

# CockroachDB: The Resilient Geo-Distributed SQL Database

February 6, 2025

## 1 Introduction

## 2 System Overview

### 2.1 Architecture of CockroachDB

Within a single node, CRDB has a layered architecture.

1. **SQL.** At the highest level is the SQL layer, which is the interface for all user interactions with the database. It includes the parser, optimizer, and the SQL execution engine, which convert high-level SQL statements to low-level read and write requests to the underlying key-value (KV) store.

In general, the SQL layer is not aware of how data is partitioned or distributed, because the layers below present the abstraction of a single, monolithic KV store.

2. **Transactional KV.** Requests from the SQL layer are passed to the Transactional KV layer that ensures atomicity of changes spanning multiple KV pairs.
3. **Distribution.** This layer presents the abstraction of a monolithic logical key space ordered by key. All data is addressable within this key space, whether it be system data (used for internal data structures and metadata) or user data (SQL tables and indexes).

CRDB uses range-partitioning on the keys to divide the data into contiguous ordered chunks of size ~64 MiB, that are stored across the

cluster. We call these chunks “Ranges”. The Distribution layer is responsible for identifying which Ranges should handle which subset of each query, and routes the subsets accordingly.

Ranges are ~64 MiB because it is a size small enough to allow Ranges to quickly move between nodes but large enough to store a contiguous set of data likely to be accessed together. Ranges start empty, grow, split when they get too large, and merge when they get too small. Ranges also split based on load to reduce hotspots and imbalances in CPU usage.

4. **Replication.**
5. **Storage.** RocksDB -> Pebble

## 2.2 Fault Tolerance and High Availability

1. **Replication using Raft.** Replicas of a Range form a Raft group.
2. **Membership changes and automatic load balancing.**
3. **Replica placement.** CRDB has both manual and automatic mechanisms to control replica placement.

## 2.3 Data Placement Policies

- Geo-Partitioned Replicas
- Geo-Partitioned Leaseholders
- Duplicated Indexes: Indexes are stored in Ranges that can be pinned to specific regions. By duplicating indexes on a table and pinning each index’s leaseholder to a specific region, the database can serve fast local reads while retaining the ability to survive regional failures.

# 3 Transaction

## 3.1 Overview

A SQL transaction starts at the *gateway* node for the SQL connection. This node interactively receives from and responds to the SQL client and acts as the transaction coordinator (orchestrating and ultimately committing/aborting the associated transaction).

### 3.1.1 Execution at the transaction coordinator

---

**Algorithm 1:** Transaction Coordinator

---

```
1 inflightOps  $\leftarrow \emptyset$ , txnTimestamp  $\leftarrow \text{now}()$ 
2 for op  $\leftarrow$  KV operation received from SQL layer
3   op.ts  $\leftarrow$  txnTimestamp
4   if op.commit
5     | op.deps  $\leftarrow$  inflightOps
6   else
7     | op.deps  $\leftarrow \{ x \in \text{inflightOps} \mid x.\text{key} = \text{op.key} \}$ 
8     | inflightOps  $\leftarrow (\text{inflightOps} - \text{op.deps}) \cup \{ \text{op} \}$ 
9   resp  $\leftarrow$  SendToLeaseholder(op)
10  if resp.ts > op.ts
11    | if op.key unchanged over (txnTimestamp, resp.ts)
12      | txnTimestamp  $\leftarrow$  resp.ts
13    | else
14      | return transaction failed
15  send resp to SQL layer
16  if op.commit
17    | asynchronously notify leaseholder to commit
```

---

Over the course of the transaction, the coordinator receives a series of requested KV operations from the SQL layer.

SQL requires that a response to the current operation must be returned before the next operation is issued. To avoid stalling the transaction while operations are being replicated, the coordinator employs two important optimizations: **Write Pipelining** and **Parallel Commits**. Write Pipelining allows returning a result without waiting for the replication of the current operation, and Parallel Commits lets the commit operation and the write pipeline replicate in parallel. Combined, they allow many multi-statement SQL transactions to complete with the latency of just one round of replication.

To enable the aforementioned optimizations, the coordinator tracks operations which may not have fully replicated yet (Line 1). It also maintains the transaction timestamp, which is initialized to the current time but may move forward over the course of the transaction. Since CRDB uses MVCC, the timestamp selects the point at which the transaction performs its reads and writes (which, thereafter, are visible to other transactions).

**Write Pipelining.** Each operation includes the key that must be read or

updated, as well as metadata indicating if the transaction should commit with the current operation. In case an operation does not attempt to commit (Line 6), it's possible to execute it immediately if it does not overlap any earlier operation (Line 7). In this way, multiple operations on different keys can be "pipelined". If an operation depends on an earlier in-flight operation, execution must wait for the earlier operation to be replicated; such dependencies introduce a "pipeline stall". The pipelining logic is outlined in Algorithm 2 (discussed below), but relies on the dependencies calculated here. Additionally, the coordinator tracks the current operation as in-flight (Line 8).

Next the coordinator sends the operation to the lease-holder for execution and waits for a response (Line 9). The response may contain an incremented timestamp (Line 10), which indicates that another transaction's read forced the leaseholder to adjust the operation timestamp. The coordinator then tries to adjust the transaction timestamp to match. This is achieved by verifying (via a round of RPCs) that repeating the previous reads in the transaction at the new timestamp will return the same value (Lines 11 and 12). If not, the transaction fails (Lines 13 and 14) and may have to be retried. **Why does the coordinator need to adjust.**

**Parallel Commits.** Now we consider what happens when the transaction wants to commit. Naively, it can only do so once all of its writes are known to have replicated, requiring at least two sequential rounds of consensus. **Like in 2PC, in the first round, the coordinator asks all nodes to prepare.**

**Once all nodes confirm they have successfully replicated the writes, the coordinator sends the final commit request.**

ChatGPT:

1. Step 1 – Replicate Writes (A): The coordinator initiates the replication of the transaction's writes to all participants. This means the data modifications or updates associated with the transaction are being sent to all replicas/nodes.
2. Step 2 – Replicate Staging Status (B): Parallel to Step 1, the coordinator also initiates the replication of the staging status (the transaction is "prepared" to be committed, but not yet fully committed). Staging status replication (B) and write replication (A) happen simultaneously (in parallel).
3. Step 3 – Confirmation: The coordinator waits for confirmation that both the writes (A) have been replicated and that the staging status

(B) has been successfully propagated to all nodes.

4. Step 4 – Commit Acknowledgment: If both the write replication (A) and the staging status replication (B) succeed, the coordinator can immediately acknowledge the commit to the SQL layer (i.e., notify the application that the transaction has been committed). At this point, the transaction can be considered “committed” to the application layer.
5. Step 5 – Final Commit Status (Asynchronous): After acknowledging the commit to the SQL layer, the coordinator asynchronously records the final, explicit commit status in the system (i.e., marking the transaction as committed in the durable store). This step is done for performance reasons and does not block the SQL commit acknowledgment.

Instead, the Parallel Commits protocol employs a staging transaction status which makes the true status of the transaction conditional on whether all of its writes have been replicated.

Instead of waiting for a separate round of consensus to ensure writes have been replicated before the commit is acknowledged, the coordinator can initiate two things in parallel:

- Replication of the Staging Status: The coordinator replicates the “staging” status across the system.
- Verification of Writes: The coordinator also verifies in parallel that all of the transaction’s writes have been replicated.

This avoids the extra round of consensus because the coordinator is free to initiate the replication of the staging status in parallel with the verification of the outstanding writes, which are also being replicated (Line 5). Assuming both succeed, the coordinator can immediately acknowledge the transaction as committed to the SQL layer (Line 15). Before terminating, the coordinator asynchronously records the transaction status as being explicitly committed (Lines 16 and 17). This is done for performance reasons.

Some talks between myself and chatgpt... Async Commit is combining A and B in essence.

Cons of Combining A and B into One Message Failure Recovery:

While it’s true that B (staging status) wouldn’t fail if A (write replication) succeeds in a combined message, it also means that if either part fails, you need to handle the failure of the entire transaction.

In Parallel Commit, by separating A and B, if A (write replication) succeeds but B (staging replication) fails, the system can retry the staging replication without needing to reapply the writes. In this combined approach, if the message fails (even due to a transient issue like network partition), you cannot simply retry just one part (writes or staging); you'd need to retry the whole transaction.

This could increase complexity in dealing with partial failures because you can't independently handle different types of failures (write failures vs. status failures).

Limited Flexibility in Recovery:

By combining A and B, you lose the flexibility to independently manage the two separate concerns (write replication and staging status replication). If B (staging) fails for some reason (e.g., network failure), you might have data inconsistencies because some nodes may have applied the writes but not received the staging status. In the current Parallel Commit protocol, since A (writes) and B (staging status) are independent, the system can still allow recovery actions without causing data corruption. No Graceful Rollback:

If you combine A and B, the message is effectively a single "commit" operation. If the combined message fails or is lost, you'd need to retry the entire operation from the beginning. In Parallel Commit, if only B fails, it's possible to just retry the staging replication without needing to repeat the entire transaction, because the writes are already replicated. This is a more efficient way to handle certain types of failures. Transaction Atomicity Beyond A and B:

While combining A and B makes the commit process more atomic (in the sense that either both succeed or both fail), you are essentially "locking" the entire commit process into a single step. This could be less flexible in certain failure scenarios, especially when partial failures occur and you might want to independently handle different parts of the transaction (like retrying just the commit status replication, if the writes have already been applied).

Specifically we verified atomicity by asserting that every staging transaction was eventually either explicitly committed or aborted, regardless of coordinator failure, and no clients were told otherwise.

### 3.1.2 Execution at the leaseholder

---

**Algorithm 2: Leaseholder**

---

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

---

It acquires latches on the keys of *op* and all the operations *op* depends on (Line 3), thus providing mutual exclusion between concurrent, overlapping requests.

Once the initial checks are complete, the leaseholder evaluates the operation to determine what data modifications are needed in the storage engine without actually making the changes (Line 7).

This results in a low level *command* detailing the necessary changes, as well as a response for the client

Note that Algorithm 2 does not delve into any details about the various scenarios that may occur during the evaluation phase (Line 7).

This is the period of time when a transaction may encounter uncommitted writes from other transactions or writes so close in time to the transaction's read timestamp that it is not possible to determine the correct order of transactions. The next sections discuss these scenarios, and how CRDB guarantees both atomicity and serializable isolation.

## 3.2 Atomicity Guarantees

An atomic commit for a transaction is achieved by considering all of its writes provisional until commit time. CRDB calls these provisional values **write intents**. An intent is a regular MVCC KV pair, except that it is preceded by meta-data indicating that what follows is an intent. This metadata

points to a transaction record, which is a special key (unique per transaction) that stores the current disposition of the transaction: *pending*, *staging*, *committed* or *aborted*.

The transaction record serves to atomically change the visibility of all the intents at once, and is durably stored in the same Range as the first write of the transaction. For long-running transactions, the coordinator periodically heartbeats the transaction record in the pending state to assure contending transactions that it is still making progress.

Upon encountering an intent, a reader follows the indirection and reads the intent's transaction record. If the record indicates that the transaction is committed, the reader considers the intent as a regular value (and additionally deletes the intent metadata). If the transaction is aborted, the intent is ignored (and cleanup is performed to remove it). If the transaction is found to be pending (indicating that the transaction is still ongoing), then the reader blocks, waiting for it to finalize.

If the coordinator node fails, contending transactions eventually detect that the transaction record has expired, and mark it aborted. If the transaction is in the staging state (which indicates that the transaction has either been committed or aborted, but the reader is unsure which), the reader attempts to abort the transaction by preventing one of its writes from being replicated. If all writes are already replicated, the transaction is in fact committed, and is updated to reflect that.

### 3.3 Concurrency Control

1. *Write-read conflicts*. A read running into an uncommitted intent with a lower timestamp will wait for the earlier transaction to finalize. Waiting is implemented using in-memory queue structures. A read running into an uncommitted intent with a higher timestamp ignores the intent and does not need to wait.
2. *Read-write conflicts*. A write to a key at timestamp  $t_a$  cannot be performed if there's already been a read on the same key at a higher timestamp  $t_b \geq t_a$ . CRDB forces the writing transaction to advance its commit timestamp past  $t_b$ .
3. *Write-write conflicts*. A write running into an uncommitted intent with a lower timestamp will wait for the earlier transaction to finalize. If it runs into a committed value at a higher timestamp, it advances its timestamp past it.



### 3.4 Read Refreshes

Advancing a transaction’s read timestamp from  $t_a$  to  $t_b > t_a$  is possible if we can prove that none of the data that the transaction read at  $t_a$  has been updated in the interval  $(t_a, t_b]$ . If the data has changed, the transaction needs to be restarted. If no results from the transaction have been delivered to the client, CRDB retries the transaction internally. If results have been delivered, the client is informed to discard them and restart the transaction.

To determine whether the read timestamp can be advanced, CRDB maintains the set of keys in the transaction’s read set (up to a memory budget). A “read refresh” request validates that the keys have not been updated in a given timestamp interval (Algorithm 1, Lines 11 to 14). This involves re-scanning the read set and checking whether any MVCC values fall in the given interval. This process is equivalent to detecting the rw-antidependencies that PostgreSQL tracks for its implementation of SSI. Similar to PostgreSQL, our implementation may allow false positives (forcing a transaction to abort when not strictly necessary) to avoid the overhead of maintaining a full dependency graph.

Advancing the transaction’s read timestamp is also required when a scan encounters an uncertain value: a value whose timestamp makes it unclear if it falls in the reader’s past or future (see Section 4.2). In this case we also attempt to perform a refresh. Assuming it is successful, the value will now be returned by the read.

### 3.5 Follower Reads

CRDB allows non-leaseholder replicas to serve requests for read-only queries with timestamps sufficiently in the past through a special ‘AS OF SYSTEM TIME’ query modifier.

To enable this functionality safely, a non-leaseholder replica asked to perform a read at a given timestamp  $T$  needs to know that no future writes can invalidate the read retroactively. It also needs to ensure that it has all the data necessary to serve the read. These conditions mean that if a follower read at timestamp  $T$  is to be served, the leaseholder must no longer be accepting writes for timestamps  $T' \leq T$ , and the follower must have caught up on the prefix of the Raft log affecting the MVCC snapshot at  $T$ .

To this end, each leaseholder tracks the timestamps of all incoming requests and periodically emits a closed timestamp, the timestamp below which no further writes will be accepted. Closed timestamps, alongside Raft log indexes at the time, are exchanged periodically between replicas. Follower

replicas use the state built up from received updates to determine if they have all the data needed to serve consistent reads at a given timestamp. For efficiency reasons the closed timestamp and the corresponding log indexes are generated at the node level (as opposed to the Range level).

Every node keeps a record of its latency with all other nodes in the system. When a node in the cluster receives a read request at a sufficiently old timestamp (closed timestamps typically trail current time by ~2 seconds), it forwards the request to the closest node with a replica of the data.

## **4 Clock Synchronization**

### **4.1 Hybrid-Logical Clocks**

Each node within a CRDB cluster maintains a hybrid-logical clock. Physical time is based on a node's coarsely-synchronized system clock, and logical time is based on Lamport's clocks

## **5 Problems**

1. 3.1.1.

## **6 References**

### **References**