CockroachDB

Architecture of a Geo-Distributed SQL Database

Peter Mattis (@petermattis), Co-founder & CTO

Cockroach LABS

CockroachDB: Geo-distributed SQL Database

Make Data Easy

- Distributed
  - Horizontally scalable to grow with your application
- Geo-distributed
  - Handle datacenter failures
  - Place data near usage
  - Push computation near data
- SQL
  - Lingua-franca for rich data storage
  - Schemas, indexes, and transactions make app development easier

AGENDA

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

Distributed, Replicated, Transactional KV*

- Keys and values are strings
  - Lexicographically ordered by key
- Multi-version concurrency control (MVCC)
  - Values are never updated “in place”, newer versions shadow older versions
  - Tombstones are used to delete values
  - Provides snapshot to each transaction
- Monolithic key-space

* Not exposed for external usage

Monolithic Key Space

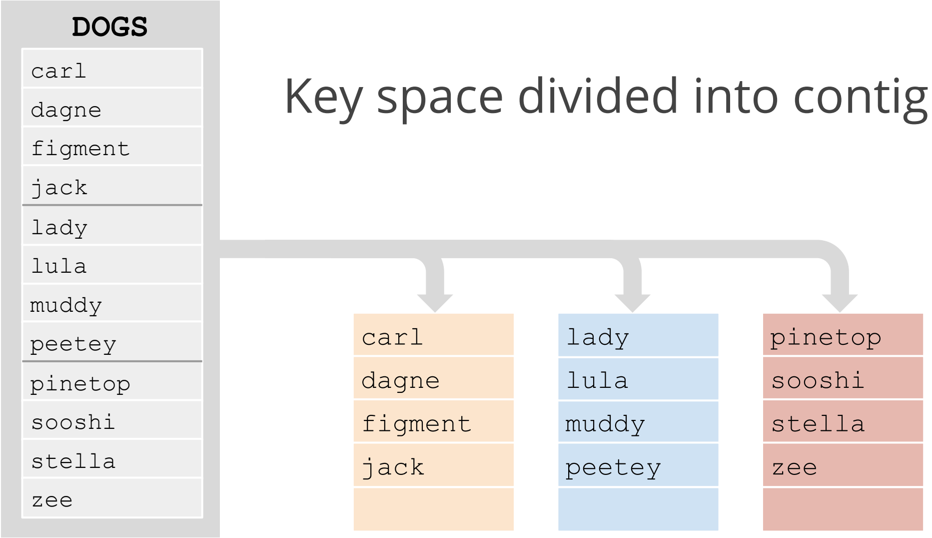
DOGS
carl
dagne
figment
jack
lady
lula
muddy
peetey
pinetop
sooshi
stella
zee

Monolithic logical key space

- Ordered lexicographically by key

Ranges

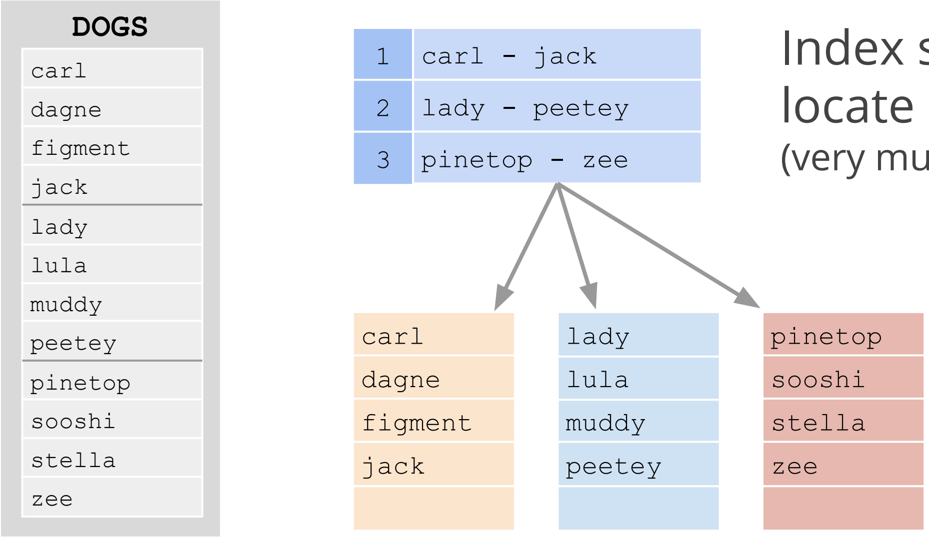
Key space divided into contiguous ~64MB ranges



Ranges are small enough to be moved/split quickly

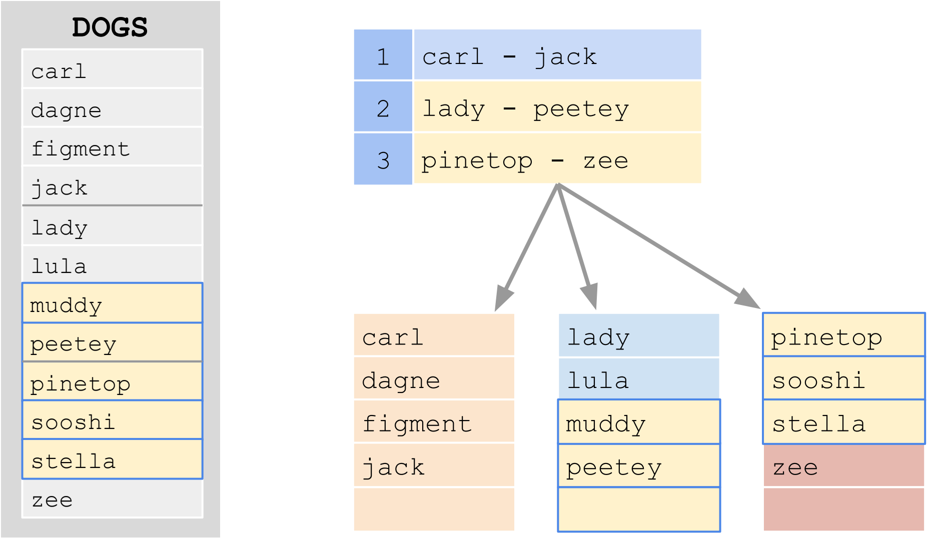
Ranges are large enough to amortize indexing overhead

Range Indexing



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

Ordered Range Scans

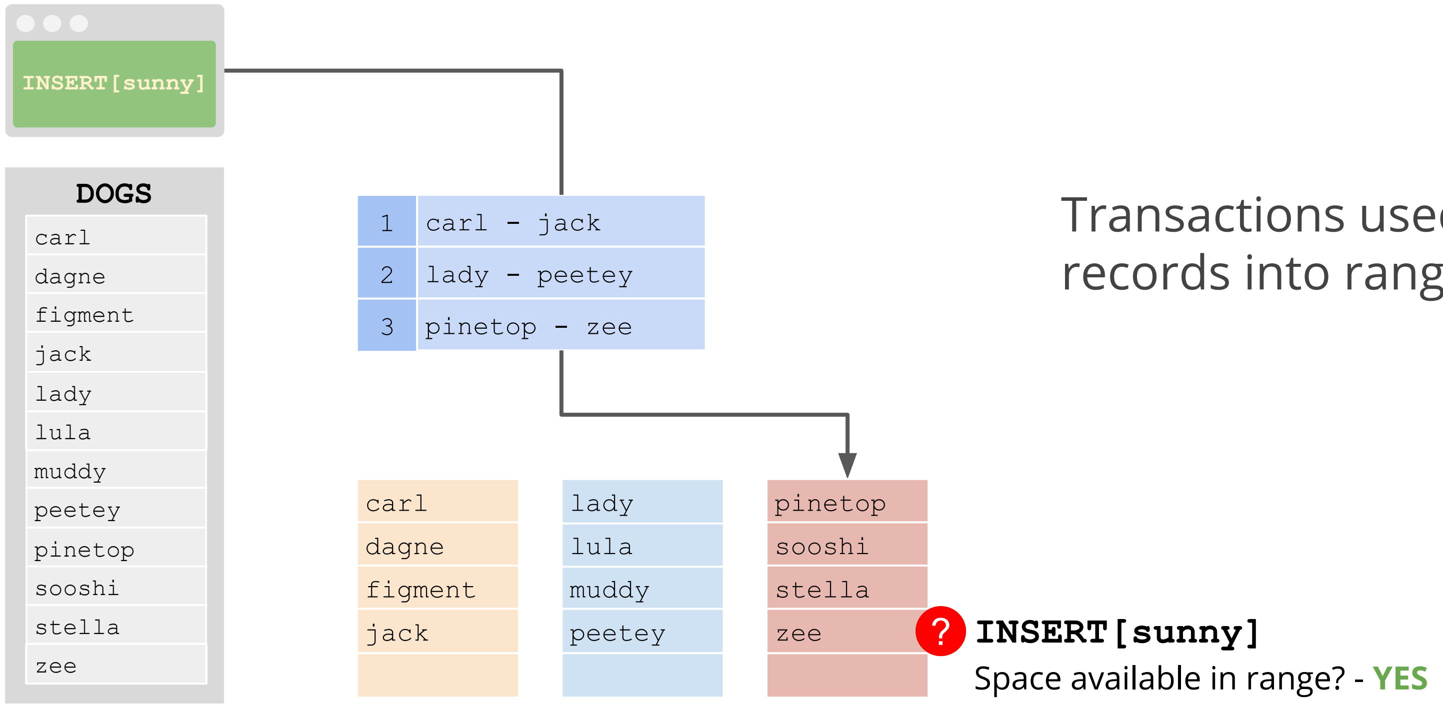


Ordered keys enable efficient range scans

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

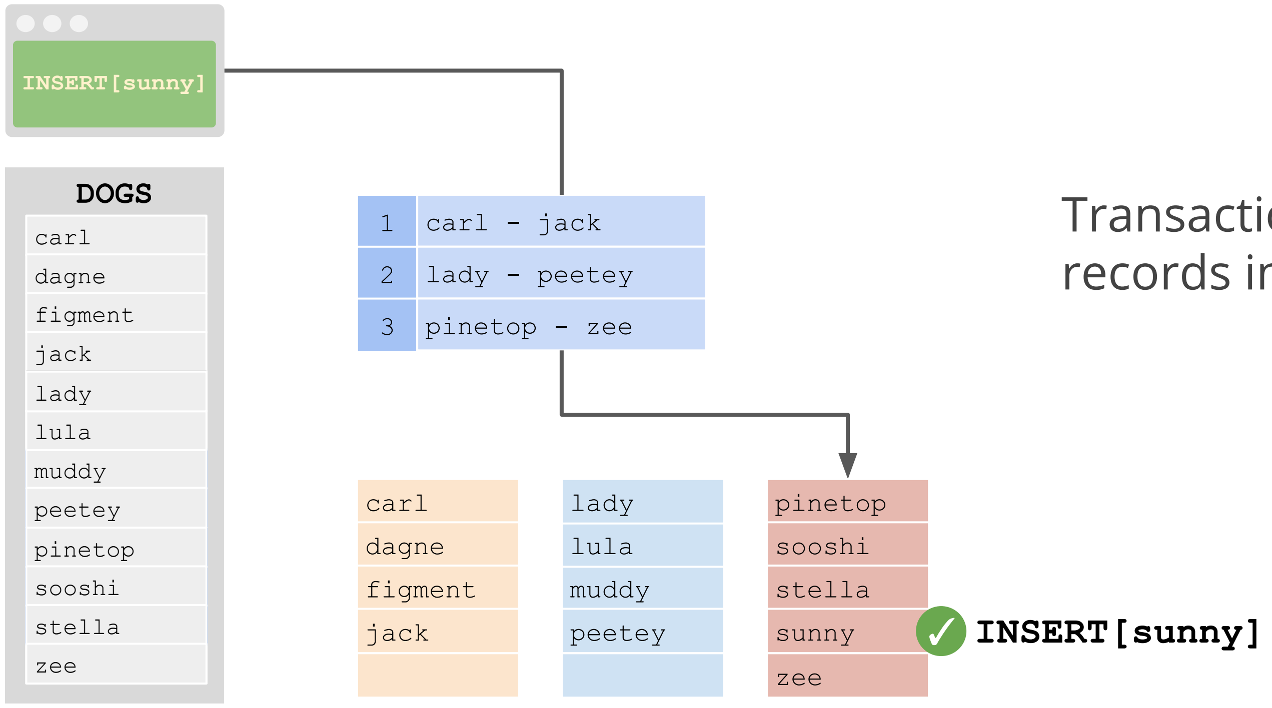
Cockroach LABS

Transactional Updates



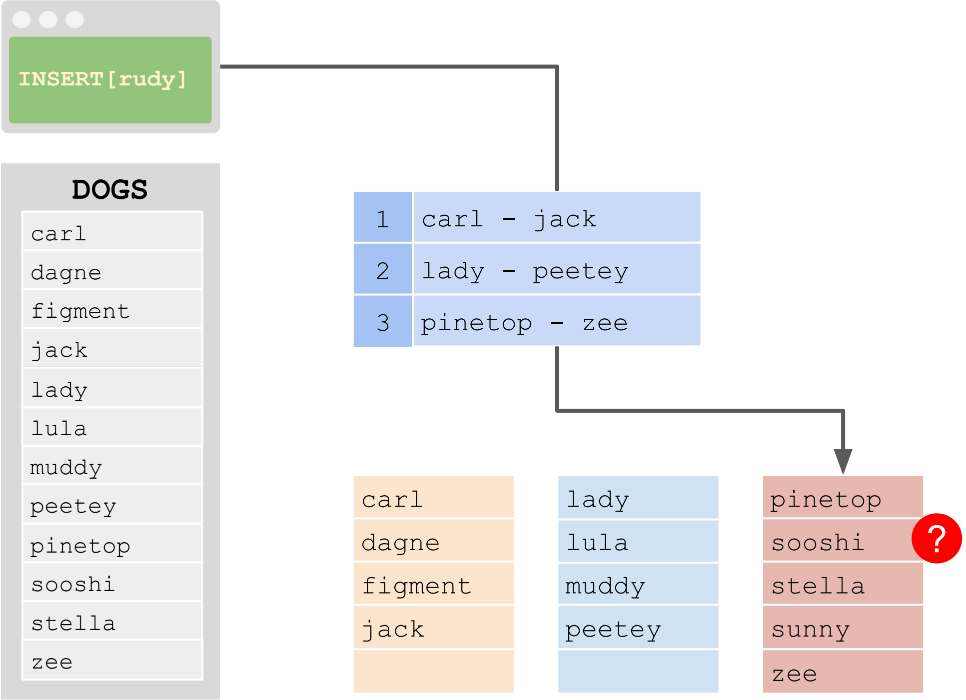
Transactions used to insert records into ranges

Transactional Updates



Transactions used to insert records into ranges

Range Splits



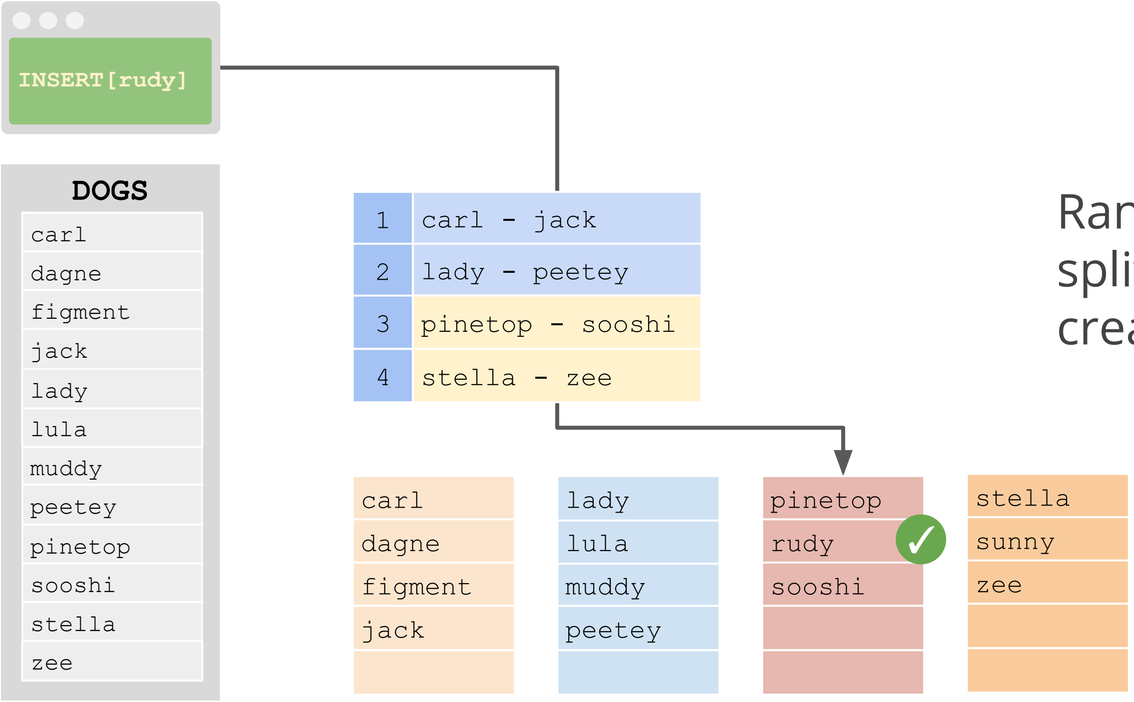
INSERT[rudy]

Space available in range? - **NO**

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

Cockroach LABS

Range Splits



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

INSERT [rudy]

split range and insert

Cockroach LABS

Raft and Replication

Ranges (~64MB) are the unit of replication

Each range is a Raft group
(Raft is a consensus replication protocol)

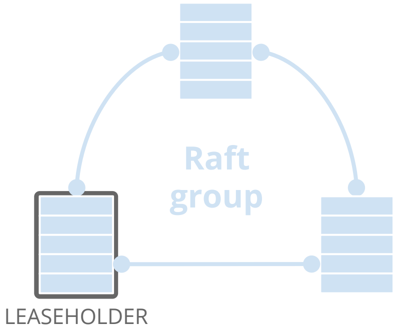
Default to 3 replicas, though this is configurable

- Important system ranges default to 5 replicas
- Note: 2 replicas doesn't make sense in consensus replication

Raft and Replication

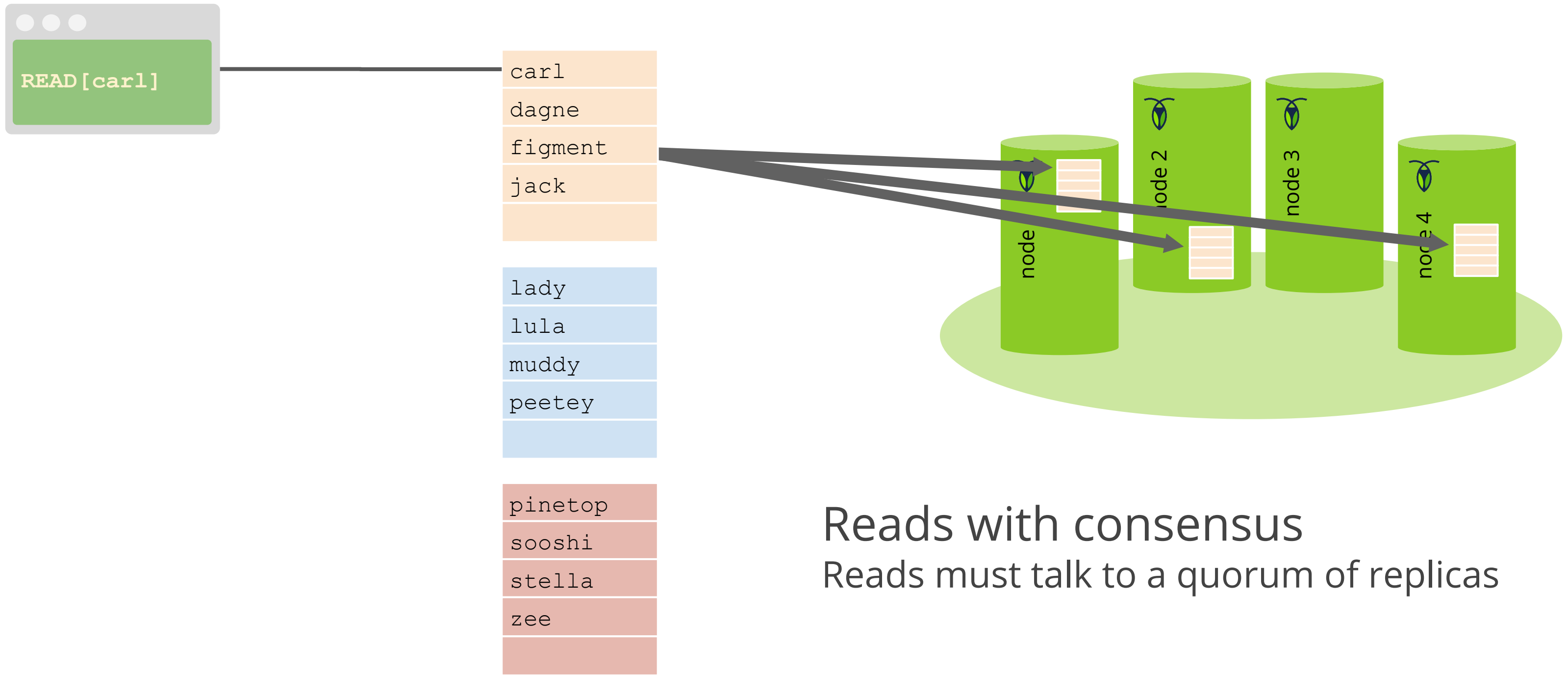
Raft provides “atomic replication” of commands

Commands are proposed by the leaseholder replica and distributed to the follower replicas, but only accepted when a quorum of replicas have acknowledged receipt



* Leaseholder == Raft leader

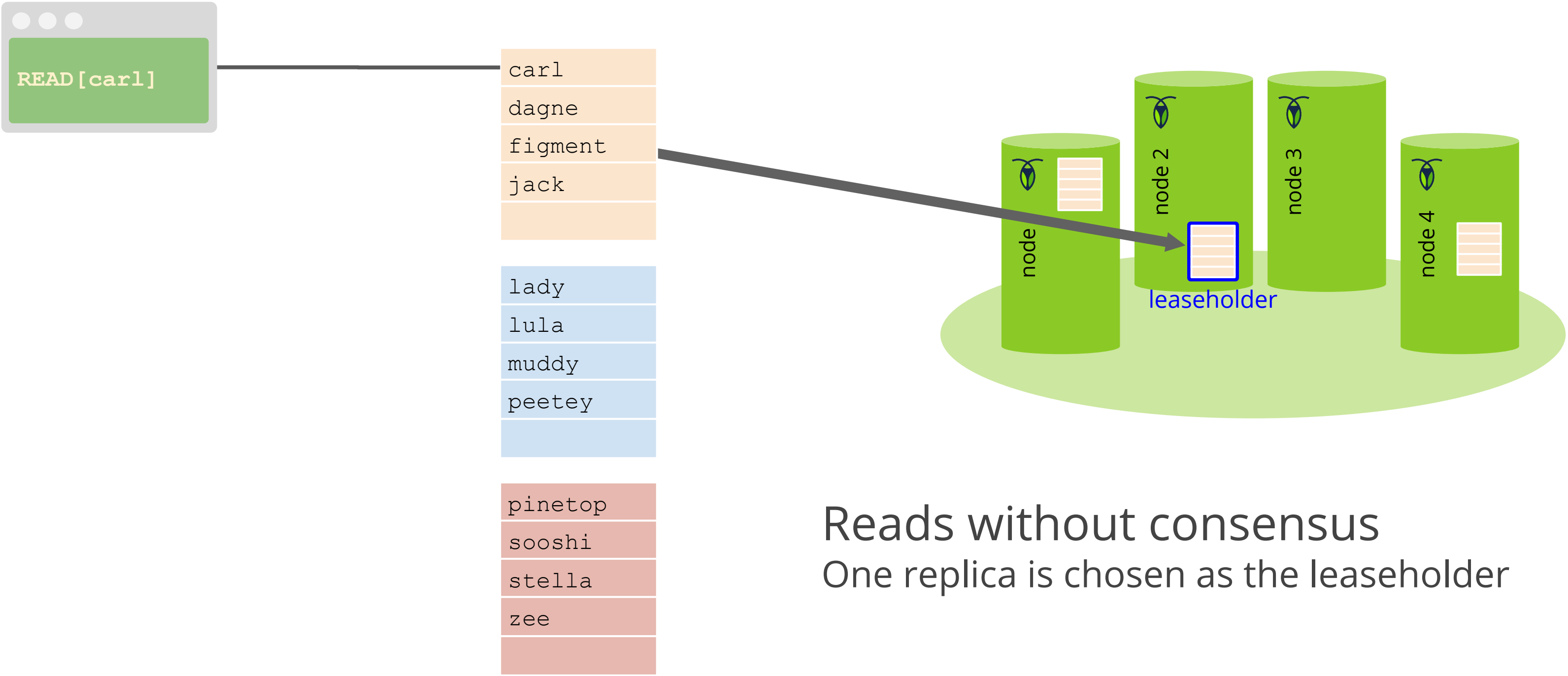
Range Leases



Reads with consensus

Reads must talk to a quorum of replicas

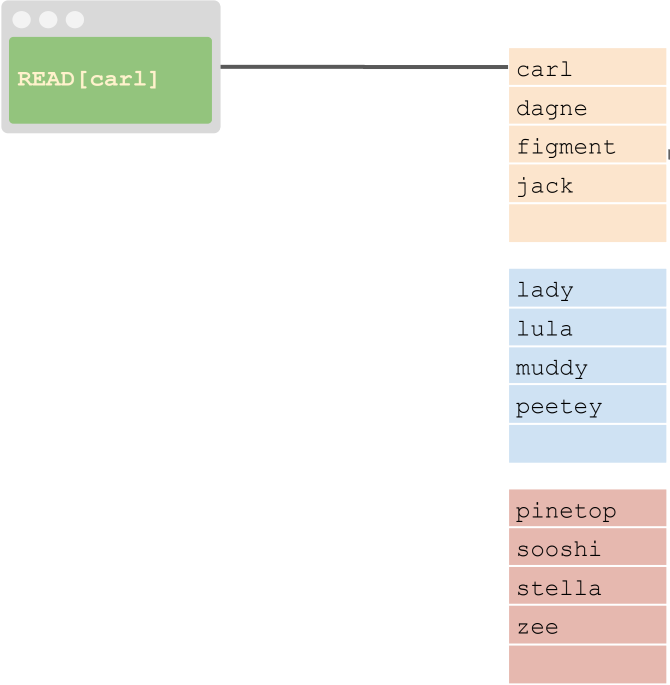
Range Leases

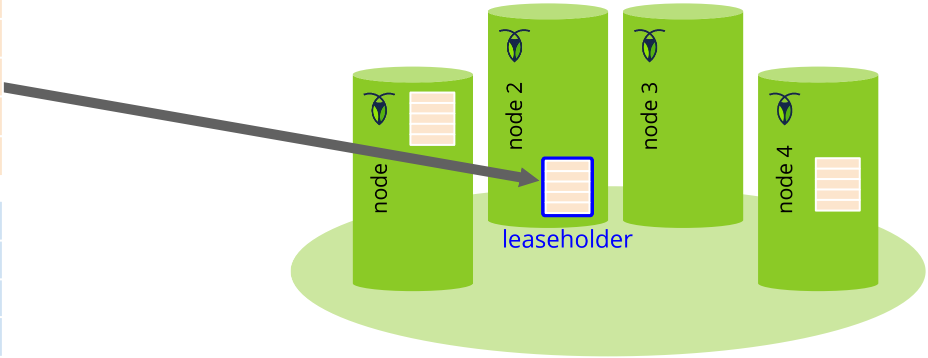


Reads without consensus

One replica is chosen as the leaseholder

Range Leases





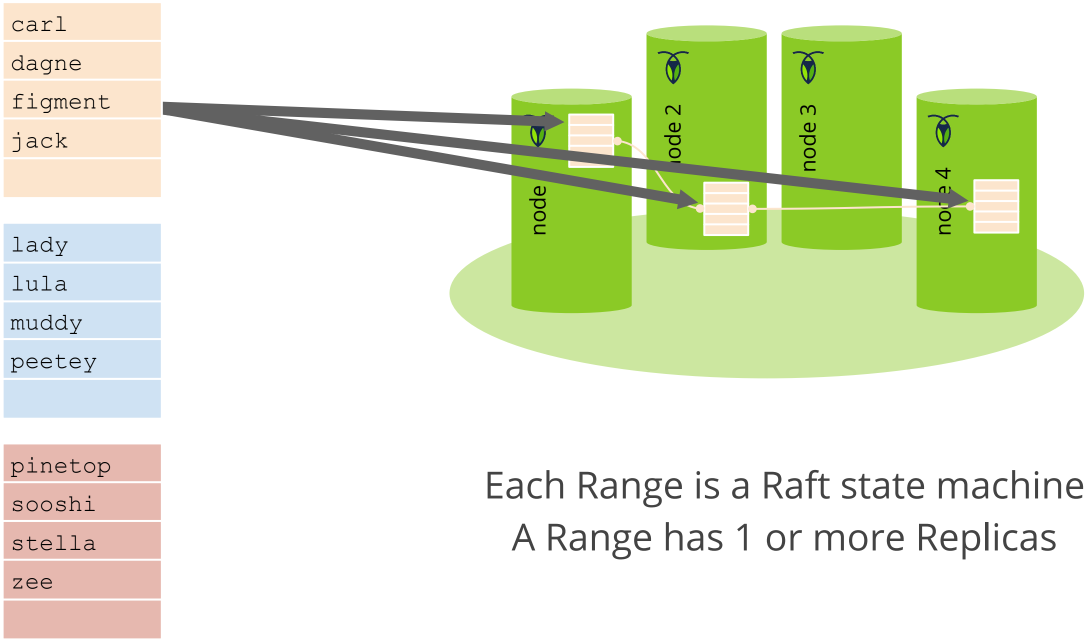
Reads without consensus

One replica is chosen as the leaseholder

- Coordinates writes (proposal, key locking)
- Performs reads

Replica Placement

- Space
- Diversity
- Load
- Latency



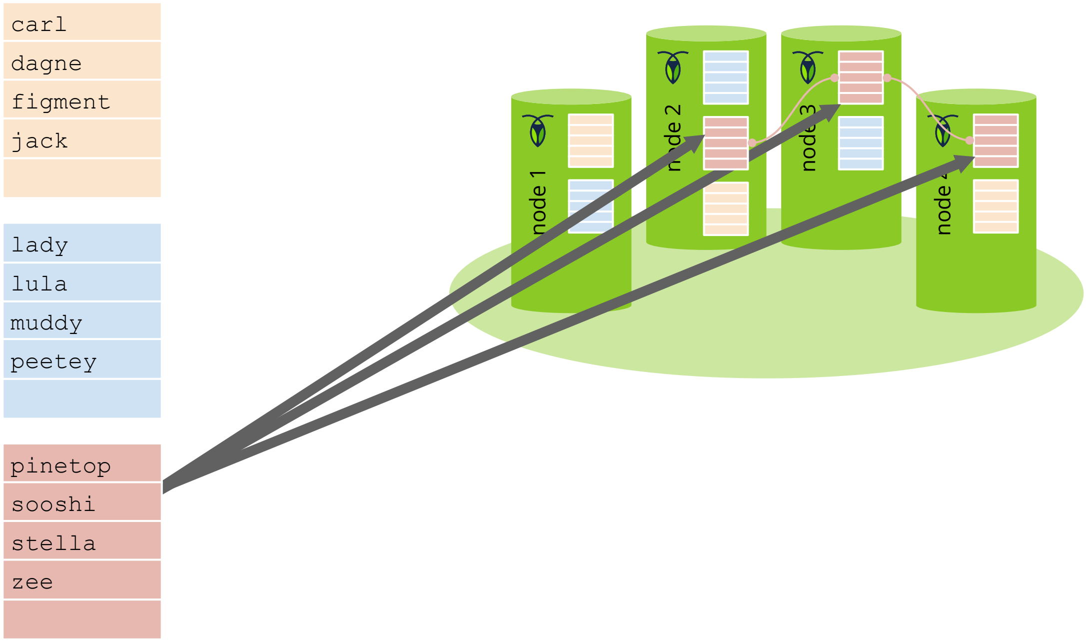
Each Range is a Raft state machine
A Range has 1 or more Replicas

Replica Placement: Diversity

Diversity

optimizes placement of replicas across “failure domains”

- Disk
- Single machine
- Rack
- Datacenter
- Region

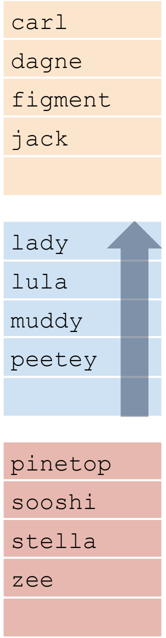


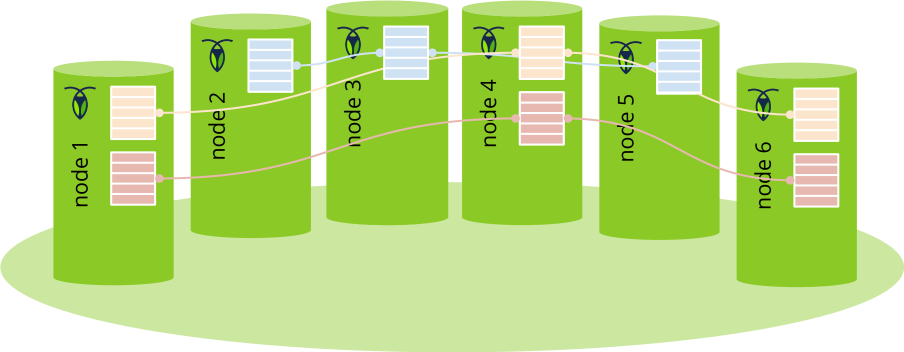
Replica Placement: Load

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

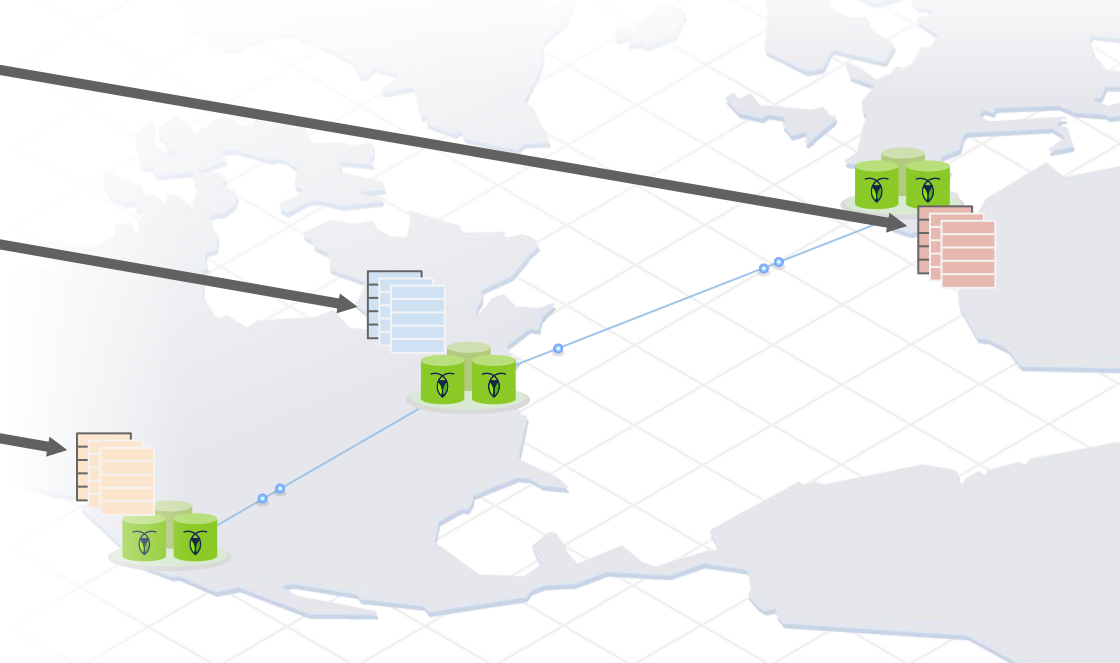
Replica Placement: Latency & Geo-partitioning

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



Cockroach LABS

Rebalancing Replicas

Scale: Add a node

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

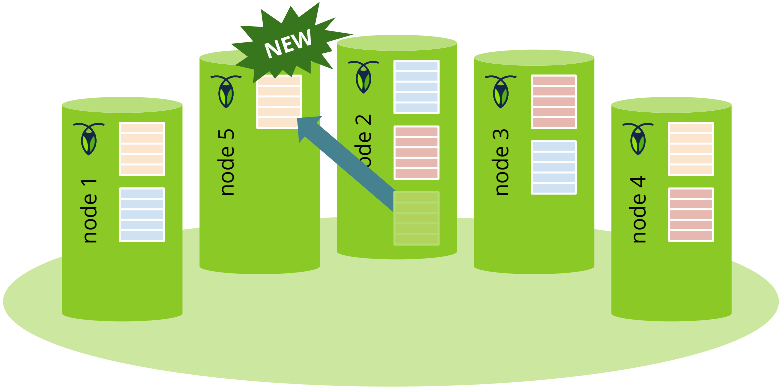
Uses the replica placement heuristics from previous slides to decide which node to add to and which to remove from

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

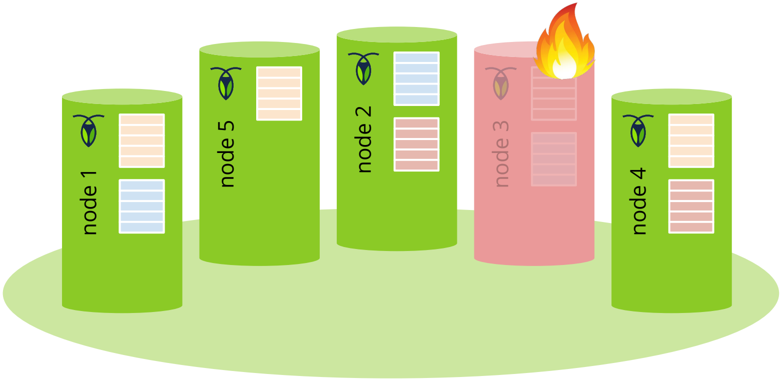
Rebalancing Replicas

Loss of a node

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



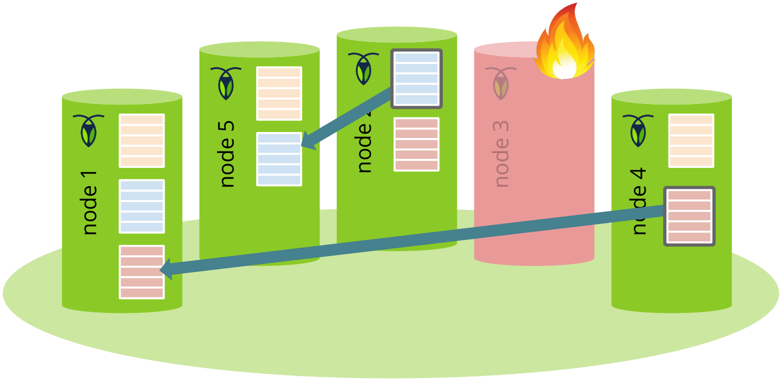
Rebalancing Replicas

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



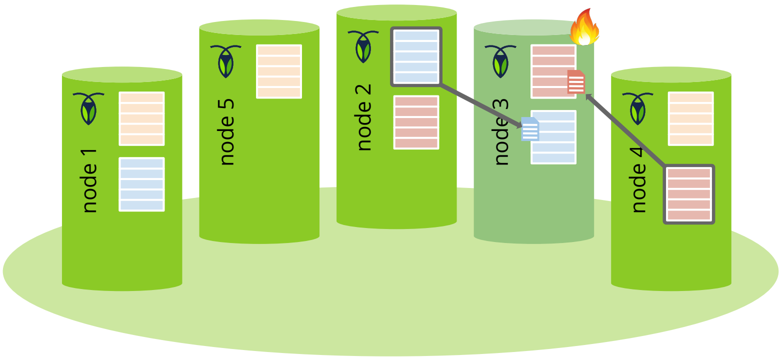
The failed replica is removed from the Raft group and a new replica created. The leaseholder sends a snapshot of the Range's state to bring the new replica up to date.

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.

AGENDA

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

Cockroach LABS

Transactions

Atomicity, **C**onsistency, **I**solation, **D**urability

Serializable Isolation

- As if the transactions are run in a serial order
- Gold standard isolation level
- Make Data Easy - weaker isolation levels are too great a burden

Transactions can span arbitrary ranges

Conversational

- The full set of operations is not required up front

Transactions

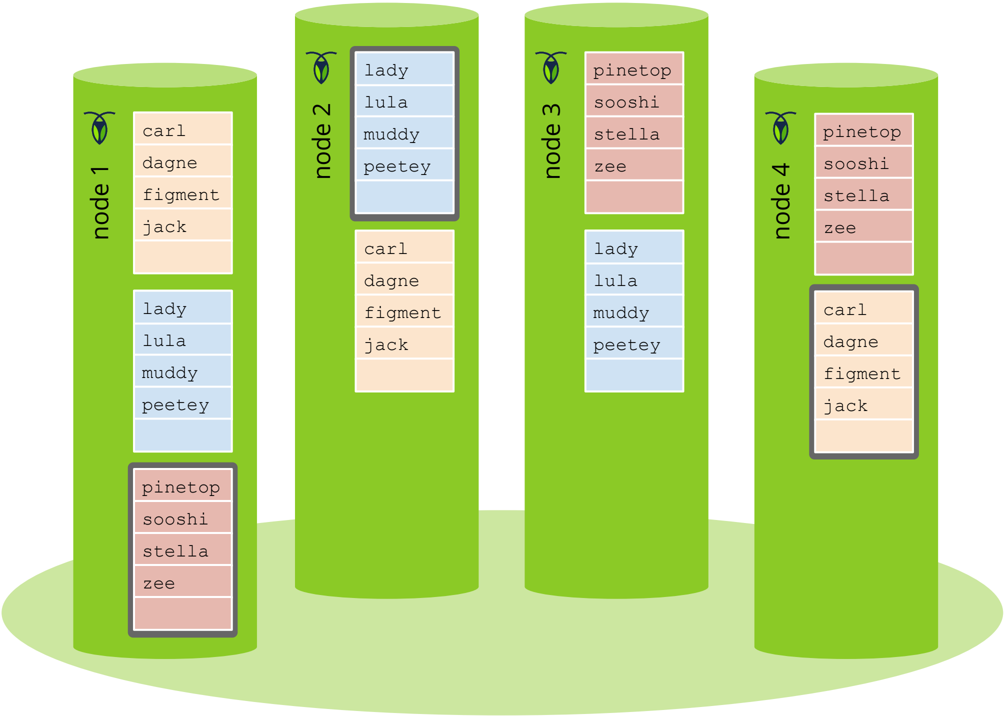
Raft provides atomic writes to individual ranges

Bootstrap transaction atomicity using Raft atomic writes

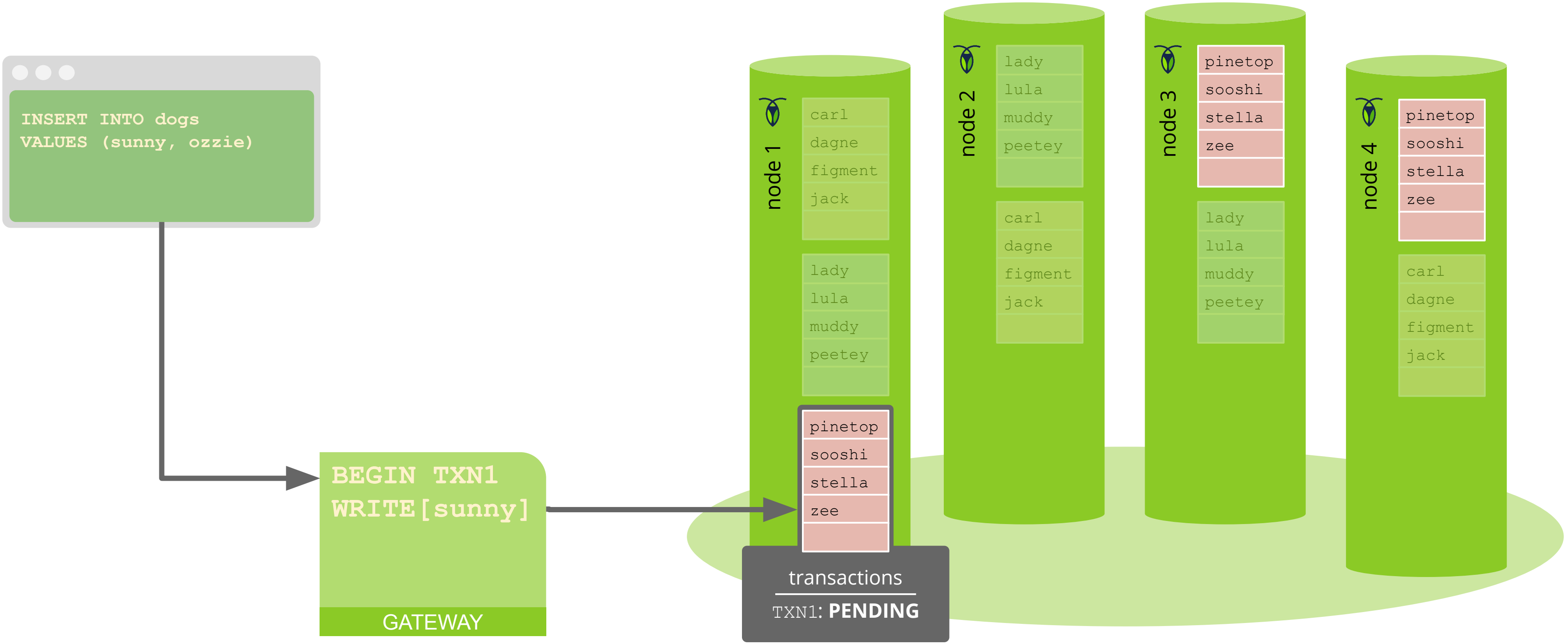
Transaction record atomically flipped from PENDING to COMMIT

Distributed Transactions

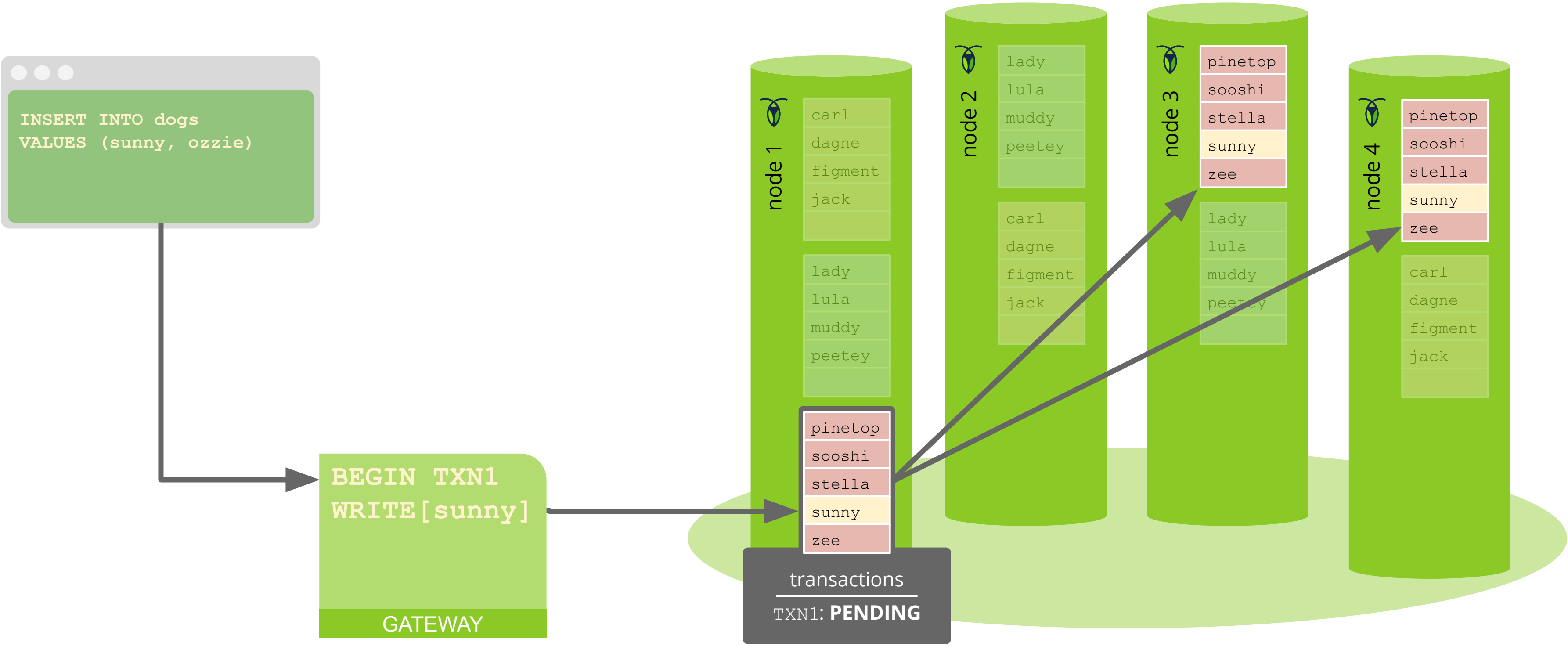
```
INSERT INTO dogs
VALUES (sunny, ozzie)
```



Distributed Transactions

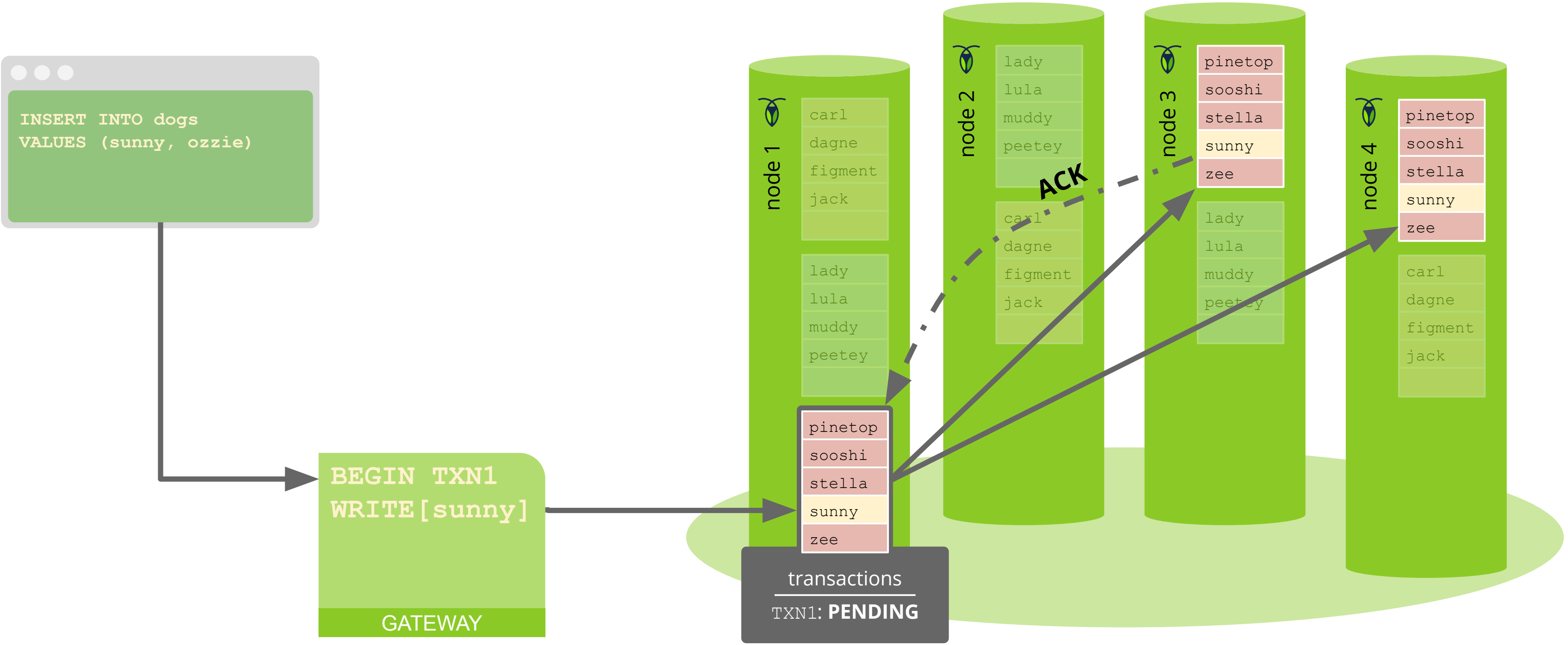


Distributed Transactions



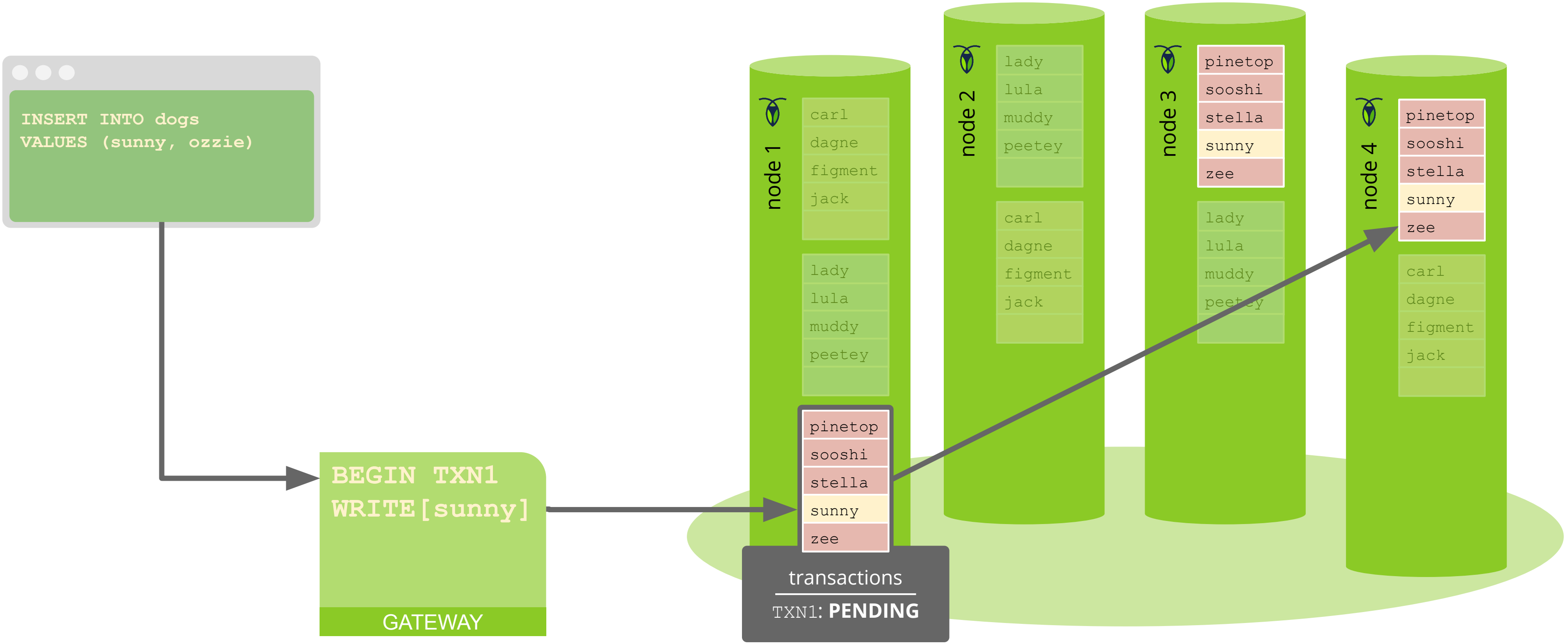
Cockroach LABS

Distributed Transactions

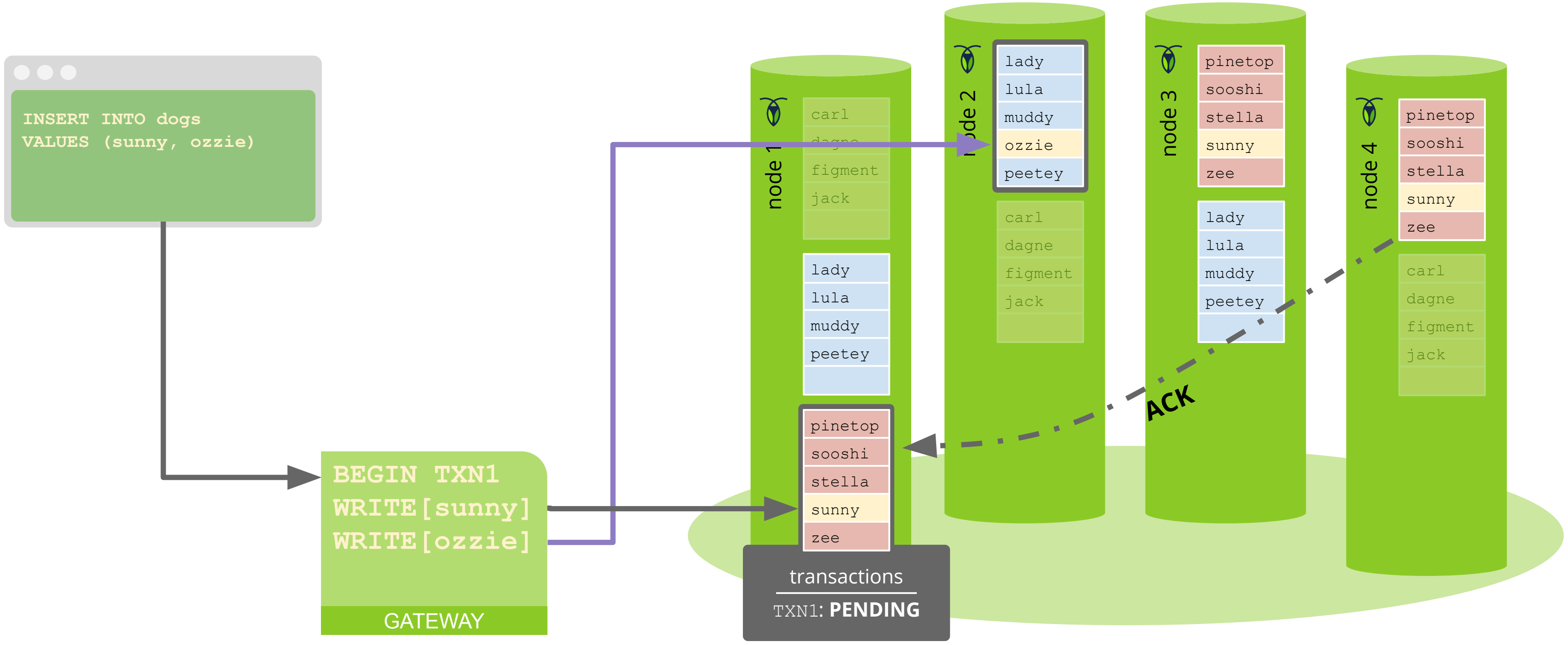


Cockroach LABS

Distributed Transactions

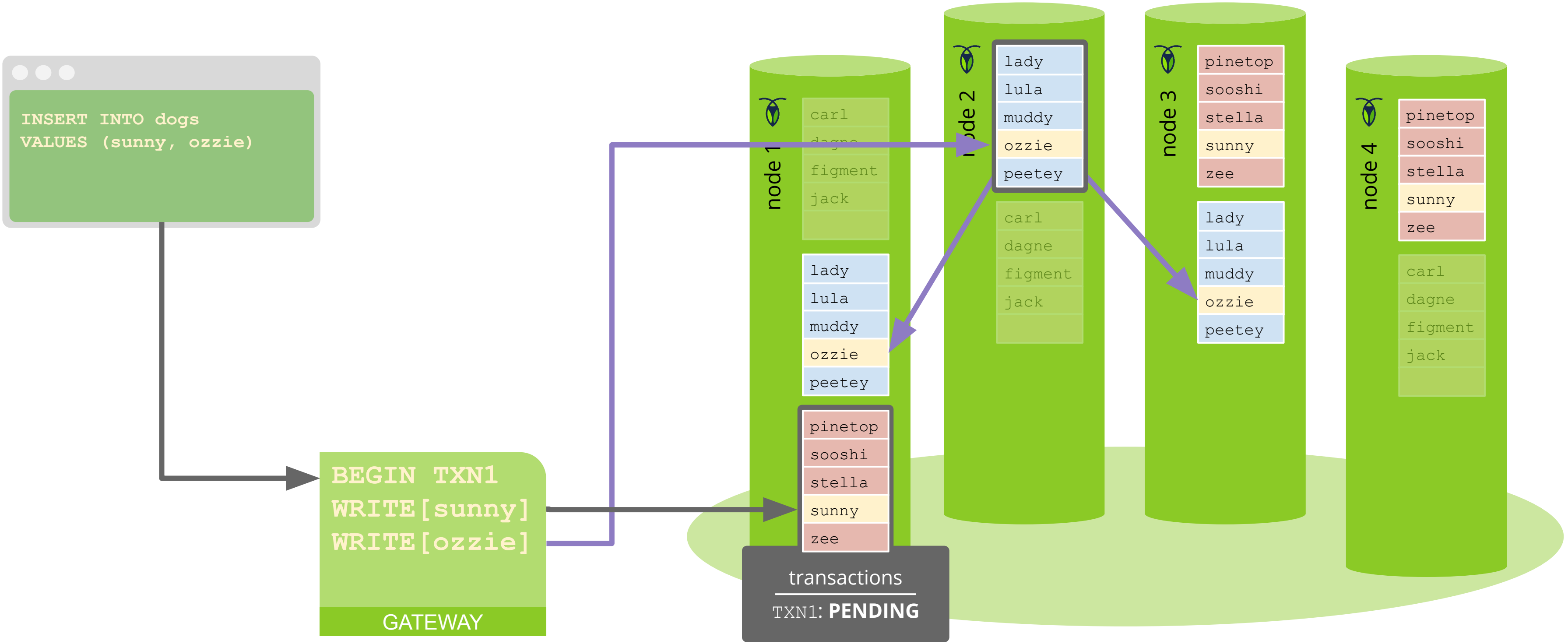


Distributed Transactions



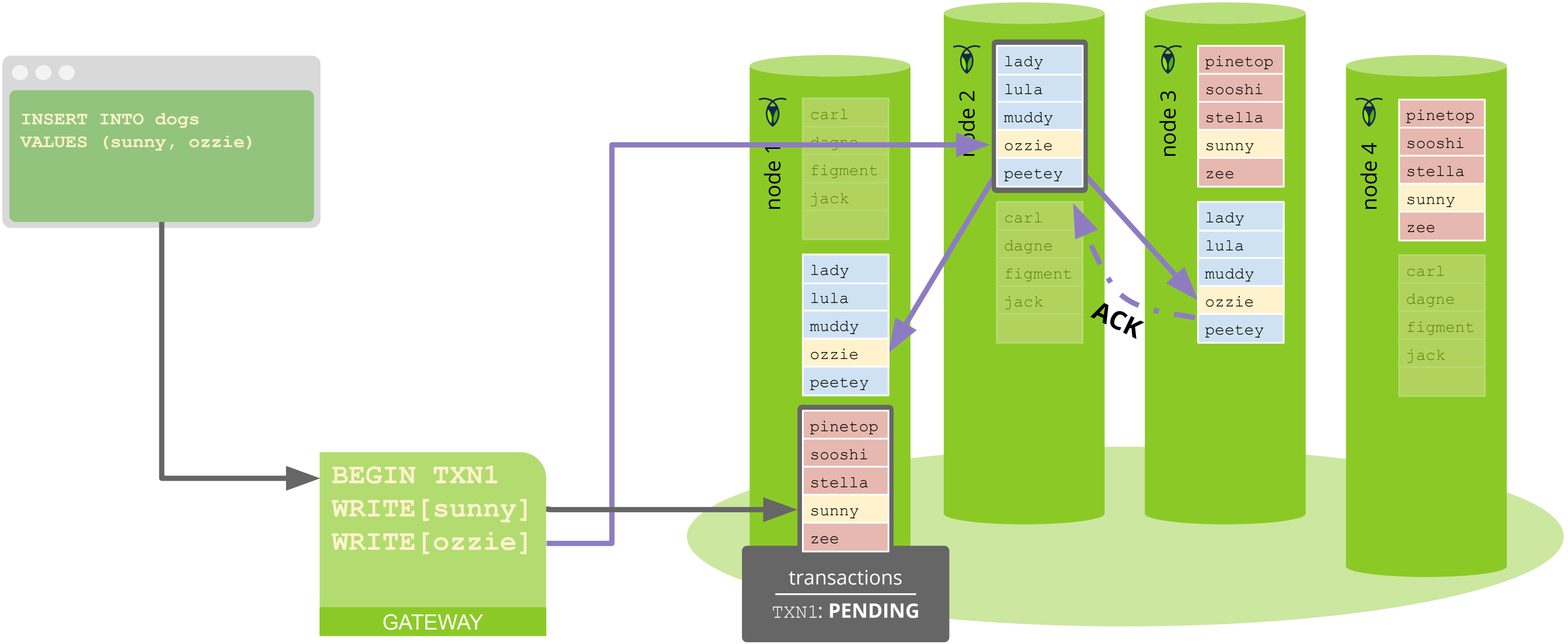
Cockroach LABS

Distributed Transactions



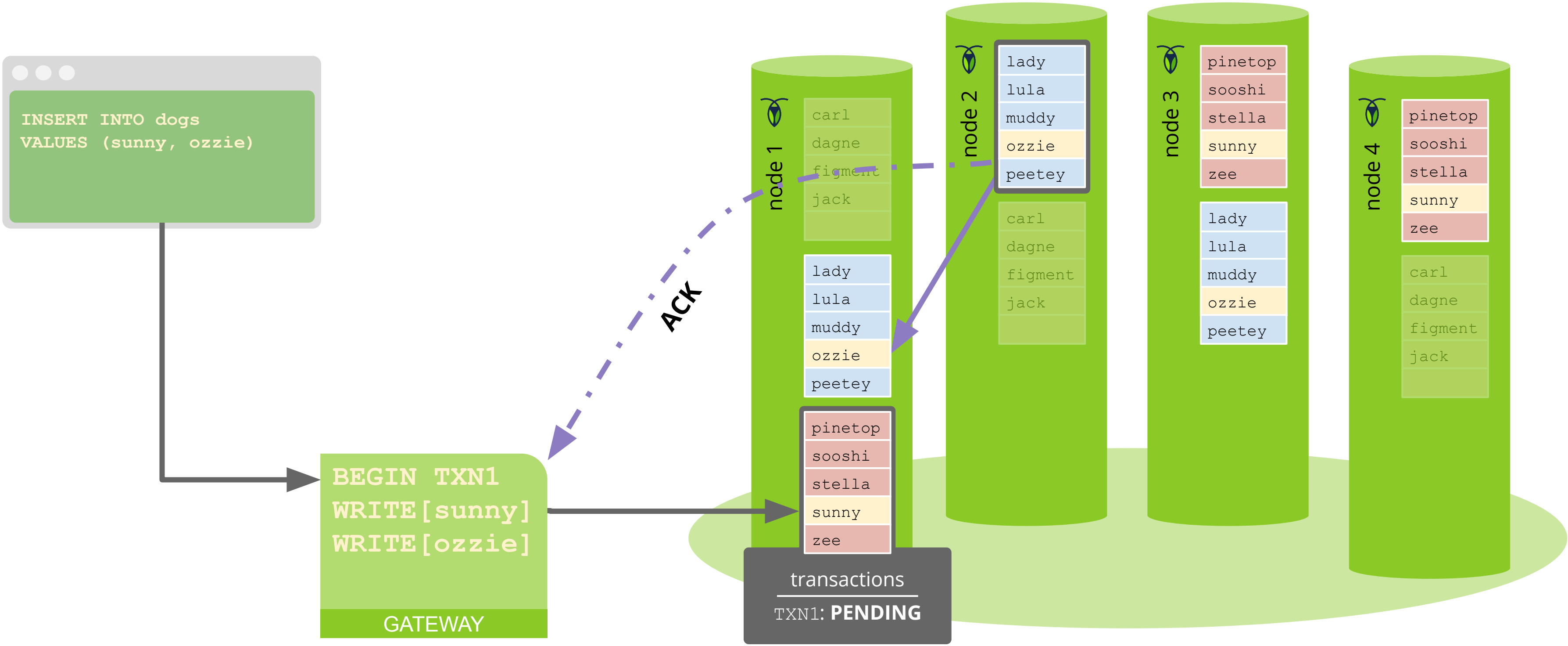
Cockroach LABS

Distributed Transactions

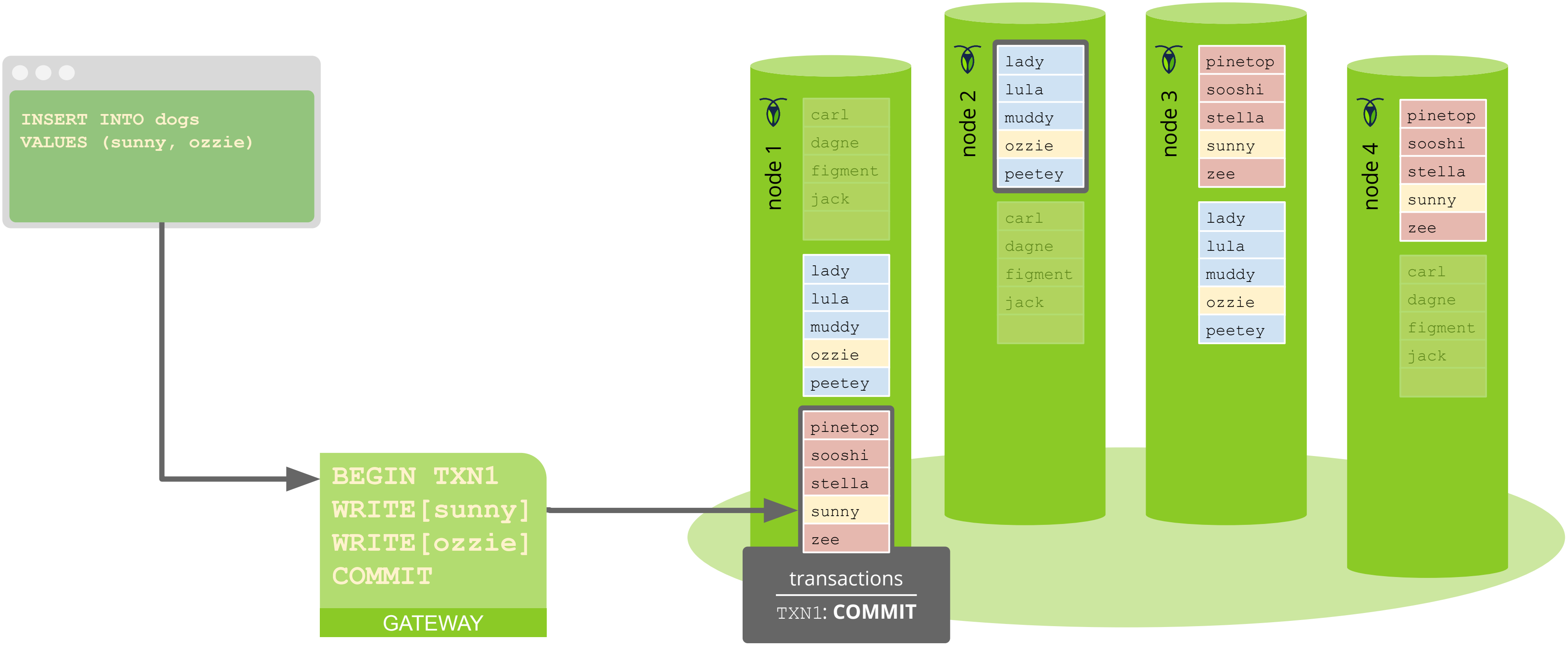


Cockroach LABS

Distributed Transactions

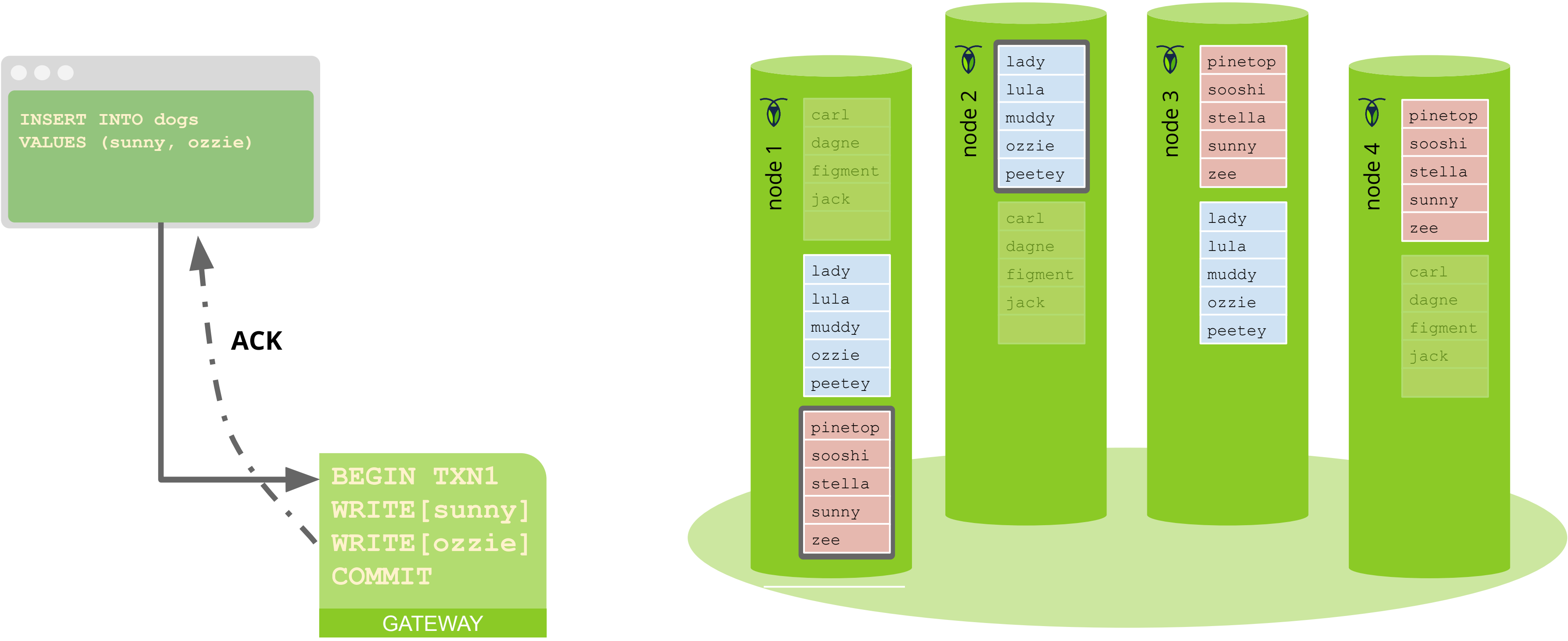


Distributed Transactions



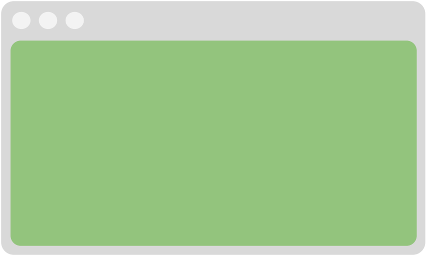
Cockroach LABS

Distributed Transactions



Cockroach LABS

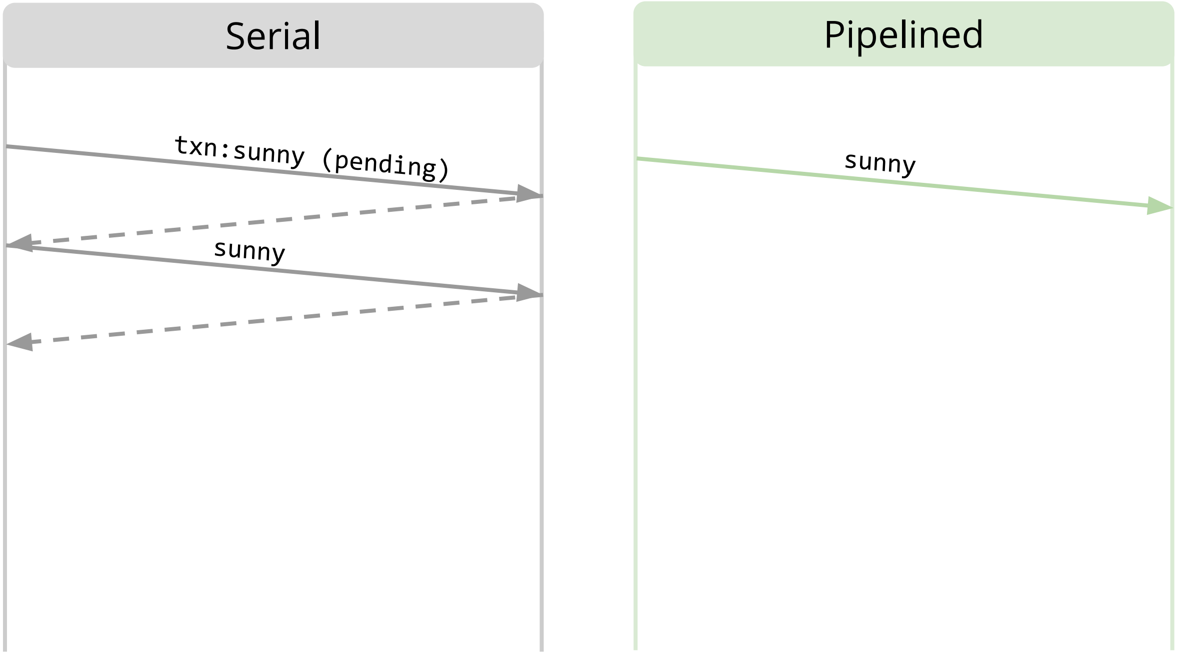
Transactions: Pipelining



Serial	Pipelined

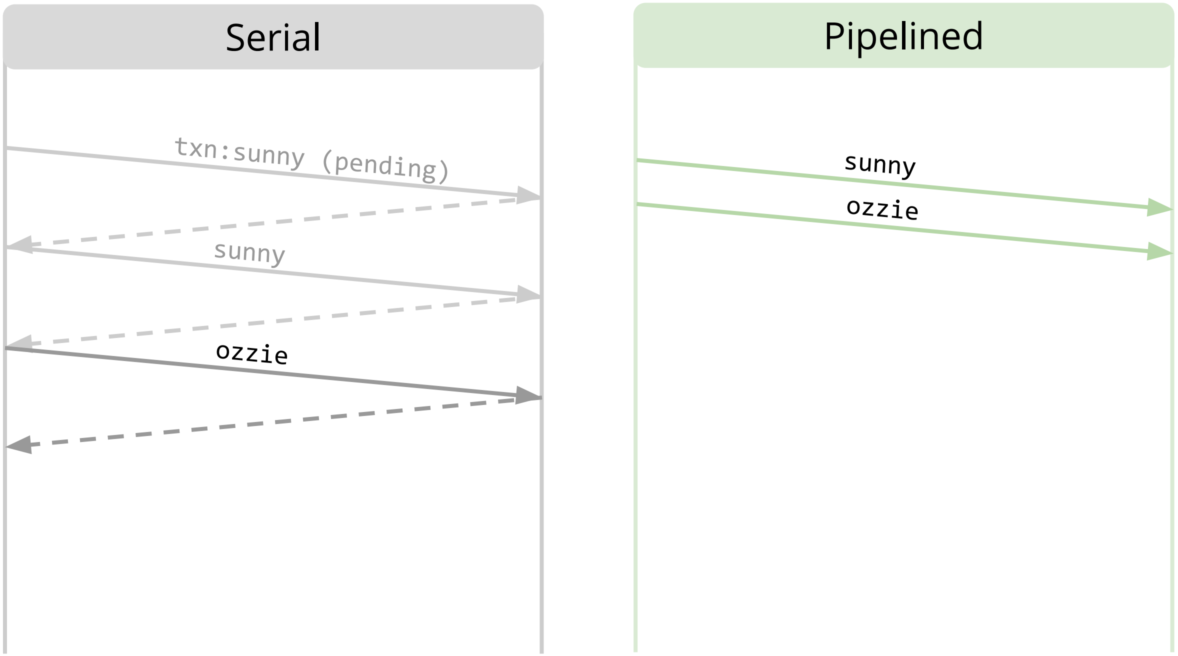
Transactions: Pipelining

```
BEGIN
WRITE[sunny]
```



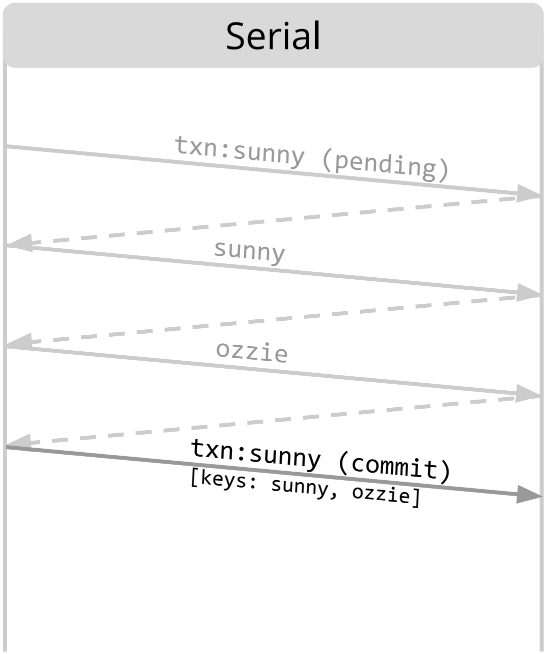
Transactions: Pipelining

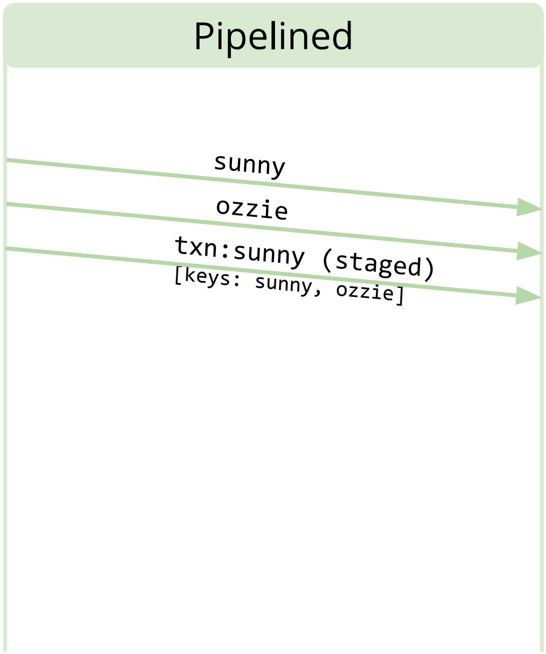
```
BEGIN
WRITE[sunny]
WRITE[ozzie]
```



Transactions: Pipelining

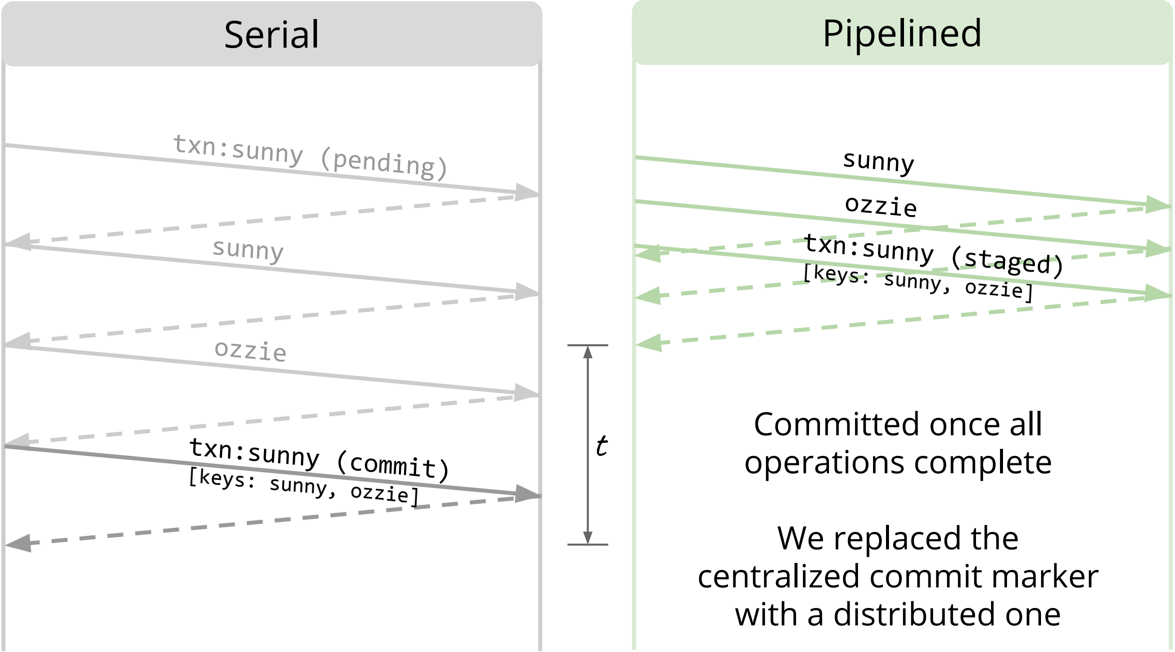
```
BEGIN
WRITE[sunny]
WRITE[ozzie]
COMMIT
```





Transactions: Pipelining

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



* “Proved” with TLA+

Cockroach LABS

AGENDA

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

Cockroach LABS

SQL

Structured **Q**uery **L**anguage

Declarative, not imperative

- These are the results I want vs perform these operations in this sequence

Relational data model

- Typed: INT, FLOAT, STRING, ...
- Schemas: tables, rows, columns, foreign keys

SQL: Tabular Data in a KV World

SQL data has columns and types?!?

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

Cockroach LABS

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

Cockroach LABS

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/”Bat”/1	∅
/inventory/name_idx/”Ball”/2	∅
/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/”Bat”/1	∅
/inventory/name_idx/”Ball”/2	∅
/inventory/name_idx/”Glove”/3	∅
/inventory/name_idx/”Bat”/4	∅

AGENDA

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

Cockroach LABS

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 input expressions 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
- 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”
```

Scan
inventory

SQL Execution: Filter

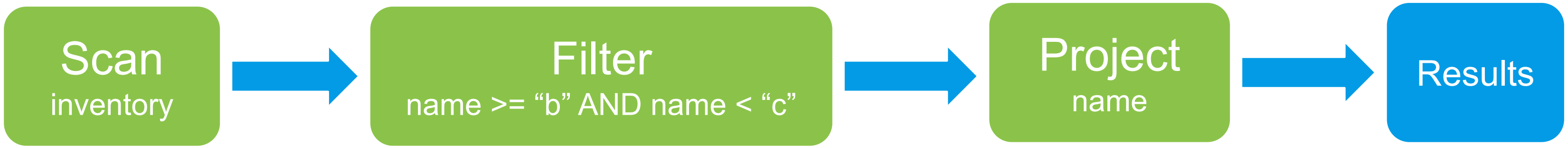
```
SELECT name
FROM   inventory
WHERE  name >= “b” AND name < “c”
```

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”
```

Scan
inventory@name [“b” - “c”)

The filter gets pushed into the scan

SQL Execution: Index Scans

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

Cockroach LABS

SQL Execution: Correctness

Correct SQL execution involves lots of bookkeeping

- User defined tables, and indexes
- Queries refer to table and column names
- Execution uses table and column IDs
- NULL handling

SQL Execution: Performance

Performant SQL execution

- Tight, well written code
- Operator specialization
  - hash group by, stream group by
  - hash join, merge join, lookup join, zig-zag join
- Distributed execution

SQL Execution: Group By

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

Name	Country
Bob	United States
Hans	Germany
Jacques	France
Marie	France
Susan	United States

SQL Execution: Hash Group By

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

Name	Country
Bob	United States
Hans	Germany
Jacques	France
Marie	France
Susan	United States

SQL Execution: Hash Group By

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

United States	**1**

Name	Country
Bob	United States
Hans	Germany
Jacques	France
Marie	France
Susan	United States

SQL Execution: Hash Group By

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

United States	1
Germany	**1**

Name	Country
Bob	United States
Hans	Germany
Jacques	France
Marie	France
Susan	United States

Cockroach LABS

SQL Execution: Hash Group By

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

United States	1
Germany	1
France	**1**

Name	Country
Bob	United States
Hans	Germany
Jacques	France
Marie	France
Susan	United States

SQL Execution: Hash Group By

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

United States	1
Germany	1
France	2

Name	Country
Bob	United States
Hans	Germany
Jacques	France
Marie	France
Susan	United States

Cockroach LABS

SQL Execution: Hash Group By

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

United States	**2**
Germany	1
France	2

Name	**Country**
Bob	United States
Hans	Germany
Jacques	France
Marie	France
Susan	United States

SQL Execution: Group By Revisited

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

Name	Country
Bob	United States
Hans	Germany
Jacques	France
Marie	France
Susan	United States

SQL Execution: Sort on Grouping Column(s)

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

Name	**Country**
Jacques	France
Marie	France
Hans	Germany
Bob	United States
Susan	United States

SQL Execution: Streaming Group By

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

France	**1**

Name	Country
Jacques	France
Marie	France
Hans	Germany
Bob	United States
Susan	United States

SQL Execution: Streaming Group By

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

France	2

Name	Country
Jacques	France
Marie	France
Hans	Germany
Bob	United States
Susan	United States

SQL Execution: Streaming Group By

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

France	2
Germany	**1**

→

Name	**Country**
Jacques	France
Marie	France
Hans	Germany
Bob	United States
Susan	United States

SQL Execution: Streaming Group By

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

France	2
Germany	1
United States	**1**

→

Name	Country
Jacques	France
Marie	France
Hans	Germany
Bob	United States
Susan	United States

SQL Execution: Streaming Group By

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

France	2
Germany	1
United States	2

Name	Country
Jacques	France
Marie	France
Hans	Germany
Bob	United States
Susan	United States

Cockroach LABS

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

Cockroach LABS

Distributed SQL Execution: Streaming Group By

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

Scan customers | Scan customers | Scan customers

Cockroach LABS

Distributed SQL Execution: Streaming Group By

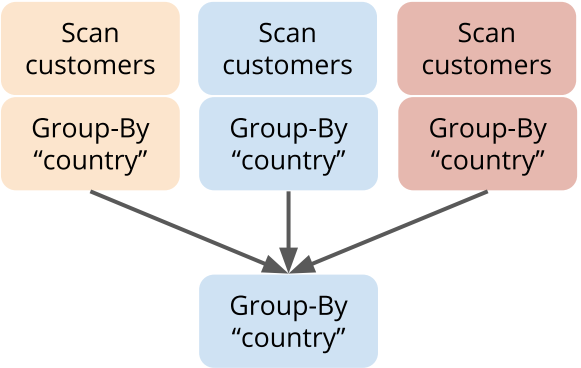
```
SELECT   COUNT(*), country
FROM     customers
GROUP BY country
```

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

Cockroach LABS

Distributed SQL Execution: Streaming Group By

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



Cockroach LABS

AGENDA

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

Cockroach LABS

SQL Optimization

An optimizer explores many plans that are logically equivalent to a given query and chooses the best one

Parse SQL

Fold Constants
Check Types
Resolve Names
Report Semantic Errors
Compute properties
Retrieve and attach stats
Cost-independent transformations

Cost-based transformations

SQL Optimization: Cost-Independent Transformations

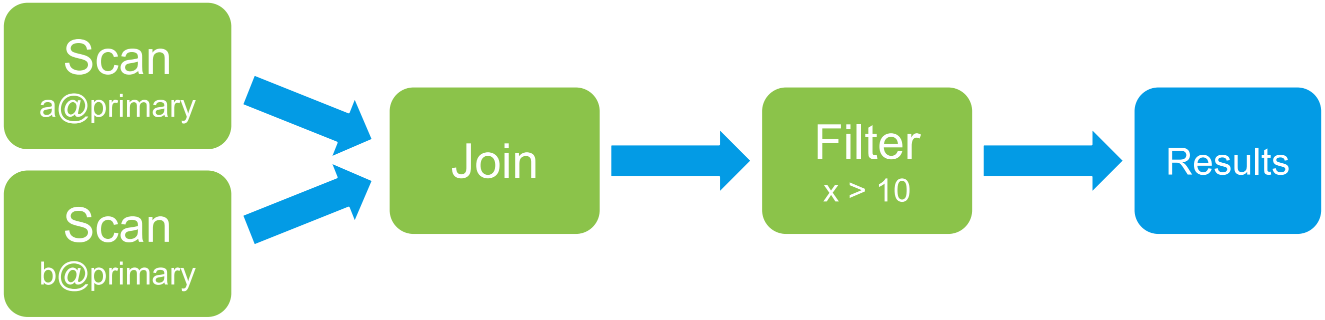
- Some transformations always make sense
  - Constant folding
  - Filter push-down
  - Decorrelating subqueries*
  - …
- These transformations are cost-independent
  - If the transformation can be applied to the query, it is applied
- **D**omain **S**pecific **L**anguage for transformations
  - Compiled down to code which efficiently matches query fragments in the memo
  - ~200 transformations currently defined

* Actually cost-based, but we're treating it as cost-independent right now

SQL Optimization: Filter Push-Down

```
SELECT * FROM a JOIN b WHERE x > 10
```

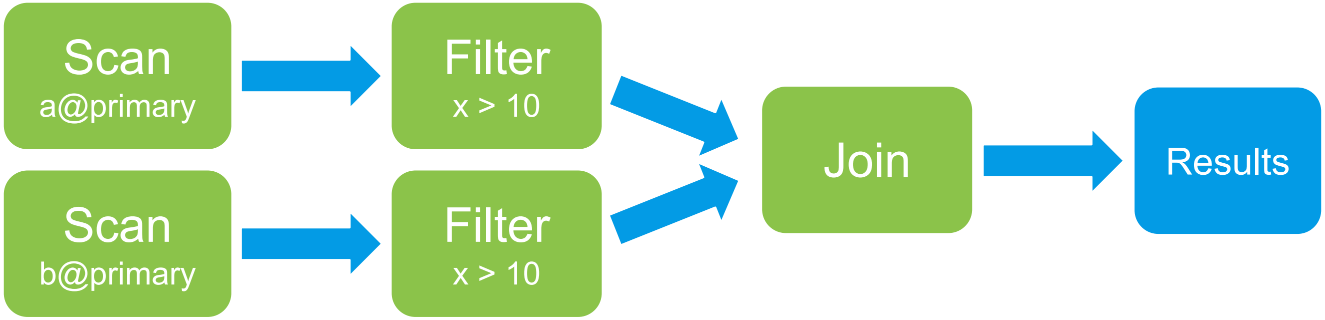
Initial plan



SQL Optimization: Filter Push-Down

```
SELECT * FROM a JOIN b WHERE x > 10
```

After filter push-down



SQL Optimization: Cost-Based Transformations

- Some transformations are not universally good
  - Index selection
  - Join reordering
  - …
- These transformations are cost-based
  - When should the transformation be applied?
  - Need to try both paths and maintain both the original and transformed query
  - State explosion: thousands of possible query plans
    - Memo data structure maintains a forest of query plans
  - Estimate cost of each query, select query with lowest cost
- Costing
  - Based on table statistics and estimating cardinality of inputs to relational expressions

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

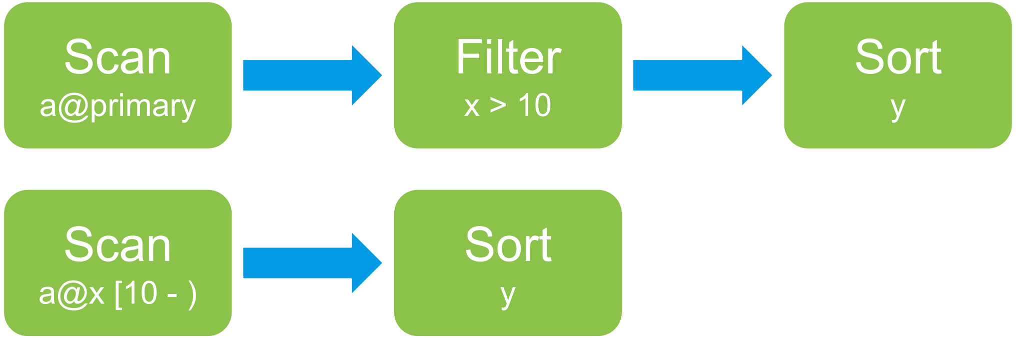
SQL Optimization: Cost-based Index Selection

```
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



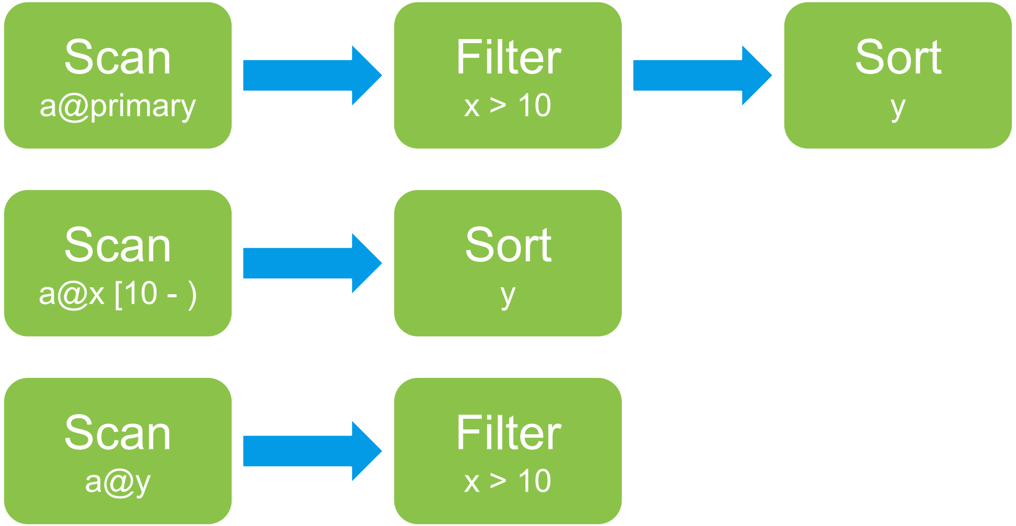
SQL Optimization: Cost-based Index Selection

```
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



Cockroach LABS

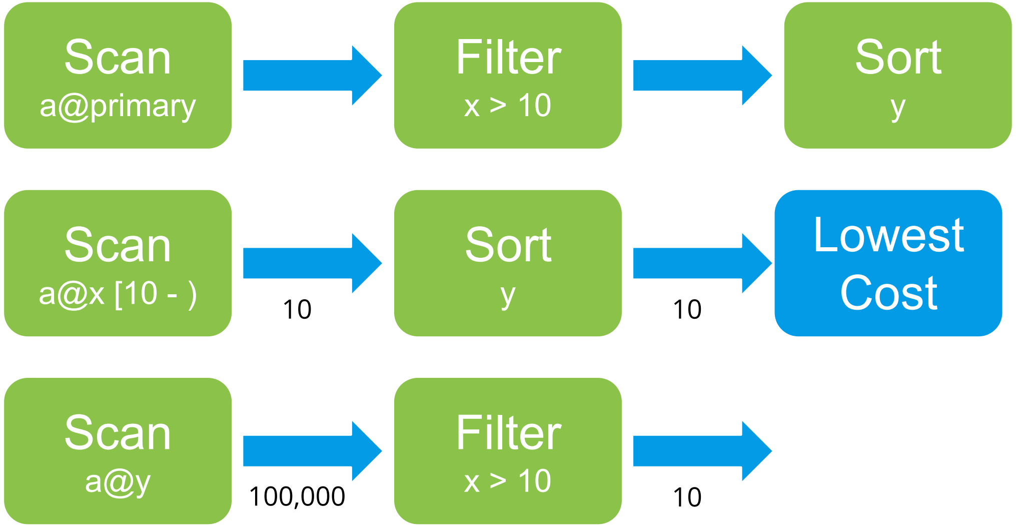
SQL Optimization: Cost-based Index Selection

```
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



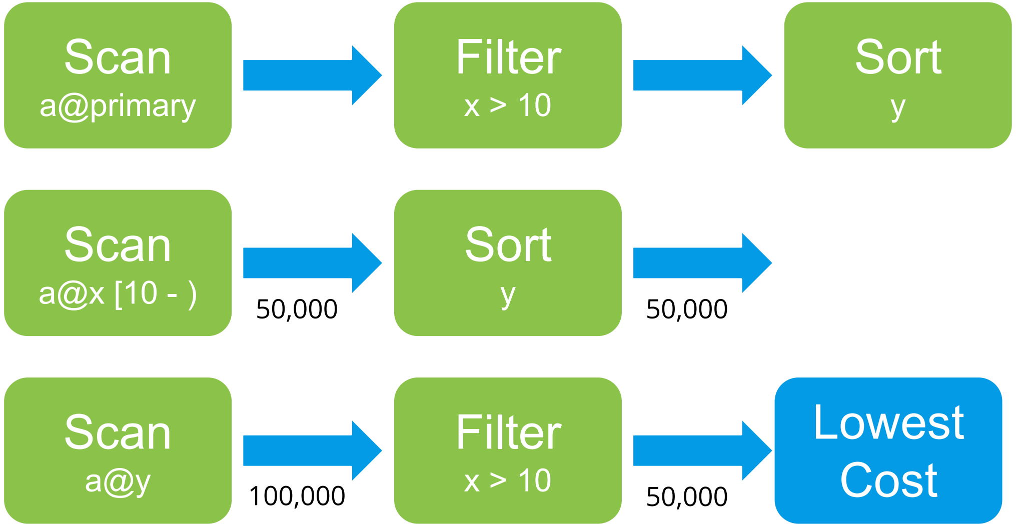
SQL Optimization: Cost-based Index Selection

```
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



Cockroach LABS

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

Cockroach LABS

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)
)
```

Cockroach LABS

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)
)
```

Cockroach LABS

Conclusion

- Distributed, replicated, transactional key-value store
- Monolithic key space
- Raft replication of ranges (~64MB)
- Replica placement signals: space, diversity, load, latency
- Pipelined transaction operations
- Mapping SQL data to KV storage
- Distributed SQL execution
- Distributed SQL optimization

Cockroach LABS

Thank You

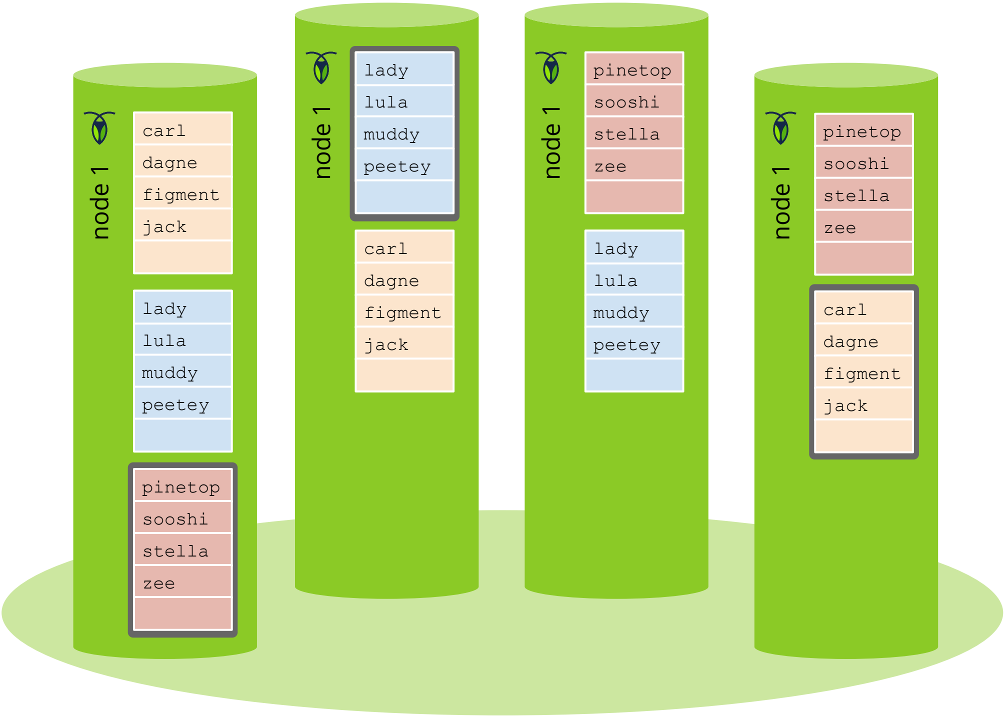
www.cockroachlabs.com

github.com/cockroachdb/cockroach

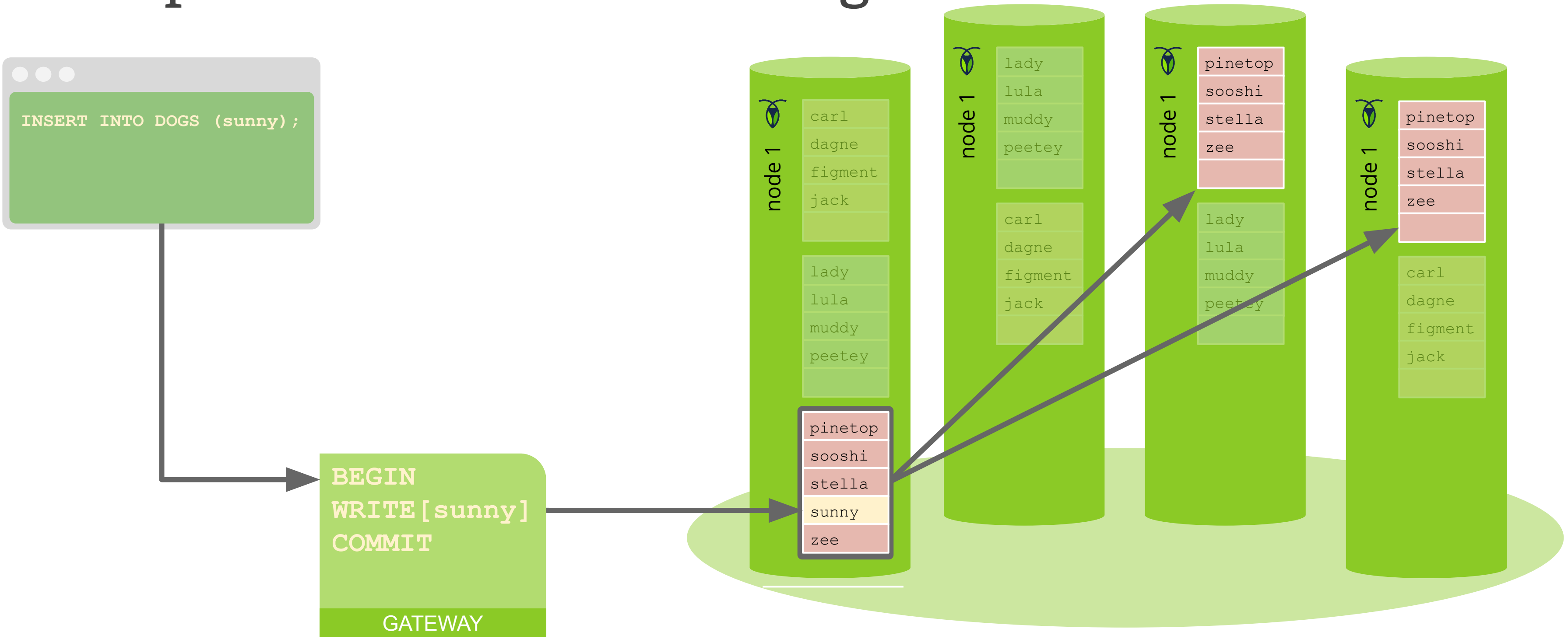
Cockroach LABS

A Simple Transaction

```
INSERT INTO DOGS (sunny);
```



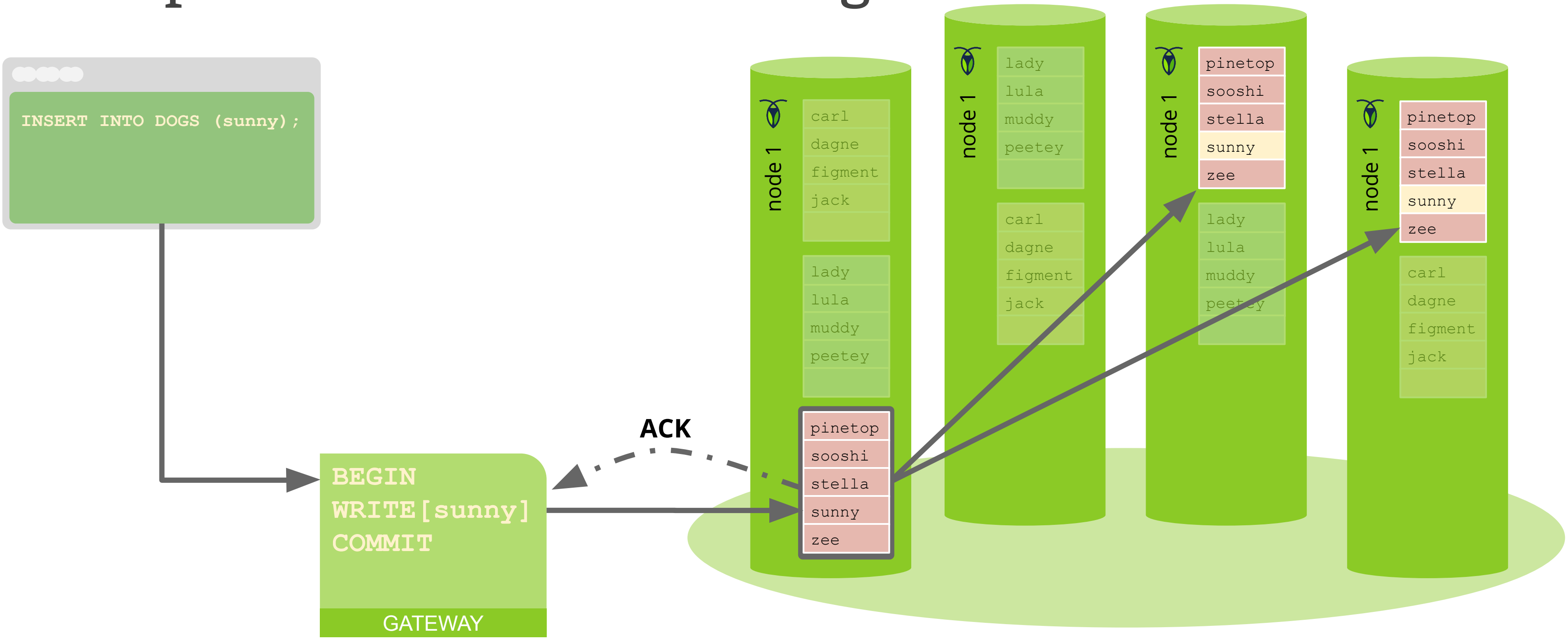
A Simple Transaction: One Range



NOTE: a gateway can be ANY CockroachDB instance. It can find the leaseholder for any range and execute a transaction

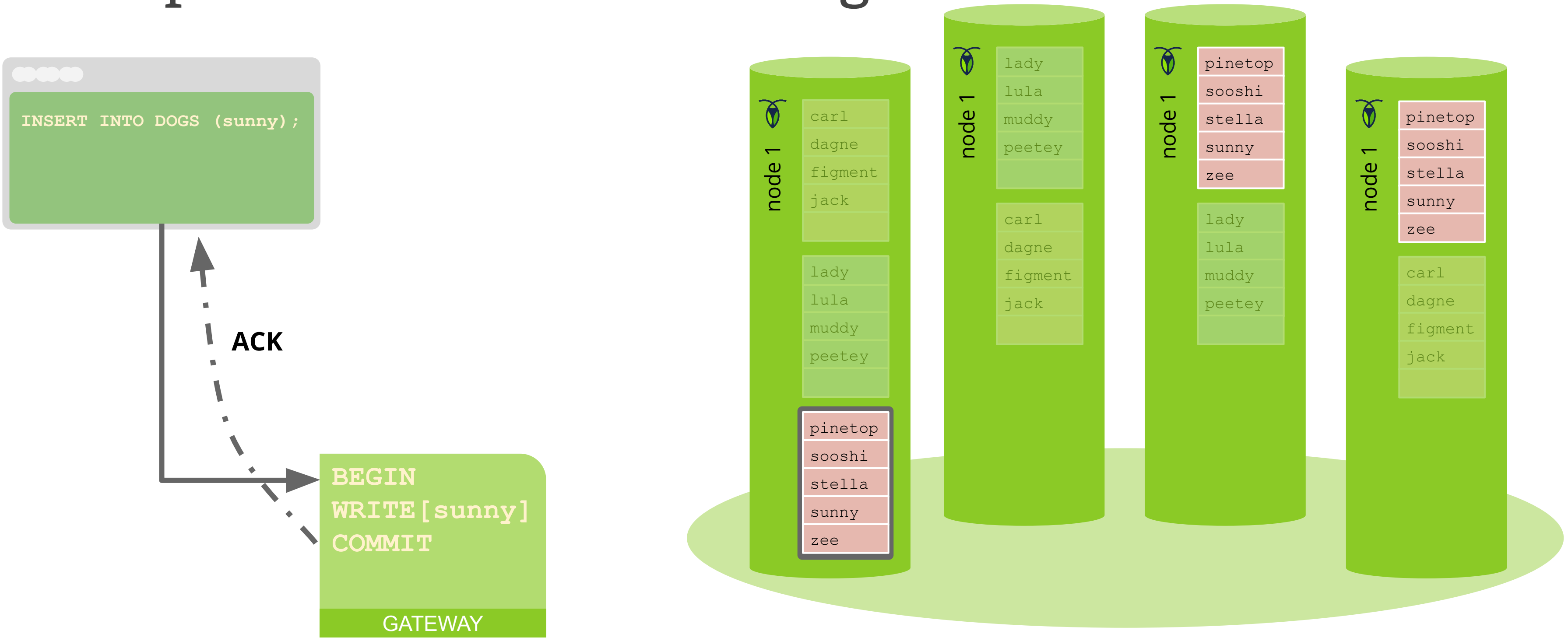
Cockroach LABS

A Simple Transaction: One Range



Cockroach LABS

A Simple Transaction: One Range



Ranges

CockroachDB implements **order-preserving data distribution**

- Automates sharding of key/value data into “ranges”
- Supports efficient range scans
- Requires an indexing structure

Foundational capability that enables efficient distribution of data across nodes within a CockroachDB cluster

* This approach is also used by Bigtable (tablets), HBase (regions) & Spanner (ranges)