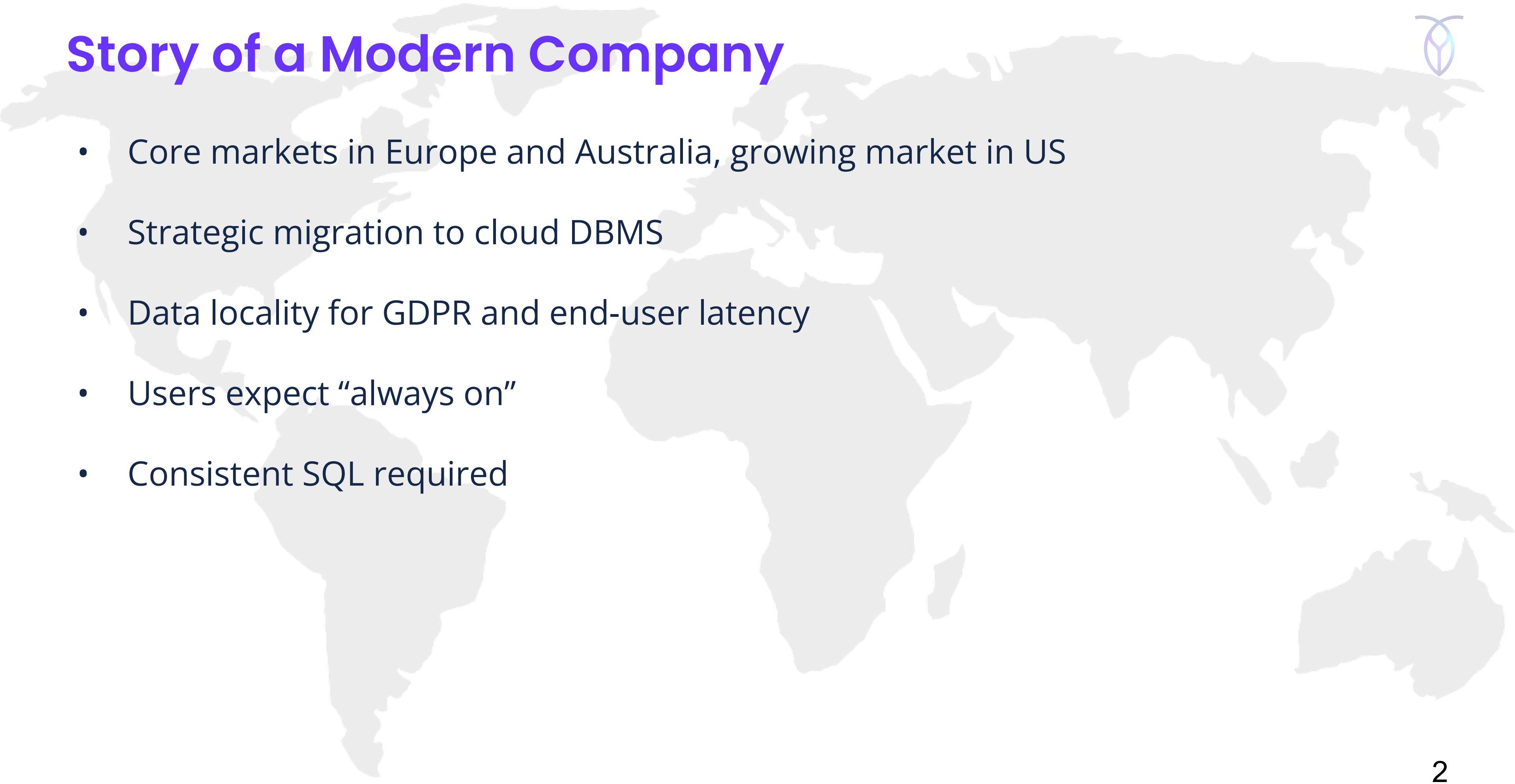
Cockroach Labs

CockroachDB: The Resilient Geo-Distributed SQL Database

Stanford University CS 245, March 3, 2022

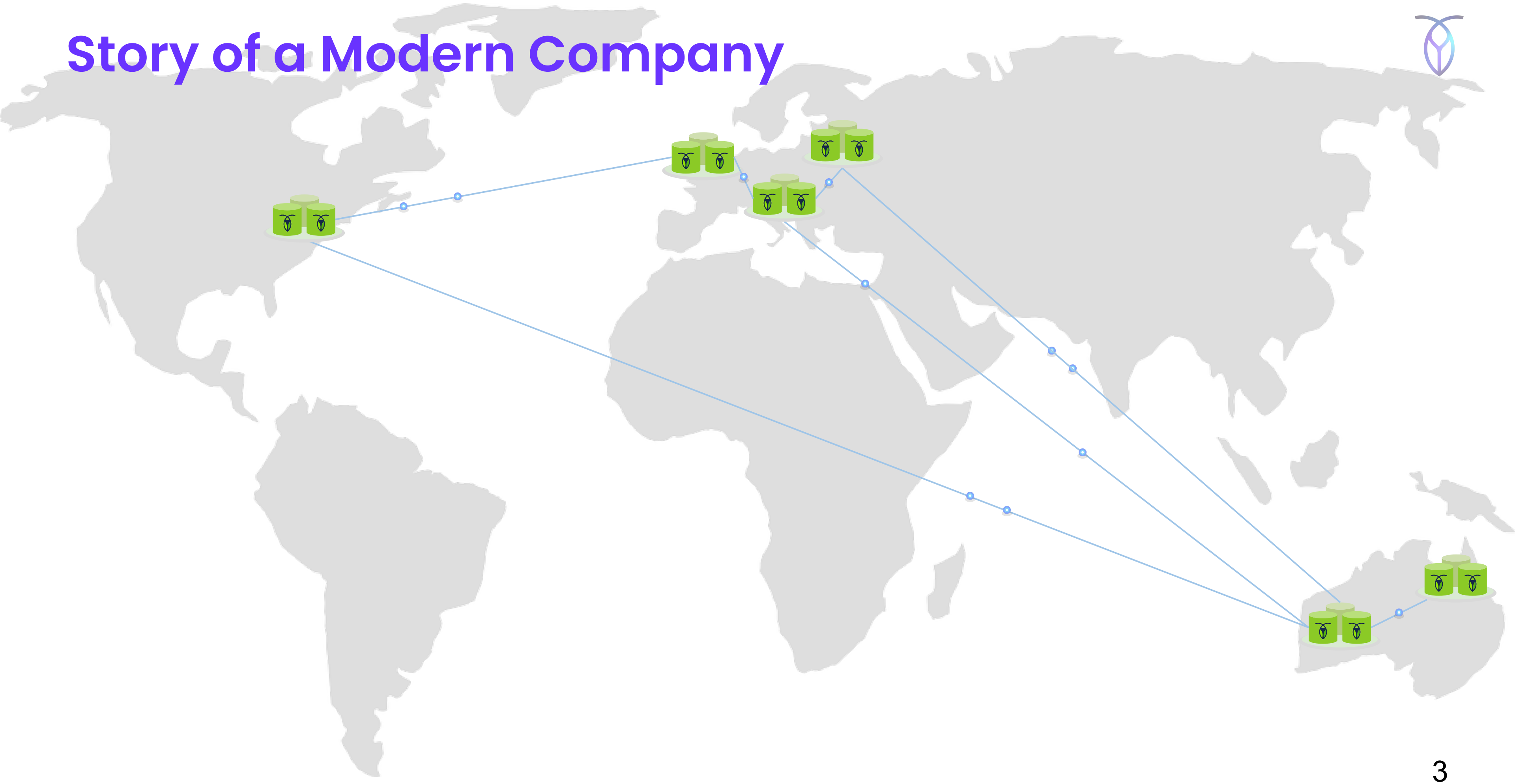
Presented by Rebecca Taft



Story of a Modern Company

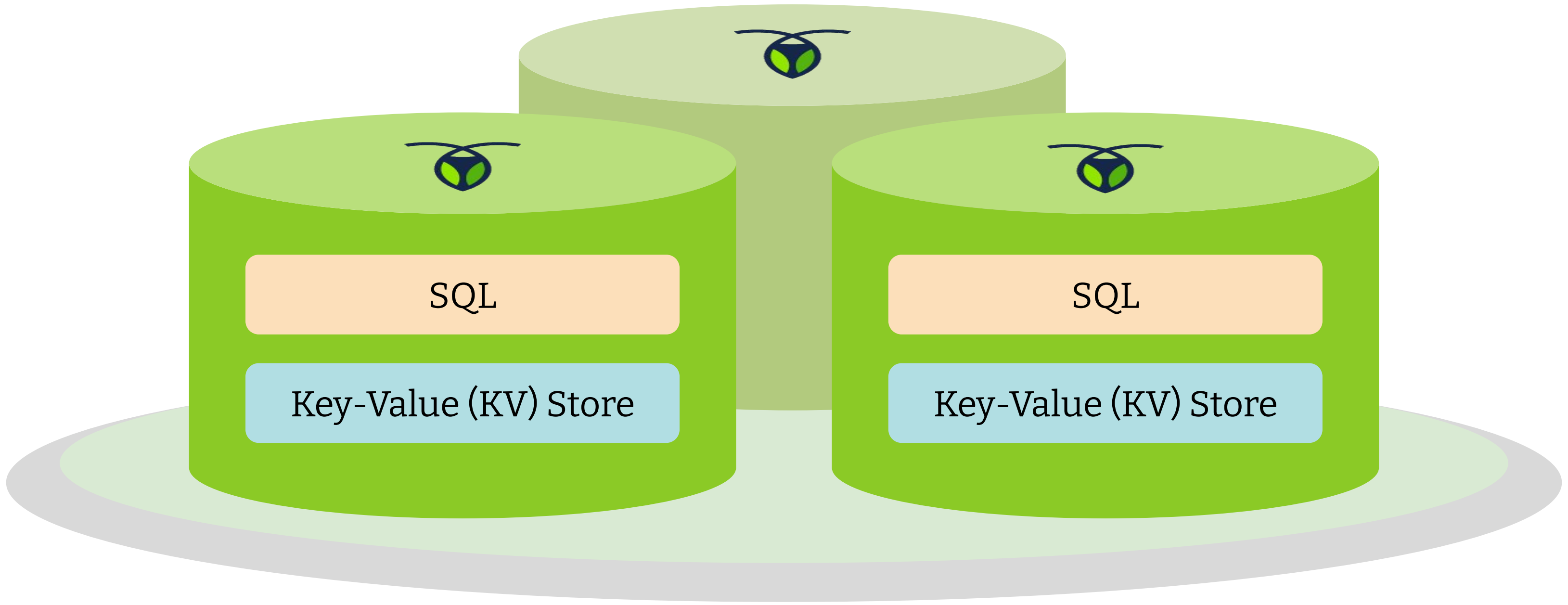
- Core markets in Europe and Australia, growing market in US
- Strategic migration to cloud DBMS
- Data locality for GDPR and end-user latency
- Users expect “always on”
- Consistent SQL required

Story of a Modern Company

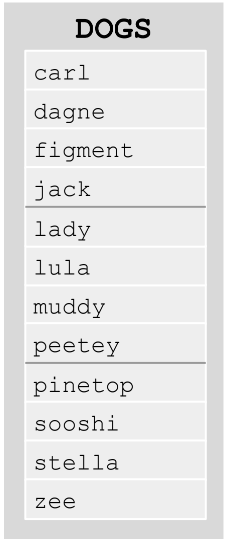


Agenda

Architecture of CockroachDB



Monolithic Key Space



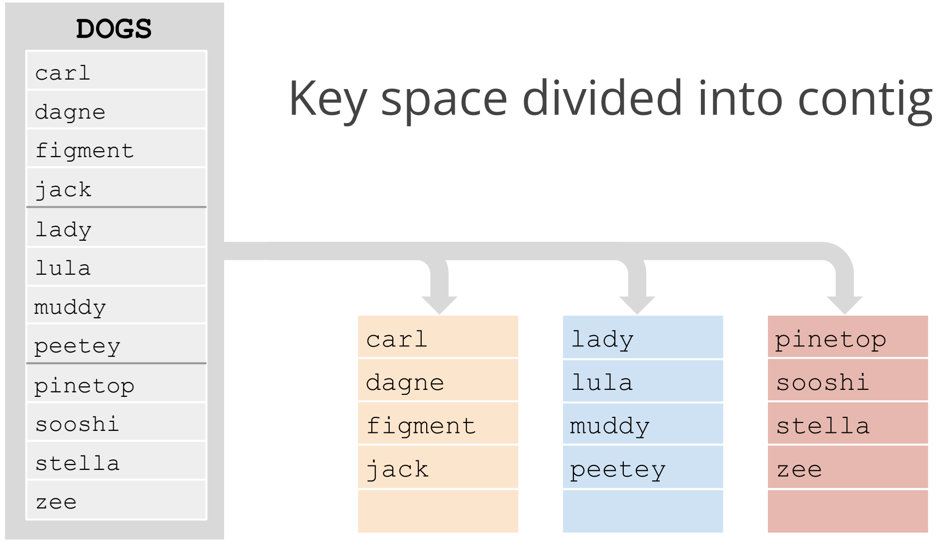
DOGS
carl
dagne
figment
jack
lady
lula
muddy
peetey
pinetop
sooshi
stella
zee

Monolithic logical key space

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

Ranges

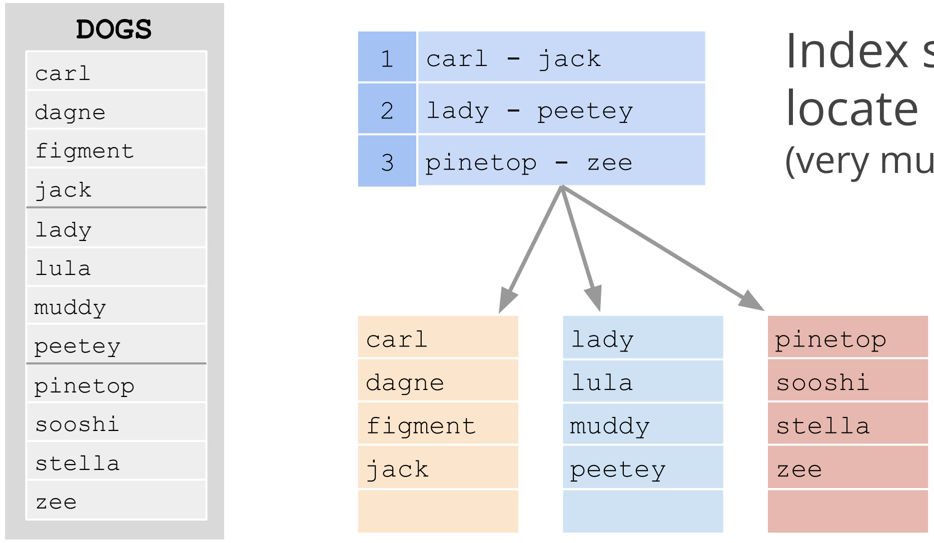
Key space divided into contiguous ~512MB ranges



Ranges are small enough to be moved/split quickly

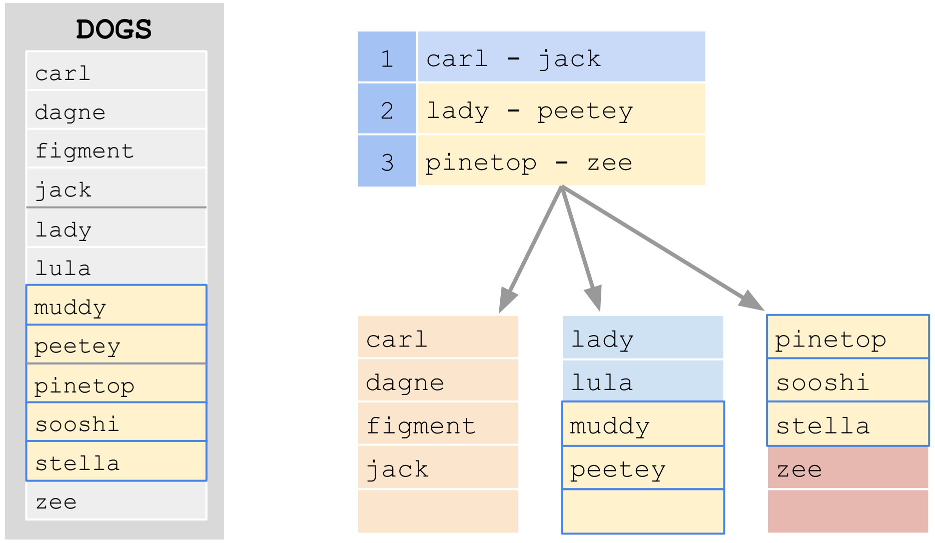
Ranges are large enough to amortize indexing overhead

Range Indexing



Index structure used to locate ranges
(very much like a B-tree)

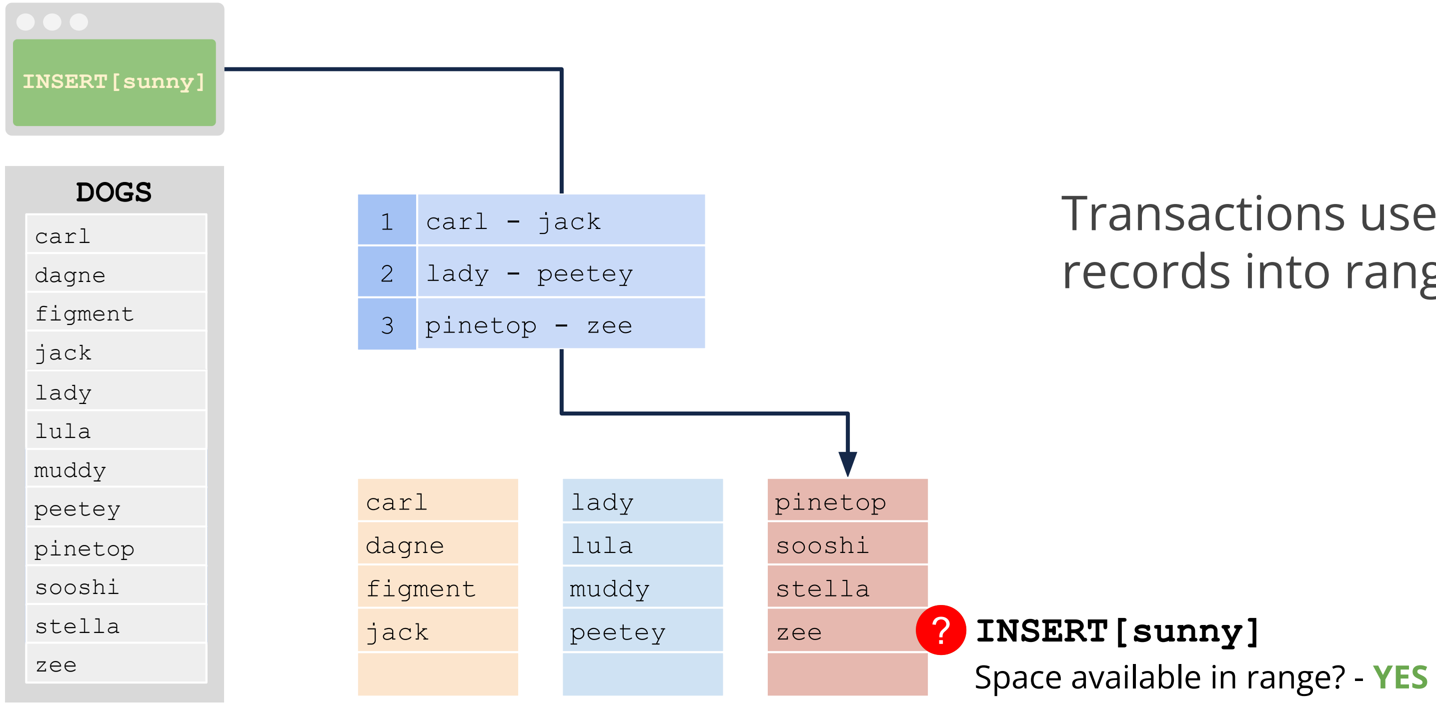
Ordered Range Scans



Ordered keys enable efficient range scans

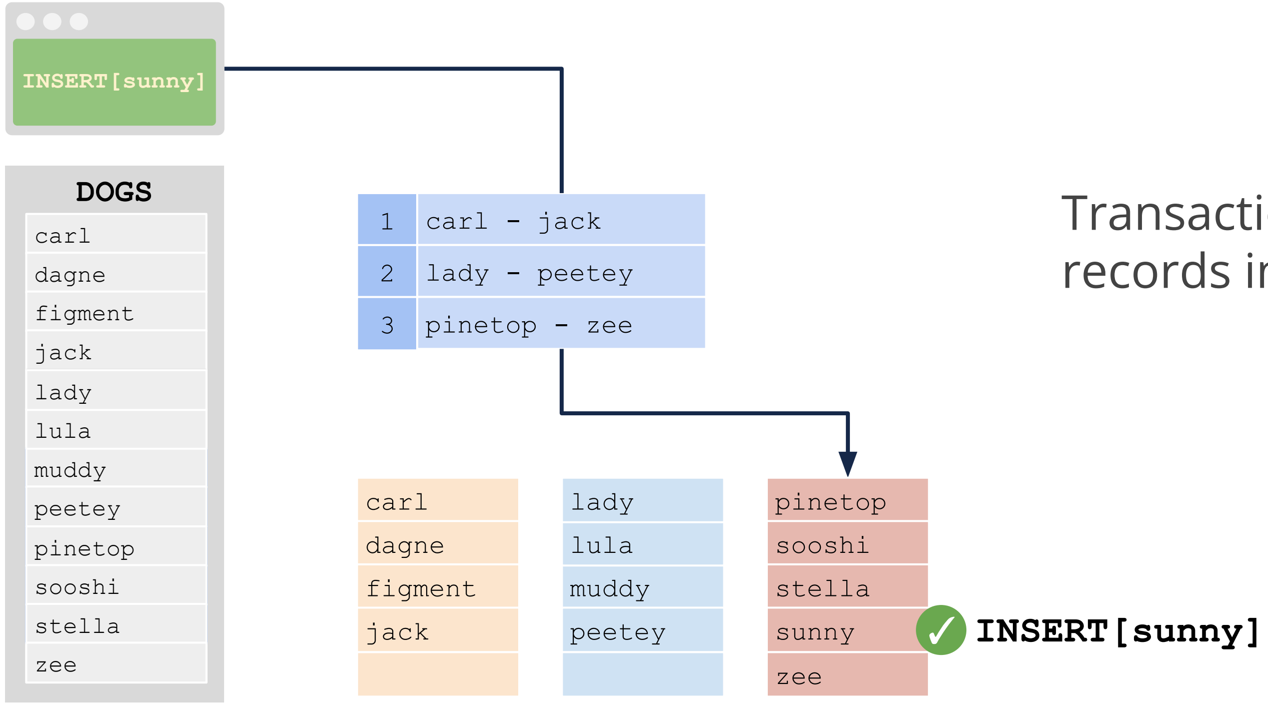
```
dogs >= “muddy” AND <= “stella”
```

Transactional Updates



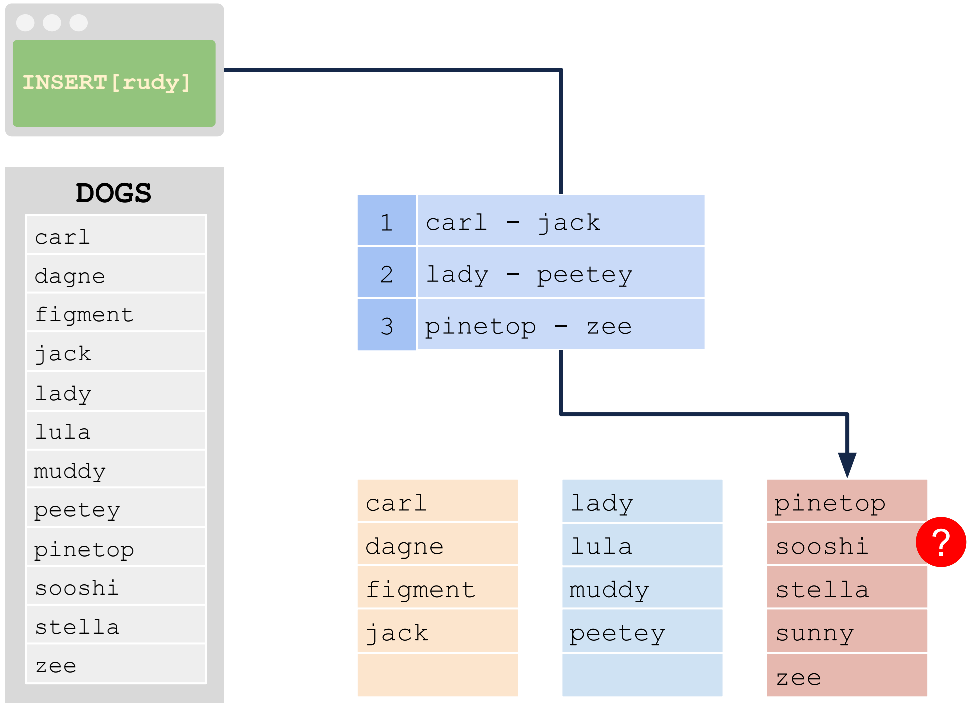
Transactions used to insert records into ranges

Transactional Updates



Transactions used to insert records into ranges

Range Splits

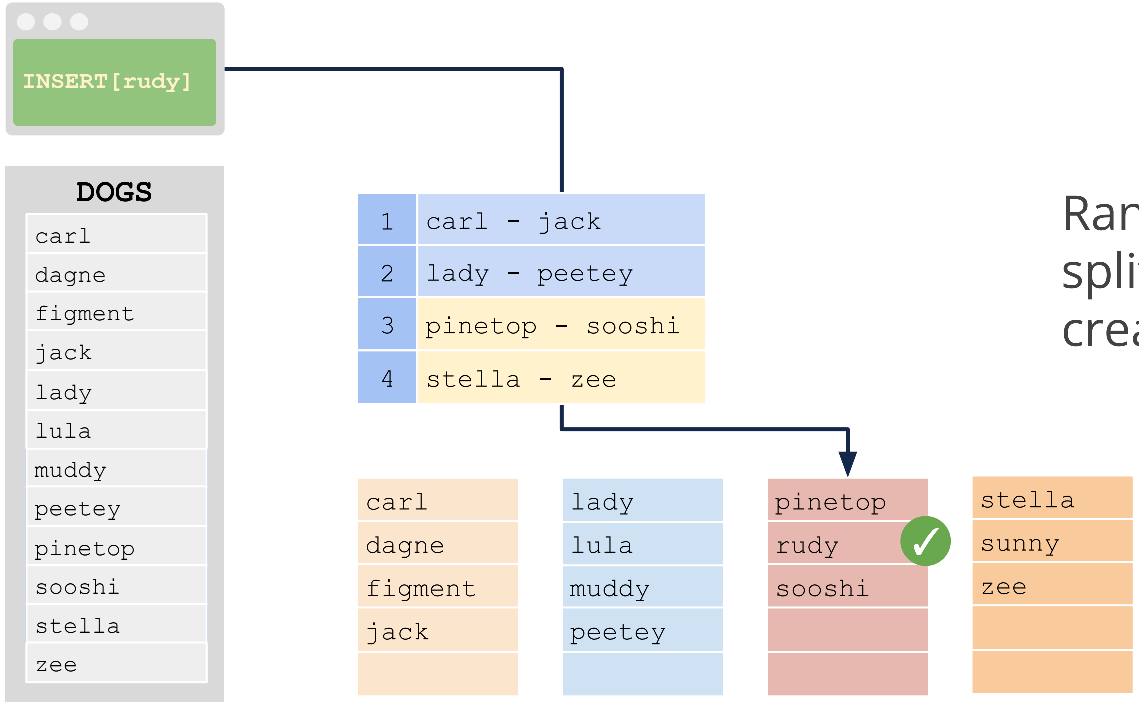


BUT... what happens when a range is full?

INSERT [rudy]

Space available in range? - **NO**

Range Splits



Ranges are automatically split, a new range index is created & order maintained

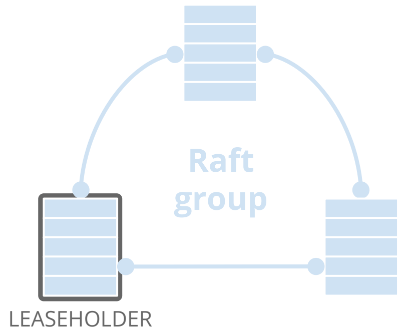
`INSERT [rudy]`

split range and insert

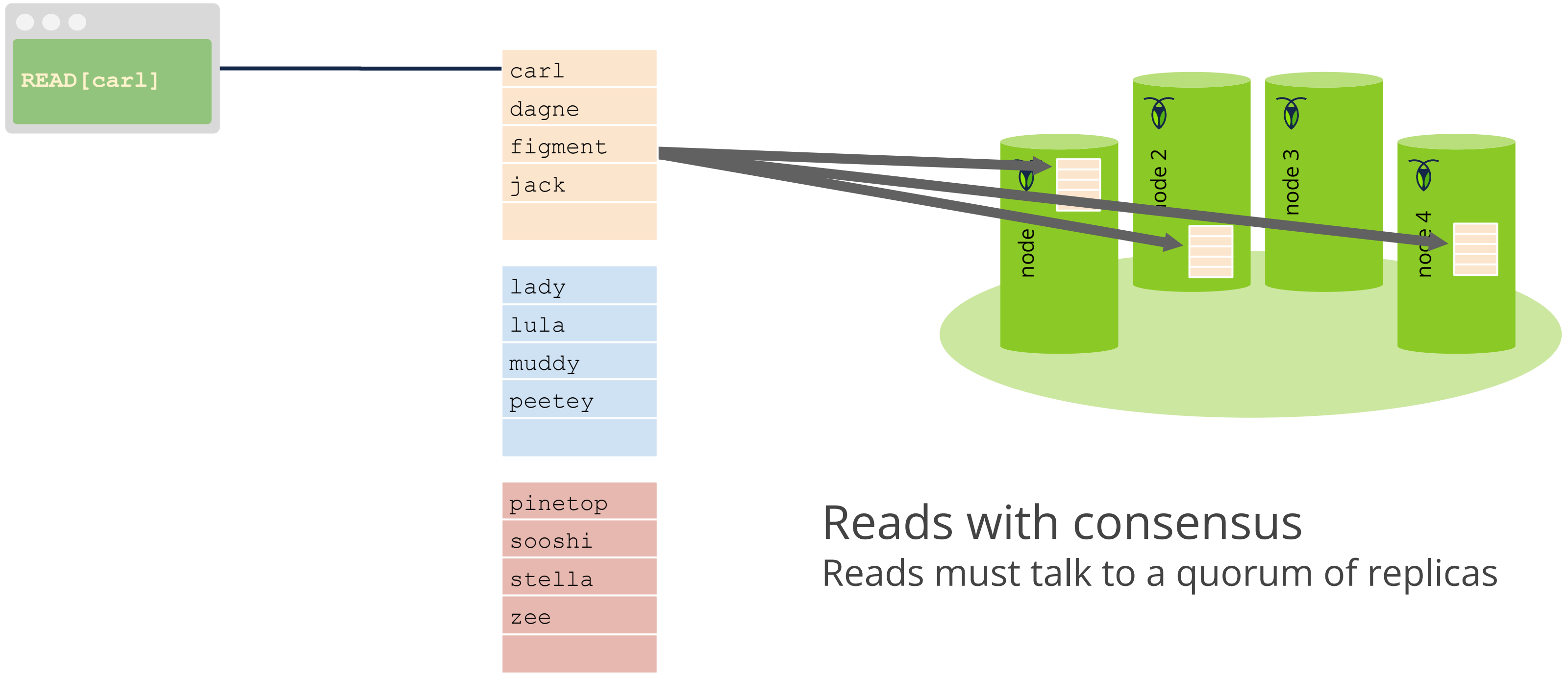
Ranges are the unit of replication

- 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



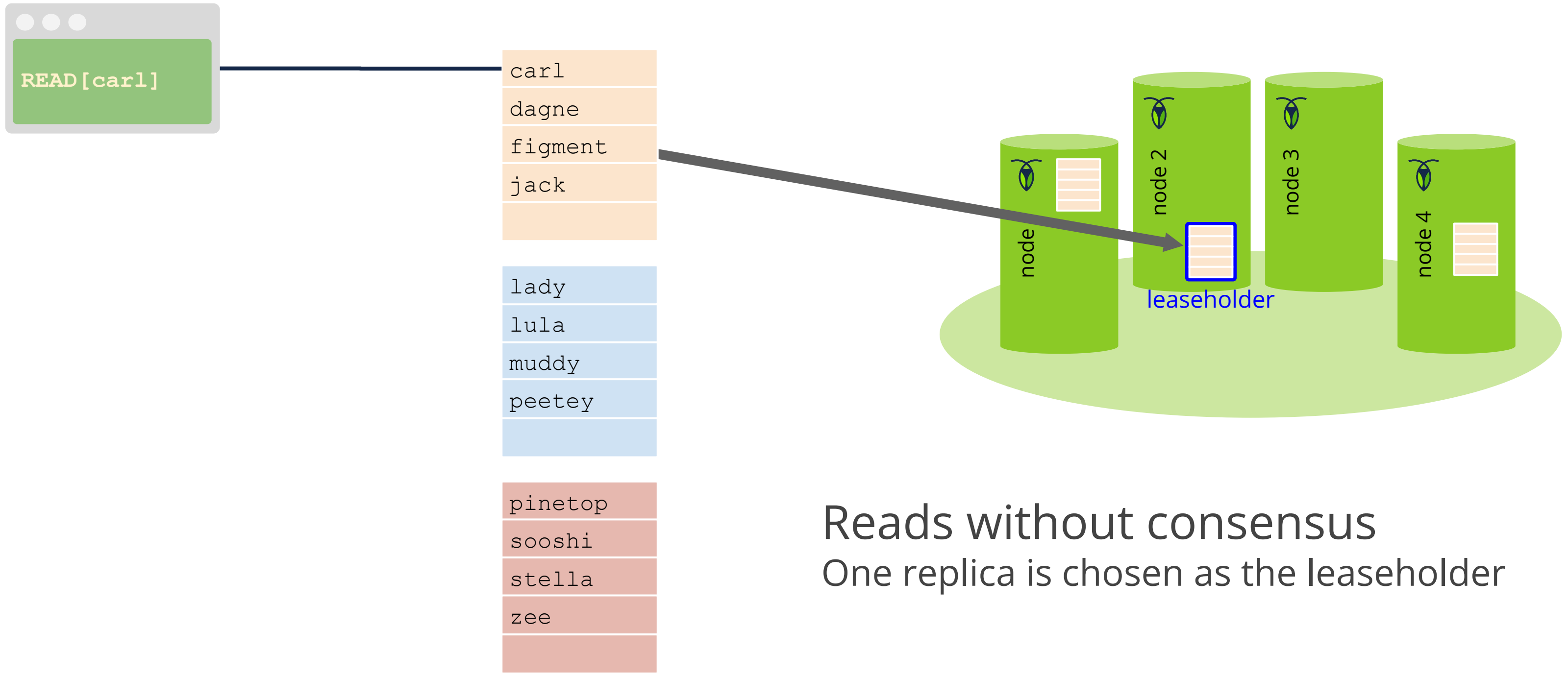
Range Leases



Reads with consensus

Reads must talk to a quorum of replicas

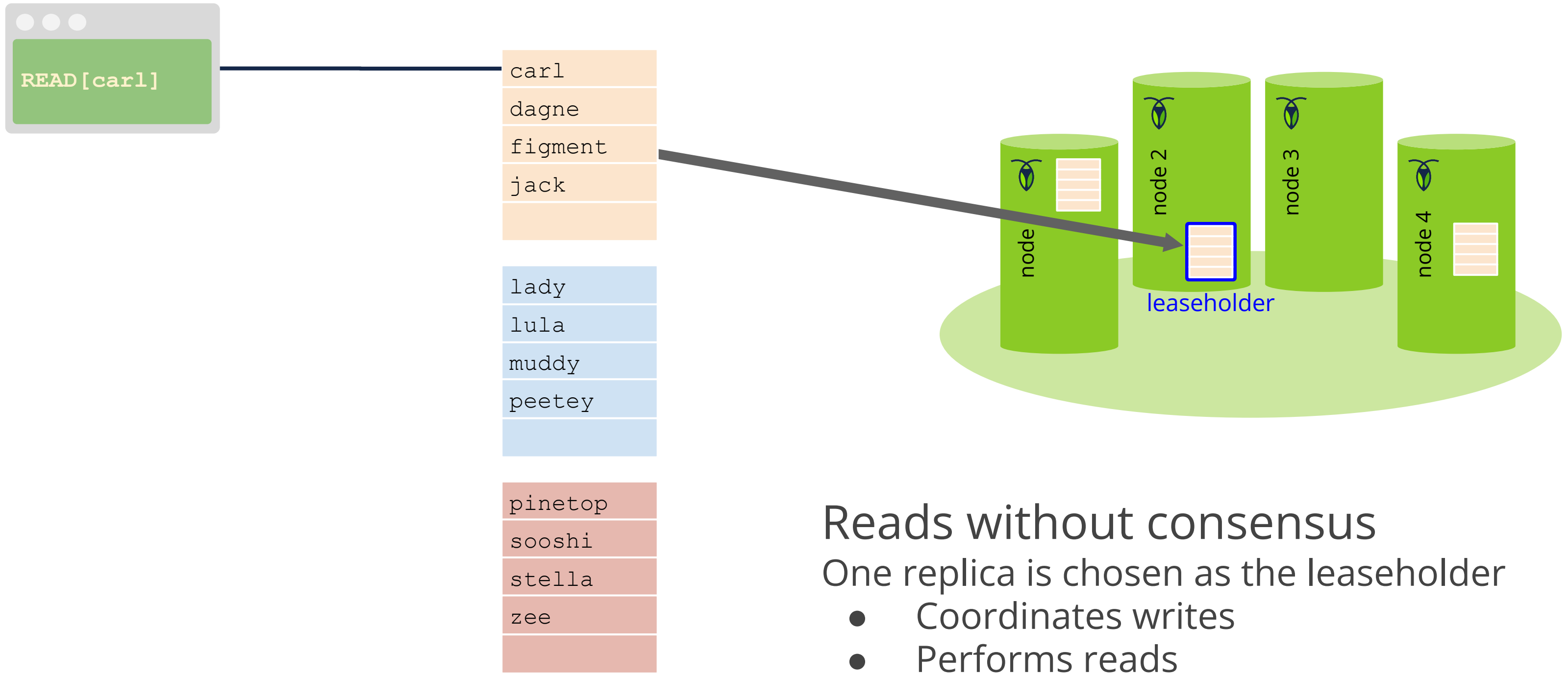
Range Leases



Reads without consensus

One replica is chosen as the leaseholder

Range Leases



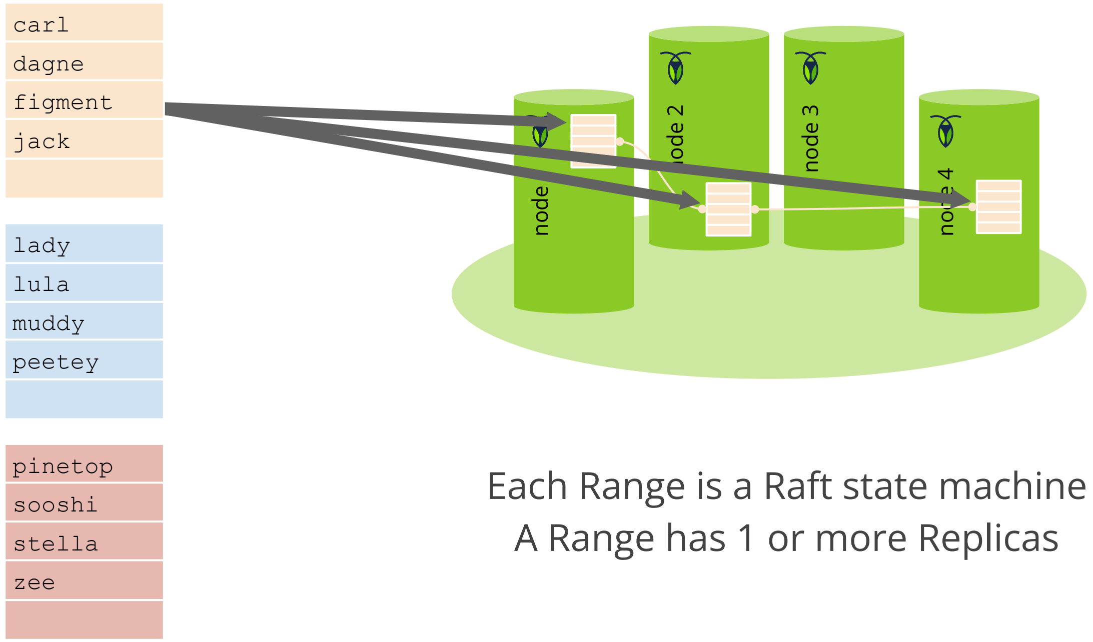
Reads without consensus

One replica is chosen as the leaseholder

- Coordinates writes
- Performs reads

Replica Placement

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



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

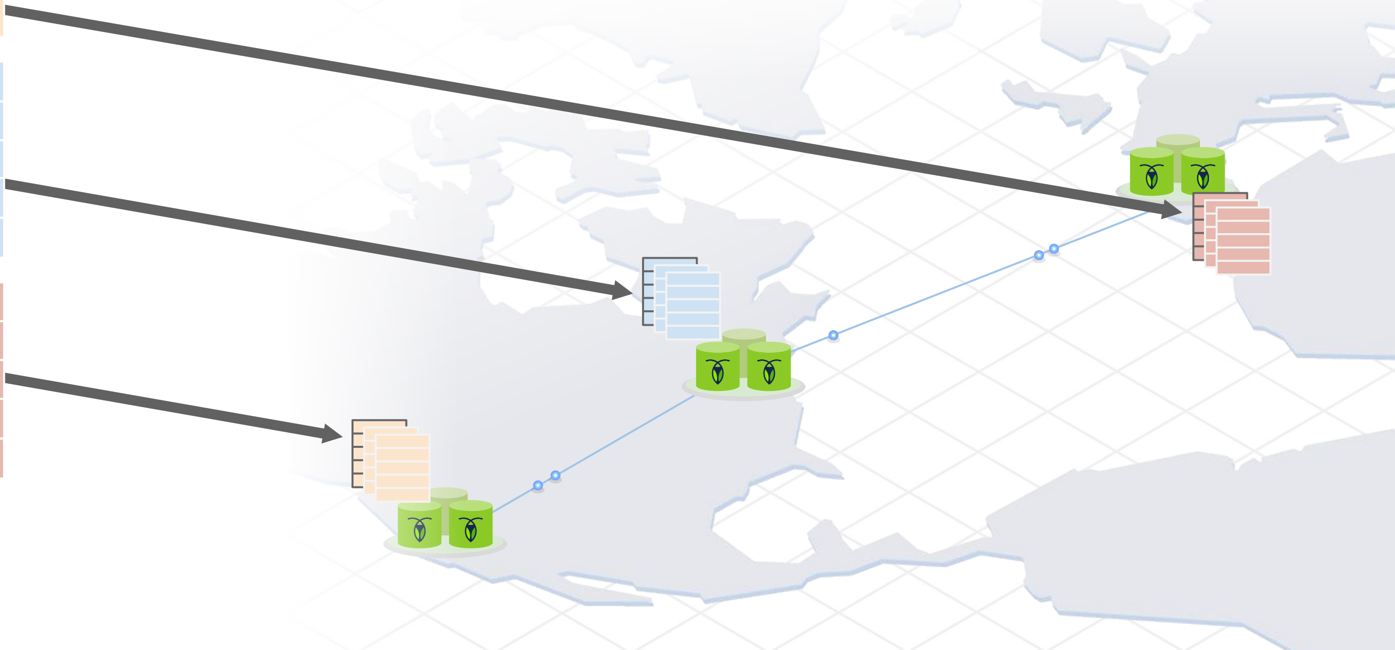
Replica Placement: User-defined constraints & Latency

carl	**EU**/carl
dagne	**EU**/lula
figment	**EU**/sooshi
jack	**EU**/zee

lady	**USE**/dagne
lula	**USE**/figment
muddy	**USE**/muddy
peetey	**USE**/stella

pinetop	**USW**/jack
sooshi	**USW**/lady
stella	**USW**/peetey
zee	**USW**/pinetop

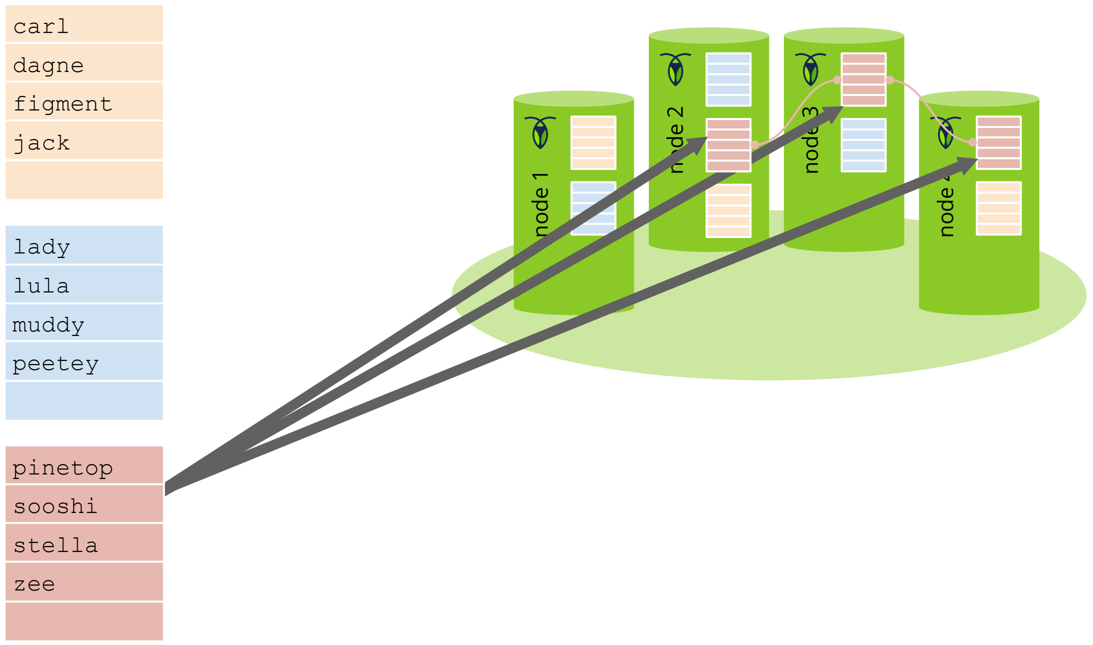
We apply a constraint that indicates regional placement so we can ensure low latency access or jurisdictional control of data



Replica Placement: Diversity

Diversity
optimizes placement of replicas across "failure domains"

- Disk
- Single machine
- Rack
- Datacenter
- Region

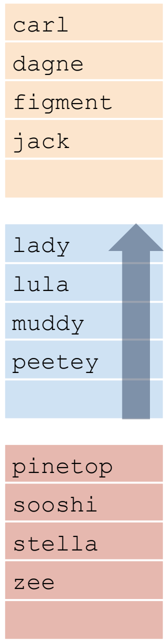


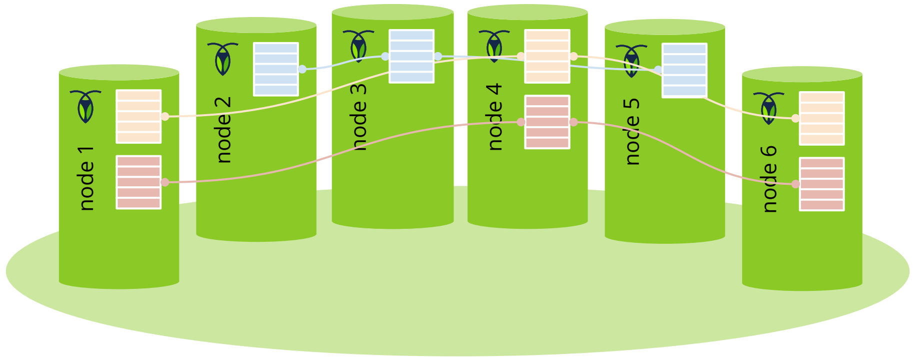
Replica Placement: Load & Space

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





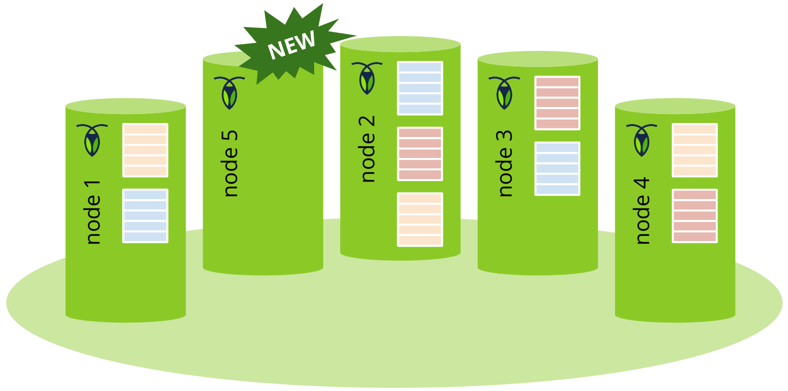
While we show this for ranges within a single table, this is also applicable across all ranges across ALL tables, which is the more typical situation

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

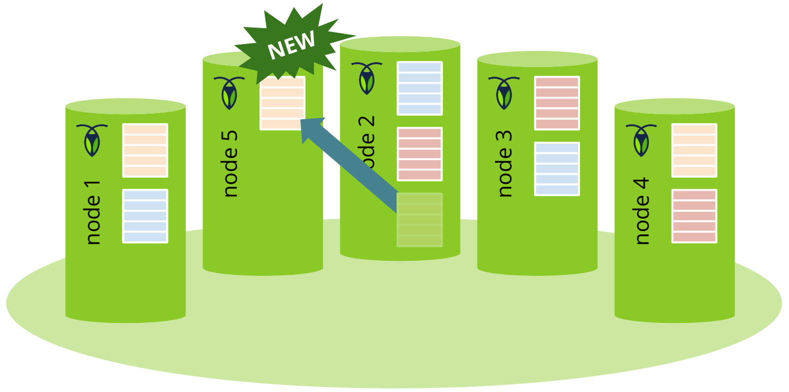


Rebalancing Replicas

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



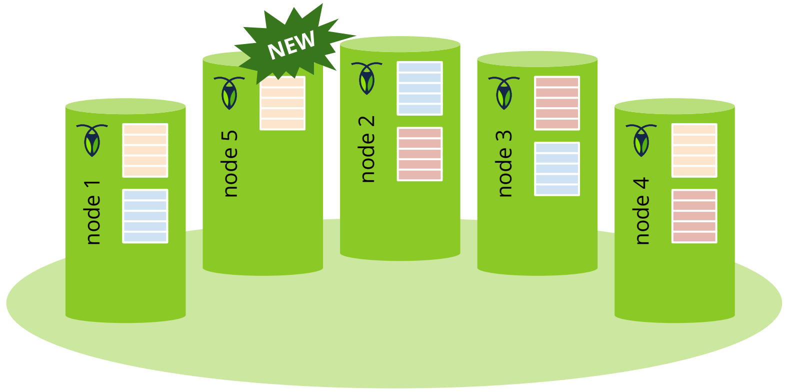
Movement is decomposed into adding a replica followed by removing a replica

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



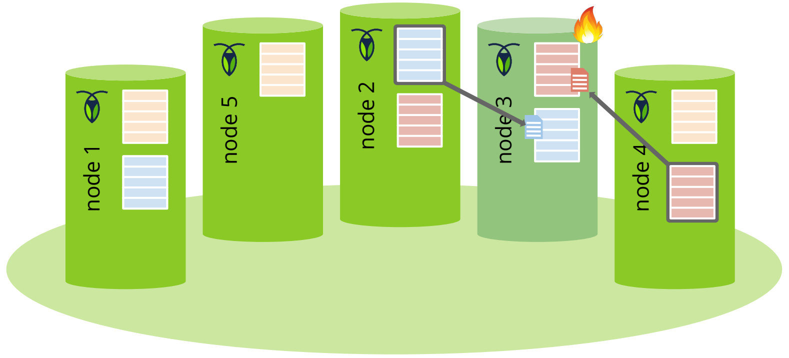
Movement is decomposed into adding a replica followed by removing a replica

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.

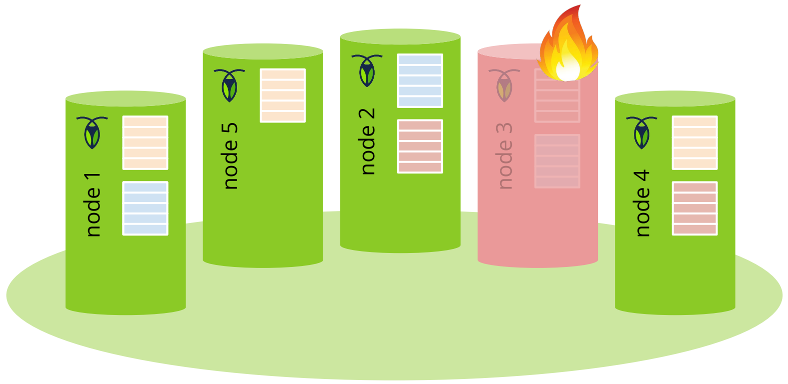
Rebalancing Replicas

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



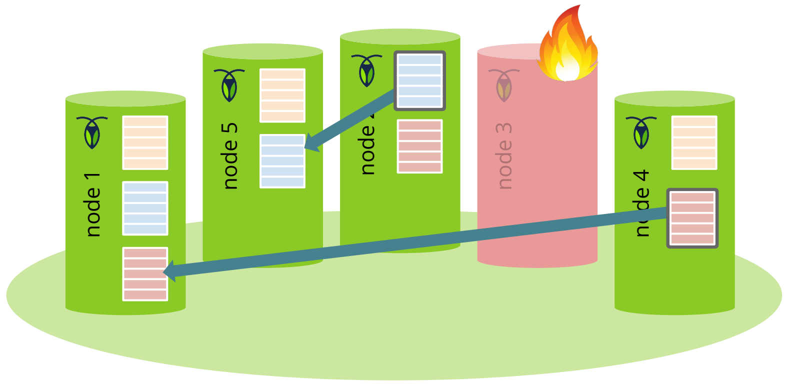
Rebalancing Replicas

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.

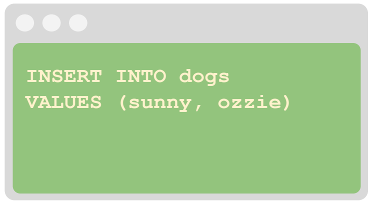
Agenda

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

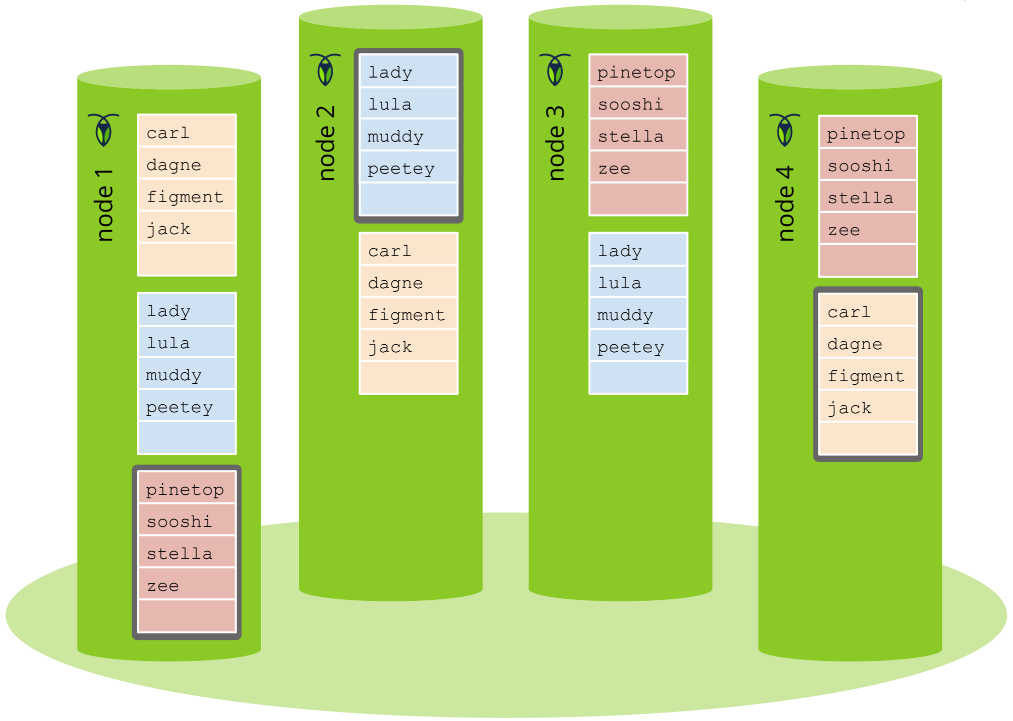
Transactions in CockroachDB are serializable, always

- 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

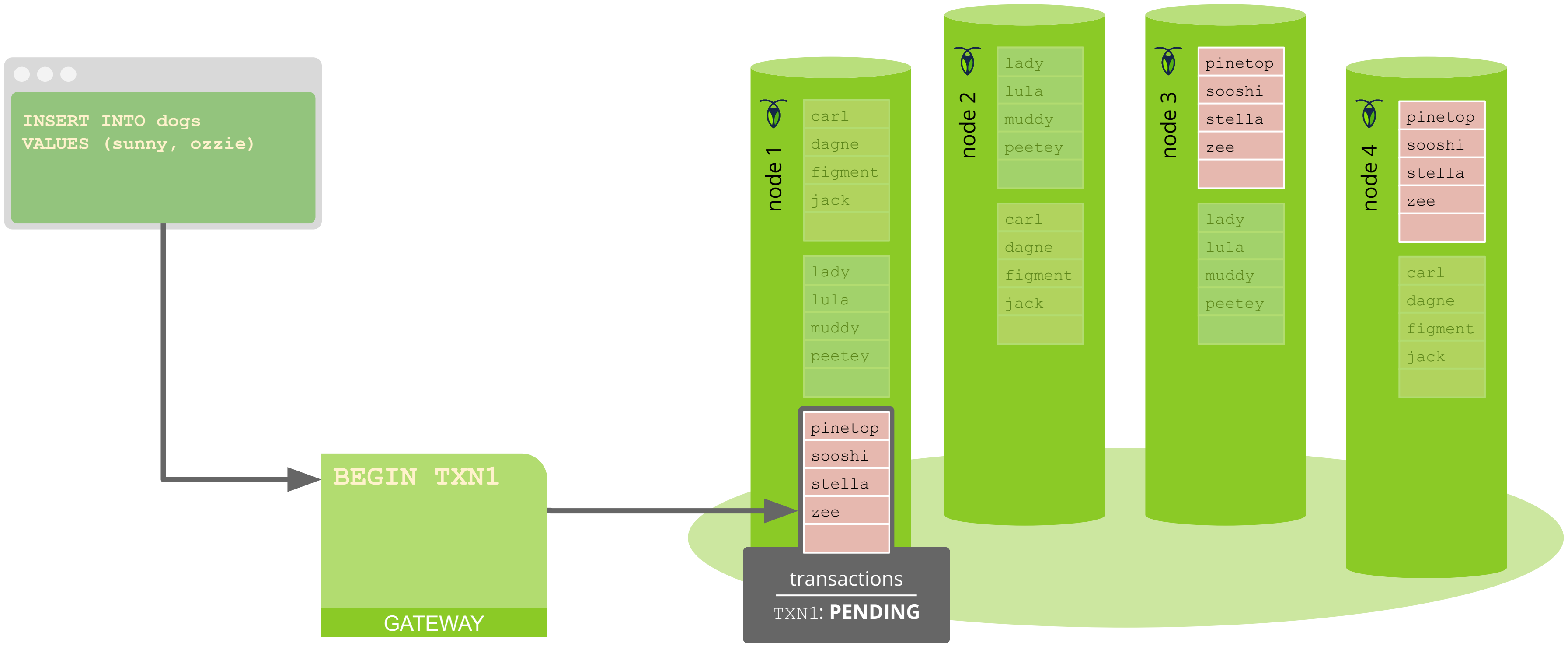
Distributed Transactions



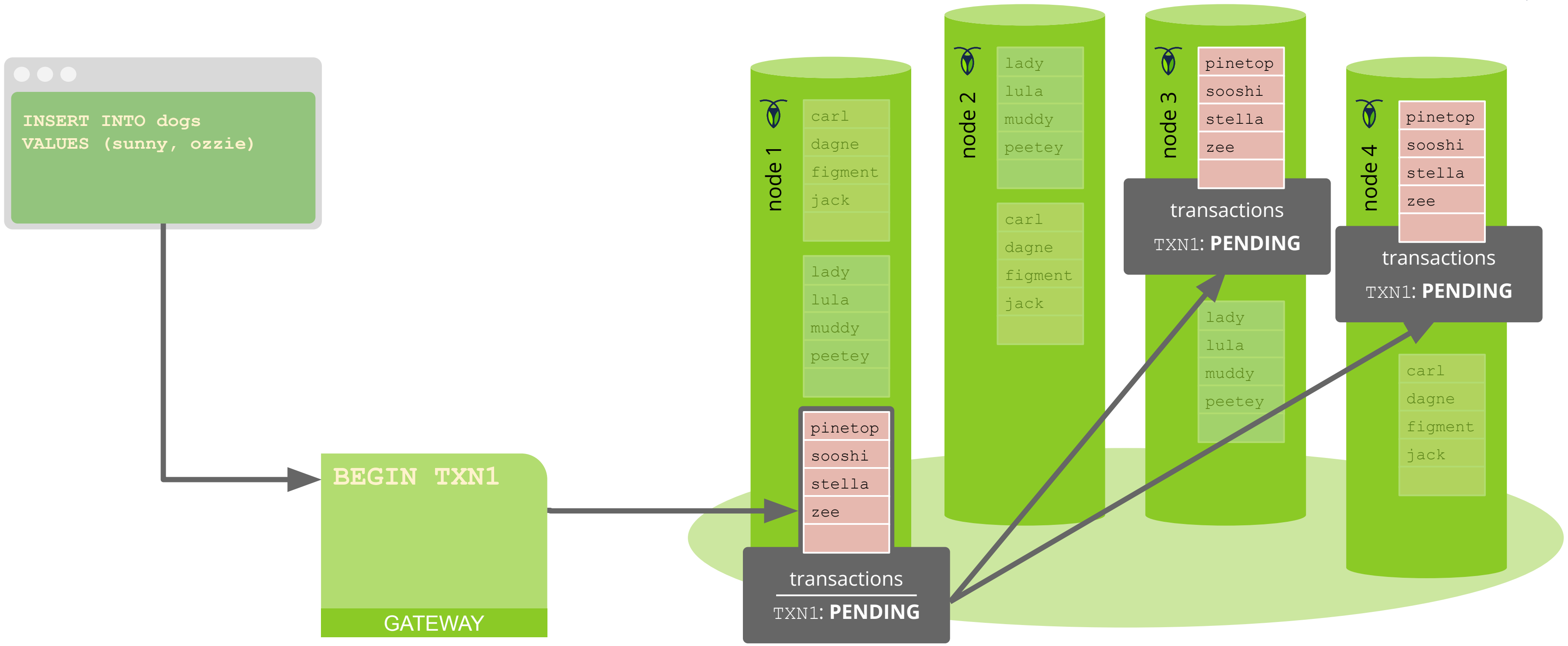
```
INSERT INTO dogs
VALUES (sunny, ozzie)
```



Distributed Transactions

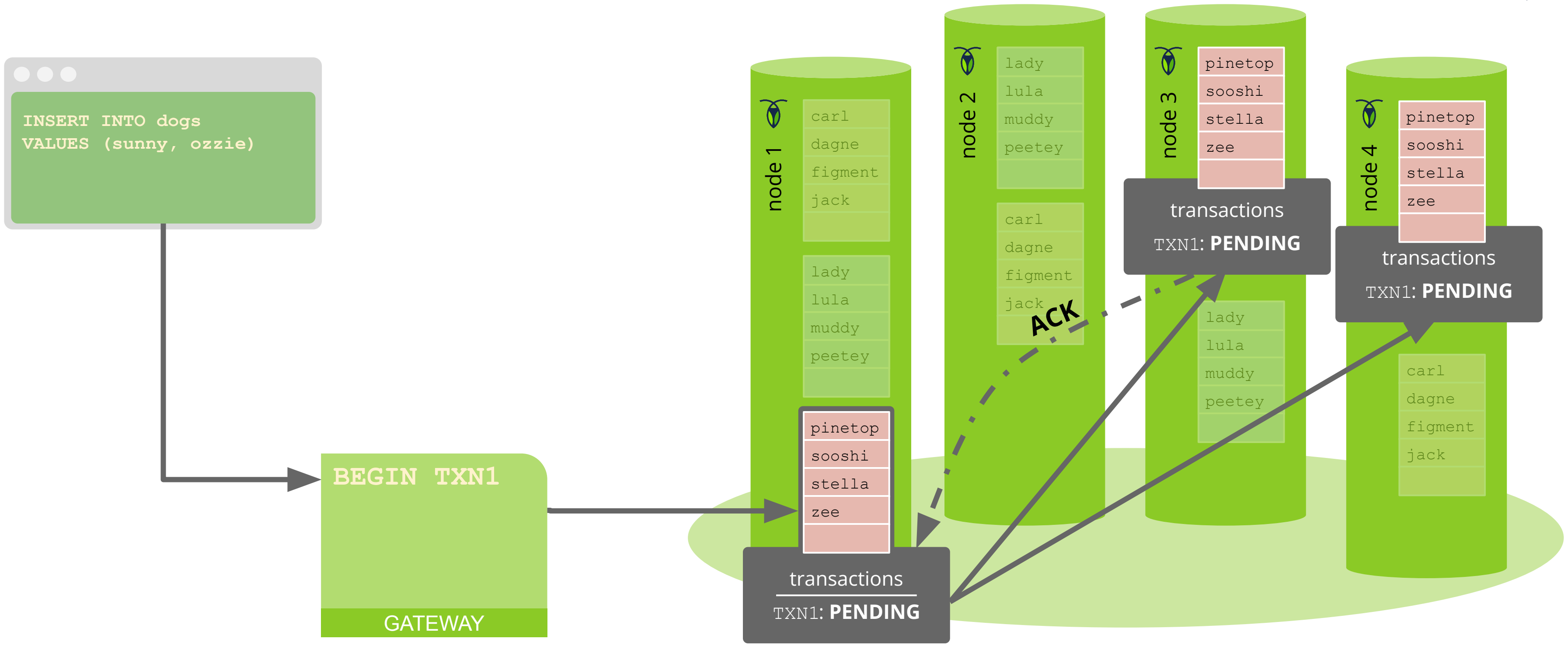

```
INSERT INTO dogs
VALUES (sunny, ozzie)
```

Distributed Transactions

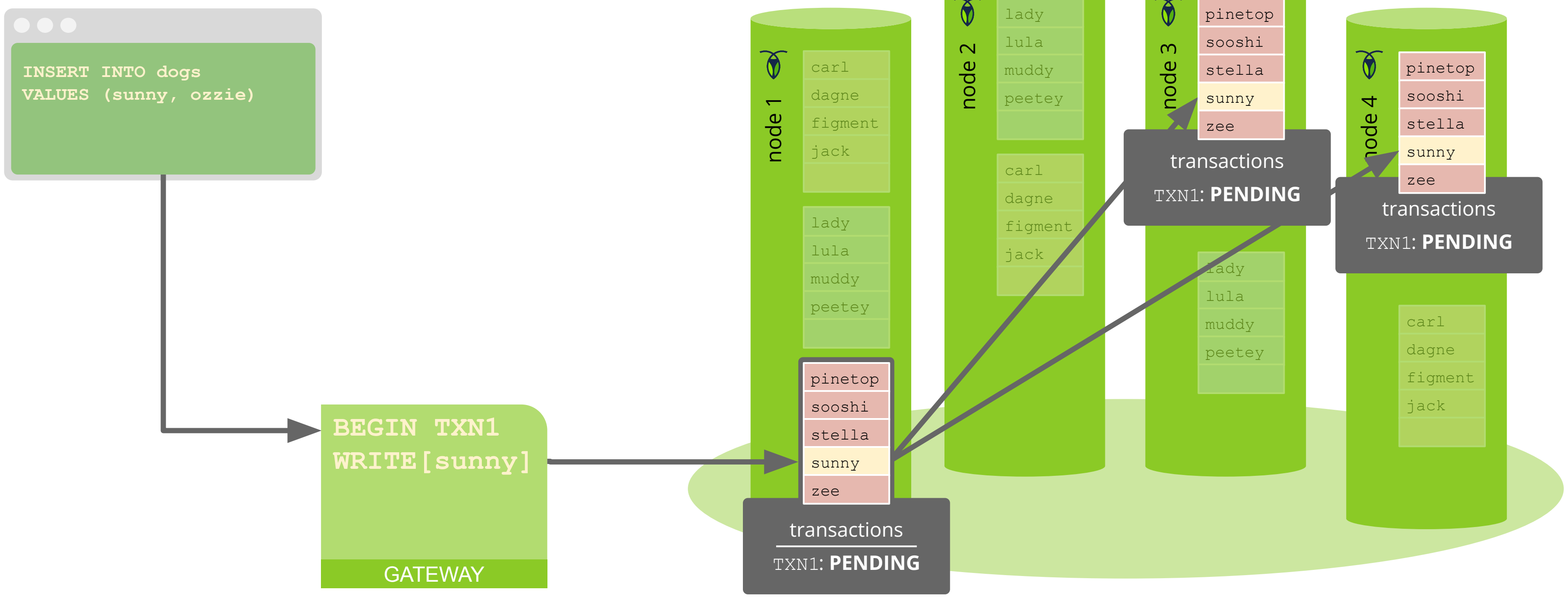


```
INSERT INTO dogs
VALUES (sunny, ozzie)
```

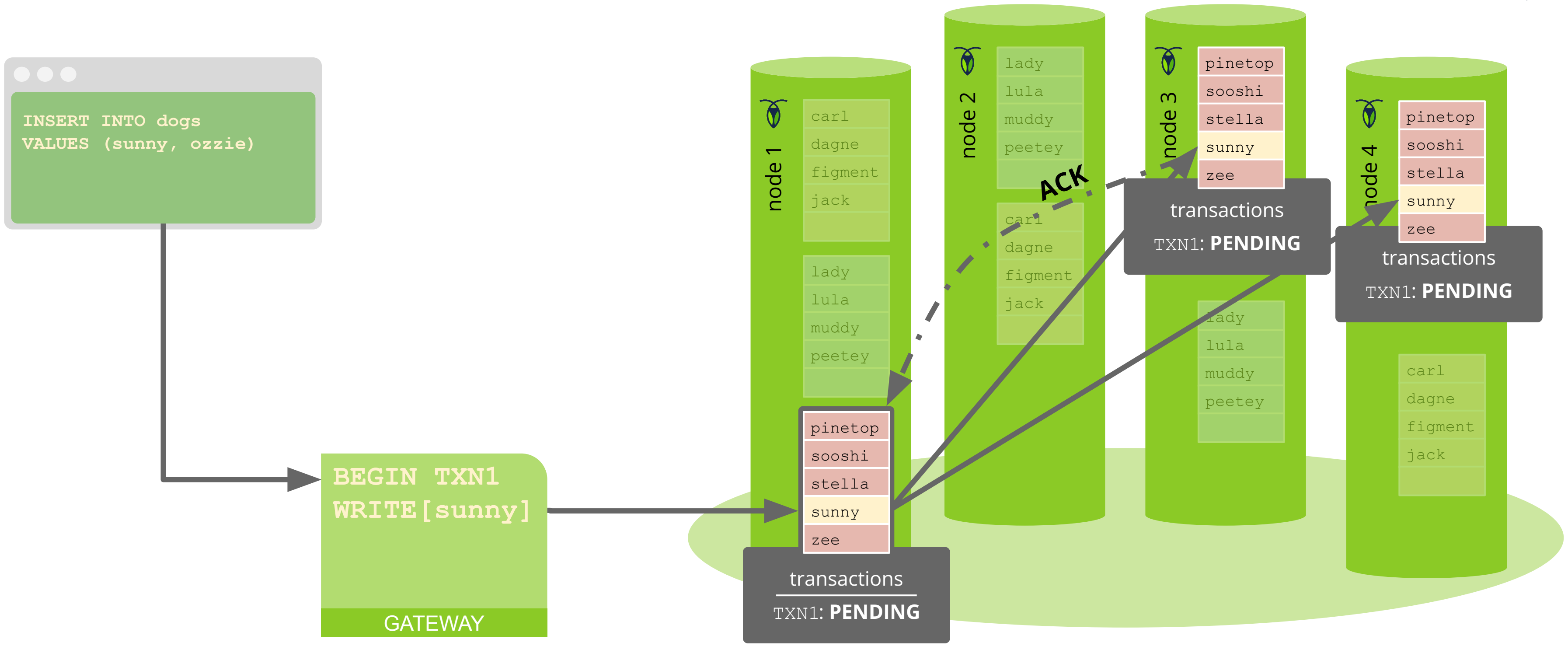
Distributed Transactions



Distributed Transactions

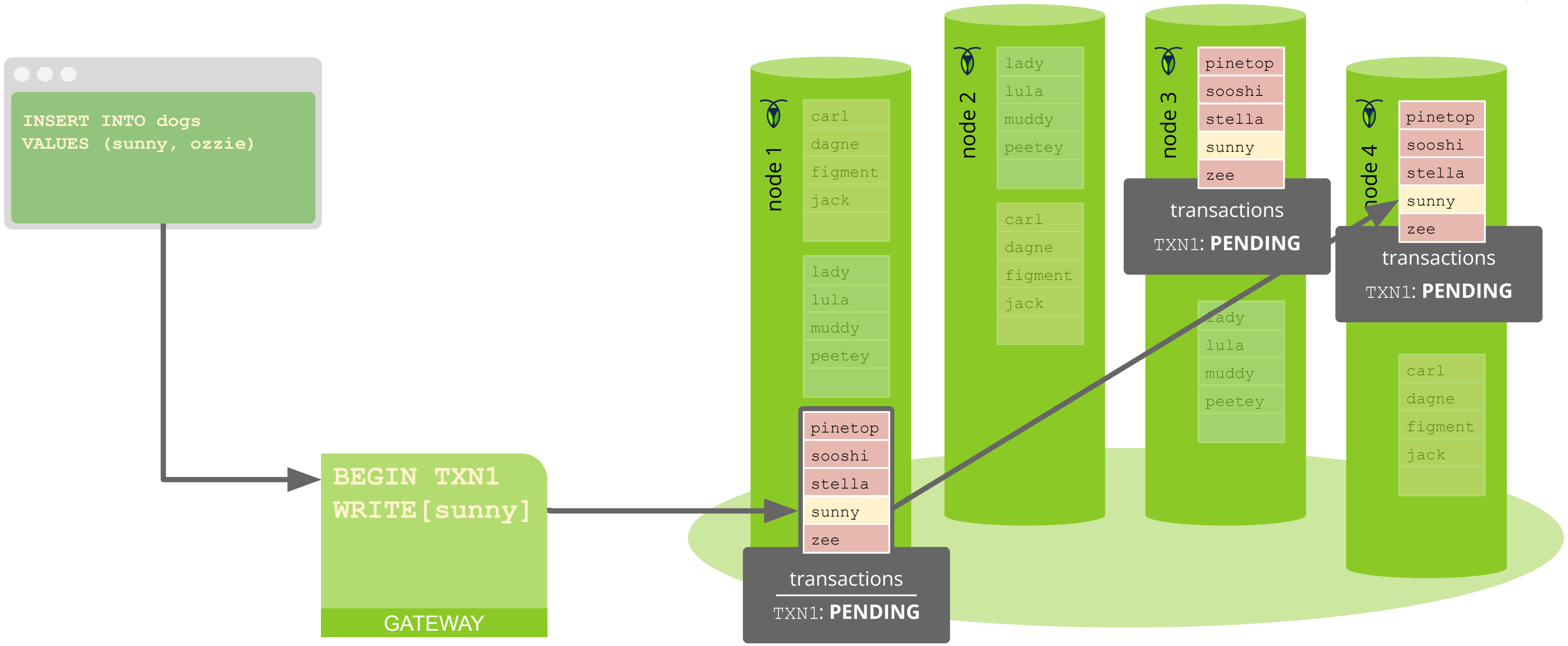


Distributed Transactions

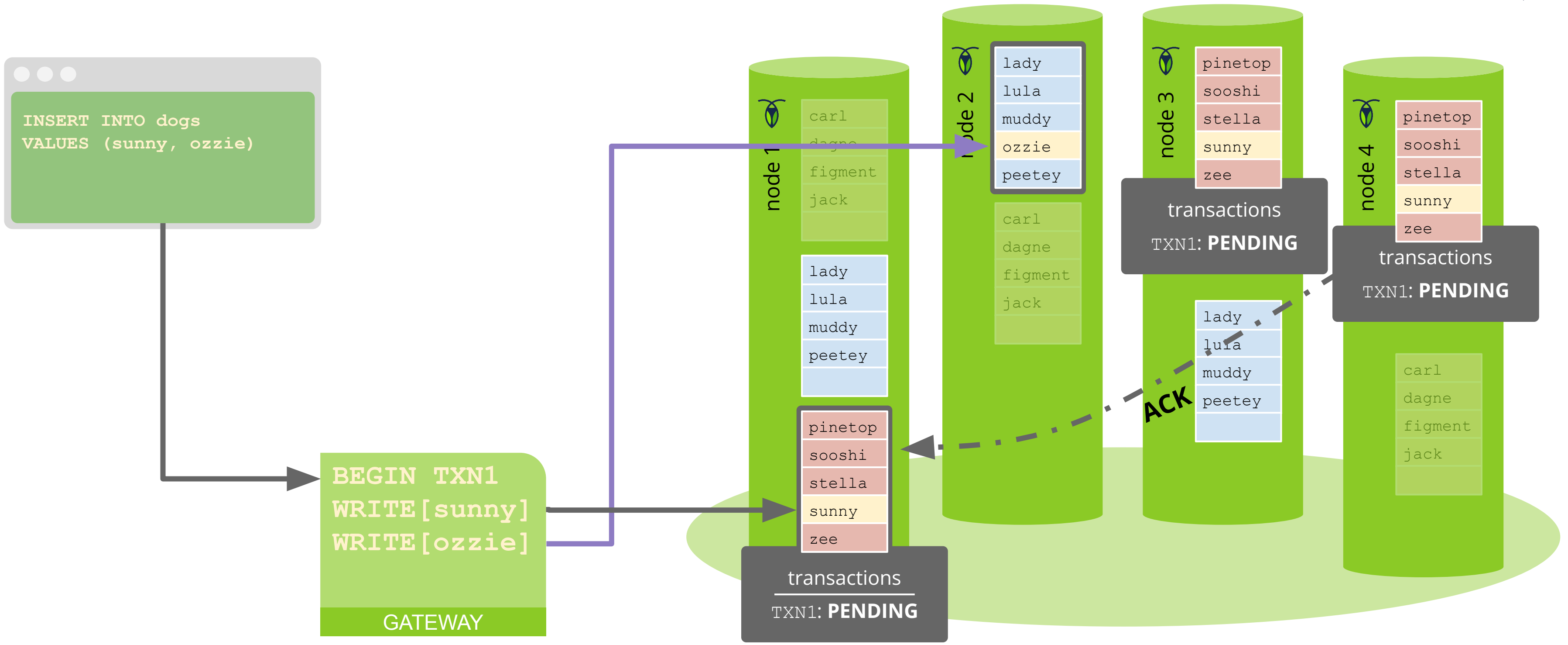

```
INSERT INTO dogs
VALUES (sunny, ozzie)
```

```
BEGIN TXN1
WRITE[sunny]
```

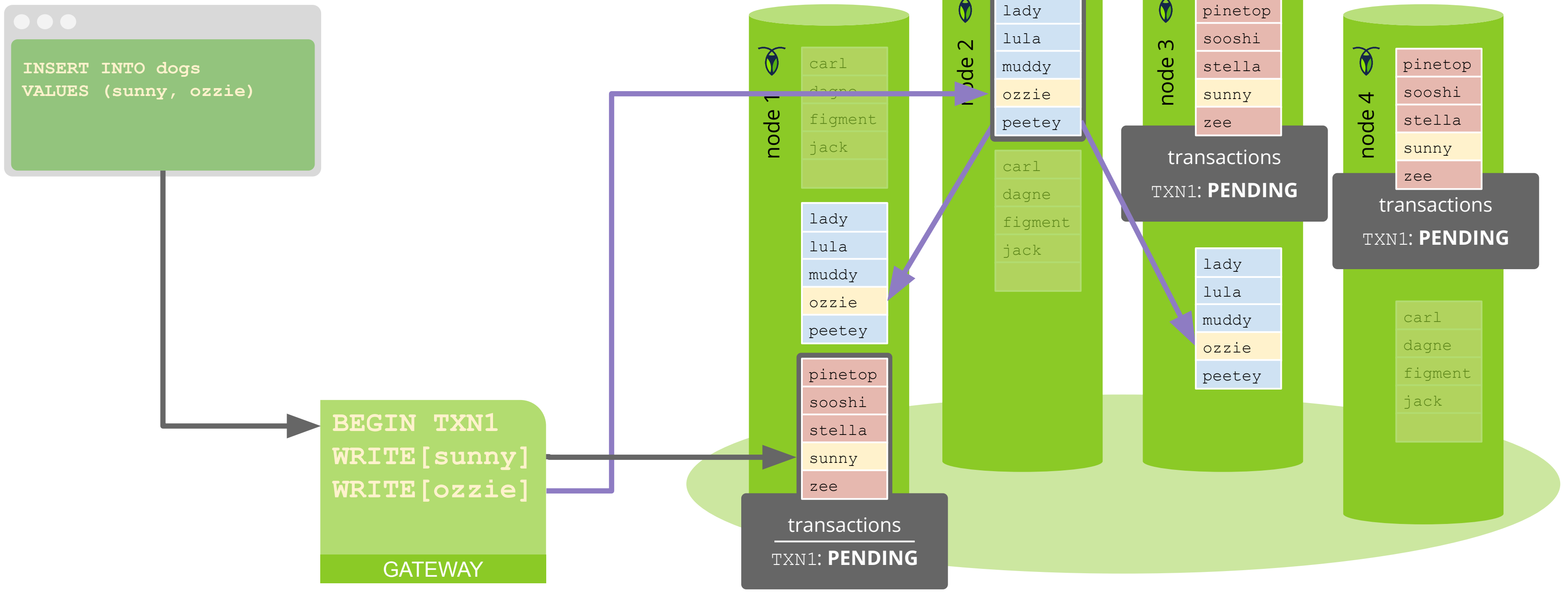
Distributed Transactions



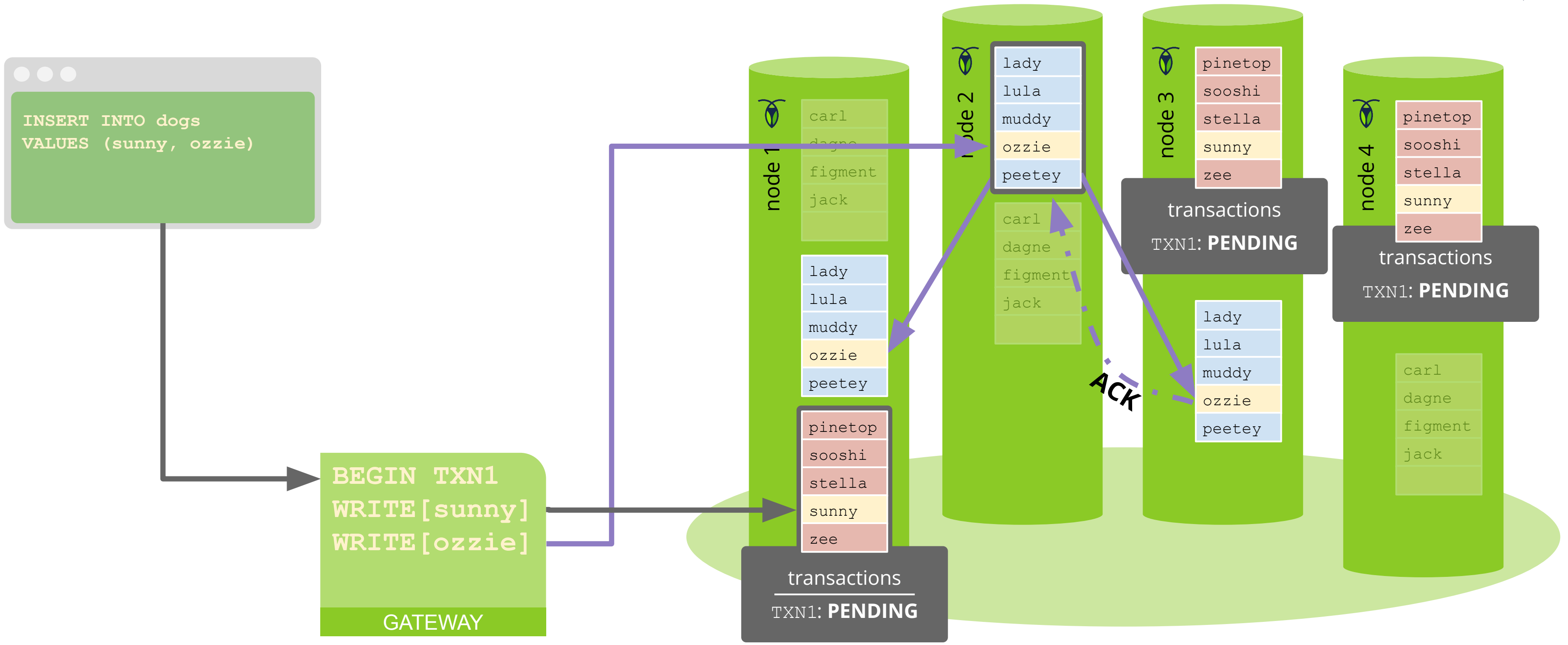
Distributed Transactions



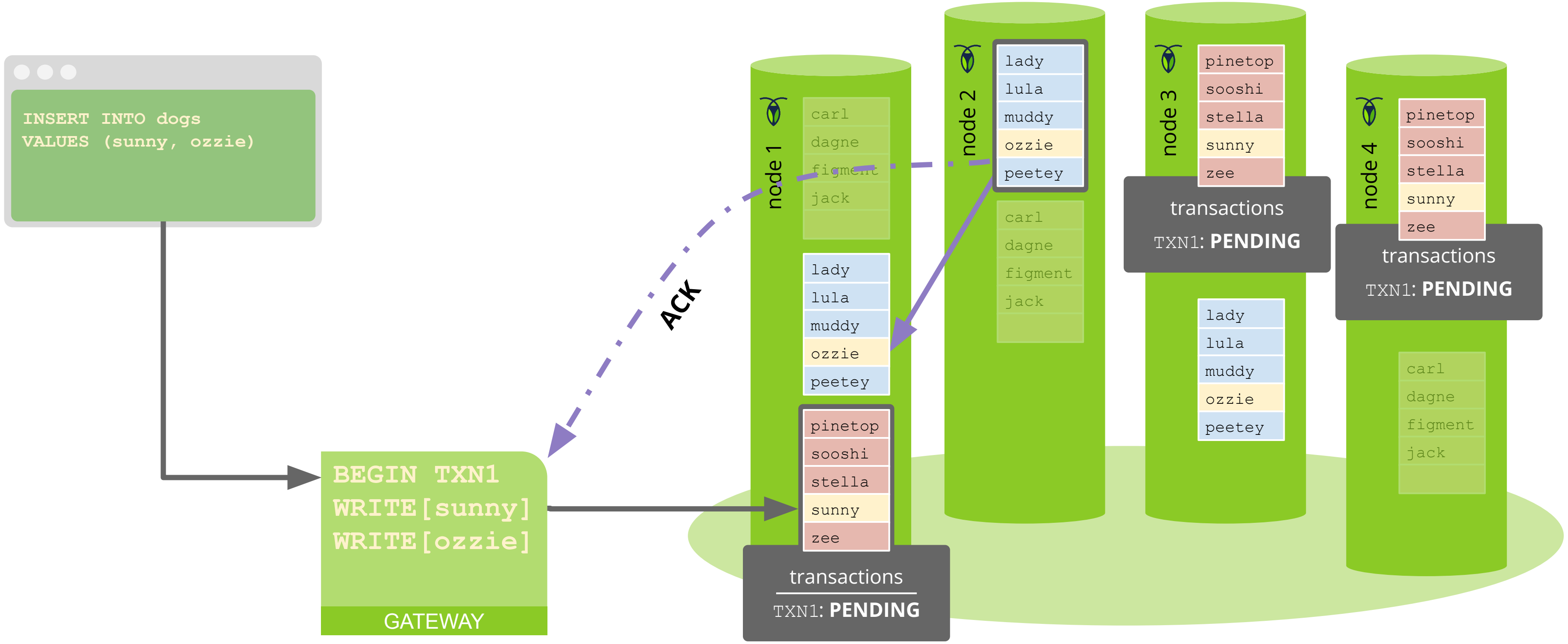
Distributed Transactions



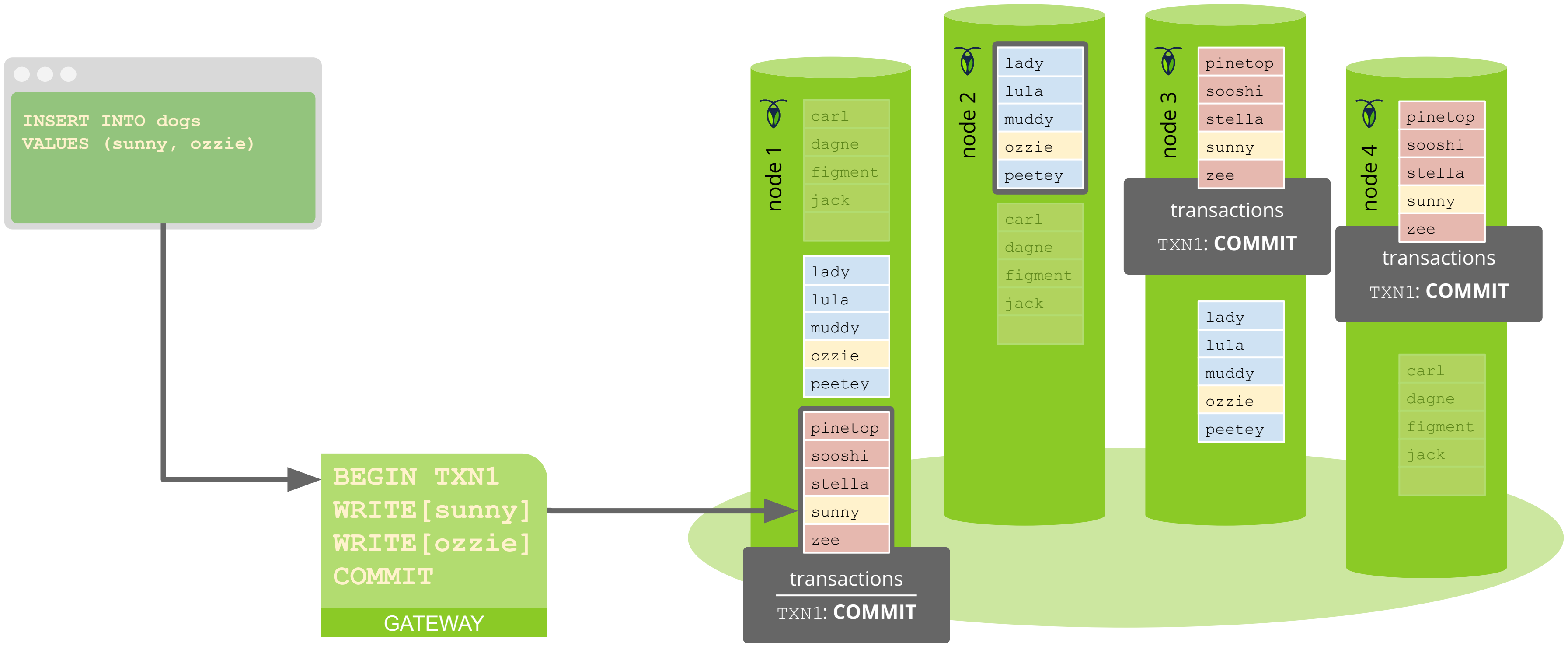
Distributed Transactions



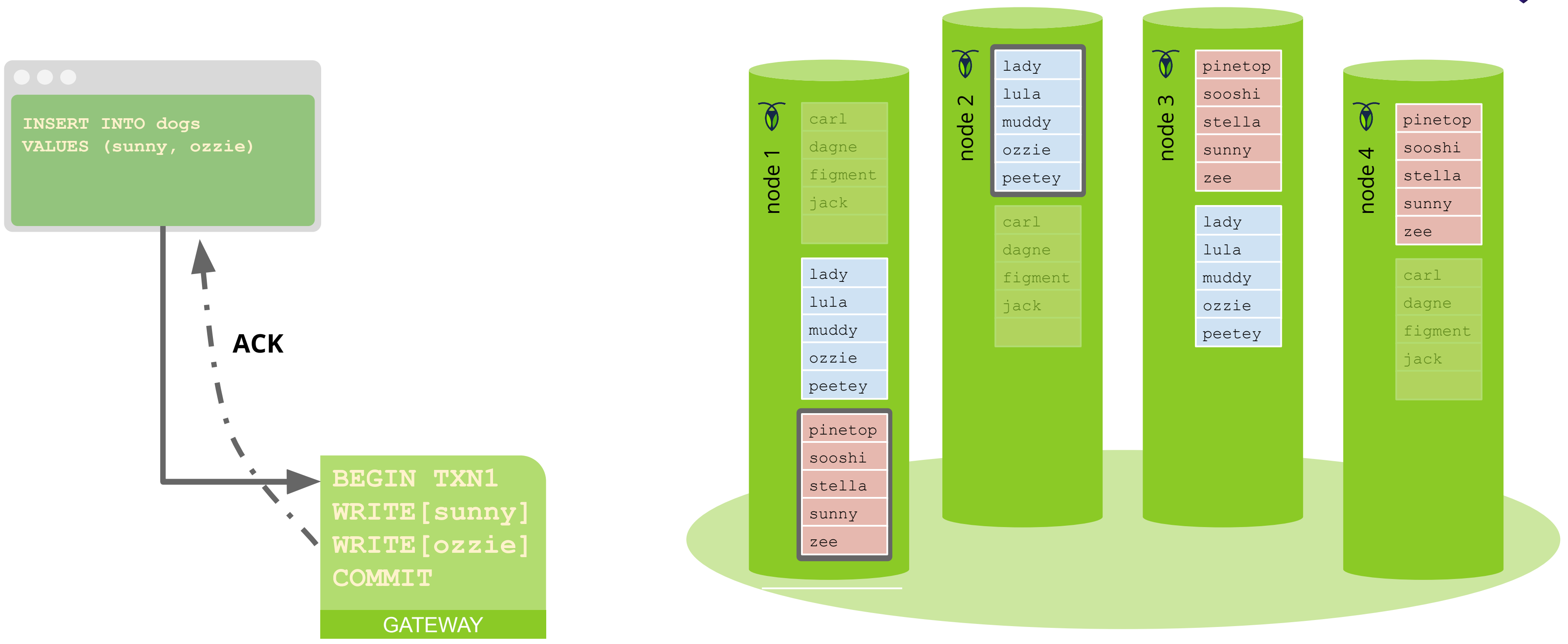
Distributed Transactions



Distributed Transactions



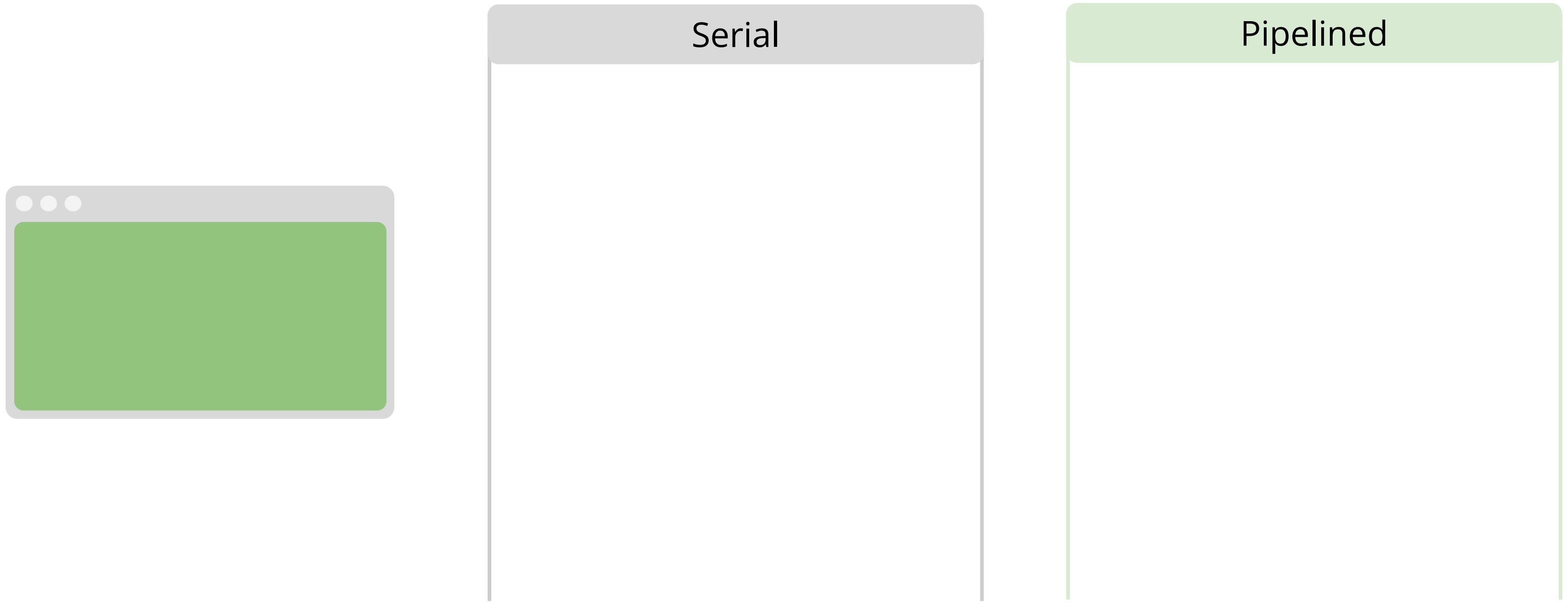
Distributed Transactions



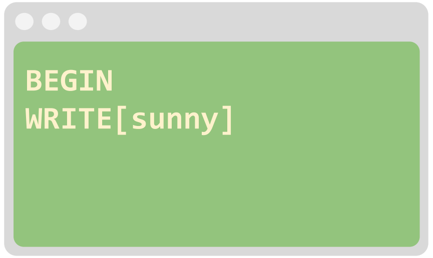
```
INSERT INTO dogs
VALUES (sunny, ozzie)
```

```
BEGIN TXN1
WRITE[sunny]
WRITE[ozzie]
COMMIT
```

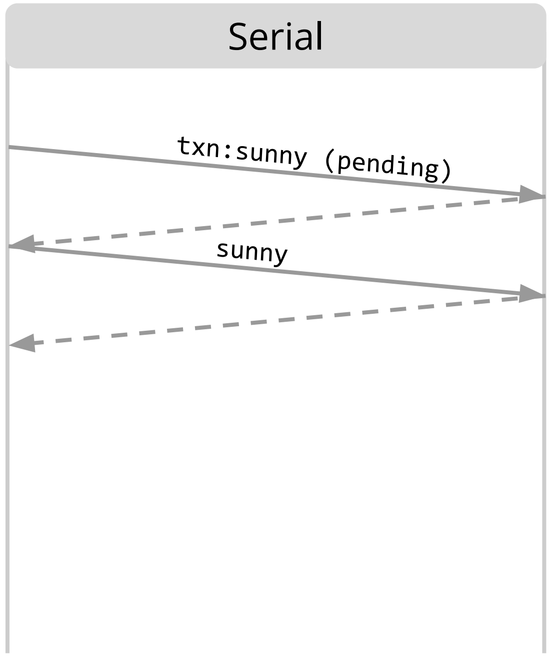
Transactions: Pipelining

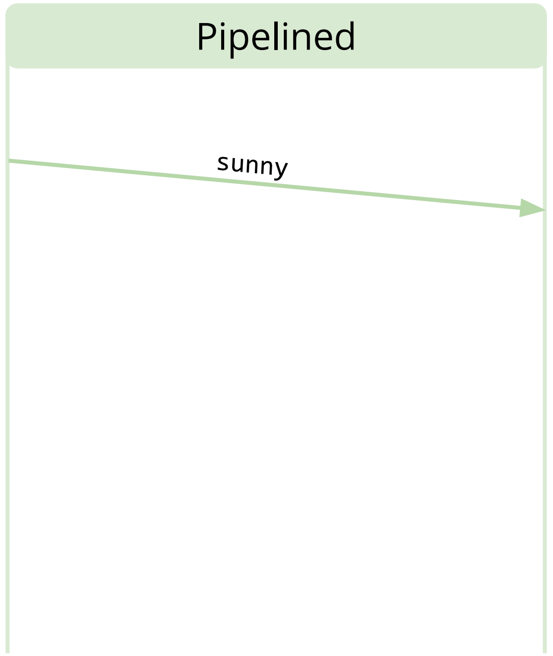


Transactions: Pipelining

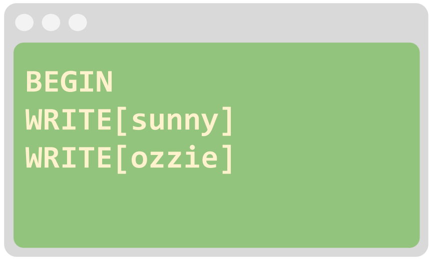


```
BEGIN
WRITE[sunny]
```

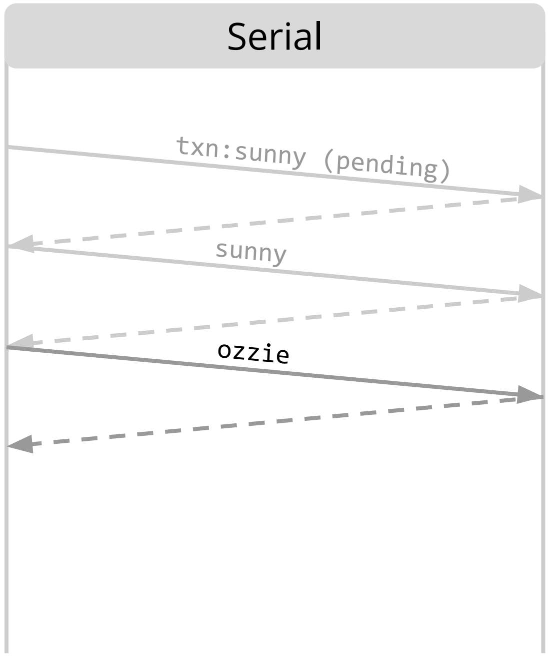


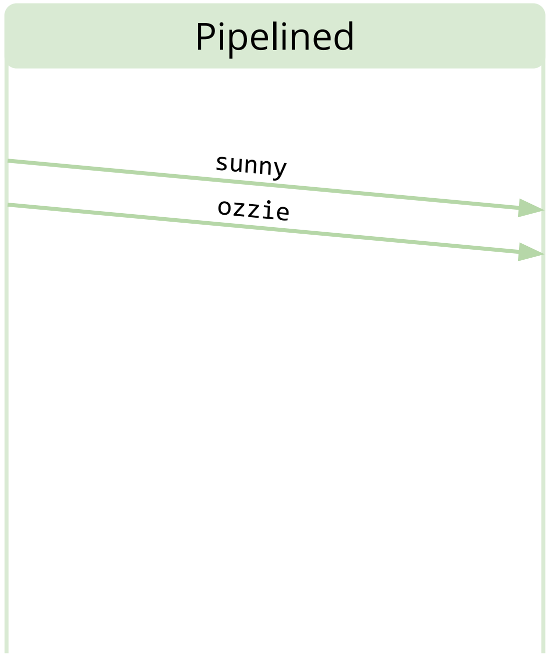


Transactions: Pipelining

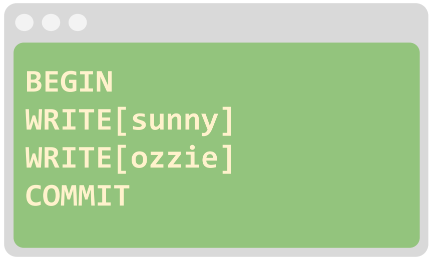


```
BEGIN
WRITE[sunny]
WRITE[ozzie]
```

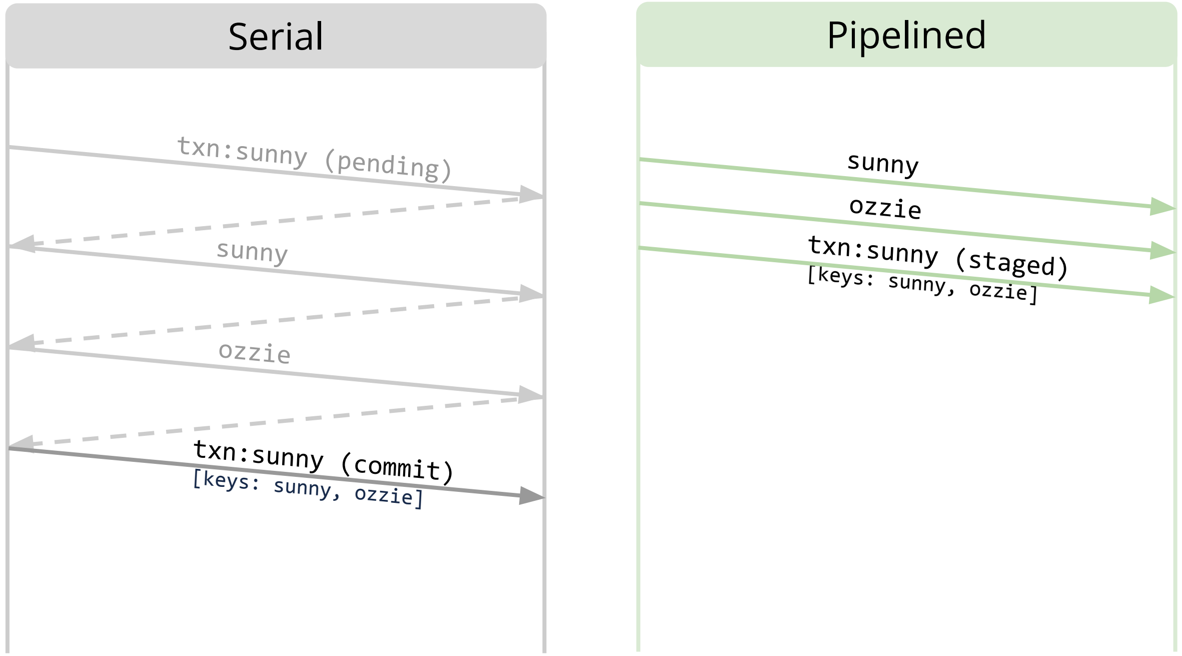




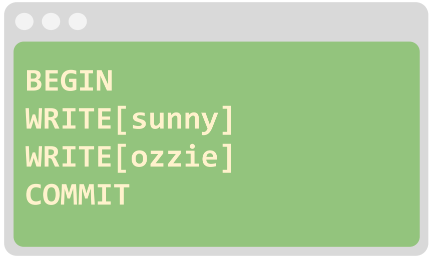
Transactions: Pipelining



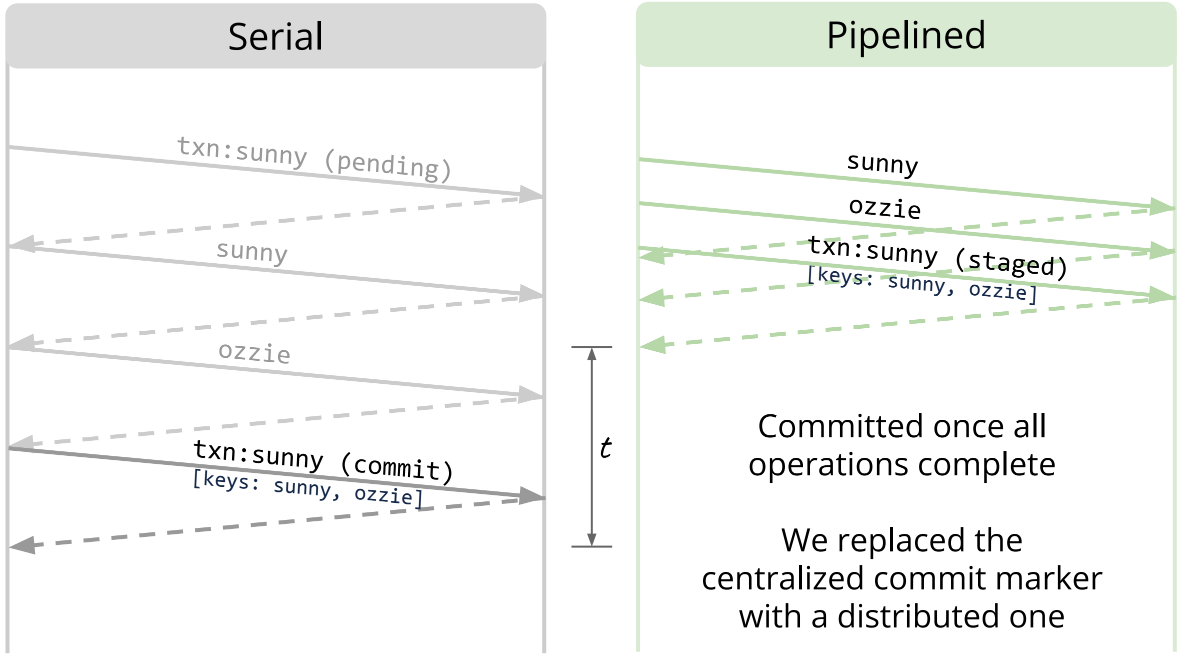
```
BEGIN
WRITE[sunny]
WRITE[ozzie]
COMMIT
```



Transactions: Pipelining



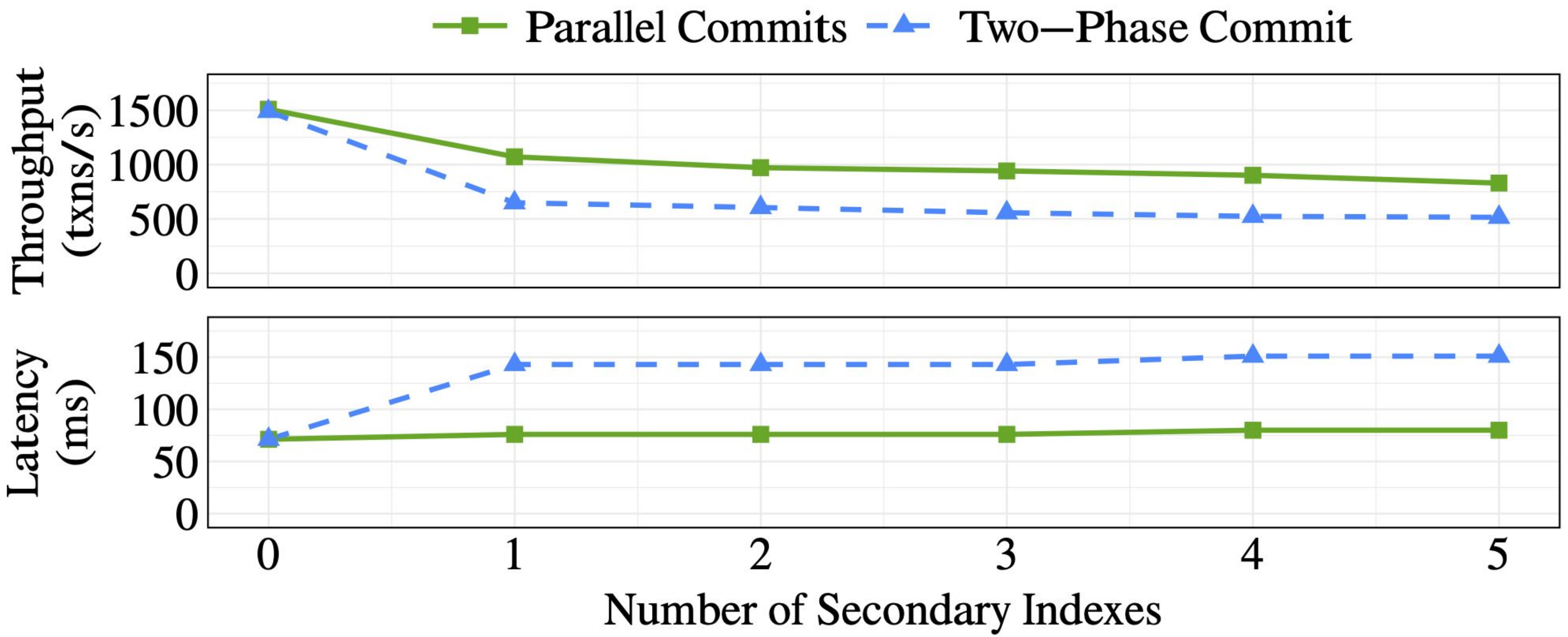
```
BEGIN
WRITE[sunny]
WRITE[ozzie]
COMMIT
```



Committed once all operations complete

We replaced the centralized commit marker with a distributed one

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



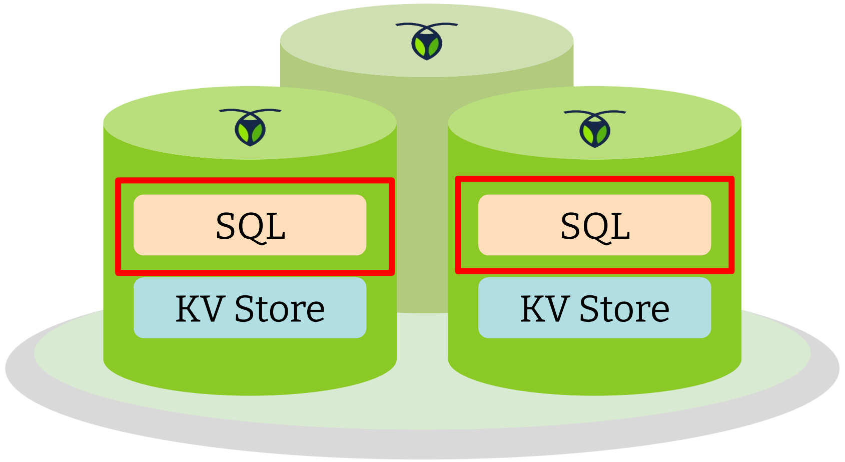
Agenda

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

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



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

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>/<Index>/1	“Bat”,1.11
/<Table>/<Index>/2	“Ball”,2.22
/<Table>/<Index>/3	“Glove”,3.33

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

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	∅

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	∅

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	∅

Agenda

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 Execution

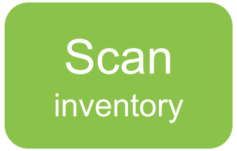
- Relational expressions have 0-2 input expressions
- Query plan is a tree of relational expressions
- SQL execution takes a query plan and runs the operations to completion

SQL Execution: Example

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

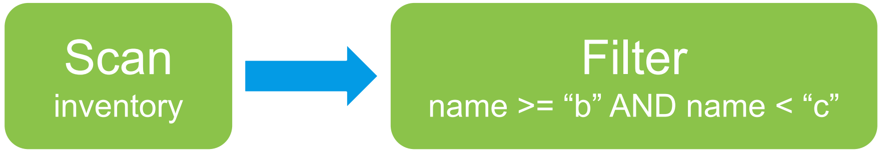
SQL Execution: Scan

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



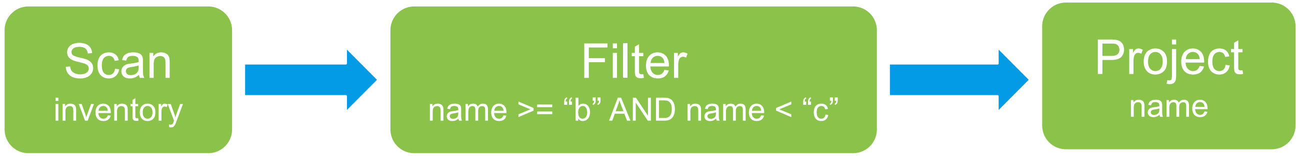
SQL Execution: Filter

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



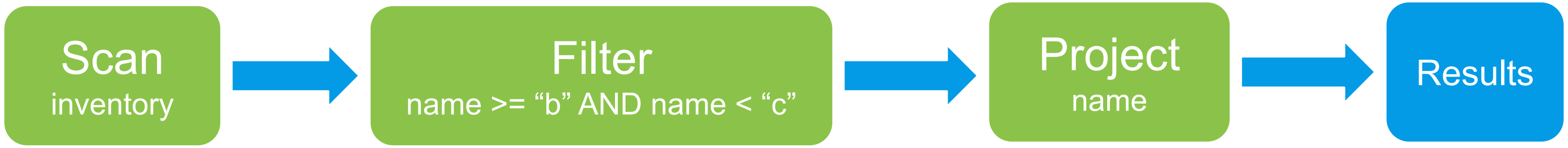
SQL Execution: Project

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



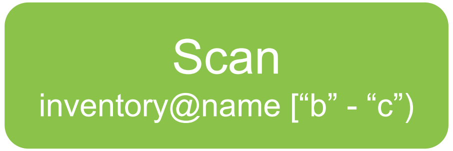
SQL Execution: Project

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



SQL Execution: Index Scans

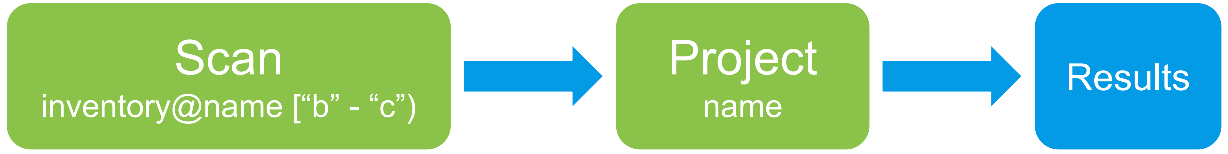
```
SELECT name
FROM   inventory
WHERE  name >= “b” AND name < “c”
```



The filter gets pushed into the scan

SQL Execution: Index Scans

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



Distributed SQL Execution

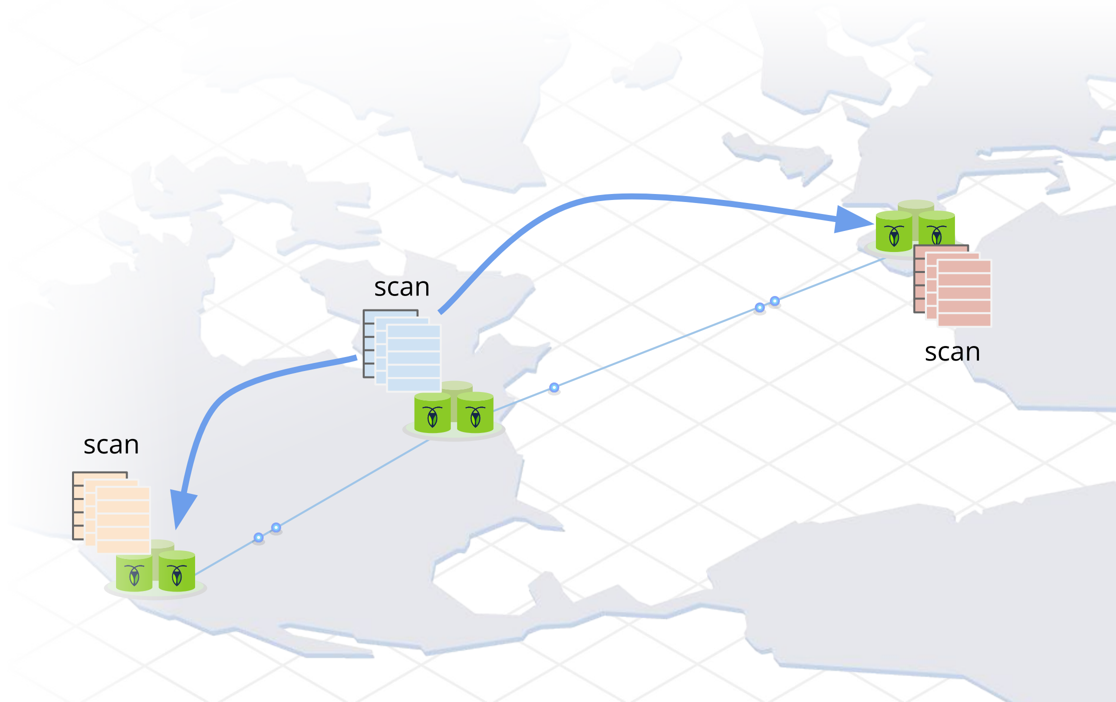
Network latencies and throughput are important considerations in geo-distributed setups

Push fragments of computation as close to the data as possible

Distributed SQL Execution: Streaming Group By

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

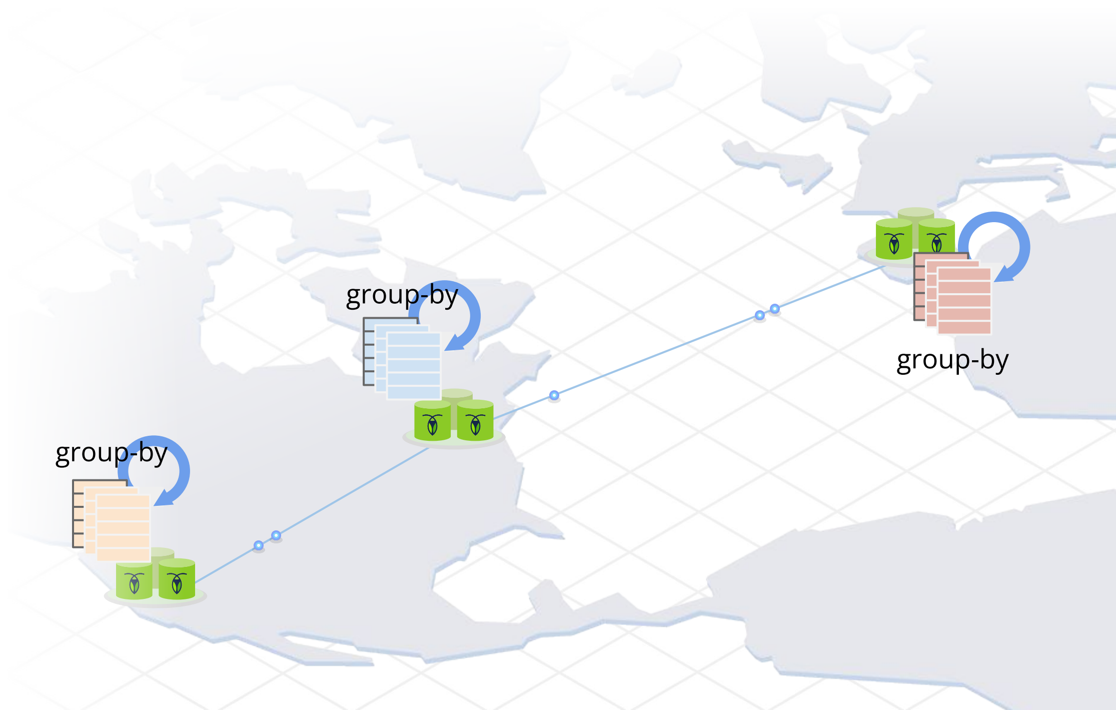
Scan customers | Scan customers | Scan customers



Distributed SQL Execution: Streaming Group By

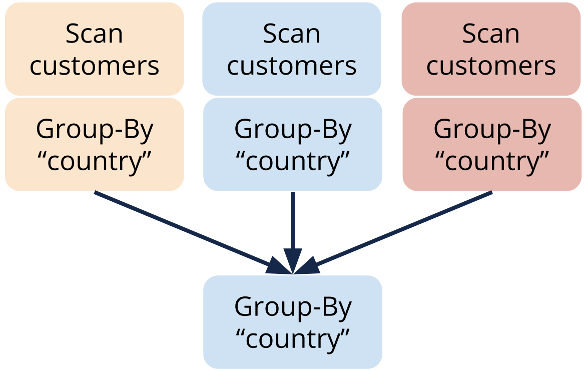
```
SELECT   COUNT(*), country
FROM     customers
GROUP BY country
```

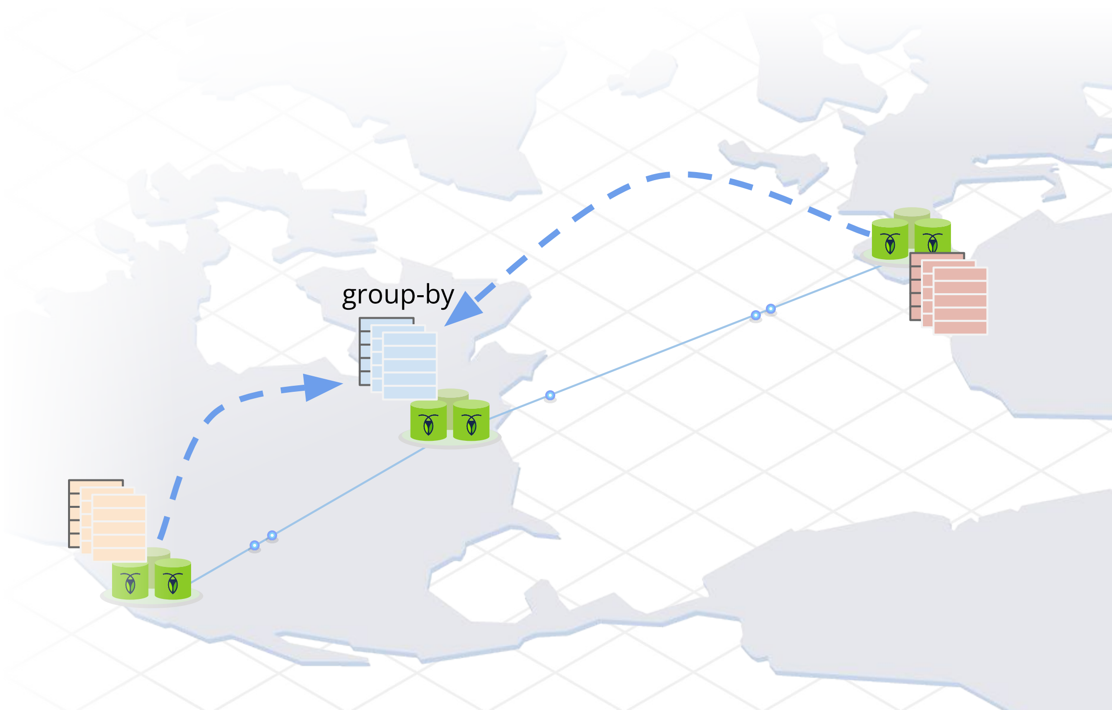
Scan customers	Scan customers	Scan customers
Group-By “country”	Group-By “country”	Group-By “country”



Distributed SQL Execution: Streaming Group By

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





Agenda

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

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

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

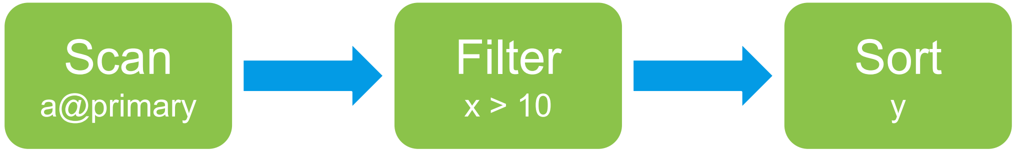
SQL Optimization: Cost-based Index Selection

```
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



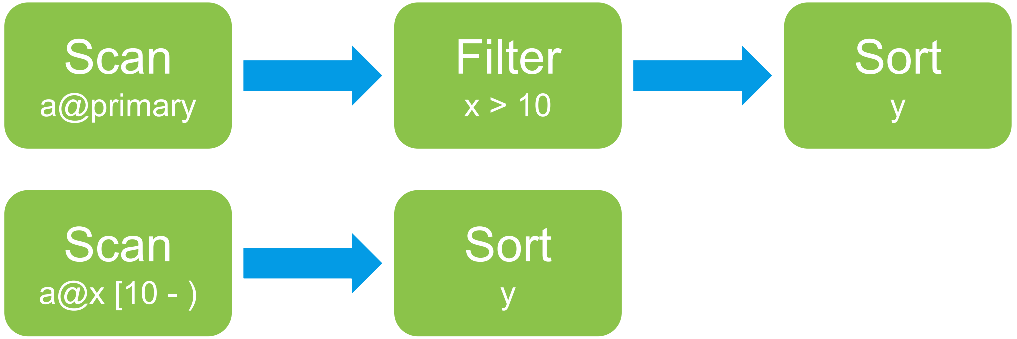
SQL Optimization: Cost-based Index Selection

```
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



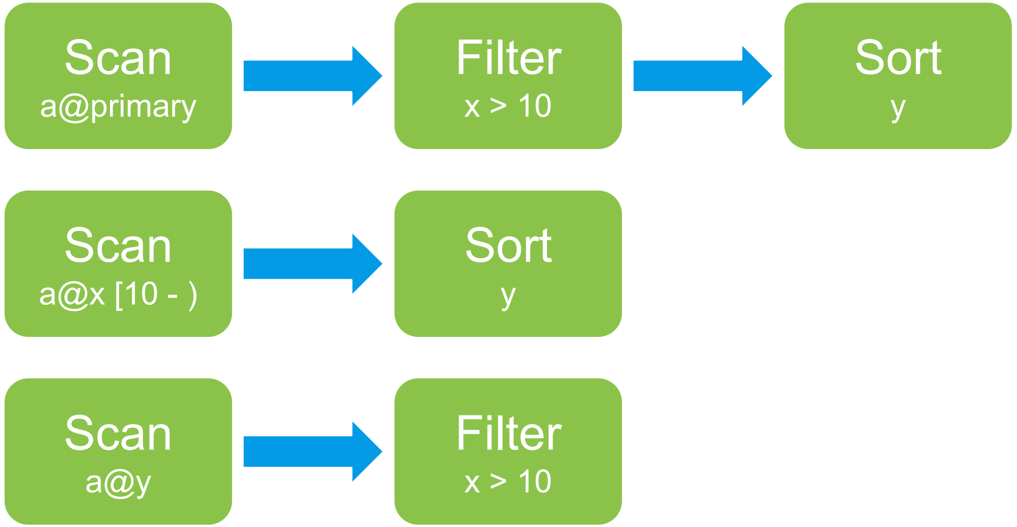
SQL Optimization: Cost-based Index Selection

```
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



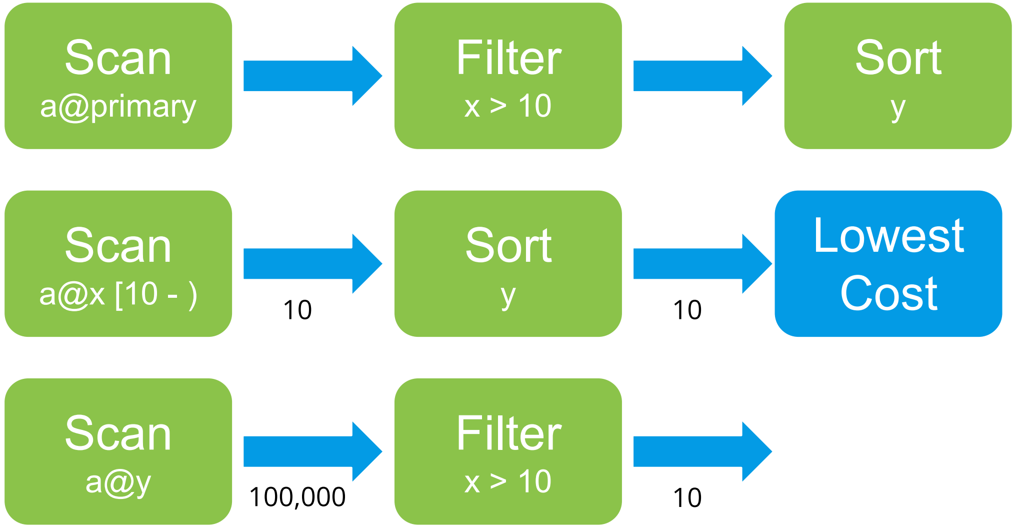
SQL Optimization: Cost-based Index Selection

```
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



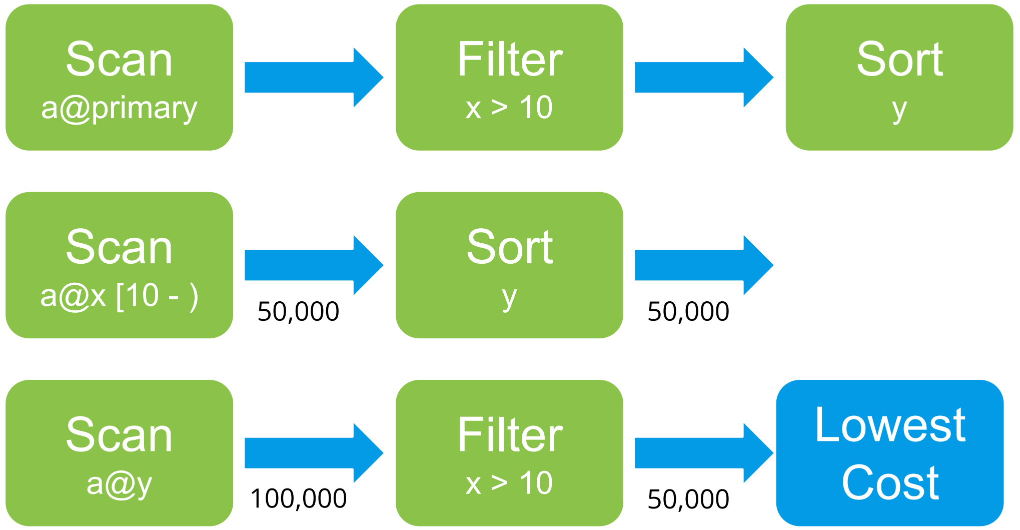
SQL Optimization: Cost-based Index Selection

```
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



Agenda

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

Locality-Aware SQL Optimization and Execution

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

Historically required expert users to shard and place data in specific regions.

Locality-Aware SQL Optimization and Execution

Database should be aware of regions, so users don't need to be.

New concept: Table Locality
`REGIONAL` or `GLOBAL`

Tables accessed from a single region or amenable to partitioning use locality `REGIONAL`

Read-mostly tables not amenable to partitioning use locality `GLOBAL`

Queries leverage data closest to them

Regional tables

REGIONAL BY TABLE v REGIONAL BY ROW

In REGIONAL BY ROW, data is partitioned by a hidden crdb_region column, which is set to the local region on insert.

Post-query uniqueness checks ensure that email remains unique.

```
CREATE TABLE users (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid()
    email STRING UNIQUE,
    name STRING
) LOCALITY REGIONAL BY ROW
```

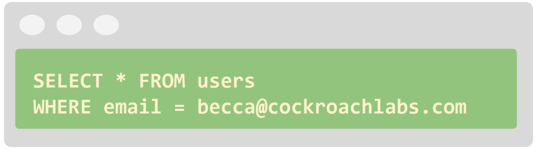
Inserting into a Regional by Row table

```
> EXPLAIN (OPT) INSERT INTO users (email, name)
  VALUES ('becca@cockroachlabs.com', 'Rebecca Taft');
                         info
-------------------------------------------------------
  insert users
   ├── values
   │    └── ('becca@cockroachlabs.com', 'Rebecca Taft',
   │         gen_random_uuid(), 'us-west1')
   └── unique-check: users(email)
        └── semi-join (lookup users@users_email_key)
             ├── with-scan &1
             └── filters
                  └── (id != users.id) OR
                      (crdb_region != users.crdb_region)
```

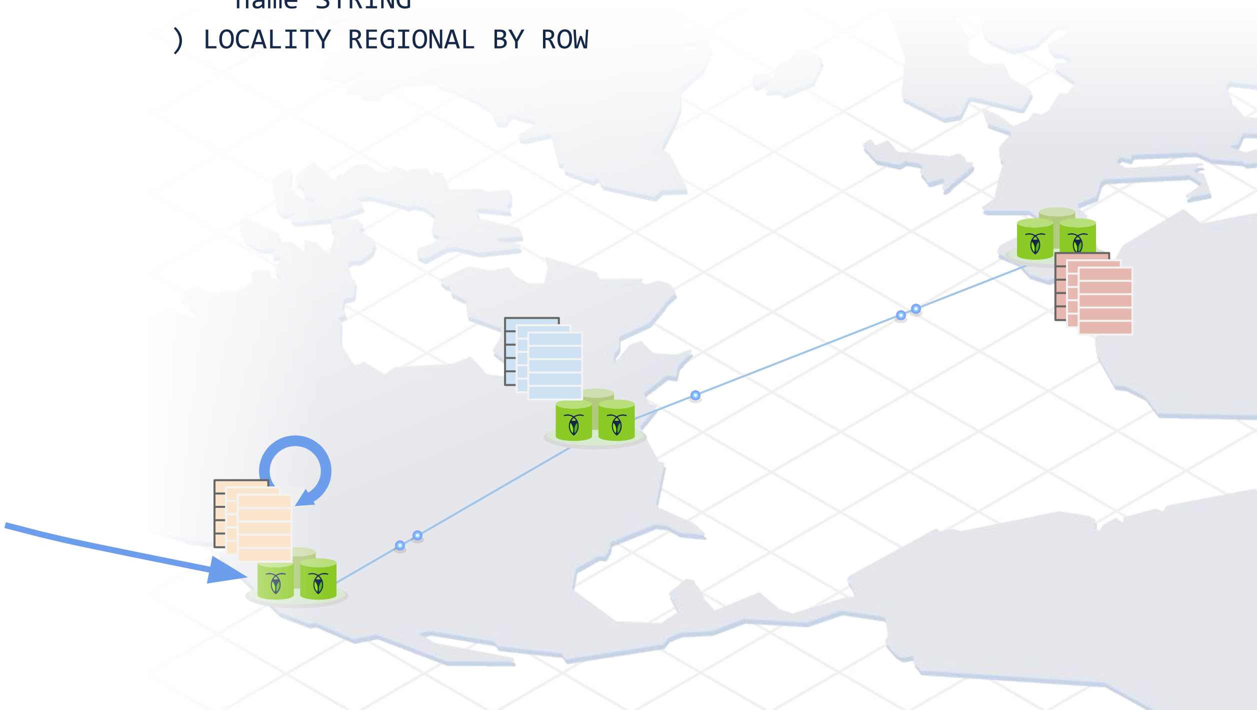
Reading from a Regional by Row table

Automatically checks the local region first before fanning out to remote regions.

```
CREATE TABLE users (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid()
    email STRING UNIQUE,
    name STRING
) LOCALITY REGIONAL BY ROW
```



```
SELECT * FROM users
WHERE email = becca@cockroachlabs.com
```



Reading from a Regional by Row table

```
> EXPLAIN (OPT) SELECT * FROM users
  WHERE email = 'becca@cockroachlabs.com';
                           info
------------------------------------------------------------
-
  index-join users
   └── locality-optimized-search
        ├── scan users@users_email_key
        │    └── [/'us-west1'/'becca@cockroachlabs.com']
        └── scan users@users_email_key
             ├── [/'europe-west1'/'becca@cockroachlabs.com']
             └── [/'us-east1'/'becca@cockroachlabs.com']
```

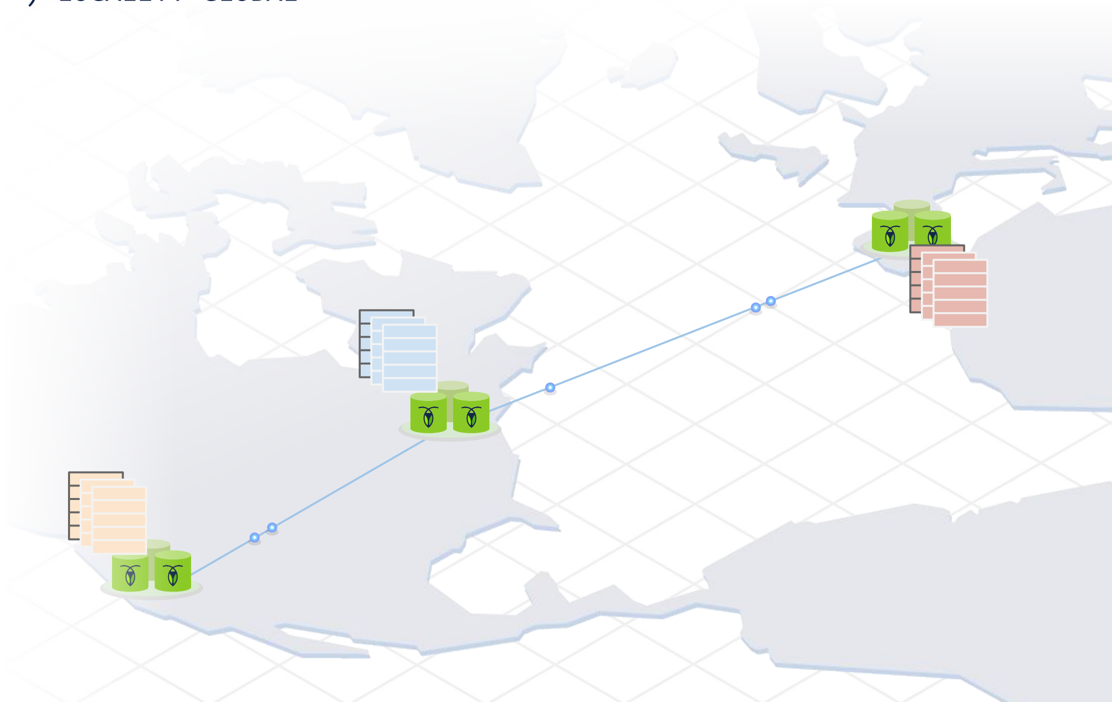
Global tables

Non-voting replicas which don't impact write latency

System automatically places a non-voting replica in regions without a voting replica

"Non-blocking" transactions cause writes to commit at a future timestamp and avoid blocking reads

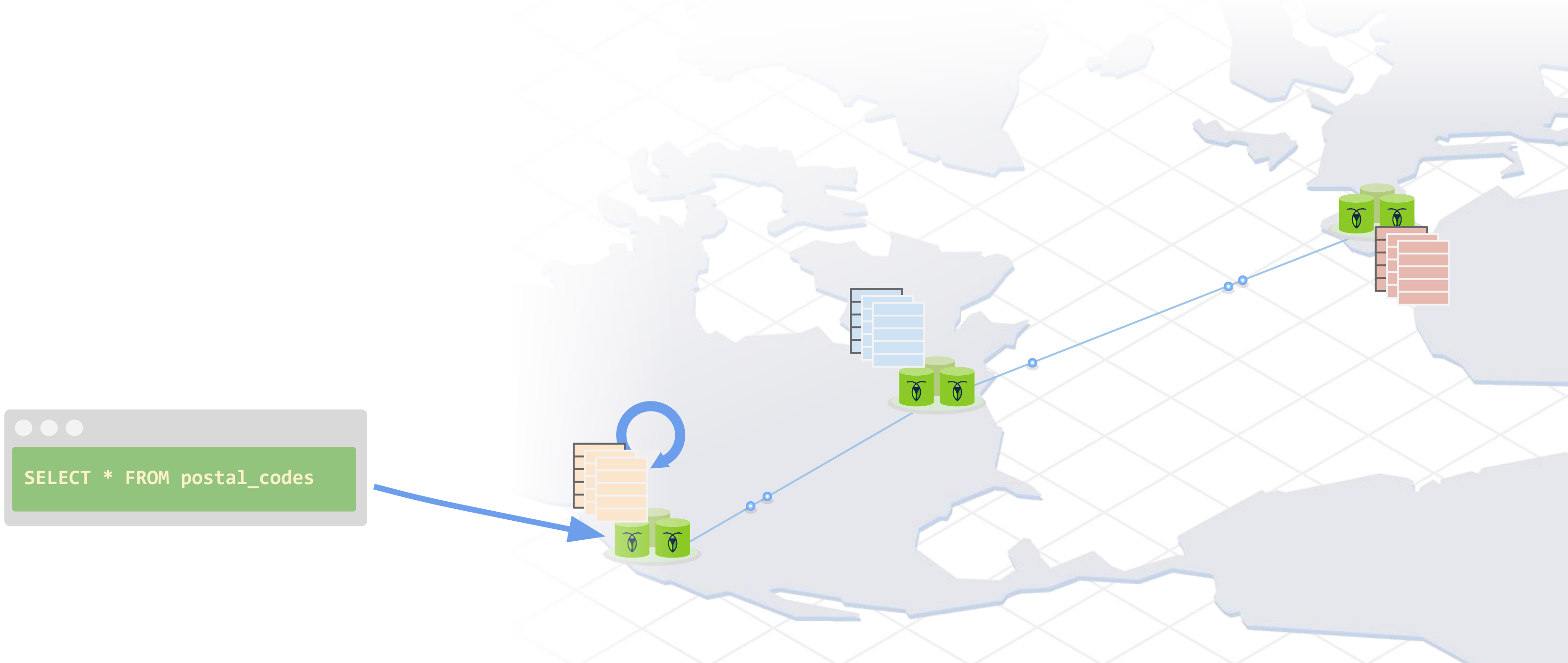
```
CREATE TABLE postal_codes (
    id INT PRIMARY KEY,
    code STRING
) LOCALITY GLOBAL
```



Local reads from Global tables

Automatically reads from replica (voting or non-voting) in the read's region

```
CREATE TABLE postal_codes (
    id INT PRIMARY KEY,
    code STRING
) LOCALITY GLOBAL
```

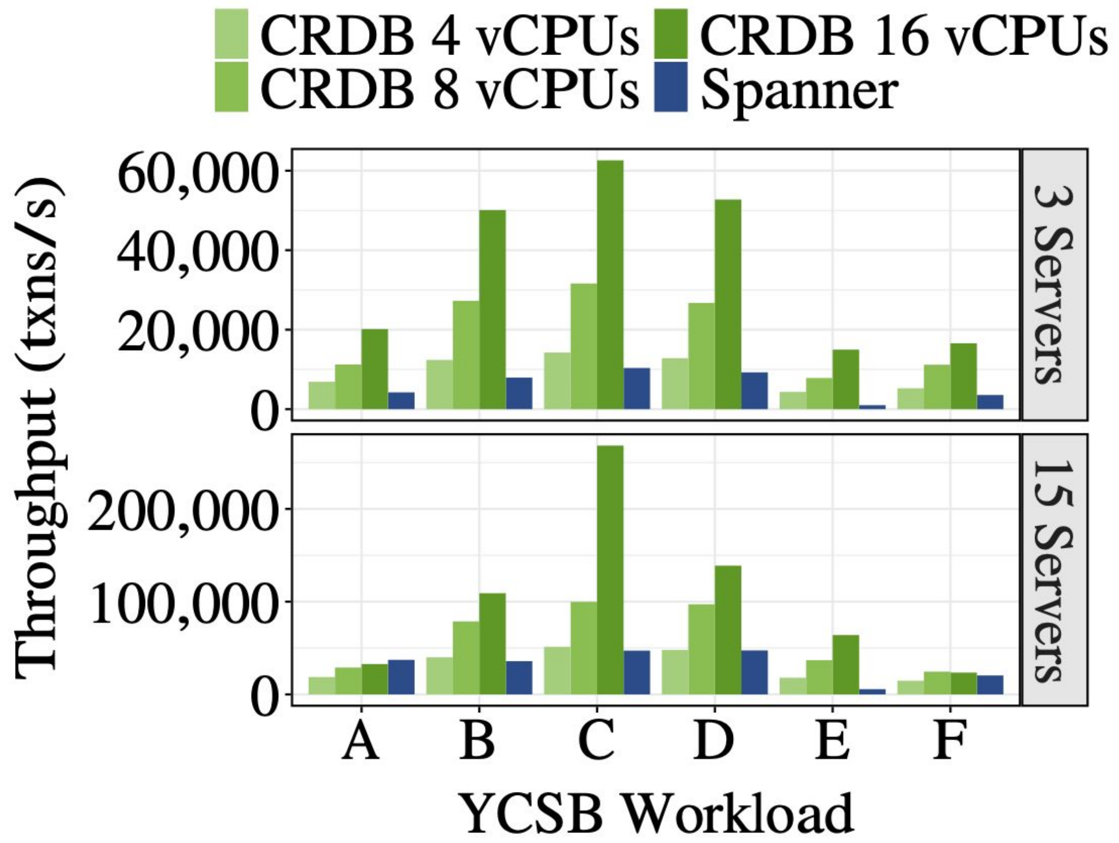


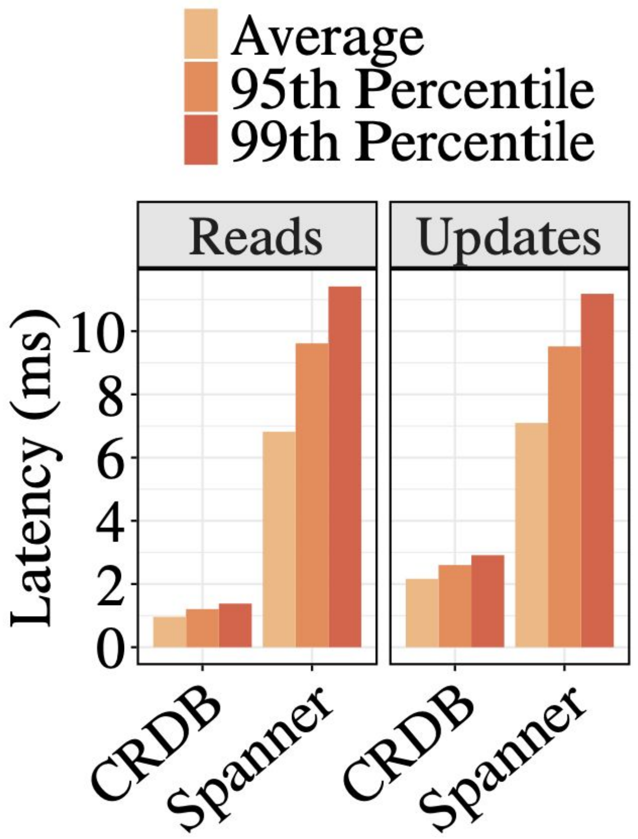
```
SELECT * FROM postal_codes
```

Agenda

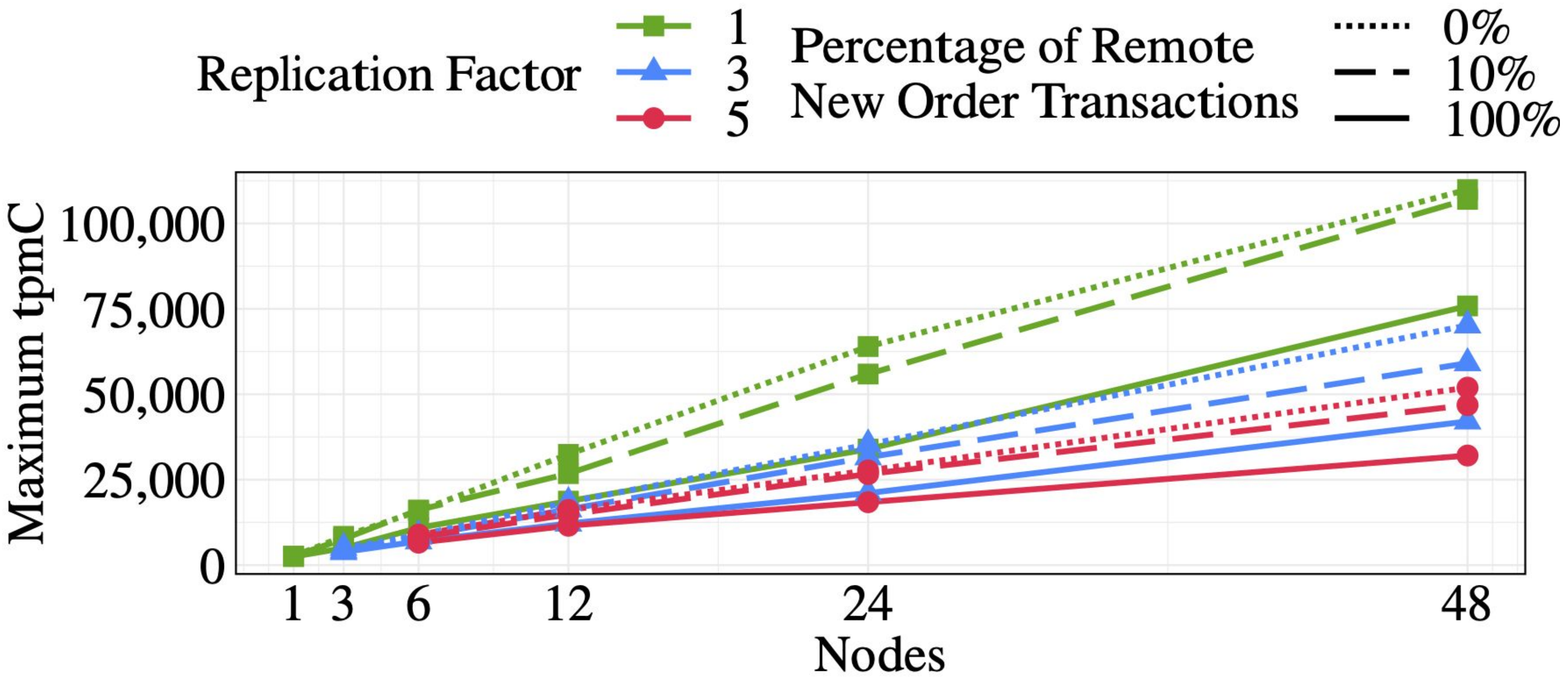
- Introduction
- Ranges and Replicas
- Transactions
- SQL Data in a KV World
- SQL Execution
- SQL Optimization
- Locality Awareness
- Evaluation

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

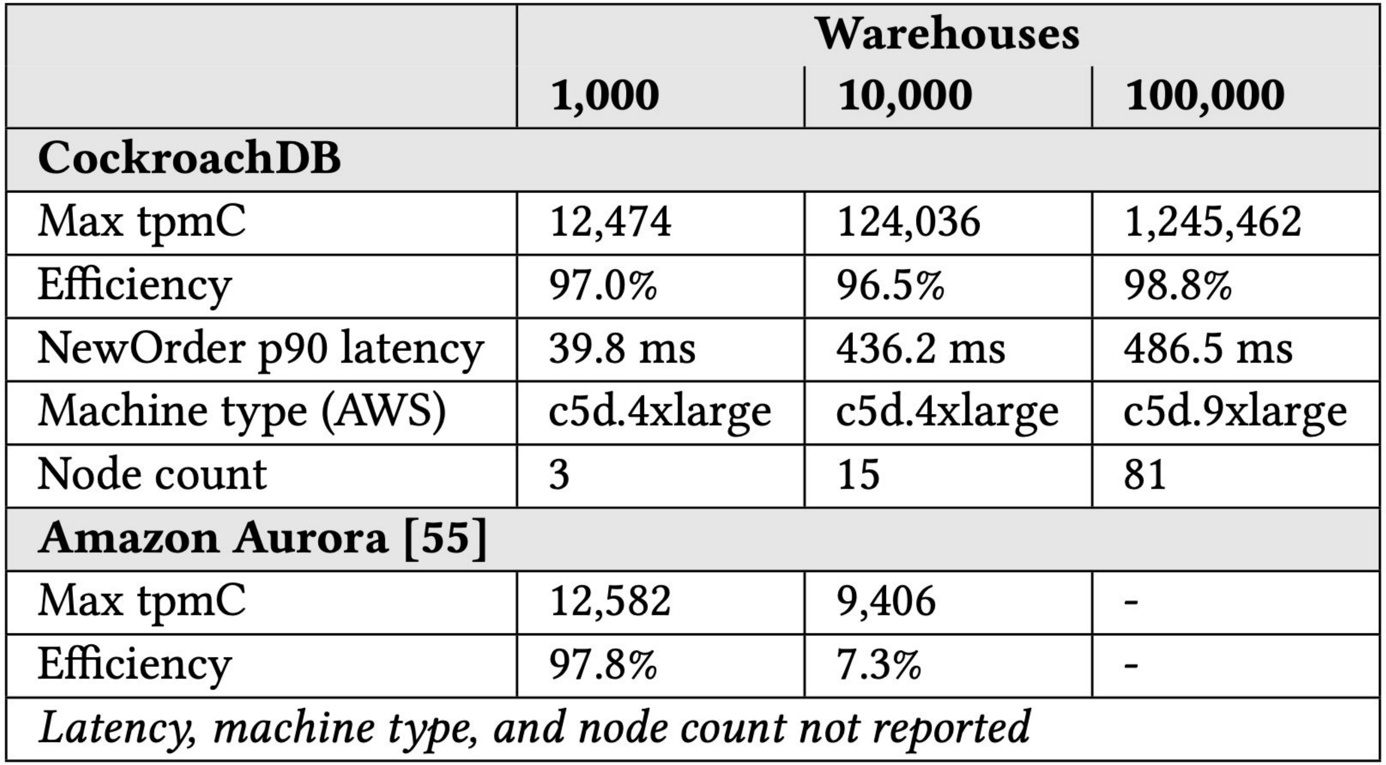
Thank You

We are hiring! www.cockroachlabs.com/careers

github.com/cockroachdb/cockroach

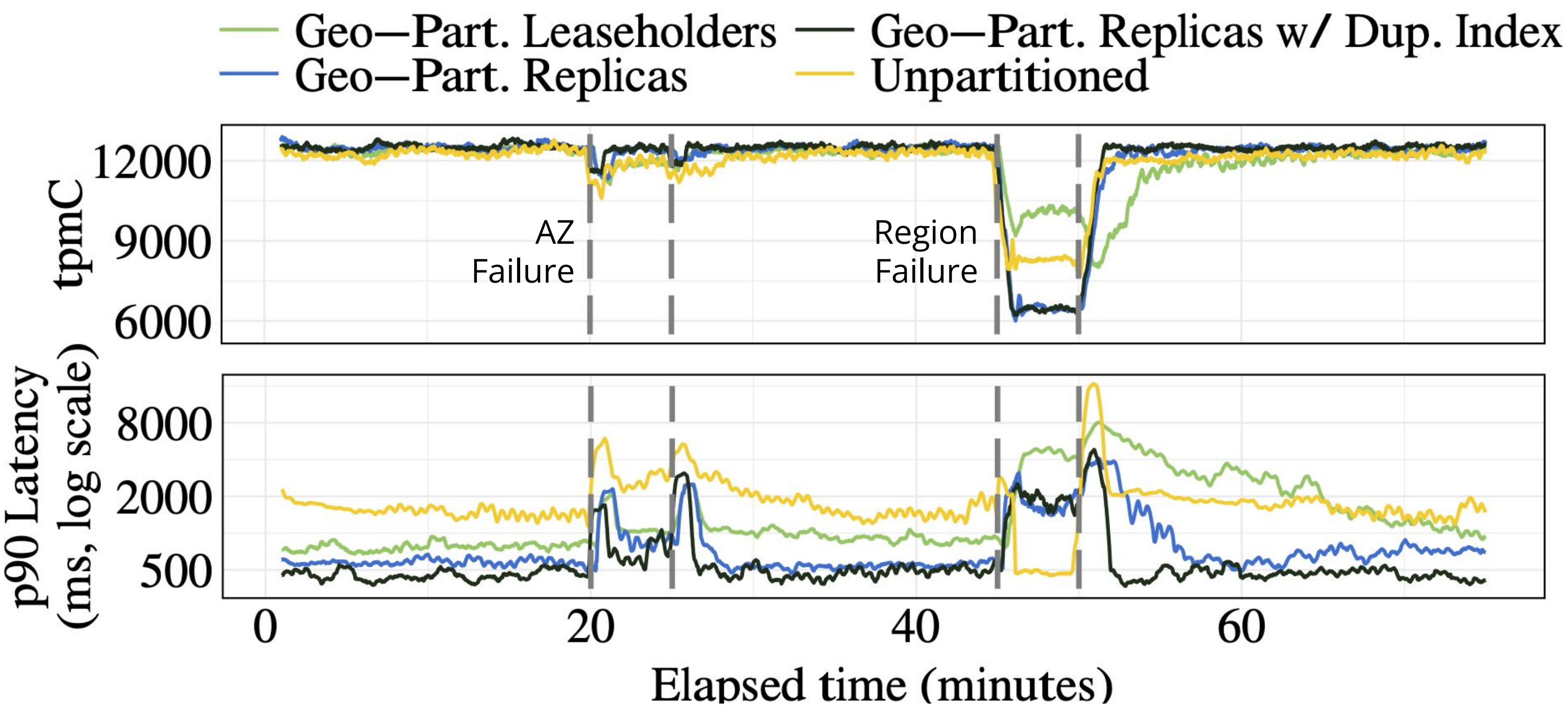
becca@cockroachlabs.com

Comparison with Amazon Aurora on TPC-C

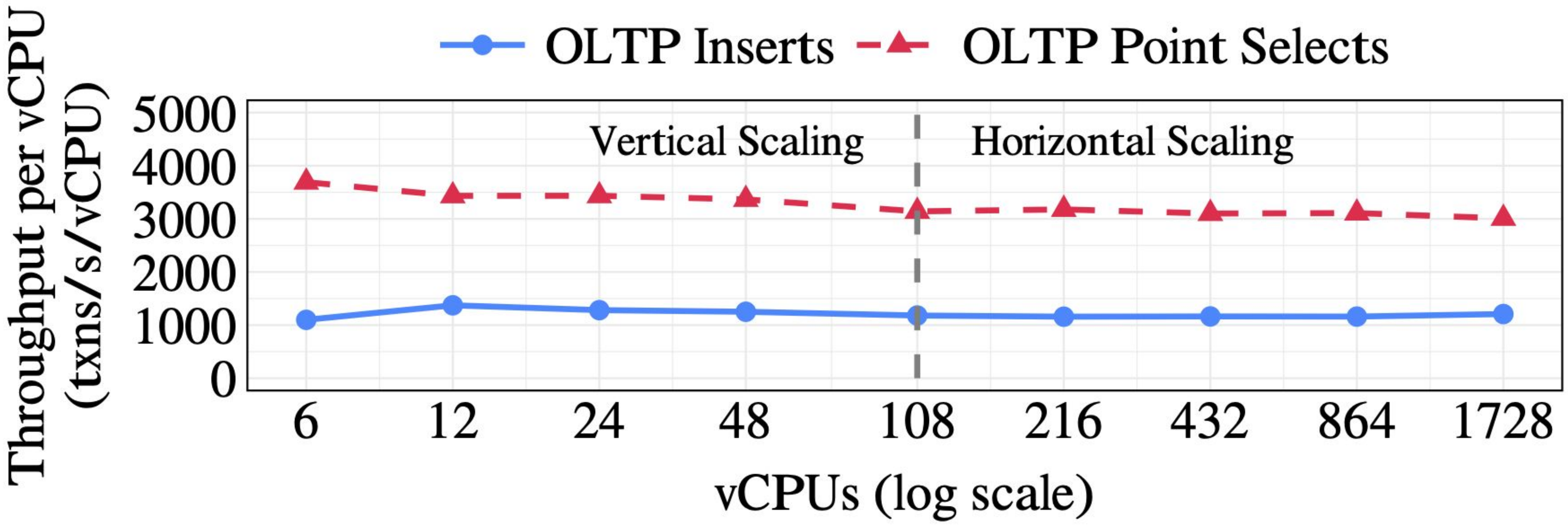


	Warehouses		
	1,000	**10,000**	**100,000**
CockroachDB			
Max tpmC	12,474	124,036	1,245,462
Efficiency	97.0%	96.5%	98.8%
NewOrder p90 latency	39.8 ms	436.2 ms	486.5 ms
Machine type (AWS)	c5d.4xlarge	c5d.4xlarge	c5d.9xlarge
Node count	3	15	81
Amazon Aurora [55]			
Max tpmC	12,582	9,406	-
Efficiency	97.8%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

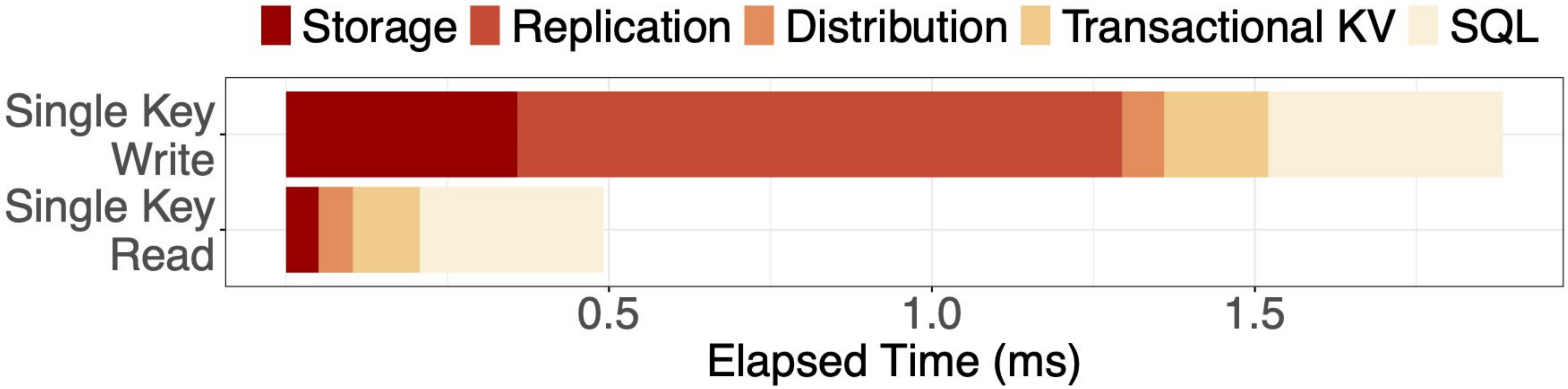
Multi-Region TPC-C Performance with AZ and Region Failure



Scalability on sysbench



Overheads of CRDB Layers


Storage
Replication
Distribution
Transactional KV
SQL
Single Key Write
Single Key Read
Elapsed Time (ms)