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

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

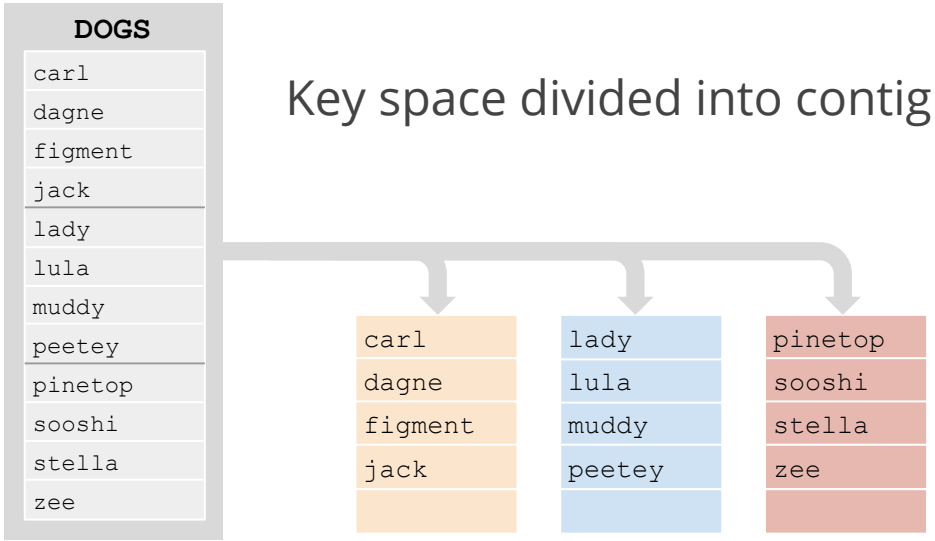
* Not exposed for external usage

Monolithic Key Space

Monolithic logical key space

- Ordered lexicographically by key

Ranges

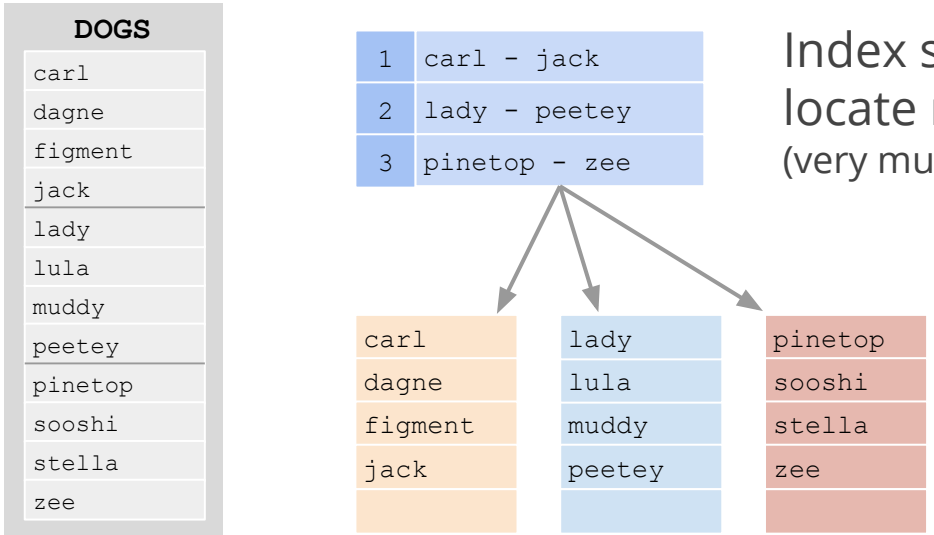


Key space divided into contiguous ~64MB ranges

Ranges are small enough to be moved/split quickly

Ranges are large enough to amortize indexing overhead

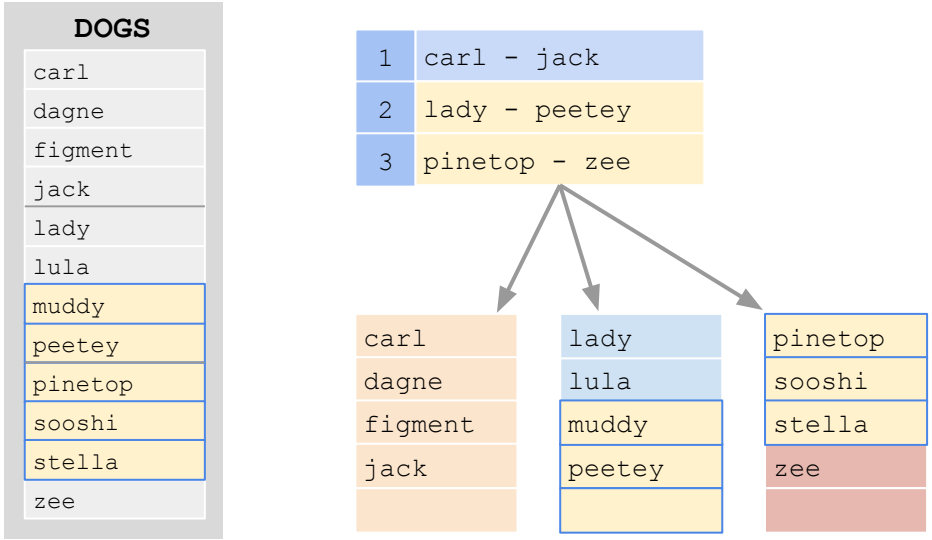
Range Indexing



Index structure used to locate ranges

(very much like a B-tree)

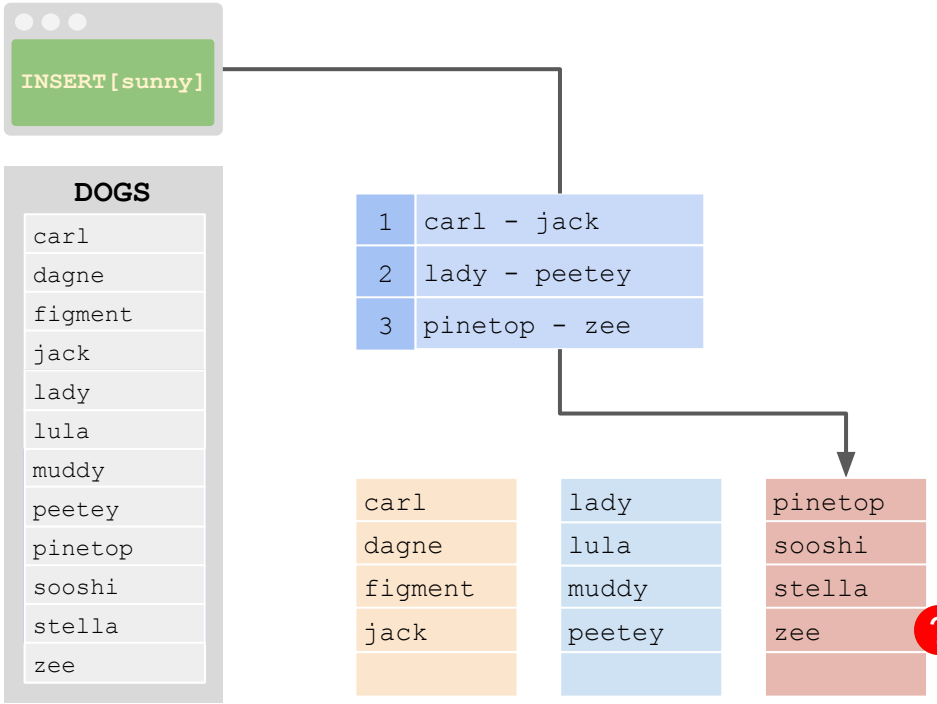
Ordered Range Scans



Ordered keys enable efficient range scans

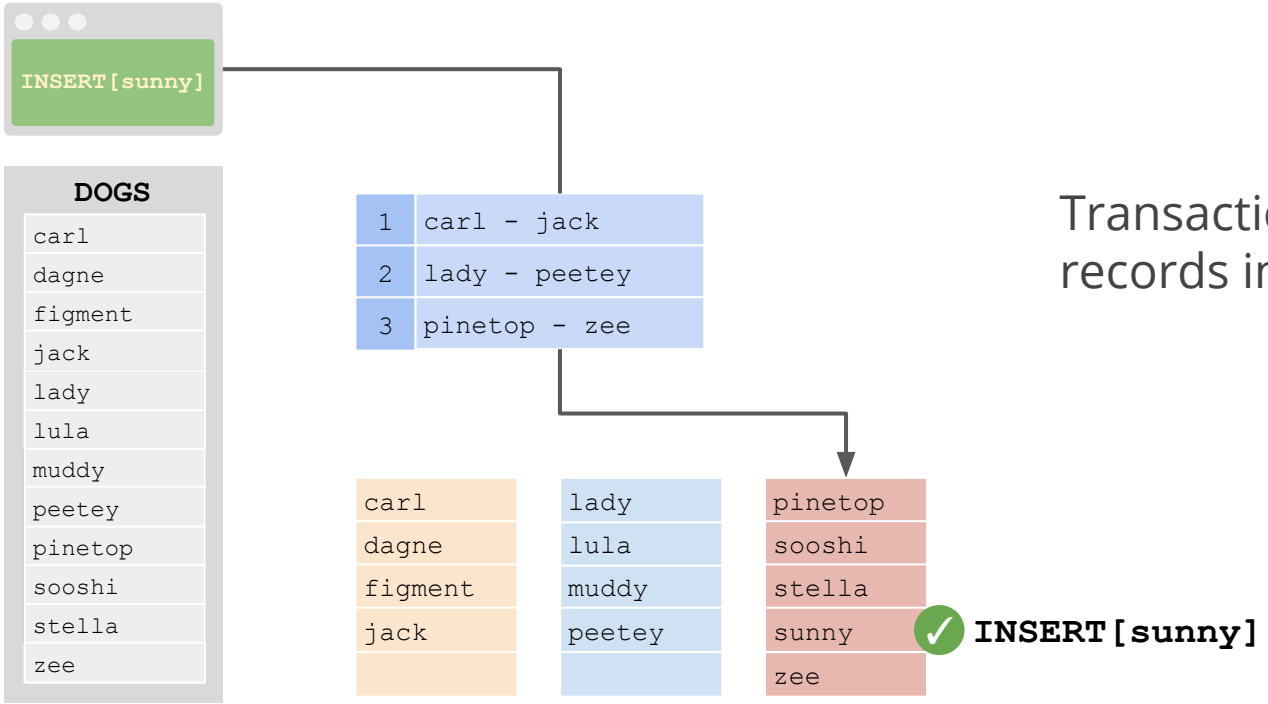
```
dogs >= "muddy" AND <= "stella"
```

Transactional Updates



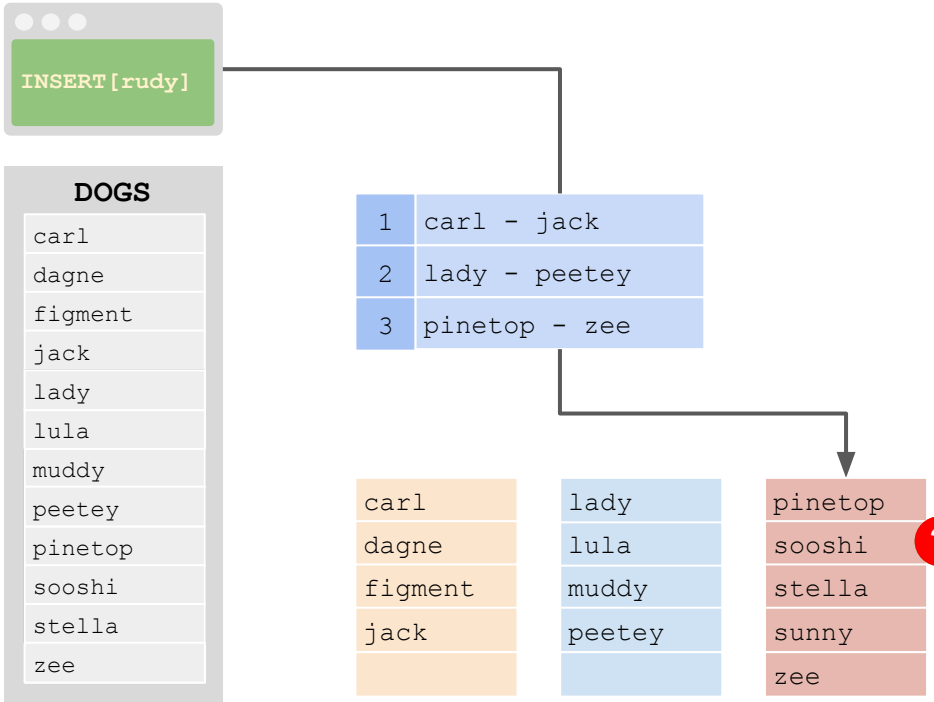
Transactions used to insert records into ranges

Transactional Updates



Transactions used to insert records into ranges

Range Splits



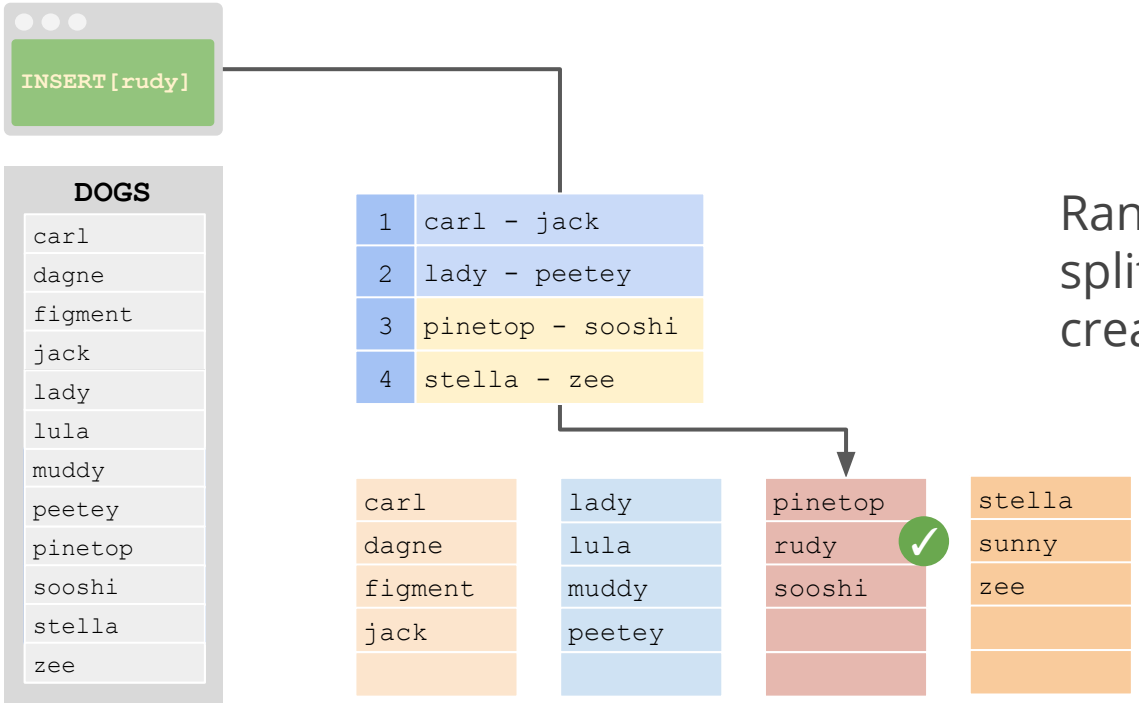
BUT... what happens when a range is full?

? **INSERT [rudy]**

Space available in range? - **NO**

Cockroach LABS

Range Splits



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

INSERT[rudy]
split range and insert

Raft and Replication

Ranges (~64MB) are the unit of replication

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

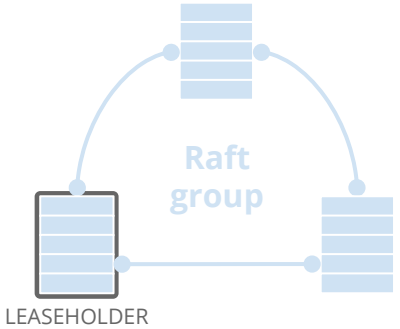
Default to 3 replicas, though this is configurable

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

Raft and Replication

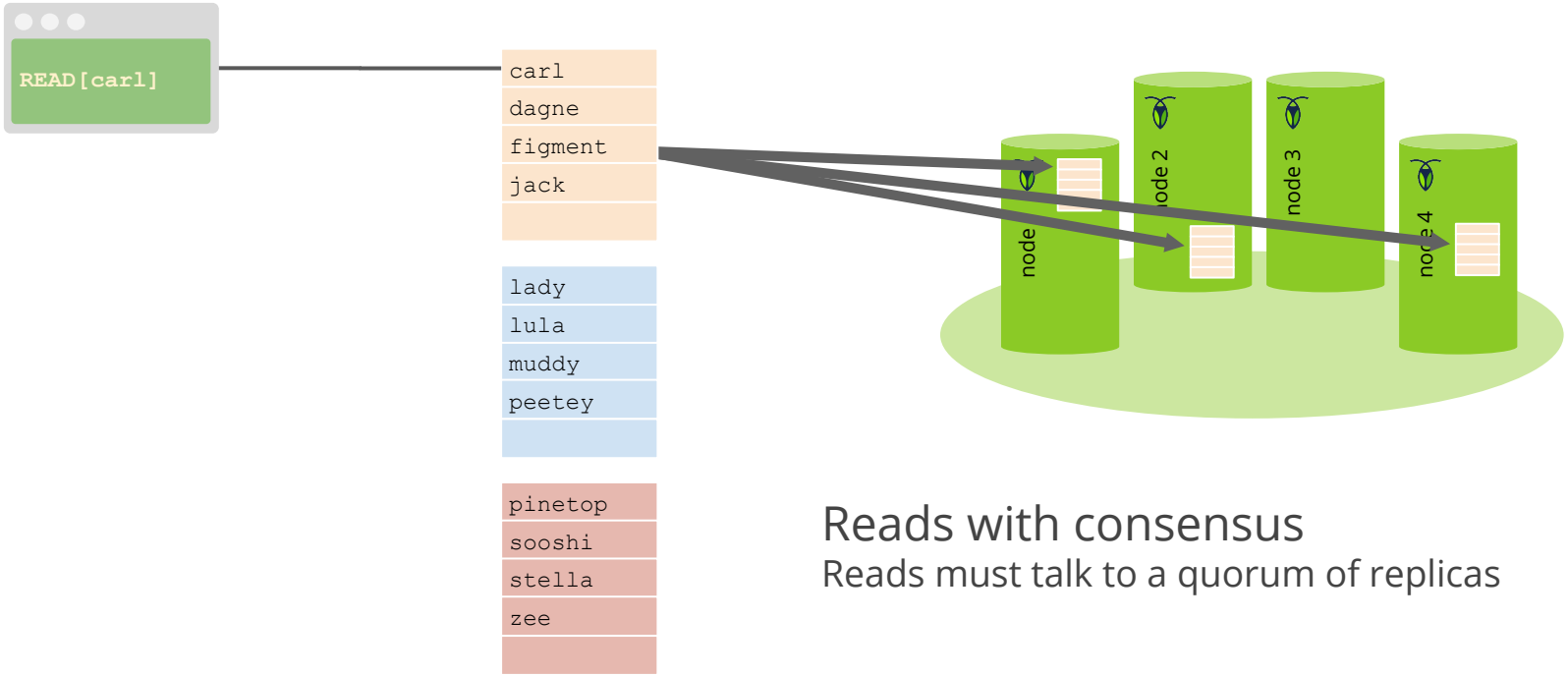
Raft provides “atomic replication” of commands

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

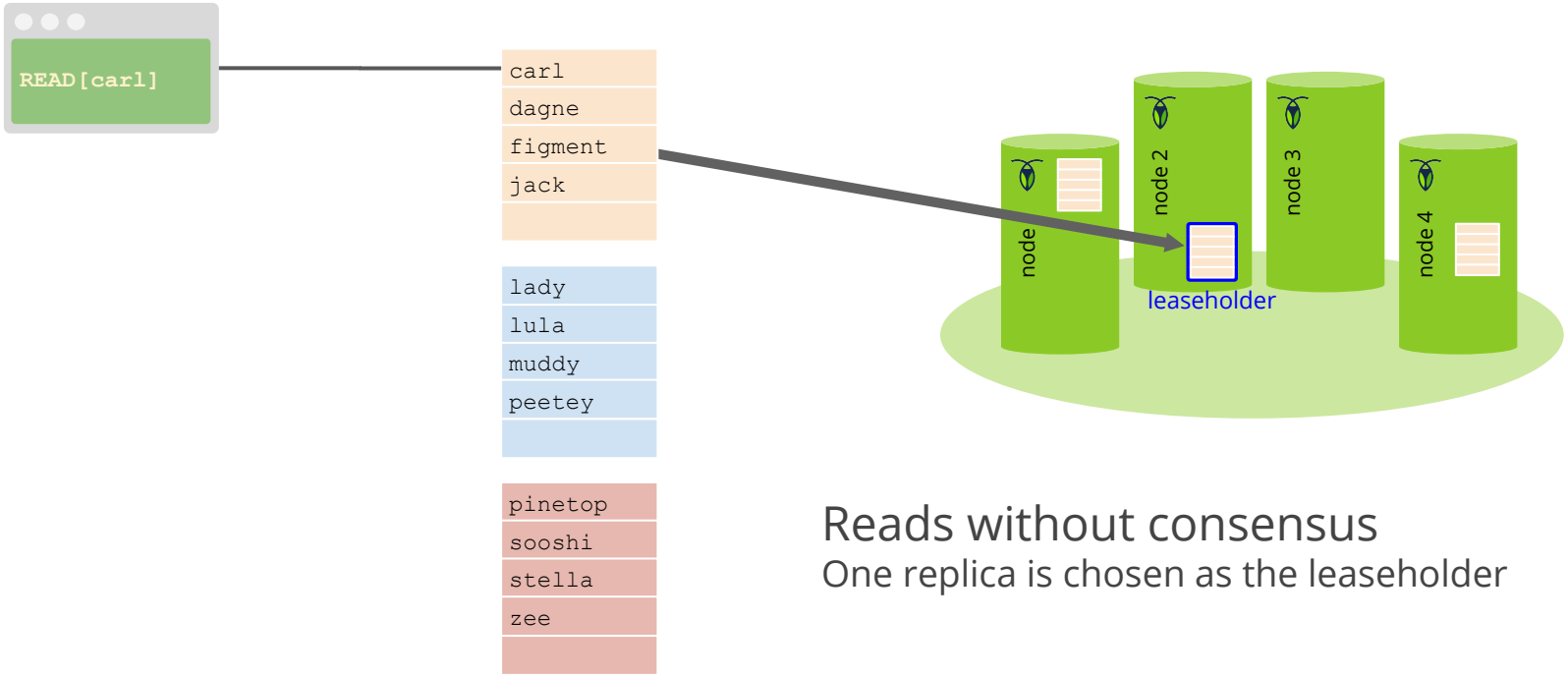
Range Leases



Reads with consensus

Reads must talk to a quorum of replicas

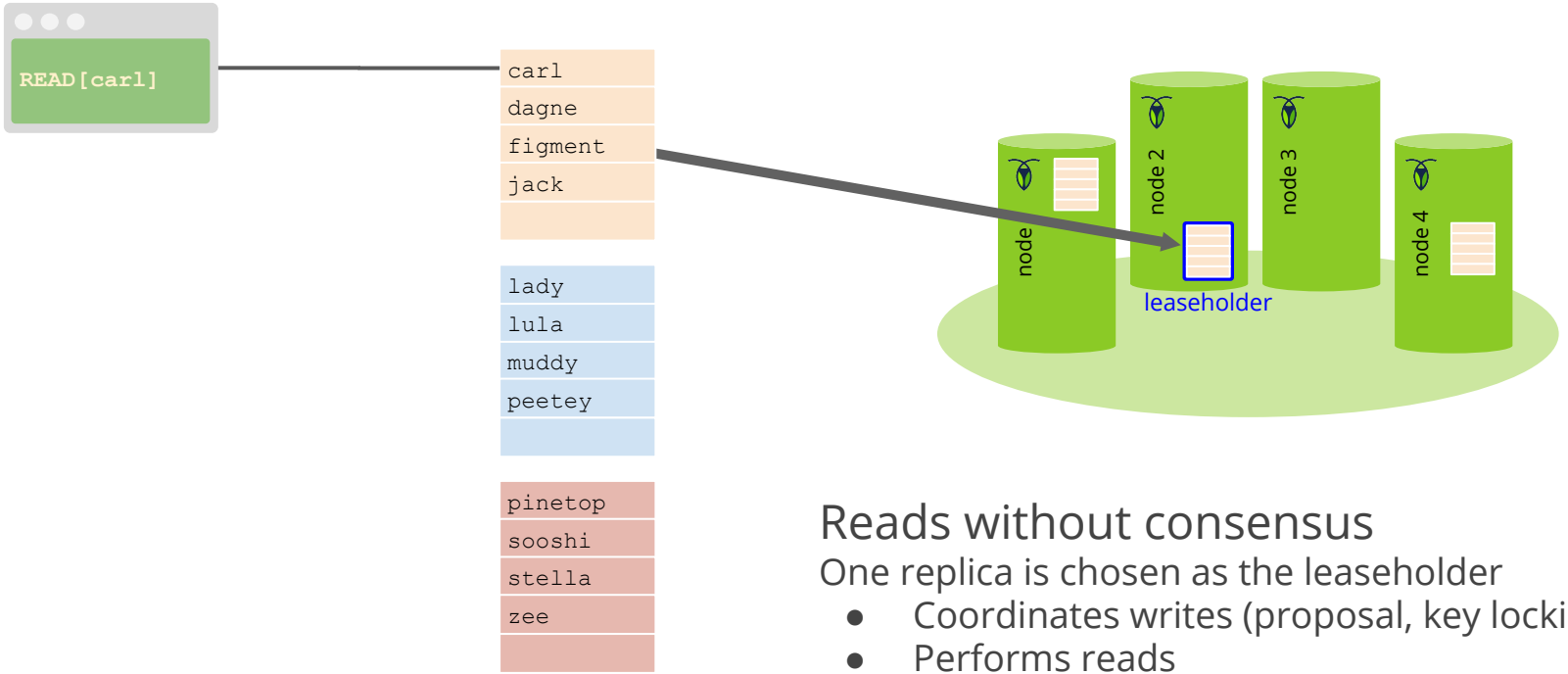
Range Leases



Reads without consensus

One replica is chosen as the leaseholder

Range Leases



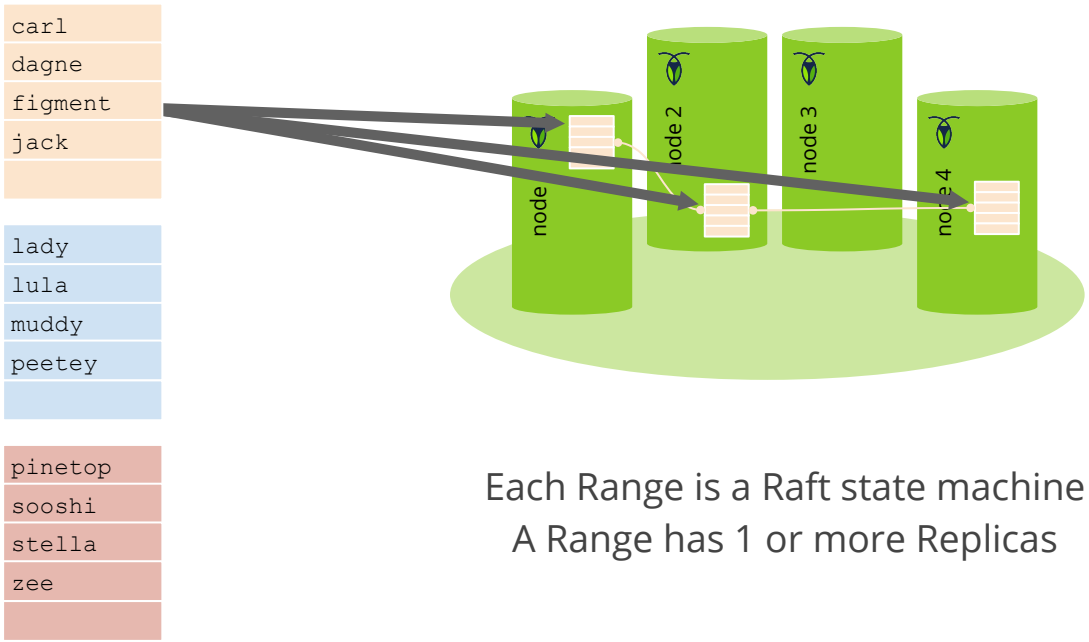
Reads without consensus

One replica is chosen as the leaseholder

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

Replica Placement

- Space
- Diversity
- Load
- Latency



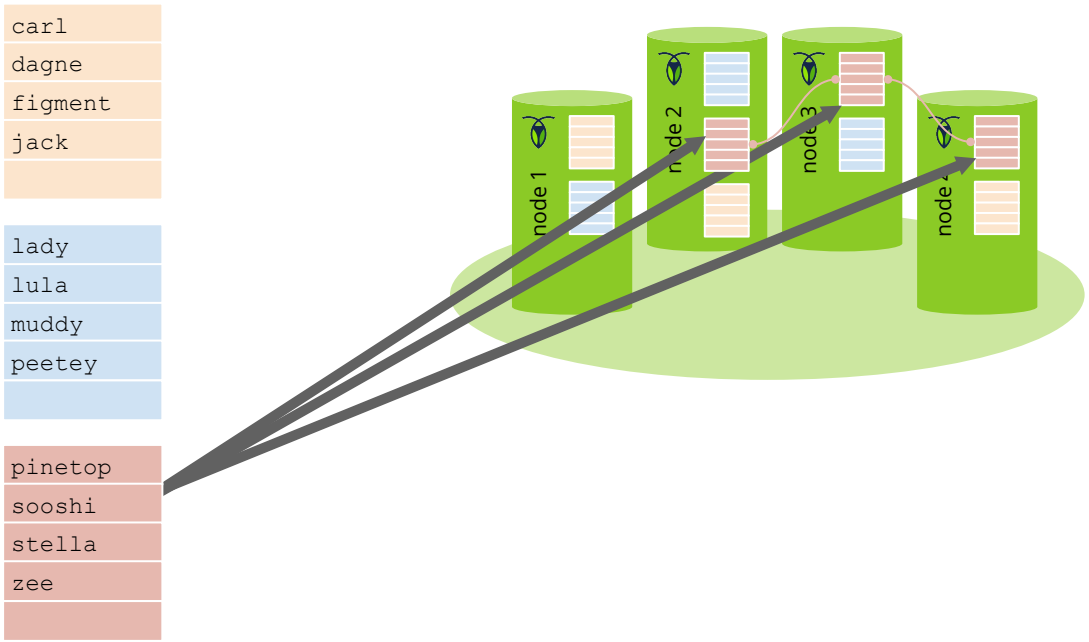
Each Range is a Raft state machine
A Range has 1 or more Replicas

Replica Placement: Diversity

Diversity

optimizes placement of replicas across “failure domains”

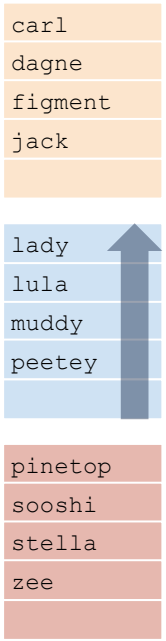
- Disk
- Single machine
- Rack
- Datacenter
- Region



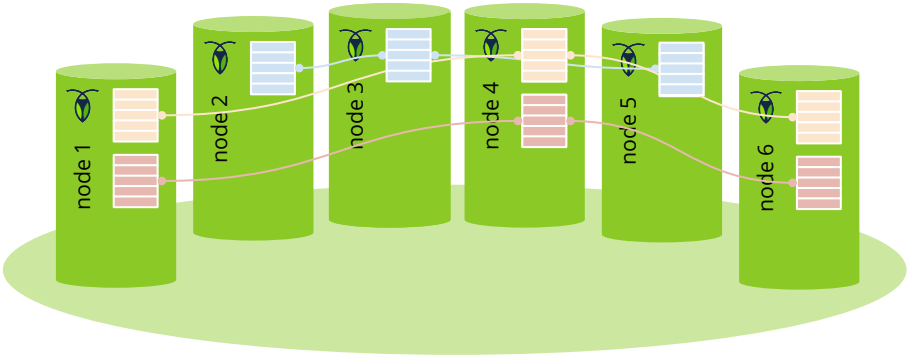
Replica Placement: Load

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

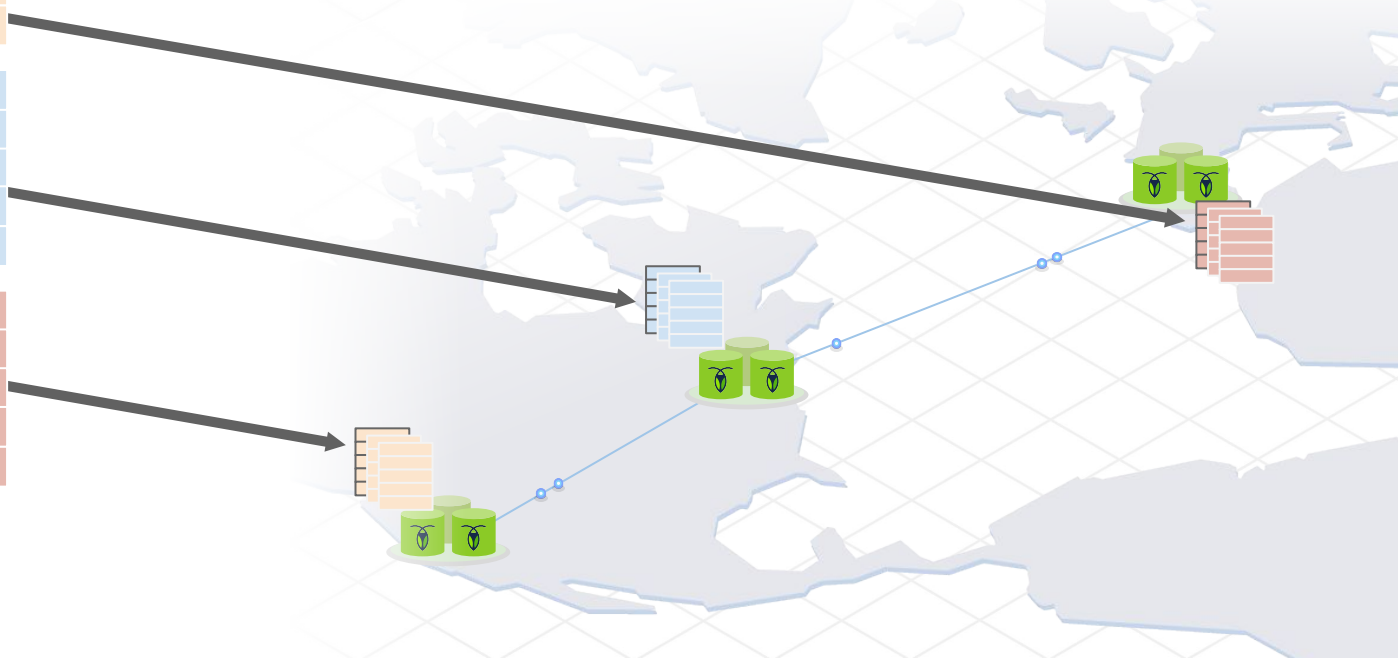


While we show this for ranges within a single table, this is also applicable across all ranges across ALL tables, which is the more typical situation

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

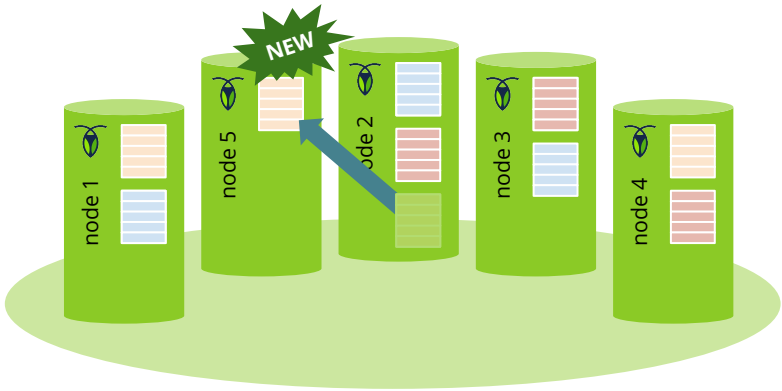
Uses the replica placement heuristics from previous slides to decide which node to add to and which to remove from

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

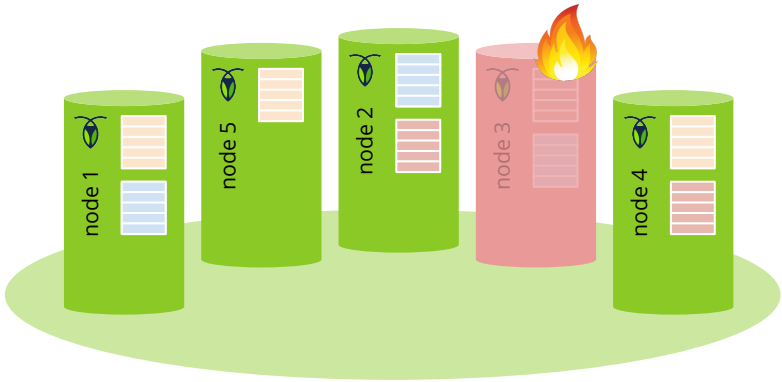
Rebalancing Replicas

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



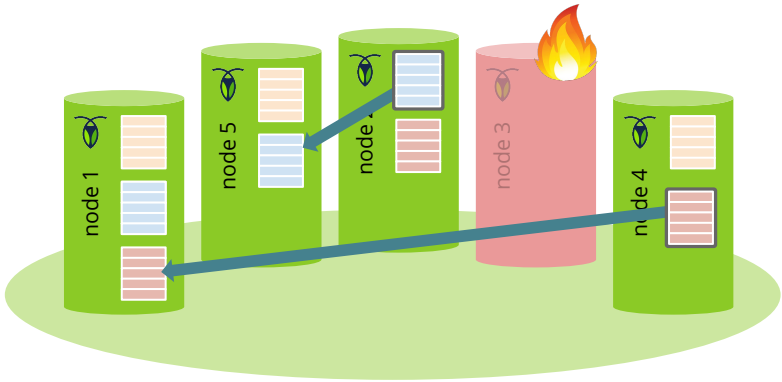
Rebalancing Replicas

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



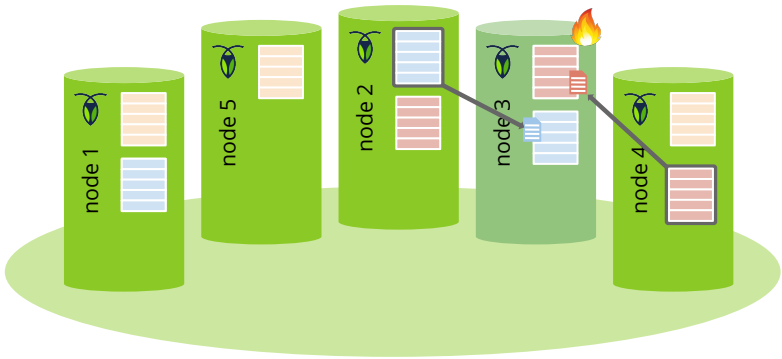
The failed replica is removed from the Raft group and a new replica created. The leaseholder sends a snapshot of the Range's state to bring the new replica up to date.

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.

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

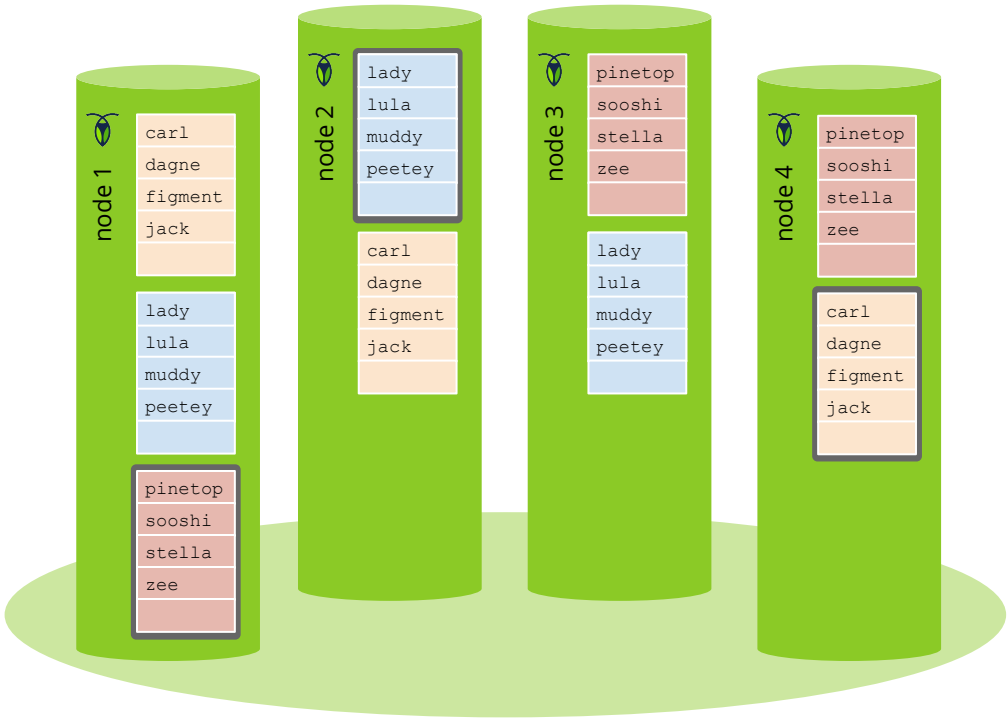
Raft provides atomic writes to individual ranges

Bootstrap transaction atomicity using Raft atomic writes

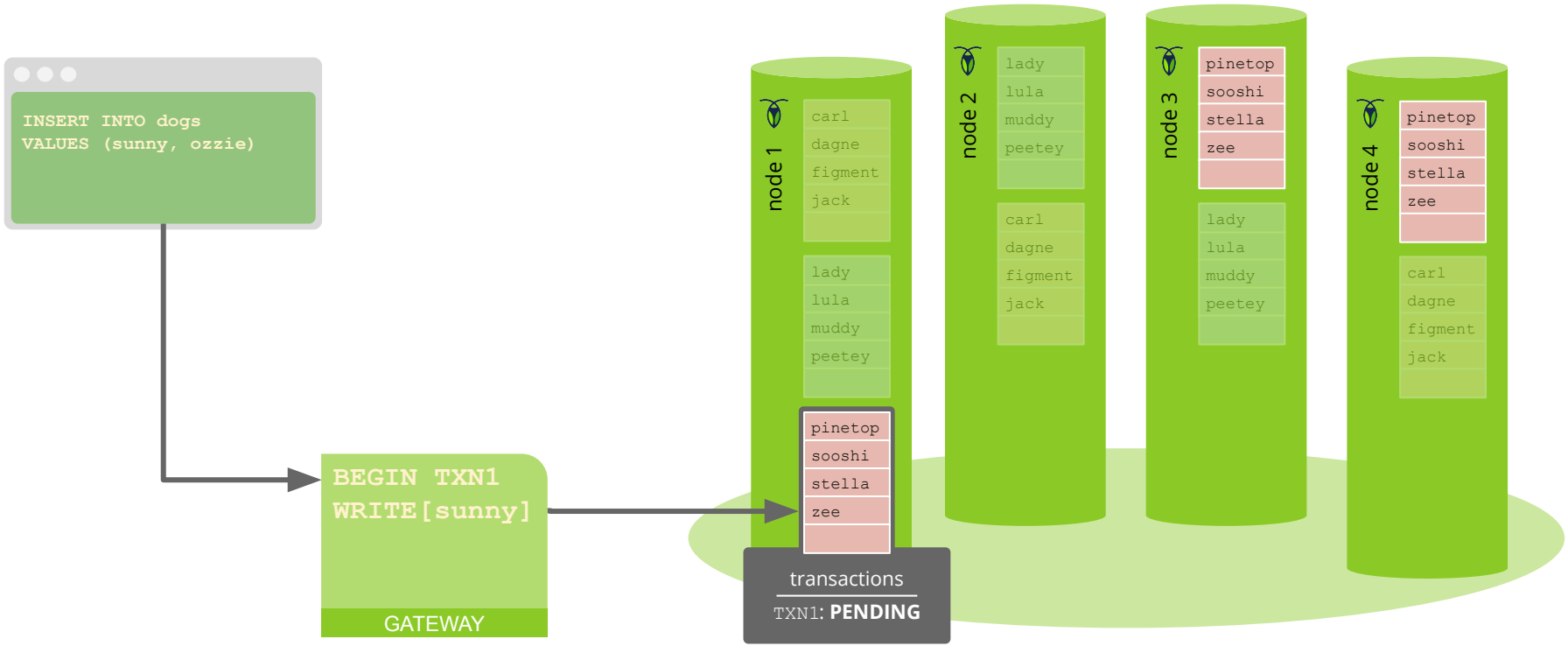
Transaction record atomically flipped from PENDING to COMMIT

Distributed Transactions

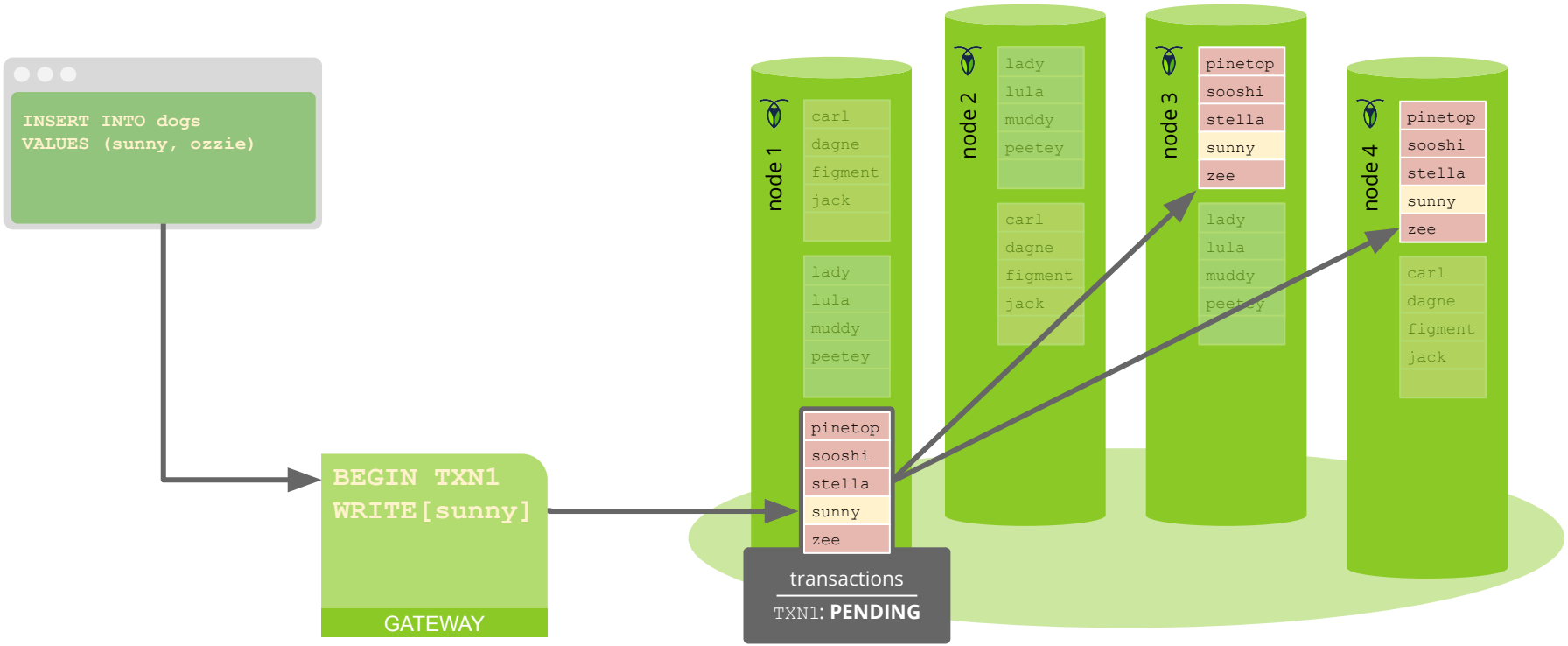
```
INSERT INTO dogs
VALUES (sunny, ozzie)
```



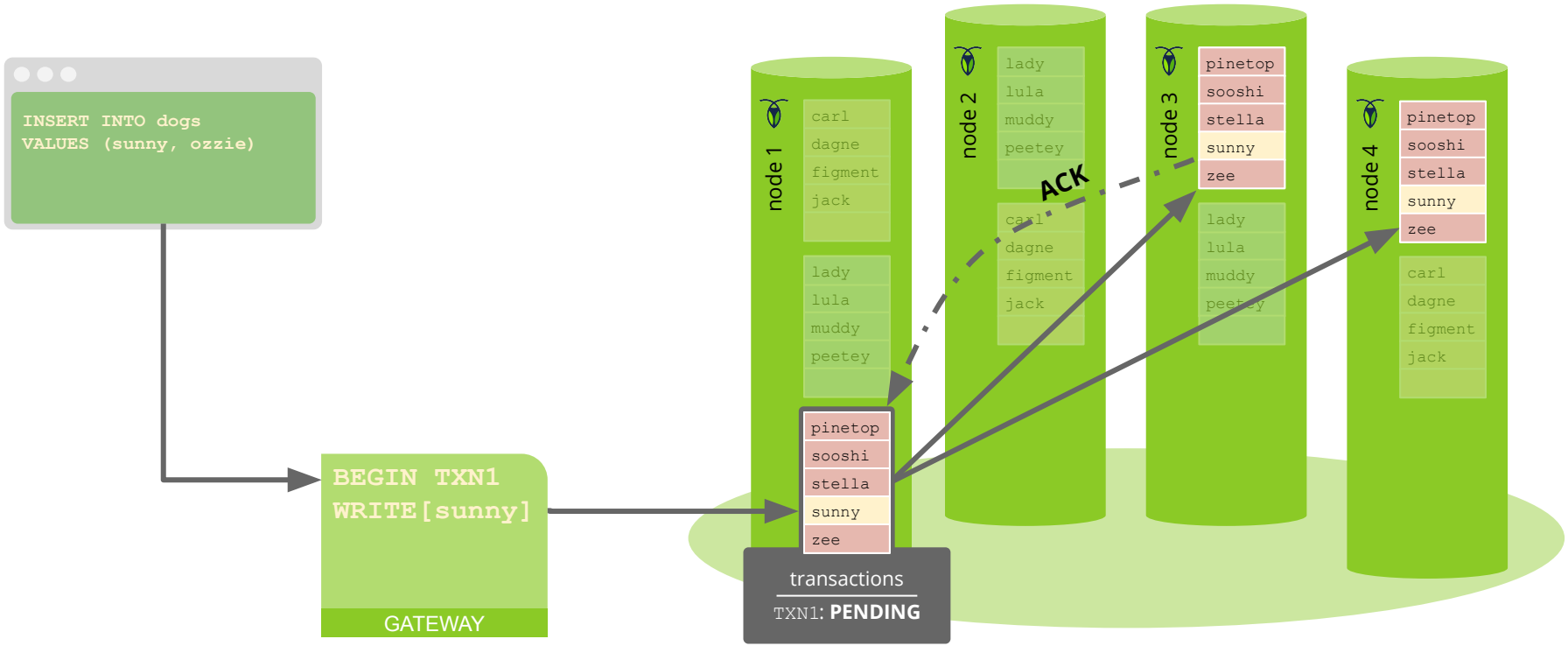
Distributed Transactions



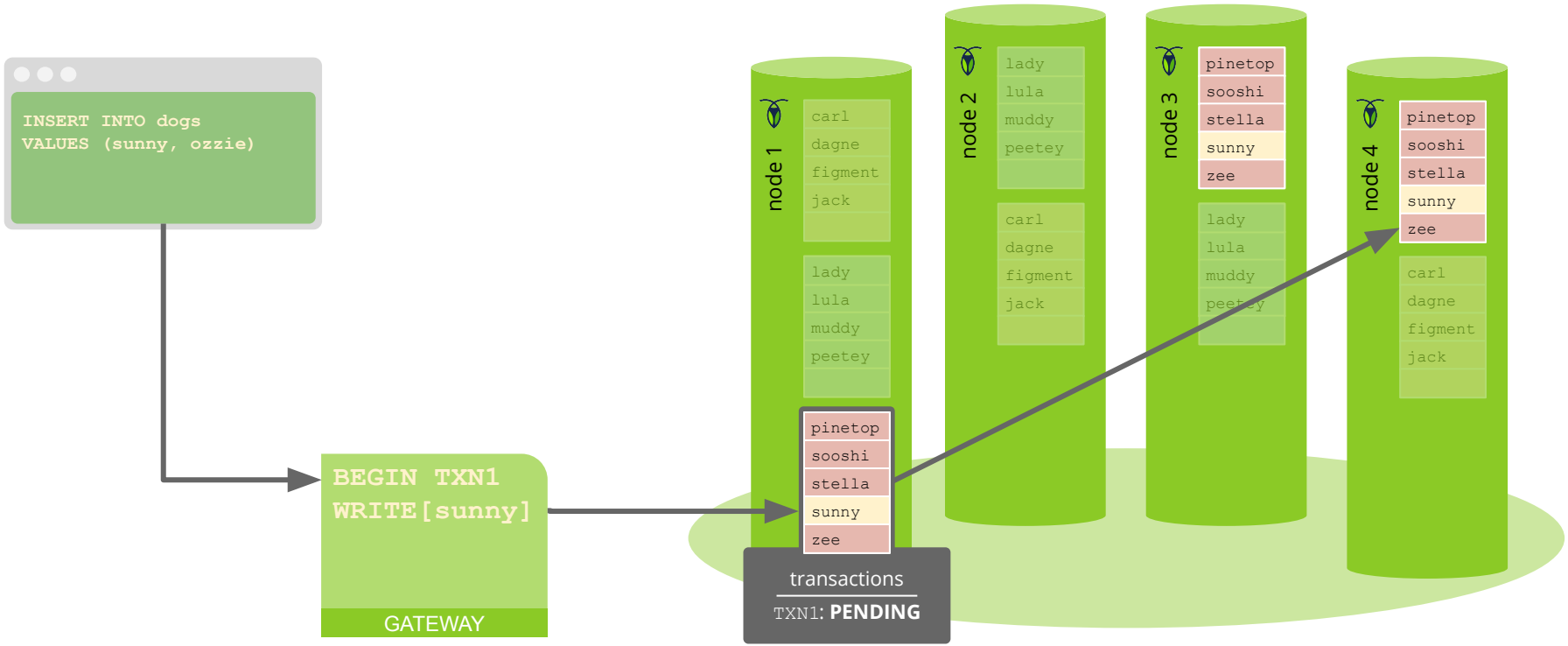
Distributed Transactions



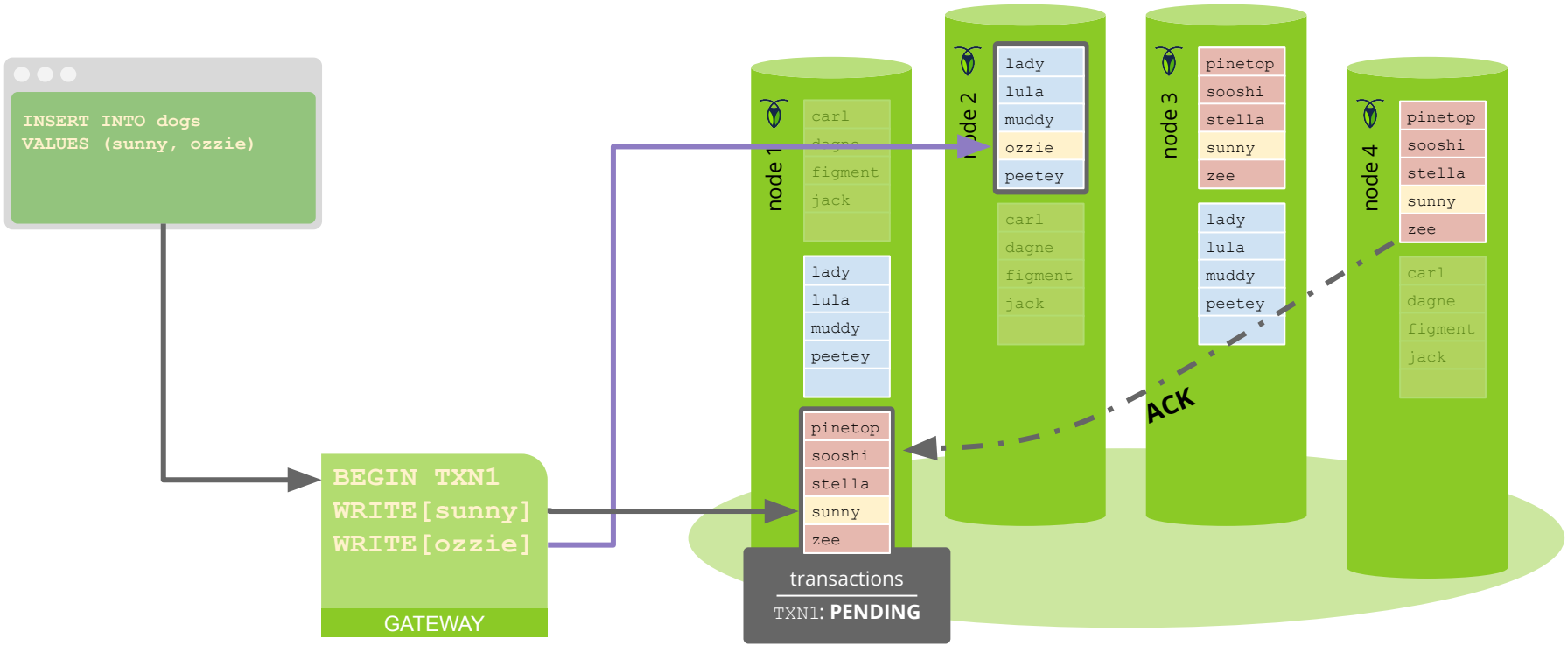
Distributed Transactions



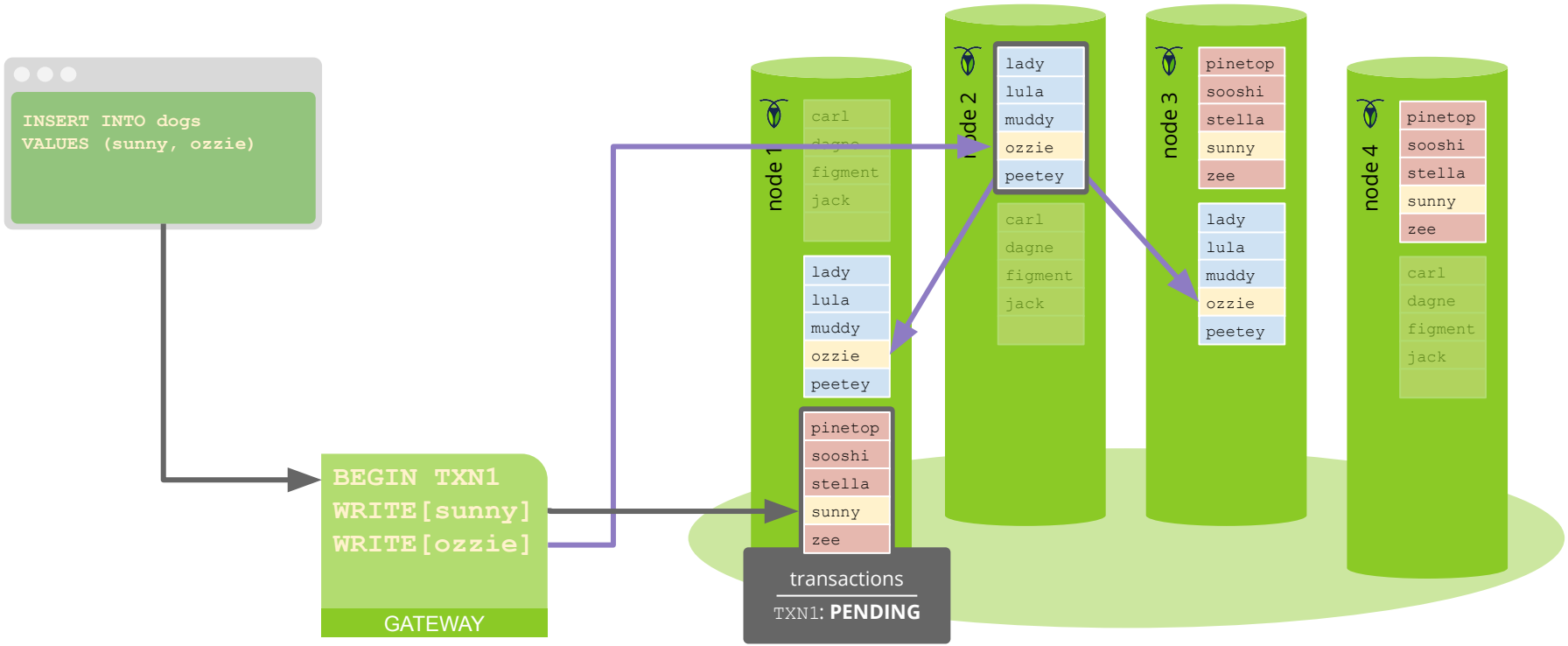
Distributed Transactions



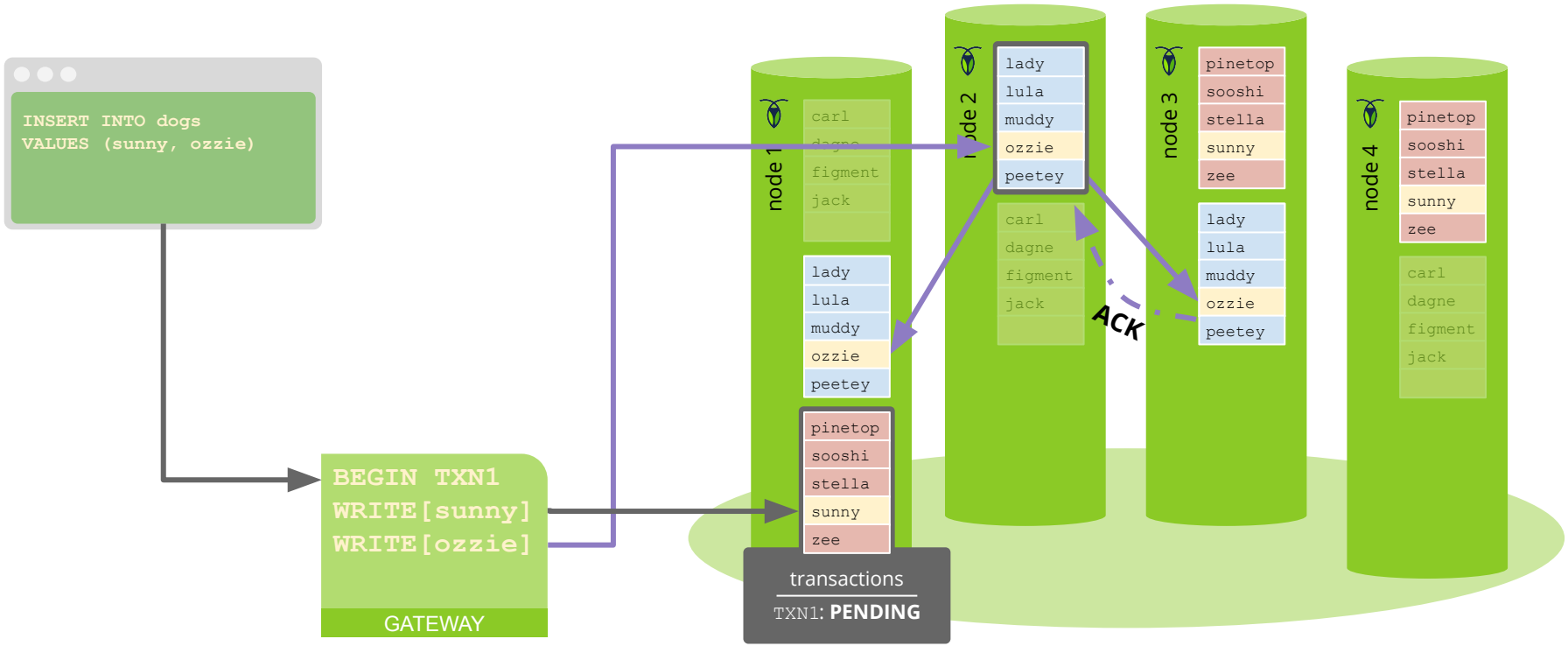
Distributed Transactions



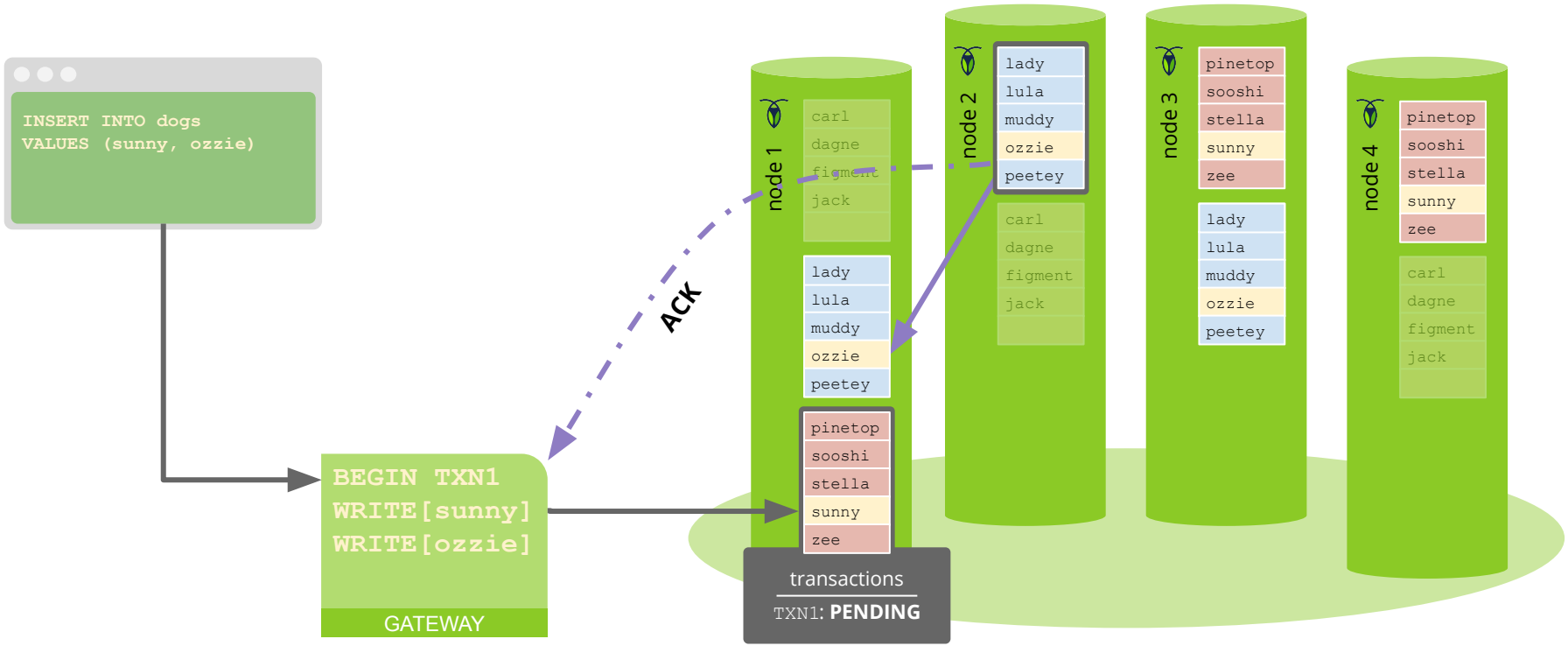
Distributed Transactions



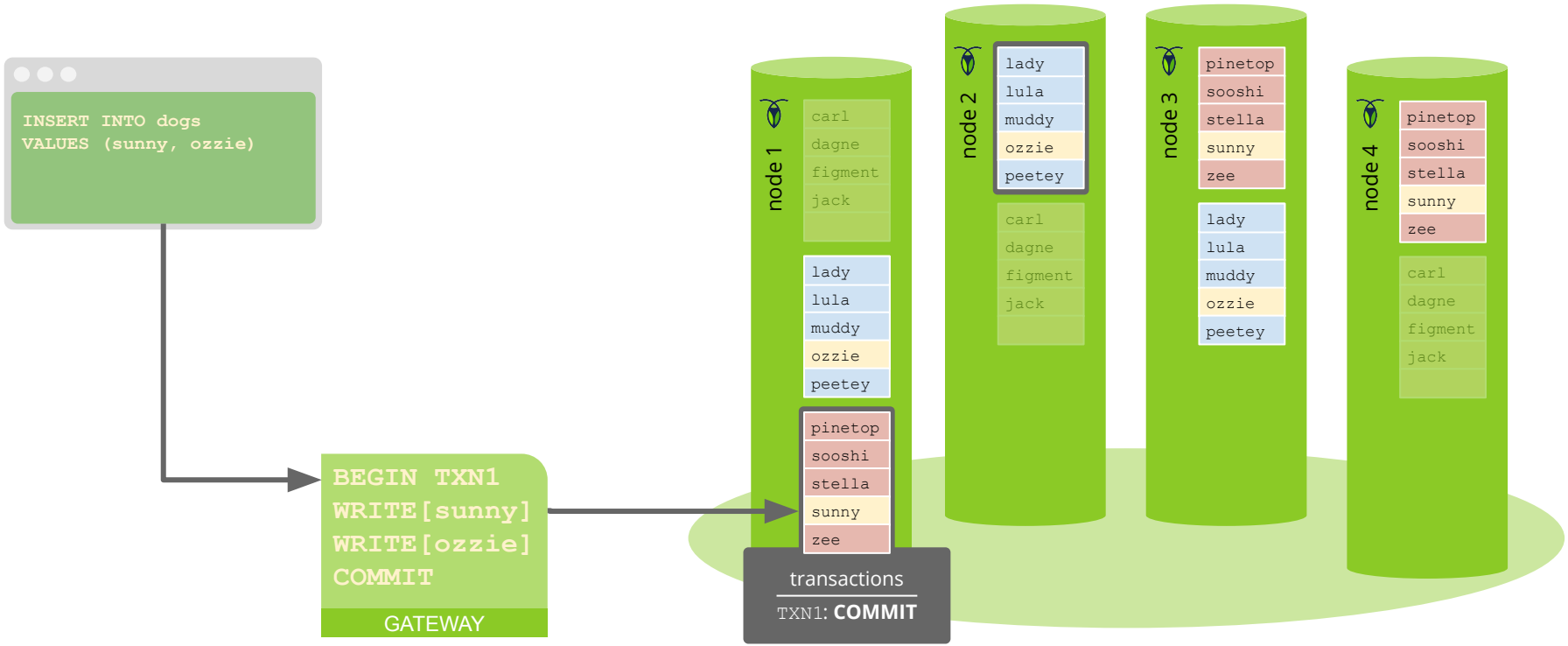
Distributed Transactions



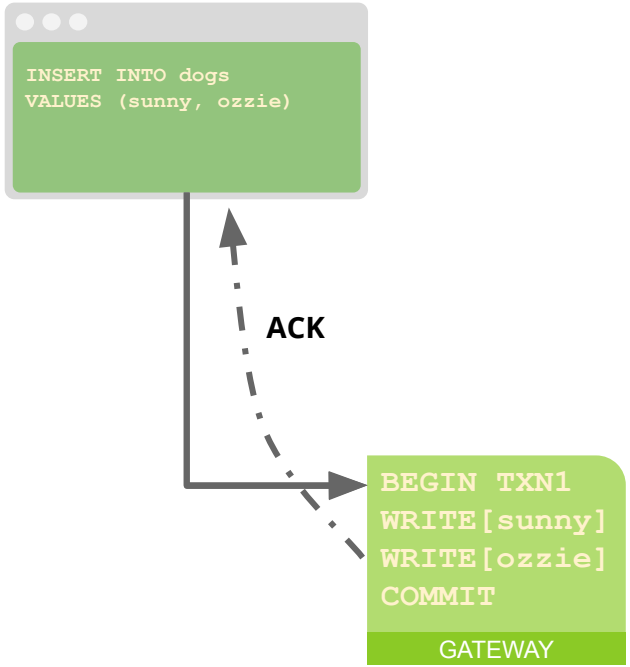
Distributed Transactions

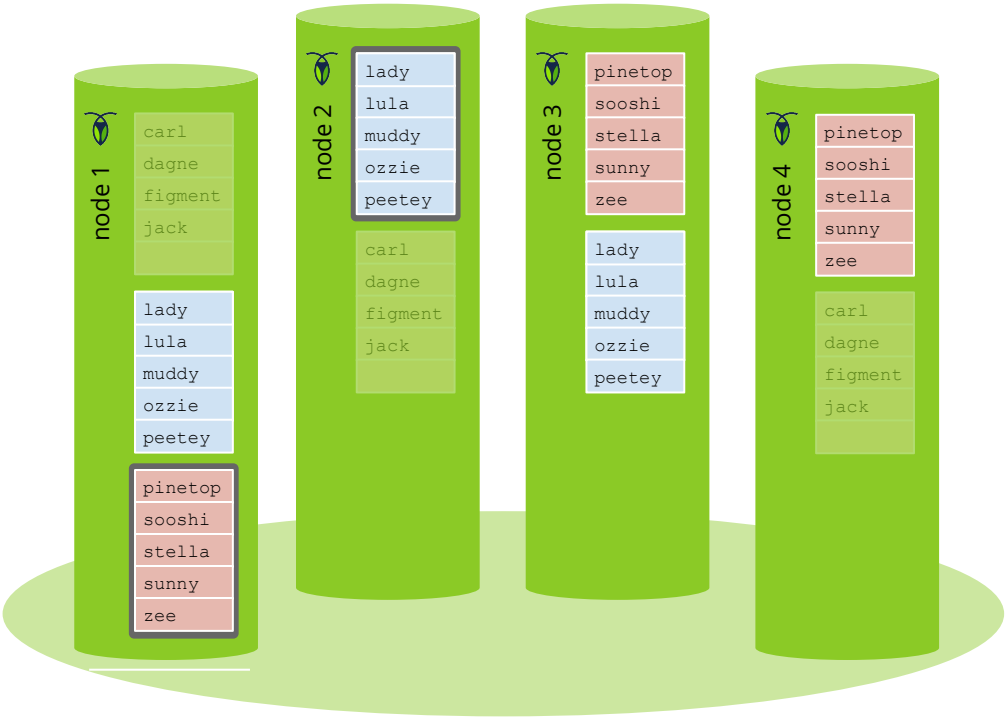


Distributed Transactions

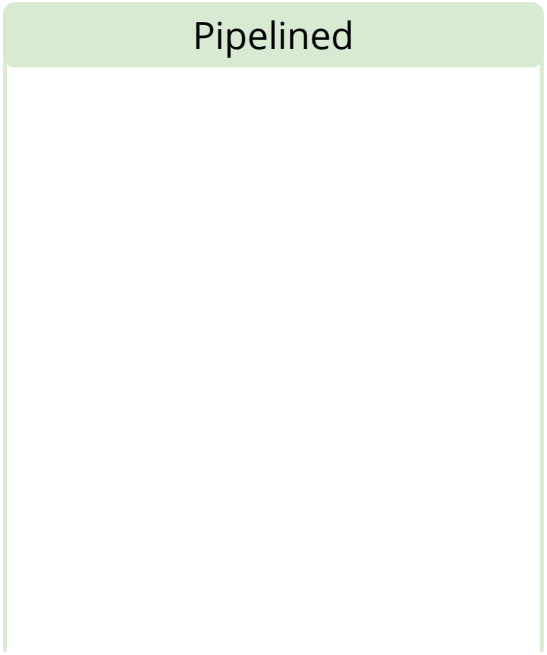


Distributed Transactions



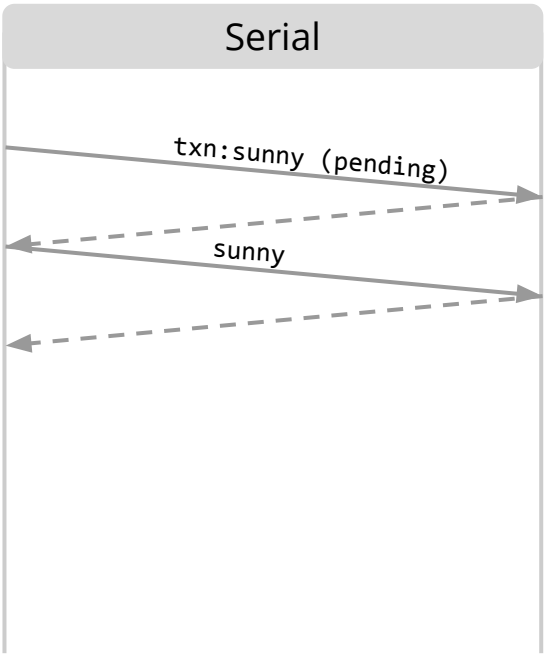


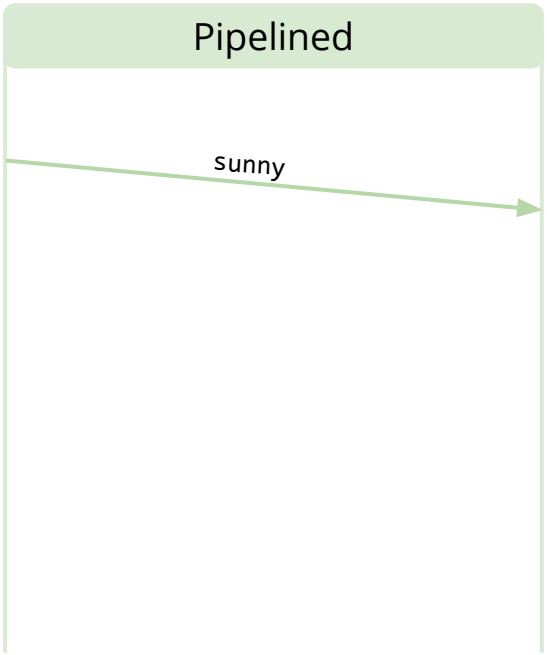
Transactions: Pipelining



Transactions: Pipelining

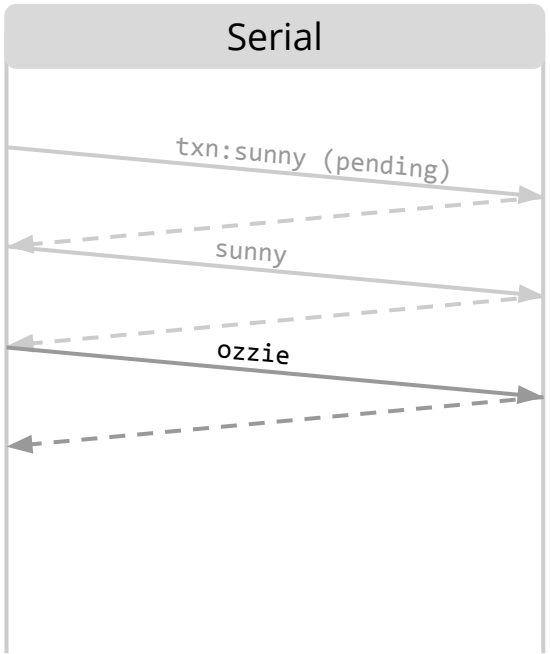
```
BEGIN
WRITE[sunny]
```

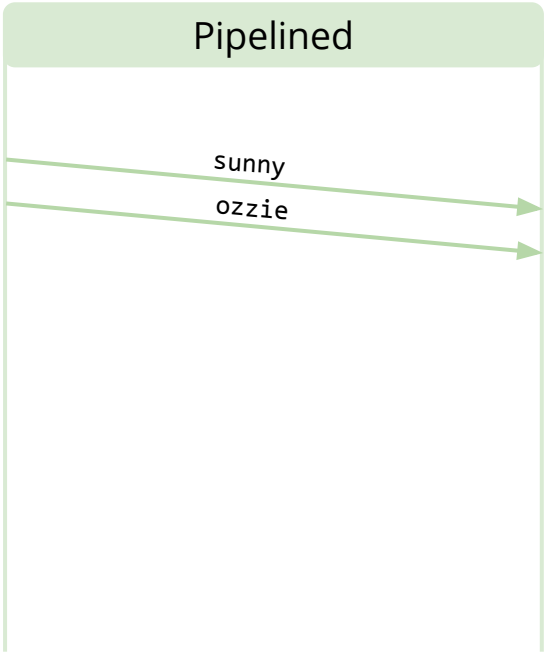




Transactions: Pipelining

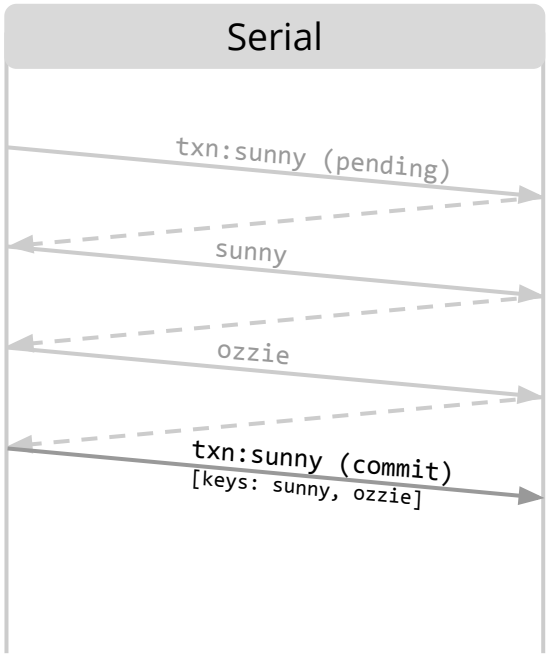
```
BEGIN
WRITE[sunny]
WRITE[ozzie]
```

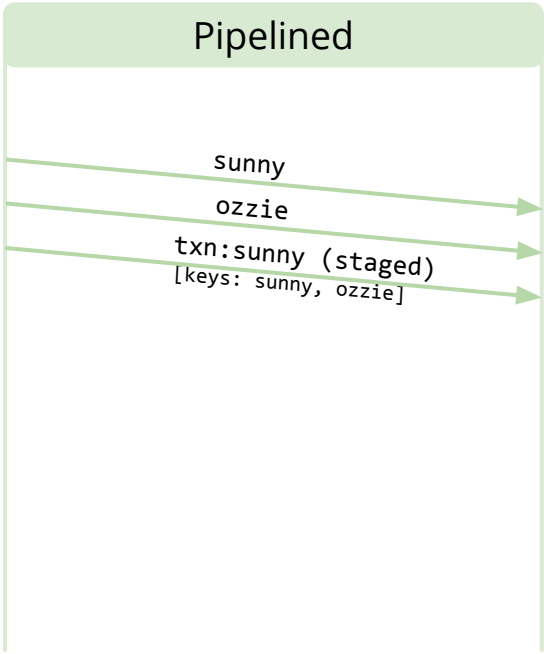




Transactions: Pipelining

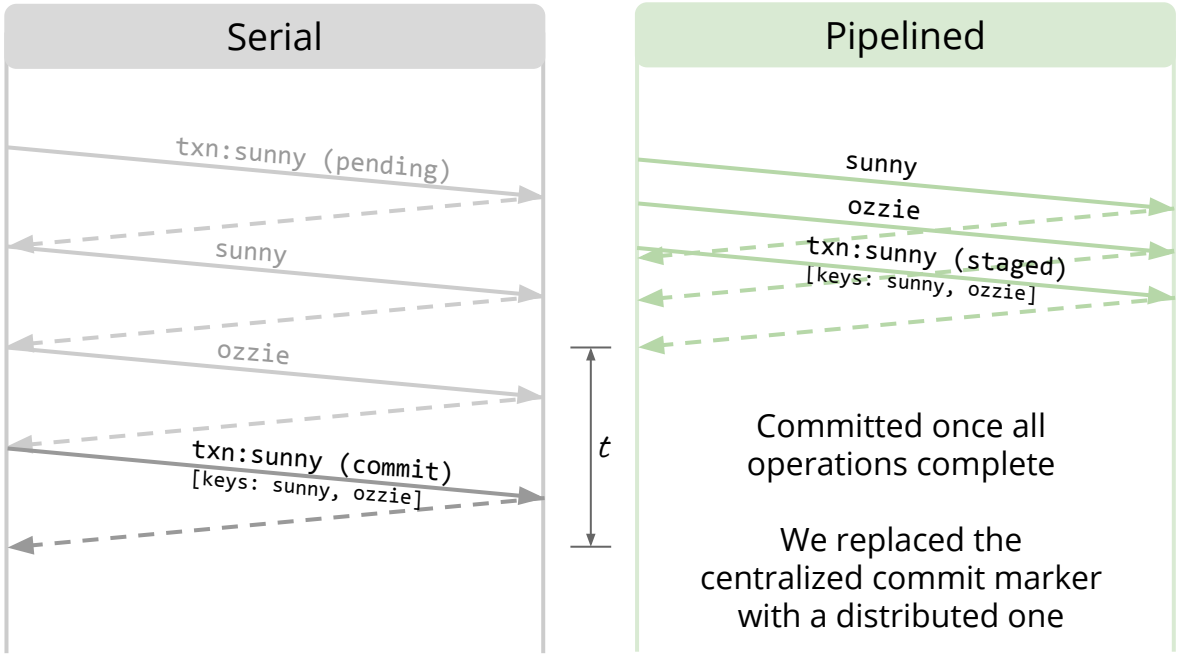
```
BEGIN
WRITE[sunny]
WRITE[ozzie]
COMMIT
```





Transactions: Pipelining

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



* “Proved” with TLA+

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

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	∅

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	∅