

# CockroachDB Architecture

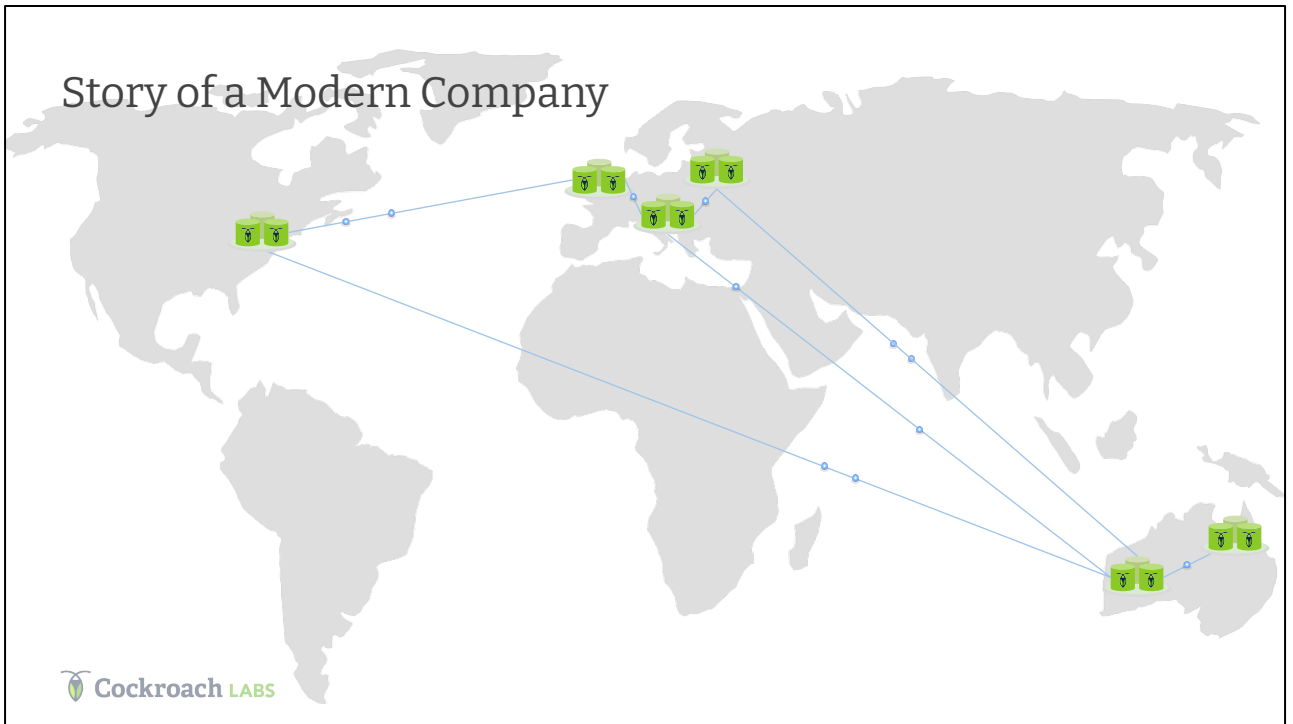
Oliver Tan  
(Slides adapted from Rebecca Taft)



## Story of a Modern Company

- Core markets in Europe and/or US, growing in Australia/Asia
- Strategic migration to cloud DBMS
- Data locality for GDPR and end-user latency
- Users expect database to always be available
- Ease of use without building their own system
- Consistency and ACID!

First let me tell you a story of a modern company. This company has core markets in Europe and Australia, and a growing market in the US. In order to support this global user base, they have made the strategic decision to migrate to a cloud database management system. But they have a number of requirements they need to consider. As a global company, they need fine-grained control over data placement to ensure compliance with local regulations such as GDPR. Their users expect an “always on” experience, so the database must be fault tolerant and highly available. Finally, to simplify application development, the database must support strongly consistent SQL. This is in fact a real company, and as you might expect, I’m telling you this story because the database they chose was CockroachDB.



This graphic shows the plan for their CockroachDB deployment. This is just one company, but we designed CockroachDB to support workloads like this because we think their requirements are becoming more and more common.

# Survive Anything Thrive Anywhere



<click> like cockroaches!

# What is CockroachDB?

- "NewSQL" database
  - SQL frontend, non-traditional backend
- PostgreSQL wire compatible but scalable underneath!
- **Survive anything, thrive everywhere.**



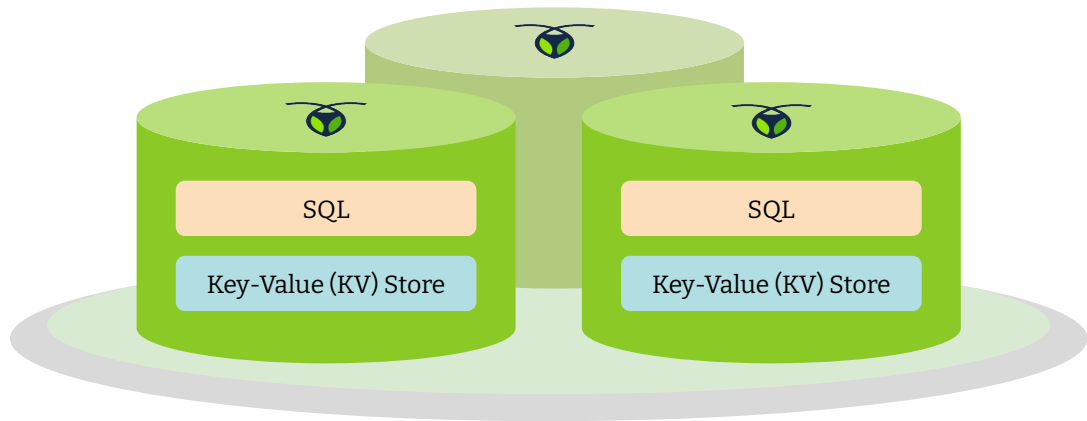
## AGENDA

- Introduction
- Ranges and Replicas
- Transactions
- SQL Data in a KV World
- SQL Execution
- SQL Optimization
- Evaluation



There are many challenges with supporting workloads like this, so today I'll give you an overview of CockroachDB that will hopefully give some insight into how we support these requirements. Here is the agenda for this talk. There is a lot to cover, so let's get started.

# Architecture of CockroachDB



Before I begin, you need to understand a bit about the architecture of the system. CockroachDB is a shared nothing system and consists of a distributed SQL layer on top of a distributed key value store. I'll start by focusing on the key value store and talk about the SQL layer later in the talk.

# Monolithic Key Space

DOGS
carl
dagne
figment
jack
lady
lula
muddy
peetey
pinetop
sooshi
stella
zee

## Monolithic logical key space

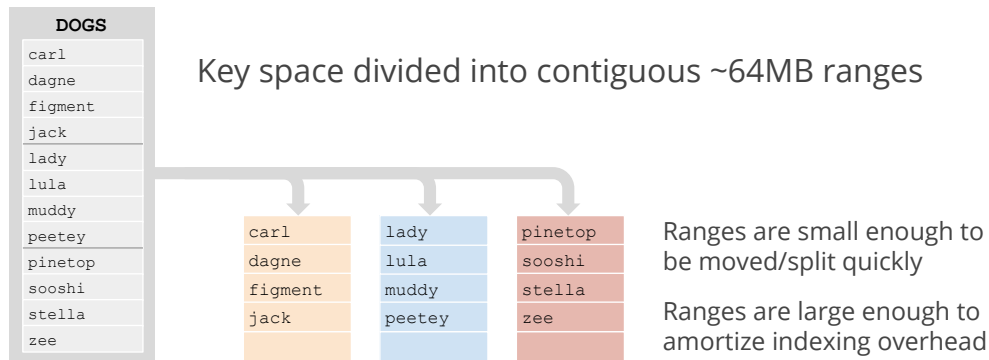
- Keys and values are strings
- Ordered lexicographically by key
- Multi-version concurrency control (MVCC)



Logically, the data in the key-value store is laid out in a single monolithic, ordered keyspace. For example, here is a database of dogs, which I'll use as a running example throughout this talk. Notice that for simplicity I'm only showing the keys here, but you can imagine the real database would also include values with other data about the dogs. I'm also only showing a single version here, but CockroachDB uses MVCC, so we actually store multiple versions of each key-value pair.



# Ranges



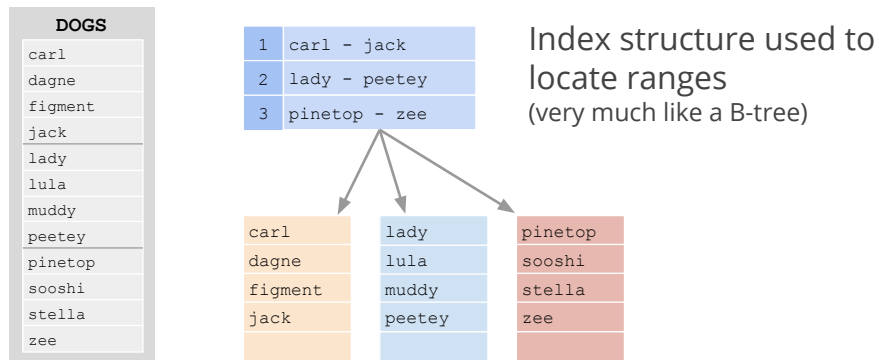
The logical key space is physically realized by dividing the it into ranges of contiguous keys, which we call “Ranges” with a capital R. Ranges are ~64 MB in size, though the size is configurable. Ranges start empty, grow, split when they get too large, and merge with their neighbors when they get too small.

Ranges are physically stored in a per-node key-value store. We currently use RocksDB for our storage layer, but we’re in the process of changing that.

Data is placed into ranges using an order-preserving data distribution. We **do not** use a hashing scheme such as consistent hashing. This is an extremely important design decision as keeping the key space ordered imposes implementation complexity. But note that keeping the key space ordered is more or less required in order to implement SQL, so there wasn’t much choice in the matter.

Within a range, keys are ordered by storing them within a per-node key-value store which maintains ordering. We use RocksDB.

# Range Indexing



In order to maintain the ordering between ranges, a range indexing structure is needed. We store this indexing structure inside a set of well-known ranges that lie in a special part of the key space known as the system key space. There seems to be a chicken & egg problem with this approach. How do we find these index ranges? These index ranges also need an index, and we store that index in a single range: the very first range. This provides a two-level indexing scheme that is similar to BigTable, HBase, and Spanner. The location of the first range is thus the root of the tree for all data stored in a CockroachDB cluster.

# Ordered Range Scans

DOGS	
carl	
dagne	
figment	
jack	
lady	
lula	
muddy	
peetey	
pinetop	
sooshi	
stella	
zee	

1	carl - jack
2	lady - peetey
3	pinetop - zee

carl	lady	pinetop
dagne	lula	sooshi
figment	muddy	stella
jack	peetey	zee

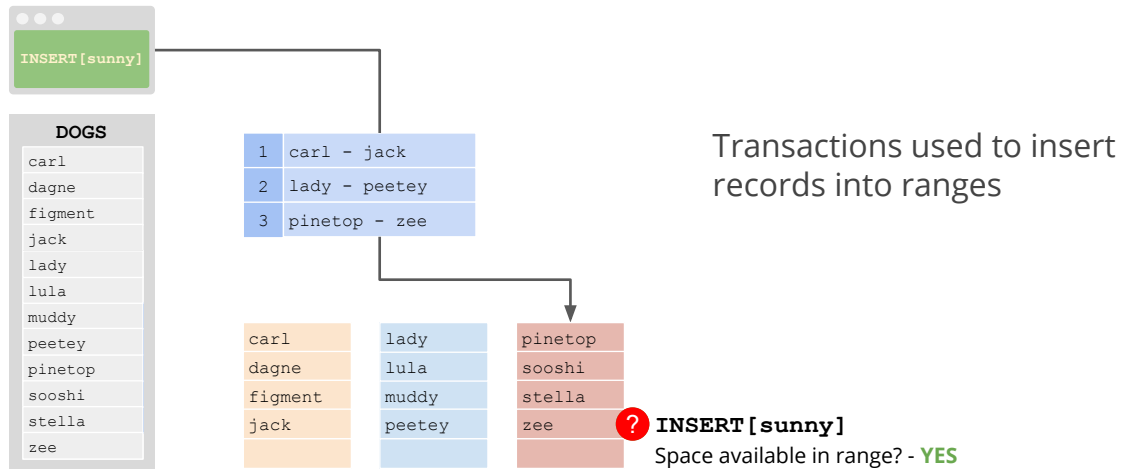
Ordered keys enable efficient range scans

`dogs >= "muddy" AND <= "stella"`



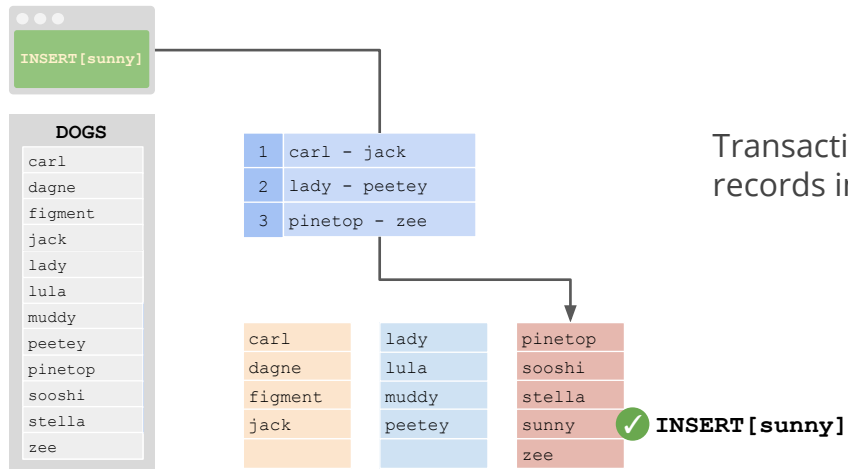
Fully ordered keys enable efficient range scans that span ranges. Here is a depiction of scanning for dog names between “muddy” and “stella”. Notice that the scan touches two ranges and we can easily locate those two ranges from the indexing structure.

# Transactional Updates



Transactions are used to insert and delete data. These are distributed transactions, that can touch an arbitrary set of ranges. I'll cover transactions in more detail in a bit.

# Transactional Updates

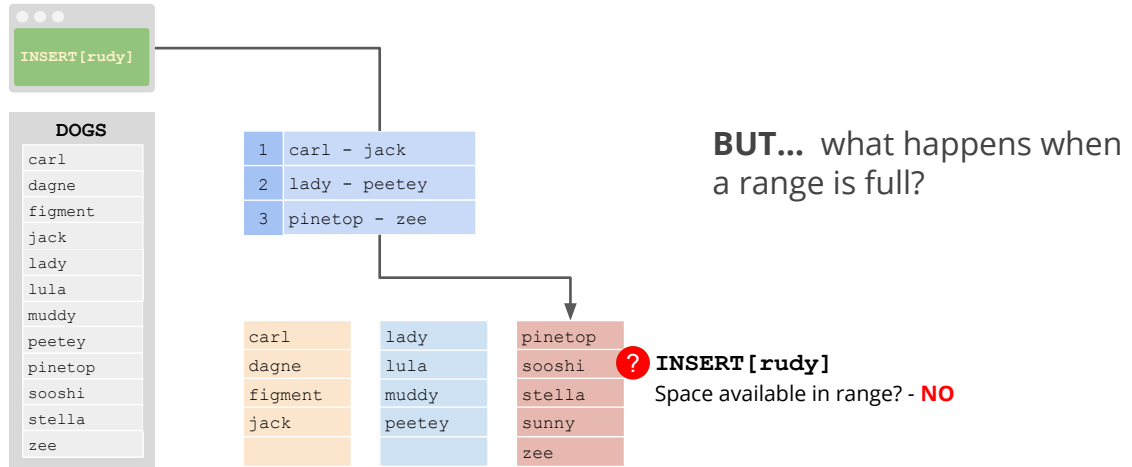


Transactions used to insert records into ranges



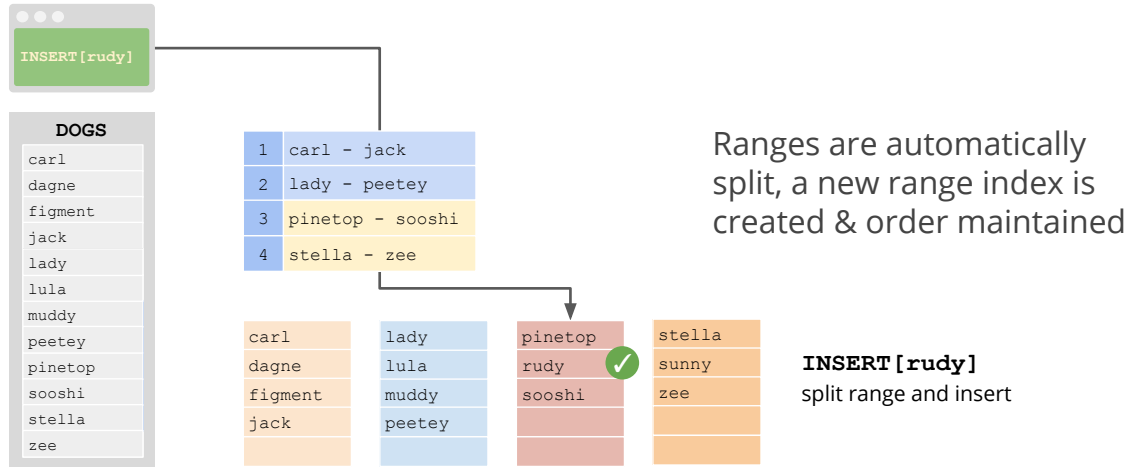
If a range is not full, an insert simply adds the key to the range.

## Range Splits



When a range becomes full, it is split into two ranges.

## Range Splits

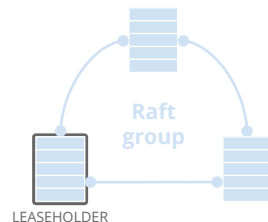


The old range still exists, and approximately half the data is moved into a new range. Note that we also had to update the indexing structure. This is done using the same distributed transactions that are used elsewhere in the system.

This description of range splits is slightly inaccurate, as ranges don't have a fixed size. In reality, we allow the insert to be performed which causes the range to grow beyond its target size, then we asynchronously split it. And reality also contains a back pressure mechanism that kicks in when ranges exceed their size, in order a range getting pounded with writes from becoming excessively large before it asynchronously splits.

## Ranges are the unit of replication

- Each Range is a Raft (consensus) group
  - Default to 3 replicas, but configurable
- Raft provides “atomic replication” of writes
  - Proposed by the leaseholder (Raft leader)
  - Accepted when a quorum of replicas ack

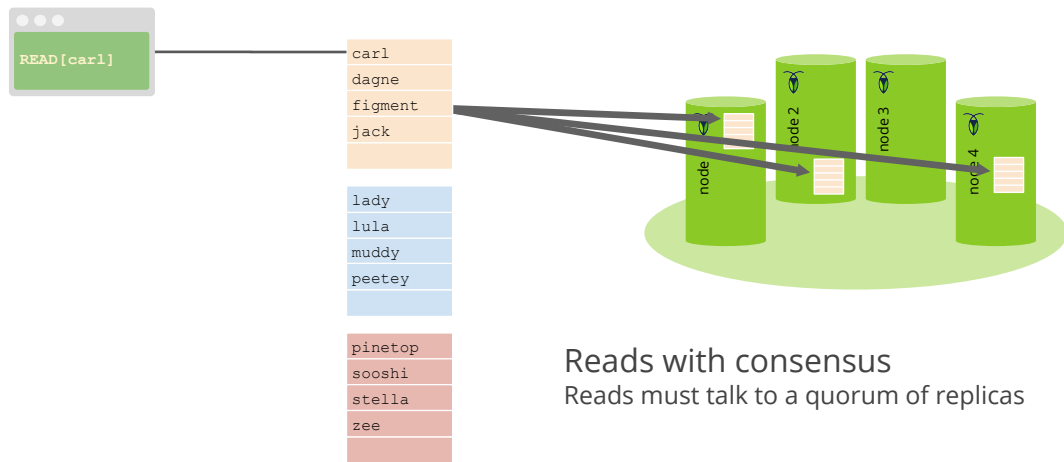


Ranges are not only how we store data, but they are an important concept in CockroachDB since they are also the unit of replication. We use the Raft consensus protocol for replication, and each Range is a Raft group. I just want to emphasize that: each Range is a Raft group because we do not replicate at the level of nodes. We replicate at the level of ranges. This gives us fine grained control over data placement, and also allows us to control the replication factor on a per range basis to make certain important ranges more fault tolerant.

A distributed consensus algorithm like Raft provides a useful building block which is “atomic replication”. Write commands are proposed by the leaseholder (which for the purposes of this talk you can assume is the same as the raft leader) and distributed to the followers, but only accepted when a quorum of replicas have written the command to disk.



## Range Leases

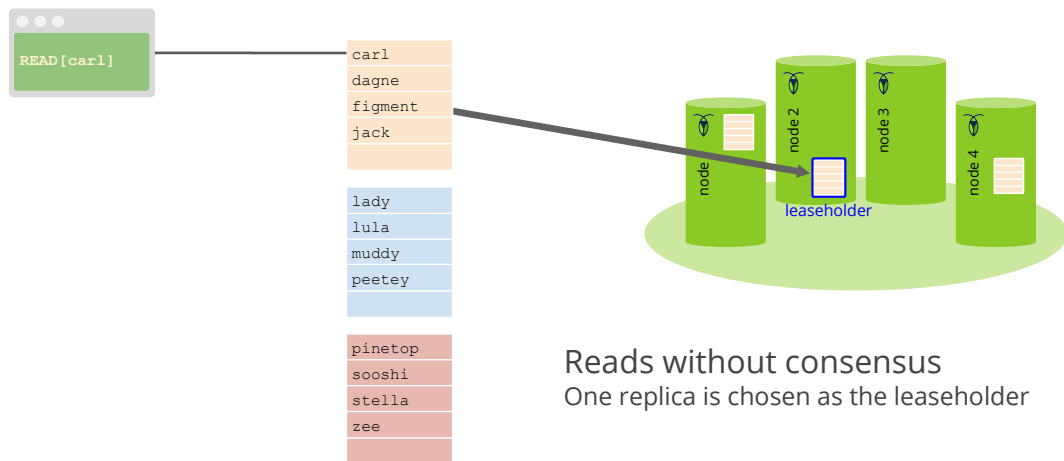


Quorum replication is reasonable for writes, but burdensome for reads.

Consider this example of reading the key “carl”. Using a consensus read would involve talking to 3 replicas. Our reads would be limited to the network latency we see from a quorum of replicas. And we’d be sending a lot of extra data over the network. Ouch.

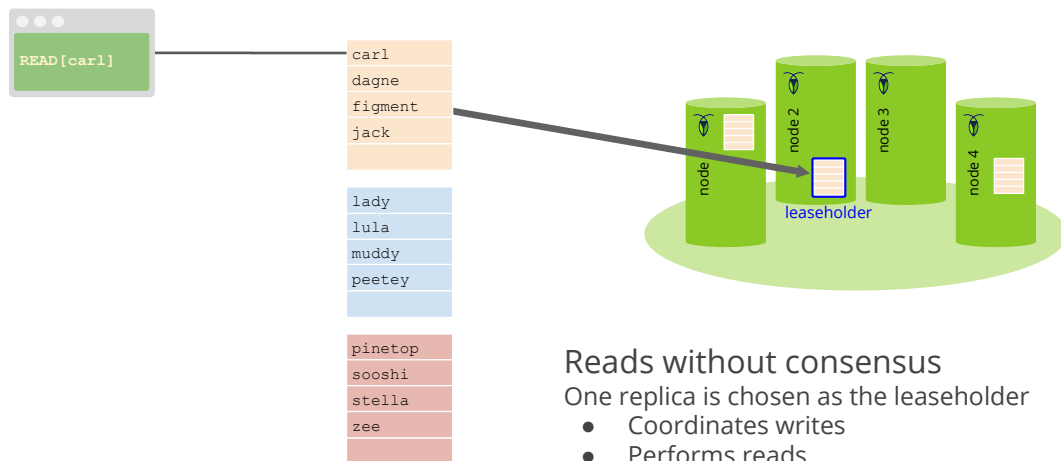
Wouldn’t it be nice if we could read from a single replica and avoid the network round trips to the other replicas?

## Range Leases



Unfortunately we can't read from any replica, an arbitrary replica may be behind which would cause stale reads. Yet the replica which coordinates writes, the leaseholder, is aware of which writes have been committed. The leaseholder can service reads without communicating with other replicas.

## Range Leases



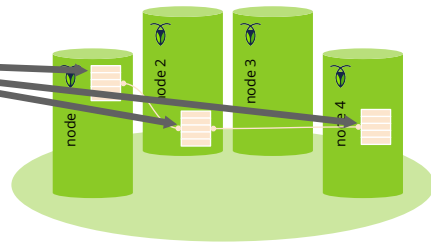
The leaseholder is said to hold the “range lease”. Range leases give CockroachDB reads without consensus. Additionally, the leaseholder implements some of the concurrency control mechanisms used by transactions, such as key-range locking.

Coordinates writes (proposal, key locking)

# Replica Placement

- User-defined constraints
- Latency
- Diversity
- Load
- Space

carl
dagne
figment
jack
lady
lula
muddy
peetey
pinetop
sooshi
stella
zee



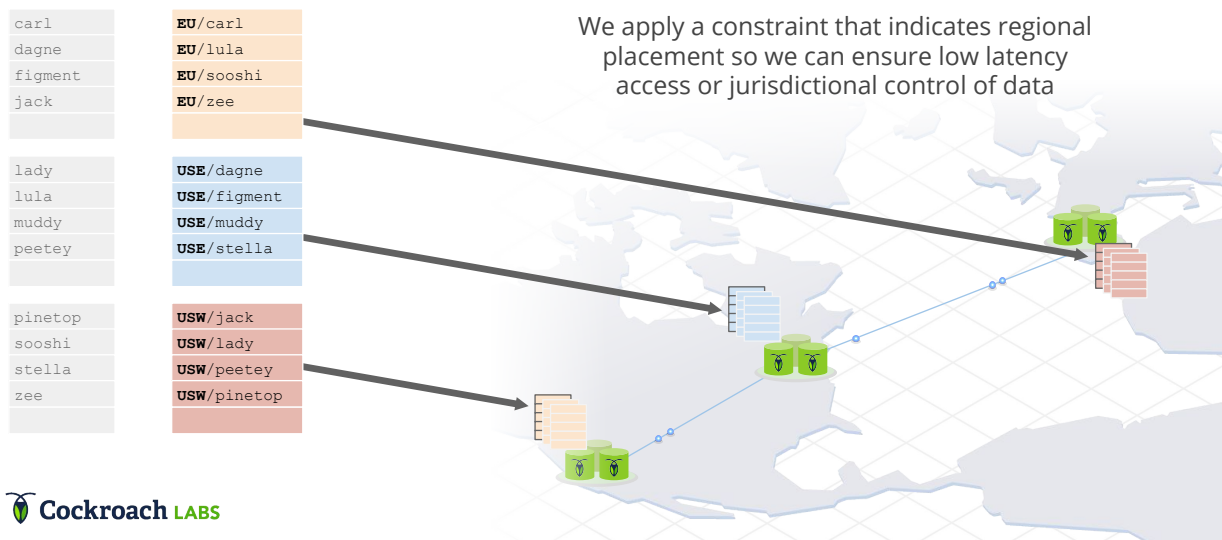
Each Range is a Raft state machine  
A Range has 1 or more Replicas



Now that I've explained how we store and replicate ranges, how do we decide where to place them in the cluster?

There are five signals that CockroachDB uses for replica placement, which you see here.

# Replica Placement: User-defined constraints & Latency



The first signal is the most important, and will override all the others. User-defined constraints basically allow a DBA or application developer to specify on a per-row basis where the data is allowed to reside. For example, you can enforce GDPR requirements by specifying that the data for European customers must not leave the EU data centers. You can also get performance benefits by specifying that certain data should reside close to the users who will be using it most frequently.

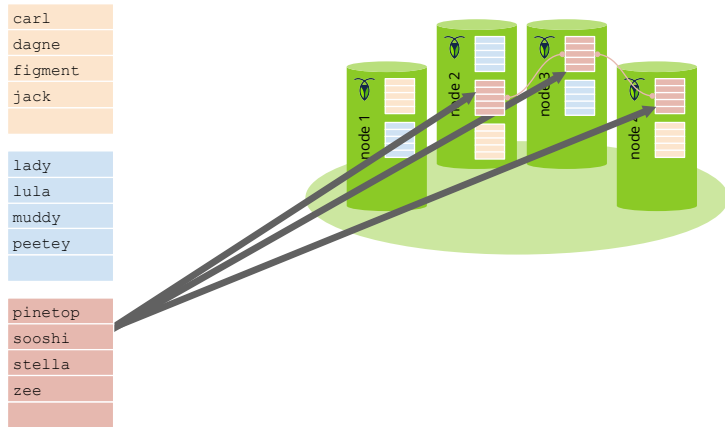
The other mechanism to place replicas based on latency is automatic: CockroachDB attempts to move leaseholders and replicas close to where they are being accessed from. We describe this as “follow the workload”, because this allows CockroachDB to automatically move leaseholders and replicas around in a cluster as the workload shifts over the course of a day or week.

# Replica Placement: Diversity

## Diversity

optimizes placement of replicas across “failure domains”

- Disk
- Single machine
- Rack
- Datacenter
- Region



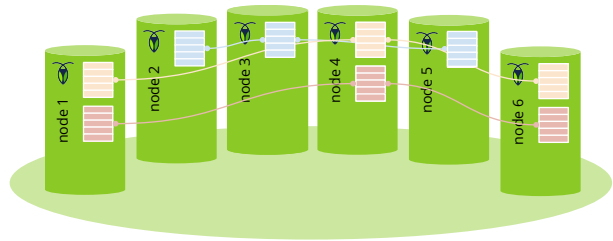
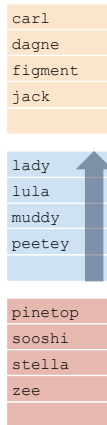
Next let's discuss diversity-based placement. This signal is the most basic when it comes to redundancy. As we all know, diversity improves teams and companies. For a distributed system, diversity also improves availability when replicas are spread across more diverse failure domains. What is a failure domain? It is a unit of failure. For example, a disk, or a single machine, or a rack, or a datacenter, or a region. If CockroachDB ignored failure domains, it might decide to place all of the replicas for a range on a single machine. That's a silly extreme, but it makes the point. Instead, CockroachDB attempts to spread replicas across as many failure domains as possible, while adhering to administrator preferences and constraints about where replicas can be placed.

## Replica Placement: Load & Space

### Load

Balances placement using heuristics that considers real-time usage metrics of the data itself

This range is high load as it is accessed more than others



While we show this for ranges within a single table, this is also applicable across all ranges across ALL tables, which is the more typical situation



The next heuristic CockroachDB uses for replica placement is load. You might imagine that balancing space across the nodes in a cluster is sufficient to balance load, but unfortunately it isn't. For one, there is an imbalance in the work performed between leaseholder replicas and follower replicas. Recall that reads are handled by the leaseholder replica. If all of the leaseholders are on a single node, we'll see severely imbalanced CPU and network load. So the first mechanism in load-based replica placement is to distribute leaseholders throughout the cluster.

Unfortunately, load is not evenly balanced across ranges. That would make our lives too easy. In this slide, the blue range is seeing much higher load than the other ranges. This frequently happens in customer workloads. For example, the blue range might be a small reference table that is being accessed on every query. Or it might contain a particularly hot row. CockroachDB tracks per-range load metrics, and then balances the load across the nodes in the cluster. In the example here, the blue range ends up as the only range on a node. In practice, there are usually thousands of ranges per node, but this load-based balancing is still important.

In the interests of time, I'm going to skip over describing space-based placement as it can be summed up as: "we try to balance space usage across the nodes in the cluster".

# Rebalancing Replicas

## Scale: Add a node

If we add a node to the cluster, CockroachDB automatically redistributed replicas to even load across the cluster

Uses the replica placement heuristics from previous slides



Replica placement in a static cluster is an interesting problem, but clusters are not static. Nodes get added, old nodes get replaced. Nodes go down temporarily or permanently.

Making Data Easy means we want to handle all of these scenarios gracefully. Let's start with adding a node to a cluster.



# Rebalancing Replicas

## Scale: Add a node

If we add a node to the cluster, CockroachDB automatically redistributed replicas to even load across the cluster

Uses the replica placement heuristics from previous slides



Movement is decomposed into adding a replica followed by removing a replica



A newly added node starts out as empty. The other nodes in the cluster see the new node and start moving replicas to it. Replica movement is decomposed into adding a replica followed by removing a replica.

# Rebalancing Replicas

## Scale: Add a node

If we add a node to the cluster, CockroachDB automatically redistributed replicas to even load across the cluster

Uses the replica placement heuristics from previous slides



Movement is decomposed into adding a replica followed by removing a replica



The previously described placement heuristics are used to choose which replicas to move in order to balance space, load, diversity, and latency.

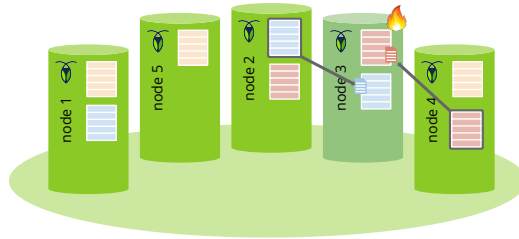
Note that there is a small additional challenge here which is to avoid any sort of thundering herd problem which overwhelms a new node with replicas. There are various throttling mechanisms and randomization used to avoid this problem.

# Rebalancing Replicas

Loss of a node

## Temporary Failure

If a node goes down for a moment, the leaseholder can “catch up” any replica that is behind



The leaseholder can send commands to be replayed OR it can send a snapshot of the current Range data. We apply heuristics to decide which is most efficient for a given failure.



We also need to handle temporary node failure. This is actually more frequent than permanent node failure. For example, an operator may take down a node in order to upgrade its version. Or, and I realize this is going to be hard for you to believe, CockroachDB contains a bug which causes a node to crash. While the node is down, the replicas on the node fall behind. When the node comes back up, the replicas need to be “caught up”. There are two mechanisms to catch up a replica. The first is to send a snapshot of the full range data. If all of the data in a range has changed while the node is down, this is efficient. But what if only a few entries have changed? In that case, it is better to send a list of writes to be replayed. CockroachDB decides on a per-range basis which technique to use based on the number of writes that have occurred to the range.

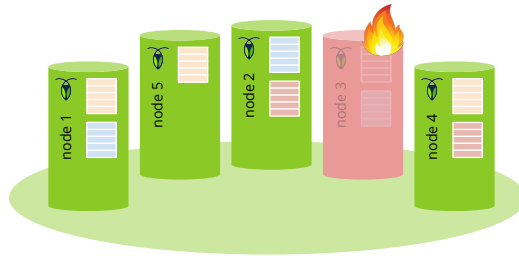
# Rebalancing Replicas

Loss of a node

## Permanent Failure

If a node goes down, the Raft group realizes a replica is missing and replaces it with a new replica on an active node

Uses the replica placement heuristics from previous slides



The permanent failure of a node triggers the inverse behavior of adding a new node.

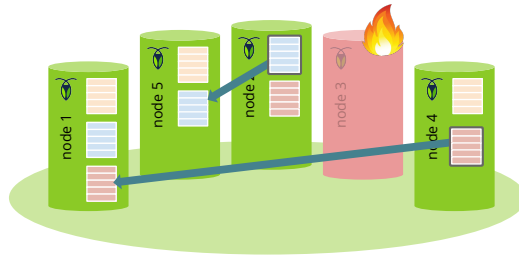
# Rebalancing Replicas

## Loss of a node

### Permanent Failure

If a node goes down, the Raft group realizes a replica is missing and replaces it with a new replica on an active node

Uses the replica placement heuristics from previous slides



The failed replica is removed from the Raft group and a new replica created. The leaseholder sends a snapshot of the Range's state to bring the new replica up to date.

The ranges with replicas on the failed node will notice the node has failed and add new replicas on live nodes.

## AGENDA

- Introduction
- Ranges and Replicas
- Transactions
- SQL Data in a KV World
- SQL Execution
- SQL Optimization
- Evaluation



Moving on up. The distributed, replicated, key-value store is also transactional. Let's dive into the transactional bit.

## Transactions in CockroachDB are serializable, always

- Transactions can span arbitrary Ranges
- Conversational
  - Full set of operations not required up front
- Transaction atomicity supported with Raft atomic writes
  - Transaction record atomically flipped from PENDING to COMMIT

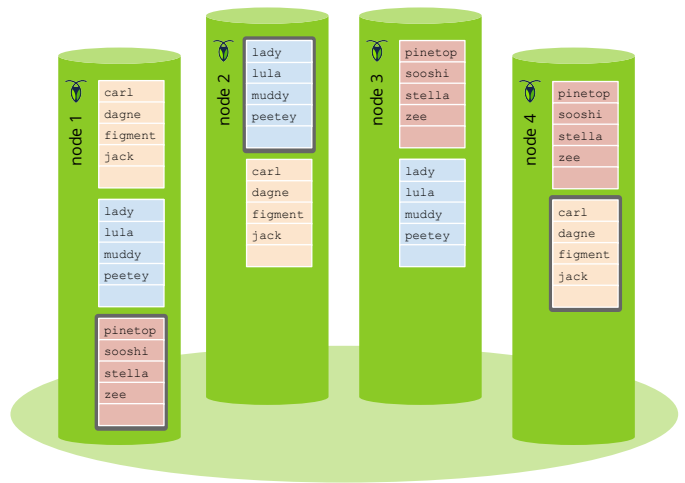
The first thing to note about transactions in CockroachDB is that they are always serializable. We don't support any lower isolation levels.

Transactions can also span arbitrary ranges, and they support a conversational protocol. The conversational protocol is important for supporting SQL, where the full set of operations may not be known up front.

As we all know, one of the important properties of transactions is that they are atomic. In order to guarantee atomicity even in the case of transactions that span many ranges, CockroachDB takes advantage of the range level atomicity provided by Raft. Each transaction has an associated transaction record, which is stored in a Range just like other data. Updates to the transaction record go through Raft, which is how we provide atomicity for transactions.

# Distributed Transactions

```
INSERT INTO dogs  
VALUES (sunny, ozzie)
```

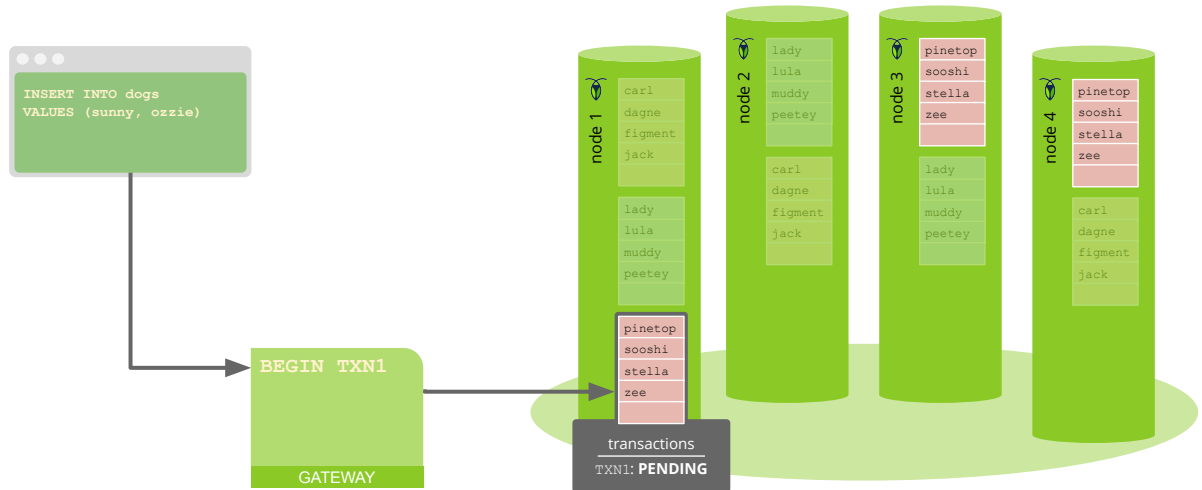


Let's walk through a simple example so you can see how transactions work in action. Just so you're aware, I'm going to start by showing you an older, suboptimal version of the real algorithm since it's a bit easier to understand.

What you see here is a cluster with 3 ranges spread across 4 nodes. The leaseholders for each range are highlighted with a black outline.



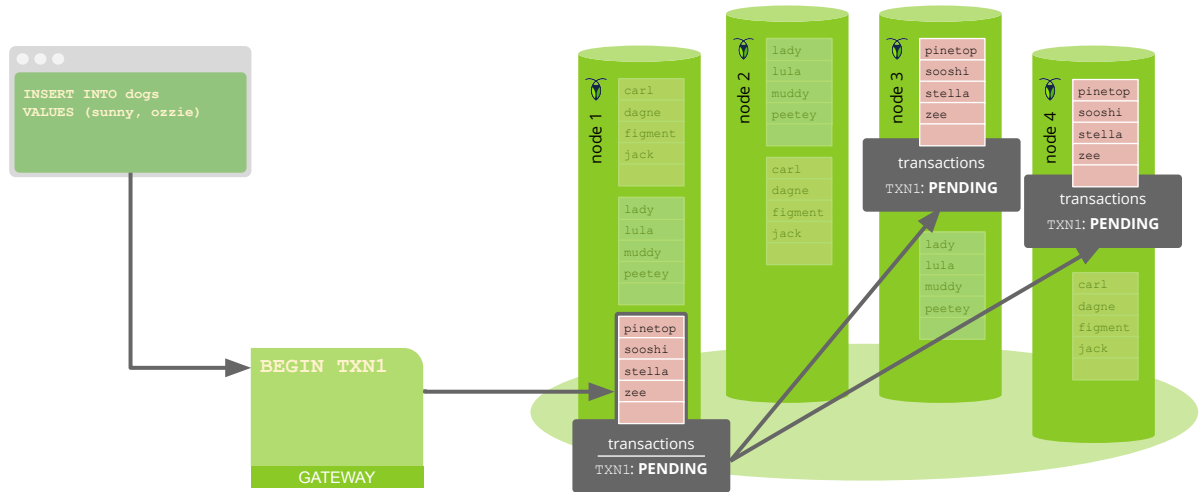
# Distributed Transactions



This INSERT statement is inserting two rows into our “dogs” table, “sunny” and “ozzie”. To begin, the client connects to a gateway node, which connects to the leaseholder for the range containing “sunny”.

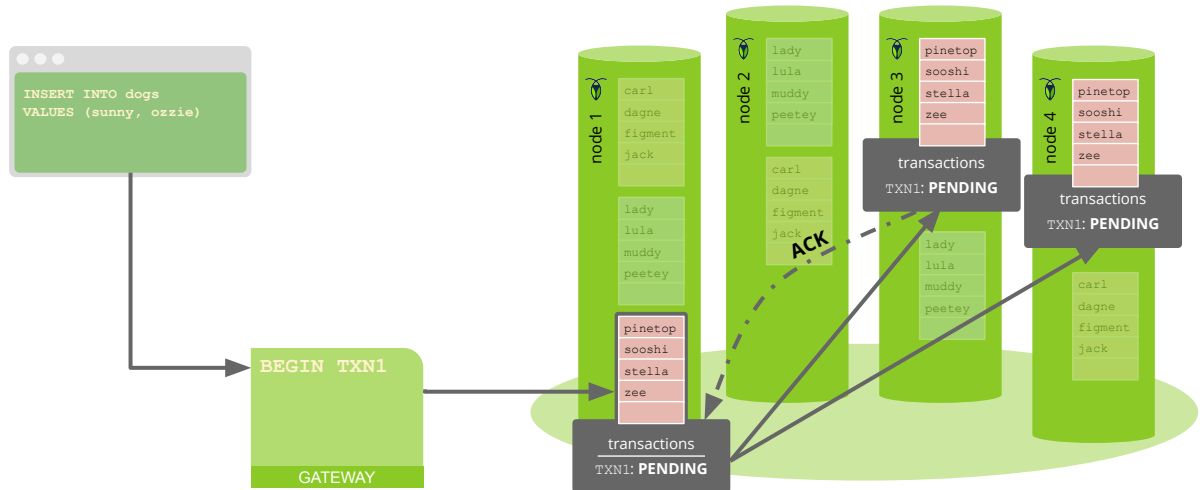
Because “sunny” is the first key written by the transaction, a transaction record is created on the range containing “sunny”.

# Distributed Transactions



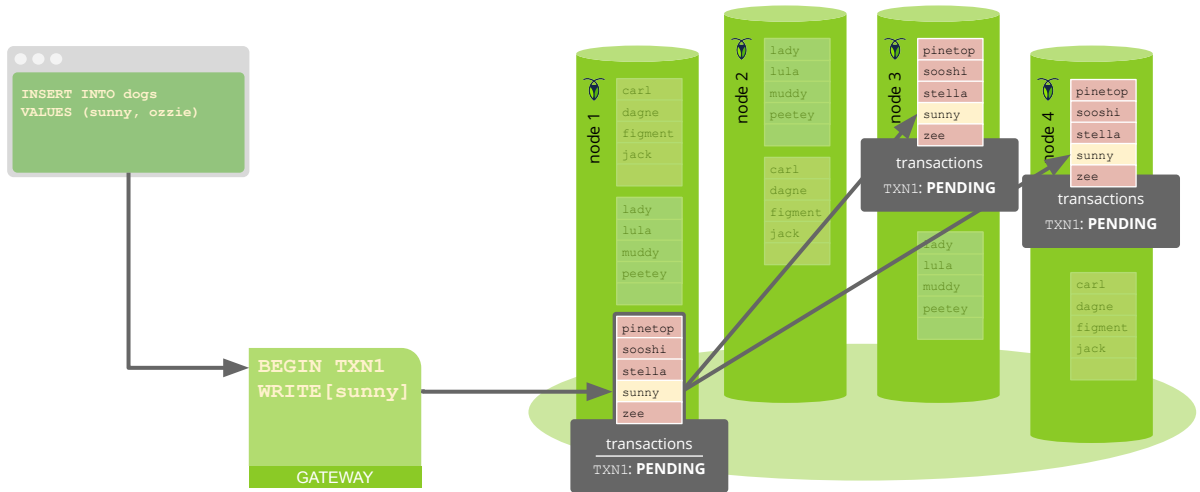
To replicate the the transaction record, the leaseholder proposes a Raft command which writes the record to itself and the follower replicas.

# Distributed Transactions



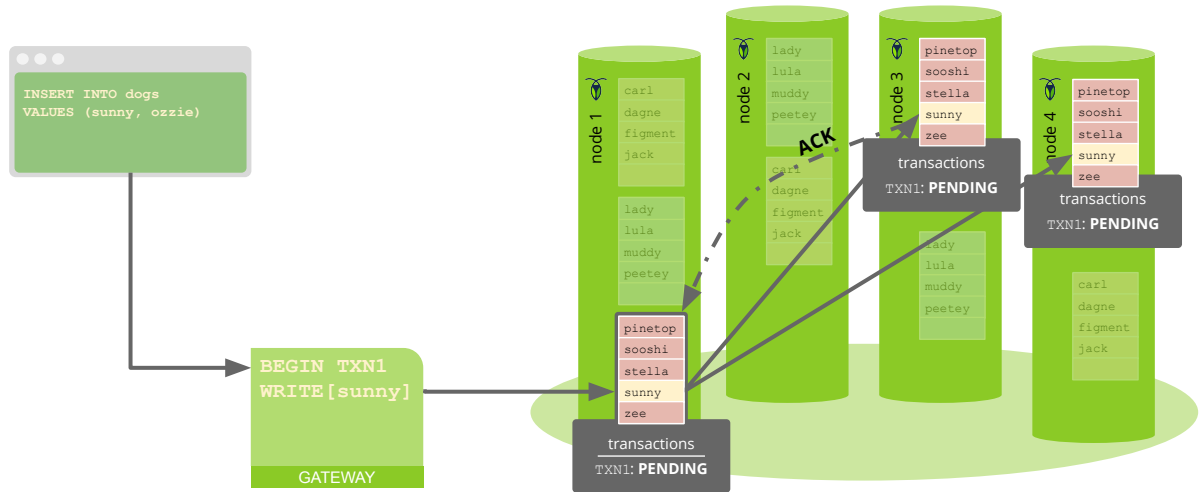
Once the leaseholder and at least one of the followers accept the creation of the transaction record, the transaction is in progress.

# Distributed Transactions



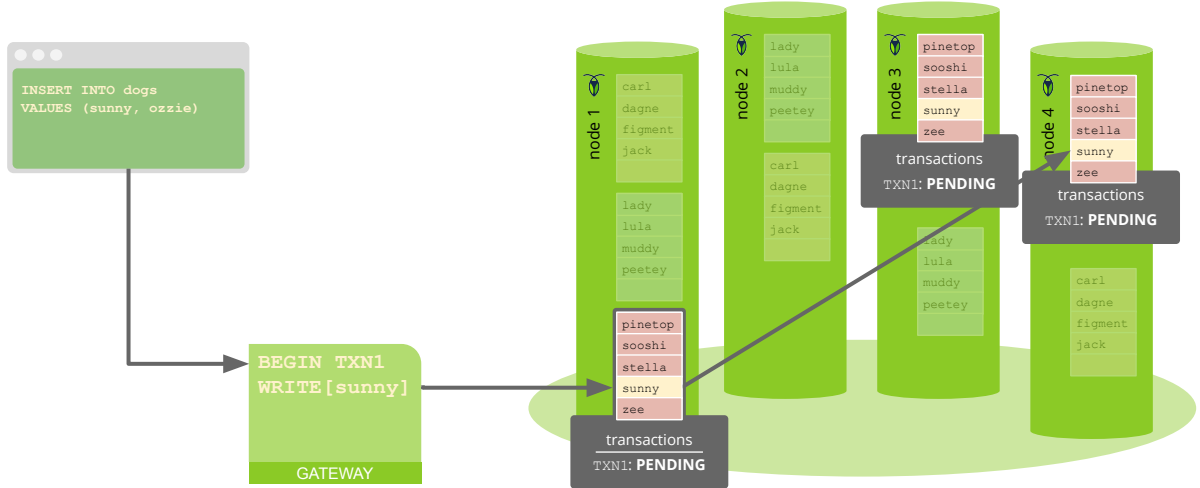
Next the same leaseholder proposes a Raft command which writes “sunny” to itself and the follower replicas.

# Distributed Transactions

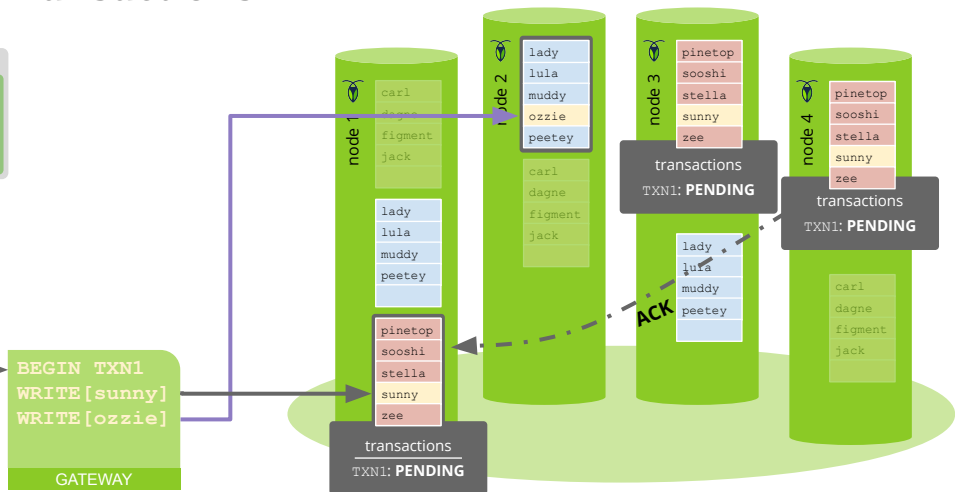


It again waits for a quorum of replicas before moving on.

# Distributed Transactions

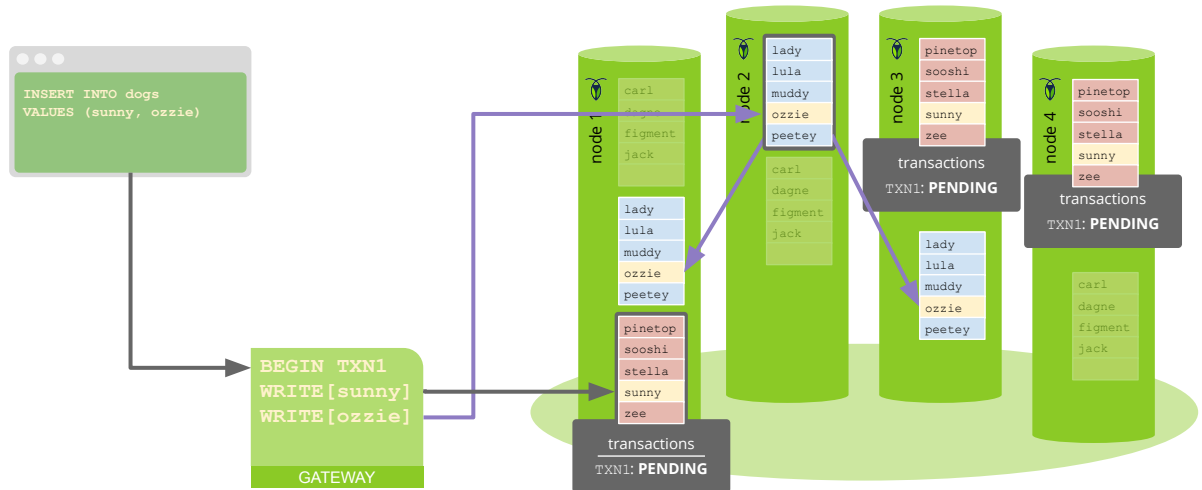


Notice that I've left "sunny" highlighted in yellow. That indicates that "sunny" has been updated by an active transaction that may or may not be committed. We tag each updated key with the transaction ID, so other transactions can check the corresponding transaction record to determine the status.



We then move on to the next operation, writing the key “ozzie”. Similar to the previous write, the gateway node sends this operation to the leaseholder for the range containing “ozzie”. Notice that this is happening in parallel with the final replica of sunny acknowledging the write, since we already had a quorum before and didn't need to wait.

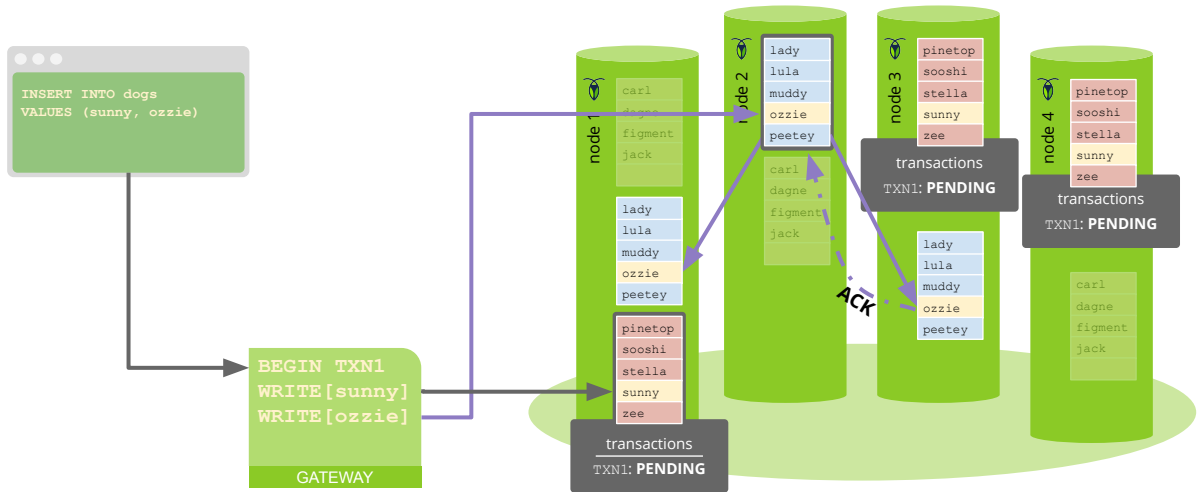
# Distributed Transactions



Just like with the first write, the leaseholder of “ozzie” proposes a command which is sent to the follower replicas.

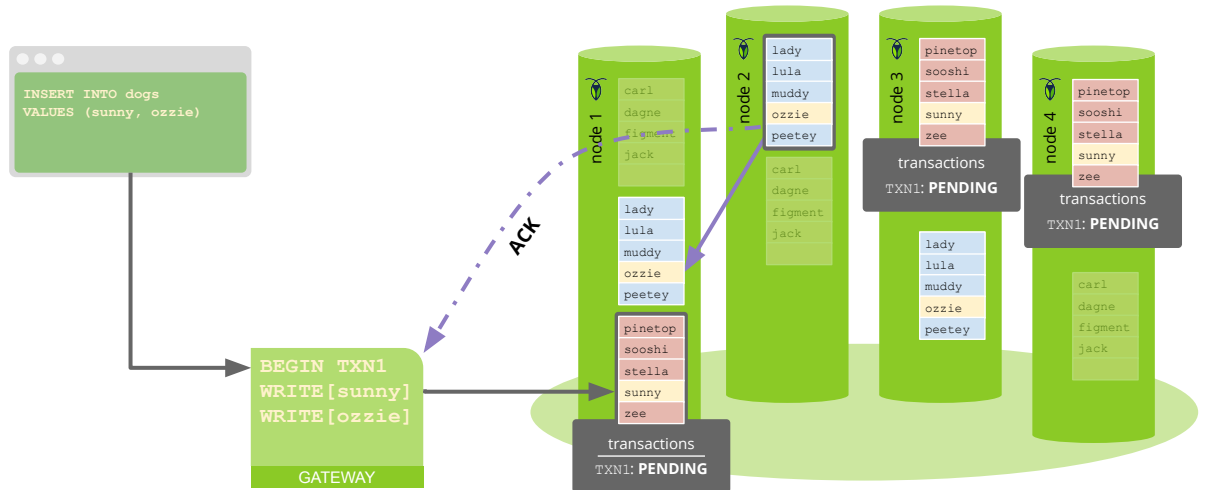


# Distributed Transactions



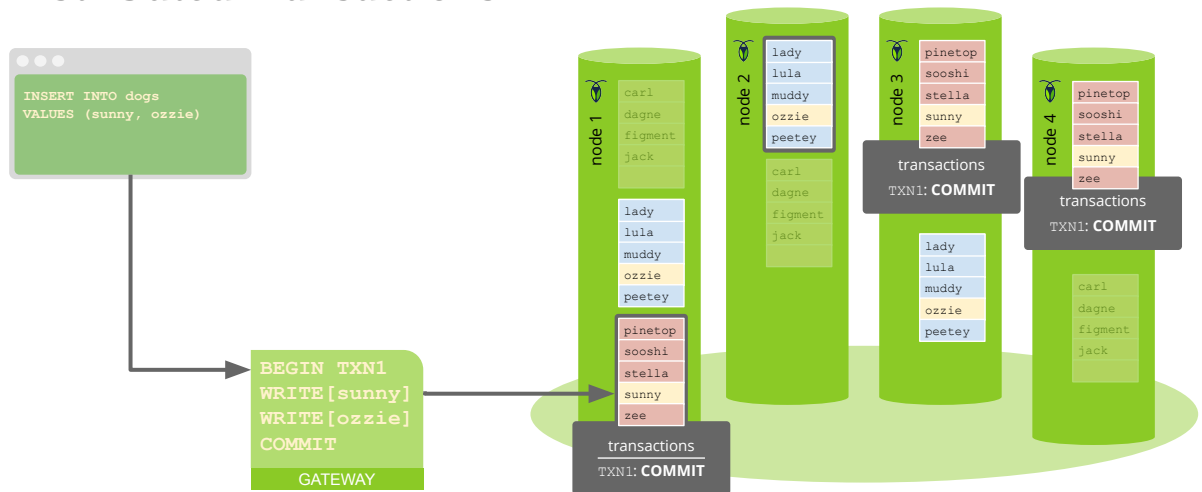
A follower acknowledges the command.

# Distributed Transactions



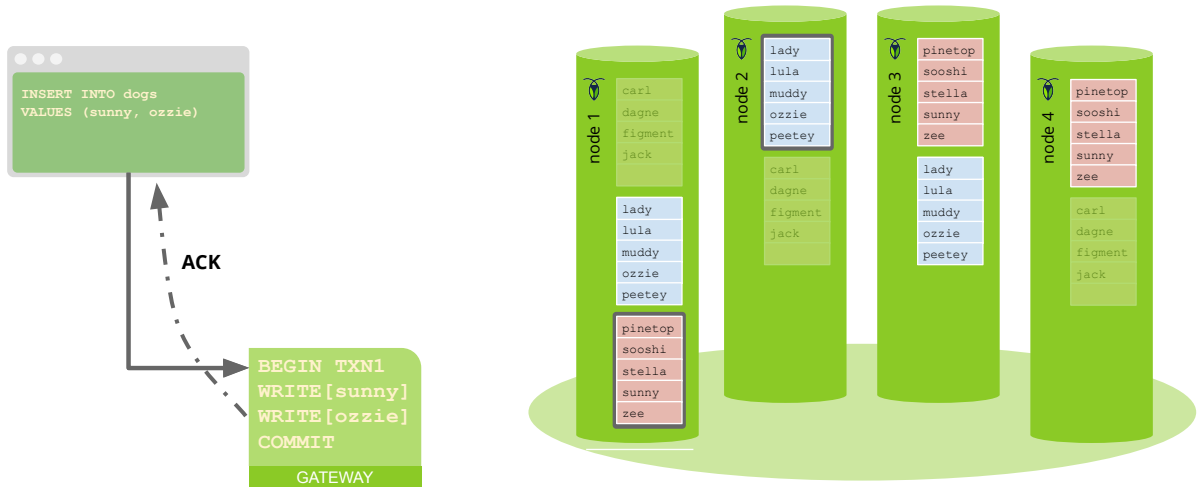
And now that we have a quorum, the write operation is done.

# Distributed Transactions



The gateway node then commits the transaction, which involves marking the transaction record as COMMITTED. I'm not showing it, but this is also a replicated operation requiring a round of consensus.

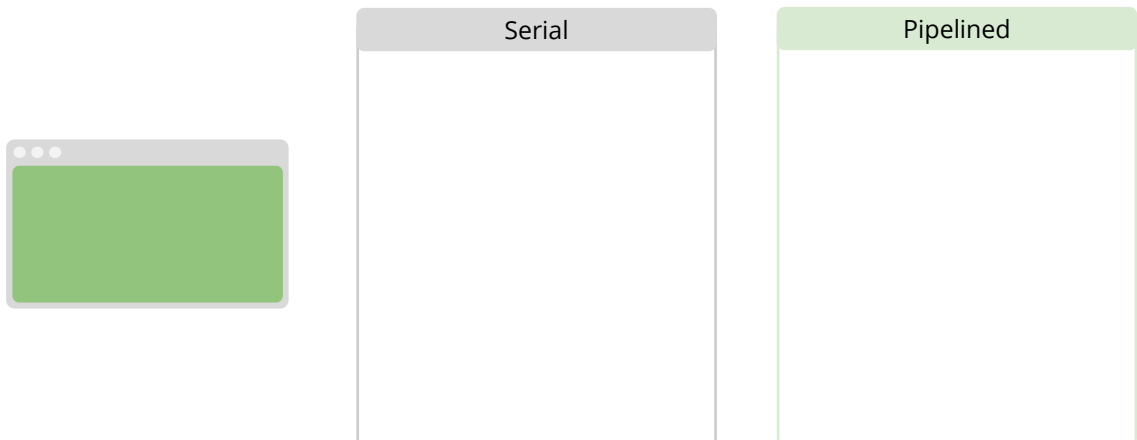
# Distributed Transactions



Once the commit operation is done, we can tell the client that the transaction is complete. There is also a background operation which releases the transaction markers attached to the records. We remove those markers to make reads of the keys faster.

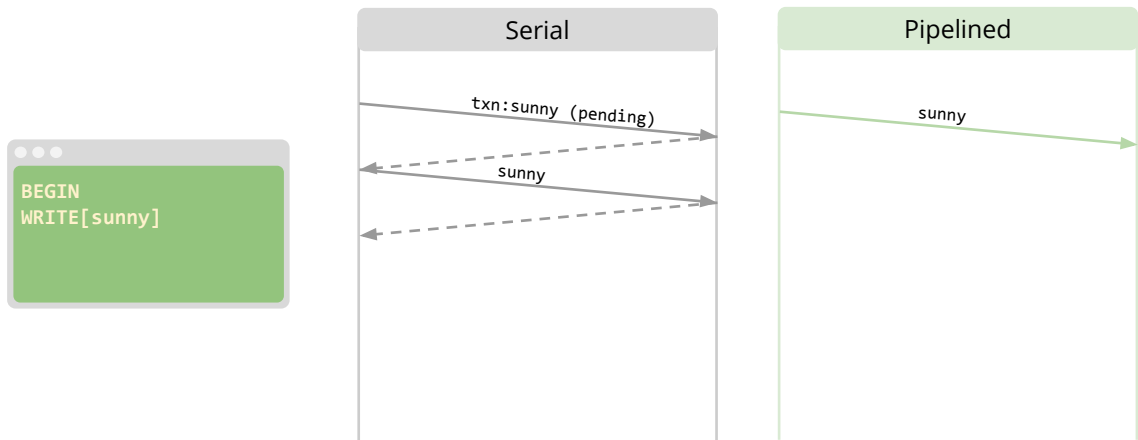
I've omitted a huge number of details in this explanation. I haven't described how read-write and write-write conflicts are handled. I haven't described distributed deadlock detection. I haven't talked about how long running transactions are handled.

## Transactions: Pipelining



What I described was also the original transaction protocol CockroachDB implemented. It is correct, but it requires multiple round trips of Raft writes. We've been evolving this protocol over time, and the current protocol can commit a distributed transaction in a single round trip of latency, with additional asynchronous round trips to perform cleanup. We call this "pipelining", and the original protocol is referred to here as "serial". Let me walk through the same example, showing how pipelining reduces the impact of round trips.

## Transactions: Pipelining



Here we have the first two operations, begin transaction and write “sunny”. With serial transaction operations, we were waiting for each write to complete before performing the next write. So we’d write the txn record, wait for it to complete. Then we’d write “sunny” and wait for it to complete.

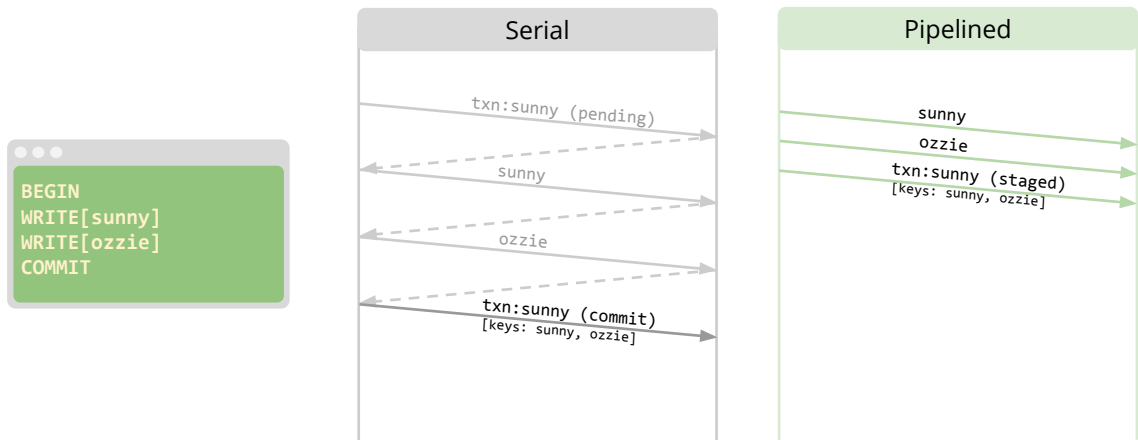
With pipelining, we fire off the write for “sunny”, but don’t wait for it to complete. The txn record is not written until later. This is safe because a txn is given a small grace window in which a non-existent txn record is considered as in progress instead of aborted.

## Transactions: Pipelining



With pipelined operations, the second write operation is initiated before the first completes.

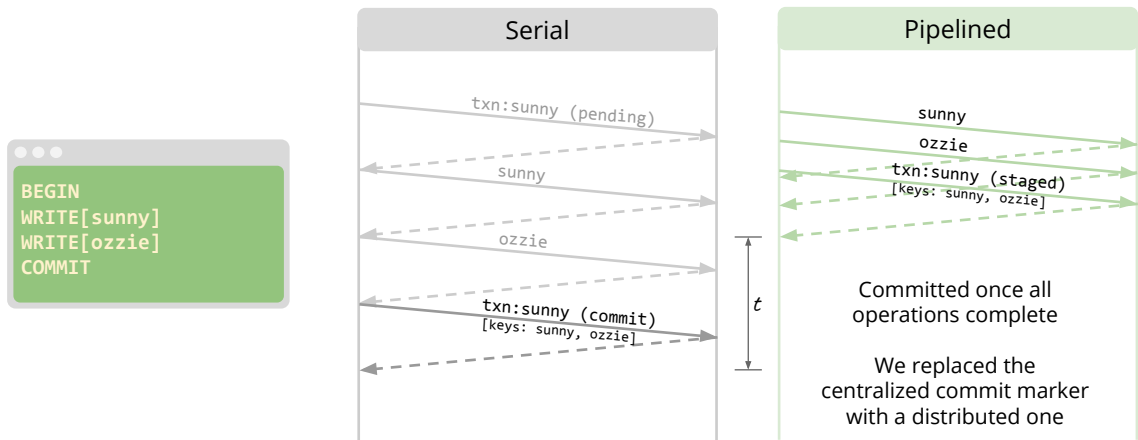
## Transactions: Pipelining



The transaction commit is also pipelined, but the txn record is written as STAGED instead of COMMITTED. This basically means that the transaction is finished, but it's not clear yet whether the commit was successful.



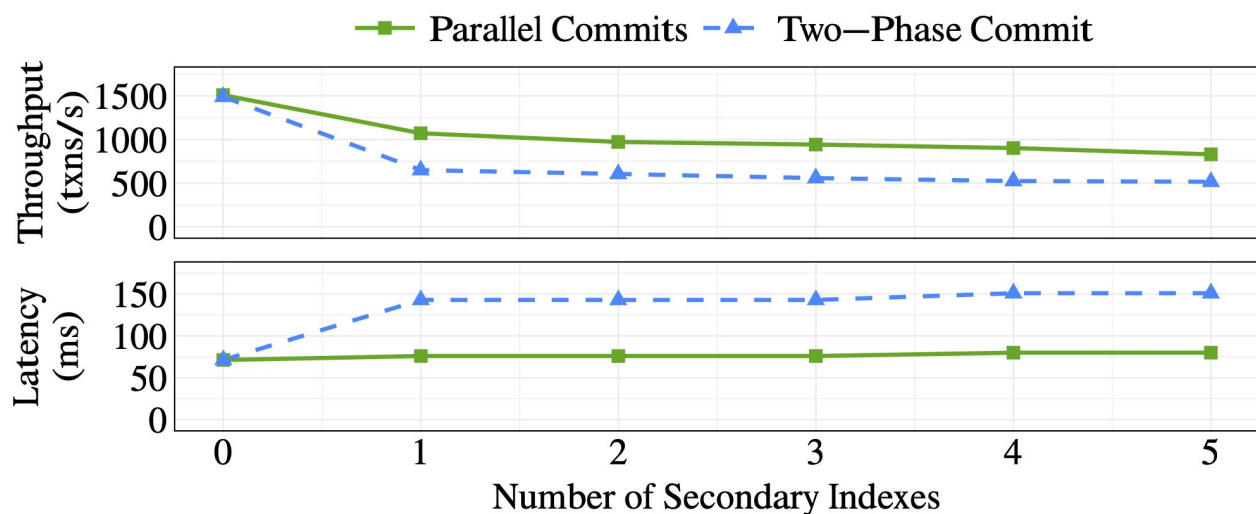
## Transactions: Pipelining



Once all the write operations have completed successfully, the gateway node knows that the commit was successful, and can therefore return to the client. Cleanup operations to actually mark the transaction as committed happen asynchronously.

With the pipelining protocol, you'll notice that we've reduced the latency of this transaction from four round trips down to a single round trip.

## Parallel Commits v. Two-Phase Commit (Pipelined v. Serial)



This performance improvement shows up in practice too. This chart shows the results of running a workload of single-row writes to a table with ten columns and a variable number of secondary indexes on those columns. As soon as we add one secondary index we now require a multi-range transaction. With parallel commits (the pipelined protocol I showed you), throughput only drops slightly while latency remains constant, even as we increase the number of secondary indexes. Meanwhile, the traditional two-phase commit protocol (similar to the serial protocol) shows a more significant drop in throughput and nearly 2x increase in latency.

## AGENDA

- Introduction
- Ranges and Replicas
- Transactions
- SQL Data in a KV World
- SQL Execution
- SQL Optimization
- Evaluation



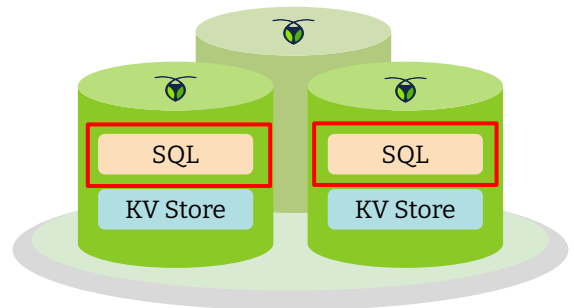
CockroachDB is a geo-distributed SQL database. So far I've been talking about a distributed, replicated, transaction key-value store. Now it is time to talk about SQL.

CockroachDB implements standard SQL, with a few extensions. We support the whole kit-and-kaboodle of SQL, from indexes, to transactions, to foreign keys. Our SQL dialect is compatible with Postgres, modulo some esoteric corner cases, and we support the Postgres wire protocol so drivers already exist for almost every language.

## SQL: Tabular Data in a KV World

How do we store typed and columnar data in a distributed, replicated, transactional key-value store?

- The SQL data model needs to be mapped to KV data
- Reminder: keys and values are lexicographically sorted



Wait a minute: SQL data has columns and types?!? All of this seems quite distant from keys and values. Yet it is not distant at all. SQL data can be mapped to KV data. Let's see how.

## SQL Data Mapping: Inventory Table

```
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	Value
/1	"Bat",1.11
/2	"Ball",2.22
/3	"Glove",3.33



Let's start with basic table data. In CockroachDB, every table has a primary key which is really just a special index. If a primary key is not specified, one will automatically be provided behind the scenes. Each row in a table maps to single key/value record for the primary index. Each secondary index for the table creates an additional key/value record.

The indexed columns are encoded into the key. The non-indexed columns are encoded into the value. You can probably imagine a scheme for storing the non-indexed columns in the value. We could have used something like protobufs, or avro. We actually use a custom encoding because this is highly performance critical, the details are not super important.

More interesting is how we encode the indexed columns into the key. Naively this seems difficult or impossible. We need an encoding such that the encoded key is ordered the same way as the columns. With our inventory table, we need to encode integers into a string such that the encoded strings are ordered the same as the integers. Some of you might be able to imagine how to do this for a single integer column. Now I get to wave my hands and say that it is possible to do this for arbitrary column tuples. The details are interesting, though too low-level for this architecture talk. We didn't invent this idea. I first saw it at Google, but Google didn't invent it either. I believe I saw a paper from the late 1980s which mentions it.

So when you see “/1”, “/2”, and “/3”, be aware that is a logical representation of the key encoding, not the physical encoding itself.

## SQL Data Mapping: Inventory Table

```
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	Value
/Table/1	"Bat", 1.11
/Table/2	"Ball", 2.22
/Table/3	"Glove", 3.33



CockroachDB supports multiple tables, and indexes by prefixing keys with table name and index name.

## SQL Data Mapping: Inventory Table

```
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	Value
/inventory/primary/1	"Bat",1.11
/inventory/primary/2	"Ball",2.22
/inventory/primary/3	"Glove",3.33



This is what that looks like for the primary index of our inventory table. You're probably thinking, wow, this is repetitive and expensive. In reality, we store table IDs and index IDs. These are both smaller and allow fast rename operations for tables and indexes. Key prefix compression at the lowest layer takes care of the remaining overhead.



## SQL Data Mapping: Inventory Table

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

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

Key	Value
/inventory/name_idx/"Ball"/2	∅
/inventory/name_idx/"Bat"/1	∅
/inventory/name_idx/"Glove"/3	∅



Here is an example showing the keys created for a secondary index. I've add an index on the column "name". Something to point out with the keys is that they contain more than just the name column. Why is that? Note that this index is non-unique. That means we can add multiple rows with the same "name" column.

## SQL Data Mapping: Inventory Table

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

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

Key	Value
/inventory/name_idx/"Ball"/2	∅
/inventory/name_idx/"Bat"/1	∅
/inventory/name_idx/"Glove"/3	∅



Let's say we add another row with the name "Bat". The KV layer doesn't support having different values for two identical keys. We need to provide unique keys. The SQL data mapping layer handles this by appending a uniqueifier to the key: the columns of the primary key.

## SQL Data Mapping: Inventory Table

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

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

Key	Value
/inventory/name_idx/"Ball"/2	∅
/inventory/name_idx/"Bat"/1	∅
/inventory/name_idx/"Bat"/4	∅
/inventory/name_idx/"Glove"/3	∅



So our second “Bat” row encodes to a different key than the first “Bat” row.

## AGENDA

- Introduction
- Ranges and Replicas
- Transactions
- SQL Data in a KV World
- SQL Execution
- SQL Optimization
- Evaluation



SQL is about more than tabular data storage. SQL is a powerful query language. And since CockroachDB is a SQL database, we need to implement that query language. I'm going to give you a quick overview of how basic query execution works, then describe distributed query execution, and finally talk about query optimization.

# SQL Execution

## Relational operators

- Projection (SELECT <columns>)
- Selection (WHERE <filter>)
- Aggregation (GROUP BY <columns>)
- Join (JOIN), union (UNION), intersect (INTERSECT)
- Scan (FROM <table>)
- Sort (ORDER BY)
  - Technically, not a relational operator



SQL has a relatively small number of basic operations: projection, selection (filtering), aggregation, joining, etc. These operations are expressed in SQL via constructs such as SELECT, WHERE, GROUP BY, and JOIN. I'm going to give a small taste of what SQL execution looks like.

## SQL Execution

- Relational expressions have 0-2 input expressions
- Query plan is a tree of relational expressions
- SQL execution takes a query plan and runs the operations to completion



- and scalar expressions
  - For example, a “filter” expression has 1 input expression and a scalar expression that filters the rows from the child
  - The scan expression has zero inputs

## SQL Execution: Example

```
SELECT name  
FROM   inventory  
WHERE  name >= "b" AND name < "c"
```



Let me give a brief example of how SQL execution works. Here is a very simple query which is reading part of an “inventory” table.

## SQL Execution: Scan

```
SELECT name  
FROM inventory  
WHERE name >= "b" AND name < "c"
```

Scan  
inventory



SQL execution starts with the table being read from. In this query, we're reading from the "inventory" table.



## SQL Execution: Filter

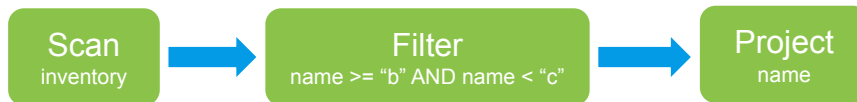
```
SELECT name  
FROM   inventory  
WHERE  name >= "b" AND name < "c"
```



The rows from the “inventory” table are fed through a filter operator.

## SQL Execution: Project

```
SELECT name  
FROM   inventory  
WHERE  name >= "b" AND name < "c"
```



The resulting filtered rows are then fed through a project operator which outputs only the column desired.

## SQL Execution: Project

```
SELECT name
FROM   inventory
WHERE  name >= "b" AND name < "c"
```



The output from the project operator are the results sent to the user. This is all straightforward and quite inefficient. Some of you probably noticed that the project operator could be run before the filter operator. A full table scan is also very inefficient. Such transformations are performed by a SQL optimizer, which I'll be talking about shortly. What would this look like if the table has an index on "name"?

## SQL Execution: Index Scans

```
SELECT name  
FROM inventory  
WHERE name >= "b" AND name < "c"
```

Scan  
inventory@name ["b" - "c")

The filter gets pushed into the scan



If there is an index on (name) we can scan a contiguous set of keys in the index.

## SQL Execution: Index Scans

```
SELECT name  
FROM   inventory  
WHERE  name >= "b" AND name < "c"
```



We still have to project the “name” column, but that is a fast operation.

## Distributed SQL Execution

Network latencies and throughput are important considerations in geo-distributed setups

Push fragments of computation as close to the data as possible



The previous section focused on basic SQL execution. With a geo-distributed database, network latencies and throughput become important considerations. The key-value store in CockroachDB allows us to pretend that all of the data is available locally, but doing so is inefficient. We need to pierce the abstraction and reveal to SQL execution how data is distributed. With distributed SQL execution we attempt to push fragments of computation as close to the data as possible.

## Distributed SQL Execution: Streaming Group By

```
SELECT  COUNT(*), country  
FROM    customers  
GROUP BY country
```

Scan customers   Scan customers   Scan customers



Let's revisit our customers query. If the table data is distributed across 3 datacenters, the query will instantiate 3 processors to scan the customers table. These processors are instantiated next to the customers table data in each datacenter.

## Distributed SQL Execution: Streaming Group By

```
SELECT  COUNT(*), country  
FROM    customers  
GROUP BY country
```

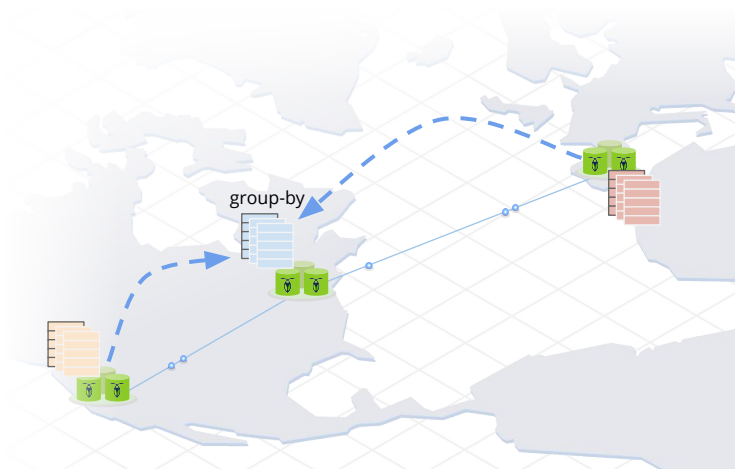
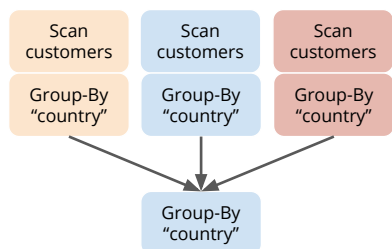


The output of those scans will be fed into group-by operators instantiated in each datacenter that are run next to the data.



## Distributed SQL Execution: Streaming Group By

```
SELECT  COUNT(*), country
FROM    customers
GROUP BY country
```



Finally, the per-datacenter group-by operator output will be fed to a centralized group-by instance which will perform the final aggregation.

In addition to reducing network usage, distributed SQL execution also provides parallelism during query execution which can make it beneficial even if you're running CockroachDB within a single datacenter.

## AGENDA

- Introduction
- Ranges and Replicas
- Transactions
- SQL Data in a KV World
- SQL Execution
- SQL Optimization
- Evaluation



Moving on to our last topic: SQL optimization.

## SQL Optimization: Cost-based Index Selection

The index to use for a query is affected by multiple factors

- Filters and join conditions
- Required ordering (ORDER BY)
- Implicit ordering (GROUP BY)
- Covering vs non-covering (i.e. is an index-join required)
- Locality



Let me give an example of a cost-based transformation: index selection. Index selection is affected by the filters in a query, a required ordering specified by an ORDER BY, implicit ordering specified by a GROUP BY, and whether the index covers the required columns.

## SQL Optimization: Cost-based Index Selection

```
SELECT  *  
FROM    a  
WHERE   x > 10  
ORDER BY y
```

Required orderings affect index selection

Sorting is expensive if there are a lot of rows

Sorting can be the better option if there are few rows



This is an example of how a required ordering affects index selection. Here we have a query with a required ordering on column “y”. How do we execute this query?

## SQL Optimization: Cost-based Index Selection

```
SELECT  *  
FROM    a  
WHERE   x > 10  
ORDER BY y
```

Required orderings affect index selection

Sorting is expensive if there are a lot of rows

Sorting can be the better option if there are few rows



The initial AST to relational algebra transformation produces a naive query plan. We scan the primary index, filter on column “x”, and sort on column “y”.

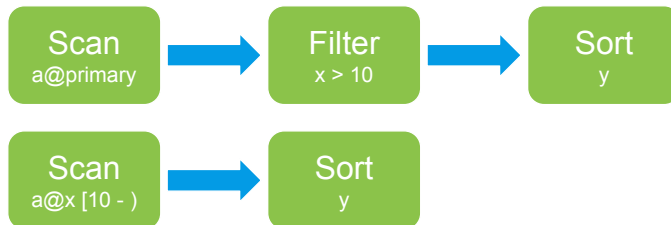
## SQL Optimization: Cost-based Index Selection

```
SELECT  *  
FROM    a  
WHERE   x > 10  
ORDER BY y
```

Required orderings affect index selection

Sorting is expensive if there are a lot of rows

Sorting can be the better option if there are few rows



If we have an index on column “x”, an alternative plan is to push the filter into the scan. This will read fewer rows from the table, and thus have fewer rows to sort.

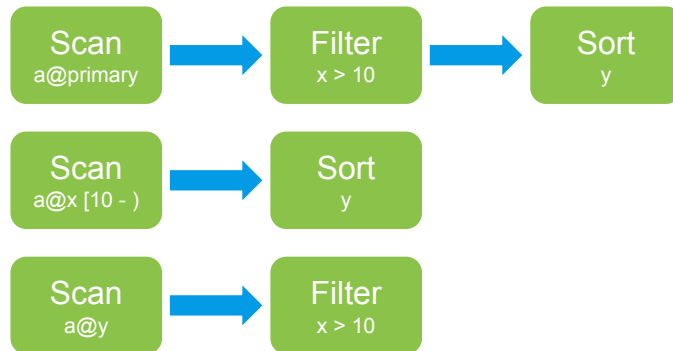
## SQL Optimization: Cost-based Index Selection

```
SELECT  *  
FROM    a  
WHERE   x > 10  
ORDER BY y
```

Required orderings affect index selection

Sorting is expensive if there are a lot of rows

Sorting can be the better option if there are few rows



If we have an index on column “y”, another plan is to scan the index on column “y” and filter the result rows. Not having to sort is a big win, but it isn’t always better than scanning the index on “x”. So which query plan should we use? Which is better depends on how many rows are in the table, and how many rows are left after the filtering step.

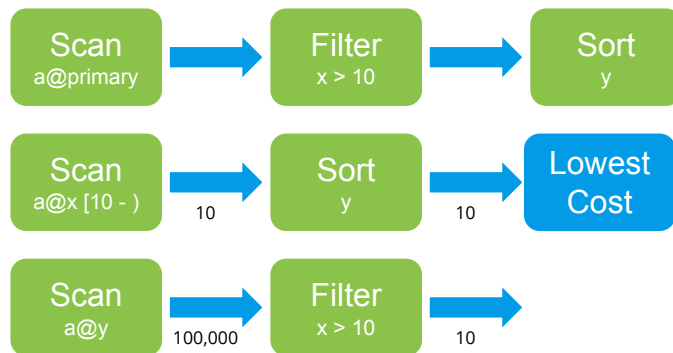
## SQL Optimization: Cost-based Index Selection

```
SELECT  *  
FROM    a  
WHERE   x > 10  
ORDER BY y
```

Required orderings affect index selection

Sorting is expensive if there are a lot of rows

Sorting can be the better option if there are few rows



For example, if only 10 rows pass through the filter, while there are 100,000 rows in the table, it is better to filter, then sort.



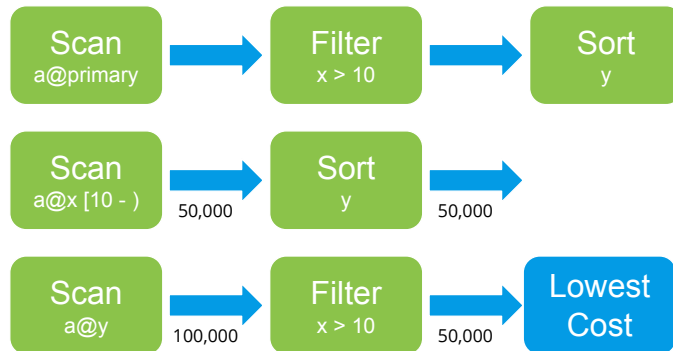
## SQL Optimization: Cost-based Index Selection

```
SELECT  *  
FROM    a  
WHERE   x > 10  
ORDER BY y
```

Required orderings affect index selection

Sorting is expensive if there are a lot of rows

Sorting can be the better option if there are few rows



But if the filter passes through 50,000 rows, then it is better to sort, then filter.

Table statistics allow the optimizer to estimate the cardinality of inputs, and thus estimate a cost for each query.

# Locality-Aware SQL Optimization

Network latencies and throughput are important considerations in geo-distributed setups

Duplicate read-mostly data in each locality

Plan queries to use data from the same locality



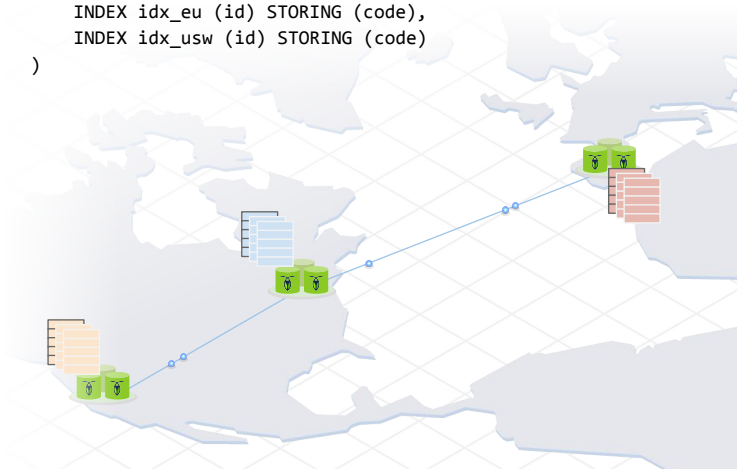
A distributed SQL database has additional opportunities and challenges for optimization over a monolithic SQL database. An example of this is locality-aware optimization.

# Locality-Aware SQL Optimization

Three copies of the  
postal\_codes table data

Use replication constraints to  
pin the copies to different  
geographic regions (US-East,  
US-West, EU)

```
CREATE TABLE postal_codes (  
  id INT PRIMARY KEY,  
  code STRING,  
  INDEX idx_eu (id) STORING (code),  
  INDEX idx_usw (id) STORING (code)  
)
```



Here I've defined a "postal\_codes" table. The table has a primary index on "id", and two identical secondary indexes also on "id". Note that "STORING (code)" syntax causes the index to also contain the "code" column which makes these indexes contain exactly the same data as the primary index.

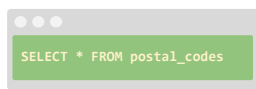
So we have three copies of the "postal\_codes" table data. Using replication constraints we can pin these copies to different geographic regions: us-east, us-west, and europe.

# Locality-Aware SQL Optimization

Optimizer includes locality in cost model

Automatically selects index from same locality: primary, `idx_eu`, or `idx_usw`

```
CREATE TABLE postal_codes (  
  id INT PRIMARY KEY,  
  code STRING,  
  INDEX idx_eu (id) STORING (code),  
  INDEX idx_usw (id) STORING (code)  
)
```



```
SELECT * FROM postal_codes
```



The optimizer considers the locality of where a query is being executed in its cost model. So a scan of the “postal\_codes” table from us-west will use the us-west index data allowing us to avoid a geographic network hop.

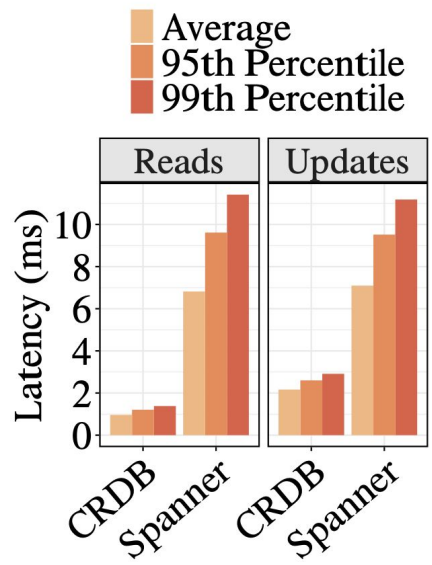
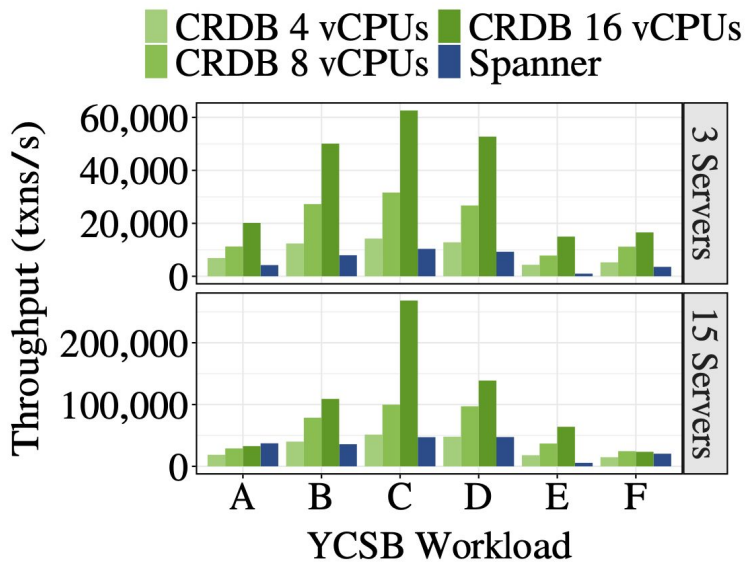
This was a simplistic example, but the mechanism is general. The `postal_codes` table is an example of a reference table and would likely be joined with another table or used a foreign key reference.

# AGENDA

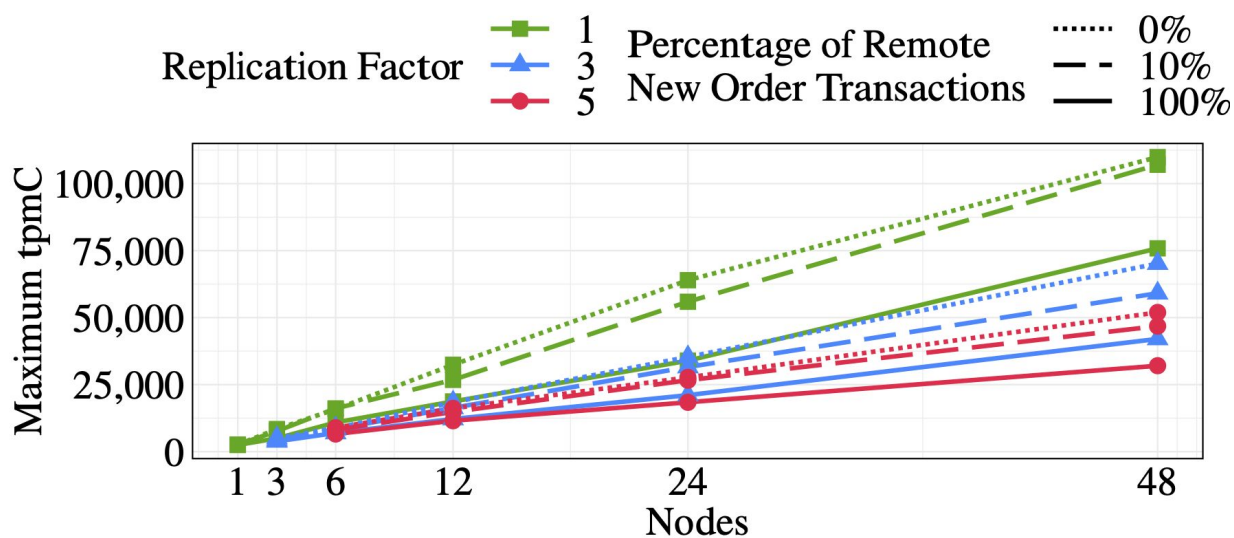
- Introduction
- Ranges and Replicas
- Transactions
- SQL Data in a KV World
- SQL Execution
- SQL Optimization
- Evaluation



## Comparison with Spanner on YCSB



## TPC-C With Varying Cross-Node Coordination



## AGENDA

- Introduction
- Ranges and Replicas
- Transactions
- SQL Data in a KV World
- SQL Execution
- SQL Optimization
- Evaluation

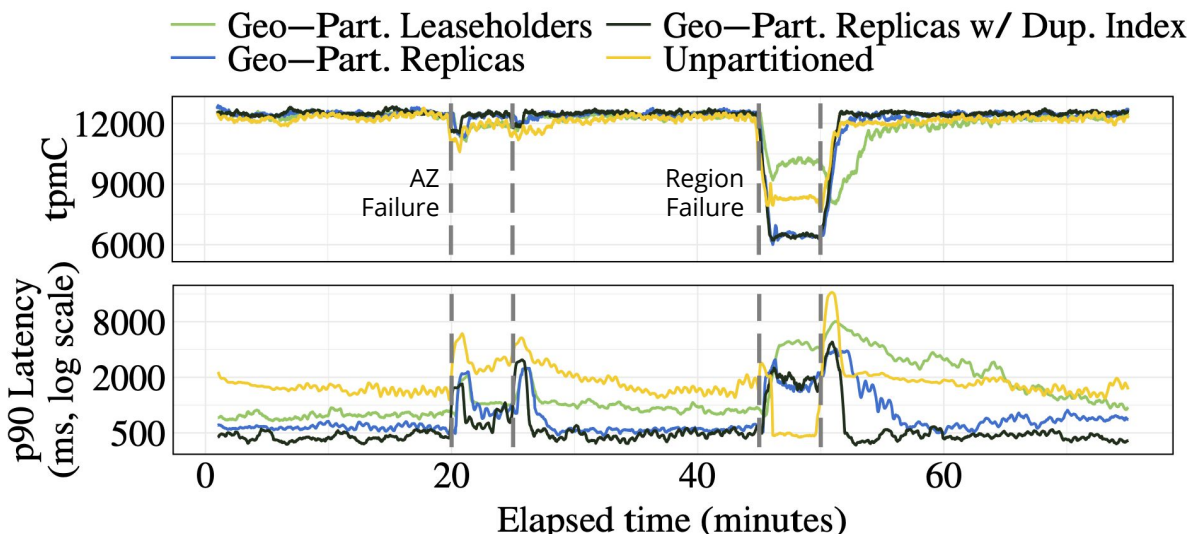




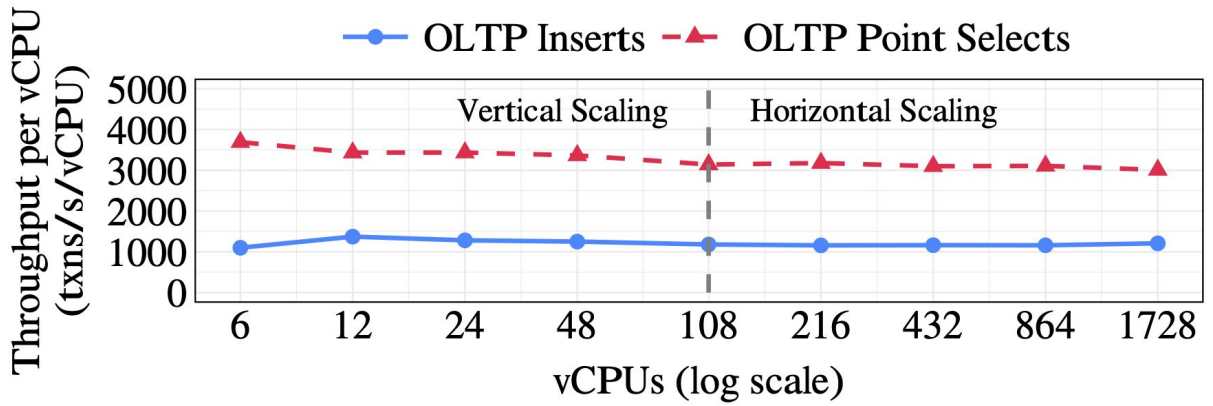
## Comparison with Amazon Aurora on TPC-C

	Warehouses		
	1,000	10,000	100,000
<b>CockroachDB</b>			
Max tpmC	12,474	124,036	1,245,462
Efficiency	97.0%	96.5%	98.8%
NewOrder p90 latency	39.8 ms	436.2 ms	486.5 ms
Machine type (AWS)	c5d.4xlarge	c5d.4xlarge	c5d.9xlarge
Node count	3	15	81
<b>Amazon Aurora [55]</b>			
Max tpmC	12,582	9,406	-
Efficiency	97.8%	7.3%	-
<i>Latency, machine type, and node count not reported</i>			

# Multi-Region TPC-C Performance with AZ and Region Failure



## Scalability on sysbench



# Overheads of CRDB Layers

