we needed to continue to enforce the property that transactions can't commit until all of their writes are replicated. This is a critical property for ensuring durability of transaction writes and continuing to provide consistency of each individual write, in the CAP sense of the word.

In addition, we needed to maintain the property that the commit status itself is durably replicated through some form of replicated transaction record. We have seen systems forgo this requirement and get themselves in trouble as a result. Simply put, a system can't compromise the availability of transaction commit information without compromising the availability of the entire system.

Third, we wanted to maintain the pessimism and eager discovery of transaction conflicts that had influenced our design of transactional pipelining. Early in the development of CockroachDB, we moved away from pure optimistic concurrency control. This was due to its negative effects on the performance of contended workloads and because of the complications that requiring transaction retry loops imposed on client applications. Instead, we moved towards a model more closely aligned with traditional two-phase locking, where each statement in a transaction discovers any conflicting writes synchronously and queues behinds the

writers before proceeding. This avoids entire classes of transaction restarts.

Finally, we determined that we couldn't use any approach that required the write set for a transaction to be known before the transaction began. This sounds like a strange requirement, but requiring transactions to pre-declare which rows they intend to write to ahead of time is a common theme in various attempts to solve this problem. For example, see some of the academic solutions above, along with more exotic approaches like Thomson et al.'s Calvin: Fast Distributed Transactions for Partitioned Database Systems. Unfortunately, it doesn't work in a SQL system where transactions are interactive.

With these requirements in mind, we went back to the drawing board and developed a new atomic commit protocol that we call Parallel Commits. Let's see how it works.

Changing the commit condition

To accomplish its goal while adhering to these requirements, the Parallel Commits protocol changes the condition under which we consider a transaction to be committed. Before, we defined a committed transaction as one with a transaction record in the COMMITTED state. This meant that transactions needed to wait until all of their intent writes had succeeded before beginning the process of writing a transaction record in this state.

We can visualize the structure of a committed transaction record like:

```
TransactionRecord{ Status: COMMITTED, ... }
```

With Parallel Commits, we introduce a new transaction record state called STAGING. The meaning of this state is that the transaction has

determined all of the keys that it intends to write to, even if those writes haven't yet completed successfully. In addition to containing a STAGING status, a transaction record in this state lists the key of each of these in-progress writes. A transaction is considered committed if its record is in this state and an observer can prove that all of the writes listed in its transaction record have successfully achieved consensus.

We can visualize the structure of the new style of committed transaction record like:

```
TransactionRecord{ Status: STAGING, Writes:
[]Key{"A", "C", ...}, ... }
```

The key here is that an observer of a transaction record in the STAGING state is only allowed to consider the transaction committed if it knows that all of the transaction's writes have succeeded. This is now a distributed commit condition. For a transaction to be committed, multiple individual conditions must hold, and these conditions are no longer satisfied in a single centralized location.

The major benefit we get from this definition is that the transaction's coordinator no longer needs to wait for a transaction's writes to succeed before writing its transaction record in this state. Instead, it can pipeline each of its intent writes as SQL statements are executed and can then immediately pipeline a write to its transaction record that records all of these in-flight intent writes when a COMMIT statement arrives. Pipelining still discovers transaction conflicts eagerly and we don't require a client to declare all writes ahead of time, but the cost of the distributed consensus latency is only paid once.

Transactions only pay this cost once because the replication process for all writes, including the transaction record, is parallelized. The transaction's coordinator only begins waiting for any write to succeed

once every write has been initiated. After all of these writes have succeeded, the transaction's coordinator can immediately respond to its SQL client, informing it of the successful transaction commit.

> A transaction is considered committed if its record is in the STAGING state and an observer can prove that all of the writes listed in its transaction record have successfully achieved consensus.

Let's see what this looks like in practice, using the same SQL transaction we used earlier. As before, CockroachDB navigates SQL transactions by incrementally laying down write intents as each mutation statement is issued.

Eventually, the SQL client issues a COMMIT statement. Using the Parallel Commit protocol, CockroachDB doesn't need to wait for the intent writes to succeed before writing the commit marker to its transaction record. Instead, it immediately pipelines a write to the transaction record to move it to a STAGING status. Additionally, this write records the key of each write that the transaction has in-flight. CockroachDB then waits for all of its writes to complete the process of distributed consensus in parallel. Once they have, the transaction is considered committed and the success of the COMMIT statement is acknowledged to the SQL client.

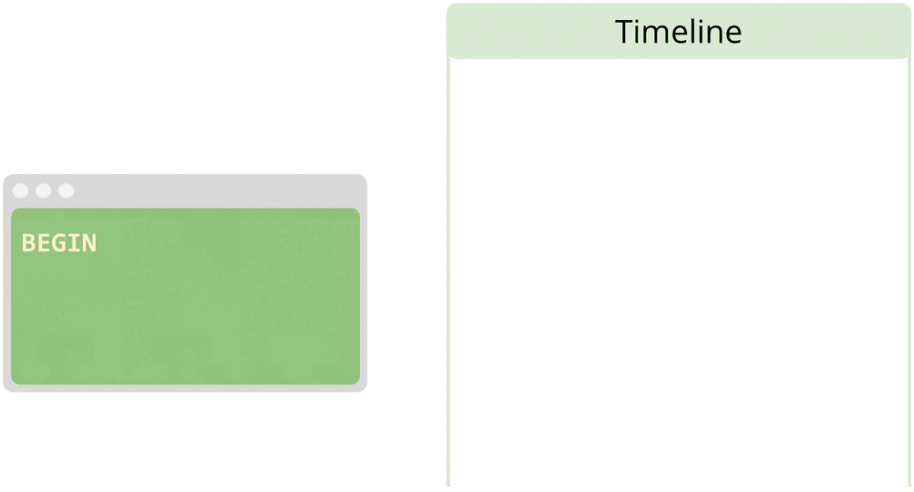

Figure 2: Transaction Timeline, With Parallel Commits

As before, once the coordinator knows that the transaction has committed, it launches an asynchronous process to clean up the transaction's intents and to delete the transaction record.

We can compare this timeline with the one from before to visualize the latency improvement provided by Parallel Commits. Instead of waiting for two synchronous round-trips of distributed consensus before acknowledging a transaction commit, CockroachDB now only waits for one.

Status recovery procedure

The transaction coordinator itself knows when each of its writes has succeeded, so it has little trouble deciding when to send a commit acknowledgment to its client. However, other transactions aren't as lucky. I said above that other observers of a transaction record in the STAGING state can only consider the transaction committed if the observer can prove that all of the transaction's writes have succeeded. We call this process the "Transaction Status Recovery Procedure", and it is the key to providing atomicity on the slow-path of Parallel Commits (e.g. in the presence of machine failures).

The procedure works as follows. When a transaction observes a write intent from a conflicting transaction, it looks up the intent's transaction record, just like it always has. The difference now is that if it finds a STAGING record, the status of the transaction is indeterminate - it could be committed or it could be aborted. To find out, the contending

transaction needs to kick off the transaction status recovery procedure.

During transaction recovery, each intent write recorded into the STAGING transaction record is consulted to see if it has succeeded. If any intents are missing, we use an in-memory data structure in CockroachDB called the “timestamp cache” to prevent the missing write from ever succeeding in the future. We then consider the transaction ABORTED. However, if all intent writes have been successfully replicated, we consider the transaction COMMITTED. Either way, at this point the observer is able to rewrite the transaction record with the new consolidated status to avoid future observers needing to go through the full recovery process.

This process is critical for determining the state of transactions while in the STAGING state. Yet, it’s fairly expensive, so we’d prefer for no observers of transactions to ever need to go through the process. We avoid this in two ways.

First, transaction coordinators asynchronously mark transaction records as COMMITTED as soon as they can. This ensures that transactions move out of the STAGING state soon after committing.

Second, transaction coordinators in CockroachDB periodically send heartbeats to their transaction record. This allows contending transactions to determine whether a transaction is still in progress. We use this notion of liveness to avoid ever kicking off the status recovery procedure for any transaction that has not expired. The intuition here is that as long as the transaction’s coordinator is still alive, it is faster to let it update its transaction record with the result of the Parallel Commit than it is to go through the whole status recovery process ourselves.

Put together, these two techniques mean that in practice the transaction status recovery procedure is only ever run in cases where the

transaction coordinator dies. This is where the procedure got its name, because its primary purpose is to recover from an untimely crash, and it is never expected to be run on the hot path of transaction processing.

Benchmark results

So with all these changes, what did we gain? Using Parallel Commits, CockroachDB is now able to commit cross-range transactions in half the time it previously was able to.

Secondary indexes in SQL provide an easy way to visualize this improvement. Over the past few releases, users of CockroachDB have repeatedly been surprised that the addition of the first secondary index to a table appeared to double the time it took to insert into that table. This was because each index in a SQL table is stored on a separate set of ranges. So the process of adding the first secondary index to a table had the effect of promoting transactions that inserted into the table from single-range transactions into full cross-range transactions. This transition forced CockroachDB to fall back from its “one-phase commit” fast-path to the standard two-phase commit protocol. This had the effect of doubling the latency of transactions once the first secondary index was added to a table.

Parallel Commit fixes this issue. Now, single-range and cross-range transactions should each take roughly the same amount of time. To demonstrate this, we performed an experiment where we ran INSERT statements to add new rows to a table with a variable number of secondary indexes. The table’s data was replicated across VMs in **us-east1-b**, **us-west1-b**, and **europe-west2-b**.

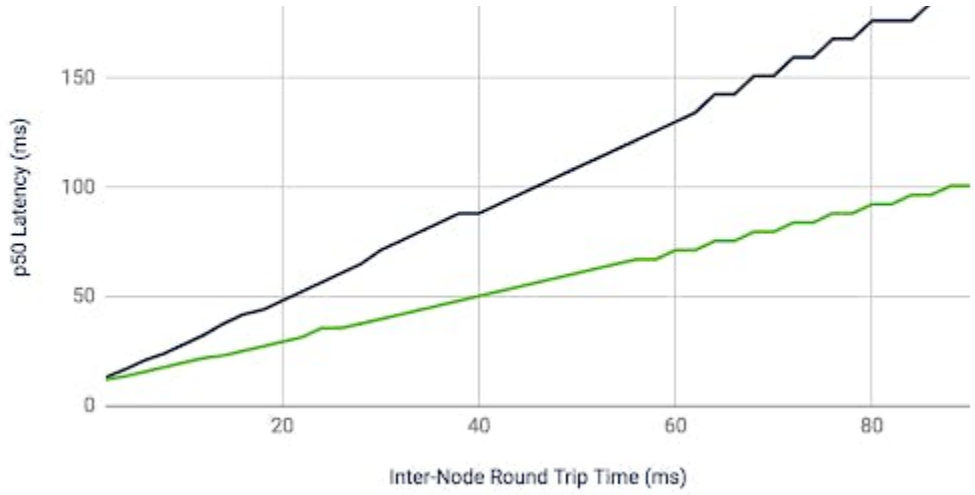

Figure 3: Transaction Latency with a Variable Number of Secondary Indexes

Figure 3 demonstrates the median latency of an INSERT statement into this table as the number of secondary indexes on the table is varied. Before Parallel Commits, there was a latency cliff when the INSERT statement's transaction was promoted to a cross-range transaction, which was exactly when the first secondary index was added. After Parallel Commits, this latency cliff disappears.

Secondary indexes aren't the only case that Parallel Commits improves. Any uncontended transaction that spans ranges should expect a similar speedup.

The TPC-C benchmark provides another avenue to explore this latency improvement. We've talked about TPC-C frequently in the past and have published guidelines for running the benchmark against CockroachDB here. At the heart of the benchmark is a transaction type called **New Order**. The **New Order** transaction simulates the process of recording a customer order into the order entry system of an arbitrary business. In doing so, it performs a series of 4 read statements and 5 write statements, modifying up to 33 rows in the process.

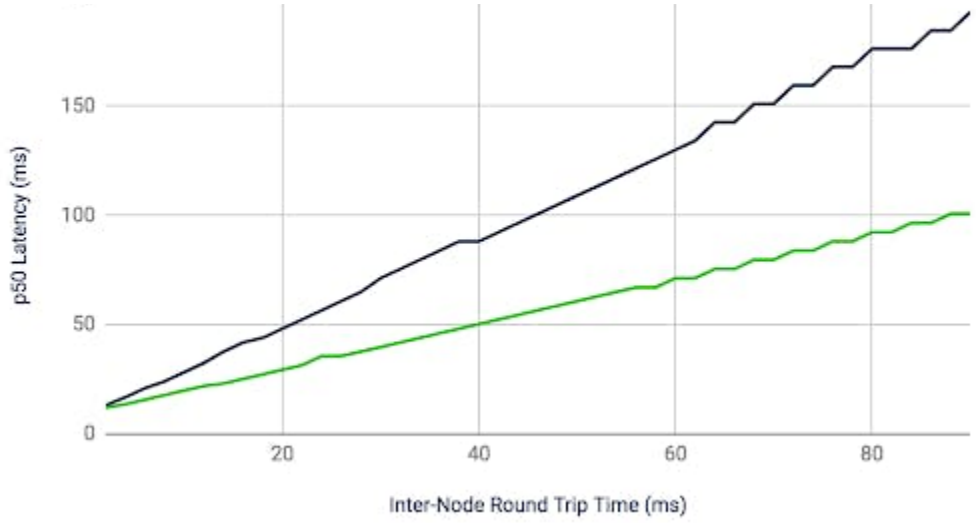

Figure 4: TPC-C New Order Transaction Latency with a Variable Inter-Node RTT

Figure 4 measures the median latency of New Order transactions in CockroachDB with and without Parallel Commits enabled. In the experiment, we replicate the TPC-C dataset across three VMs and used the `tc` linux utility to manually adjust inter-node round trip latency. As the inter-node round trip time grows in each case, we see transaction latency grow proportionally due to synchronous replication between nodes in the cluster. However, the slope of the two lines are different. Without Parallel Commits, we see the client-perceived latency of the transaction grow at twice the rate that the round trip time (RTT) grows. With Parallel Commits, we see the latency grow at exactly the same rate as the RTT.

The explanation for this is simple. With Parallel Commits, transactions have to wait for only a single round-trip of distributed consensus.

Theory and verification

As computer scientists, we weren't satisfied with an implementation that appeared to work and benchmark results that demonstrated the desired speedup. In order to claim the new commit protocol was complete, we

felt we needed to formalize the protocol in terms of existing computer science theory and verify that the protocol was sound using formal methods.

Theory

Something that has always stood out to us as unfortunate is the apparent split in the academic literature between discussion on distributed transactions and discussion on distributed consensus. Even in CockroachDB, we have fallen into the trap of separating our distributed transaction layer from our partitioned replication layer and considering them to be disjoint concerns. We speculate that this divide is because transactions are often tasked with coordinating a series of **different** changes across participants (i.e. different updates per partition) while consensus is often tasked with coordinating the **same** change across participants (i.e. replication).

During the design of our new atomic commit protocol, Parallel Commits, we were forced to take a step back and think about the process of distributed ordering and agreement holistically. This led us to attempt to unify the concepts of distributed transactions and distributed consensus under a single umbrella.

Flexible Paxos

Heidi Howard's Flexible Paxos gives us the framework to do so.

> Flexible Paxos is the simple observation that it is not necessary to require all quorums in Paxos to intersect. It is sufficient to require that the quorum used by the leader election phase will overlap with the quorums used by previous replication phases.

The implied result of this observation is that not all quorums in Paxos need to be majority quorums. Instead, quorums in Paxos can be arbitrarily sized as long as they meet certain criteria. To formalize this condition, Flexible Paxos defines the terms “Q1 quorum”, which is a leader election quorum and “Q2 quorum”, which is a replication quorum.

This generalization allows us to frame the problem of a distributed transaction run over partitioned consensus groups as one of hierarchical Paxos consensus. At the lower level of this hierarchy, each replication group performs consensus to come to an agreement about the outcome of individual writes in a transaction: “can an intent be written?”. At the upper level of this hierarchy, all replication groups perform a form of consensus to come to an agreement about the outcome of a transaction as a whole: “is the transaction committed or aborted?”. This upper-level outcome is based on the result of individual lower-level outcomes.

The intuition behind Flexible Paxos is critical here because it gives us the ability to reconfigure the quorums at each level to our needs.

At the lower (**intent-scoped**) level, we want to maximize availability of individual writes so we use majority quorums for both the leader election phase (**Q1**) and the replication phase (**Q2**) (the default Raft configuration). Each change at this level operates within the context of a single range, so the process of achieving consensus is simply the process of coordinating the change within the range’s replication group. In Raft terminology, each change must be committed to the Raft group’s log.

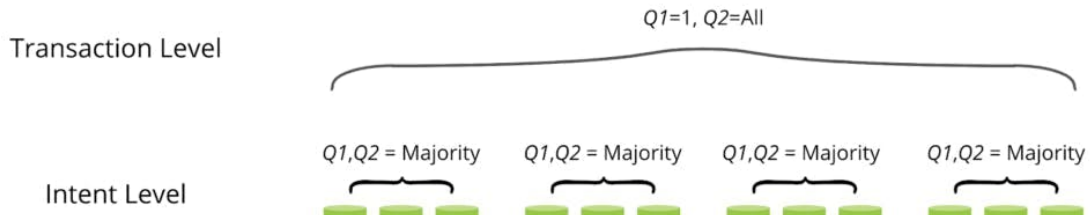

Figure 5: The Consensus Hierarchy of Transactions

At the upper (**transaction-scoped**) level, we want all nodes to be able to perform possibly conflicting transactions concurrently, so we need to avoid a "transaction leader" (**Q1**). Instead, we opt for a first phase quorum size of a single node, meaning that the first phase of consensus is resolved implicitly on a transaction's coordinator node without any external agreement. Flexible Paxos mandates that this lead to a second phase quorum (**Q2**) of all participants (i.e. all keys) in a cluster. Of course, in reality, not all transactions touch all keys, and two transactions that touch disjoint sets of keys never conflict. Expressed in a different way, the interaction between two possibly conflicting transactions can be determined fully from their interaction on mutually conflicting keys. We can use this to trivially reduce the second phase quorum size down to just those keys where the transaction actually makes changes.

This formalization is fascinating, as it reduces the problem of distributed transactions run over partitioned consensus groups to one of achieving consensus across all keys that the transaction writes to within those consensus groups. This is made possible by treating each individual intent write as a "vote" towards the commit of its transaction. This voting process is exactly what we end up doing with Parallel Commits. If all intent writes succeed in achieving consensus then they have all voted for the transaction to commit, so the transaction itself has achieved consensus.

This formalization also allows us to quickly answer a number of questions. For instance, a natural question to ask would be whether the need for all participants to vote in transaction-level consensus reduces

system availability. The answer here is that all participants in the transaction-level consensus decision are keys, which are already individually replicated for high-availability, so the system remains highly available as a whole.

Changing quorum sizes

This all leads to another question: What if we had a larger Q1 quorum at our transaction level? The Flexible Paxos equation says that if every participant voted for the Q1 quorum at the transaction level then consensus could be achieved without any remote Q2 votes. This means that we could consider a transaction committed before any of its individual writes had achieved consensus.

This seems a little out there. To get an intuition for what this would mean in practice, let's ask what it would mean to have a full Q1 quorum at the transaction level. In the terminology of Flexible Paxos, a Q1 quorum is a leadership election quorum, so we would be electing a "transaction leader". What do leaders do in distributed consensus? They assign an ordering to operations so that these operations can't conflict. In other words, the transaction leader would sequence transactions before actually executing them so that none of them would ever conflict and abort each other.

If we squint, this ends up looking a lot like a deterministic database along the lines of CalvinDB. So this two-level Paxos framework we've built here can be specialized to arrive at CockroachDB or to arrive at CalvinDB, all by changing the size of quorums. That's a pretty cool result and it raises the question of whether there are any useful configurations between these two extremes.

Verification

In addition to our desire to determine how Parallel Commits fits into the broader landscape of distributed systems theory, we also wanted to formally specify the protocol and prove its safety properties through verification. To do so, we turned to TLA+, a formal specification language developed by Leslie Lamport. TLA+ has been used to great success to verify systems and algorithms ranging from DynamoDB and S3 all the way to the Raft Consensus Algorithm used by CockroachDB.

It just so happened that around the time that we were finishing up work on Parallel Commits, we hosted an internal workshop for Cockroach Labs engineers to learn about TLA+ from Hillel Wayne, the author of **Practical TLA+**. Over the course of a week, we developed a formal specification of Parallel Commits with an associated model that asserted the atomicity and durability properties we expected from the new commit protocol.

The full specification can be found here in ParallelCommits.tla. Let's take a look at few short snippets to get a feel for the kinds of safety and liveness invariants it enforces.

The spec starts off with a few definitions.

```
ImplicitlyCommitted == /\ RecordStaging /\ \A k \in
KEYS: /\ intent_writes\[k].epoch = record.epoch /\
intent_writes\[k].ts &lt;= record.ts
```

The `ImplicitlyCommitted` operator defines what it means for a transaction to be in the new distributed commit state that Parallel Commits introduced. We read this just like we would a mathematical statement: "A transaction is implicitly committed if and only if its record

has a STAGING status and for all of its intent writes, each intent write has succeeded with the correct epoch and timestamp". This is exactly the definition we gave earlier in this blog post, only expressed using formal notation.

```
ExplicitlyCommitted == RecordCommitted
```

The `ExplicitlyCommitted` operator is more straightforward. It defines what it means for a transaction to be in the traditional centralized commit state. The only condition here is that the transaction's record must have a COMMITTED status.

The spec then defines some correctness properties that must hold for Parallel Commits to be considered correct.

```
\\* If the transaction ever becomes implicitly committed, it should \\* eventually become explicitly committed.
ImplicitCommitLeadsToExplicitCommit == ImplicitlyCommitted ~> ExplicitlyCommitted \* If the client is acked, the transaction must be committed.
AckImpliesCommit == commit_ack => ImplicitlyCommitted / ExplicitlyCommitted
```

The first of these properties, `ImplicitCommitLeadsToExplicitCommit`, is a liveness property. It uses the **leads to** temporal operator (**~>**) to assert that if a transaction ever enters the implicit commit state, it eventually enters the explicit commit state. The TLA+ model checker ensures this property holds even in the presence of transaction coordinator failures, so long as some other actor remains around to kick off the Transaction Status Recovery Procedure. The spec indicates that the transaction coordinator process can terminate at any time by marking it as an **unfair** process.

The second property, `AckImpliesCommit`, is a safety property. It uses the **implication** propositional-logic operator (**=>**) to assert that if a transaction coordinator acknowledges the success of a commit operation to its client then its transaction must be committed. Under no circumstances should the client ever be acknowledged before a transaction has committed successfully. The TLA+ model checker will enumerate the entire state space of the Parallel Commits protocol and check that this invariant holds in all of them.

Notice that the invariant doesn't say whether the transaction is implicitly or explicitly committed, just that the transaction is in either of the two states. However, the previous property ensures that the transaction will eventually make it to the explicit commit state. Additionally, another property in the spec ensures that once a transaction is committed, it stays committed, no matter what.

We found that the process of writing this specification gave us more confidence in the Parallel Commit protocol itself and in its integration into CockroachDB. Anyone with access to the TLA+ Toolbox can download the specification and model and run it with the TLA+ model checker.

COMMIT

In conclusion, Parallel Commits combines with transactional pipelining to provide interactive SQL transactions that complete in a single round-trip of distributed consensus. For geo-replicated clusters, we expect this to result in a dramatic reduction in client-observed latency. The new atomic commit protocol is enabled by default in CockroachDB v19.2, and users should upgrade when 19.2 is released to begin taking advantage of the change.

In addition to the work on Parallel Commits, we still have big plans to continue pushing the performance of transaction processing forward in CockroachDB. Over the next few releases, we intend to reduce the contention footprint of transactions, reduce write amplification of transactional writes, improve the facilities CockroachDB provides to localize data accesses within transactions, and much more. If improving the performance of strongly consistent distributed transactions interests you, check out our open positions.