CockroachDB

Architecture of a Geo-Distributed SQL Database

Nathan VanBenschoten (@natevanben), Staff Software Engineer

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”
  - Tombstones are used to delete values
  - Provides snapshot to each transaction
- Monolithic key-space

* Not exposed for external usage

Monolithic Key Space

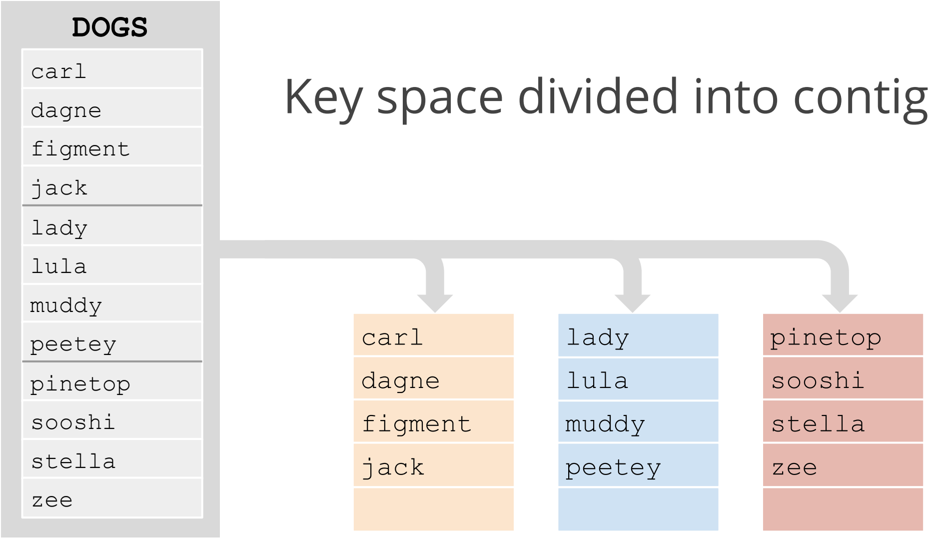
DOGS
carl
dagne
figment
jack
lady
lula
muddy
peetey
pinetop
sooshi
stella
zee

Monolithic logical key space

- Ordered lexicographically by key

Ranges

Key space divided into contiguous ~64MB ranges

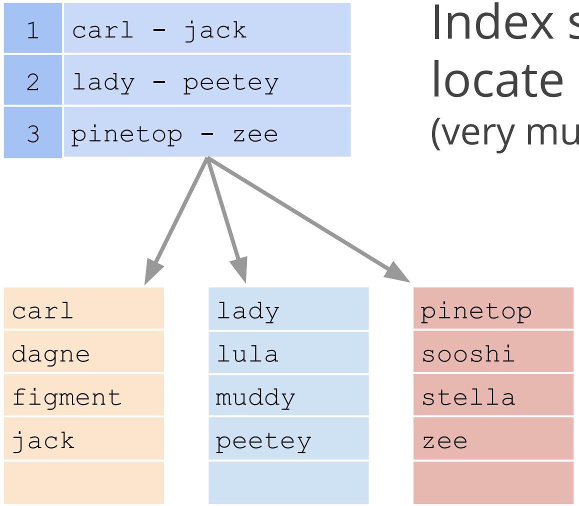


Ranges are small enough to be **moved/split quickly**

Ranges are large enough to **amortize indexing overhead**

Range Indexing

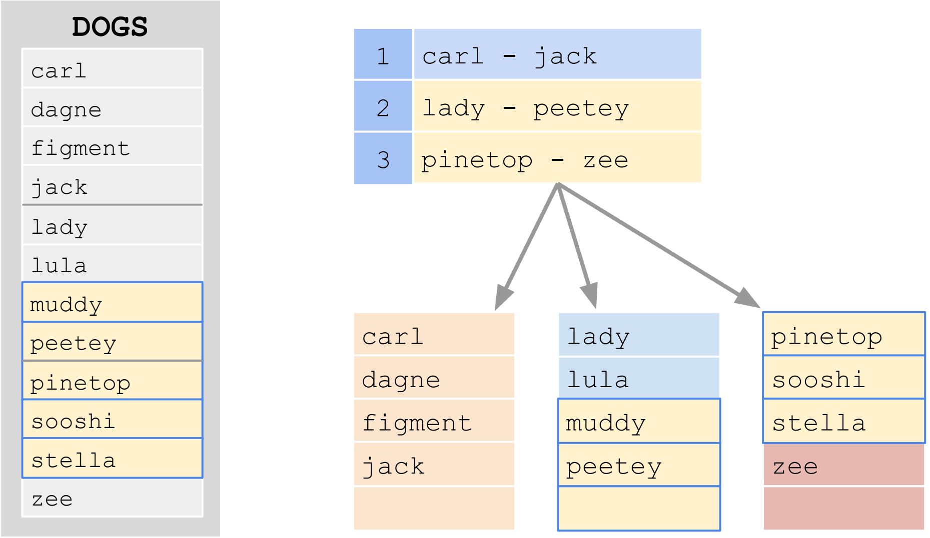
DOGS
carl
dagne
figment
jack
lady
lula
muddy
peetey
pinetop
sooshi
stella
zee



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

Raft and Replication

Ranges (~64MB) are the unit of replication

Each range is a Raft group
(Raft is a **distributed consensus protocol**)

Default to 3 replicas, though this is configurable

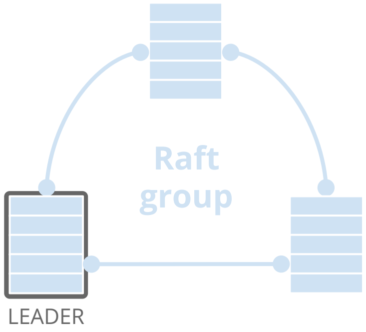
- Important system ranges default to 5 replicas

Raft and Replication

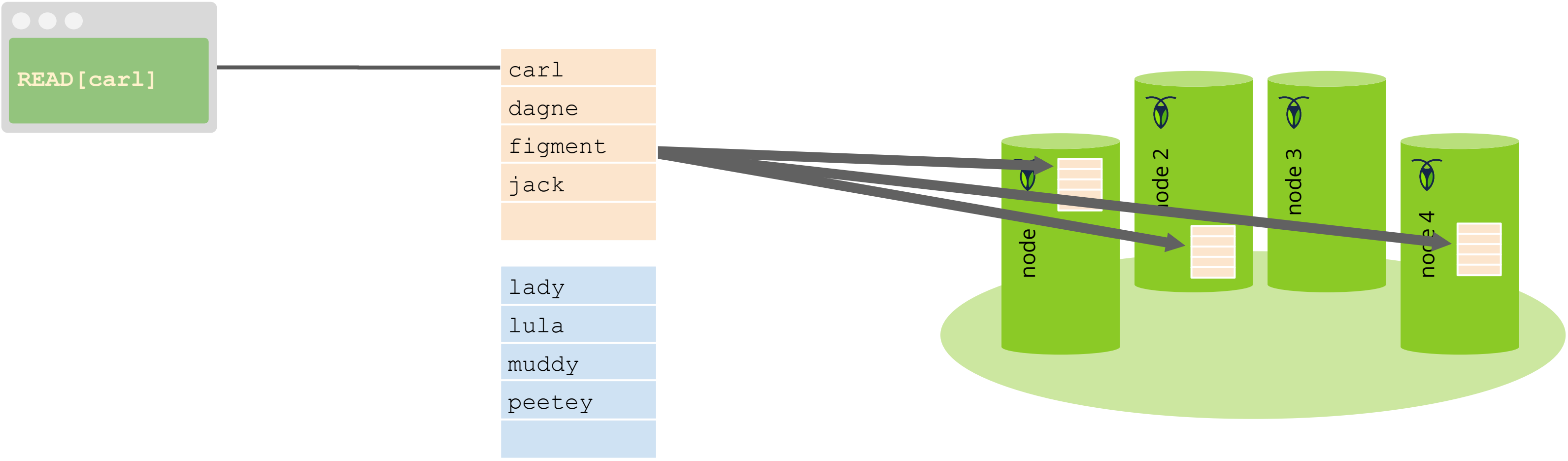
Raft provides “**atomic replication**” of commands

Each group elects a leader

Commands require **majority** of replicas to vote



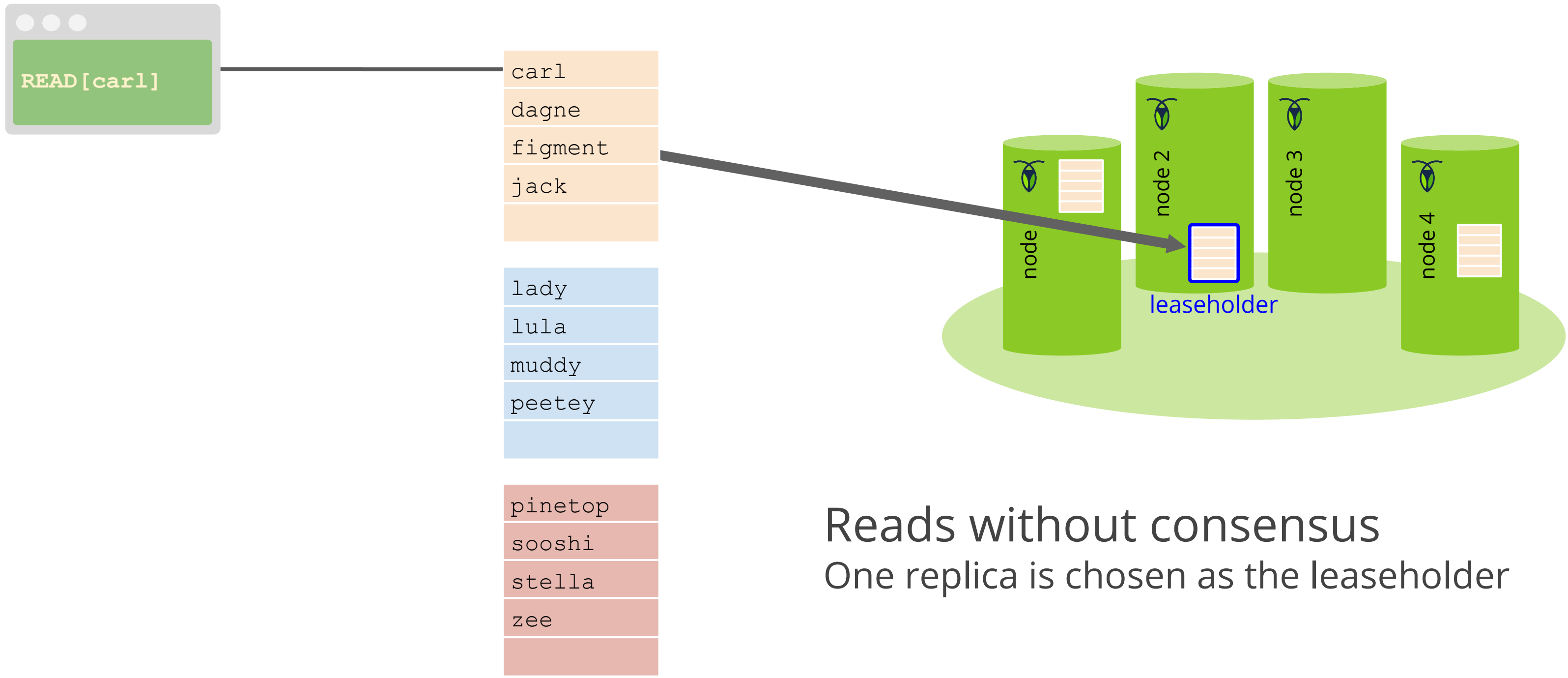
Range Leases



Reads with consensus

Reads must talk to a quorum of replicas

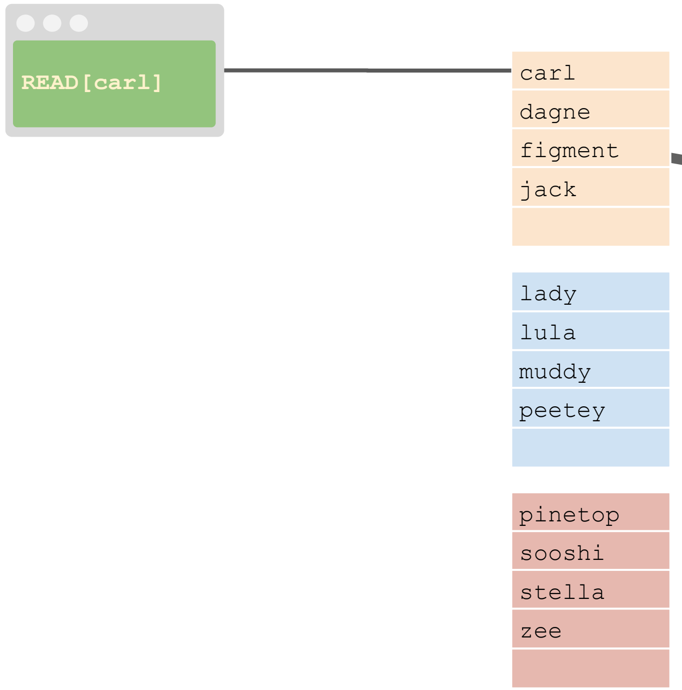
Range Leases

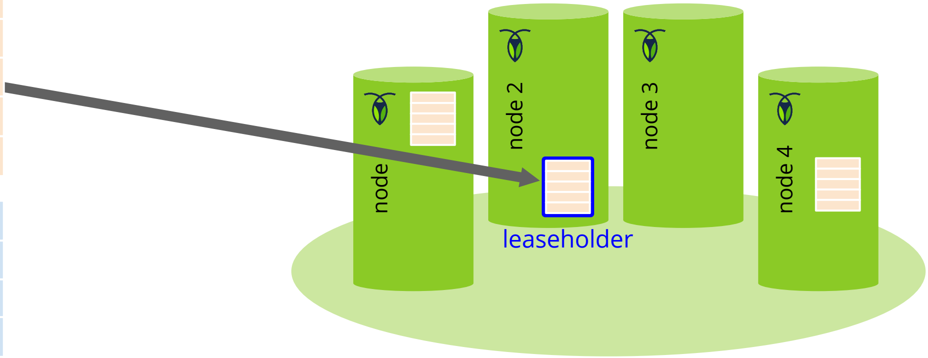


Reads without consensus

One replica is chosen as the leaseholder

Range Leases





Reads without consensus

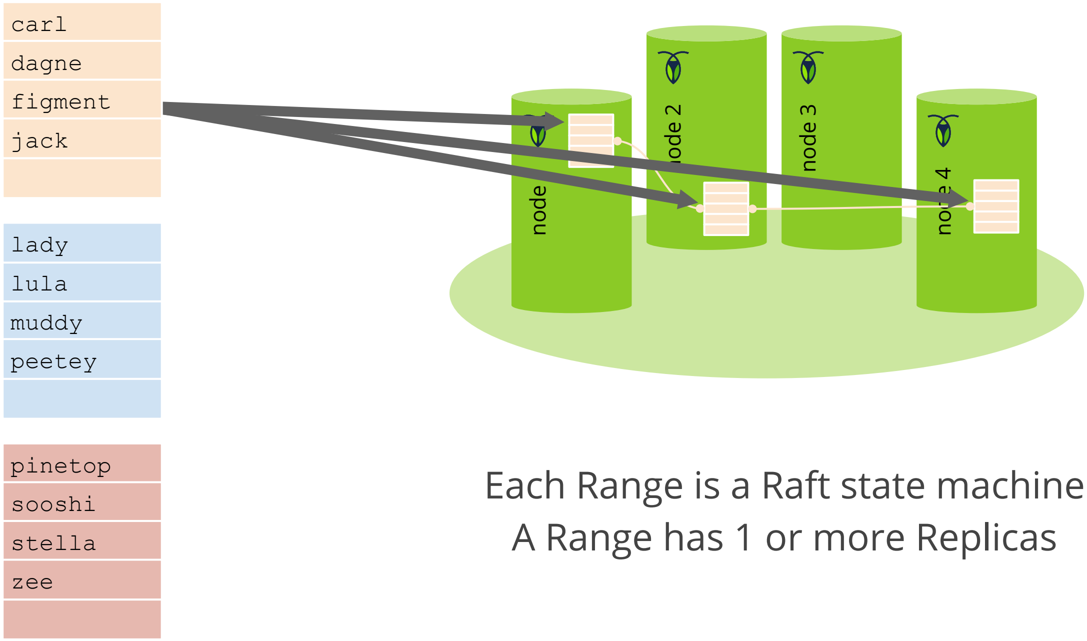
One replica is chosen as the leaseholder

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

Cockroach LABS

Replica Placement

- Space
- Diversity
- Load
- Latency



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

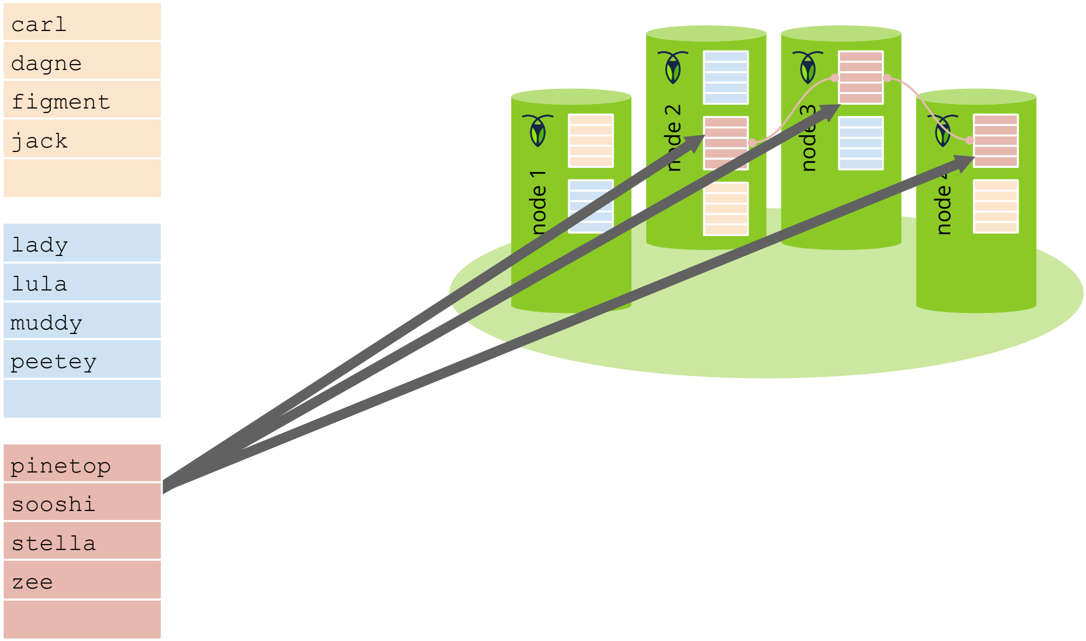
Cockroach LABS

Replica Placement: Diversity

Diversity

optimizes placement of replicas across “failure domains”

- Disk
- Single machine
- Rack
- Datacenter
- Region



Replica Placement: Load

Load

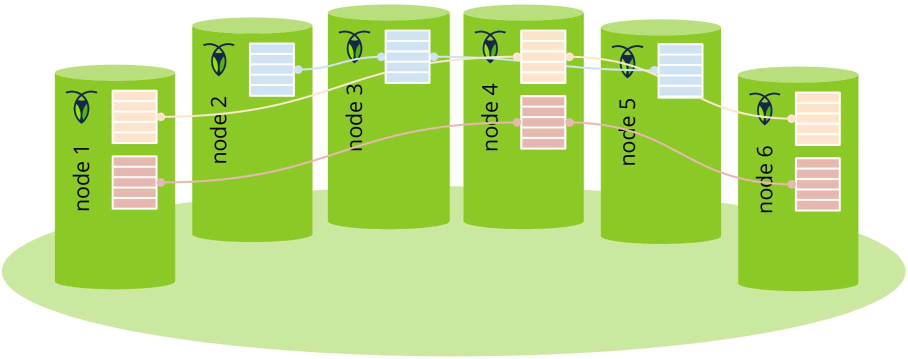
Balances placement using heuristics that considers real-time usage metrics of the data itself

carl
dagne
figment
jack

lady
lula
muddy
peetey

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

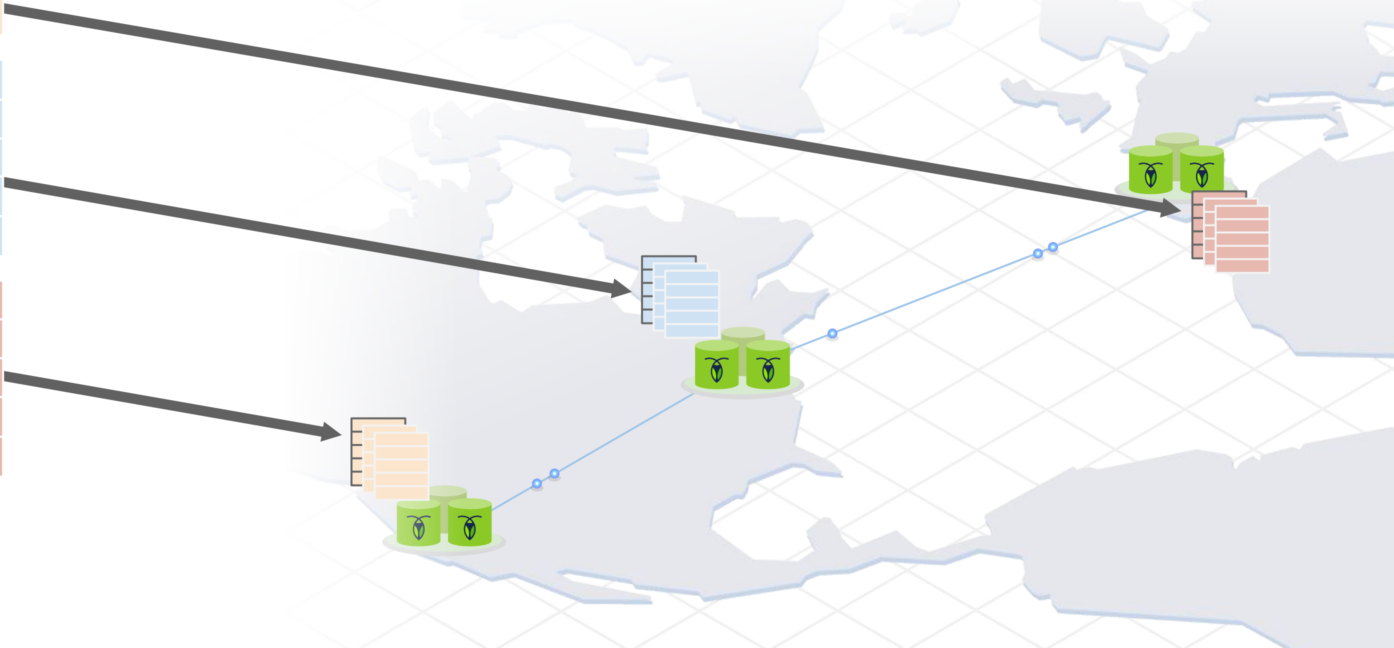
pinetop
sooshi
stella
zee



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

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

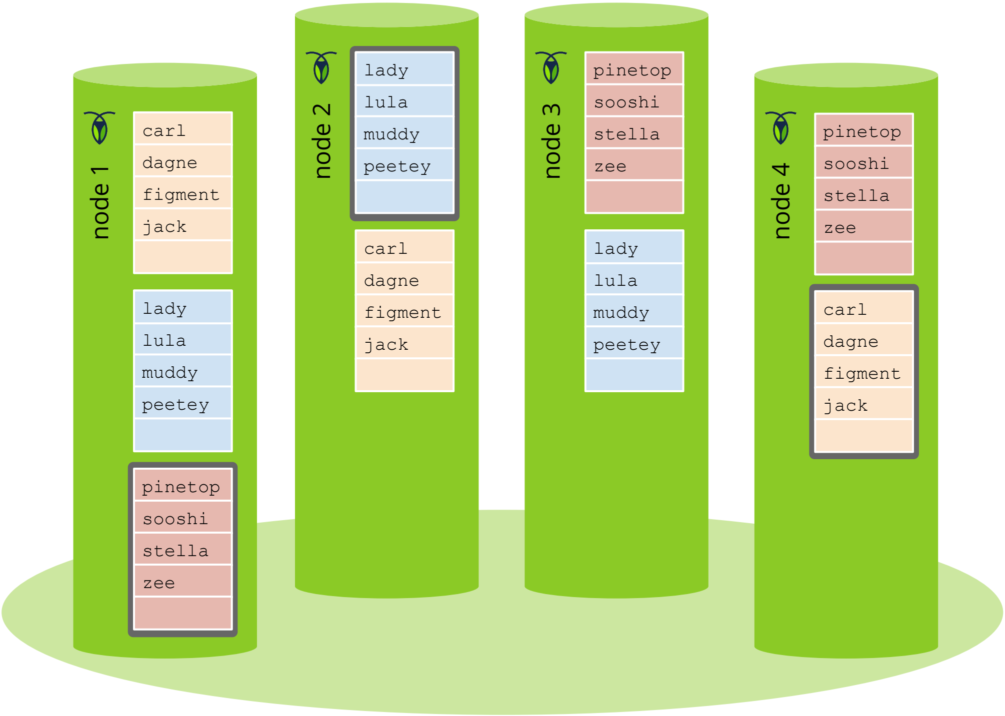
Raft provides atomic writes to individual ranges

Bootstrap transaction atomicity using Raft atomic writes

Transaction record atomically flipped from PENDING to COMMIT

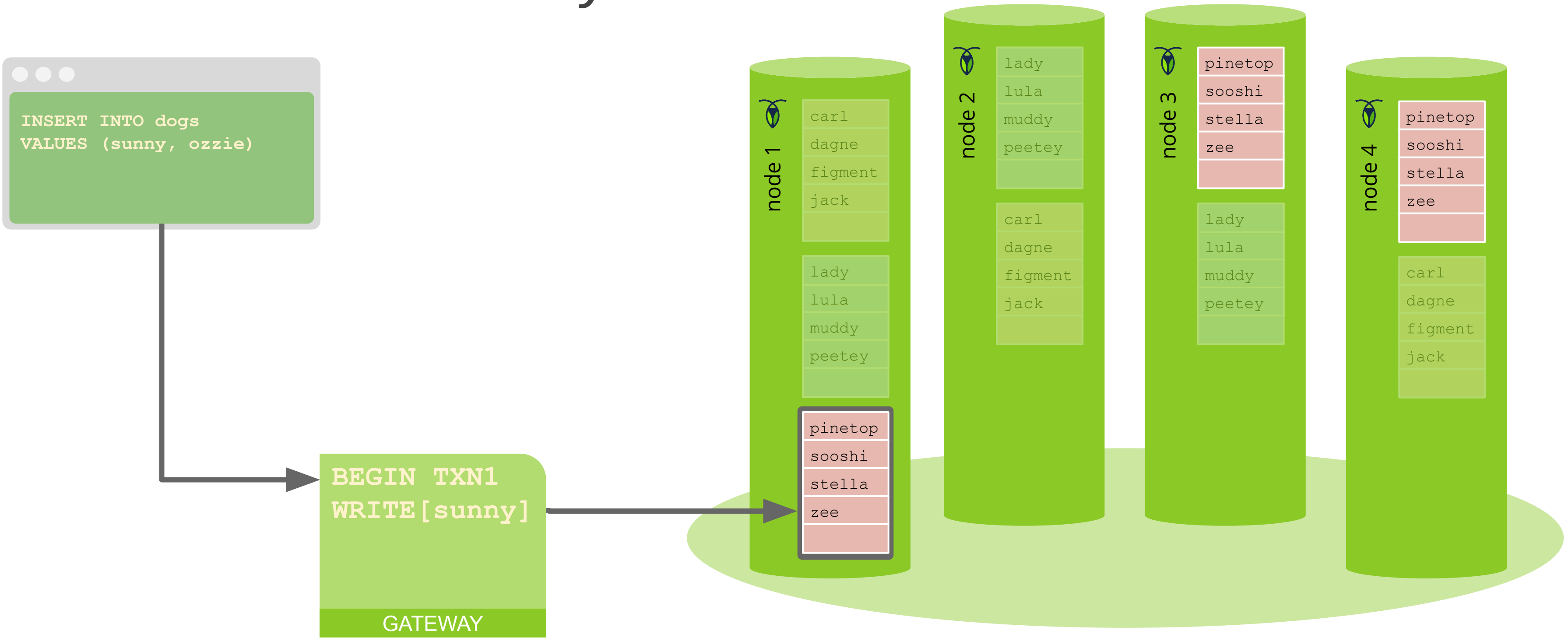
Transaction Atomicity

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



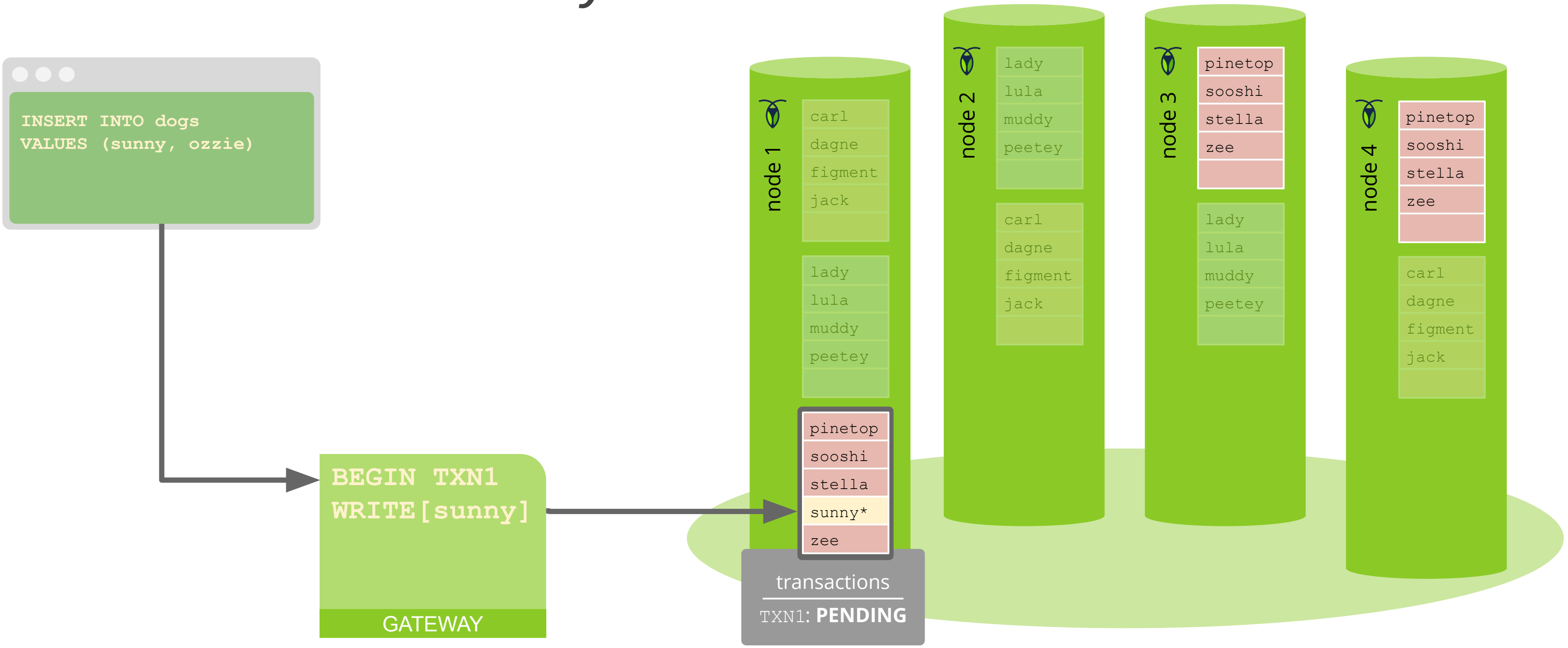
Cockroach LABS

Transaction Atomicity



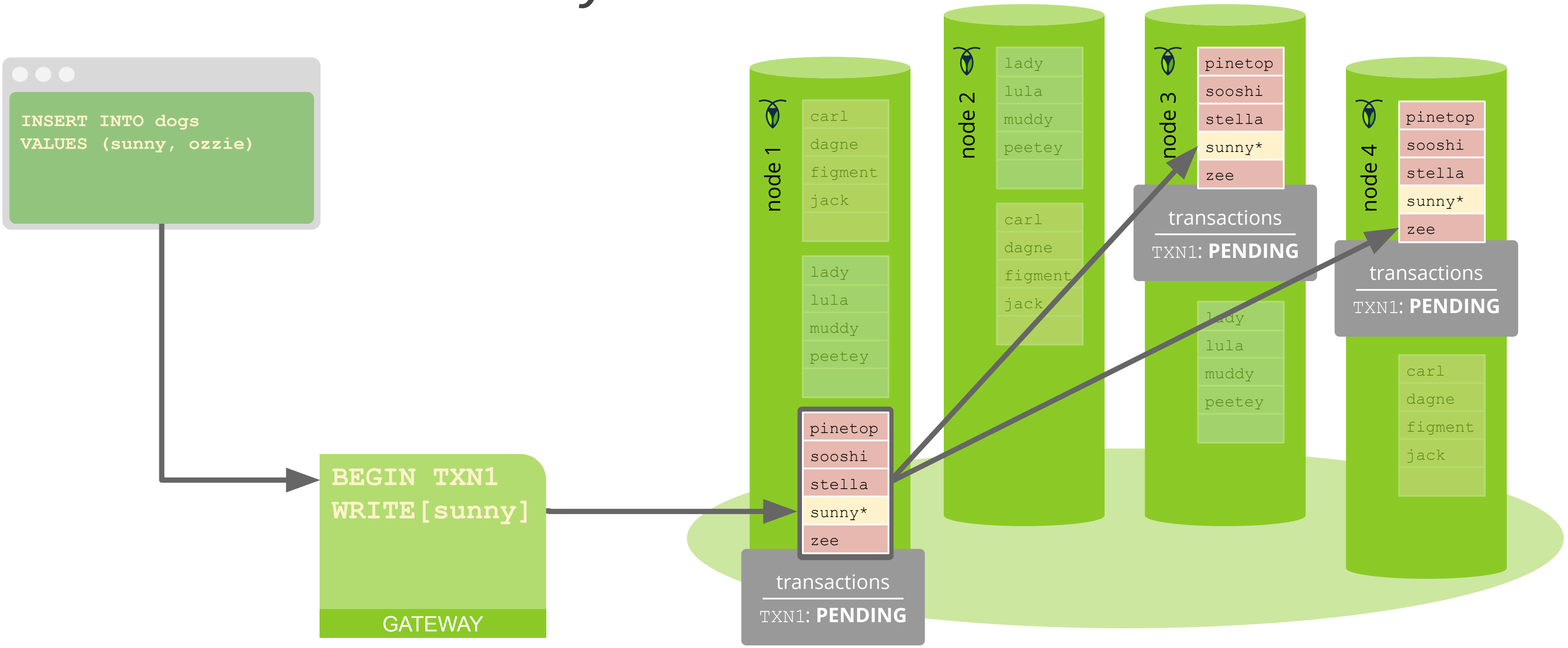
Cockroach LABS

Transaction Atomicity



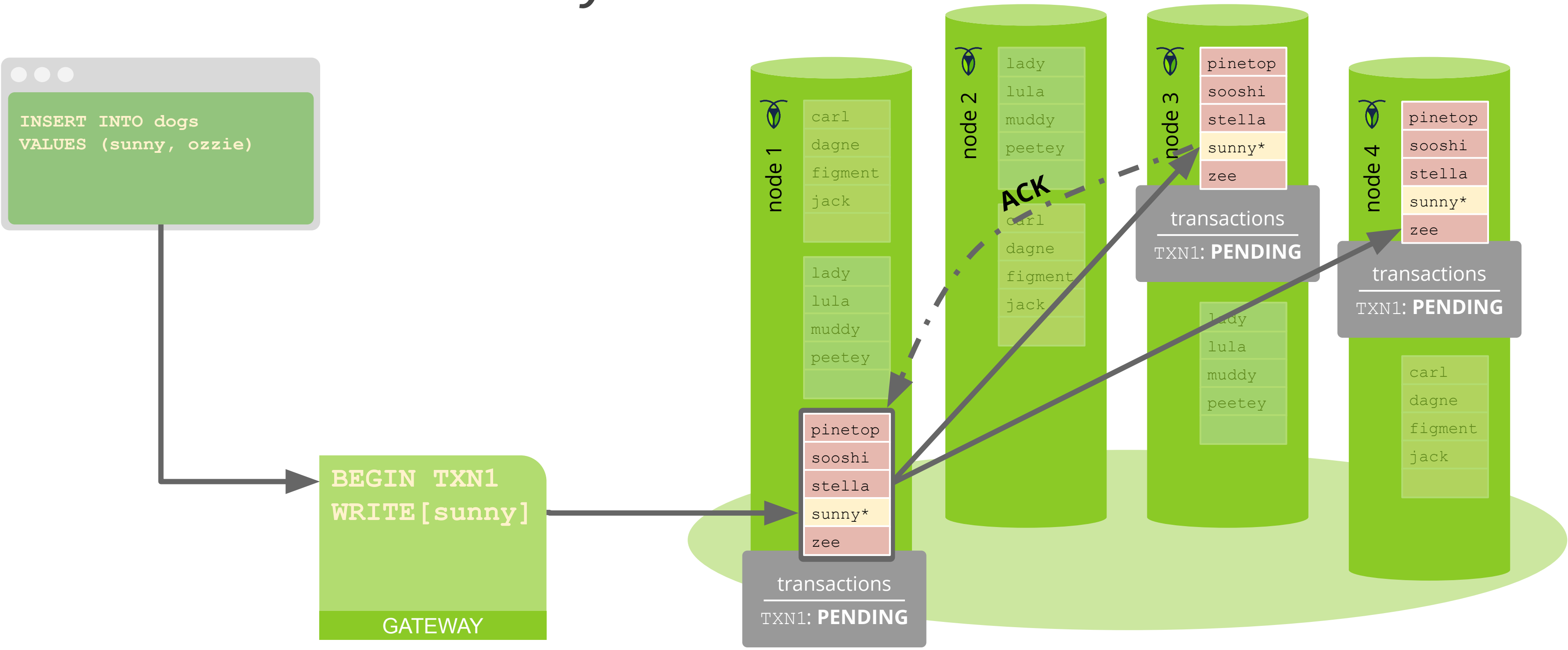
Cockroach LABS

Transaction Atomicity

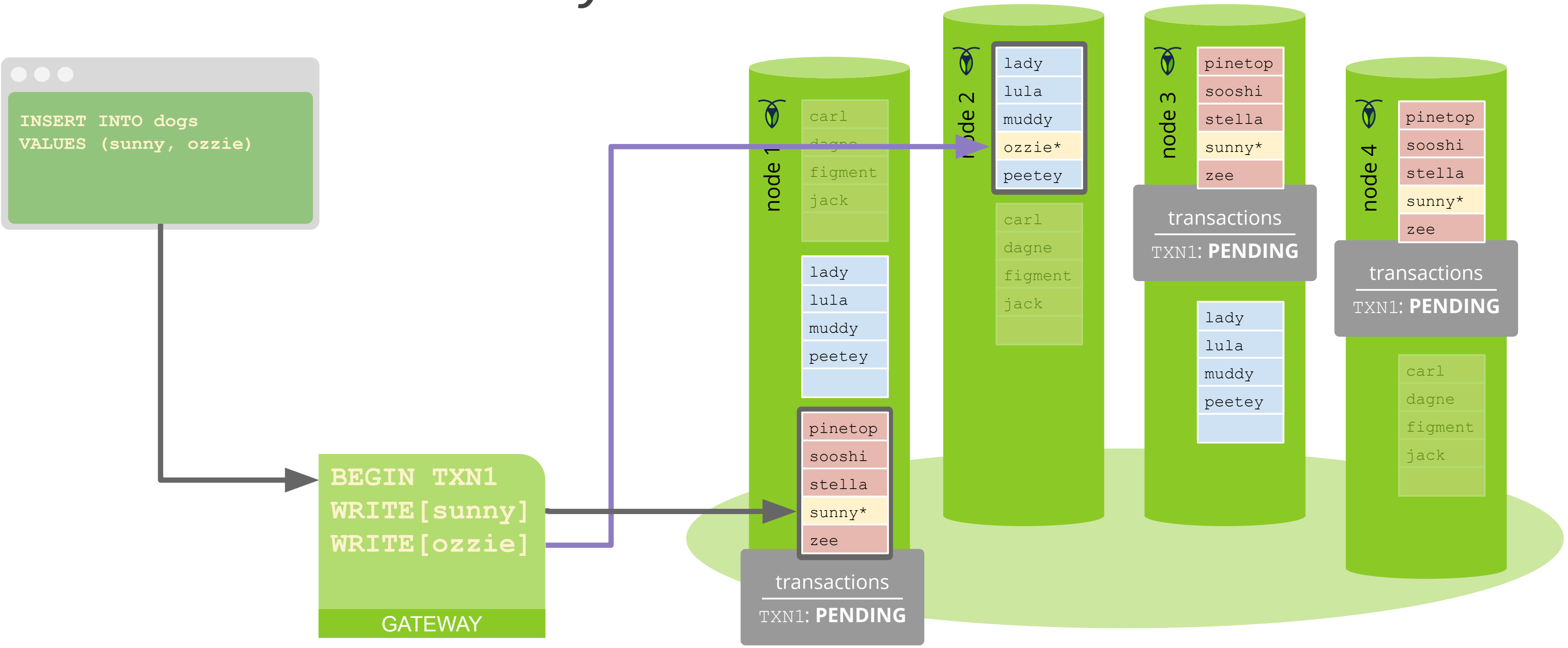


Cockroach LABS

Transaction Atomicity

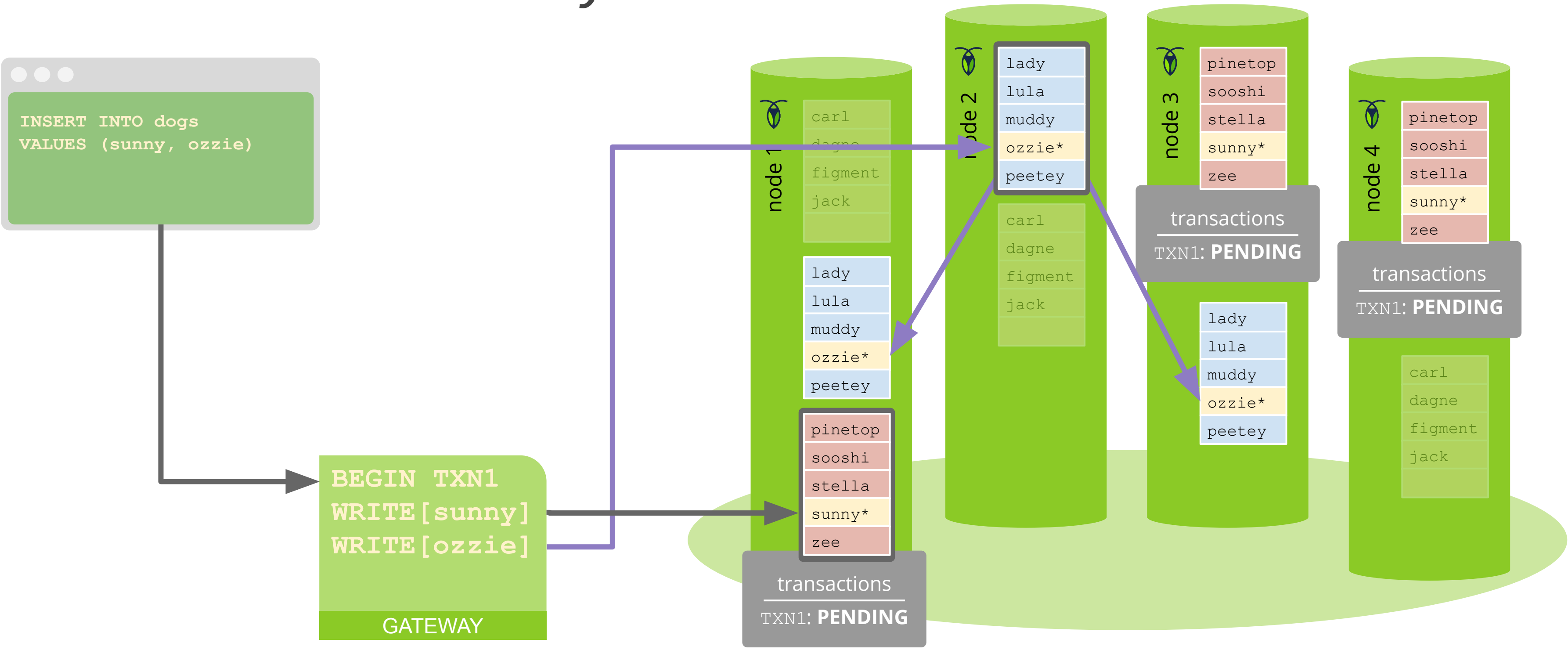


Transaction Atomicity

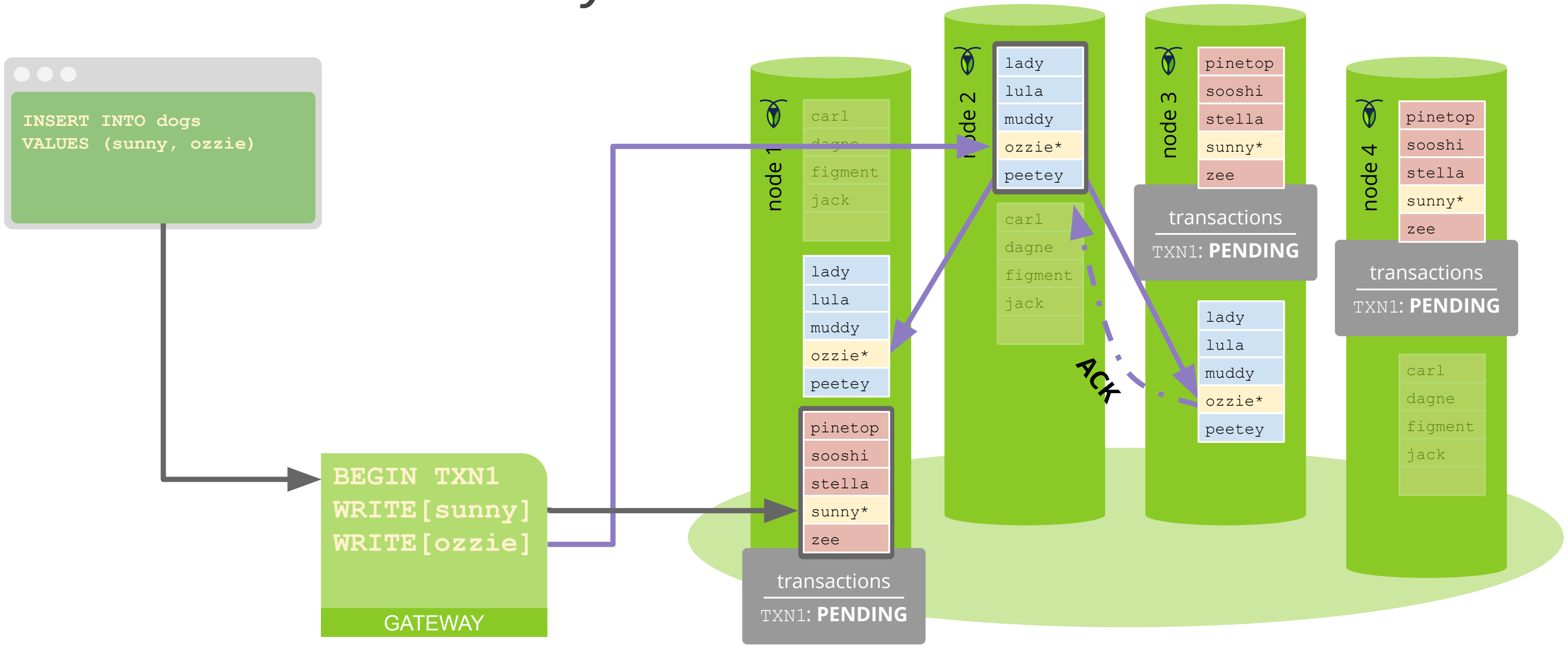


Cockroach LABS

Transaction Atomicity

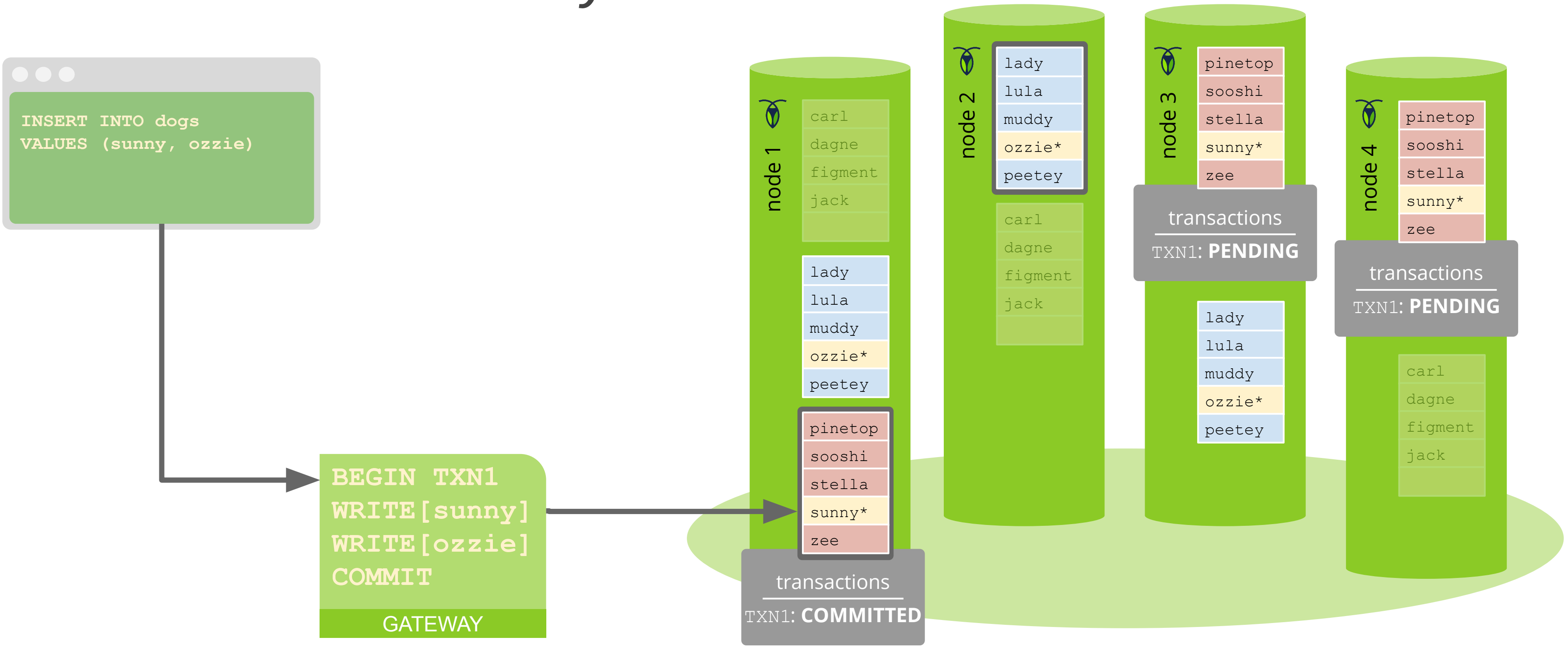


Transaction Atomicity



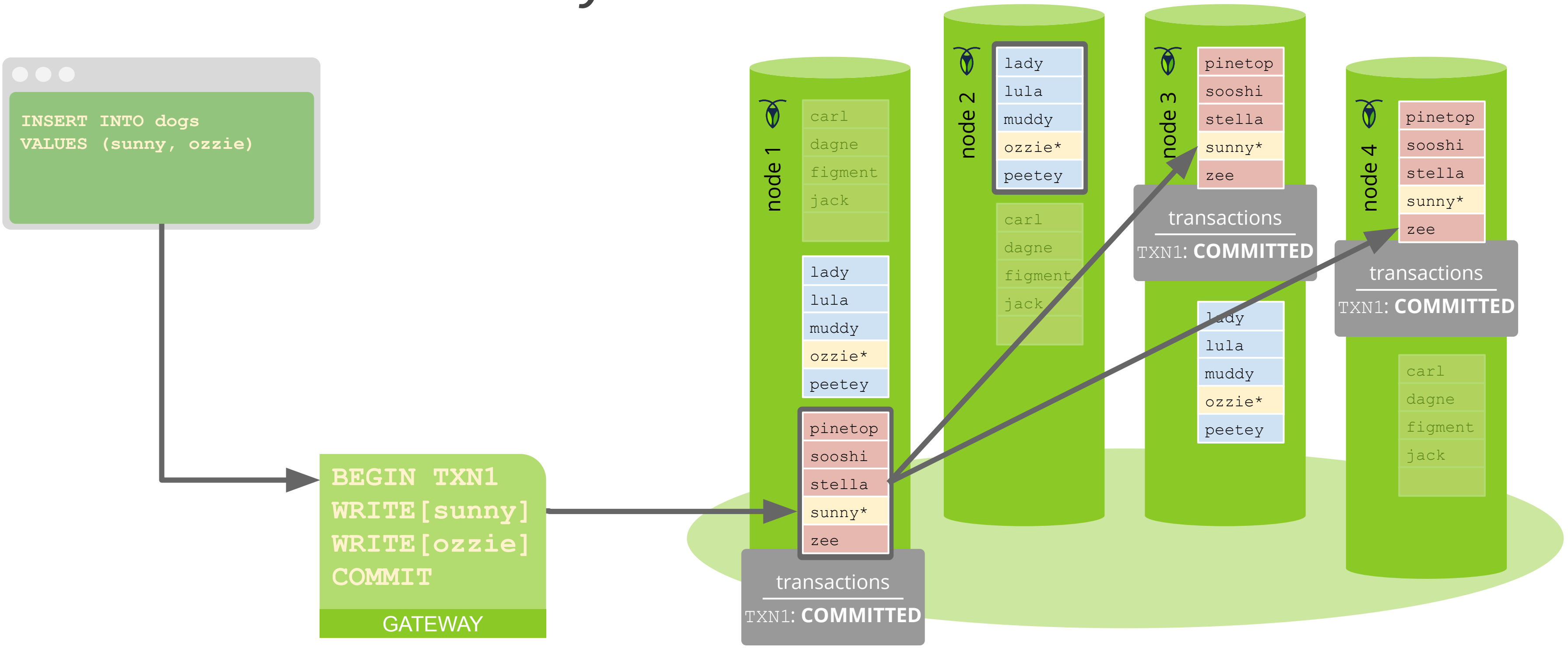
Cockroach LABS

Transaction Atomicity



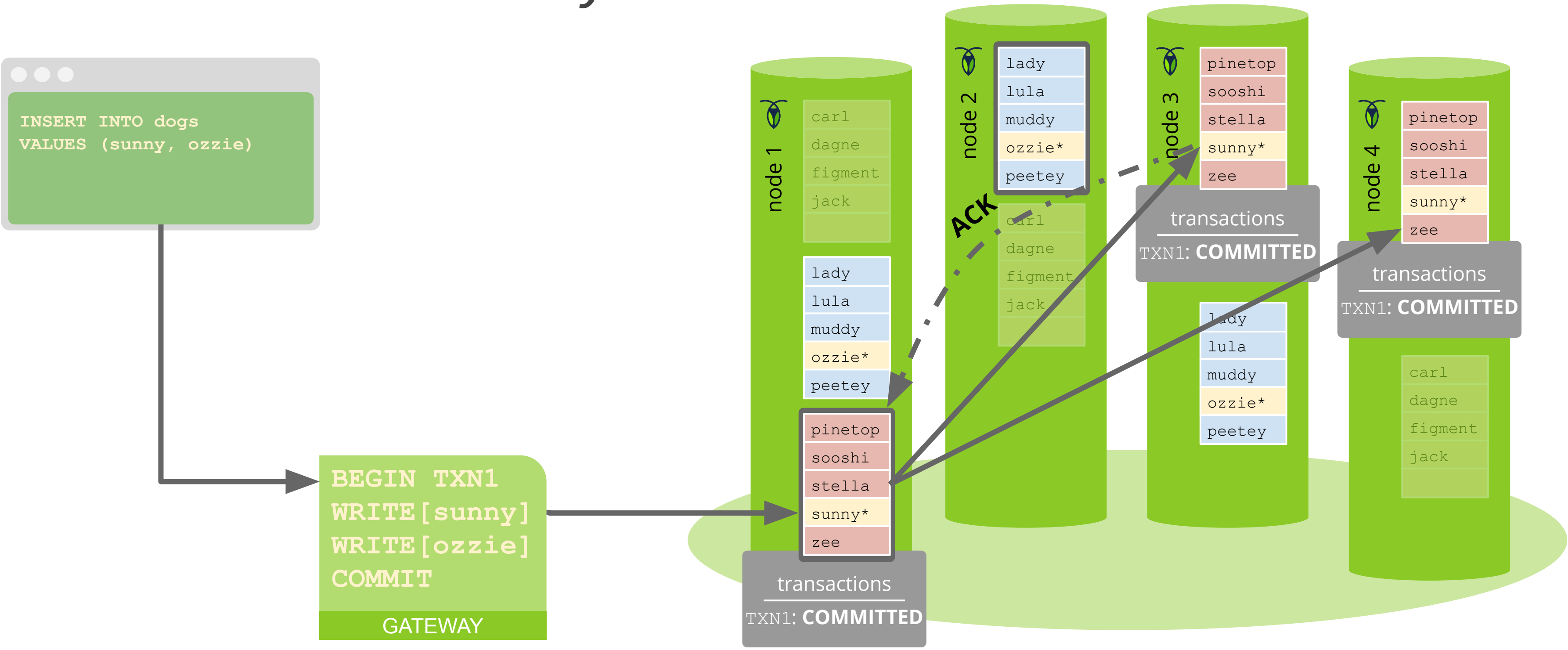
Cockroach LABS

Transaction Atomicity



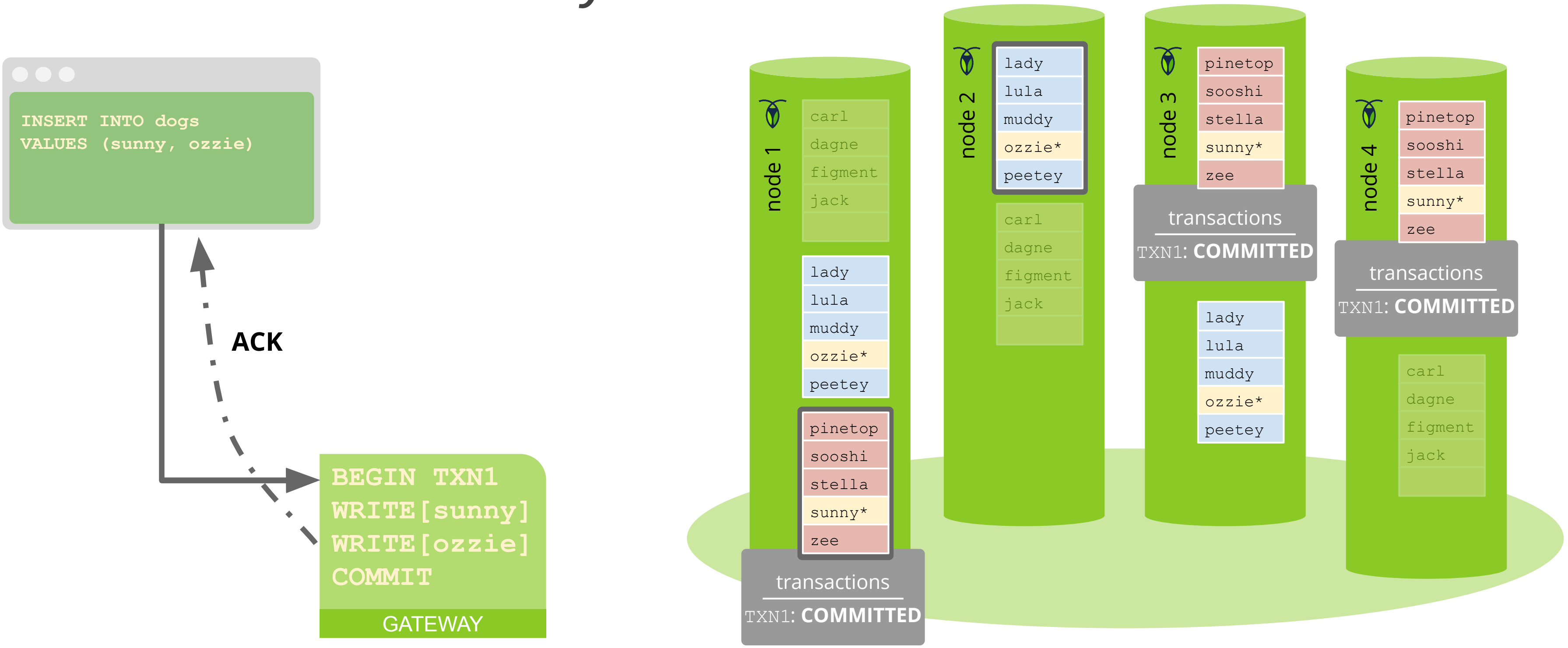
Cockroach LABS

Transaction Atomicity

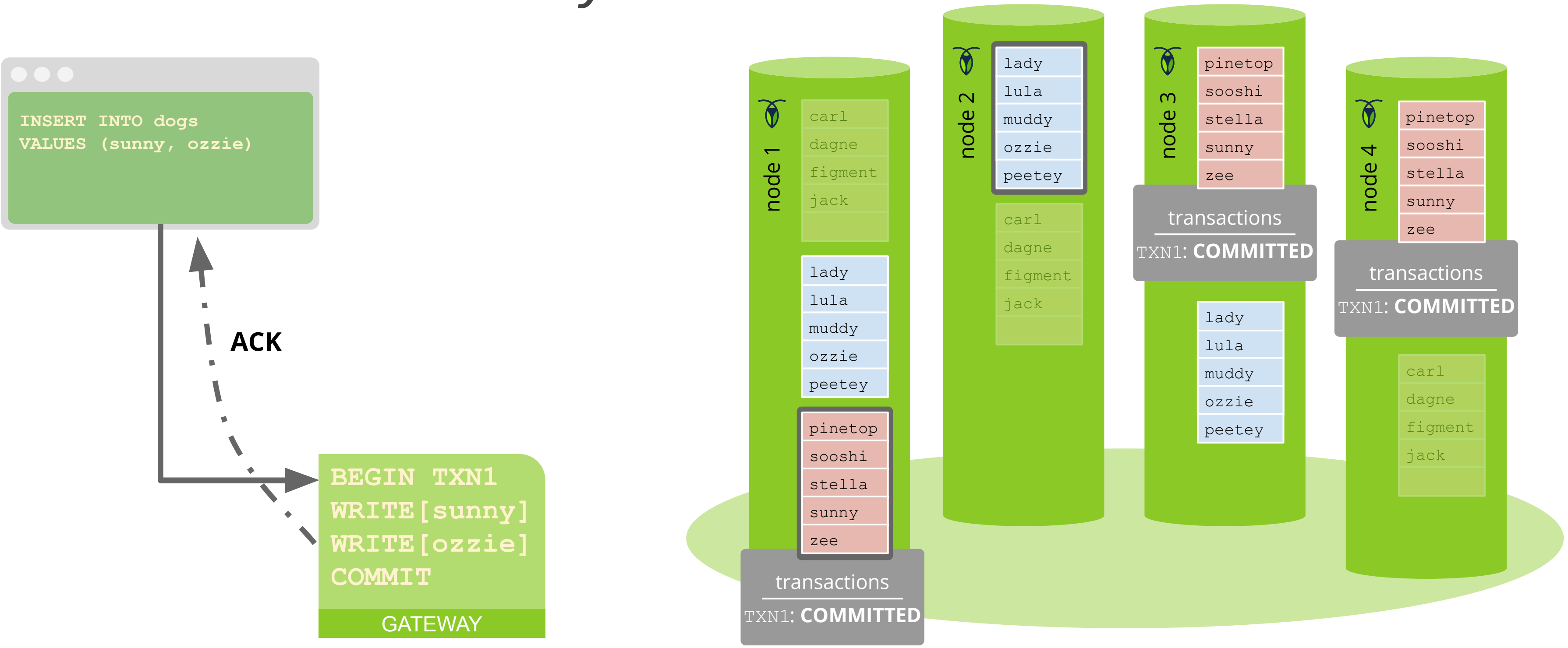


Cockroach LABS

Transaction Atomicity



Transaction Atomicity



Cockroach LABS

AGENDA

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

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 key-value store?

- The SQL data model needs to be **mapped** to KV data

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

AGENDA

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

- Relational expressions have input expressions and scalar expressions
  - The filter expression has 1 input expression and a predicate scalar expression
  - 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”
```

SQL Execution: Filter

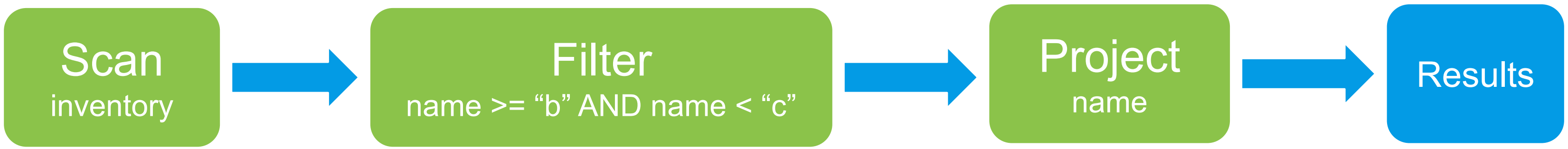
```
SELECT name
FROM   inventory
WHERE  name >= “b” AND name < “c”
```

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”
```

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

Distributed SQL Execution

Network latencies and throughput are important

Push fragments of computation **as close to the data as possible**

Leverage aggregate compute resources

Distributed SQL Execution: Streaming Group By

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

Scan customers | Scan customers | Scan customers

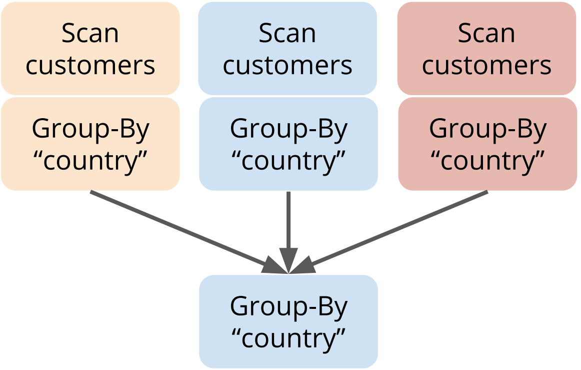
Distributed SQL Execution: Streaming Group By

```
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
```



Cockroach LABS

AGENDA

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

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
  - …
- 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

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

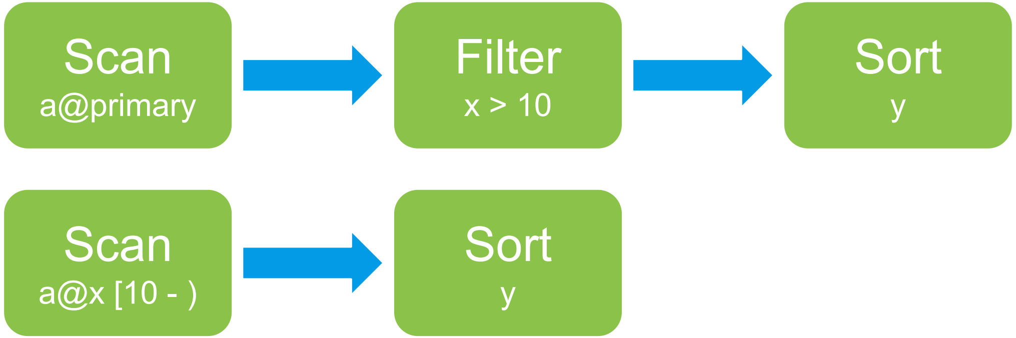
SQL Optimization: Cost-based Index Selection

```
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



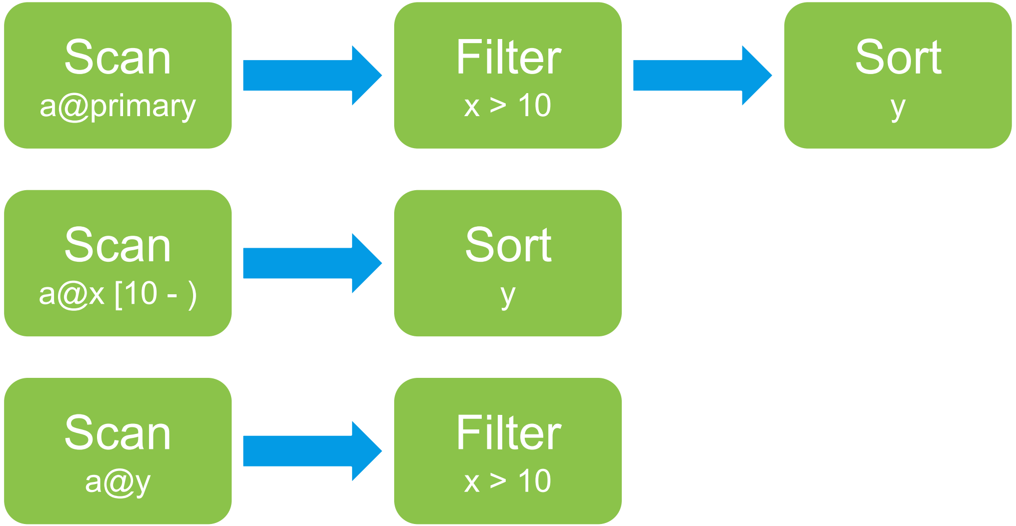
SQL Optimization: Cost-based Index Selection

```
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



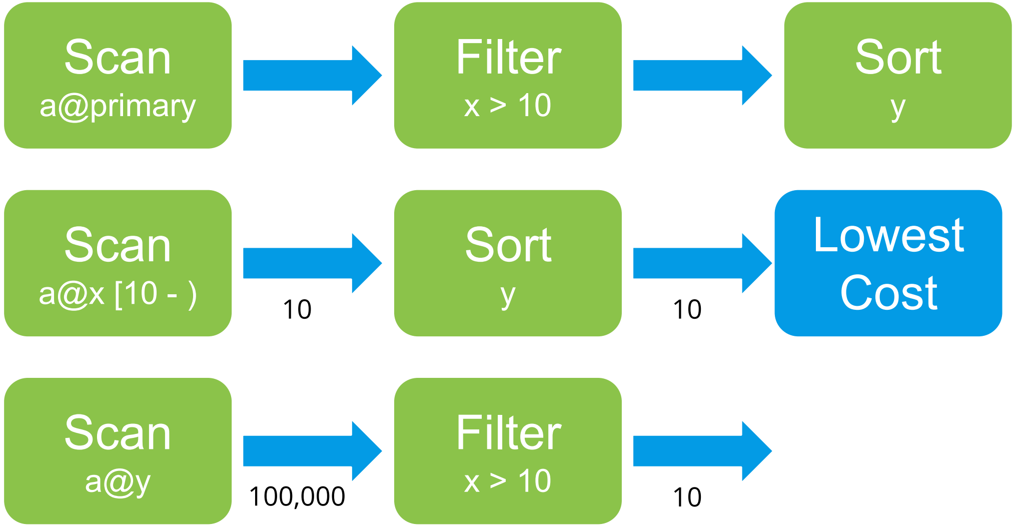
SQL Optimization: Cost-based Index Selection

```
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



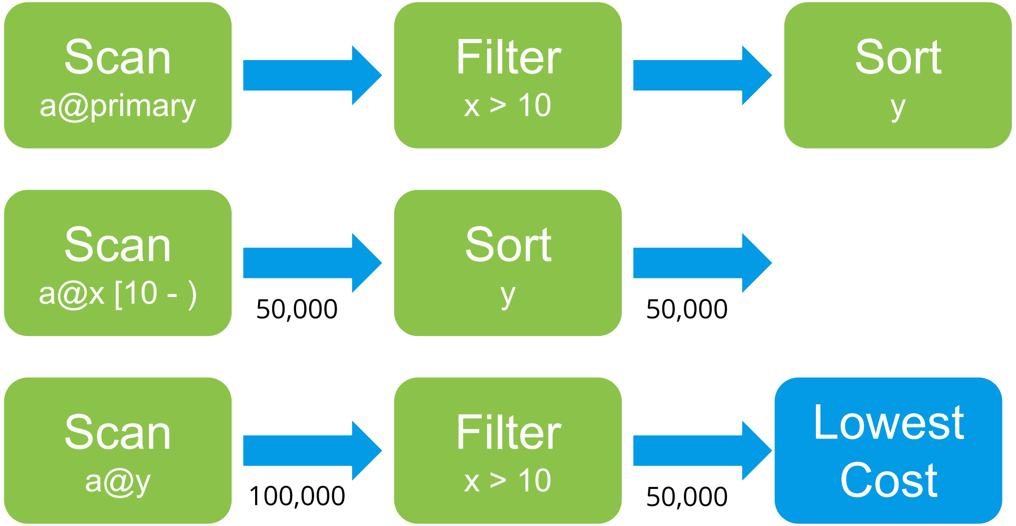
SQL Optimization: Cost-based Index Selection

```
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



Locality-Aware SQL Optimization

Network latencies and throughput are important

Duplicate read-mostly data in each locality

Plan queries to **use data from the same locality**

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)
)
```

Conclusion

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

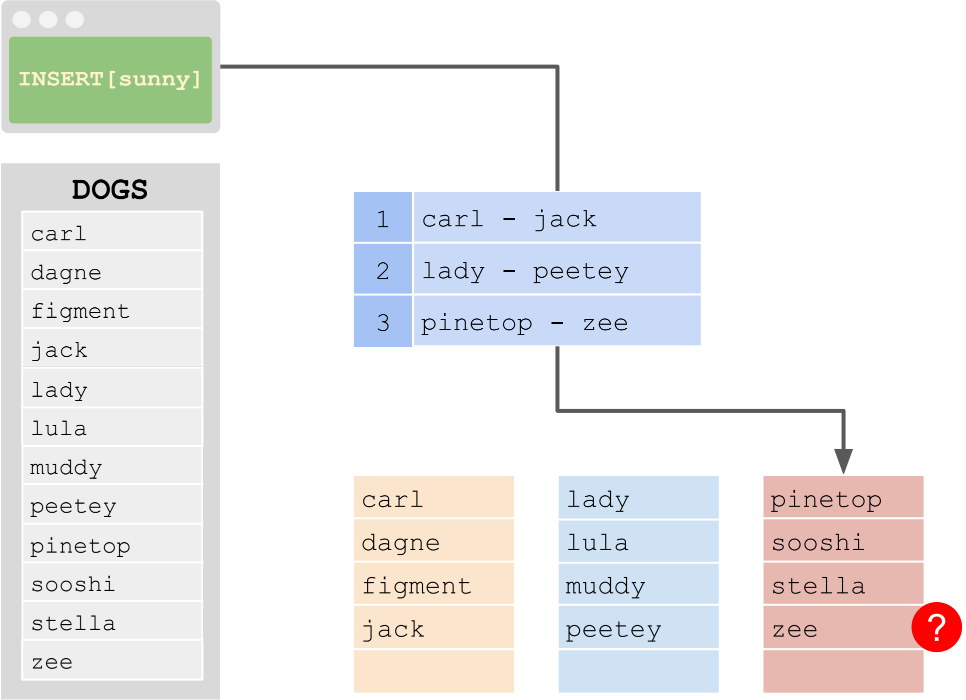
Cockroach LABS

Thank You

www.cockroachlabs.com

github.com/cockroachdb/cockroach

Transactional Updates



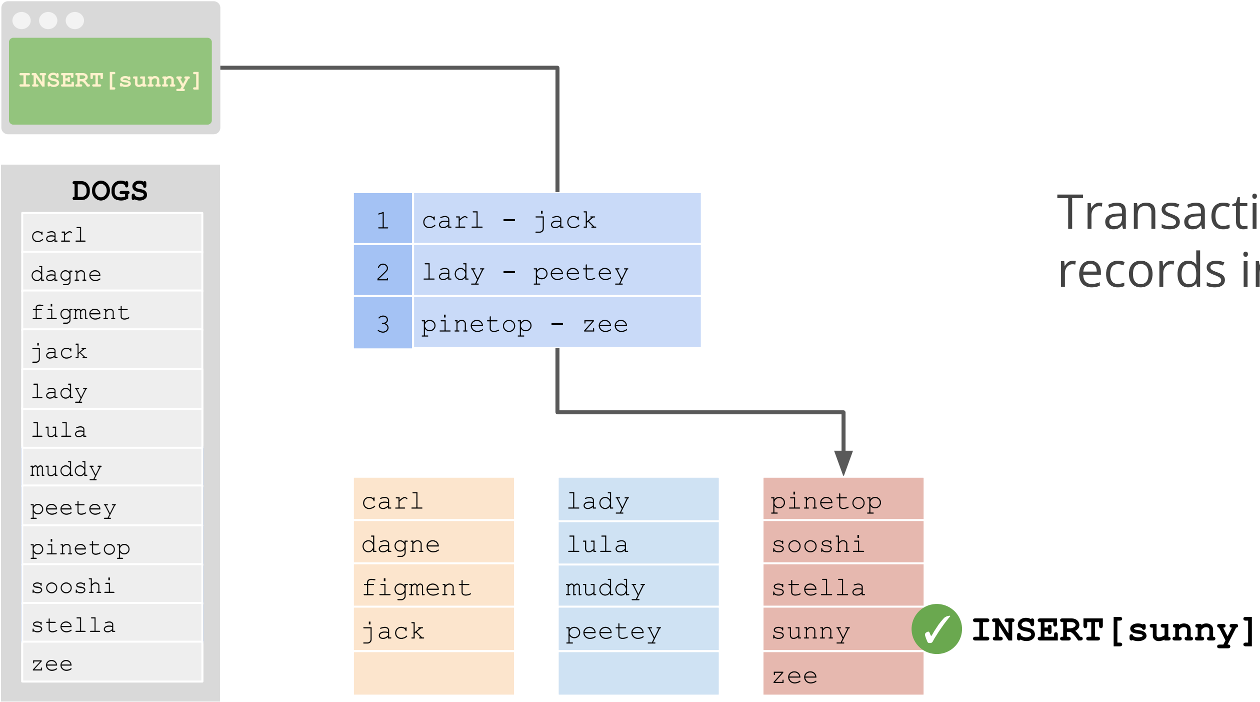
Transactions used to insert records into ranges

INSERT [sunny]

Space available in range? - **YES**

Cockroach LABS

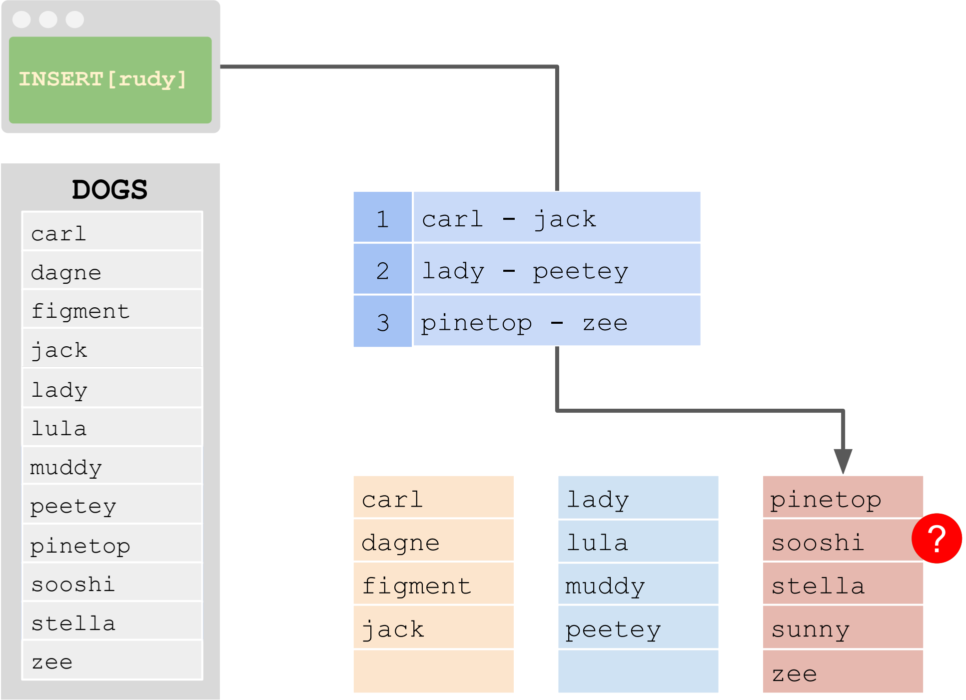
Transactional Updates



Transactions used to insert records into ranges

Cockroach LABS

Range Splits

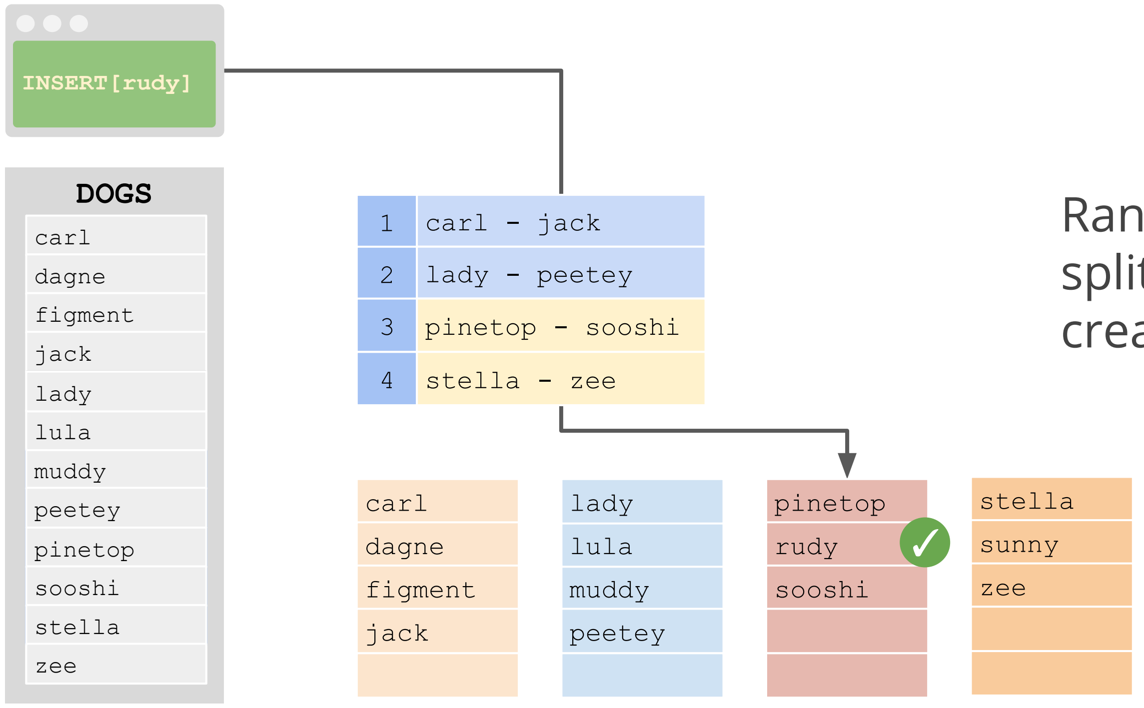


INSERT [rudy]

Space available in range? - **NO**

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

Range Splits



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

INSERT [rudy]
split range and insert

Rebalancing Replicas

Scale: Add a node

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

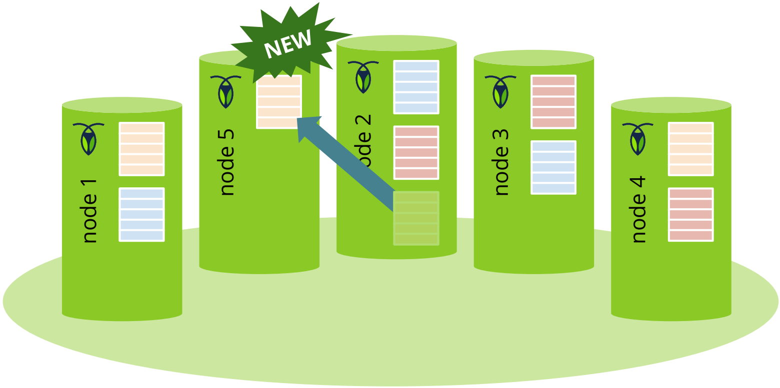
Uses the replica placement heuristics from previous slides

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

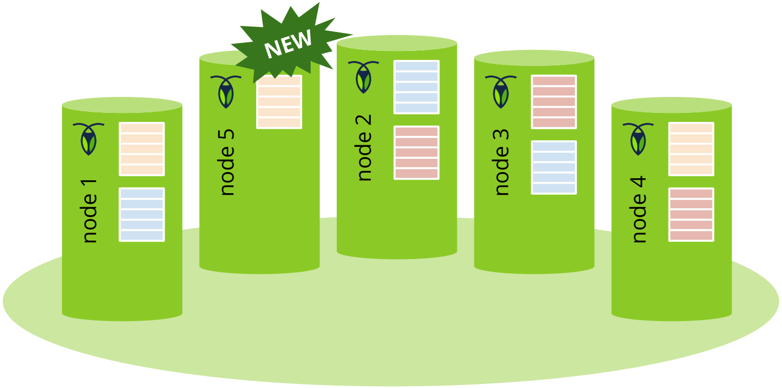
Cockroach LABS

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



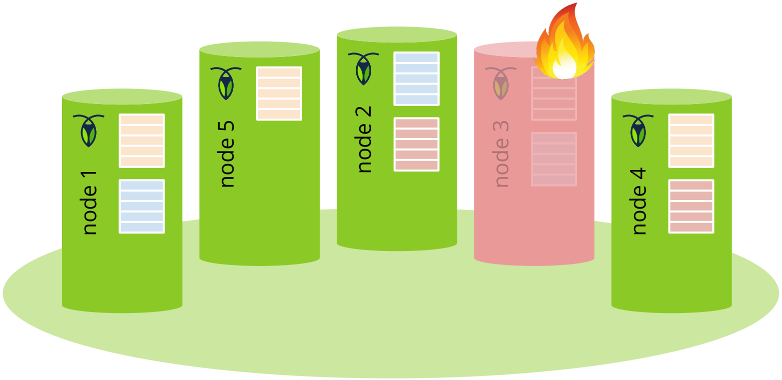
Movement is decomposed into adding a replica followed by removing a replica

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



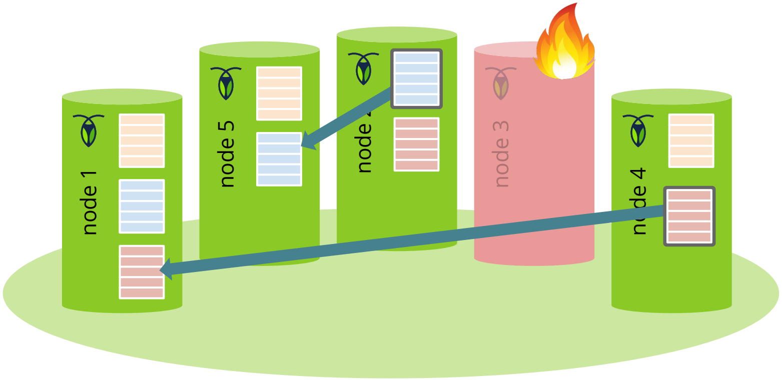
Cockroach LABS

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

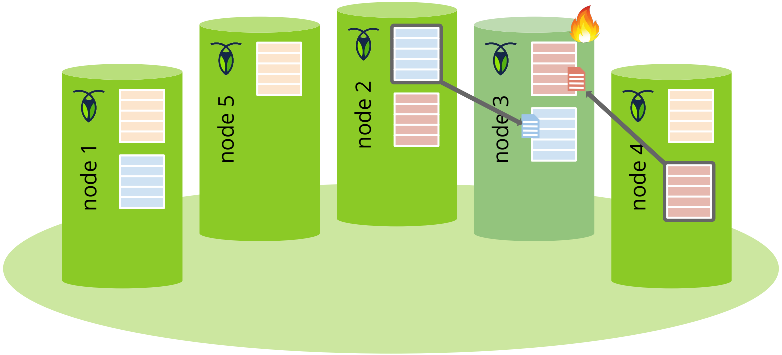
Uses the replica placement heuristics from previous slides



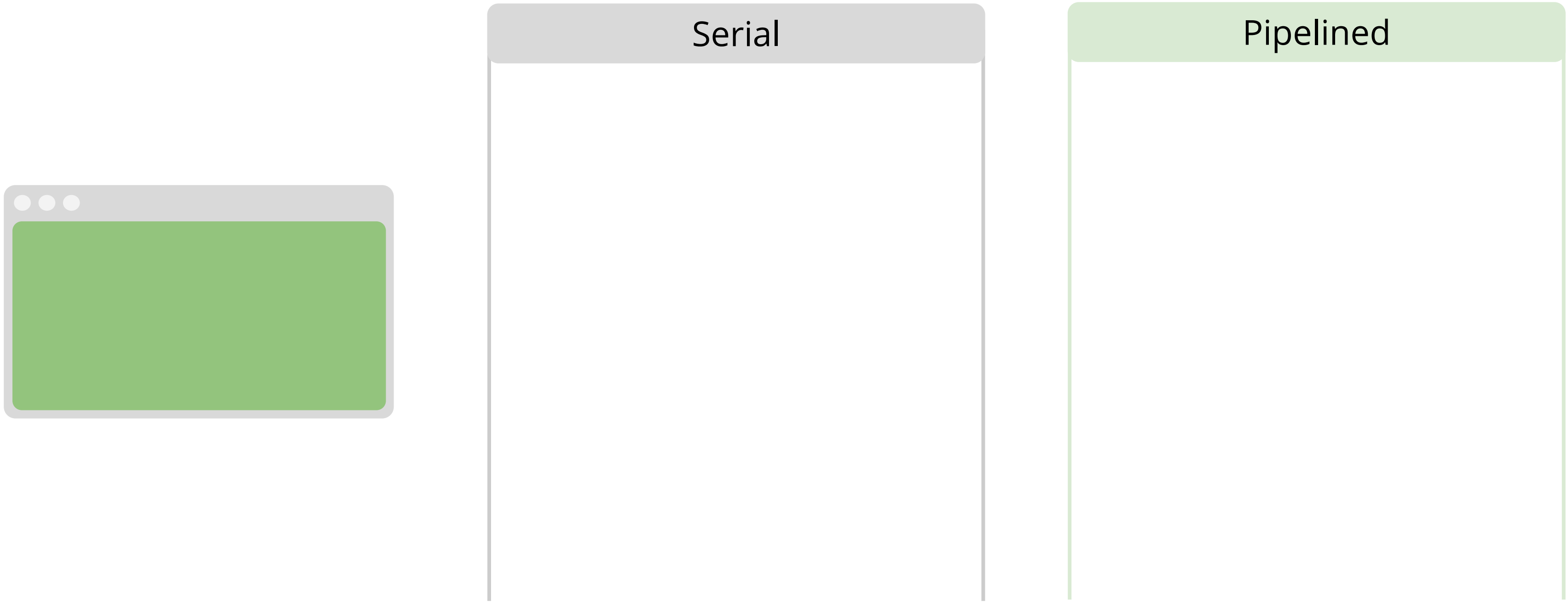
Rebalancing Replicas

Loss of a node
Temporary Failure

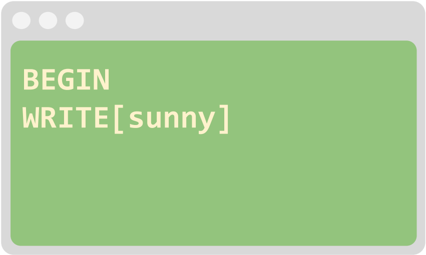
If a node goes down for a moment, the leaseholder can "catch up" any replica that is behind

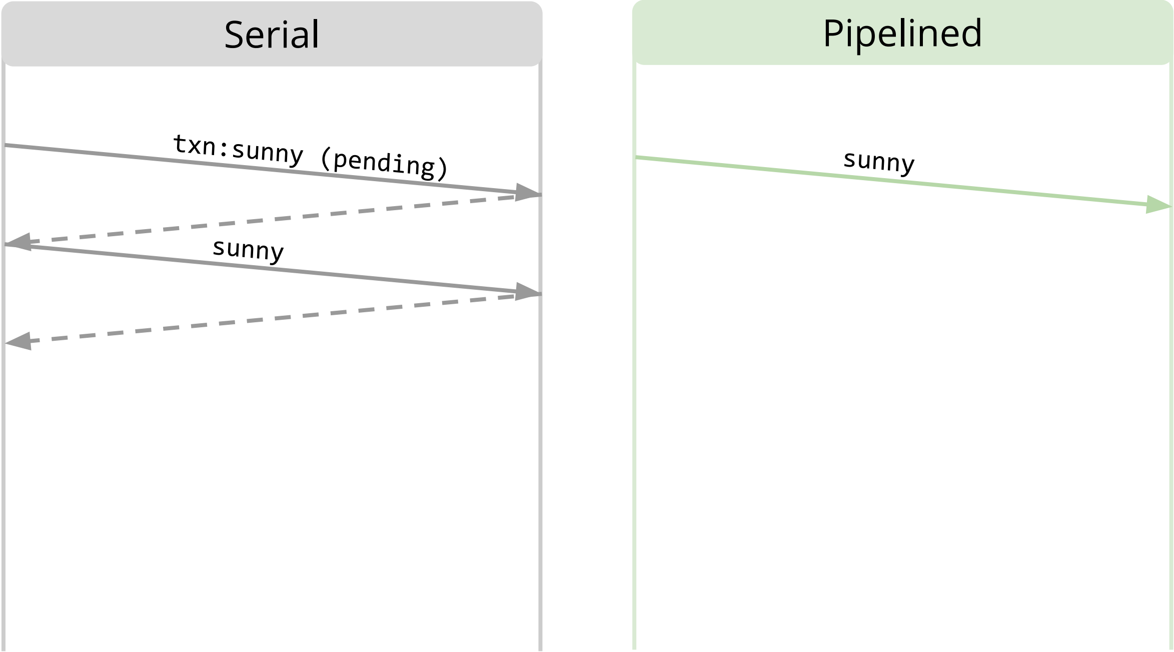


Transactions: Pipelining



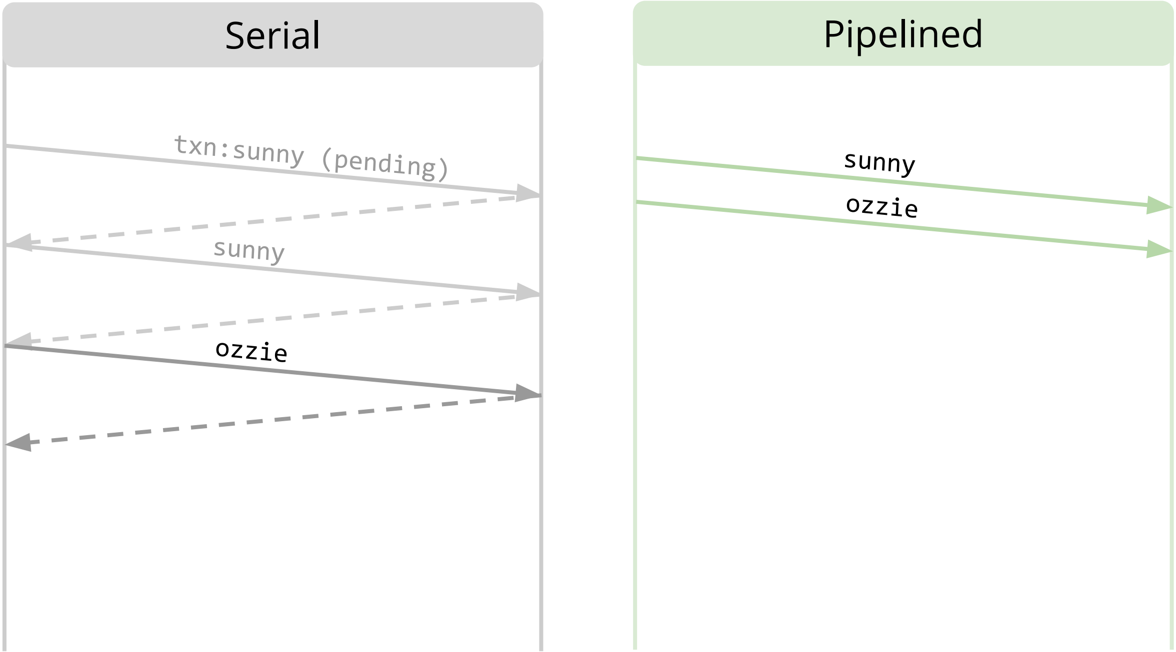
Transactions: Pipelining





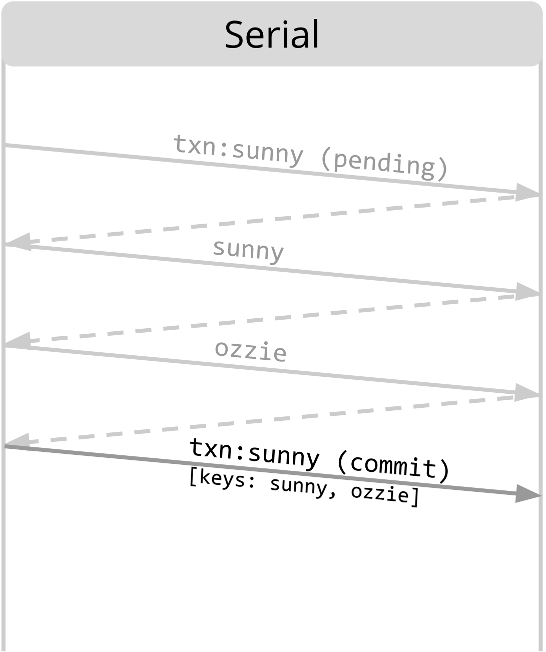
Transactions: Pipelining

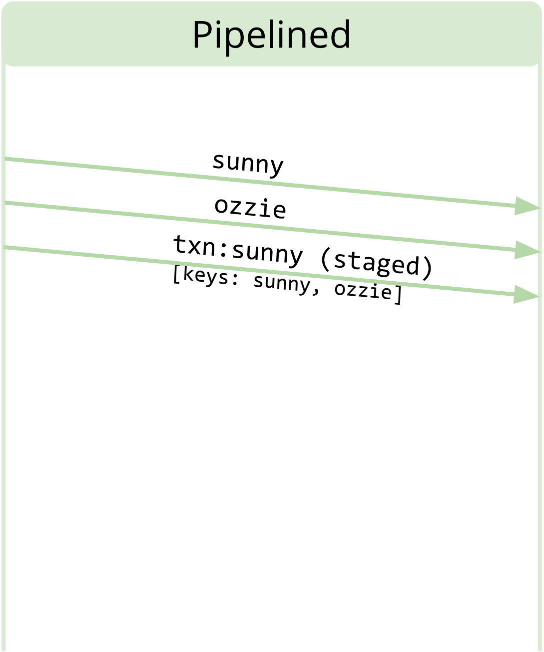
```
BEGIN
WRITE[sunny]
WRITE[ozzie]
```



Transactions: Pipelining

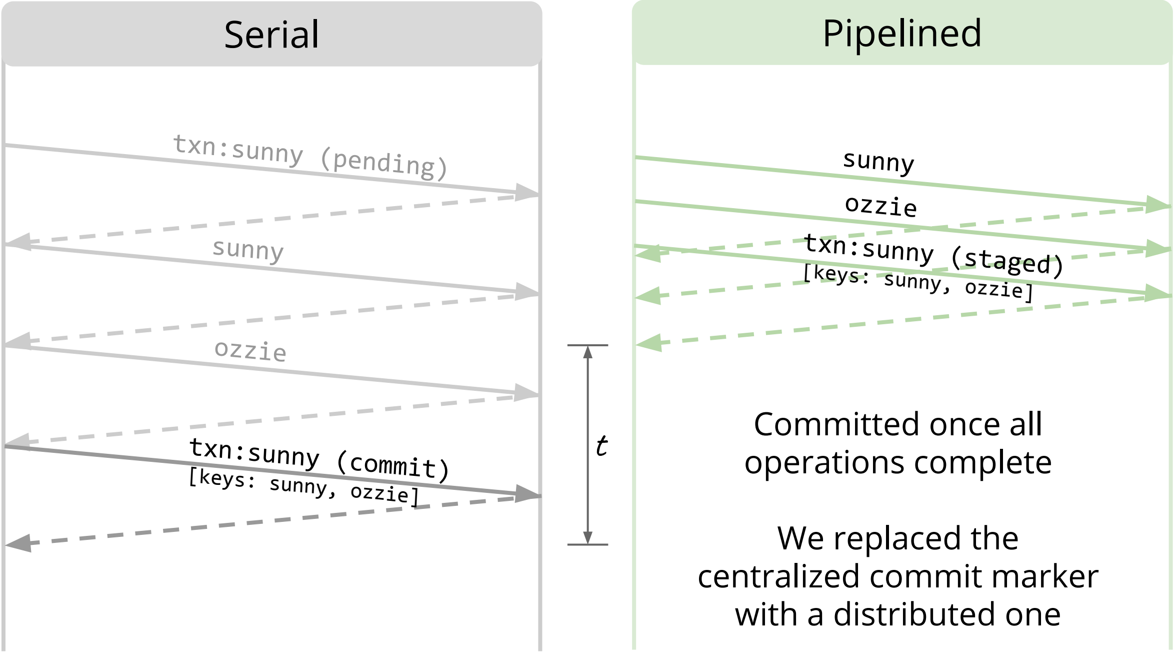
```
BEGIN
WRITE[sunny]
WRITE[ozzie]
COMMIT
```





Transactions: Pipelining

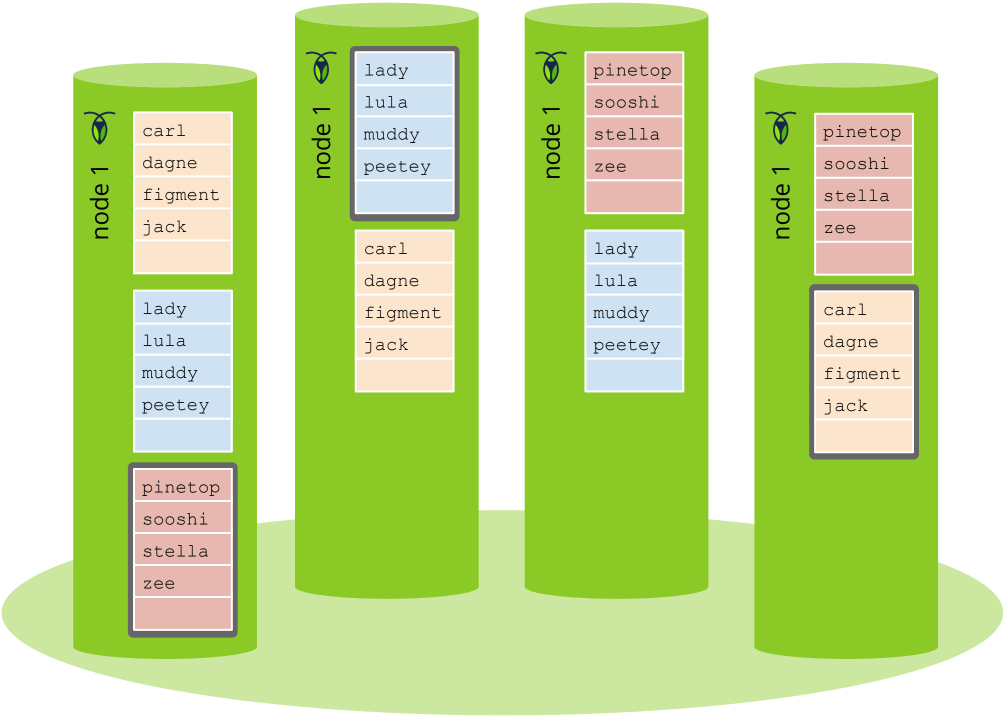
```
BEGIN
WRITE[sunny]
WRITE[ozzie]
COMMIT
```



* “Proved” with TLA+

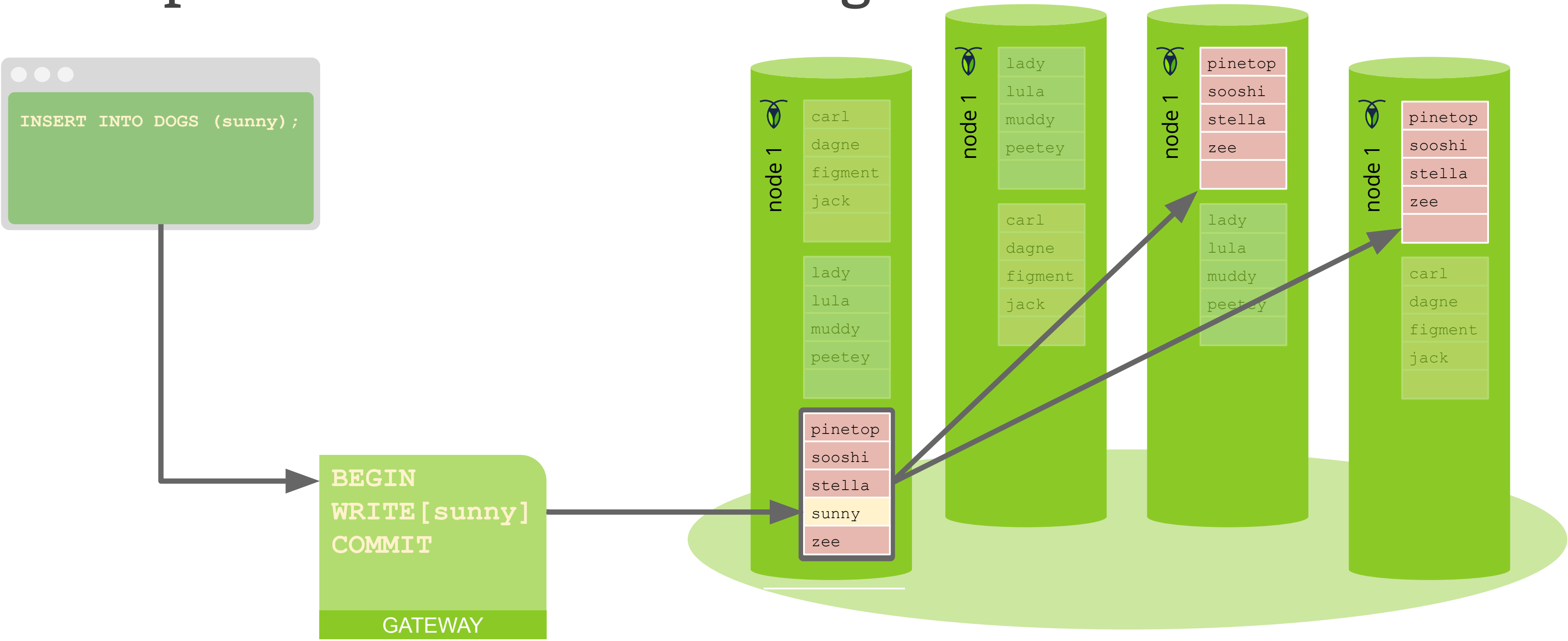
A Simple Transaction

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



Cockroach LABS

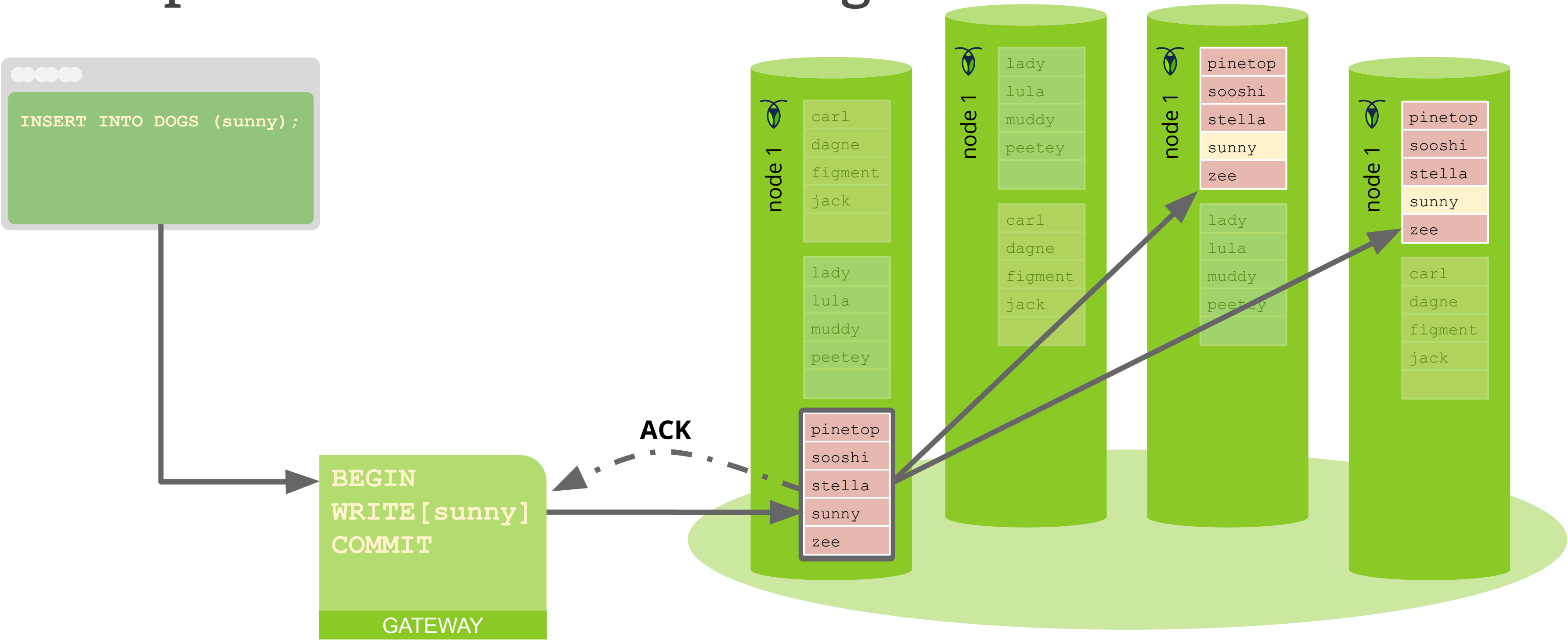
A Simple Transaction: One Range



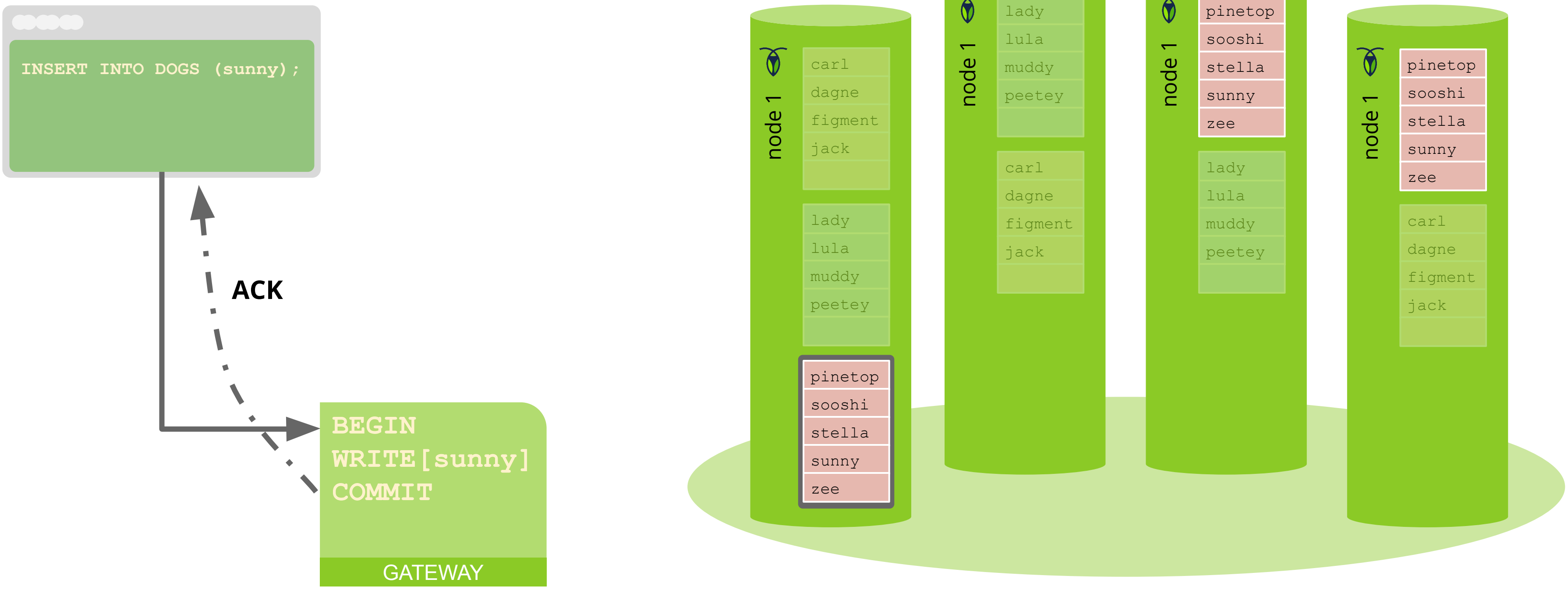
NOTE: a gateway can be ANY CockroachDB instance. It can find the leaseholder for any range and execute a transaction

Cockroach LABS

A Simple Transaction: One Range



A Simple Transaction: One Range



Cockroach LABS

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)

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	∅

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	∅

Cockroach LABS

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

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

Cockroach LABS

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

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

Cockroach LABS

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

Cockroach LABS

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

Cockroach LABS

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

Cockroach LABS

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**

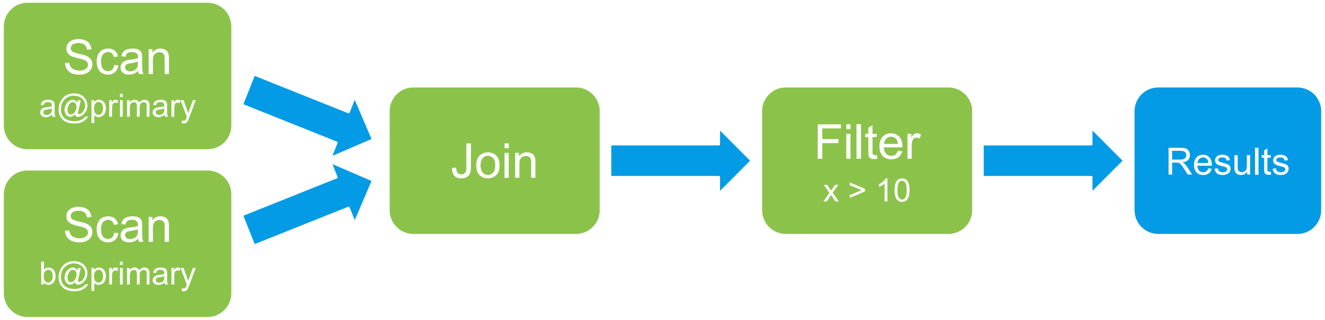
Name	**Country**
Jacques	France
Marie	France
Hans	Germany
Bob	United States
Susan	United States

Cockroach LABS

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