RoachFest

# Build better.
# Dream bigger.

#RoachFest22

# CockroachDB architecture

Ben Darnell, Chief Architect and Co-Founder of Cockroach Labs

# Agenda

- **Introduction**
- Design Goals and Tradeoffs
- Cluster Architecture
- CockroachDB Software Stack
- Other Topics

# Which came first?

The table, or the index?

# Agenda

- Introduction
- **Design Goals and Tradeoffs**
- Cluster Architecture
- CockroachDB Software Stack
- Other Topics

# Scalable SQL

# Additional Design Goals

**SQL**
- Familiar interface
- Broad feature set
- Tool compatibility

**Transactions**
- Important for users and internal mechanisms
- Highest level of isolation
- Optimized for distributed usage

**Distribution**
- No node holds all the data
- Place data for efficiency and other requirements

**Replication**
- Hardware faults are inevitable at scale
- Fast, seamless recovery

**Storage**
- Efficiently manage data on disk

# Design Trade-Offs

| Consistency over Availability | Guarantee consistency at all costs.<br>In CAP theorem parlance, CockroachDB is aCP system. |
| --- | --- |
| **Favor Transactional Workloads** | Transactional workloads are favored over other types of workloads, such as analytics. |

# Agenda

- Introduction
- Design Goals and Tradeoffs
- **Cluster Architecture**
- CockroachDB Software Stack
- Other Topics

# One executable. Three roles

- SQL Gateways parse and plan
- SQL Executors run the query
- KV nodes hold the data

PostgreSQL
Protocol

RPC

Node 1
SQL Gateway
SQL Executor
KV

Node 2
SQL Gateway
SQL Executor
KV

Node 3
SQL Gateway
SQL Executor
KV

Node 4
SQL Gateway
SQL Executor
KV

# Agenda

- Introduction
- Design Goals and Tradeoffs
- Cluster Architecture
- **CockroachDB Software Stack**
- Other Topics

# CockroachDB Software Stack

| Layer | Responsibilities |
| --- | --- |
| SQL Layer | Process PostgreSQL wire protocol requests<br>Parse and optimize SQL statements<br>Map SQL to key-value operations |
| Transaction Layer | Enforce ACID semantics and serializable isolation<br>Optimize for distributed transactions<br>Handle transaction conflicts |
| Distribution Layer | Manage ranges and leases<br>Provide multi-region support |
| Replication Layer | Replicate data while ensuring consistency<br>Achieve consensus using the Raft protocol |
| Storage Layer | Implements key-value storage using LSM trees (Pebble)<br>MVCC, caching, garbage collection |

SQL Gateway

SQL Executor

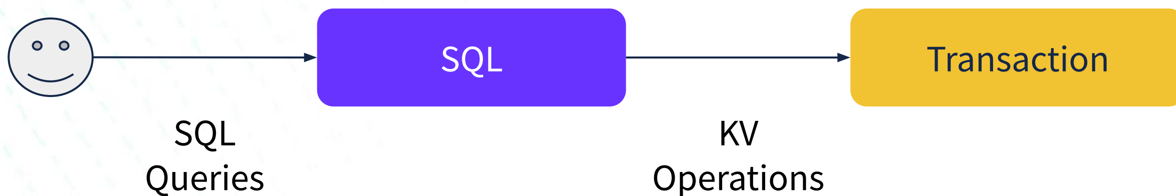
KV

# Agenda

- Introduction
- Design Goals and Tradeoffs
- Cluster Architecture
- **CockroachDB Software Stack**
  - **SQL Layer**
  - Transaction Layer
  - Distribution Layer
  - Replication Layer
  - Storage Layer
- Other Topics

# SQL Components

- **SQL API**: SQL user interface, ANSI SQL standard, ACID-semantic transactions
- **Parser**: Converts SQL text into an abstract syntax tree (AST)
- **Cost-based optimizer**: Converts the AST into an optimized logical query plan
- **Physical planner:**
  Converts the logical query plan into a physical query plan for execution by one or more nodes in the cluster
- **SQL execution engine:**
  Executes the physical plan by making read and write requests to the underlying key-value store




See SQL Layer documentation

SQL

# Cost-Based Optimizer

After a SQL query parsed into an abstract syntax tree (AST), the cost-based optimizer seeks the lowest cost for a query, usually related to time.

Cost is roughly calculated by:

- Time: estimating how much time each node will use to process all results
- Data Flows: modeling how data flows through the query plan

The most important factor in determining the quality of a plan is cardinality (i.e., the number of rows). The fewer rows each SQL operator needs to process, the faster the query will run.

Table statistics are used to find the optimal execution plan. These are collected periodically or when table data changes significantly.

See Cost-Based Optimizer documentation

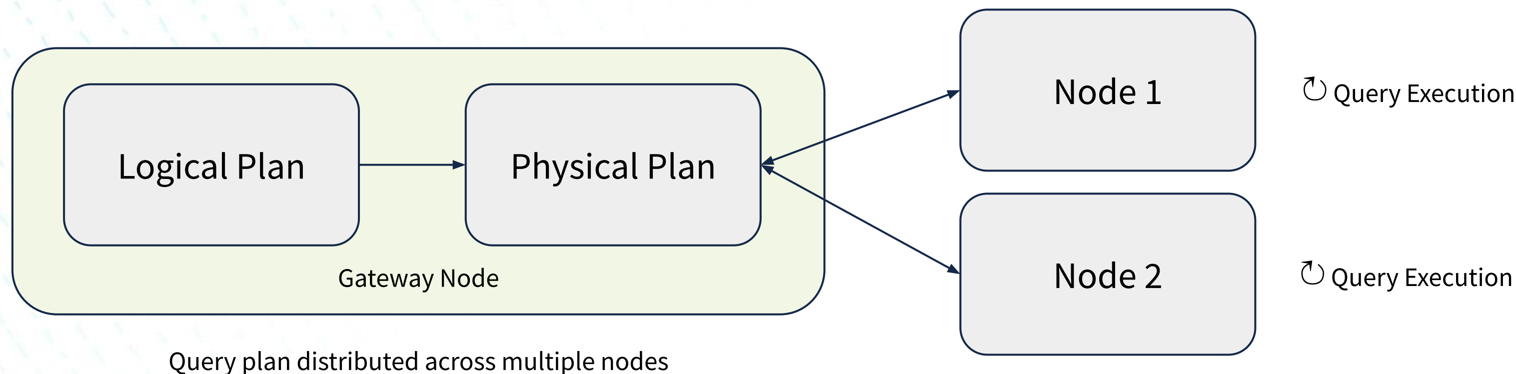
# Physical Planning

Physical planning determines which nodes will participate in the execution of the query.

This allows some computations are performed closer to where the data is stored.

SQL operators behave identically whether planned in gateway or distributed mode.

Small number of rows = executed on gateway node
Larger number of rows = distributed across multiple nodes




Query plan distributed across multiple nodes

SQL

# Distributed SQL (DistSQL)

DistSQL is an approach that moves computation closer to where the data is stored, enabling:

1. Remote-side filtering
2. Remote-side updates and deletes
3. Distributed SQL operations, such as joins, aggregation and sorting

```
SELECT id
FROM inventory
WHERE price < 5.00;
```

| ID | Name | Price |
|---|---|---|
| 1 | Bat | 1.11 |
| 2 | Ball | 2.22 |
| 3 | Glove | 3.33 |
| 4 | Hat | 4.44 |



| ID | Name | Price |
|---|---|---|
| 1 | Bat | 1.11 |
| 2 | Ball | 2.22 |
| +10k rows w/ price >= 5.00 | | |

| ID | Name | Price |
|---|---|---|
| 10003 | Glove | 3.33 |
| 10004 | Hat | 4.44 |
| +10k rows w/ price >= 5.00 | | |

See Distributed SQL (DistSQL) RFC

SQL

# Vectorized Execution Engine

The vectorized execution engine converts row-oriented data into in-memory column-oriented structures.

This dramatically improves performance by processing each component of a query plan on type-specific batches of column data.

```
SELECT sum(price)
FROM inventory;
```

| ID | Name | Price |
|---|---|---|
| 1 | Bat | 1.11 |
| 2 | Ball | 2.22 |
| 3 | Glove | 3.33 |
| 4 | Hat | 4.44 |

Result: 11.00

Node 1

CockroachDB Process

Gateway Node

Result: 3.33

Result: 7.77

Node 2

CockroachDB Process

Range 1

Node 3

CockroachDB Process

Range 2

Leaseholder Nodes

Vectorized Processing

Row format

| ID | Name | Price |
|---|---|---|
| 1 | Bat | 1.11 |
| 2 | Ball | 2.22 |

Column format

| ID | Price |
|---|---|
| 1 | 1.11 |
| 2 | 2.22 |

sum()

Vectorized Processing

| ID | Name | Price |
|---|---|---|
| 3 | Glove | 3.33 |
| 4 | Hat | 4.44 |

| ID | Price |
|---|---|
| 3 | 3.33 |
| 4 | 4.44 |

sum()

See Vectorized Execution Engine documentation

SQL

# Converting SQL to Key-Values

Each entry in the KV store has the following structure:

```
/Table/<table-id>/<index-id>/<index-key-value>/<column-family>
```

```
CREATE TABLE inventory (
  id INT PRIMARY KEY,
  name STRING,
  price FLOAT);
```

| ID | Name | Price |
|---|---|---|
| 1 | Bat | 1.11 |
| 2 | Ball | 2.22 |
| 3 | Glove | 3.33 |

| Key (pretty) | Key (encoded) | Value (pretty) |
|---|---|---|
| /Table/inventory/primary/1/primary | /Table/62/0/1/0 | "Bat",1.11 |
| /Table/inventory/primary/2/primary | /Table/62/0/2/0 | "Ball",2.22 |
| /Table/inventory/primary/3/primary | /Table/62/0/3/0 | "Glove",3.33 |

See Encoding tech note

SQL

# Column Families

```
CREATE TABLE inventory (
  id INT PRIMARY KEY,
  name STRING,
  price FLOAT,
  photo BLOB
  FAMILY f1 (name, price),
  FAMILY f2 (photo)
);
```

Column Families are used to store groups of columns in separate KV entries.

Infrequently used large columns can be excluded from queries improving cache performance.

Concurrent operations on separate column families do not interfere with each other.

| ID | Name | Price | Photo |
|---|---|---|---|
| 1 | Bat | 1.11 | /xff/xd8 … |

| Key | Value |
|---|---|
| /Table/inventory/primary/1/f1 | "Bat",1.11 |
| /Table/inventory/primary/1/f2 | /xff/xd8 … |

See Column Family RFC

SQL

# Non-Unique Indexes

```
CREATE TABLE inventory (
  id INT PRIMARY KEY,
  name STRING,
  price FLOAT,
  INDEX name_idx(name)
);
```

The key for a non-unique index includes the table and index name, the key-value and the primary key-value.

There is no “value” by default.

| ID | Name | Price |
|---|---|---|
| 1 | Bat | 1.11 |
| 2 | Ball | 2.22 |
| 3 | Glove | 3.33 |

| Key | Value |
|---|---|
| /Table/inventory/name_idx/”Bat”/1 | ∅ |
| /Table/inventory/name_idx/”Ball”/2 | ∅ |
| /Table/inventory/name_idx/”Glove”/3 | ∅ |

SQL

# Unique Indexes

```
CREATE TABLE inventory (
  id INT PRIMARY KEY,
  name STRING,
  price FLOAT,
  UNIQUE INDEX name_uidx(name)
);
```

For a unique index, the KV value defaults to the value of the primary key.

| ID | Name | Price |
|---|---|---|
| 1 | Bat | 1.11 |
| 2 | Ball | 2.22 |
| 3 | Glove | 3.33 |

| Key | Value |
|---|---|
| /Table/inventory/name_uidx/"Bat" | /1 |
| /Table/inventory/name_uidx/"Ball" | /2 |
| /Table/inventory/name_uidx/"Glove" | /3 |

SQL

# Inverted Indexes

```
CREATE TABLE inventory (
  id INT PRIMARY KEY,
  data JSONB,
  price FLOAT,
  INVERTED INDEX data_idx(data)
);
```

Inverted Indexes allow indexed searches into values included in arrays or JSON documents.

Key-values include the JSON path and value together with the primary key.

| ID | Data |
| --- | --- |
| 1 | {“name”: “Bat”, “price”: 1.11} |
| 2 | {“name”: “Ball”, “price”: 2.22} |

| Key | Value |
| --- | --- |
| /Table/inventory/data_idx/name/”Bat”/1 | ∅ |
| /Table/inventory/name_idx/price/1.11/1 | ∅ |
| /Table/inventory/data_idx/name/”Ball”/2 | ∅ |
| /Table/inventory/name_idx/price/2.22/2 | ∅ |

SQL

# Covering Indexes - Storing Clause

```
CREATE TABLE inventory (
  id INT PRIMARY KEY,
  name STRING,
  price FLOAT,
  INDEX name_sidx(name)
    STORING (price)
);
```

The STORING clause can be used to store additional columns in the value part of the KV index structure.

This can improve the performance of queries on columns that are in the STORING clause, at a small cost to write performance and storage space.

| ID | Name | Price |
|---|---|---|
| 1 | Bat | 1.11 |
| 2 | Ball | 2.22 |

| Key | Value |
|---|---|
| /Table/inventory/name_sidx/”Bat”/1 | 1.11 |
| /Table/inventory/name_sidx/”Ball”/2 | 2.22 |

SQL

# Online Schema Changes

CockroachDB performs schema changes using a protocol that allows tables to remain online (i.e., able to serve reads and writes) during the schema change.

This protocol allows different nodes in the cluster to asynchronously transition to a new table schema at different times.

The schema change protocol decomposes each schema change into a sequence of incremental changes that will achieve the desired effect.

See [Online Schema Changes documentation]

# Key Points

- The SQL layer converts SQL queries into optimized logical query plans.
- Logical query plans are converted into physical plans that can be executed on any node.
- DistSQL and the vectorized execution engine parallelize execution.
- SQL is converted to key-values based on the underlying KV system.
- Schema changes can be applied while continuing to serve read/write requests.

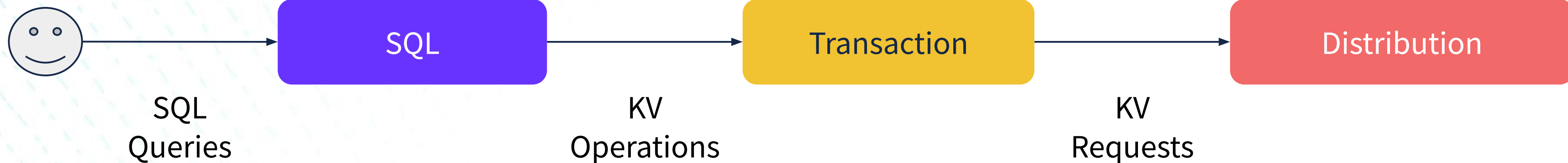
# Agenda

- Introduction
- Design Goals and Tradeoffs
- Cluster Architecture
- **CockroachDB Software Stack**
  - SQL Layer
  - **Transaction Layer**
  - Distribution Layer
  - Replication Layer
  - Storage Layer
- Other Topics

Transaction

# Functions / Responsibilities

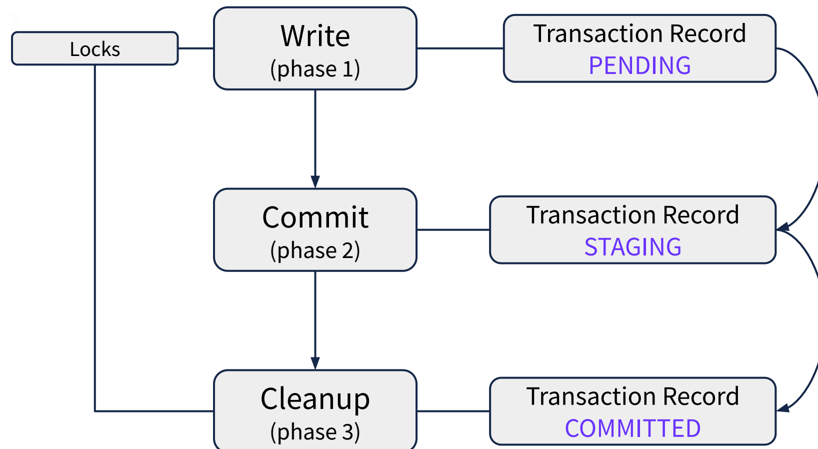
- Receives incoming KV operations generated by the SQL Layer
- Ensures the atomicity of transactions
    - Transactions are either committed or aborted. There is no middle state.
- Maintains serializable isolation from other transactions
    - Transactions are run concurrently but appear as if only one was run at a time.
- Sends KV Requests to the Distribution Layer



Transaction

# High-Level Phases

- **Writes and Reads (Phase 1)**
  - For writes, locks (e.g., write intents) are created and transaction records are established.
  - For reads and writes, transaction conflicts are negotiated.
- **Commits (Phase 2)**
  - Check transaction record
  - Handle aborted transactions
  - Change record state to STAGING
- **Cleanup (Phase 3 - asynchronous)**
  - Move transaction state to COMMITTED
  - Resolves and deletes write intents




Typical Write Operation

Transaction

# Write Intents

Values that are written in a provisional state to the storage layer.

The combination of a replicated lock and a replicated provisional value are recorded.

Operations that encounter write intents look up the status of the transaction in the transaction record.

| Key | Timestamp | Value |
|---|---|---|
| A<intent> | 500 | “proposed” |
| A | 400 | “current” |
| A | 322 | “old” |
| A | 50 | “original” |

Write Intent in MVCC

Transaction

# Transaction Records

Transaction records are used to track the progress or transactions. They are written to the same range as the first key in the transaction.

Transactions records can have the following states:

- PENDING: transaction is still in progress
- STAGING: used during parallel commits to indicate that a transaction may be committed
- COMMITTED: transaction is committed
- ABORTED: transaction was aborted

```
INSERT INTO inventory
(name, price)
VALUES ('Bat', 1.11),
('Ball', 2.22)
```

# Concurrency Control

The concurrency manager sequences incoming requests and provides isolation between transactions.

- Latch Manager
  - Sequences incoming requests and provides isolation between those requests
- Lock table
  - Provides both locking and sequencing of requests (in concert with the latch manager)
  - Per-node, in-memory data structure
  - Pulls in information about external locks, such as write intents, when they are discovered
- Uses pessimistic locking via SQL using the SELECT for UPDATE statement
  - Can increase throughput and decrease tail latency for contended operations

Transaction

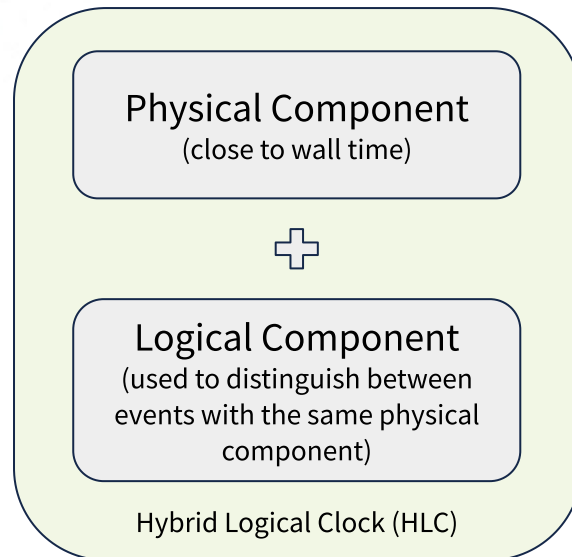
# Time and Hybrid Logical Clocks

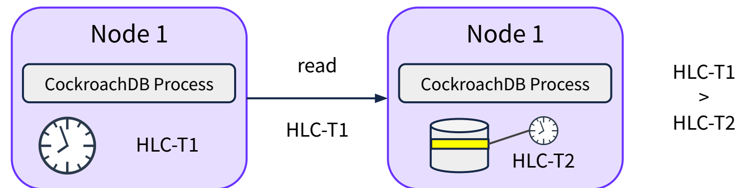
The gateway node picks a timestamp for the transaction using HLC time.

When nodes send requests to other nodes, they include the timestamp generated by their local HLCs.

When nodes receive requests, they inform their local HLC of the timestamp supplied with the event by the sender.

These are used to ensure that the transaction reading the data is at an HLC time greater than the value it's reading (i.e., the read always happens "after" the write).



Transaction

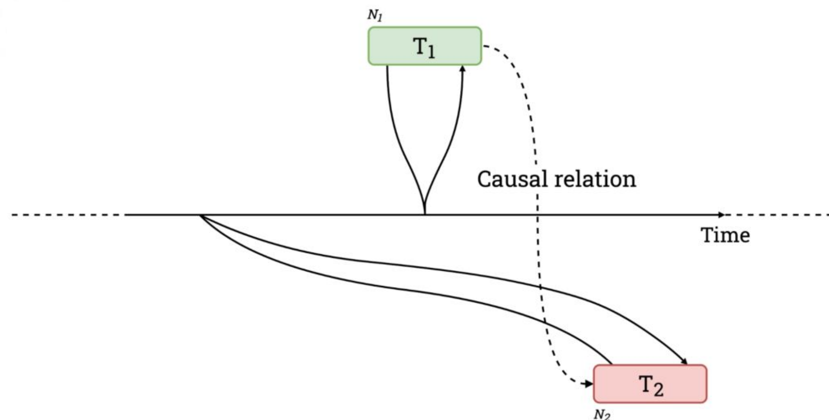
# Clock Synchronization

To ensure serializability of transactions, node clocks must be reasonably synchronized.

If node's clock gets too far out of sync with at least half the other nodes, the CockroachDB process will crash itself.

In 22.1, the default max offset is 500ms, although customers often lower this to 250ms or less.

It is recommended that all nodes are synchronized to the same time source (or one that implements leap second smearing in the same way).




Causally related transactions committing out of order due to unsynchronized clocks.

See Living Without Atomic Clocks blog post

Transaction

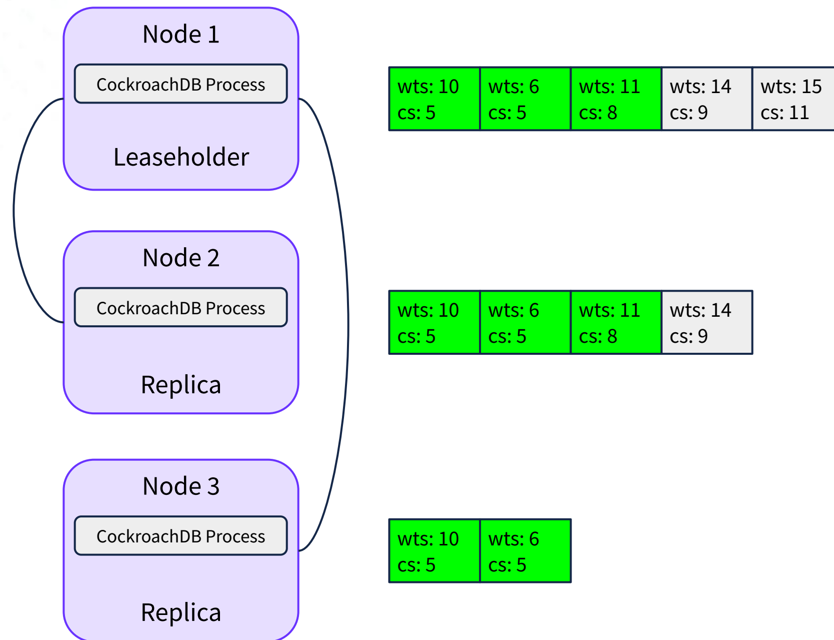
# Closed Timestamps

Each range tracks a closed timestamp, the time at which no writes can occur at or below.

The closed timestamp is advanced continuously on the leaseholder and sent to its replicas.

Generally, this occurs a few seconds in the past.

This allows replicas to serve reads at or below the closed timestamp for the range (used in follower reads).




Write timestamps (wts) and closed timestamps (cs) sent via raft commands. Green entries are known to be committed. The write timestamp must be higher than the closed timestamp in the previous entry.

See An Epic Read on Follower Reads - Closed Timestamps

# Transaction Pipelining

Transactional writes are pipelined when being replicated and when being written to disk. This dramatically reduces the latency of transactions that perform multiple writes.

Write intents are replicated from leaseholders in parallel, so the waiting all happens at the end, at transaction commit time.

(see diagram on next slide)

See Transaction Pipelining documentation

Transaction

# Transaction Pipelining

```
BEGIN;
INSERT INTO inventory
(name, price)
VALUES ('Bat', 1.11);
INSERT INTO inventory
(name, price)
VALUES (Ball, 2.22);
COMMIT;
```



See [Transaction Pipelining documentation]

Transaction

# Parallel Commits

Parallel Commits are part of an optimized atomic commit protocol that cuts the commit latency of a transaction in half, from two rounds of consensus down to one.

Under this atomic commit protocol, the transaction coordinator can return to the client eagerly when it knows that the writes in the transaction have succeeded.

It introduced the transaction status of STAGING and the recording of in-flight writes.

This allows the transaction record to be written as soon as all values are known, in parallel with individual writes.

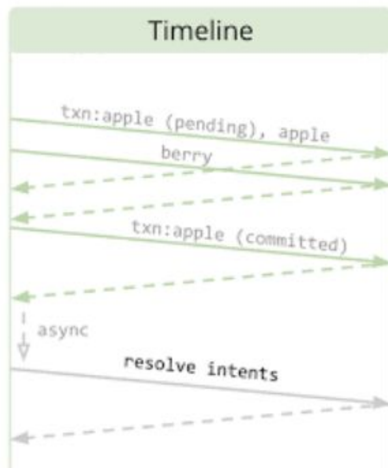
Transaction Record
—
Status: STAGING
InFlightWrite: [“write1”, “write2”, “write3”]

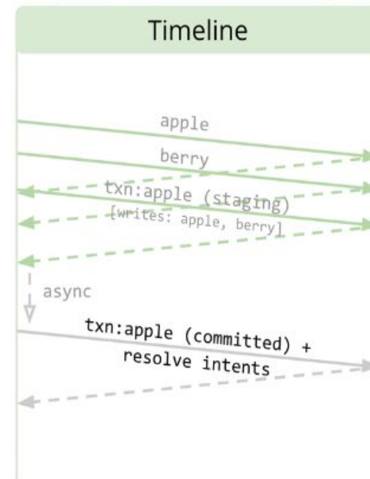
(see diagram on next slide)

See Parallel Commits documentation and blog post

Transaction

# Parallel Commits




**Without Parallel Commits**
Transaction record can only be written AFTER the individual writes are committed

**With Parallel Commits**
Transaction record is written as soon as all values are known (after COMMIT), even if individual writes have not yet returned

See Parallel Commits documentation and blog post

# Transaction Conflicts

When a transaction encounters a write intent, the following conflicts can occur:

- Write/write, where two PENDING transactions create write intents for the same key
- Write/read, where a read encounters an existing write intent with a timestamp less than its own.

Additionally, the following types of conflicts that do NOT involve encountering a write intent can arise:

- Write after read, when a write with a lower timestamp encounters a later read.
- Read within uncertainty window, when a read encounters a value with a higher timestamp but it's ambiguous whether the value should be in the future or in the past of the transaction because of possible clock skew.

# Key Points

- The transcription layer ensures ACID-compliance of transactions.
- Concurrency control mechanisms ensure the proper sequencing and isolation of transactions.
- Optimizations, such as pipelining and parallel commits reduce latency.
- Conflicts can be resolved internally or through client retries.

# Agenda

- Introduction
- Design Goals and Tradeoffs
- Cluster Architecture
- **CockroachDB Software Stack**
  - SQL Layer
  - Transaction Layer
  - **Distribution Layer**
  - Replication Layer
  - Storage Layer
- Other Topics

Distribution

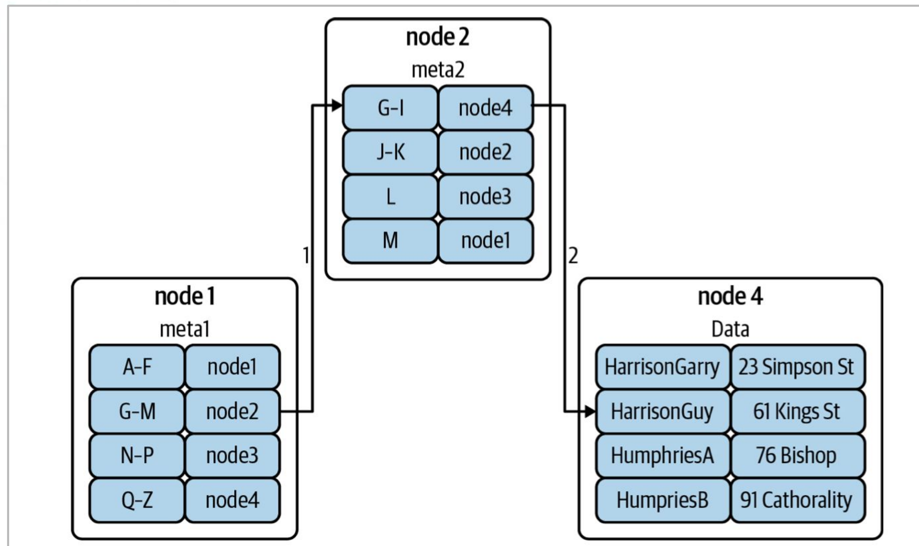
# Distribution Layer

- Receives incoming KV requests from the transaction layer on the same node
- Data is stored in a monolithic sorted map of key-value pairs
  - Describes all data in the structure, including its location
  - Provides simple lookups and efficient scans
- Distribution layer breaks data into ranges of approximately 512 MB
  - Keeps the number of ranges per node manageable
  - Keeps the ranges distributed throughout the cluster but still representable as a single logical entity
- Identifies which nodes should receive the request, then sends the request to the proper node's replication layer

Distribution

# Meta Ranges

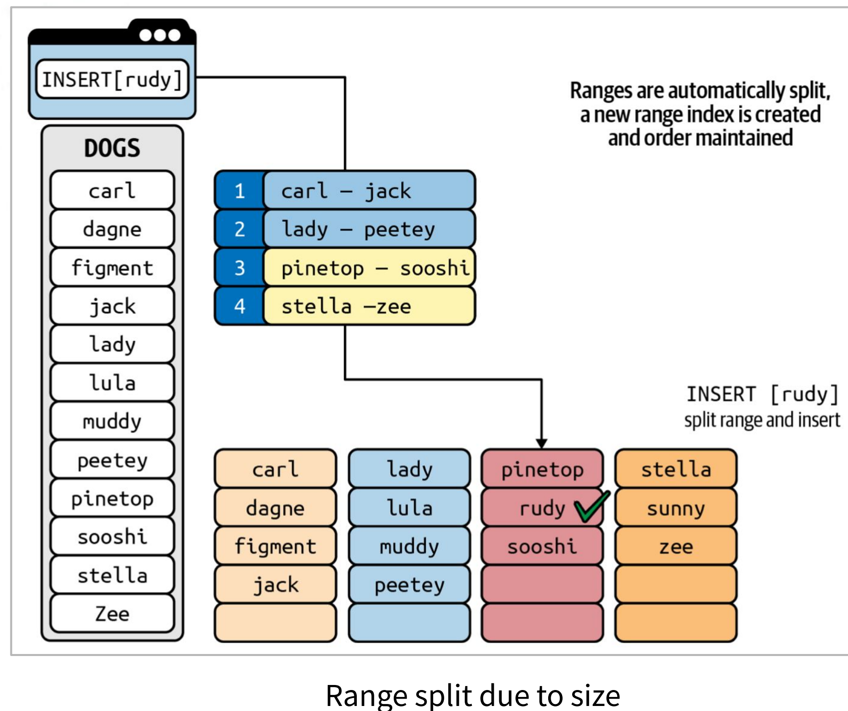
- **Distribution of ranges is stored in global keyspaces `meta1` and `meta2`**
  - `meta1` can be considered a "range of ranges" lookup that points to the node holding the `meta2` range
  - The `meta2` range points to the nodes holding every copy of every range within the "range of ranges"
- **Gossip protocol is used to share ephemeral information between nodes**
  - Maintains eventually consistent KV map
  - Used for bootstrapping (meta0)



Distribution

# Range Splits

- **CockroachDB attempts to keep a range to less than 512 MB**
  - Balance the ease of operations on a range (e.g., splitting, moving) with the bookkeeping overhead for each range.
- **Ranges are split when the max size is exceeded**
  - Results in 2 ranges of the same size
- **Ranges can be split based on load**
  - `kv.range_split.by_load` must be enabled
  - Queries per second must exceed the `kv.range_split.load_qps_threshold`
  - The split must result in a better load distribution
- **Ranges can also be merged**




Range split due to size

Distribution

# Multi-Region Distribution

- **Geo-partitioning allows data to be located within specific geographic regions**
  - May be desirable from a latency perspective
  - Can be used for data sovereignty
- **Multi-region configuration controls how data is distributed across regions**
  - Cluster regions: geographic region specific to a given node at start time
  - Regions: may have multiple zones
  - Databases within a cluster are assigned to one or more regions, one of these is the primary region.
  - Tables within a database can have locality rules (global, regional by table, regional by row) that determines how data is distributed
  - *Survival goals* dictate how many simultaneous failures a database can gracefully tolerate
- **Survival Goals**
  - Zone: database will remain fully operational when a **zone** goes down (default)
  - Region: database will remain fully operational when a **region** goes down

# Key Points

- The distribution layer maintains the monolithic key-value map that locates data throughout the cluster.
- Data is split up into ranges of approximately 512 MiB.
- Ranges can be split or merged as they grow larger or smaller.

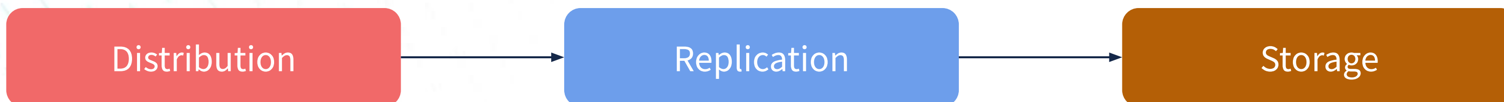
# Agenda

- Introduction
- Design Goals and Tradeoffs
- Cluster Architecture
- **CockroachDB Software Stack**
  - SQL Layer
  - Transaction Layer
  - Distribution Layer
  - **Replication Layer**
  - Storage Layer
- Other Topics

Replication

# Replication Layer

- Receives requests from and sends responses to the distribution layer
- Provides high availability
  - Multiple copies of the data ensures availability when infrastructure failures occur
  - Considered multi-active (as compared to active-passive or active-active) since all replicas can serve data
- Maintains consistency across all ranges in the cluster
  - Each range is replicated independently of other ranges
  - Uses the Raft (distributed consensus) protocol to guarantee consistency
- Writes accepted requests to the storage layer



# Raft Consensus Protocol

To ensure that data is replicated correctly, the replication layer uses the raft consensus protocol.

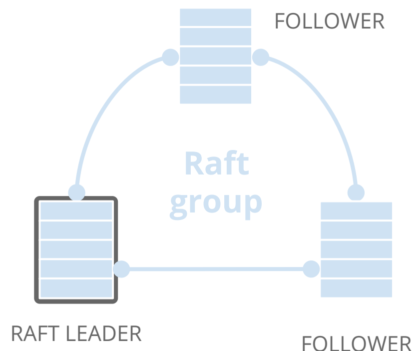
Raft leverages the following components:

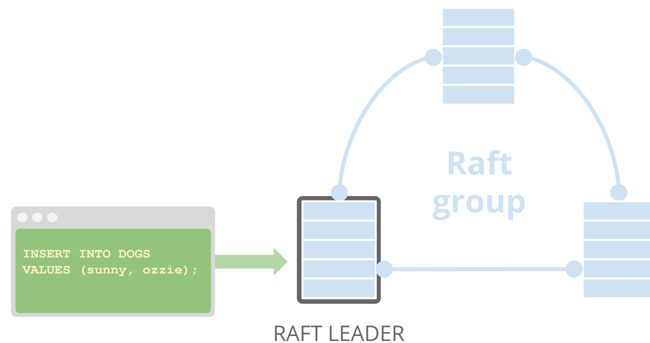
- Raft Group: Each range has its own raft group, a minimum of 3 nodes.
- Raft Leader: All proposed changes go through a single elected leader.
- Raft Log: All changes are recorded as an immutable log.

Leader election occurs when a node fails to receive a heartbeat from the leader.

Snapshots can be used to up-replicate to new nodes or enable a lagging node to quickly catch-up to the raft log.

See The Secret Lives of Data for a high-level Raft demo.



# Raft and Leaseholders

- Consensus reads are expensive.
- Rather than use consensus reads, CockroachDB has the concept of a leaseholder that coordinates the following actions:
  - Performs strong reads (i.e., current / non-stale)
  - Proposes changes to the Raft Leader
- CockroachDB will attempt to elect a leaseholder who is also a Raft leader.
  - This reduces latency for proposing changes to the Raft leader since it will be on the local node.

# Key Points

- The replication layer ensures that data is replicated with consistency across the cluster.
- Raft is the distributed consensus protocol used by the replication layer.
- Raft leaders are typically co-located on the same node as leaseholder to reduce latency for proposed changes.

# Agenda

- Introduction
- Design Goals and Tradeoffs
- Cluster Architecture
- **CockroachDB Software Stack**
  - SQL Layer
  - Transaction Layer
  - Distribution Layer
  - Replication Layer
  - **Storage Layer**
- Other Topics

Storage

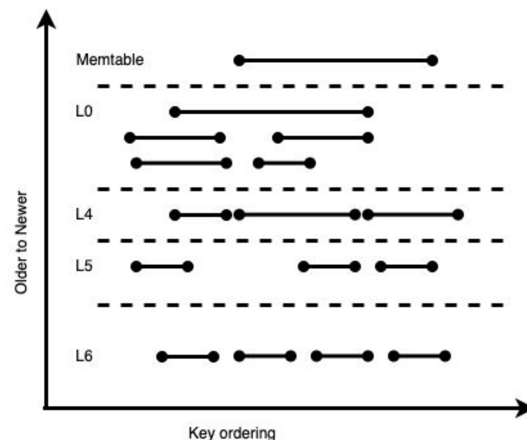
# Storage Layer

- Serves successful reads and writes from the replication layer
- Physical implementation of the KV storage engine
- Uses the Cockroach Labs developed Pebble storage engine
  - Open source KV store
  - Inspired by LevelDB and RocksDB
  - Maintain primarily by Cockroach labs
  - Optimized for CockroachDB

Storage

# Log-Structured Merge (LSM) Trees

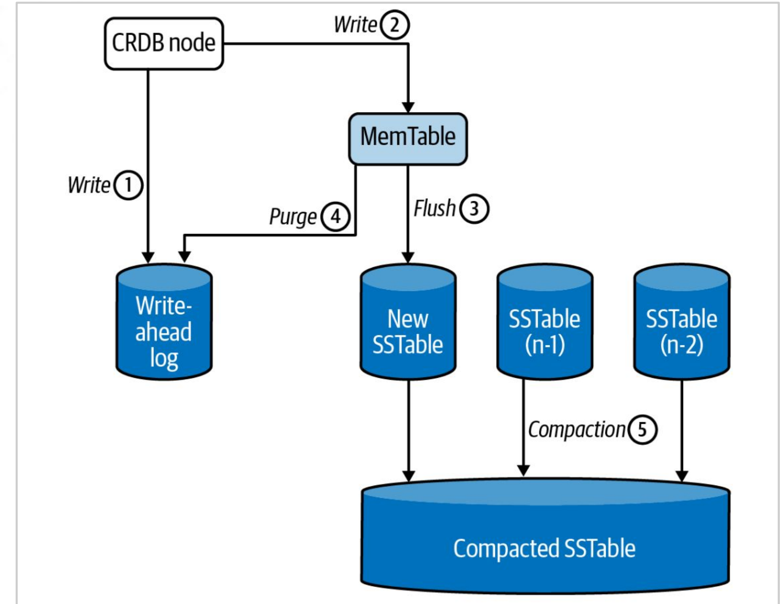
- Pebble implements a log-structured merge tree
- File format is a Sorted String Table (SSTable)
- SSTables exist on multiple levels, numbered L0 to L6
    - L0 (MemTable) contains an unordered set of SSTables, which receives new values.
    - Compaction: Periodically, SSTables are compacted into larger consolidated stores in the lower levels
    - L1 to L6: SSTables are ordered and nonoverlapping so only one SSTable per level could possibly hold a given key
- SSTables are internally sorted and indexed
- Writes are fast since the SSTables are append-only
- Write-ahead log (WAL) ensures that data is not lost in node failure



Storage

# Log-Structured Merge (LSM) - Writes

1. Writes from higher CRDB layers are applied to the write-ahead log (WAL)
2. Then applied to the MemTable
3. Once MemTable reaches a certain size, it is flushed to disk to create a new SSTable
4. WAL can be purged after the MemTable is flushed to disk
5. Multiple SSTable are routinely merged (compacted) into larger SSTables

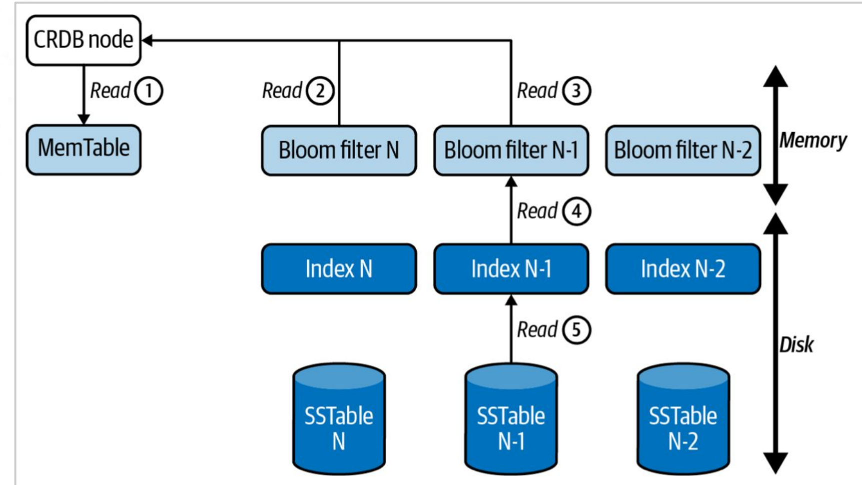



LSM Writes

Storage

# SSTables & Bloom Filters - LSM Reads

- **SSTables are indexed but …**
  - Searching through every SSTable index can be expensive with a large number of tables
  - *Bloom Filters* reduce number of lookups
- **Bloom Filters**
  - Compact and quick-to-maintain structure
  - Indicate whether an SSTable *might* contains a key
  - Indicate whether an SSTable does not contain a key (can be skipped)




**LSM Reads**. First, MemTable is checked, then the bloom filters from the L0 layer down. If a bloom filter returns a possible match, the index is examined and result returned

Storage

# Multi-Version Concurrency Control (MVCC)

- CockroachDB implements multi-version concurrency control (MVCC)
  - Readers can get a consistent view of information even as it is being modified
- Multiple versions of any row are maintained by the system
  - Transactions determine which version of the row to read based on their timestamp and the timestamp of any concurrent transactions

MVCC Store with Intent on Key A

| Key | Timestamp | Value |
|---|---|---|
| A<intent> | 500 | "proposed_value" |
| A | 400 | "current_value" |
| A | 322 | "old_value" |
| A | 50 | "original_value" |
| B | 100 | "value_of_b" |

Storage

# Delete and Updates

- SSTables are immutable
- Inserts and Updates
  - New values are inserted and added to the MemTable (L0) and the write-ahead log (WAL) before being flushed to disk as SSTables.
  - When the system retrieves records, it reads from youngest to oldest, so the most recent values is retrieved.
- Deletes
  - Similar to inserts and updates, deletes write a new value, called a tombstone marker, to the MemTable (L0), which is later flushed to disk as an SSTable.
- Compaction
  - This reduces fragmentation of rows across SSTables.
  - Tombstone markers are retained until they are compacted to the base level L6.
- Garbage Collection
  - Older records are removed after the gc.ttlseconds elapses unless covered by *protected timestamps* (e.g., part of a backup job process).

# Key Points

- Data is stored as key-value pairs on disk using the Pebble storage engine.
- Pebble uses a log structured merge (LSM) tree to manage data storage.
- SSTs are the on-disk representation of sorted lists of key-value pairs.
- Garbage collection regularly collects MVCC values to reduce the size of data on disk.

# Agenda

- Introduction
- Design Goals and Tradeoffs
- Cluster Architecture
- CockroachDB Software Stack
- **Other Topics**
  - **Admission Control**
  - Change Data Capture
  - Monitoring and Alerting

# Admission Control

- Maintains cluster performance and availability when some nodes experience high load.
- Request and response operations are sorted into work queues by priority, giving preference to higher priority operations.
- Acts per-node or per-store
- Manages the following resources:
  - CPU (via running Goroutines)
  - Store health (Pebble's L0 sub-levels and file count)
- NOT managed:
  - Memory
  - Disk bandwidth (e.g., IOPS)
- Allows full utilization of the resource, but limits over-utilization

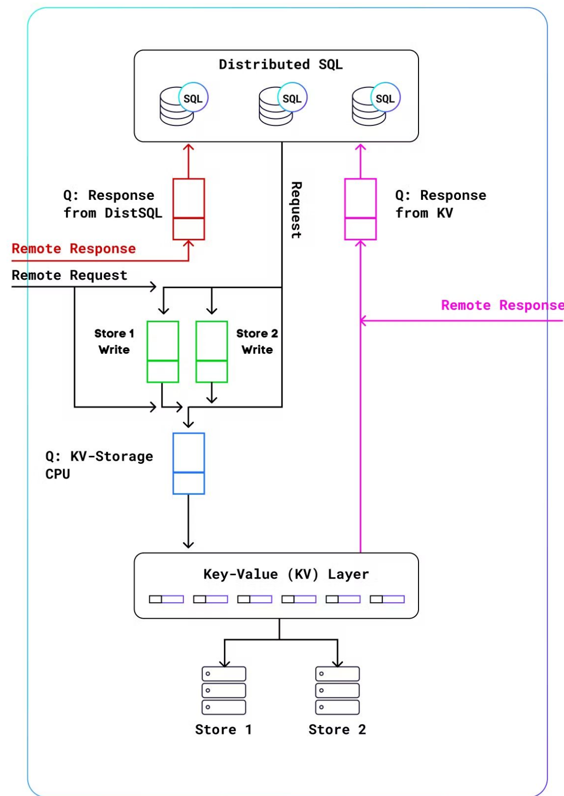
See Admission Control documentation and "Here's how CockroachDB keeps your database from collapsing under load"

# Admission Control Queues

Managed queues:

- Requests to KV
  - Consumes CPU
  - Does store reads/writes
- Response from KV
  - Consumes CPU
- Response from DistSQL
  - Consumes CPU

Work is queued until there is an available slot or token.



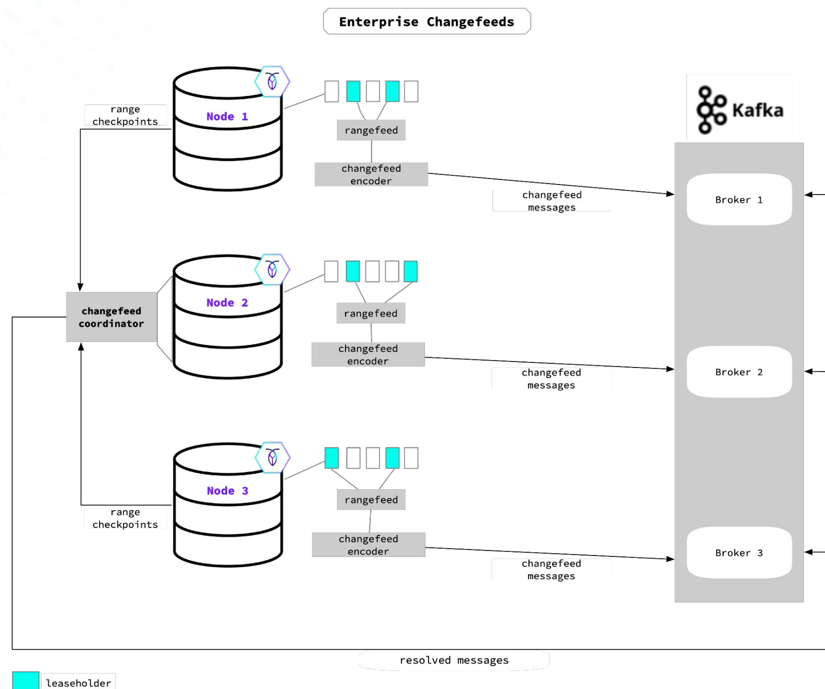

Q is an admission.WorkQueue

# Agenda

- Introduction
- Design Goals and Tradeoffs
- Cluster Architecture
- CockroachDB Software Stack
- **Other Topics**
  - Admission Control
  - **Change Data Capture**
  - Monitoring and Alerting

# Change Data Capture

- Efficient, distributed, row-level change feeds into configurable sinks
- Used for downstream processing such as reporting, caching, data warehousing or full-text indexing
- Not a low-latency publish-subscribe mechanism
- At least once delivery guarantee
- Frequently used to integrate with analytics platforms
- Support sinks: Kafka, Google Cloud Pub/Sub, S3, Azure Storage, Google Cloud Storage, and webhooks




See Change Data Capture documentation and "What is Change Data Capture?" blog post

# Agenda

- Introduction
- Design Goals and Tradeoffs
- Cluster Architecture
- CockroachDB Software Stack
- **Other Topics**
  - Admission Control
  - Change Data Capture
  - **Monitoring and Alerting**

# Monitoring and Alerting

CockroachDB provides a number of monitoring and alerting services.

- DB Console / Admin UI
  - Essential metrics about the cluster's health, including historical graphs
  - Hot range monitoring
  - SQL query observability and metrics
- Prometheus endpoint
  - Granular time series metrics in a format easy for exporting to prometheus.
- Health endpoints
  - Used to ensure that the node is healthy
- Cluster API
  - REST API provides information about the cluster and nodes
- Logging System
  - Provides detailed logs published via channels to a range of sinks, such as files, Fluentd-compatible collectors and HTTP network collectors

See Monitoring and Alerting documentation

# Additional Resources

- Official documentation
- Cockroach University
- CockroachDB, the Definitive Guide (O'Reilly book)

Questions?