Cockroach Labs

The Hitchhiker's Guide to Distributed Transactions

@irfansharifm

Abstract

Recent years have seen a flurry of research, both from academia and industry, enabling distributed transactions while minimizing the performance overhead. There are various proposals afoot; some systems constrain the transaction model, others will trade-off behavior under contention. Some systems sacrifice isolation guarantees, others will incur higher read latencies.

This talk is a survey of the various approaches that academic (Carousel, MDCC, SLOG, TAPIR) and industrial (Spanner, CockroachDB, OceanVista) systems use to achieve atomicity in their transactions. We'll define a shared terminology (ranges, replicas, txn records, etc.) and use it to explore how system unique compose their transaction models with the underlying replication protocol to achieve the theoretical minimum latency for atomic commitment: one round-trip between data centers.

Agenda

1. Foundations
2. Transactions
3. Implementations

Agenda

1. **Foundations**
2. Transactions
3. Implementations

I. Keyspace and Sharding
II. Replication and Fault Tolerance
III. APIs

Agenda

1. Foundations
2. **Transactions**
3. Implementations

I. ACID & Isolation Levels
II. Transaction Basics
III. Unpipelined Transactions

Agenda

1. Foundations
2. Transactions
3. **Implementations**

I. Spanner/Pipelined Transactions
II. Parallel Commits
III. Replicated Commit
IV. Carousel
V. MDCC
VI. SLOG/OceanVista
VII. TAPIR

Agenda

1. **Foundations**
2. Transactions
3. Implementations

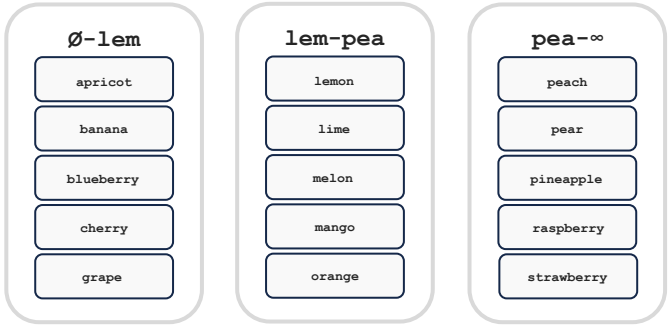
I. Keyspace and Sharding
II. Replication and Fault Tolerance
III. APIs

Monolithic, sorted, logical key space

We’re ignoring non-ordered databases (think consistent hashing), though most of the same principles will apply

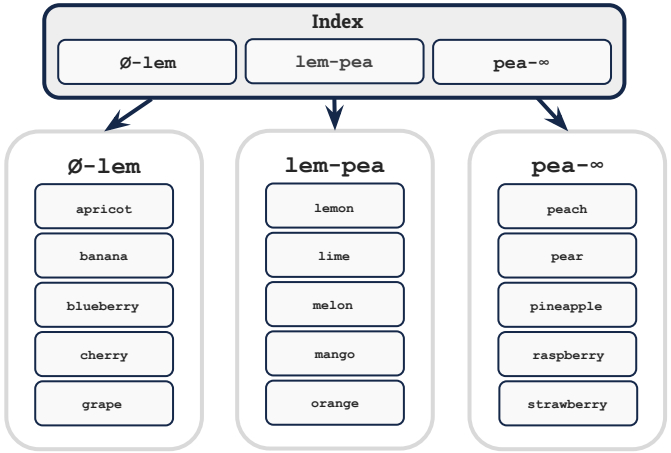
Ranges

Each *Range* (think “shards”) holds a contiguous span of the key-space



Indexes

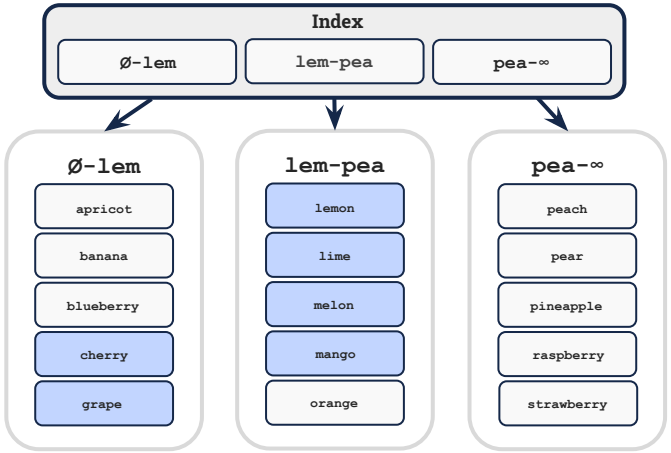
We could use indexing structures to locate shards/ranges



Range scans

Ordered keys makes range scans efficient

```
fruits >= "cherry" AND <= "mango"
```

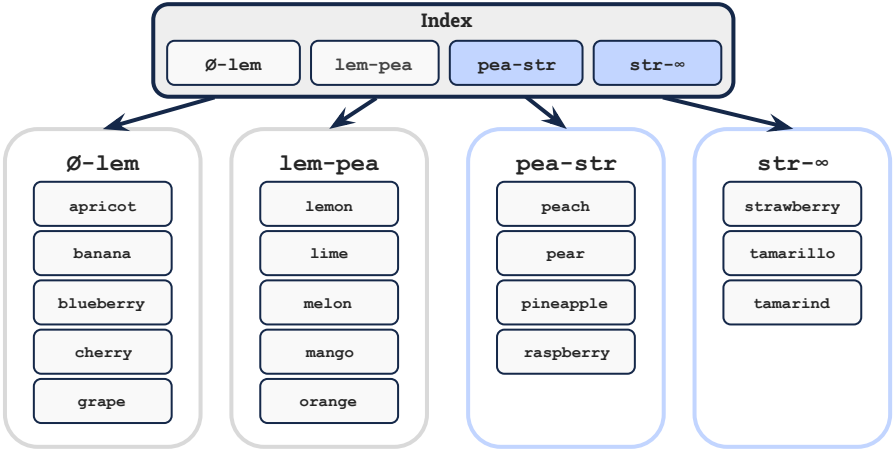


Range splits/merges

Could split ranges when they get too large, merge when they get too small

We want range sizes to be:

a. small enough to be moved quickly
b. large enough to amortize indexing overhead



Other systems

Bigtable calls these *tablets,* Hbase calls these *regions,* CRDB calls these *ranges*. Similar structures found in Spanner, YugaByte, SLOG, etc.

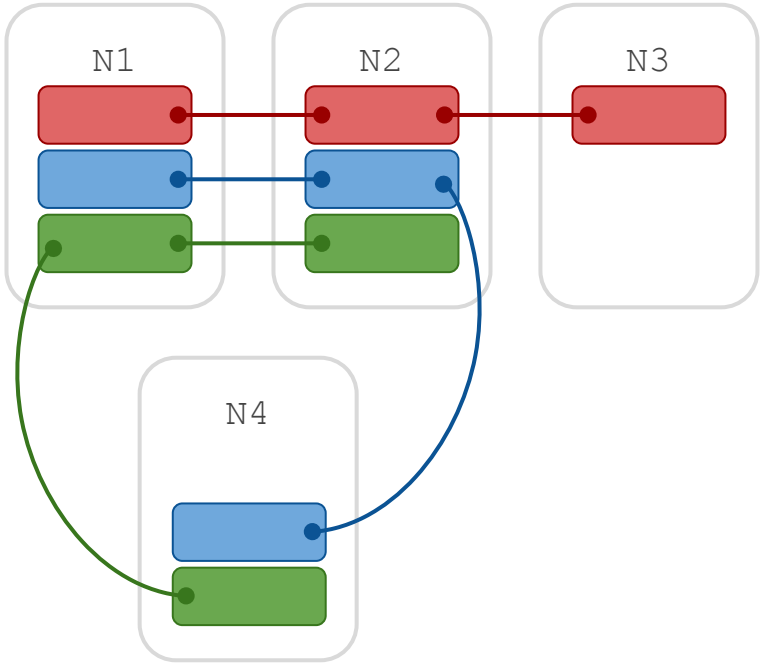
Agenda

1. **Foundations**
2. Transactions
3. Implementations

I. Keyspace and Sharding
II. Replication and Fault Tolerance
III. APIs

Consensus replication

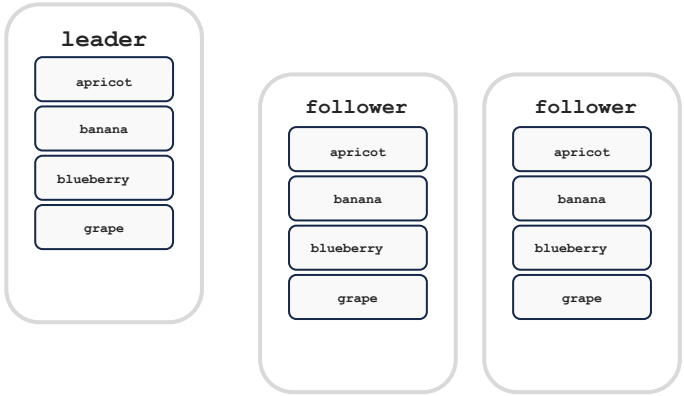
Ranges are the unit of replication, each copy is a *Replica*. A single node could hold one or more replicas.



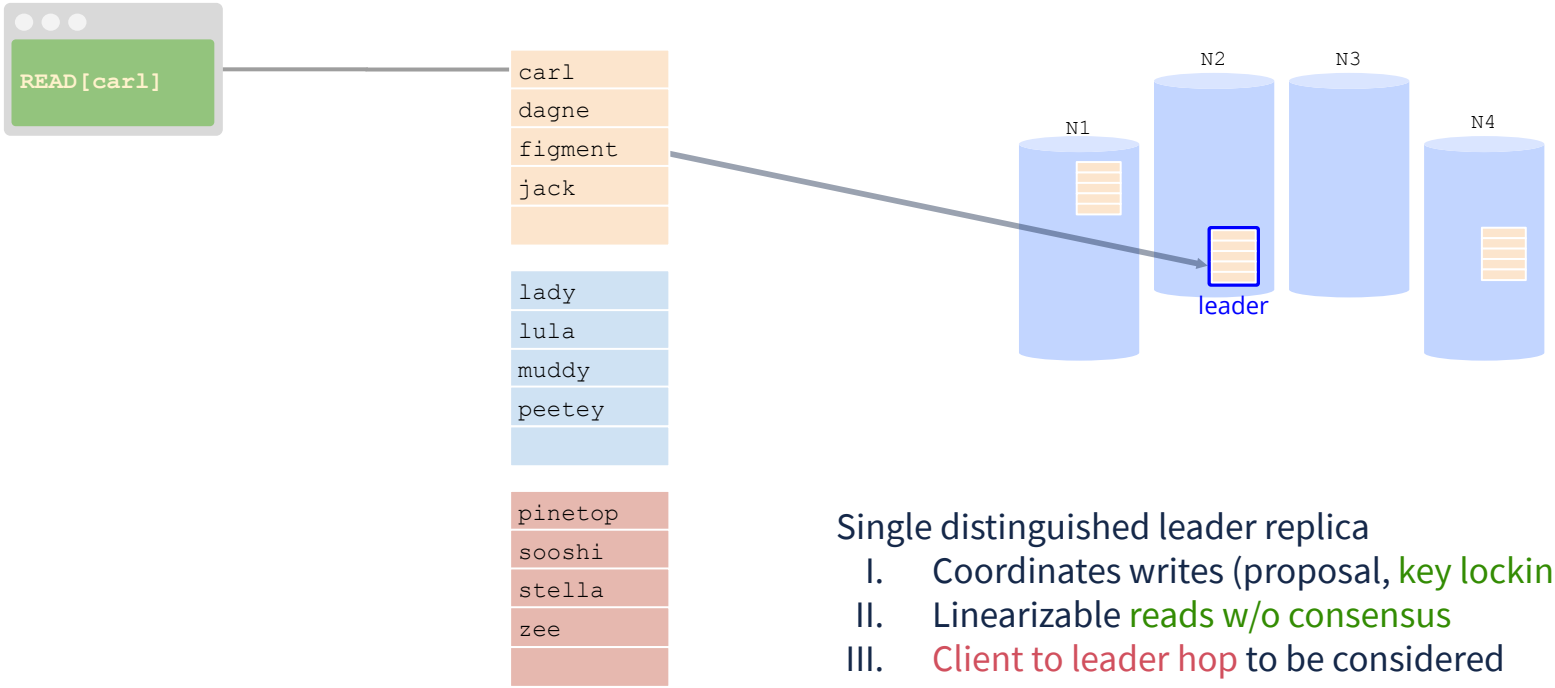
Consensus replication

There are several variants we could use, leader-based (raft, multi-paxos) and leaderless (epaxos) ones

NB: A replication factor of 2 doesn't make sense for us (but is somewhat akin to primary/secondary replication)



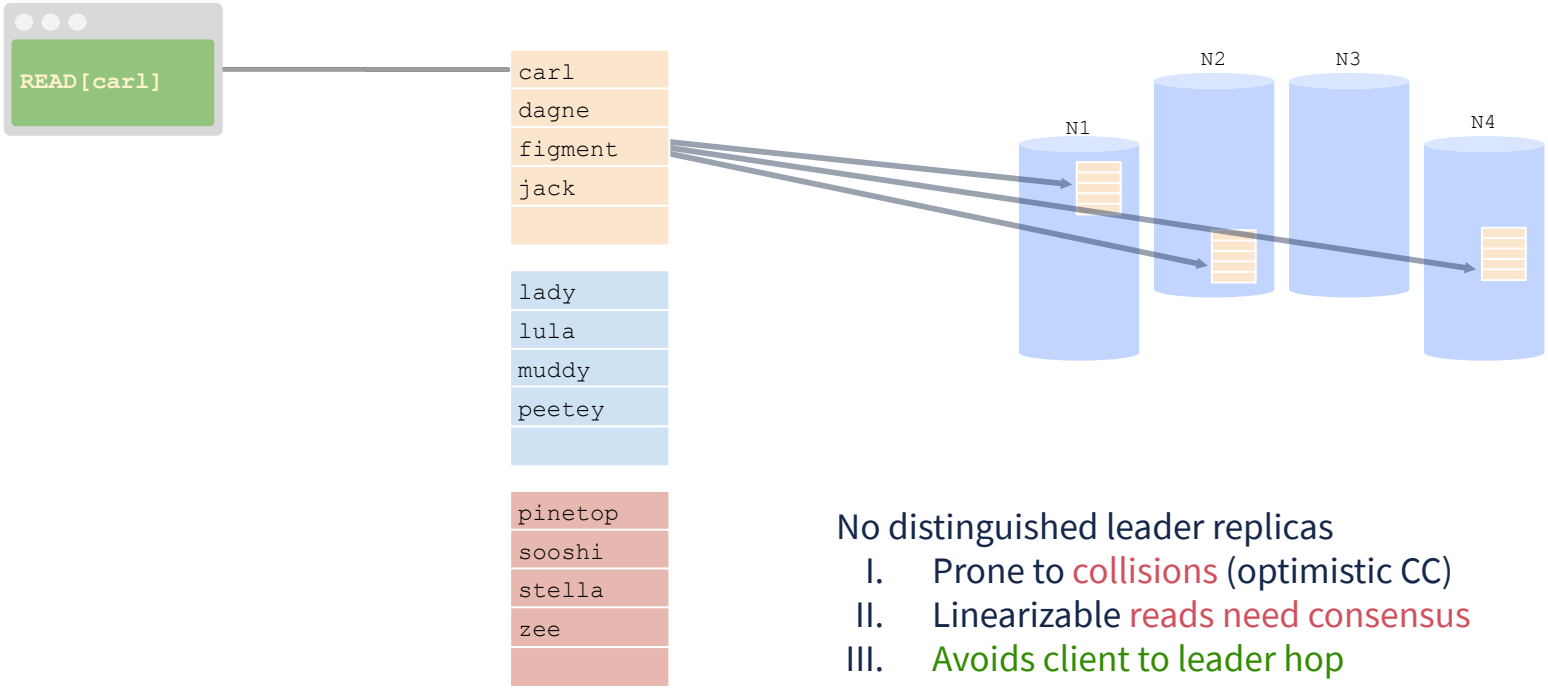
Leader-based consensus



Single distinguished leader replica

I. Coordinates writes (proposal, key locking)
II. Linearizable reads w/o consensus
III. Client to leader hop to be considered
IV. Leader node failures can cause blips
V. Disproportionate load on leader

Leaderless consensus

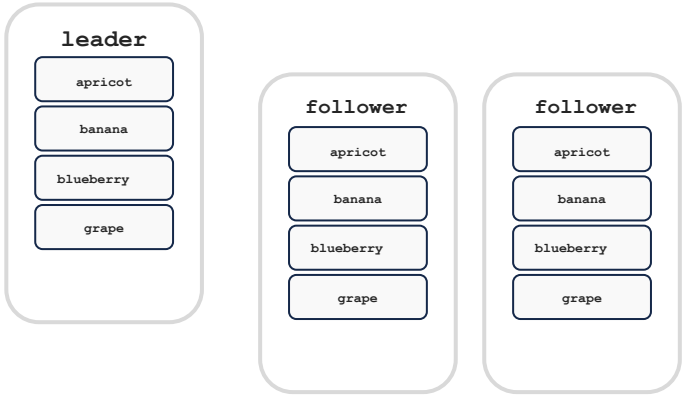


No distinguished leader replicas

I. Prone to collisions (optimistic CC)
II. Linearizable reads need consensus
III. Avoids client to leader hop
IV. No blips caused by node failures
V. More evenly distributed load on nodes

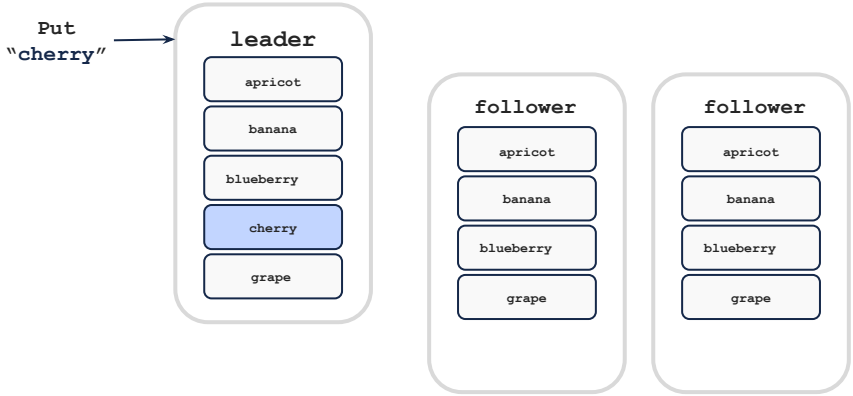
Consensus replication (Writes)

Replicate N-way, committed when acked by a quorum of replicas



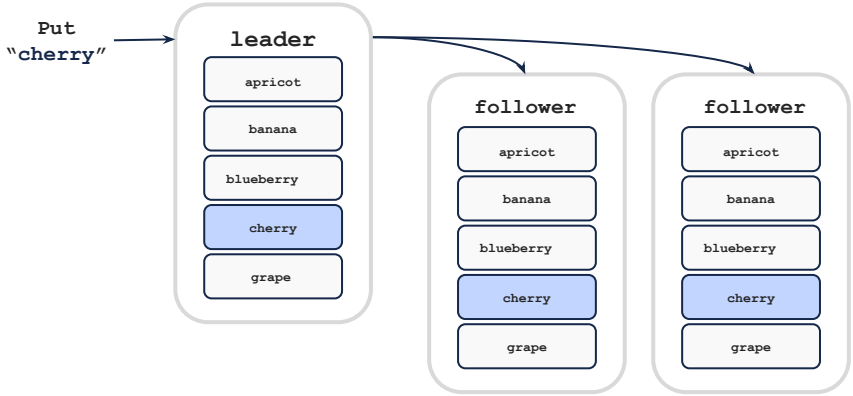
Consensus replication (Writes)

Replicate N-way, committed when acked by a quorum of replicas



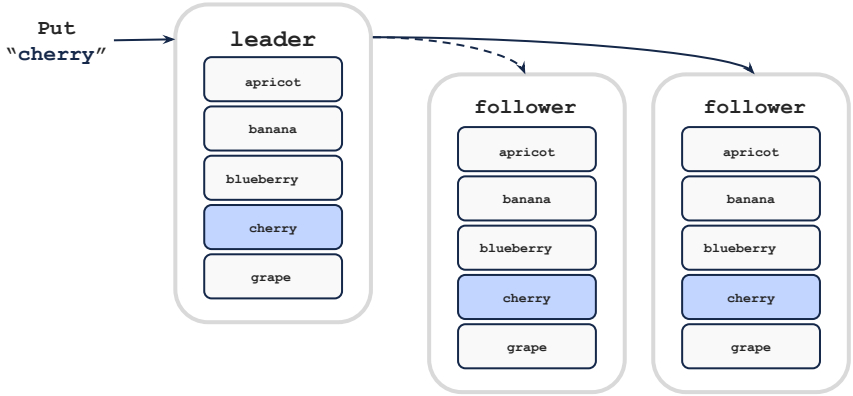
Consensus replication (Writes)

Replicate N-way, committed when acked by a quorum of replicas



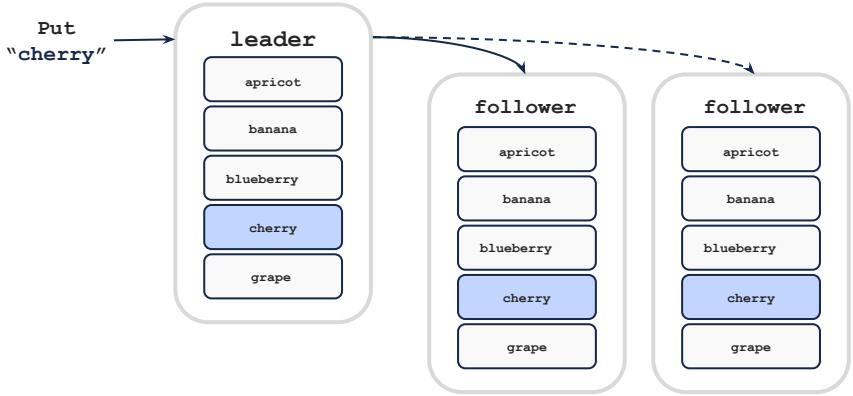
Consensus replication (Writes)

Need ack from just one other node



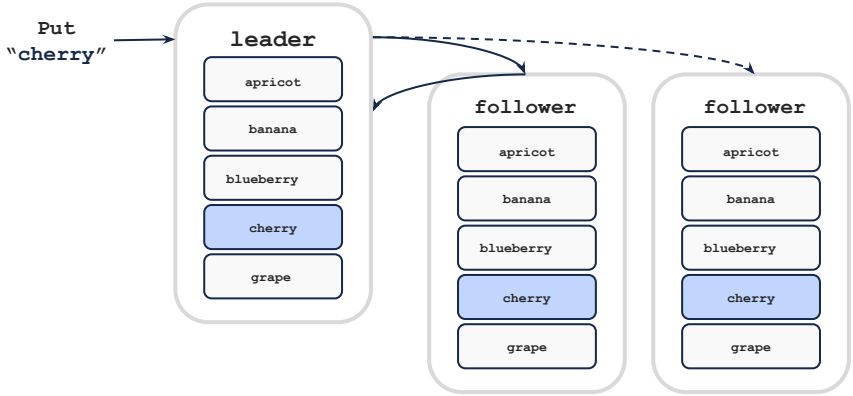
Consensus replication (Writes)

Need ack from just one other node



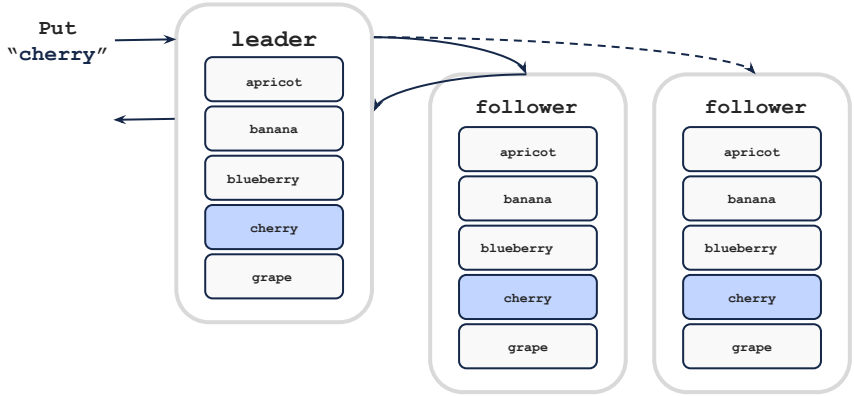
Consensus replication (Writes)

Need ack from just one other node



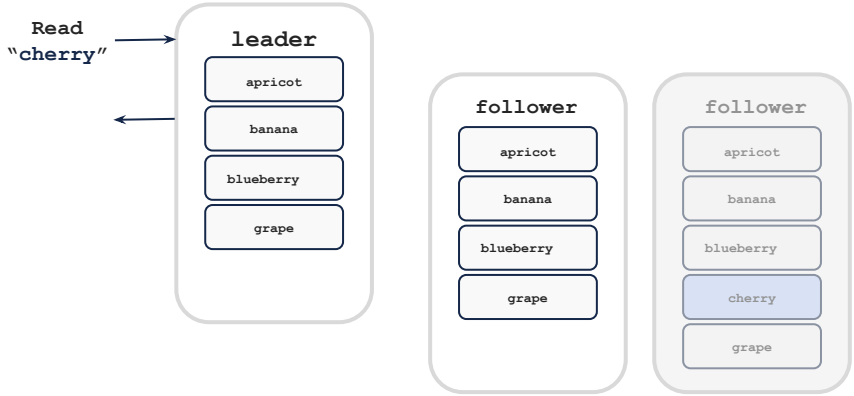
Consensus replication (Writes)

Need ack from just one other node



Consensus replication (Reads)

Only data written to a quorum is considered present



Consensus replication

Consensus provides “durable, atomic replication” of commands

Agenda

1. **Foundations**
2. Transactions
3. Implementations

I. Keyspace and Sharding
II. Replication and Fault Tolerance
III. APIs

Data Mapping: SQL

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

/<table>/<index>/<key>

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

Data Mapping: SQL

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

/<table>/<index>/<key>

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

1. Foundations
2. **Transactions**
3. Implementations

I. ACID & Isolation Levels
II. Transaction Basics
III. Unpipelined Transactions

Agenda

1. Foundations
2. **Transactions**
3. Implementations

I. ACID & Isolation Levels
II. Transaction Basics
III. Unpipelined Transactions

ACID

I. Atomicity (“all or nothing”; achieved using staged writes and txn records)
II. Consistency (“db is internally consistent”)
III. Isolation (“effects of concurrent txns on each other”; determined by locks)
IV. Durability (“don’t lose committed data”; determined by synced writes)

ACID

I. Atomicity (“all or nothing”; achieved using staged writes and txn records)
II. Consistency (“db is internally consistent”)
III. Isolation (“effects of concurrent txns on each other”; determined by locks)
IV. Durability (“don’t lose committed data”; determined by synced writes)

ACID

I. Atomicity (“all or nothing”; achieved using staged writes and txn records)
II. Consistency (“db is internally consistent”)
III. Isolation (“effects of concurrent txns on each other”; determined by locks)
IV. Durability (“don’t lose committed data”; determined by synced writes)

ACID

I. Atomicity (“all or nothing”; achieved using staged writes and txn records)
II. Consistency (“db is internally consistent”)
III. Isolation (“effects of concurrent txns on each other”; determined by locks)
IV. Durability (“don’t lose committed data”; determined by synced writes)

Isolation

Isolation Level (“effects of concurrent txns on one another”; determined by locking granularity)

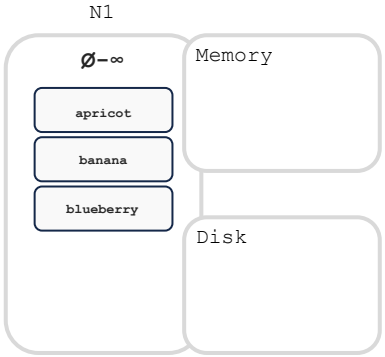
I. read uncommitted (could read ongoing-but-uncommitted writes)
II. read committed (could read different values in the same txn)
III. repeatable read (could range-read different values in the same txn)
IV. snapshot isolation (could make write decisions based on stale reads)
V. serializable (none of the above, as if run in serial order)

Agenda

1. Foundations
2. **Transactions**
3. Implementations

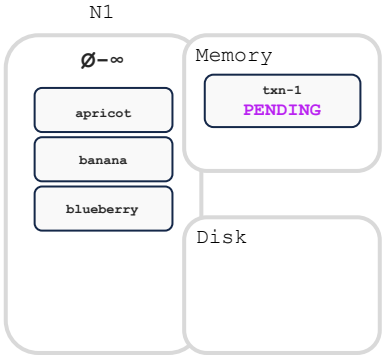
I. ACID & Isolation Levels
II. Transaction Basics
III. Unpipelined Transactions

Single-node transactions



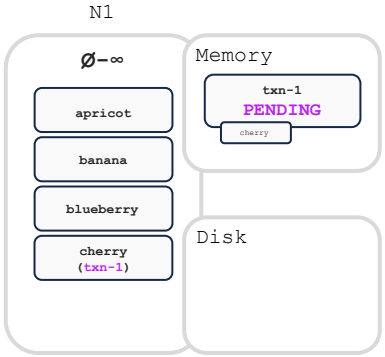
Single-node transactions

```
BEGIN TXN;
```



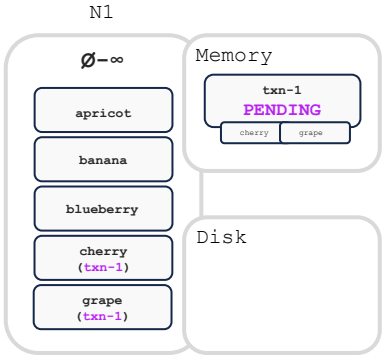
Single-node transactions

```
BEGIN TXN;
    PUT "cherry";
```



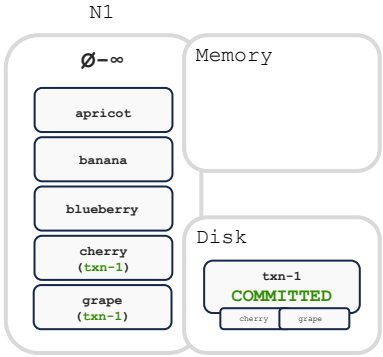
Single-node transactions

```
BEGIN TXN;
    PUT "cherry";
    PUT "grape";
```



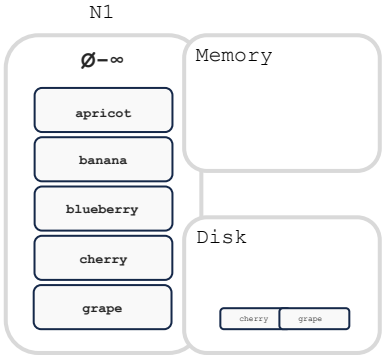
Single-node transactions

```
BEGIN TXN;
    PUT "cherry";
    PUT "grape";
COMMIT;
```



Single-node transactions

```
BEGIN TXN;
    PUT “cherry”;
    PUT “grape”;
COMMIT;
-- clean-up txn record
```



Single-node (unreplicated) transactions

Atomicity and durability are achieved by bootstrapping off a lower-level atomic/durable primitive: disk writes (fsync)

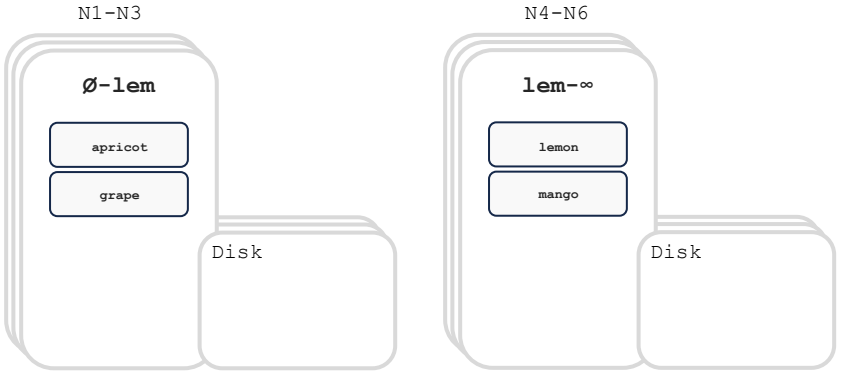
- `PENDING` transaction record, with transaction ID
- Staged (in-memory) writes, tagged with transaction ID
- Durably persist `COMMITTED` transaction record and staged writes, atomically

Multi-node (replicated) transactions

Atomicity and durability are achieved by bootstrapping off a lower-level atomic/durable primitive: consensus writes (RTT + fsync)

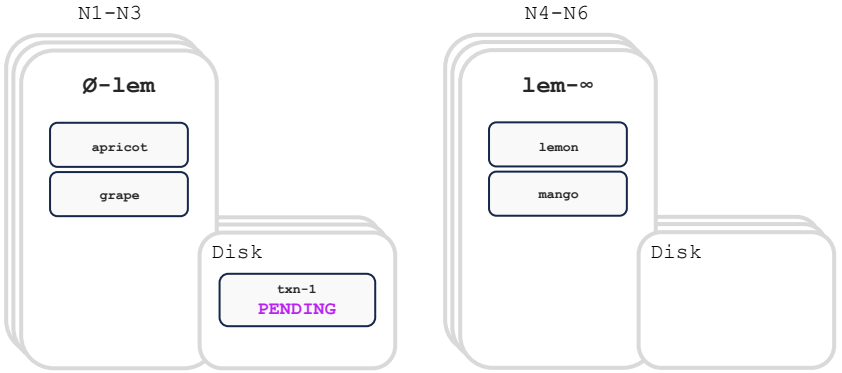
- `PENDING` transaction record, with transaction ID
- Staged (consensus) writes, tagged with transaction ID
- Durably persist `COMMITTED` transaction record, atomically

Multi-node transactions



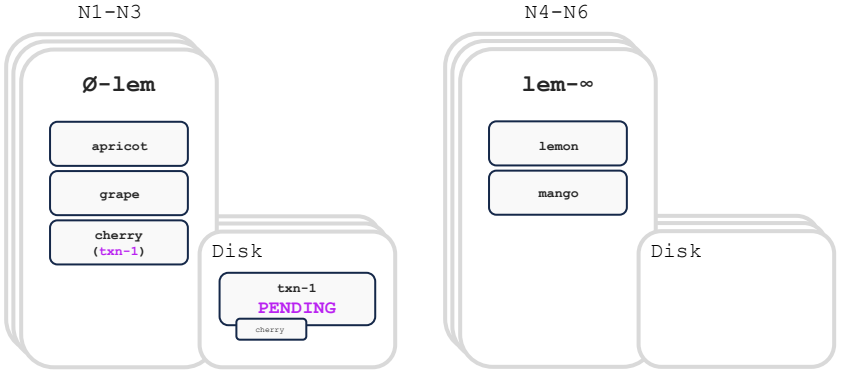
Multi-node transactions

```
BEGIN TXN;
```



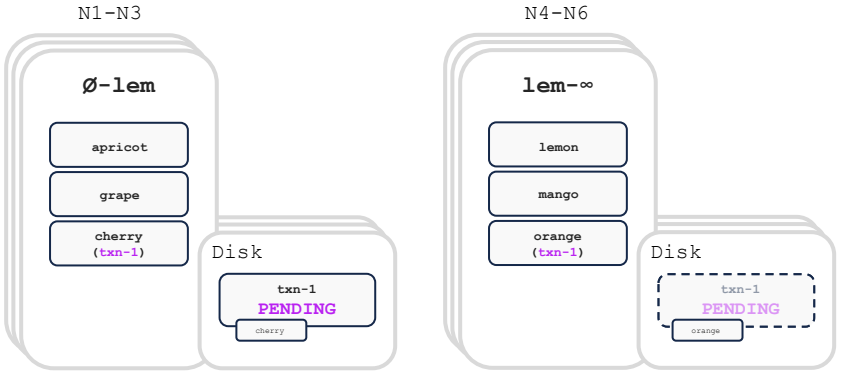
Multi-node transactions

```
BEGIN TXN;
    PUT "cherry";
```



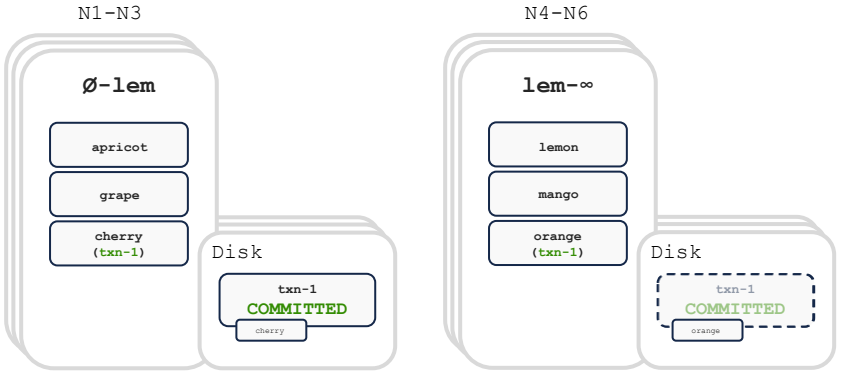
Multi-node transactions

```
BEGIN TXN;
    PUT "cherry";
    PUT "orange";
```



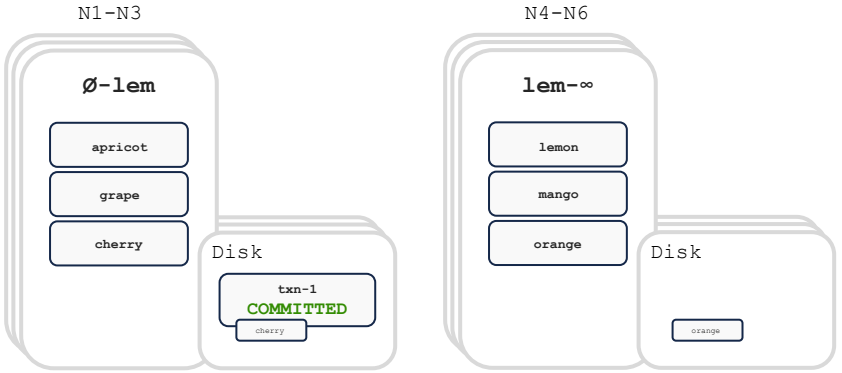
Multi-node transactions

```
BEGIN TXN;
    PUT "cherry";
    PUT "orange";
COMMIT;
```



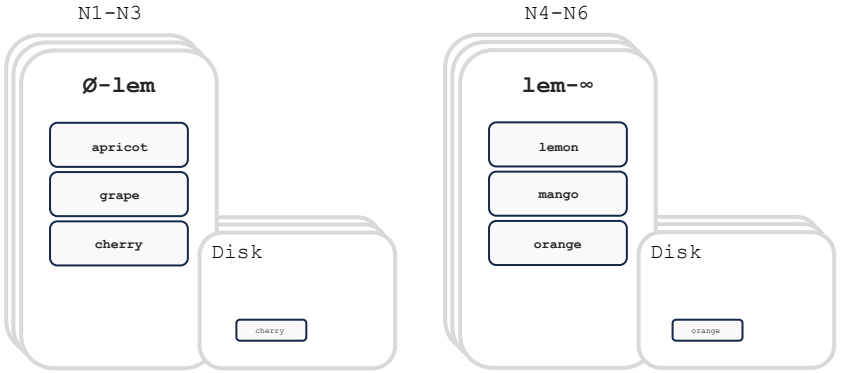
Multi-node transactions

```
BEGIN TXN;
    PUT “cherry”;
    PUT “orange”;
COMMIT;
-- clean-up staged writes
```



Multi-node transactions

```
BEGIN TXN;
    PUT “cherry”;
    PUT “orange”;
COMMIT;
-- clean-up staged writes
-- clean-up txn record
```



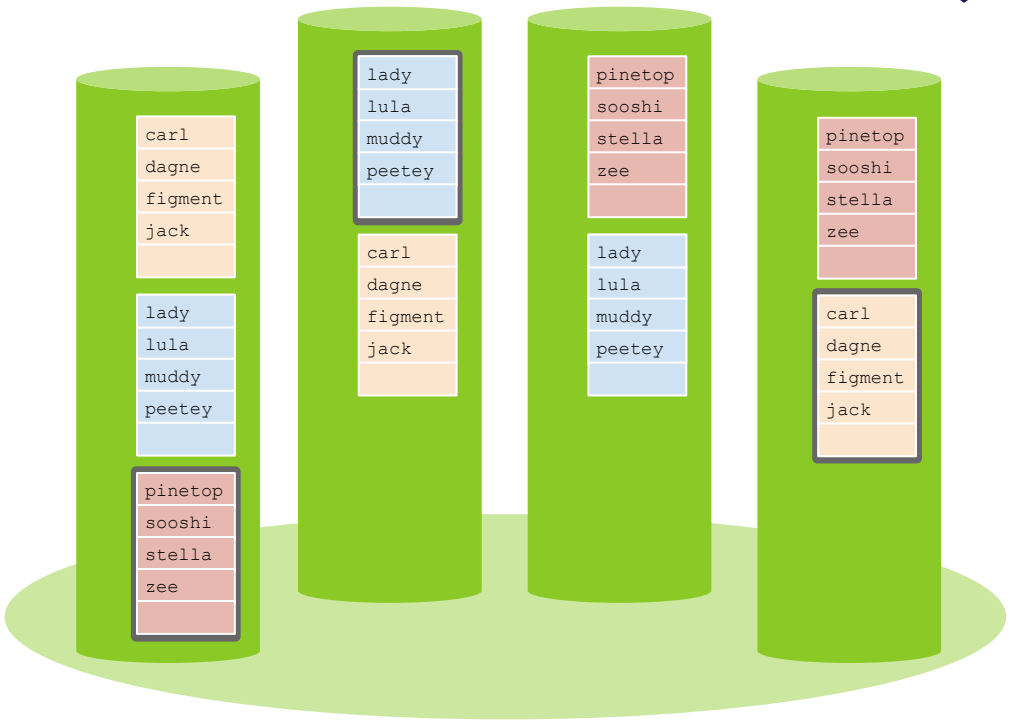
Agenda

1. Foundations
2. **Transactions**
3. Implementations

I. ACID & Isolation Levels
II. Transaction Basics
III. Unpipelined Transactions

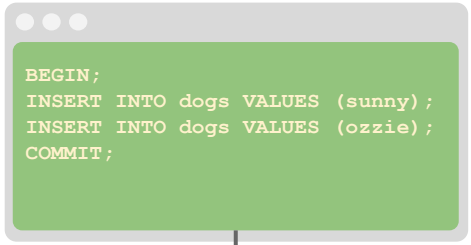
Unpipelined Transactions

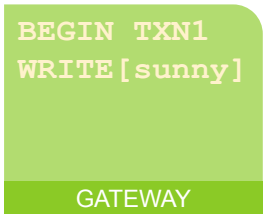
```
BEGIN;
INSERT INTO dogs VALUES (sunny);
INSERT INTO dogs VALUES (ozzie);
COMMIT;
```

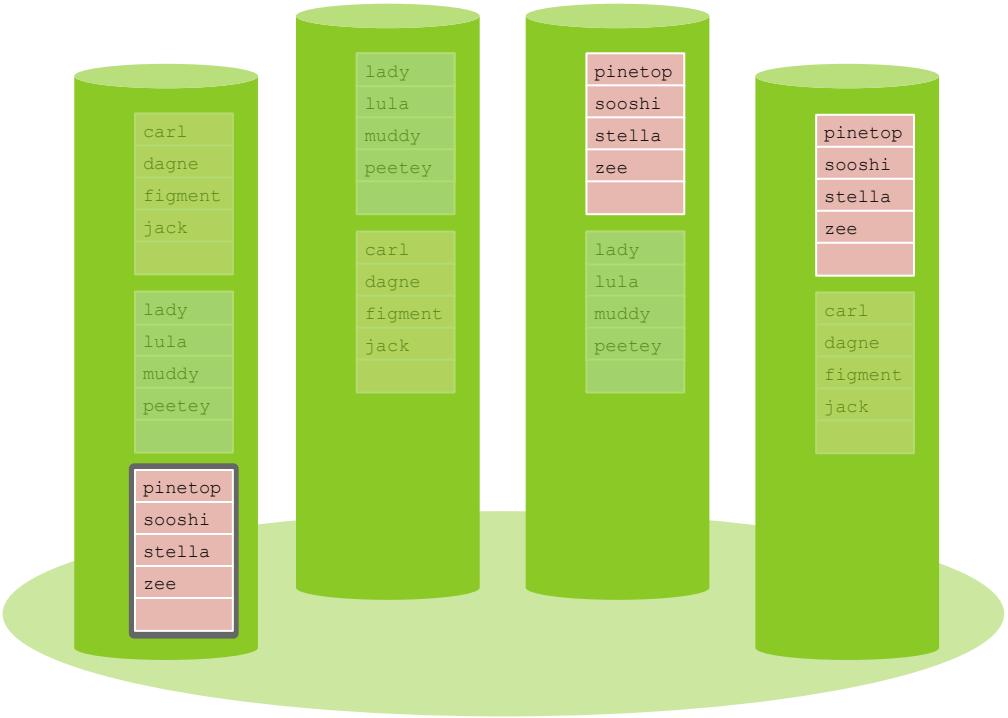


t = 0

Unpipelined Transactions

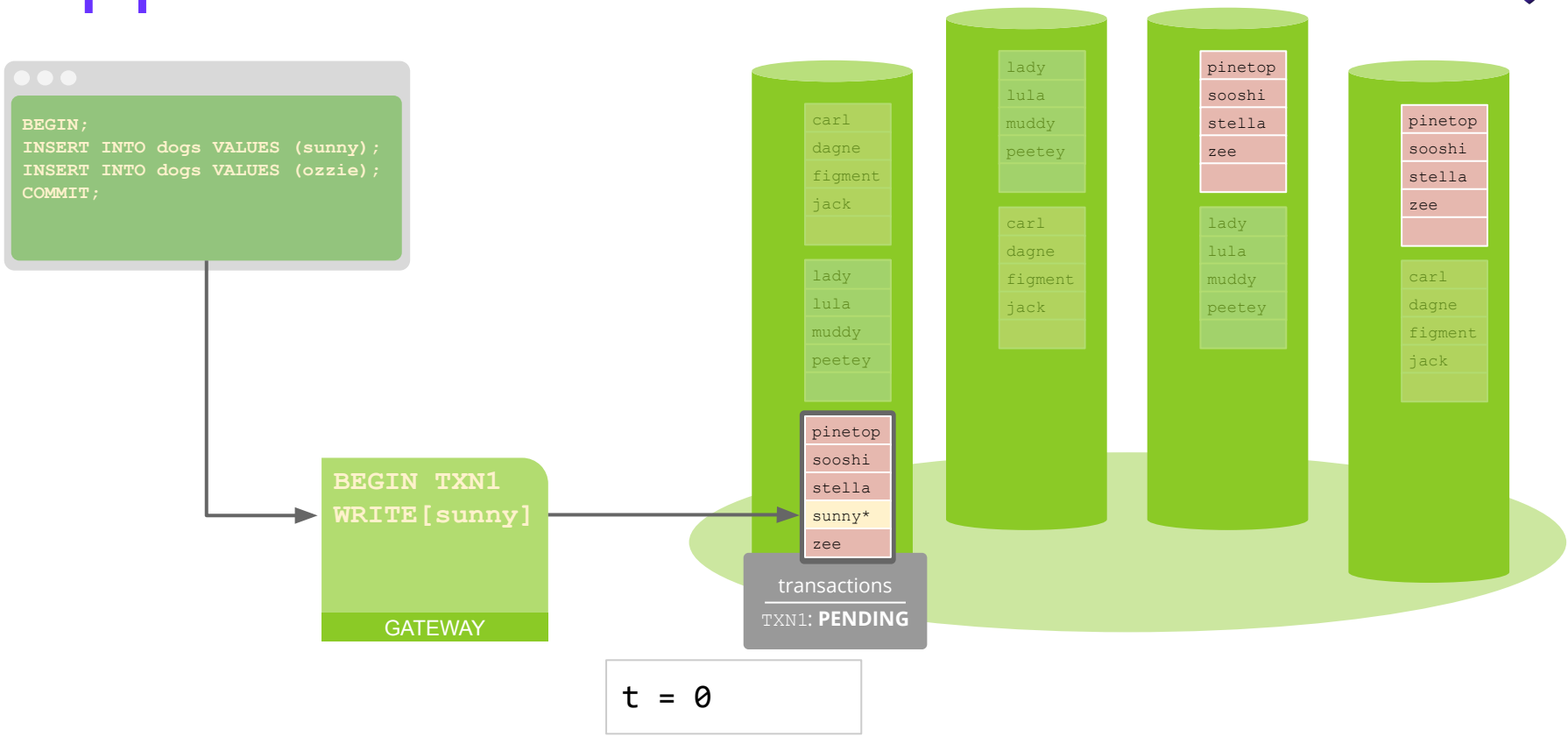




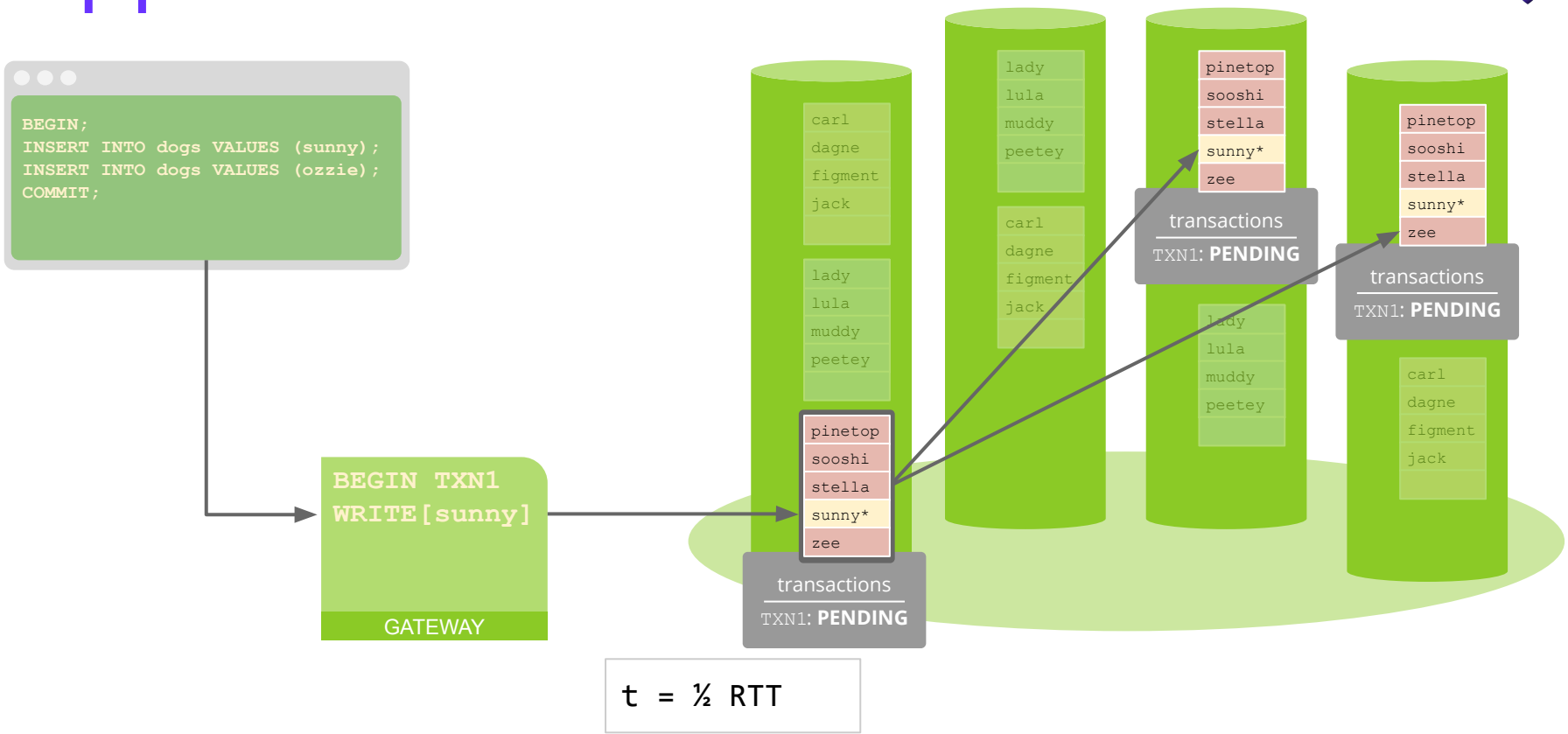


t = 0

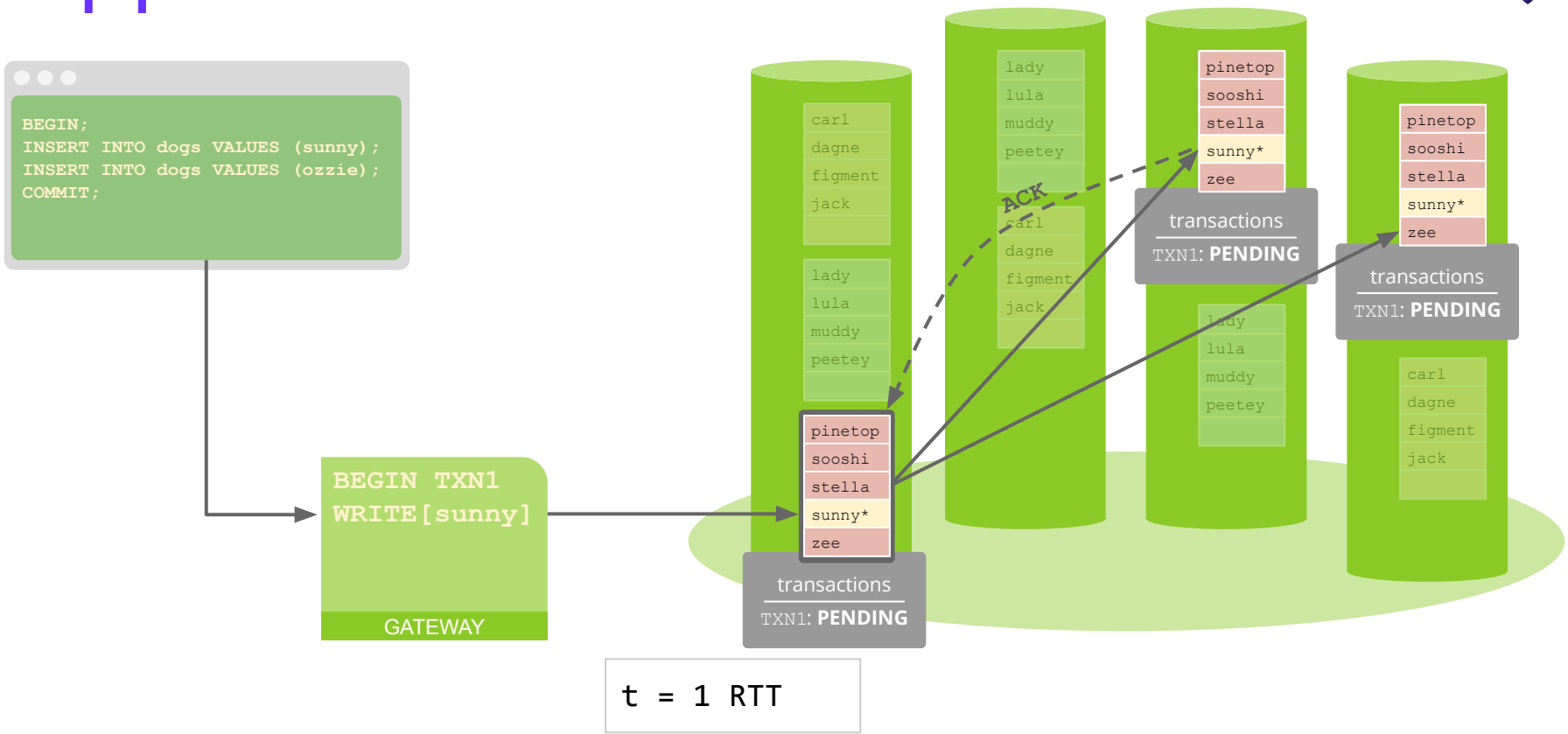
Unpipelined Transactions



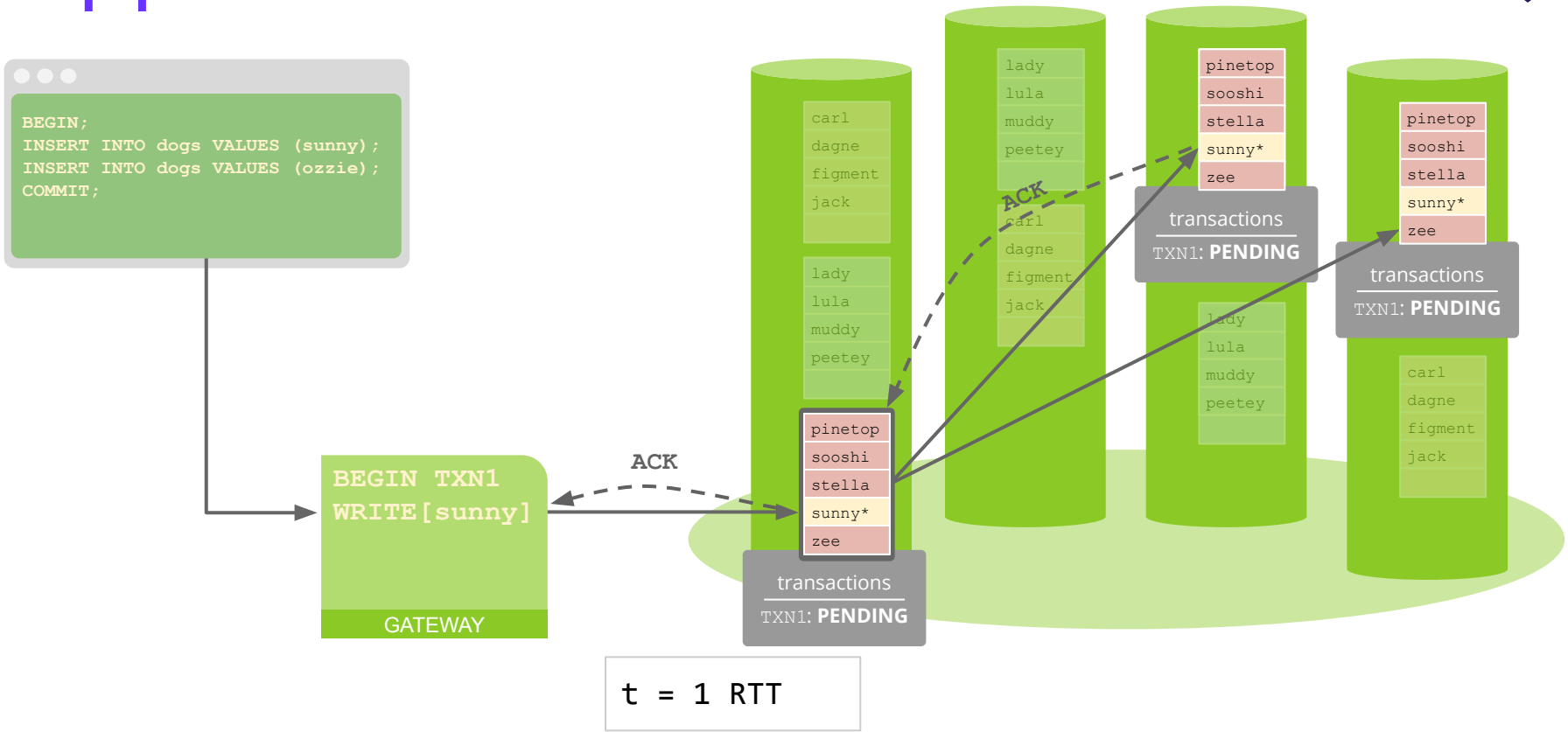
Unpipelined Transactions



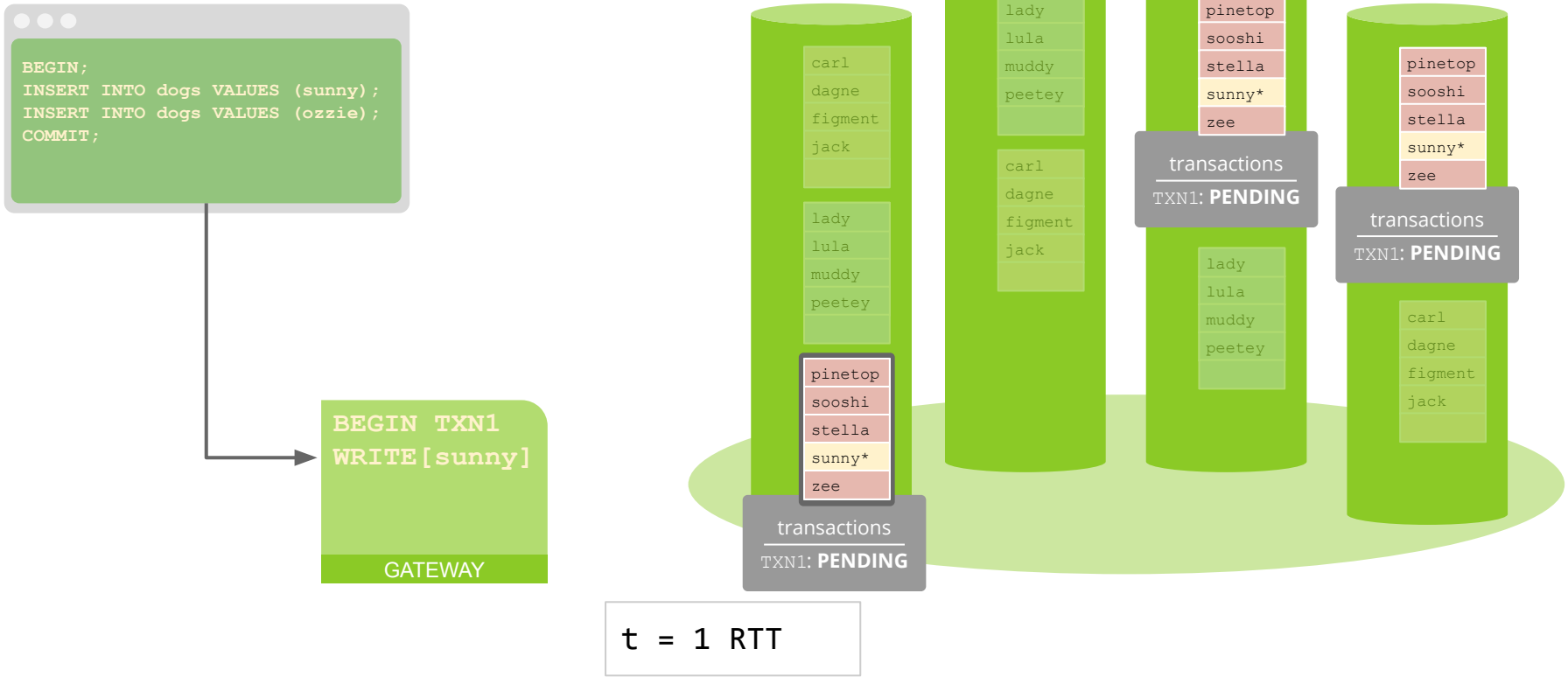
Unpipelined Transactions



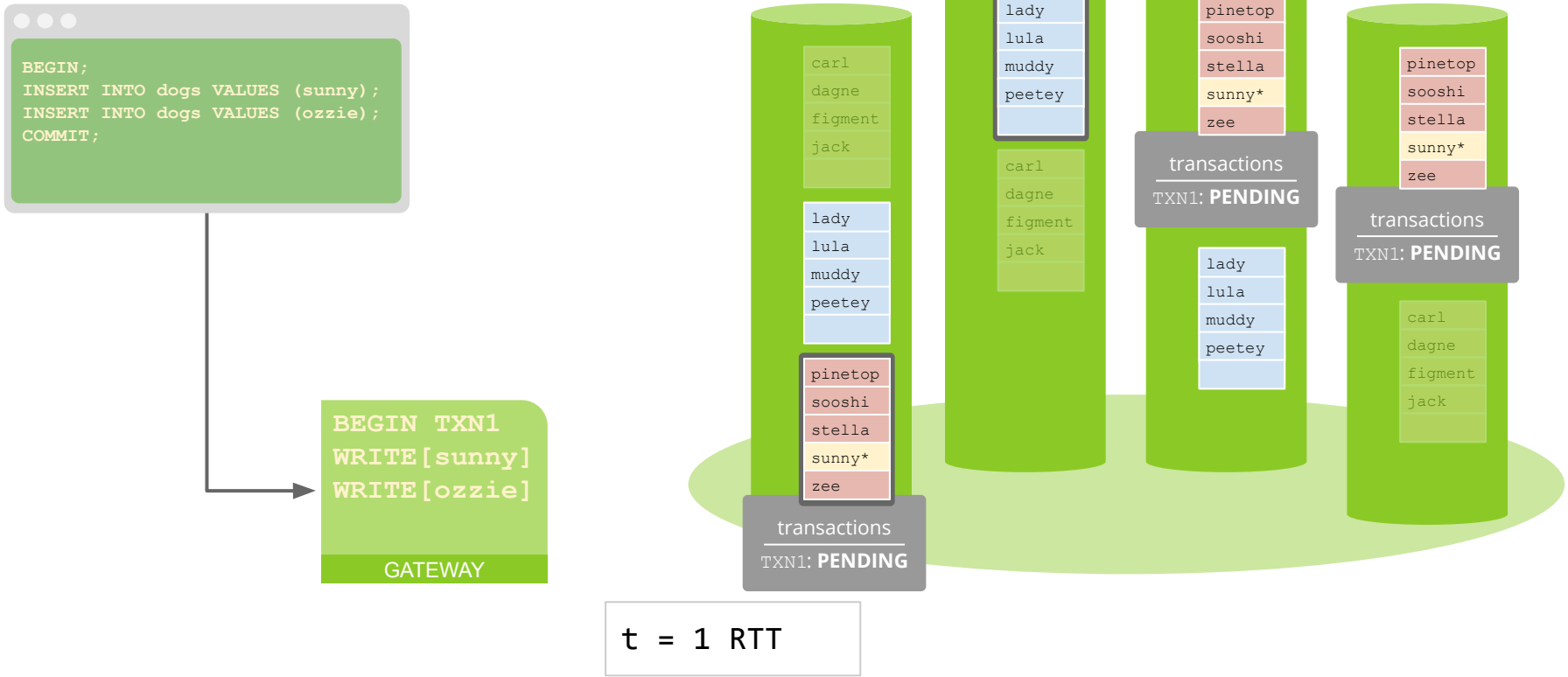
Unpipelined Transactions



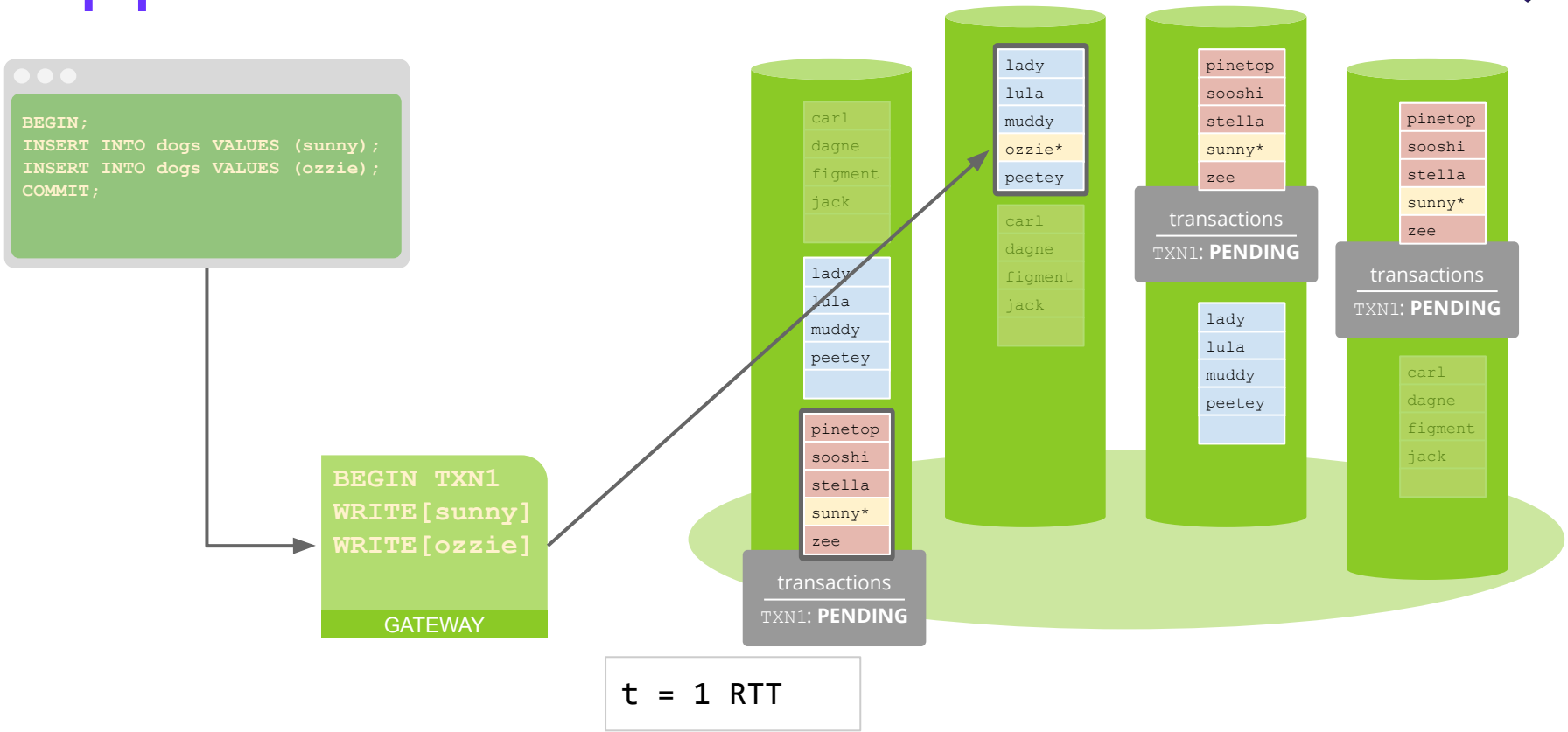
Unpipelined Transactions



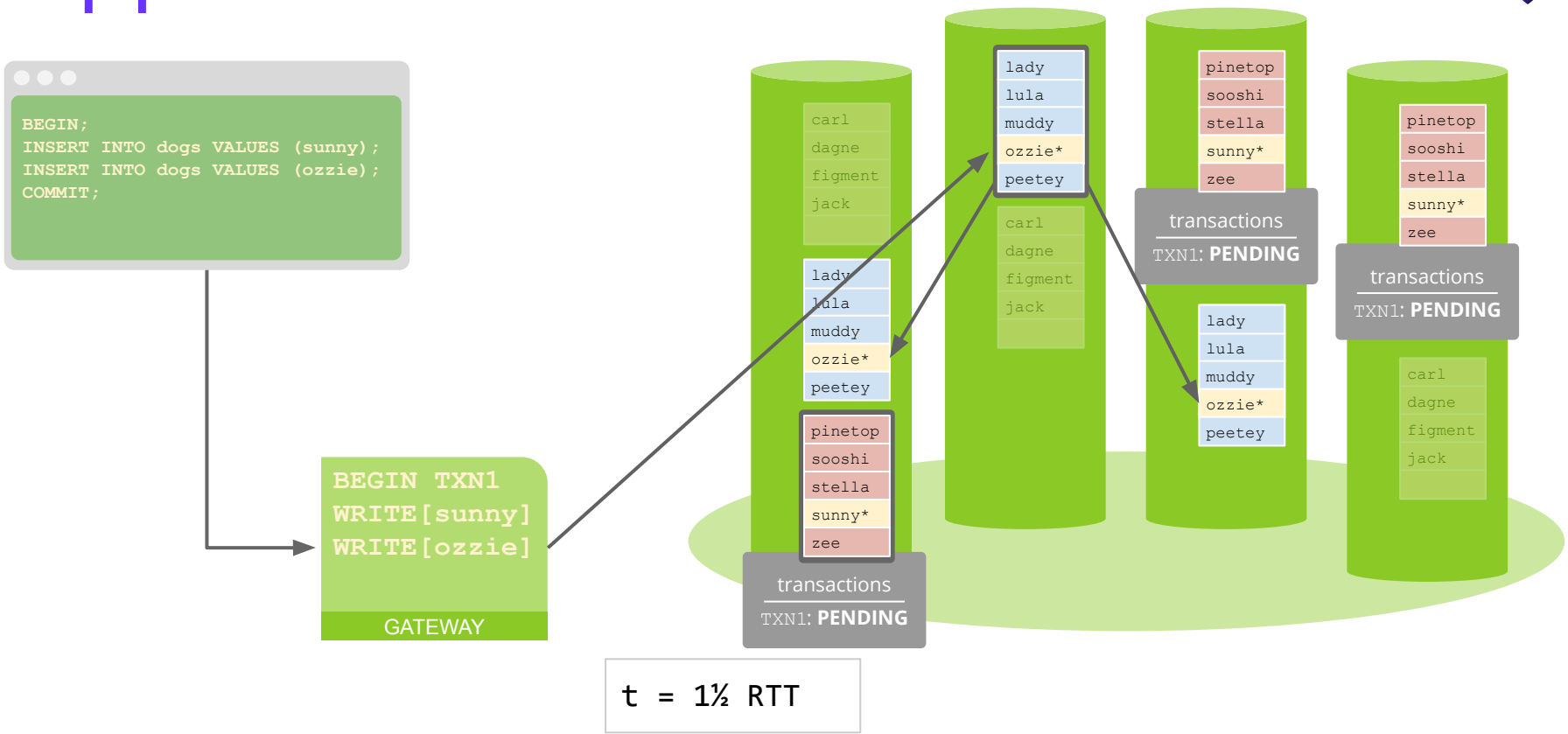
Unpipelined Transactions



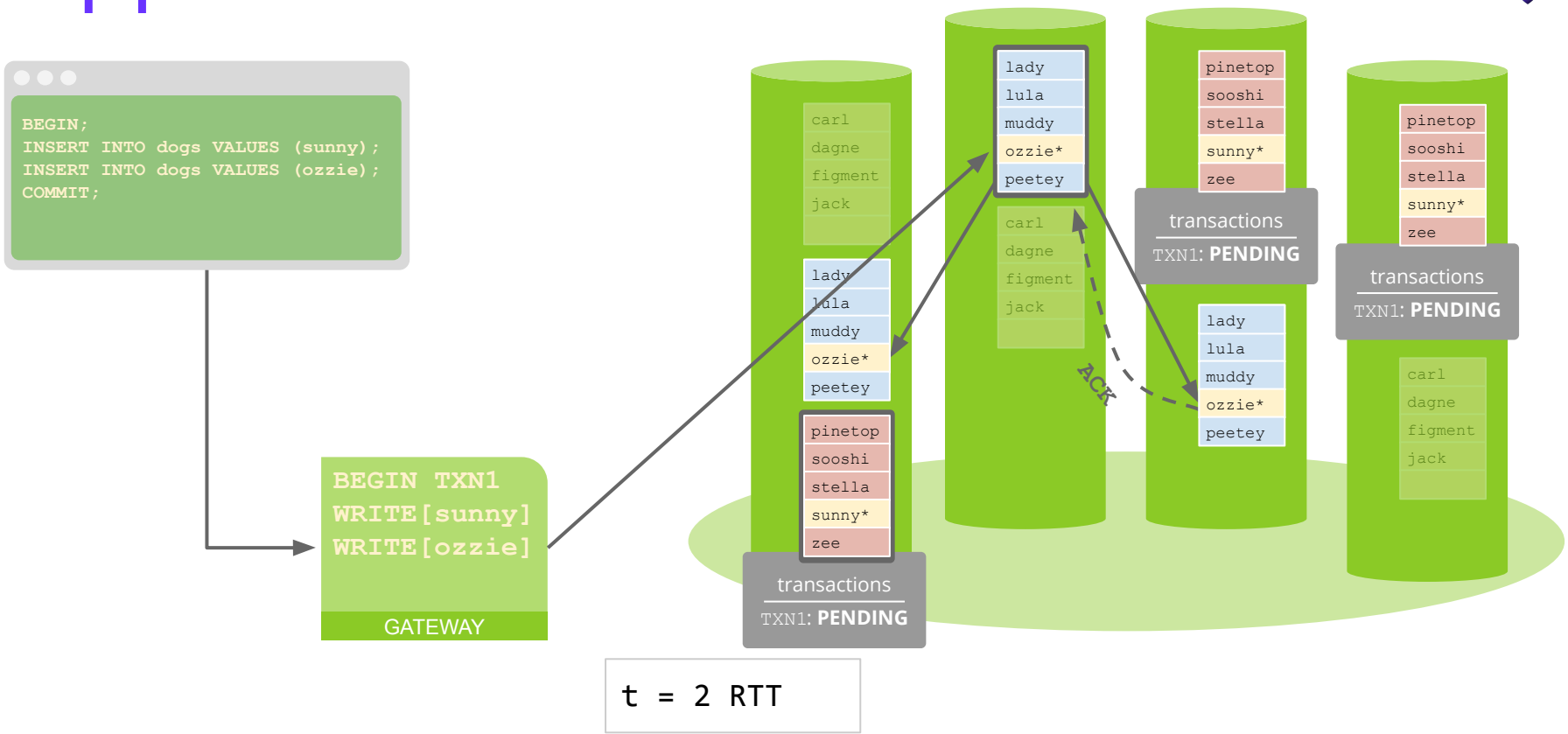
Unpipelined Transactions



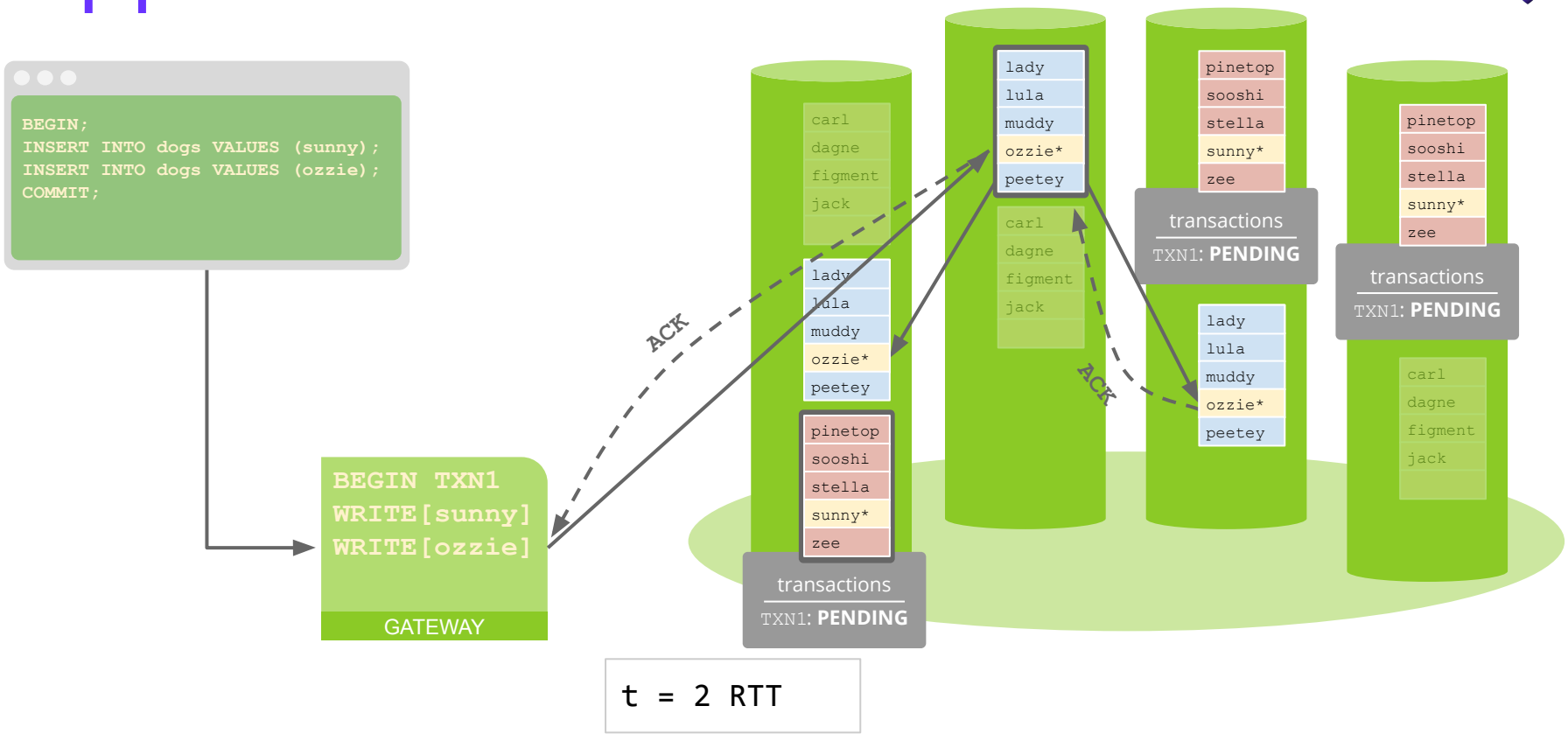
Unpipelined Transactions



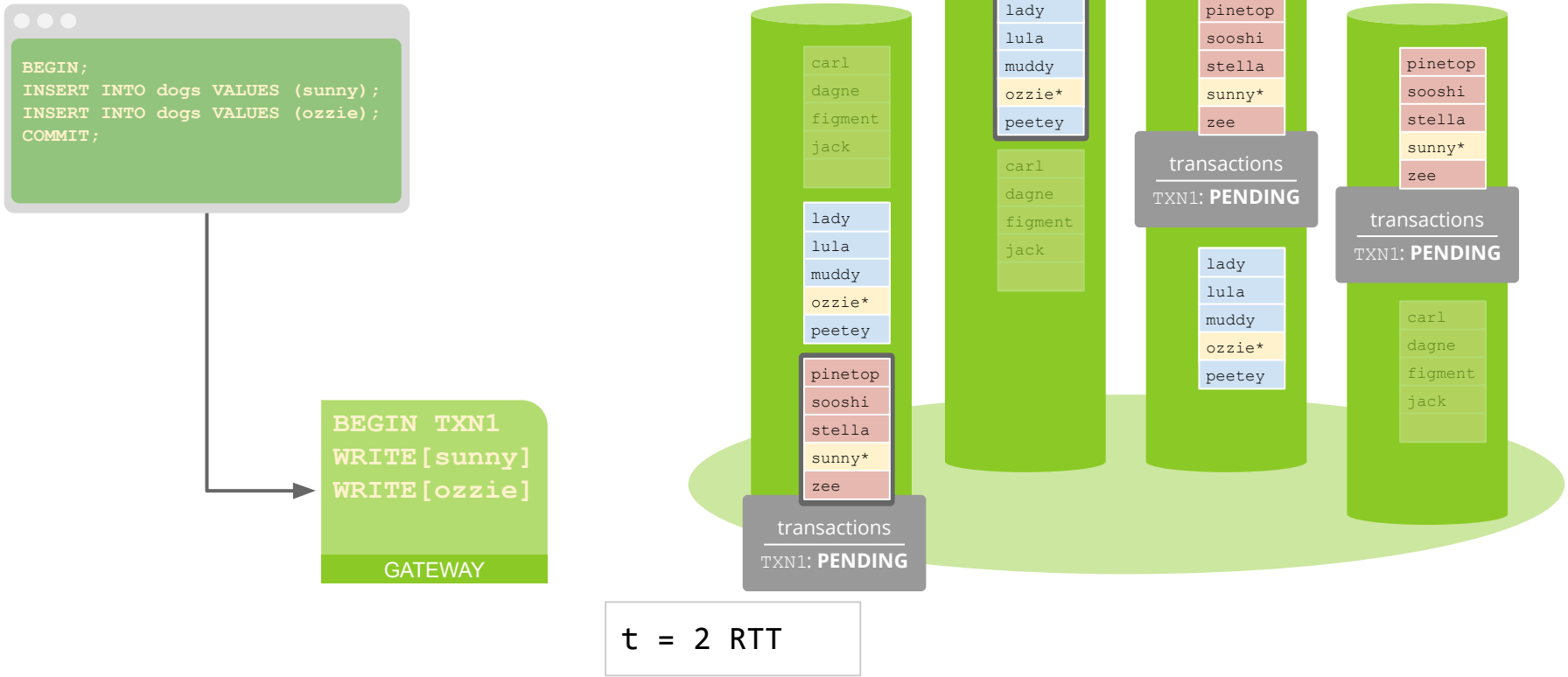
Unpipelined Transactions



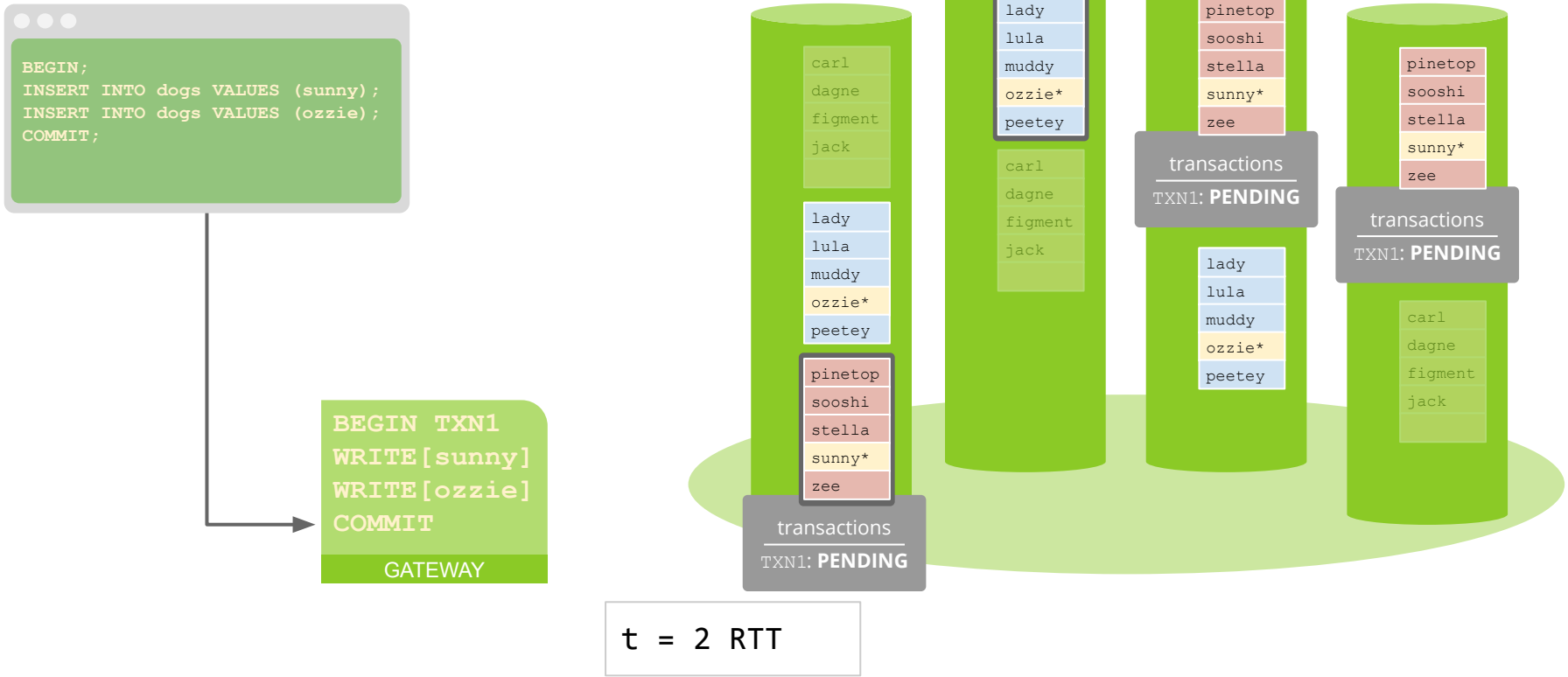
Unpipelined Transactions



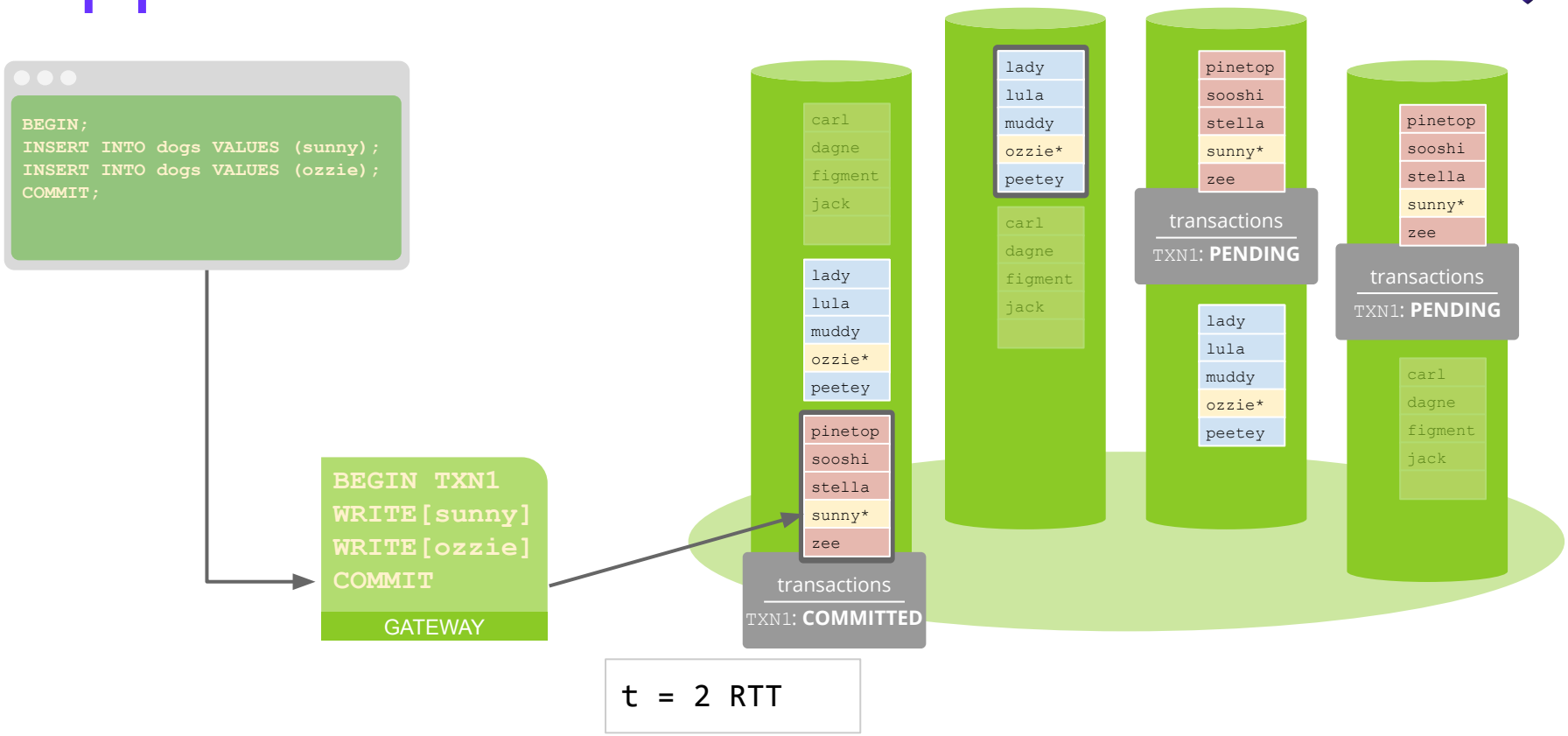
Unpipelined Transactions



Unpipelined Transactions

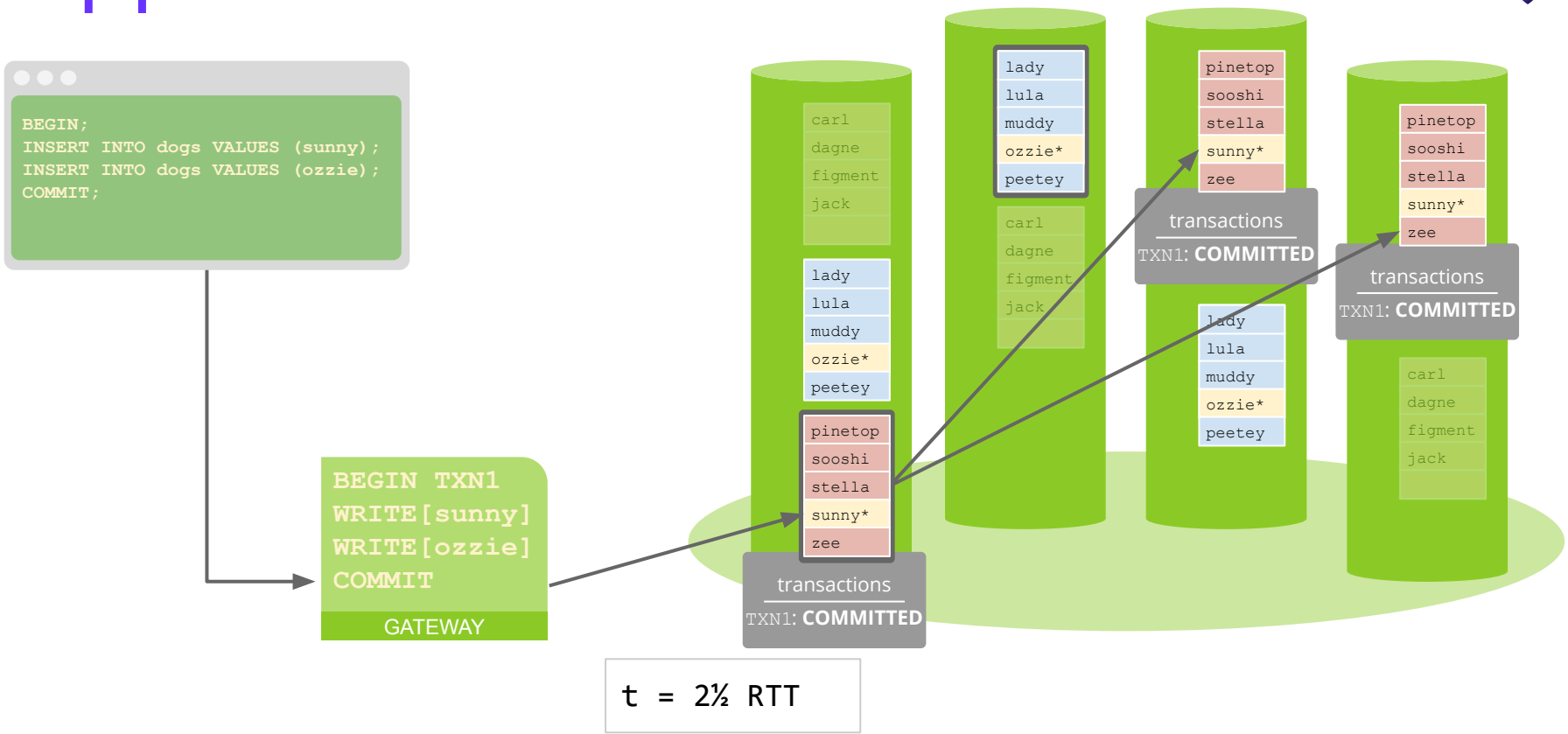


Unpipelined Transactions

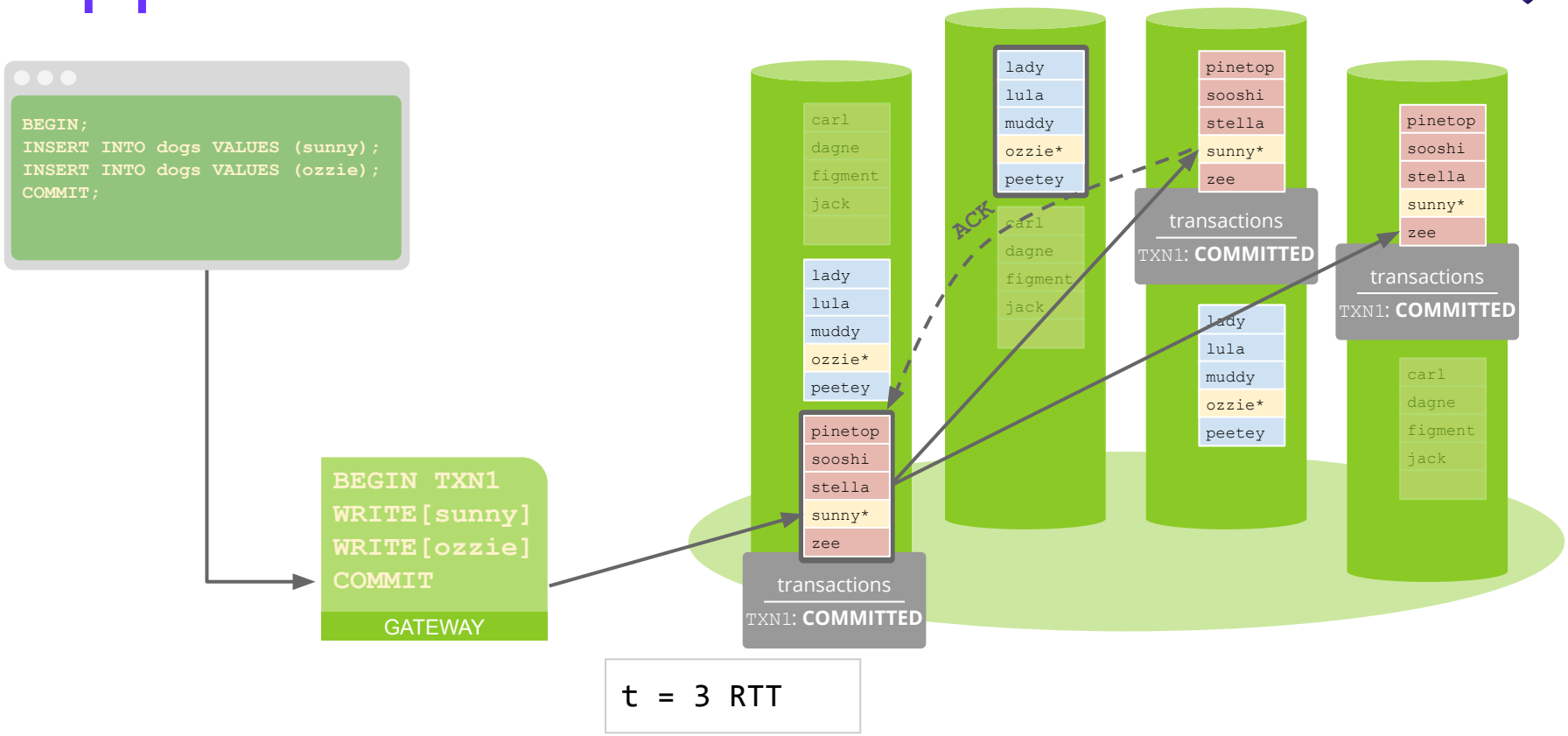


t = 2 RTT

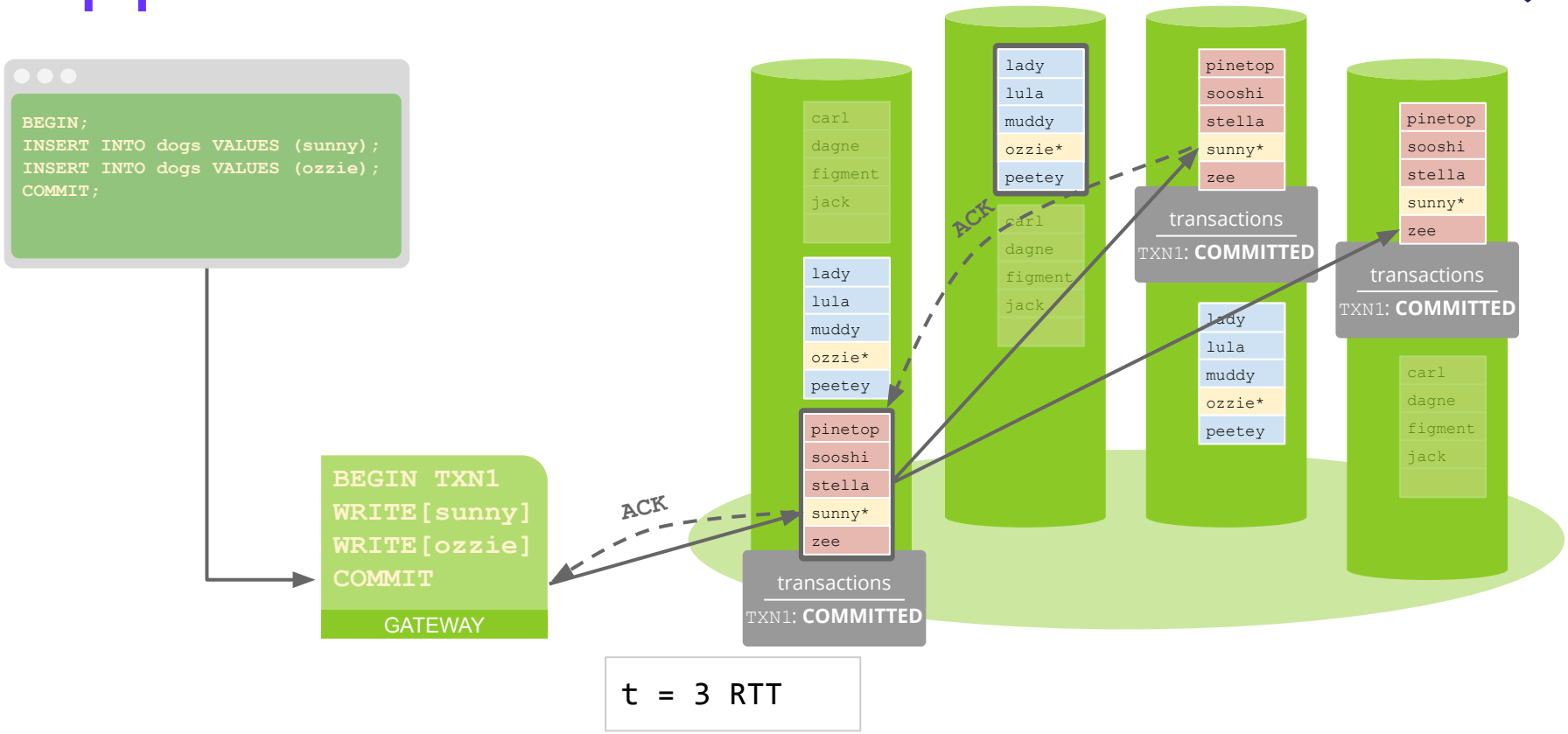
Unpipelined Transactions



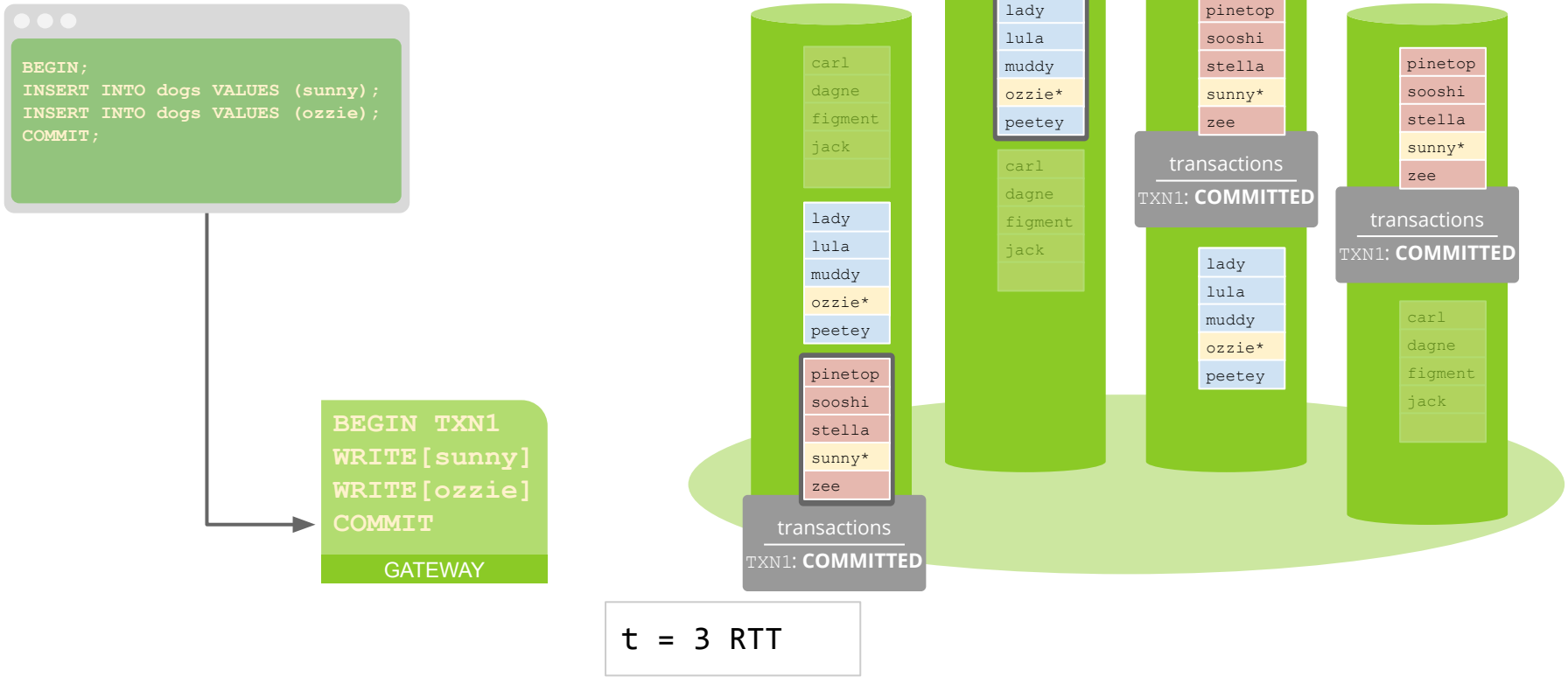
Unpipelined Transactions



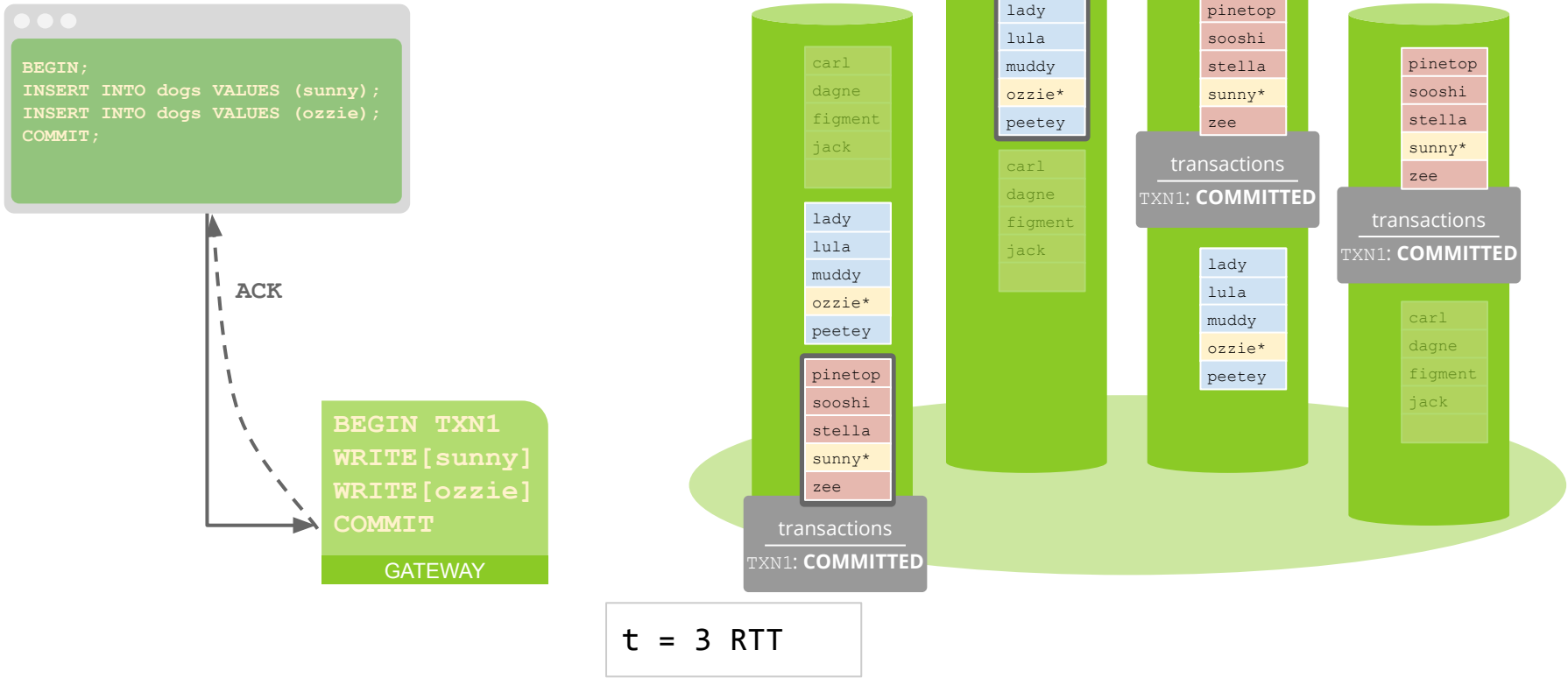
Unpipelined Transactions



Unpipelined Transactions

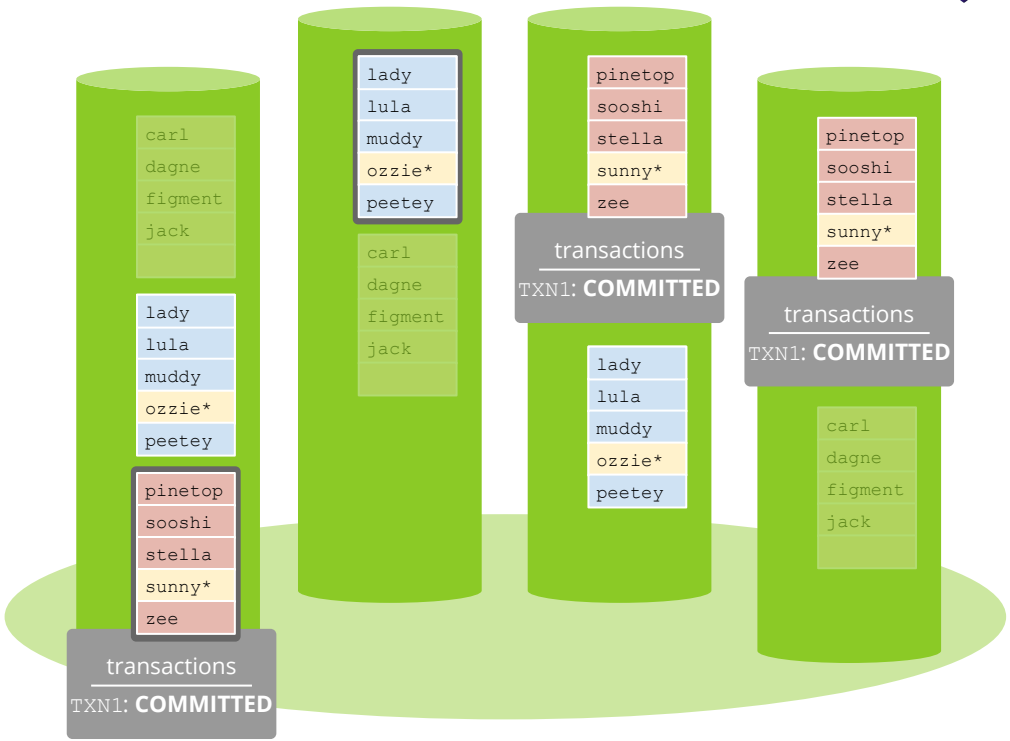


Unpipelined Transactions



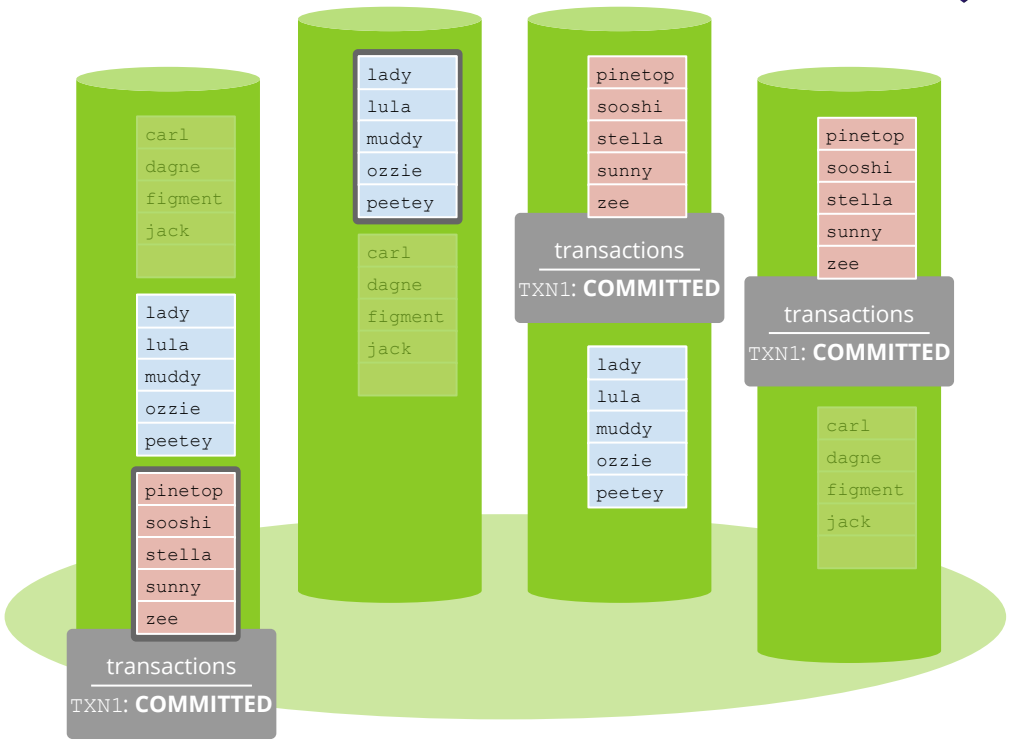
t = 3 RTT

Unpipelined Transactions



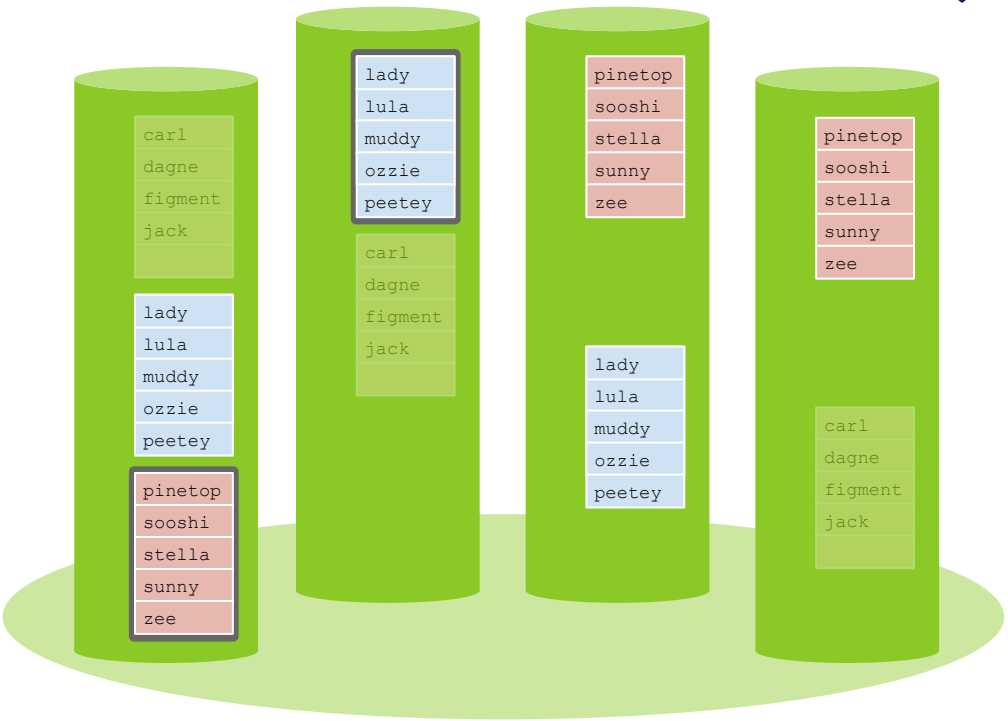
t = 3 RTT

Unpipelined Transactions



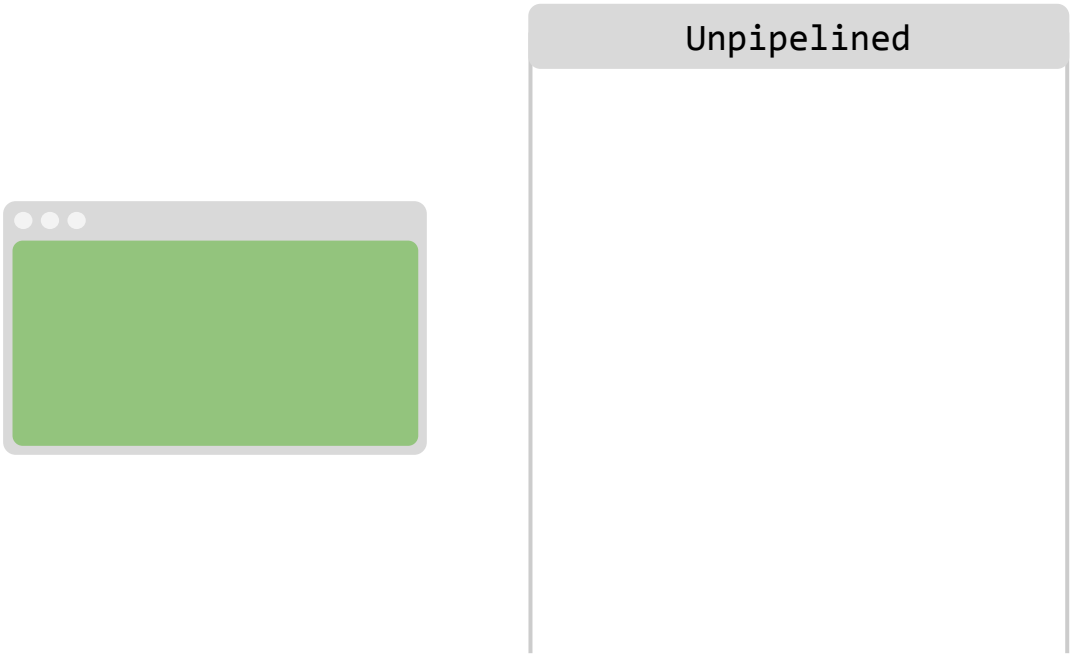
t = 3 RTT

Unpipelined Transactions

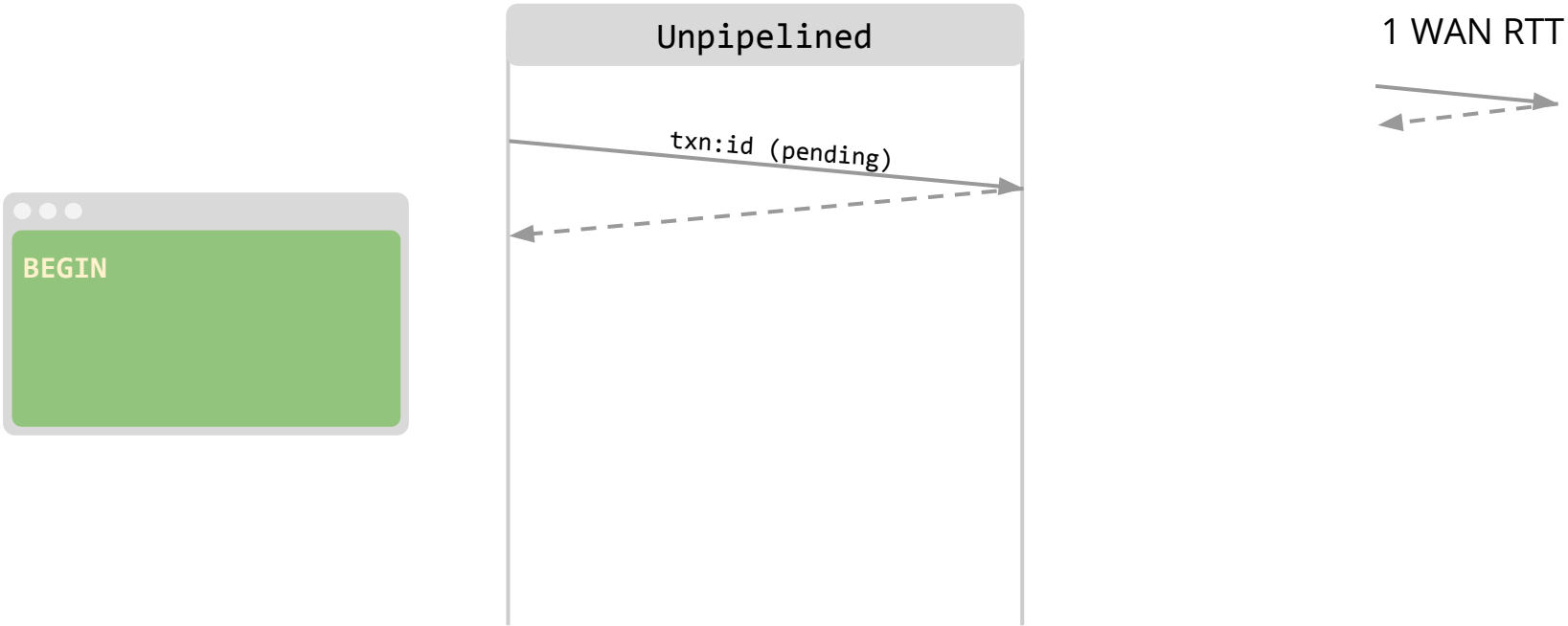



```
t = 3 RTT
```

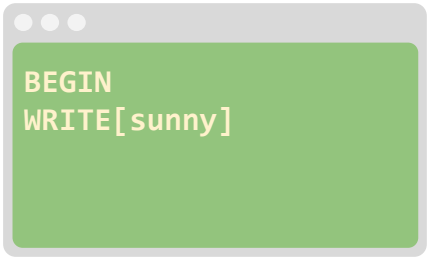
Timeline: Unpipelined Transactions

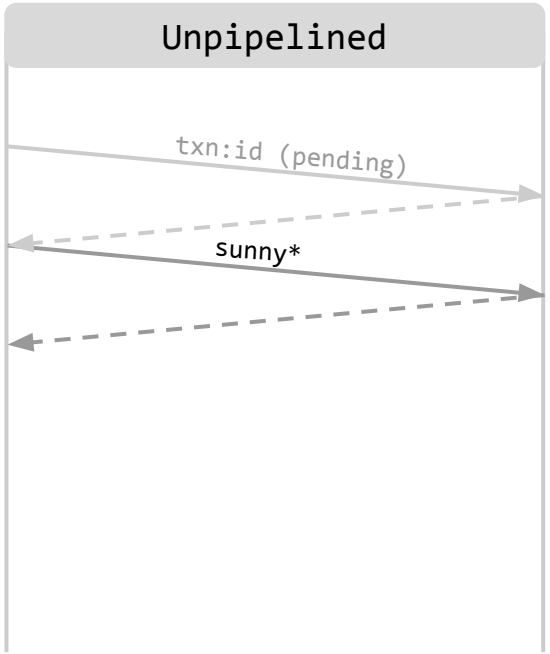


Timeline: Unpipelined Transactions

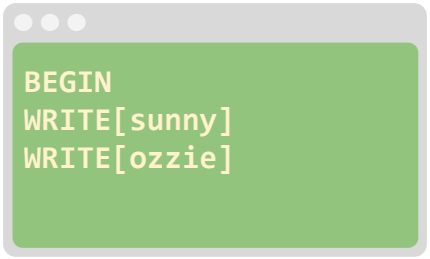


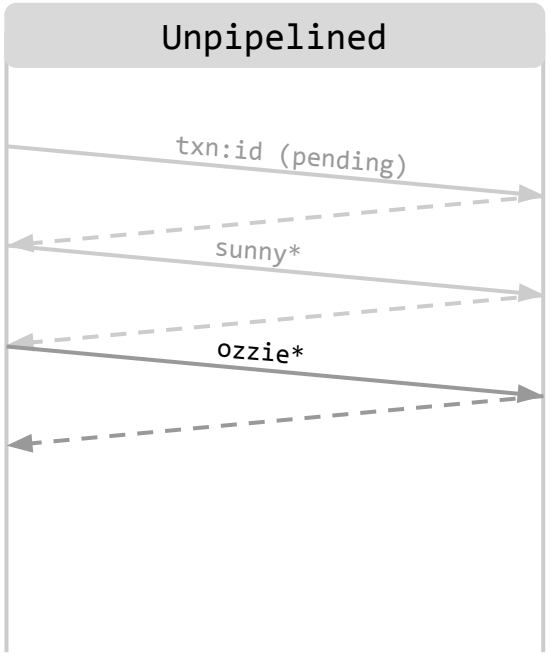
Timeline: Unpipelined Transactions



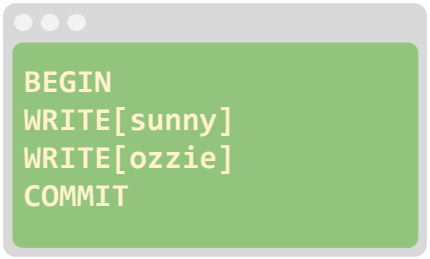


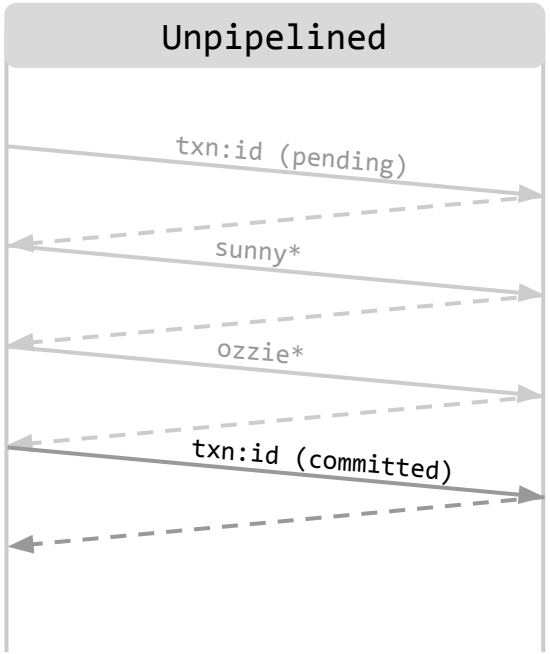
Timeline: Unpipelined Transactions





Timeline: Unpipelined Transactions





1 WAN RTT

N+2 WAN RTTS

Agenda

1. Foundations
2. Transactions
3. **Implementations**

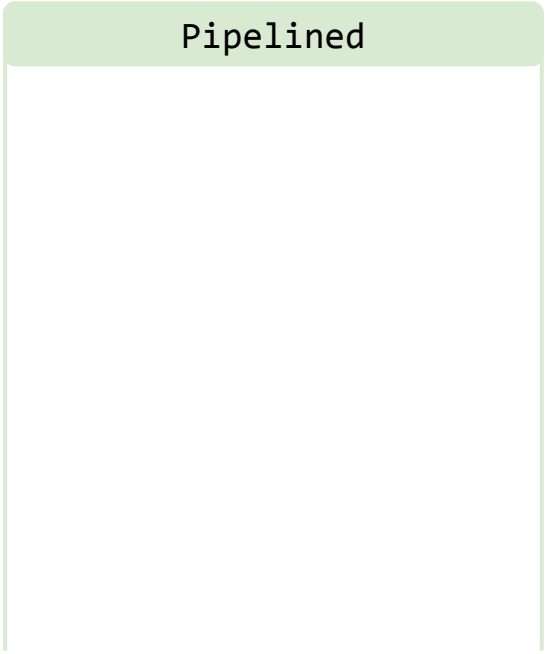
I. Spanner/Pipelined Transactions
II. Parallel Commits
III. Replicated Commit
IV. Carousel
V. MDCC
VI. SLOG/OceanVista
VII. TAPIR

Agenda

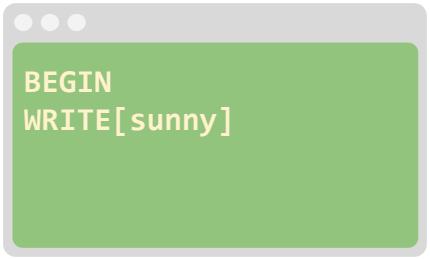
1. Foundations
2. Transactions
3. **Implementations**

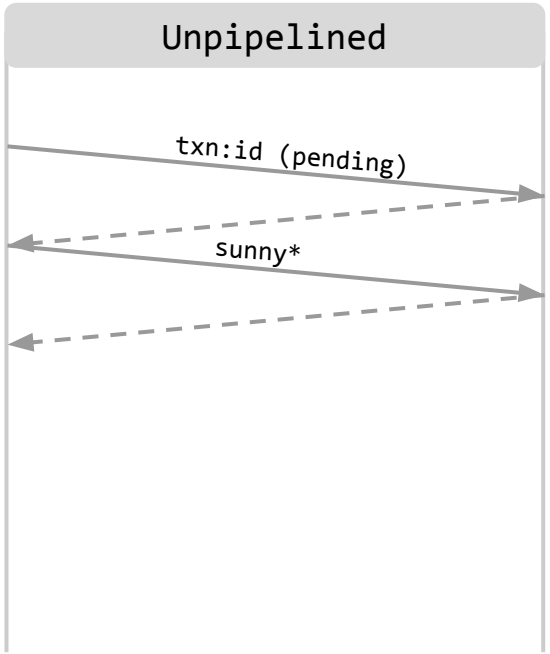
I. Spanner/Pipelined Transactions
II. Parallel Commits
III. Replicated Commit
IV. Carousel
V. MDCC
VI. SLOG/OceanVista
VII. TAPIR

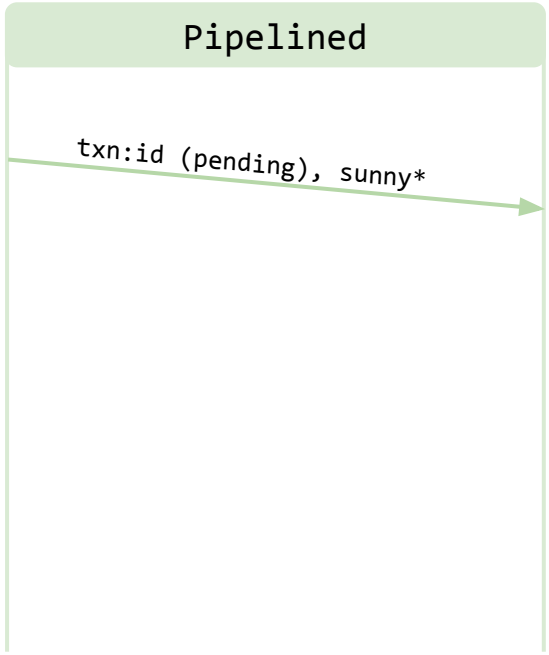
Timeline: Pipelined Transactions



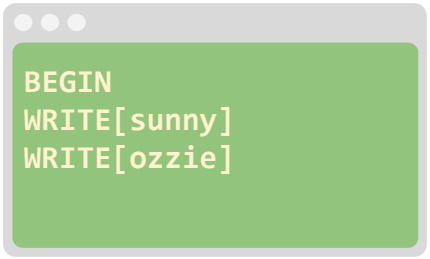
Timeline: Pipelined Transactions

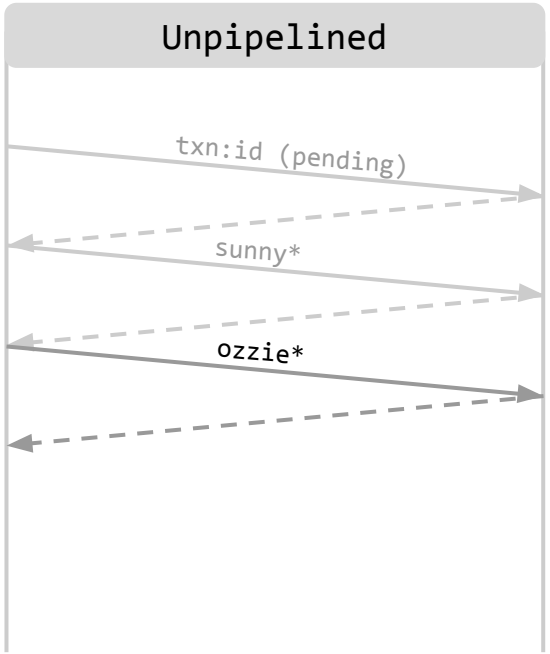


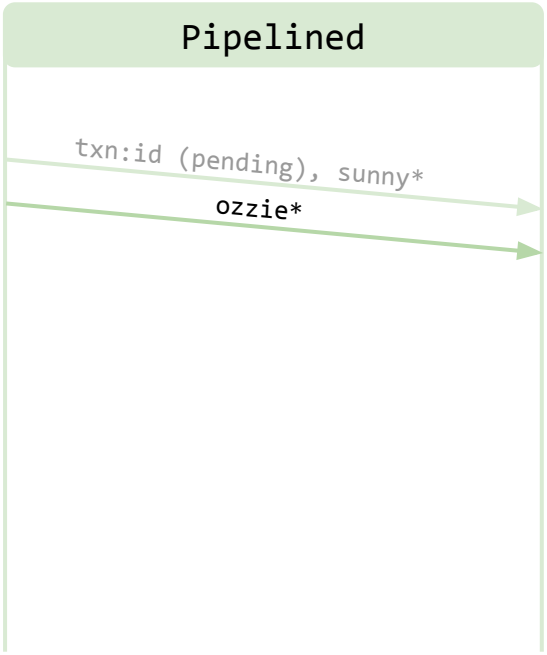




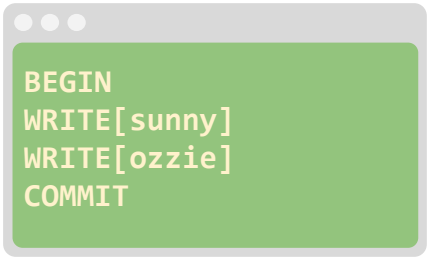
Timeline: Pipelined Transactions

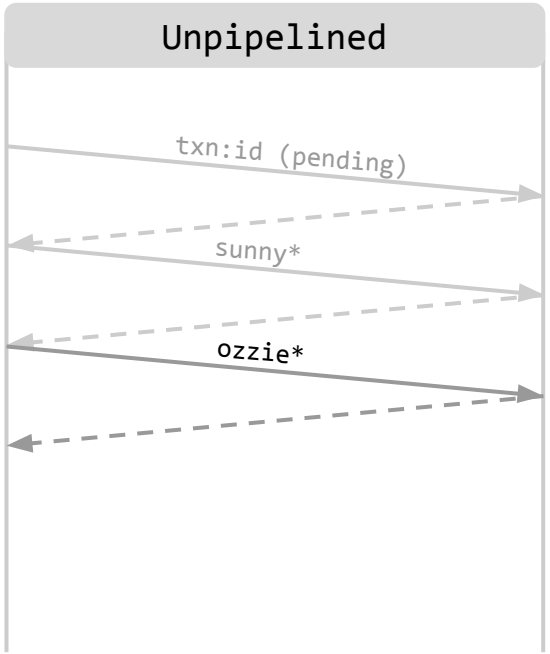


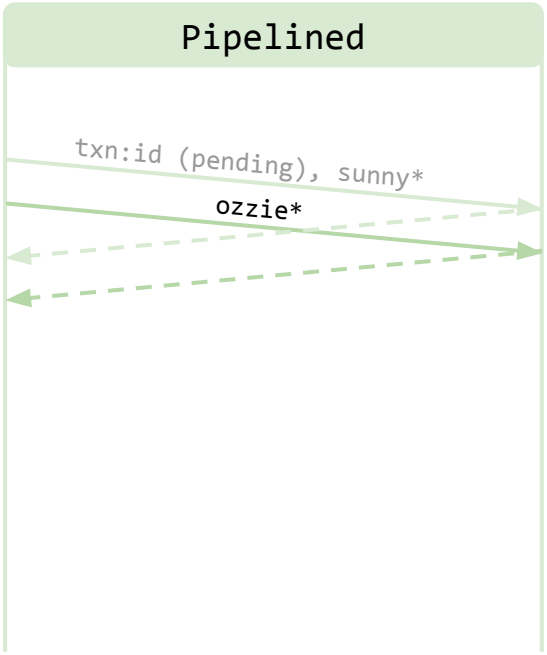




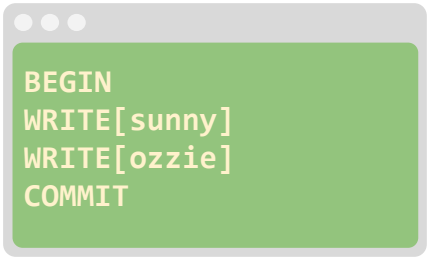
Timeline: Pipelined Transactions

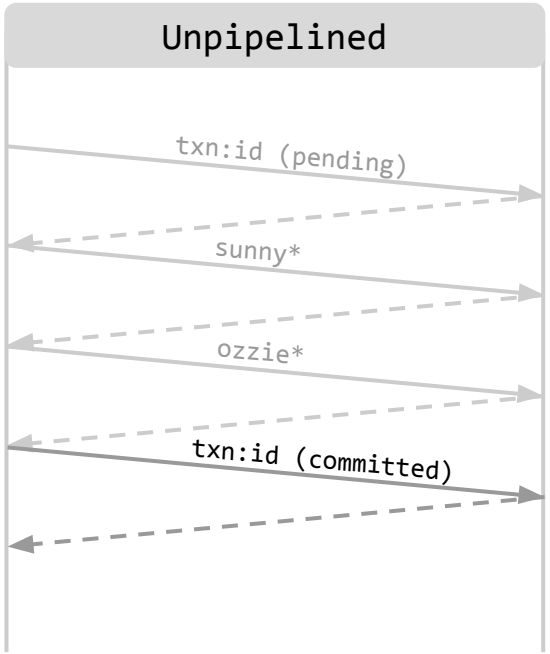


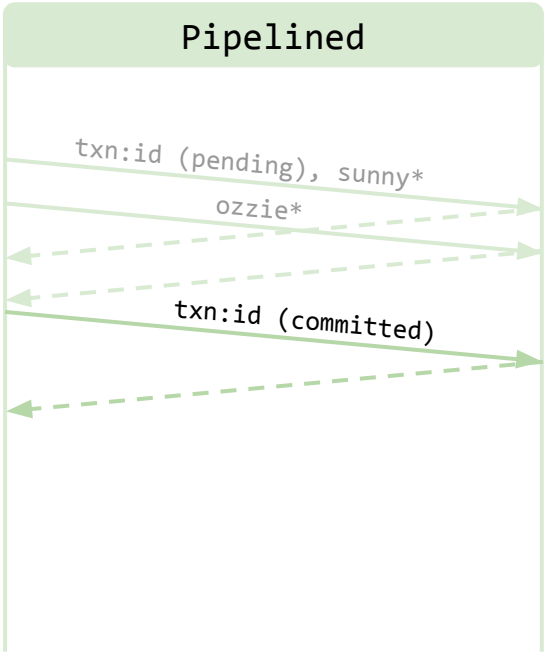




Timeline: Pipelined Transactions

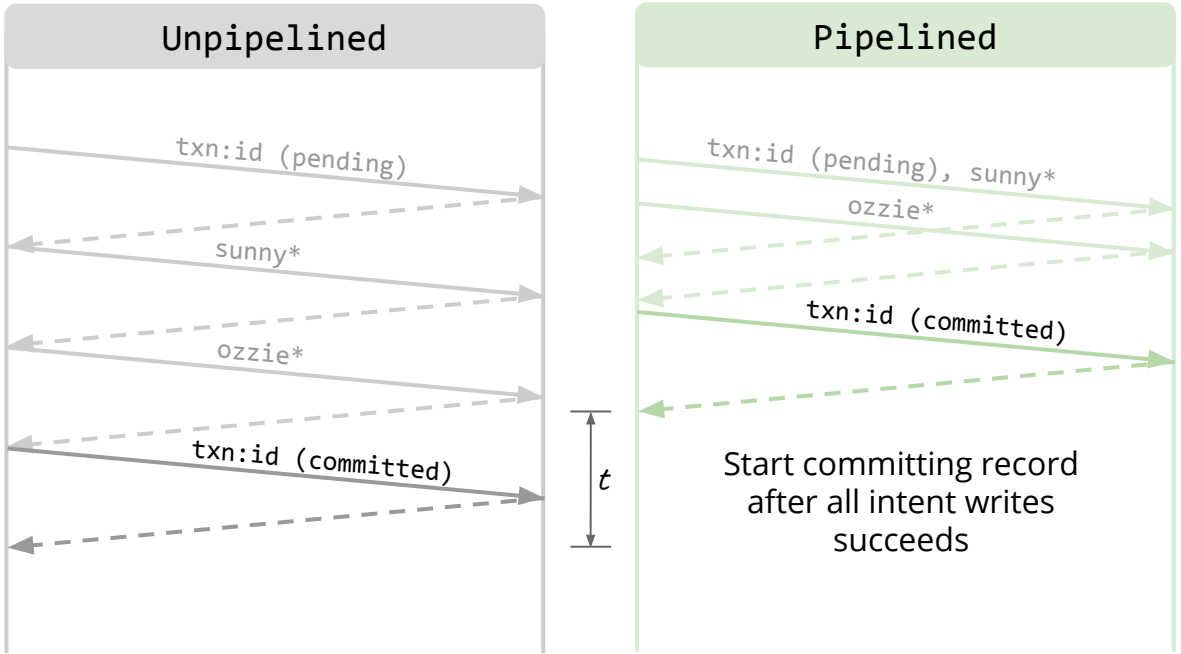






Timeline: Pipelined Transactions

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

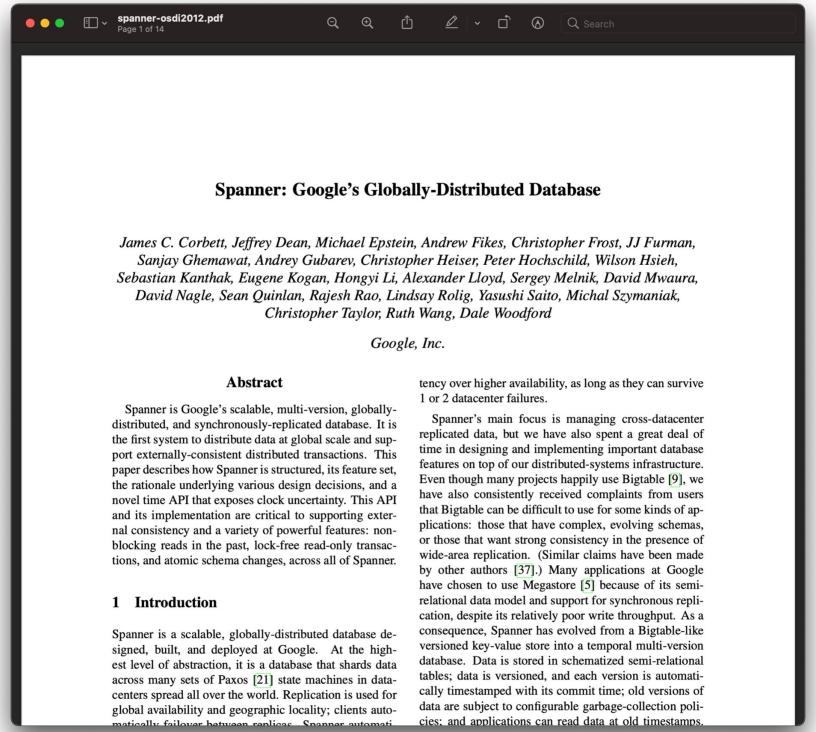


N+2 WAN RTTS

2 WAN RTTS

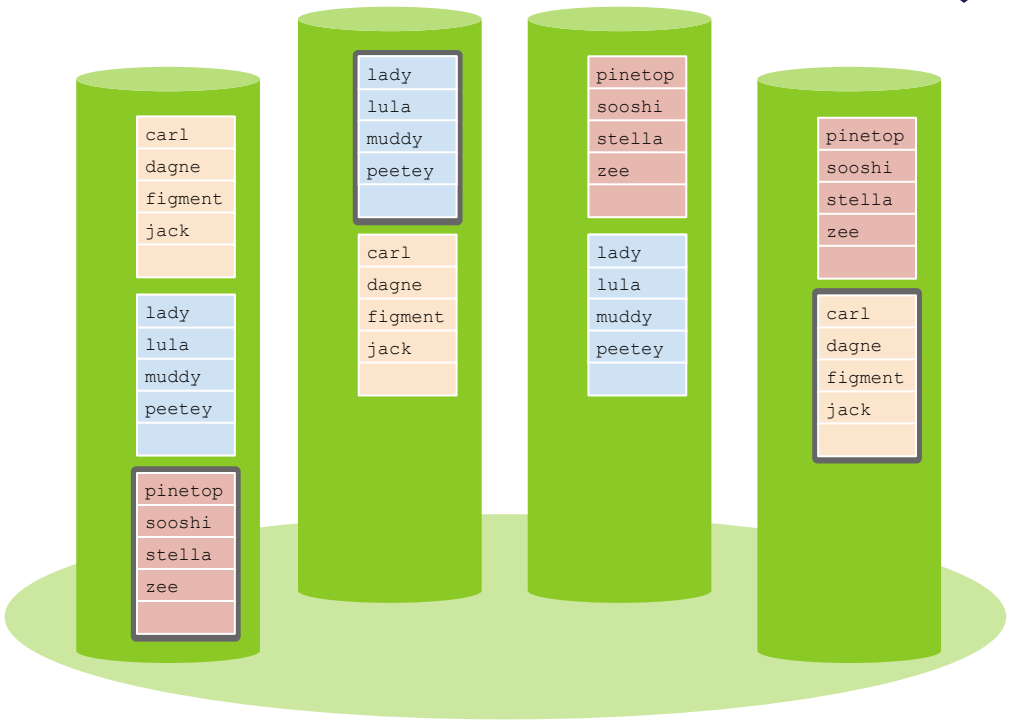
Spanner

From Google, 2012. Spurred all the following research/derivative systems.



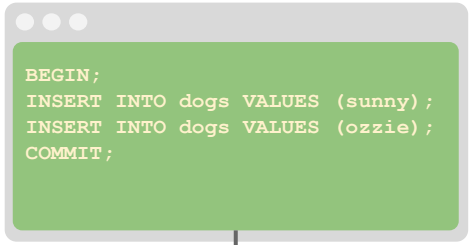
Pipelined Transactions

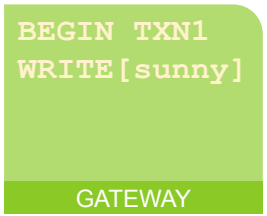
```
BEGIN;
INSERT INTO dogs VALUES (sunny);
INSERT INTO dogs VALUES (ozzie);
COMMIT;
```



t = 0

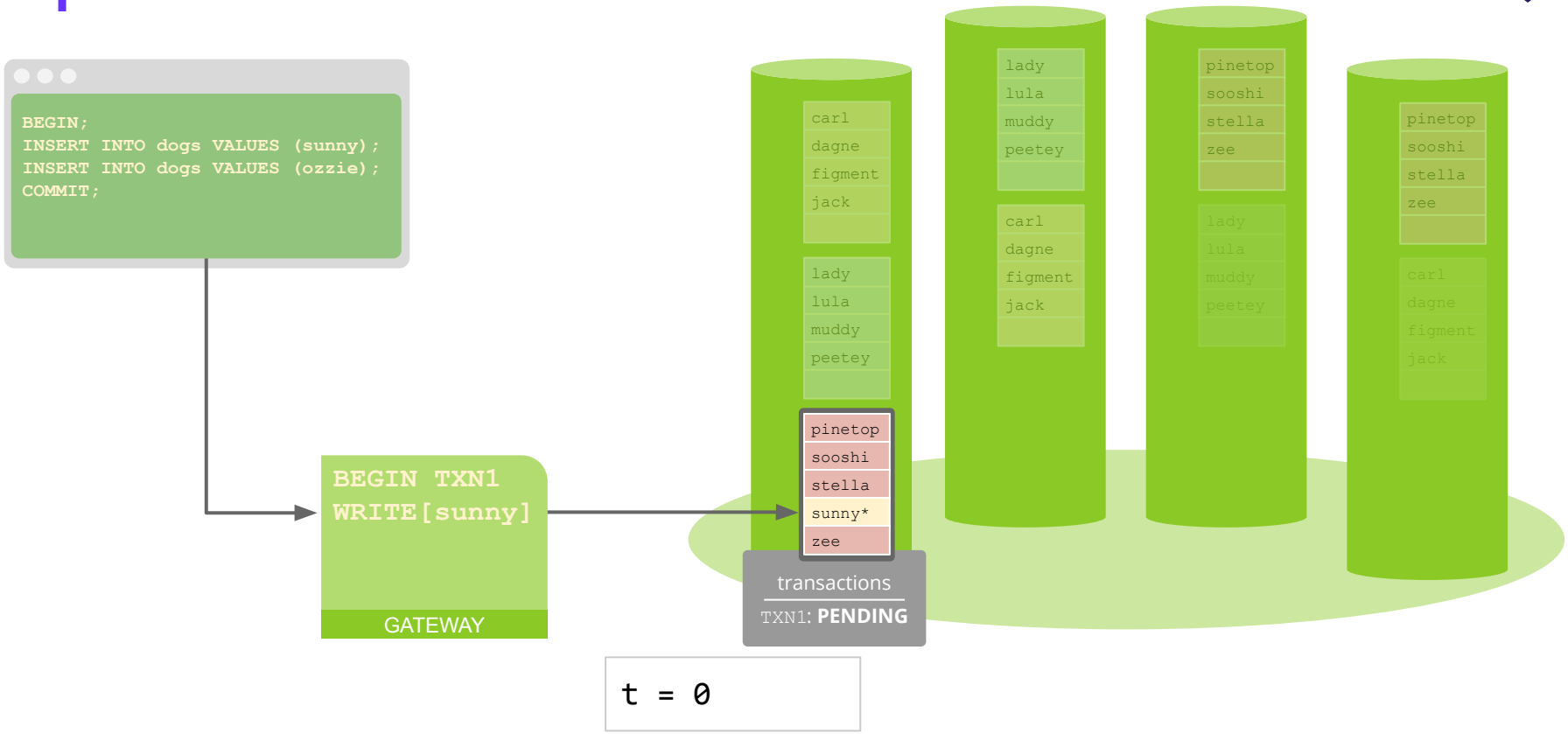
Pipelined Transactions



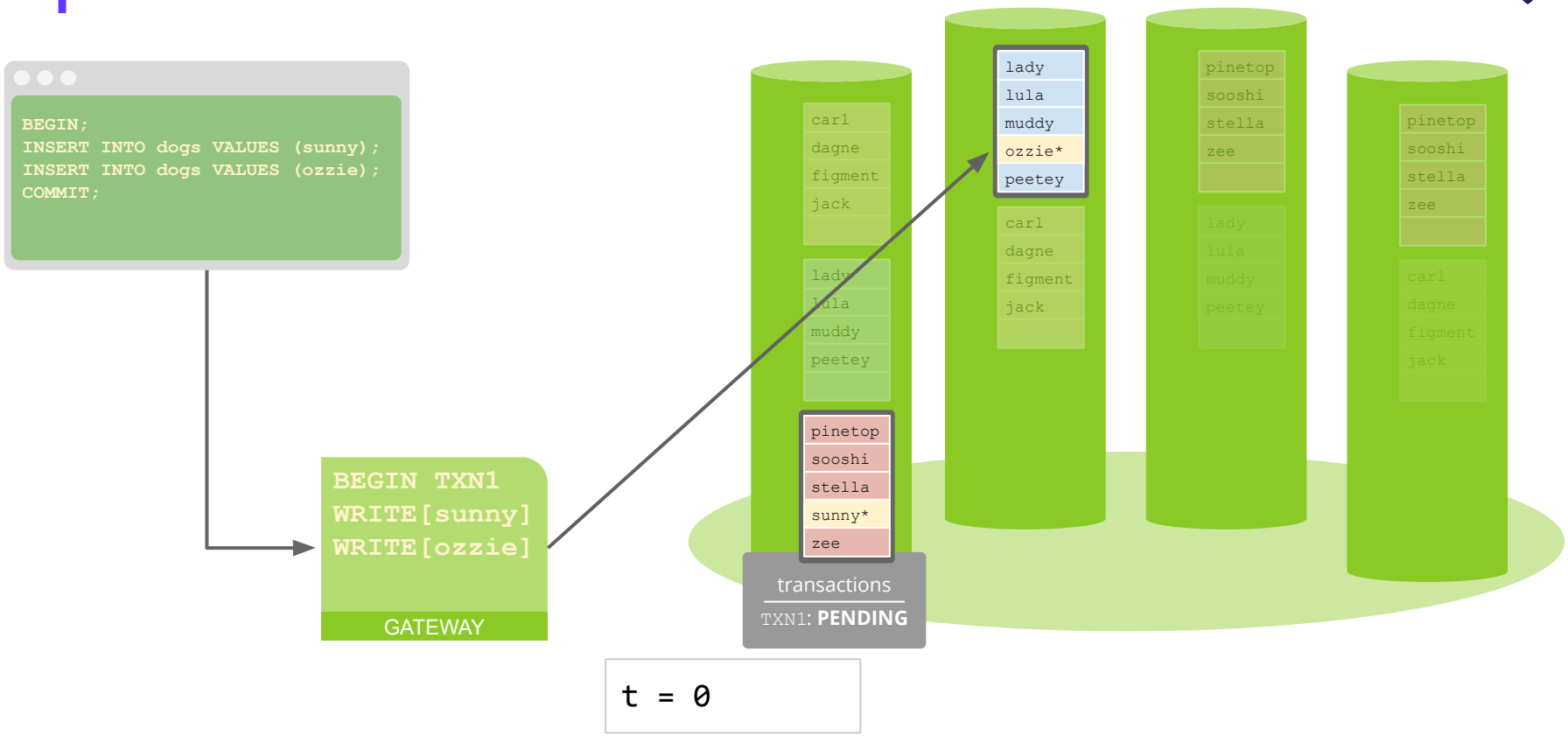


t = 0

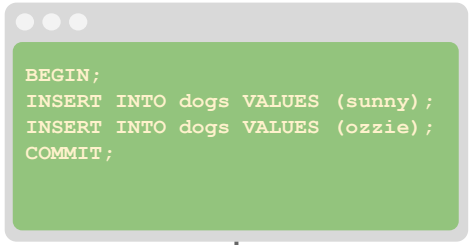
Pipelined Transactions

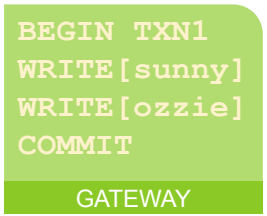


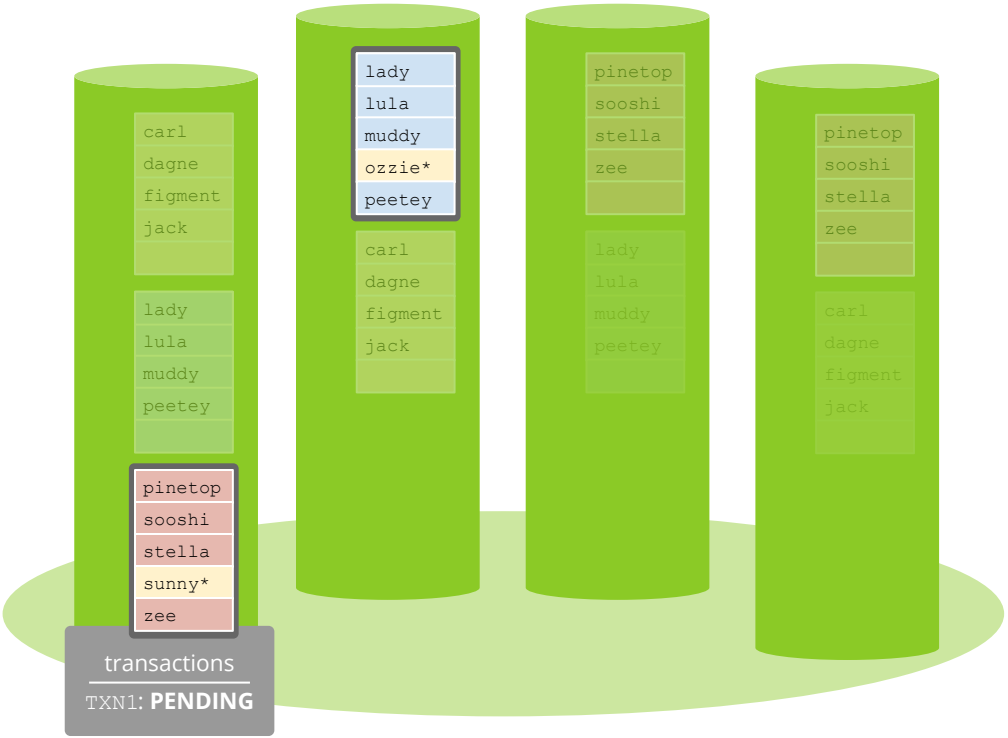
Pipelined Transactions



Pipelined Transactions

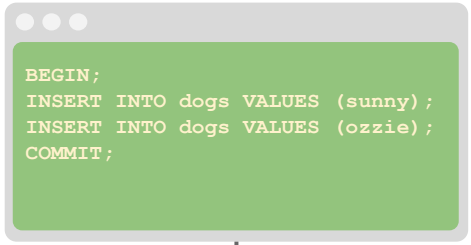


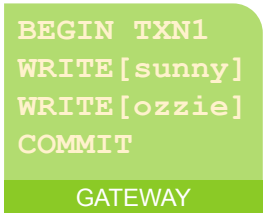


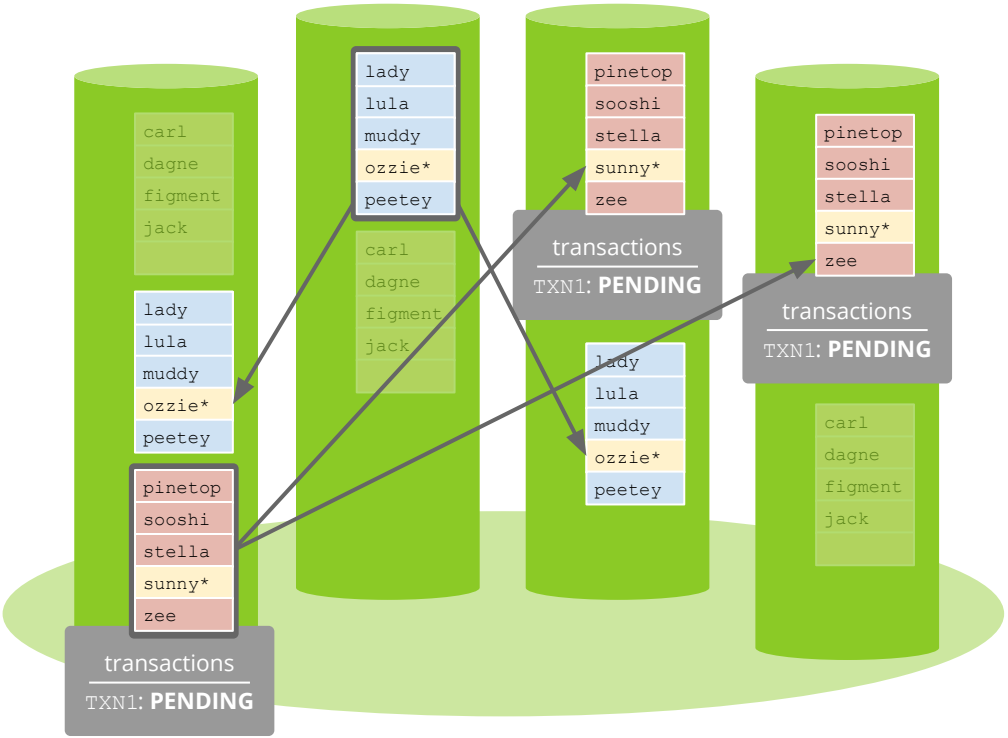


t = 0

Pipelined Transactions

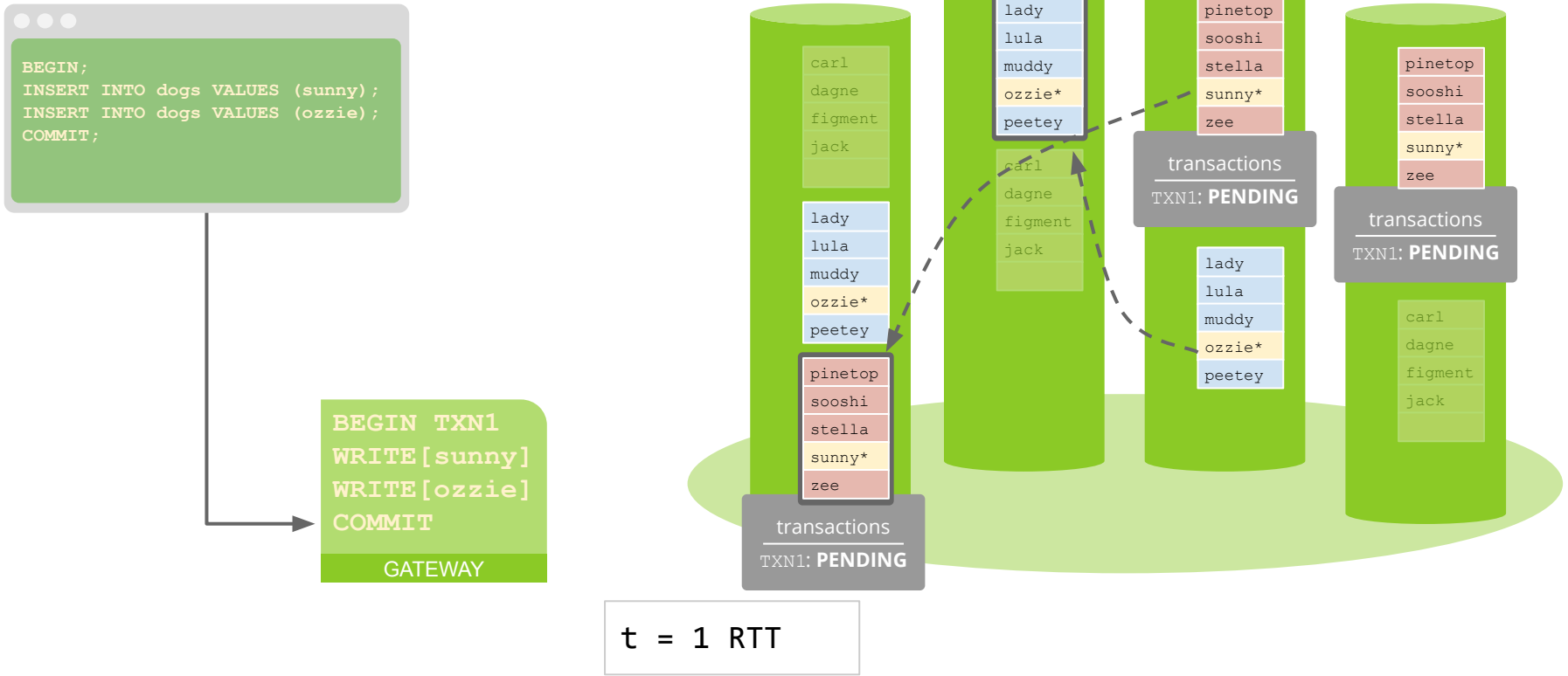




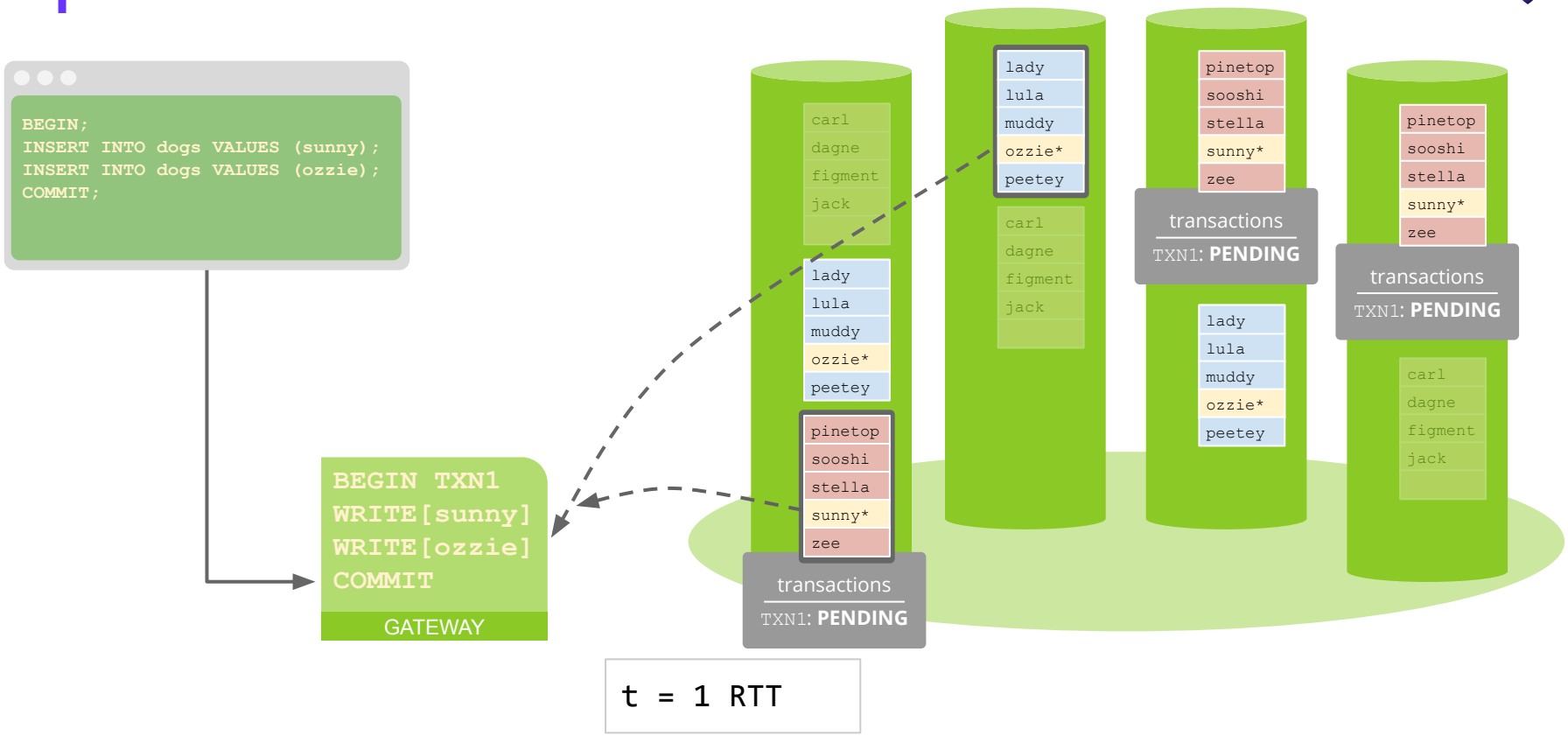


t = ½ RTT

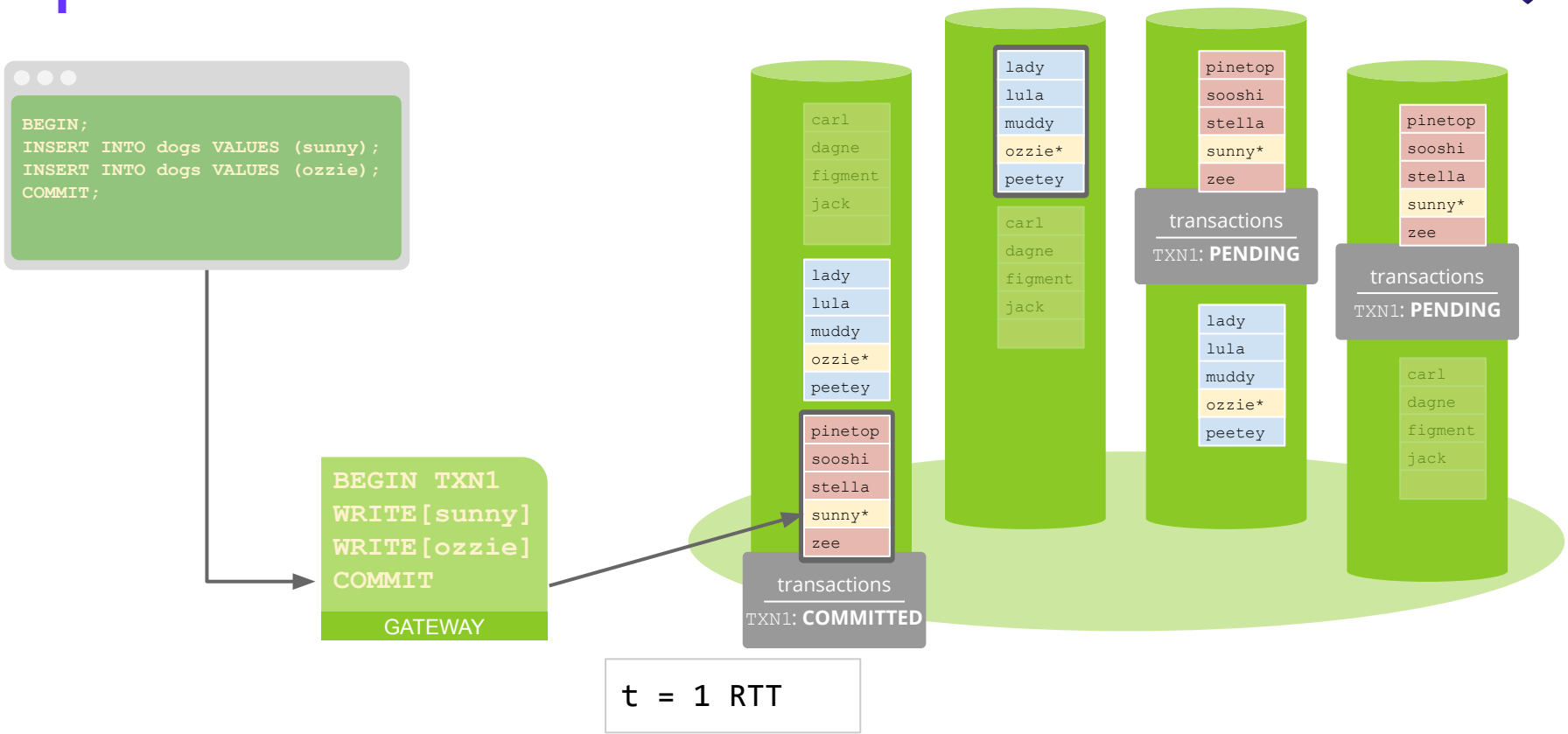
Pipelined Transactions



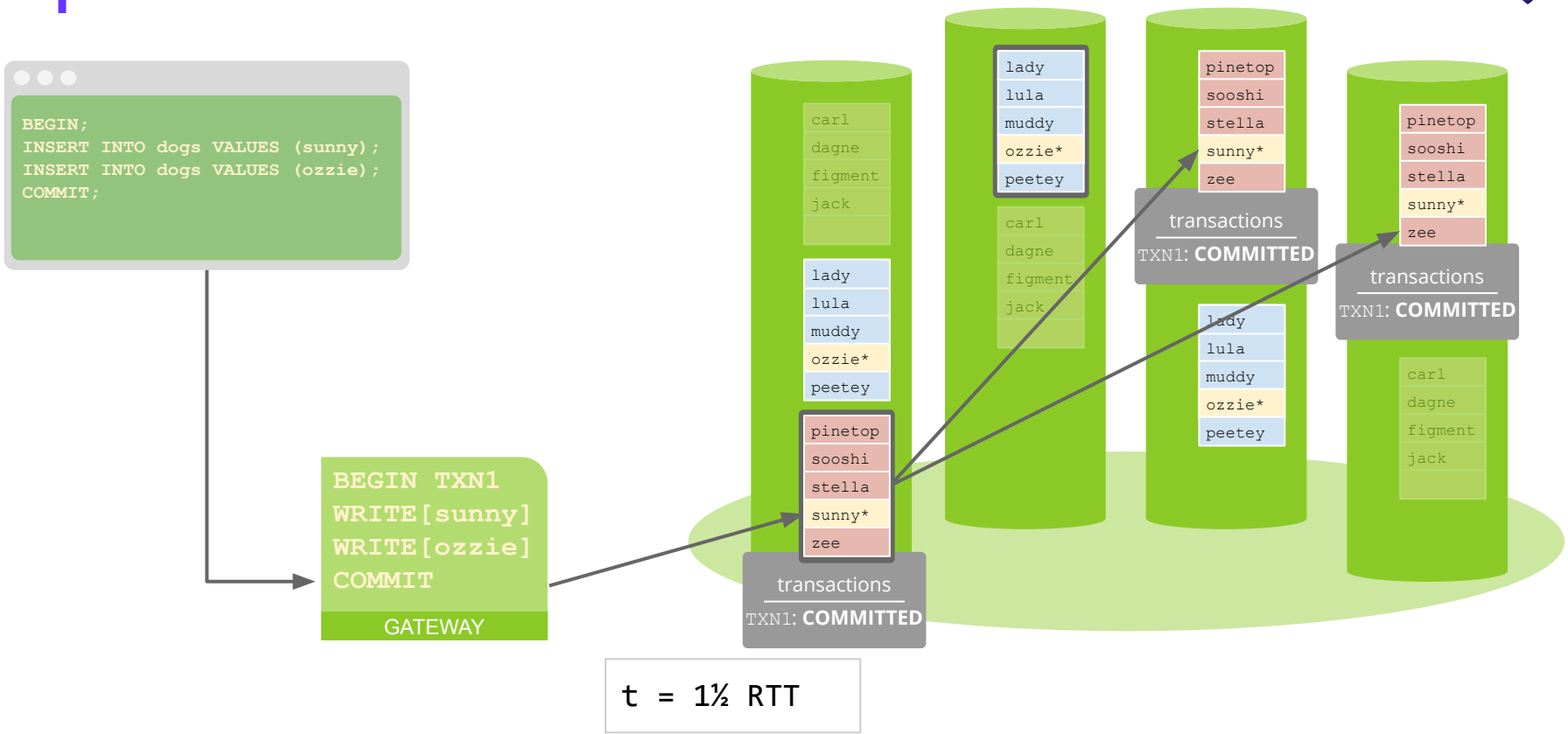
Pipelined Transactions



Pipelined Transactions

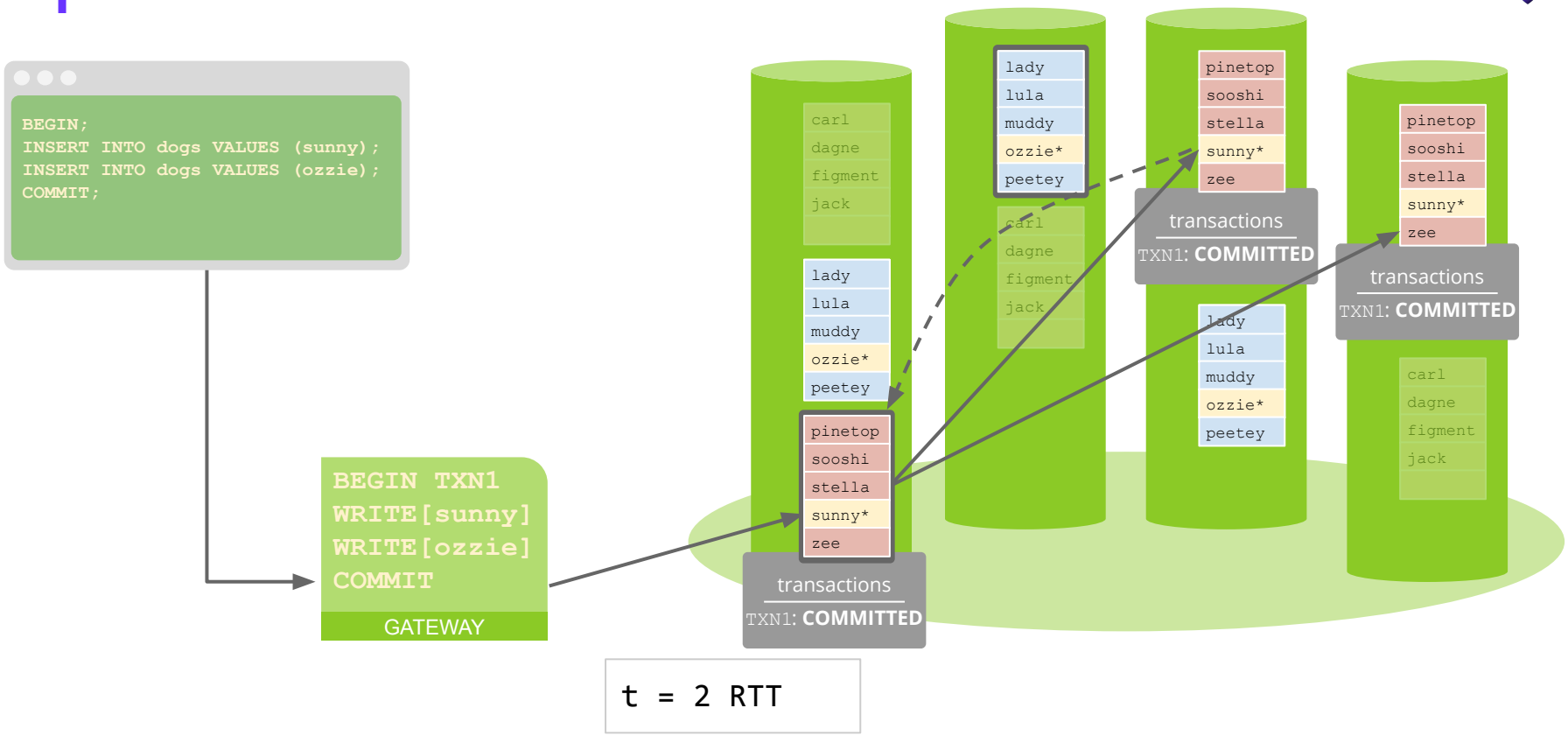


Pipelined Transactions

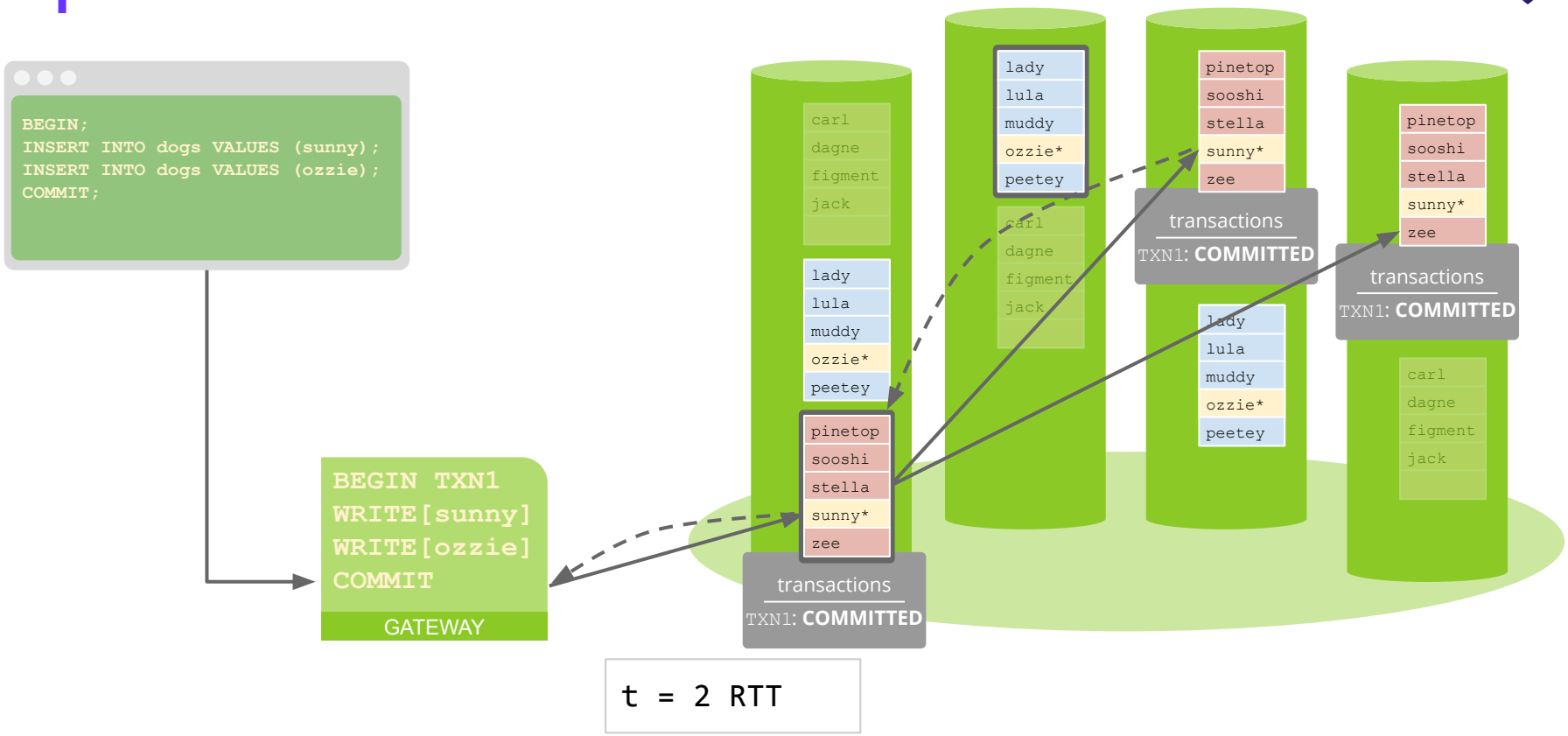


t = 1½ RTT

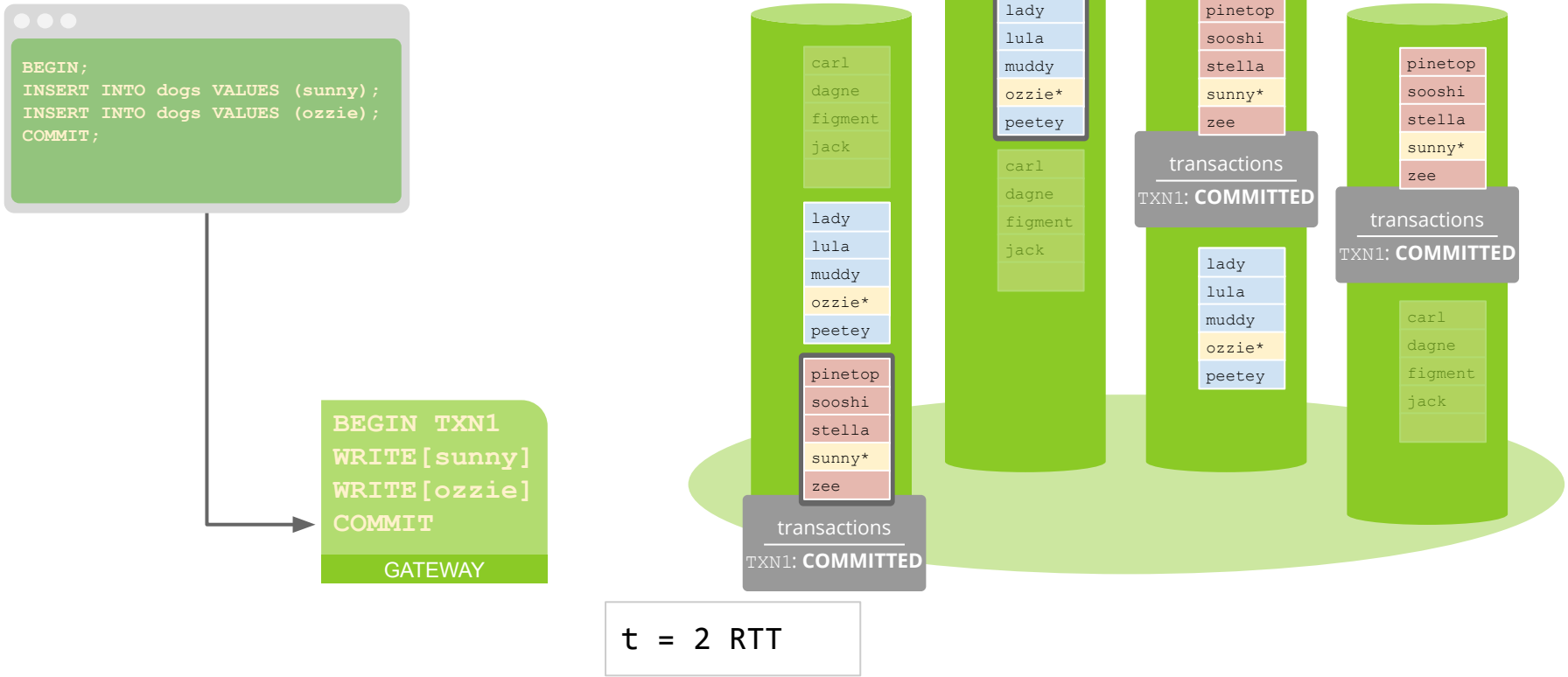
Pipelined Transactions



Pipelined Transactions

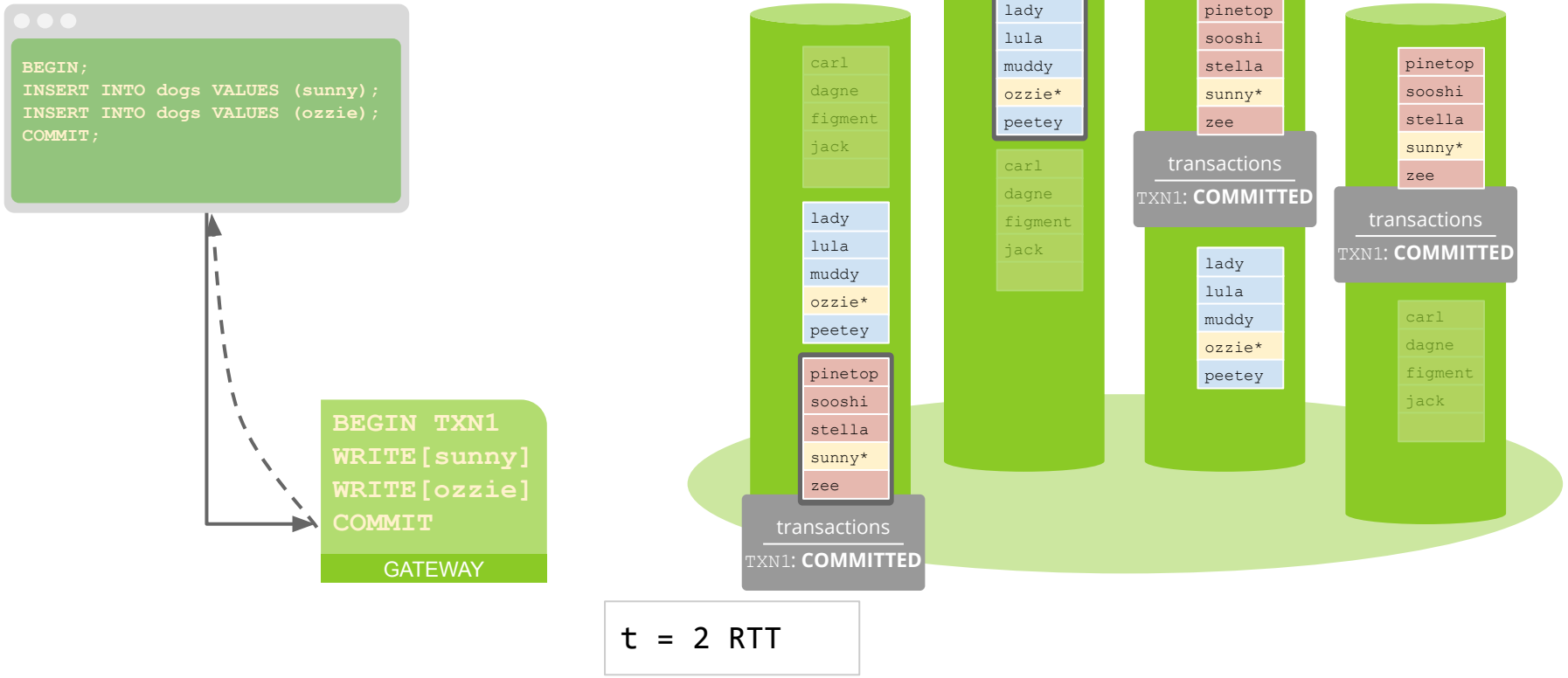


Pipelined Transactions



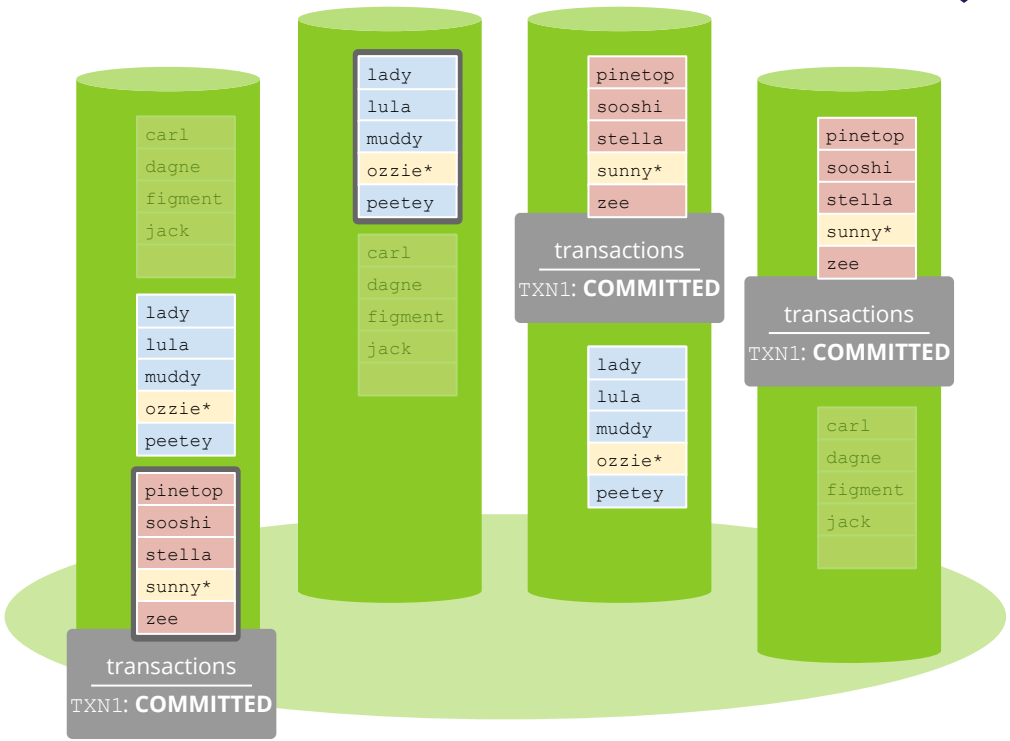
t = 2 RTT

Pipelined Transactions



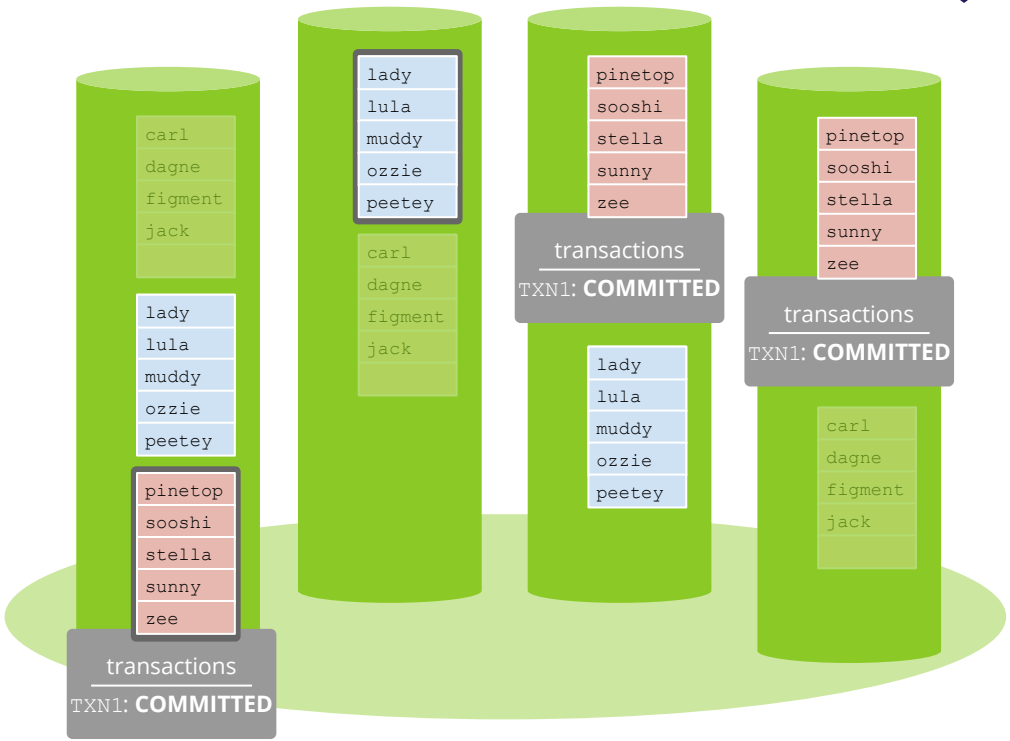
t = 2 RTT

Pipelined Transactions



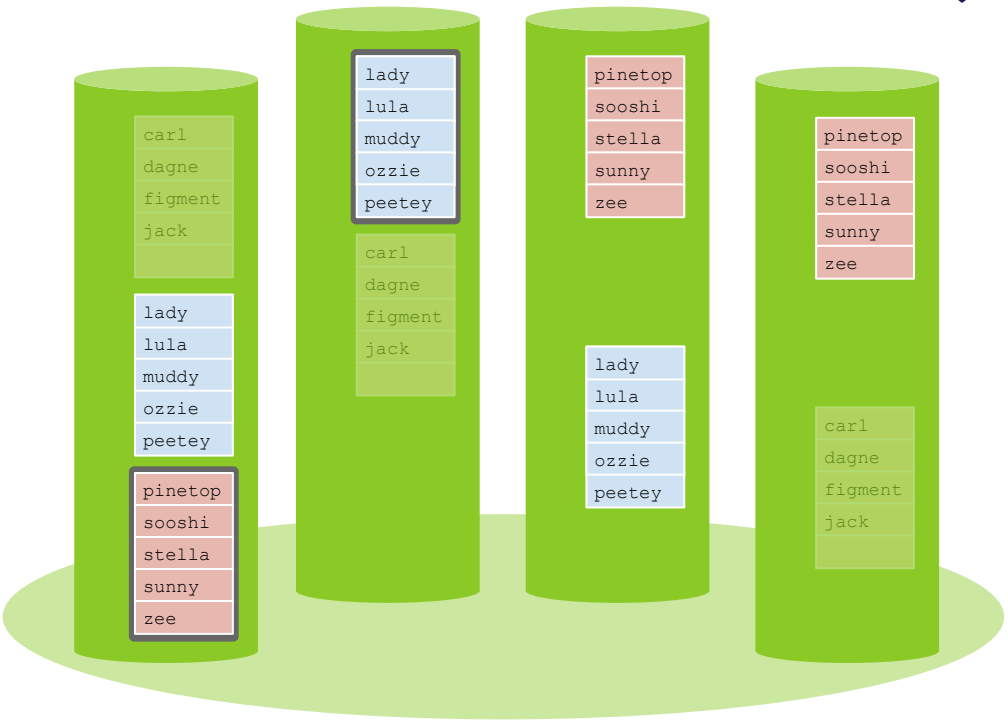
t = 2 RTT

Pipelined Transactions



t = 2 RTT

Pipelined Transactions




```
t = 2 RTT
```

Agenda

1. Foundations
2. Transactions
3. **Implementations**

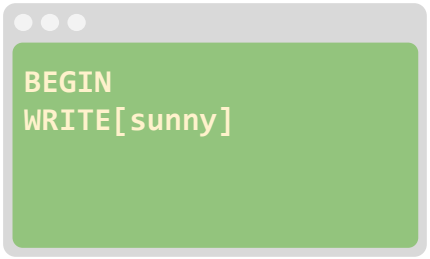
I. Spanner/Pipelined Transactions
II. Parallel Commits
III. Replicated Commit
IV. Carousel
V. MDCC
VI. SLOG/OceanVista
VII. TAPIR

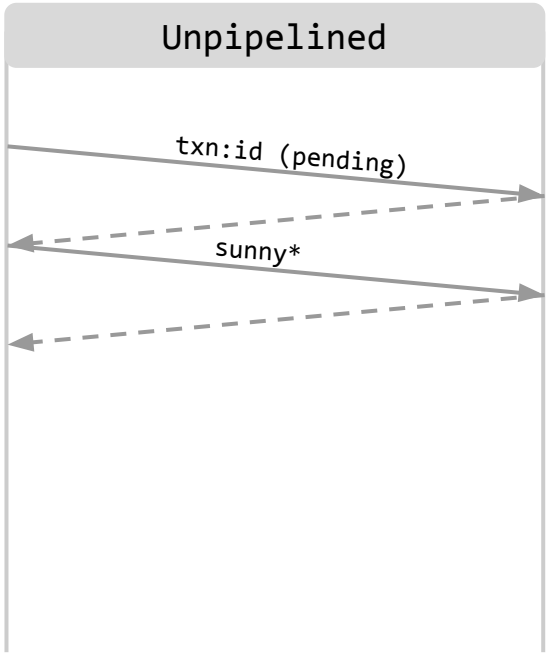
Timeline: Parallel Commits

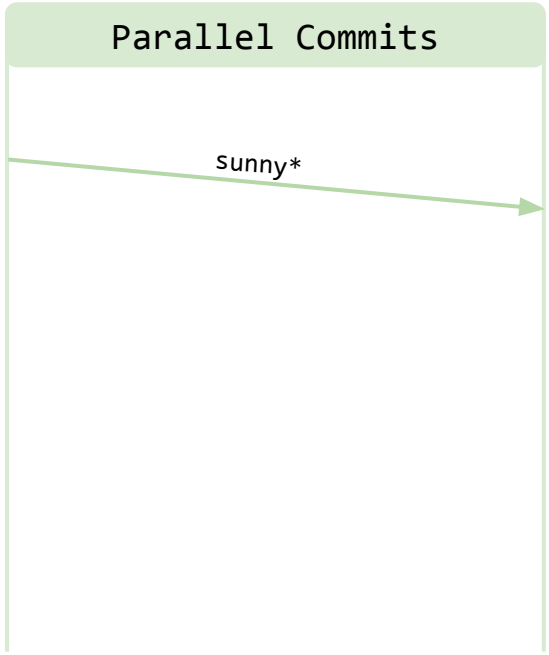
Unpipelined

Parallel Commits

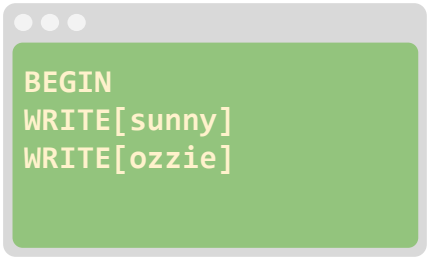
Timeline: Parallel Commits

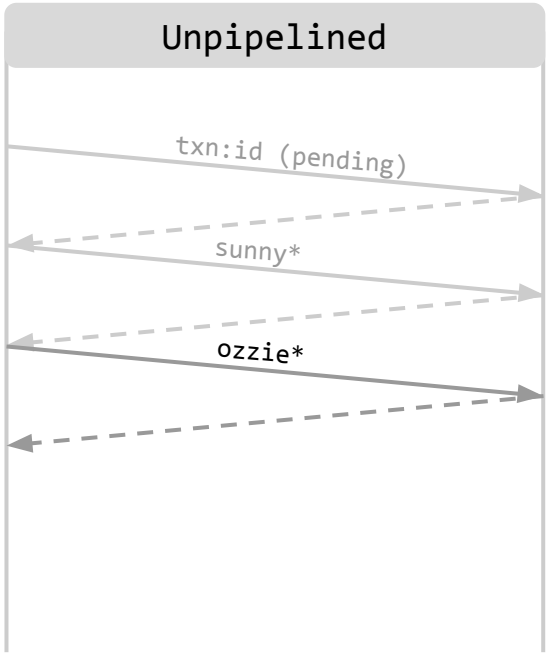


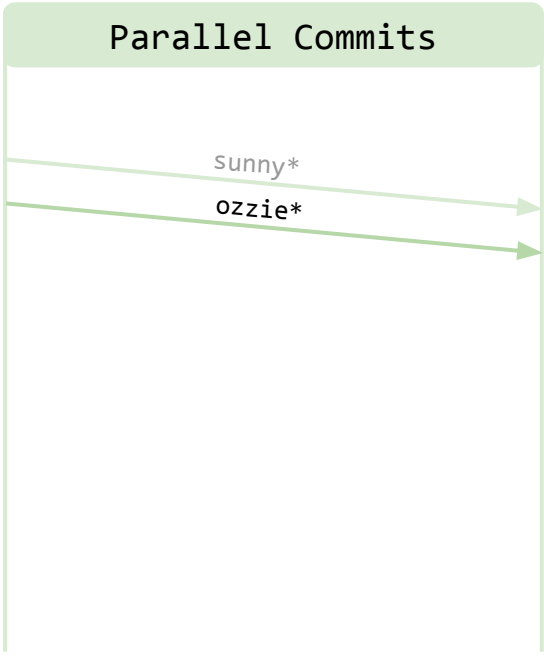




Timeline: Parallel Commits

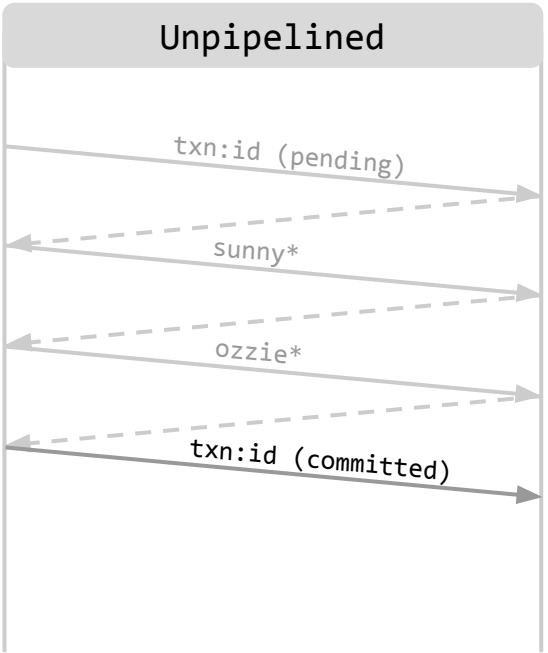


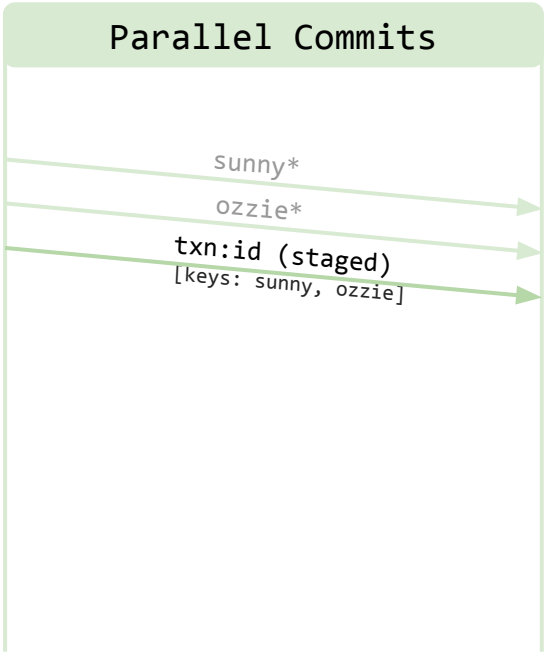




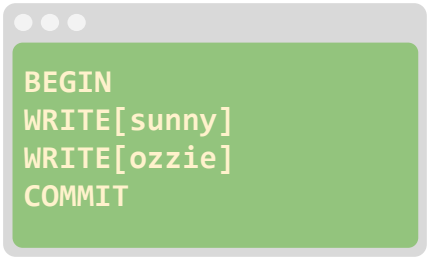
Timeline: Parallel Commits

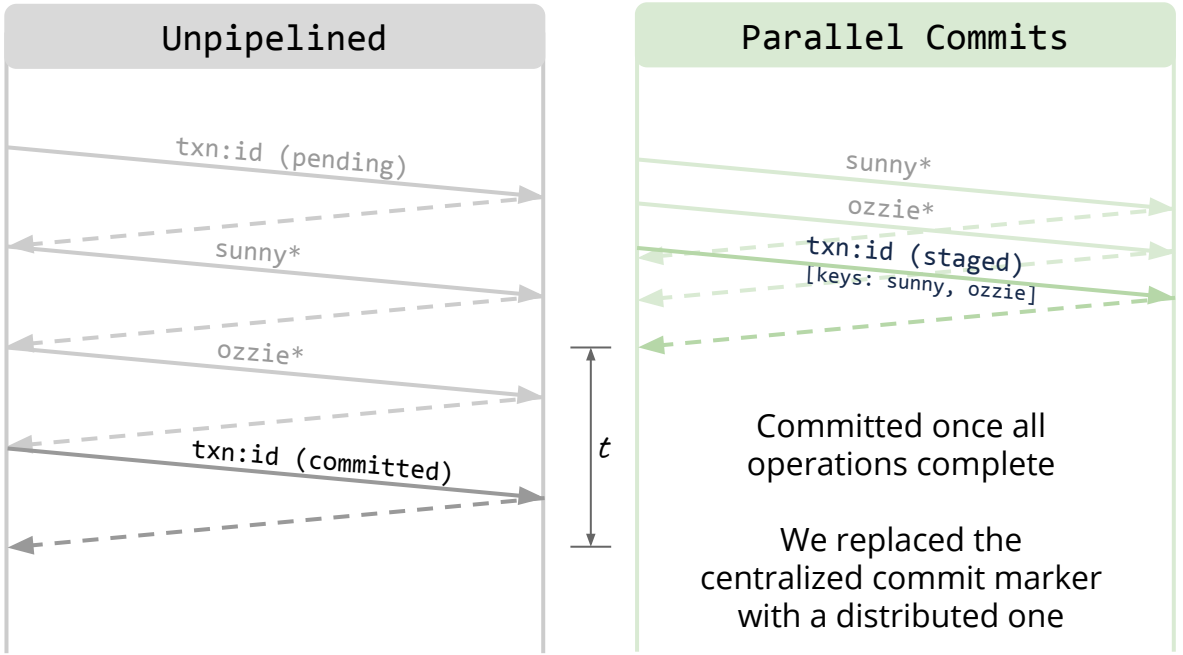
```
BEGIN
WRITE[sunny]
WRITE[ozzie]
COMMIT
```



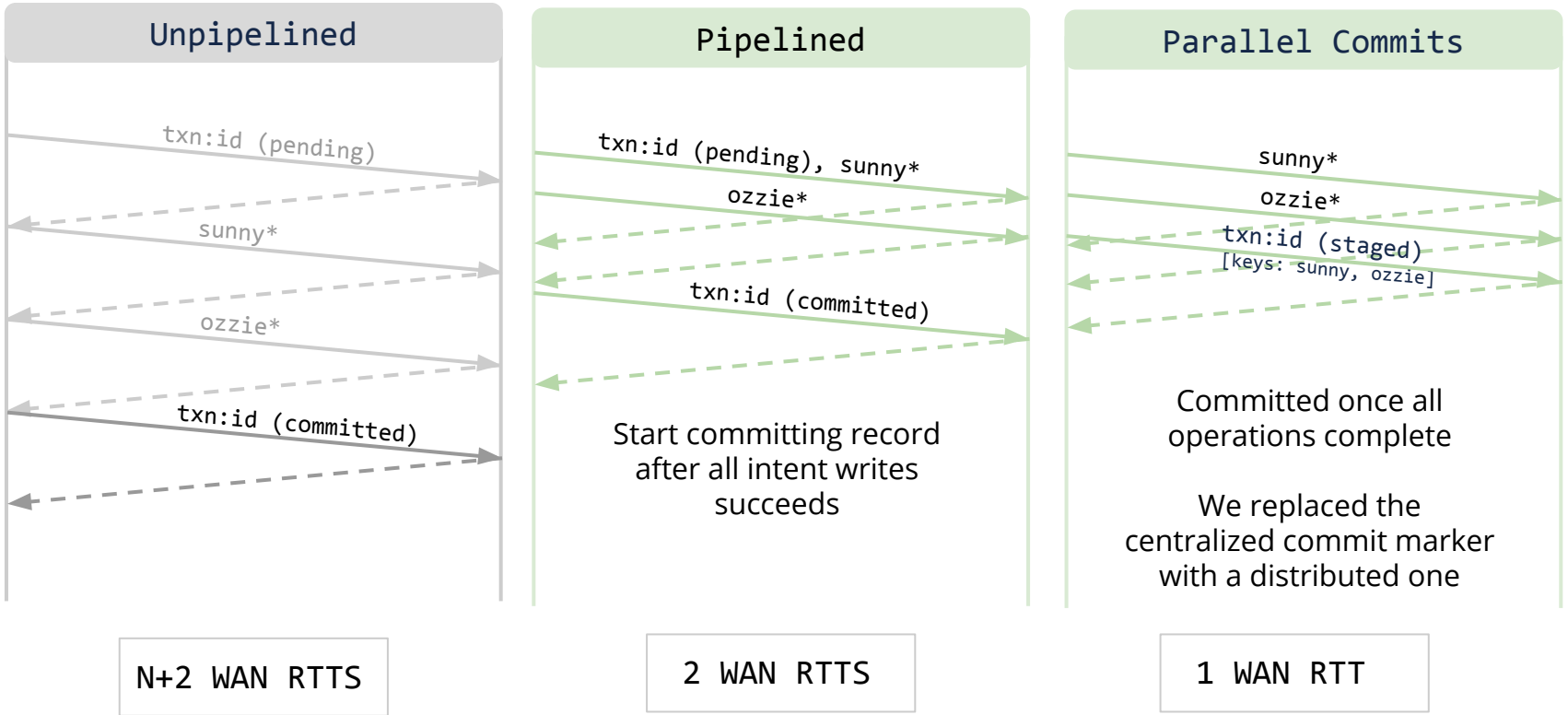


Timeline: Parallel Commits



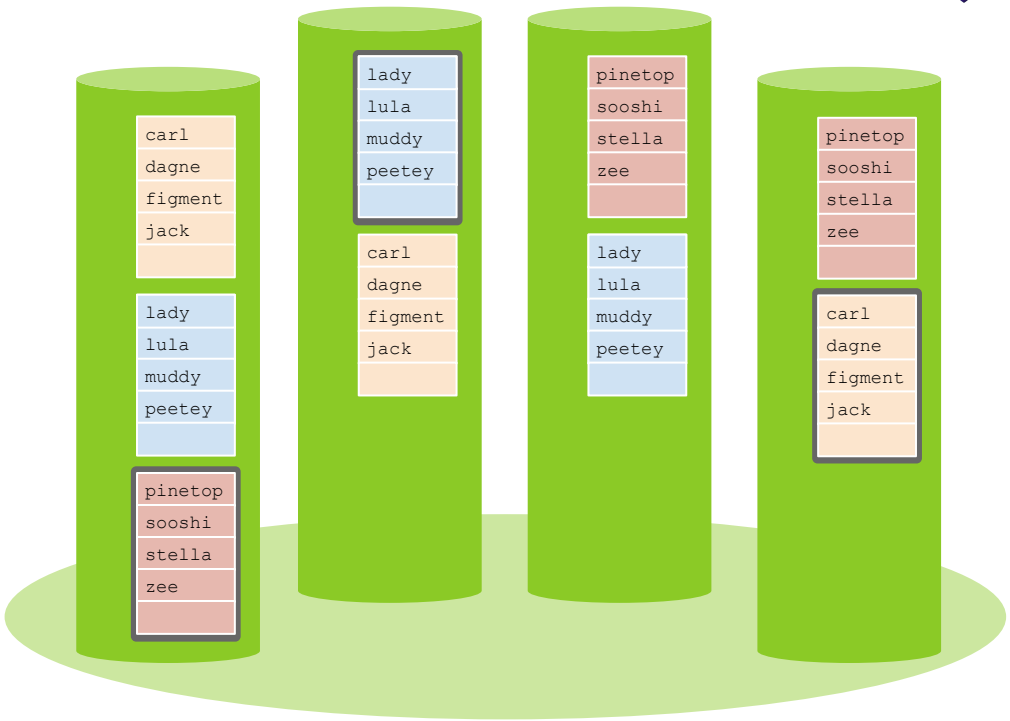


- Proved using TLA+


Unpipelined
txn:id (pending)
sunny*
ozzie*
txn:id (committed)
Pipelined
txn:id (pending), sunny*
ozzie*
txn:id (committed)
Start committing record after all intent writes succeeds
Parallel Commits
sunny*
ozzie*
txn:id (staged)
[keys: sunny, ozzie]
Committed once all operations complete
We replaced the centralized commit marker with a distributed one
N+2 WAN RTTS
2 WAN RTTS
1 WAN RTT

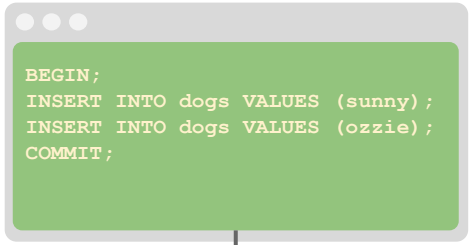
Parallel Commits

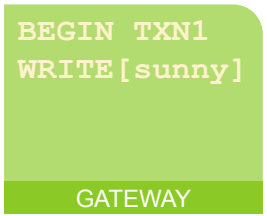
```
BEGIN;
INSERT INTO dogs VALUES (sunny);
INSERT INTO dogs VALUES (ozzie);
COMMIT;
```



t = 0

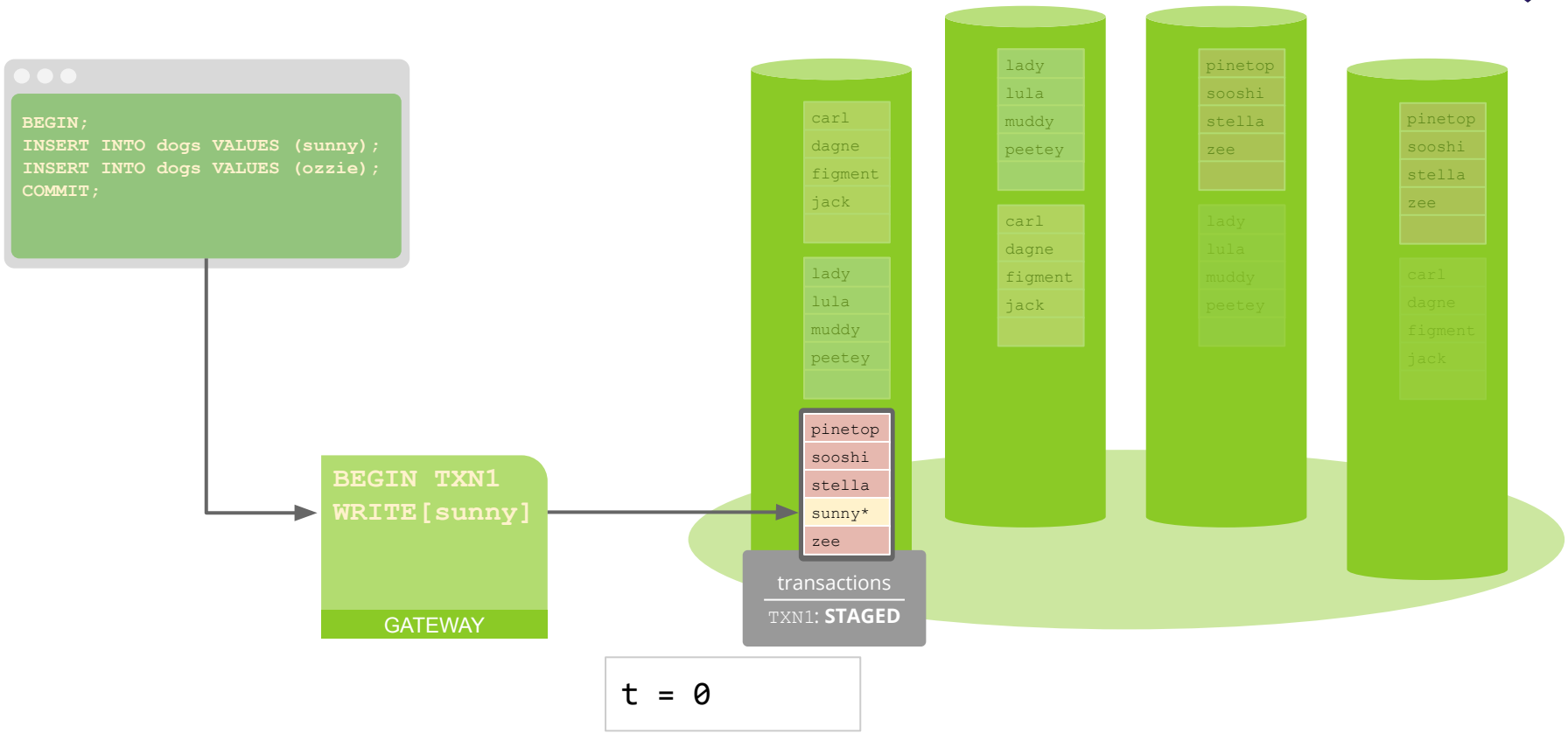
Parallel Commits



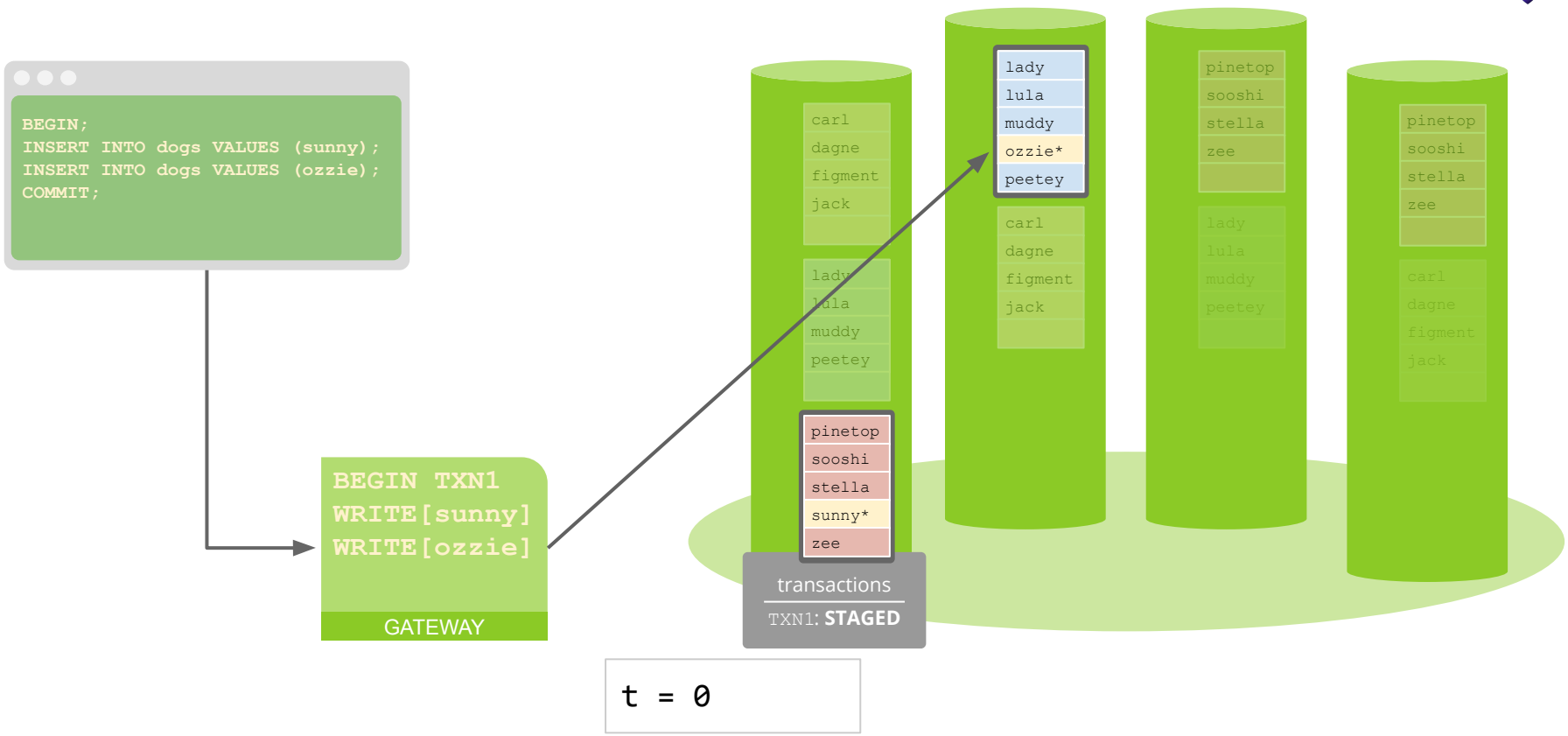


t = 0

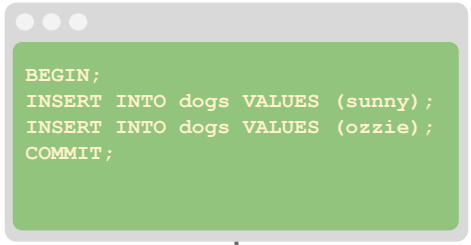
Parallel Commits

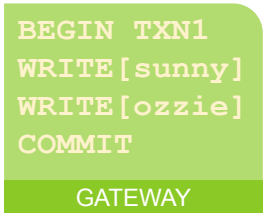


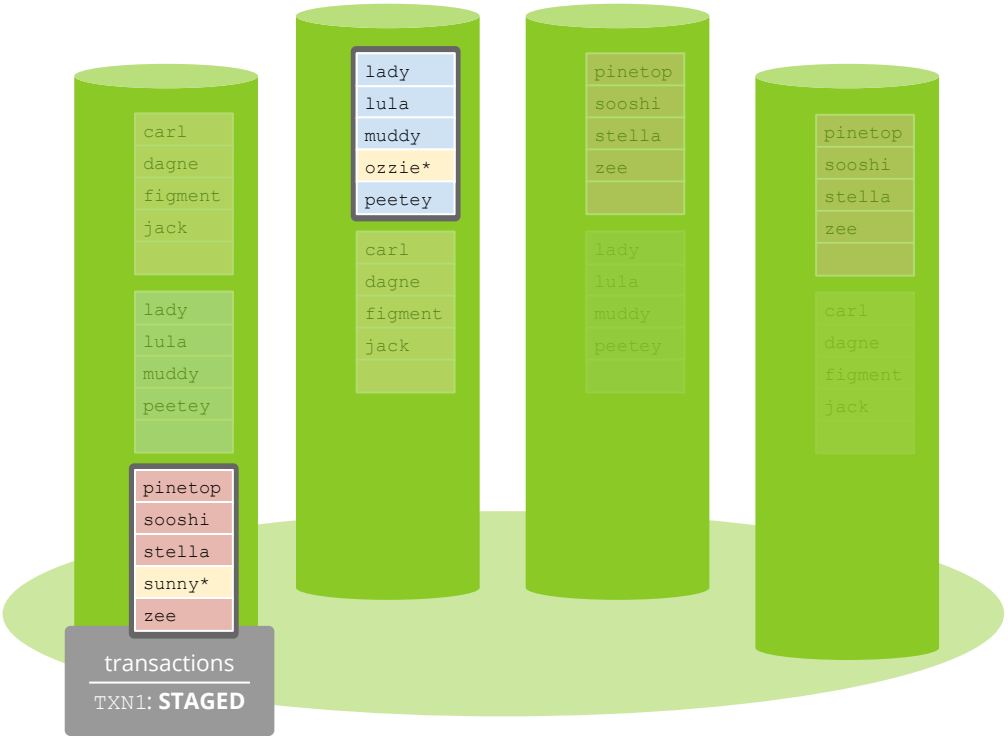
Parallel Commits



Parallel Commits

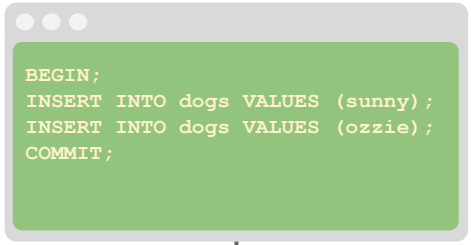


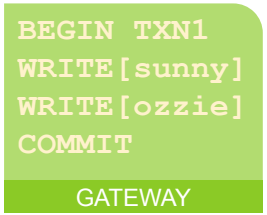


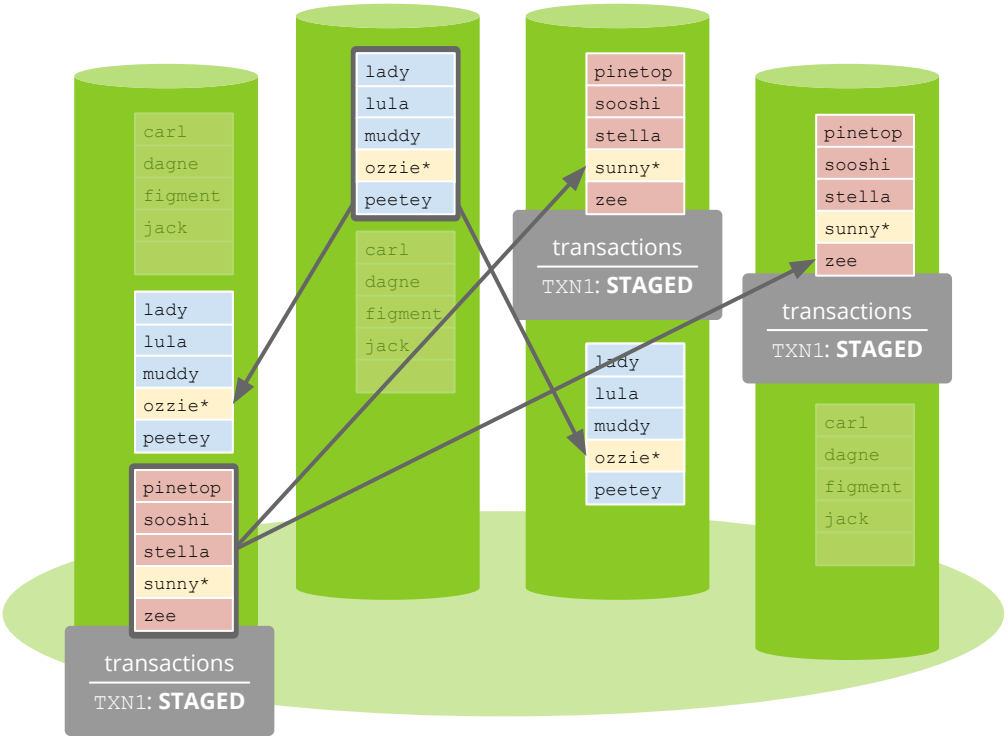


t = 0

Parallel Commits

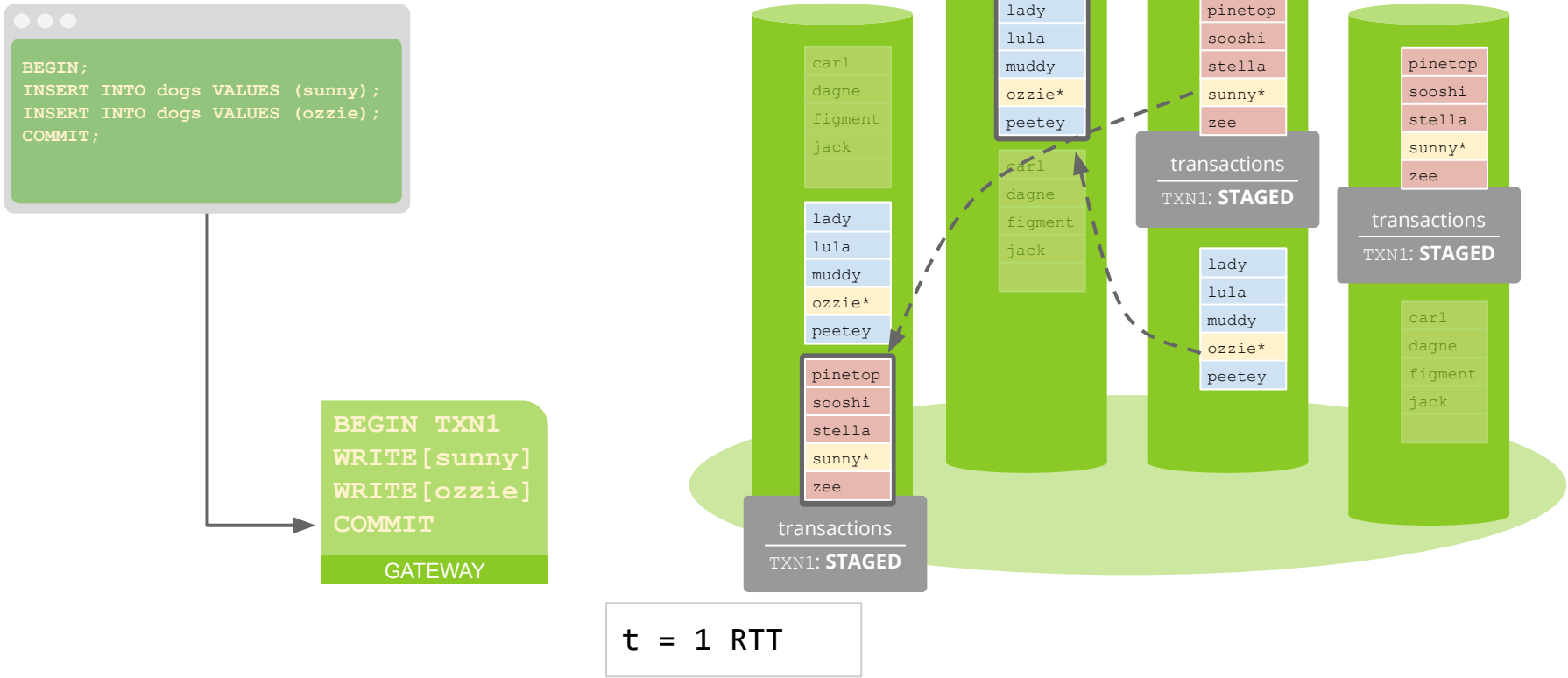






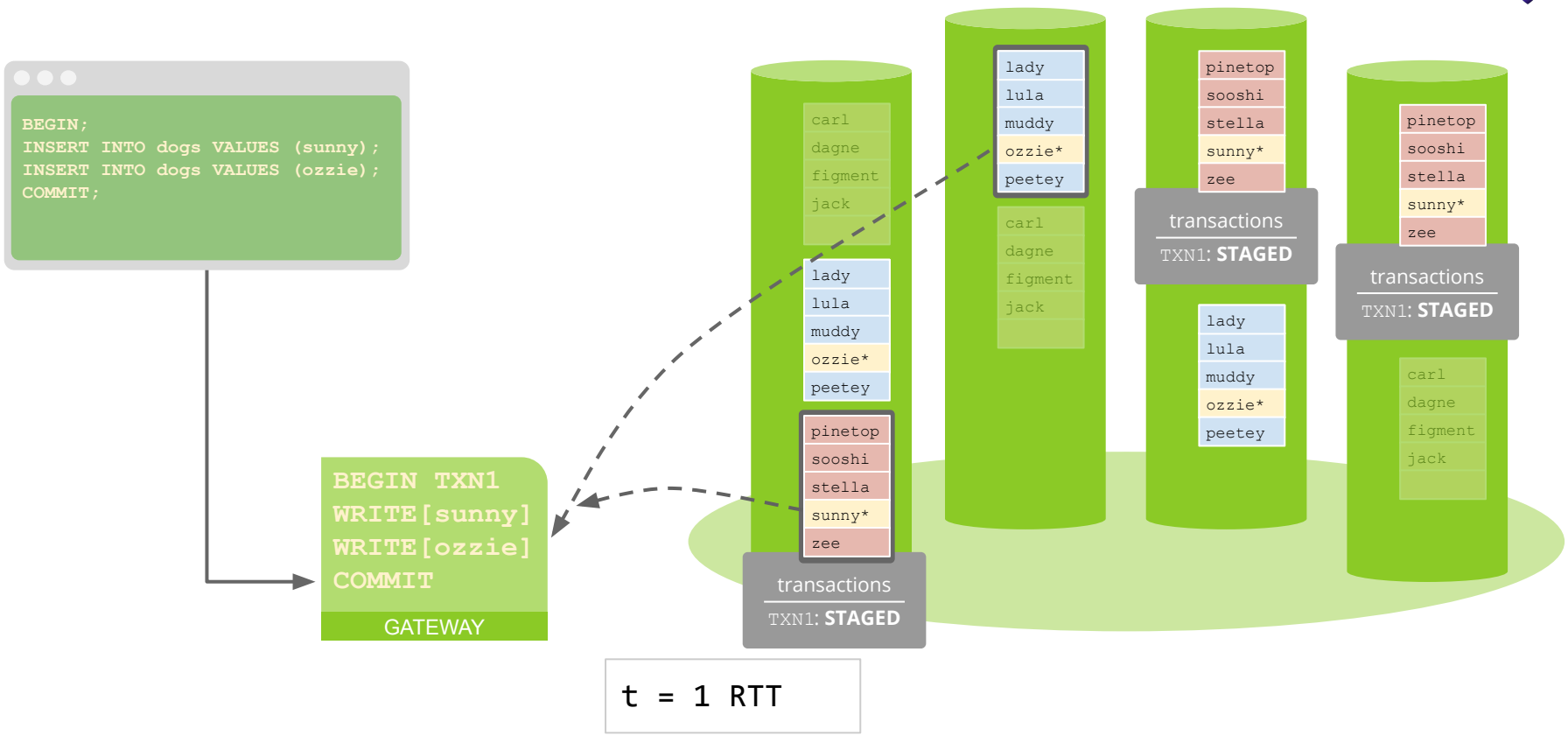
t = ½ RTT

Parallel Commits

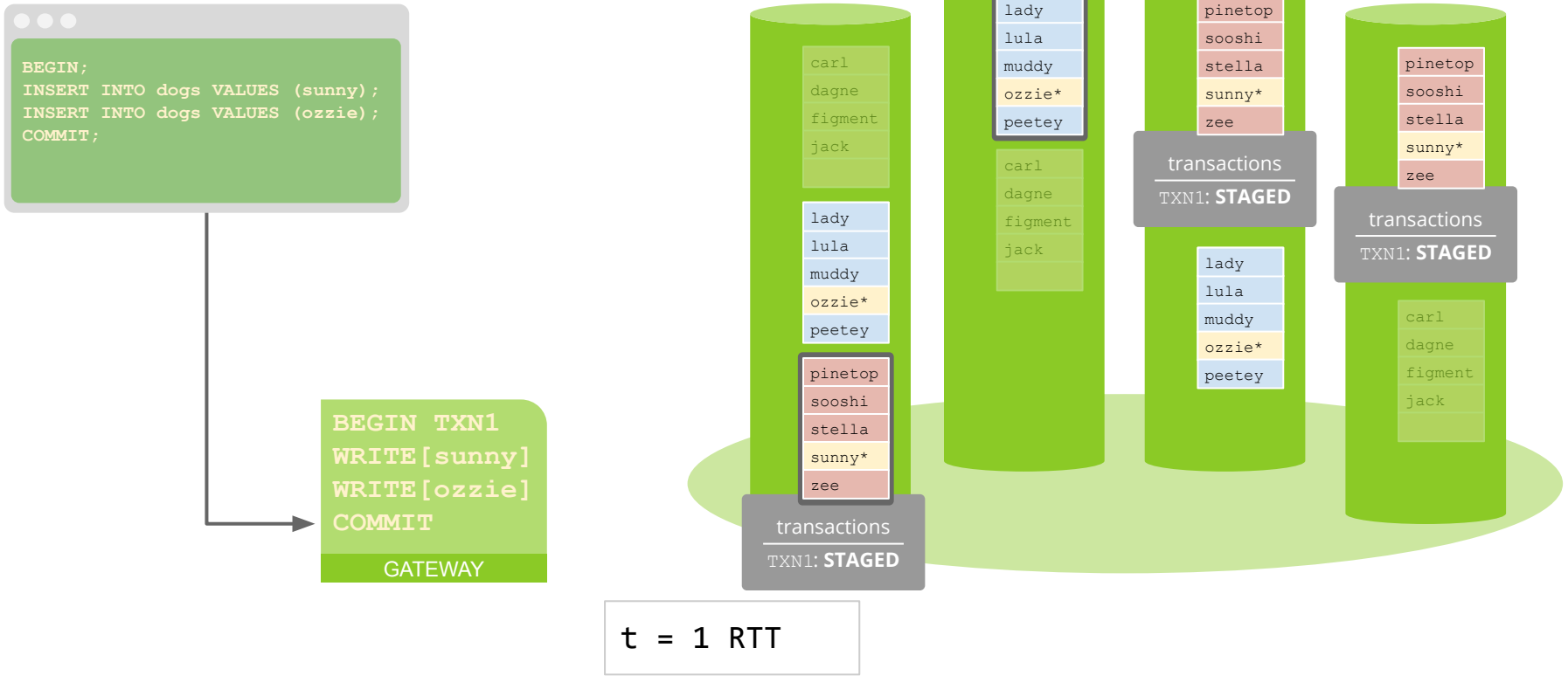


t = 1 RTT

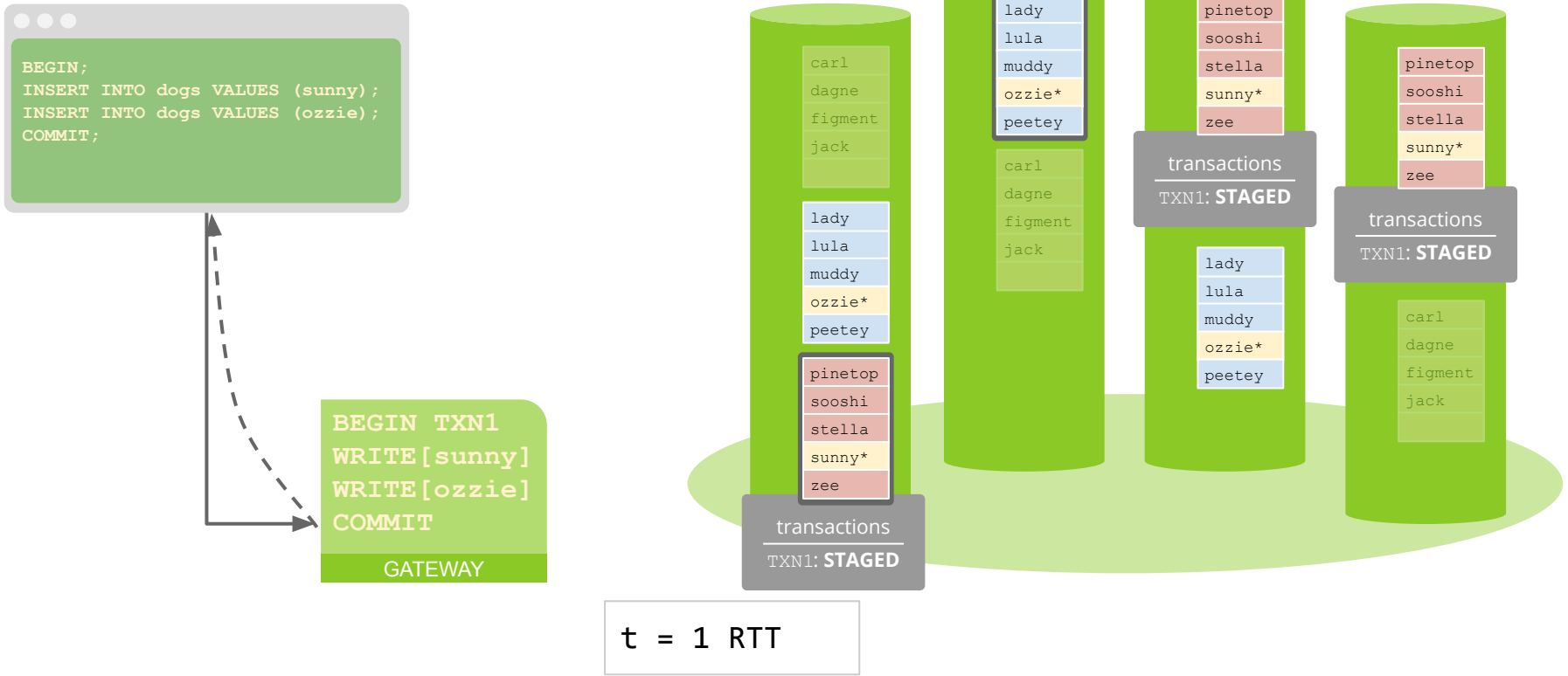
Parallel Commits



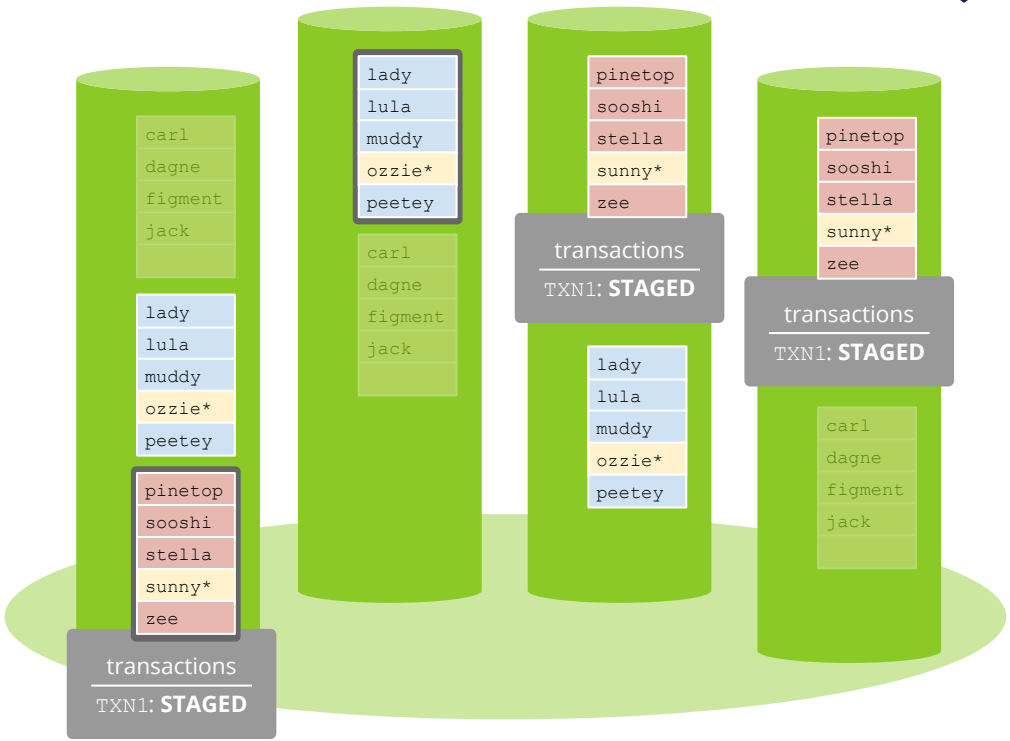
Parallel Commits



Parallel Commits

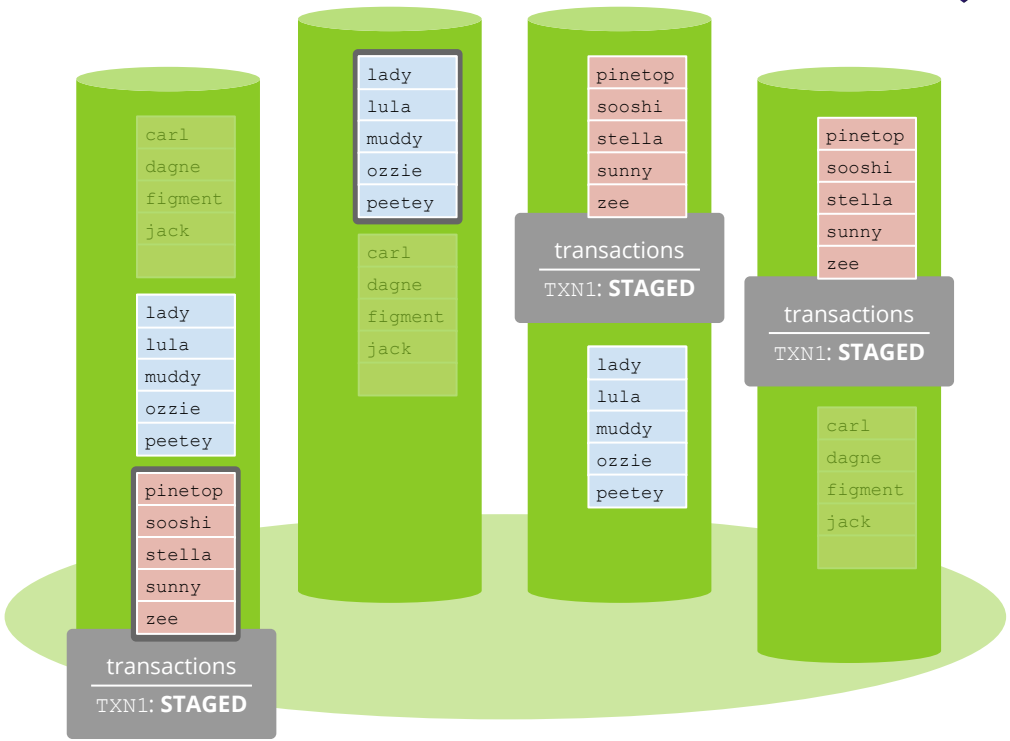


Parallel Commits



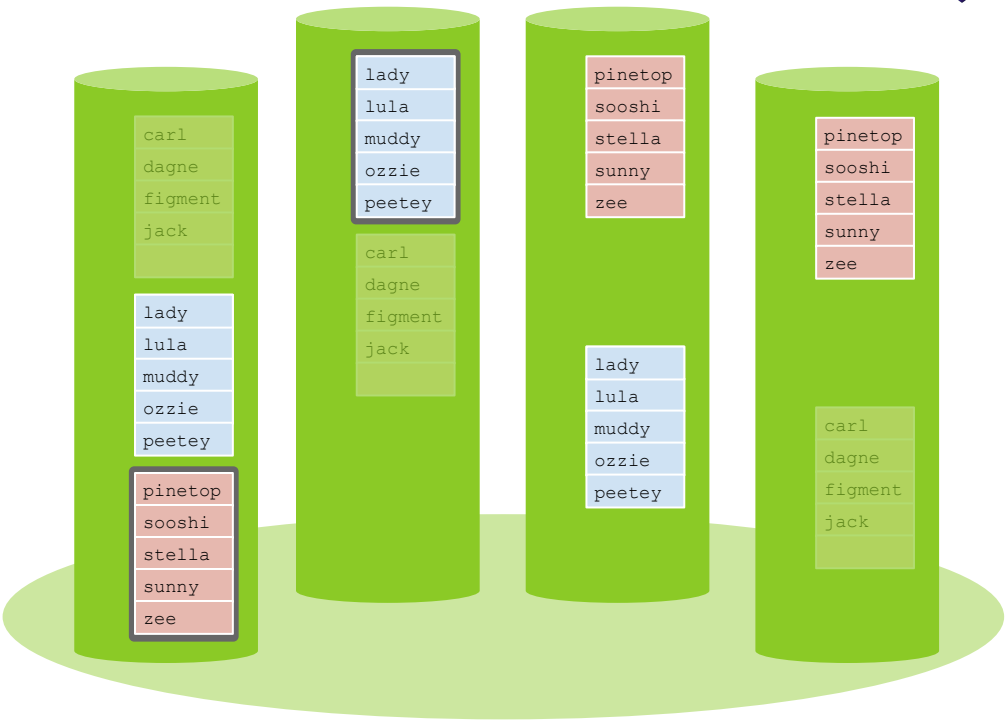

```
t = 1 RTT
```

Parallel Commits



t = 1 RTT

Parallel Commits




```
t = 1 RTT
```

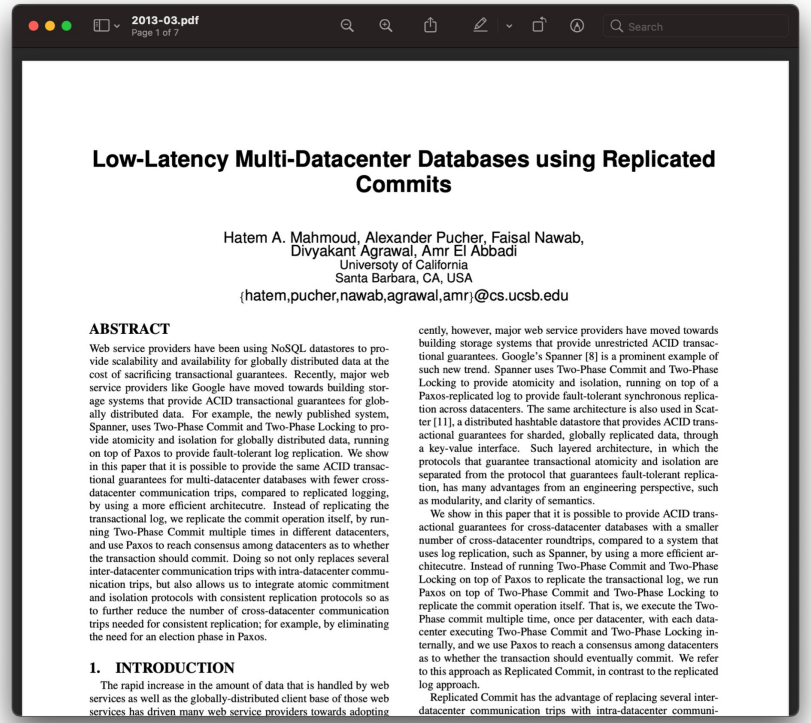
Agenda

1. Foundations
2. Transactions
3. **Implementations**

I. Spanner/Pipelined Transactions
II. Parallel Commits
III. Replicated Commit
IV. Carousel
V. MDCC
VI. SLOG/OceanVista
VII. TAPIR

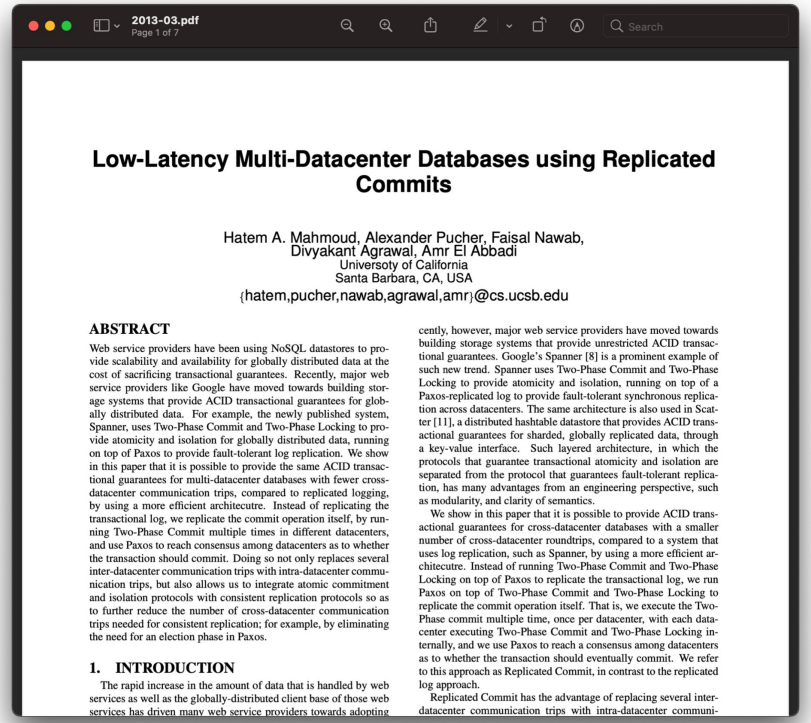
Replicated Commit

From UC Santa Barbara, 2013. Distinguishes between intra-DC (10ms) and inter-DC (100ms) latencies. Each DC contains replicas of all ranges.



Replicated Commit

Suggests running 2PC multiple times in parallel in each DC, and using Paxos across DCs to determine if txn should commit.



Low-Latency Multi-Datacenter Databases using Replicated Commits

Hatem A. Mahmoud, Alexander Pucher, Faisal Nawab,
Divyakant Agrawal, Amr El Abbadi
Universoty of California
Santa Barbara, CA, USA
{hatem,pucher,nawab,agrawal,amr}@cs.ucsb.edu

ABSTRACT

Web service providers have been using NoSQL datastores to provide scalability and availability for globally distributed data at the cost of sacrificing transactional guarantees. Recently, major web service providers like Google have moved towards building storage systems that provide ACID transactional guarantees for globally distributed data. For example, the newly published system, Spanner, uses Two-Phase Commit and Two-Phase Locking to provide atomicity and isolation for globally distributed data, running on top of Paxos to provide fault-tolerant log replication. We show in this paper that it is possible to provide the same ACID transactional guarantees for multi-datacenter databases with fewer cross-datacenter communication trips, compared to replicated logging, by using a more efficient architecutre. Instead of replicating the transactional log, we replicate the commit operation itself, by running Two-Phase Commit multiple times in different datacenters, and use Paxos to reach consensus among datacenters as to whether the transaction should commit. Doing so not only replaces several inter-datacenter communication trips with intra-datacenter communication trips, but also allows us to integrate atomic commitment and isolation protocols with consistent replication protocols so as to further reduce the number of cross-datacenter communication trips needed for consistent replication; for example, by eliminating the need for an election phase in Paxos.

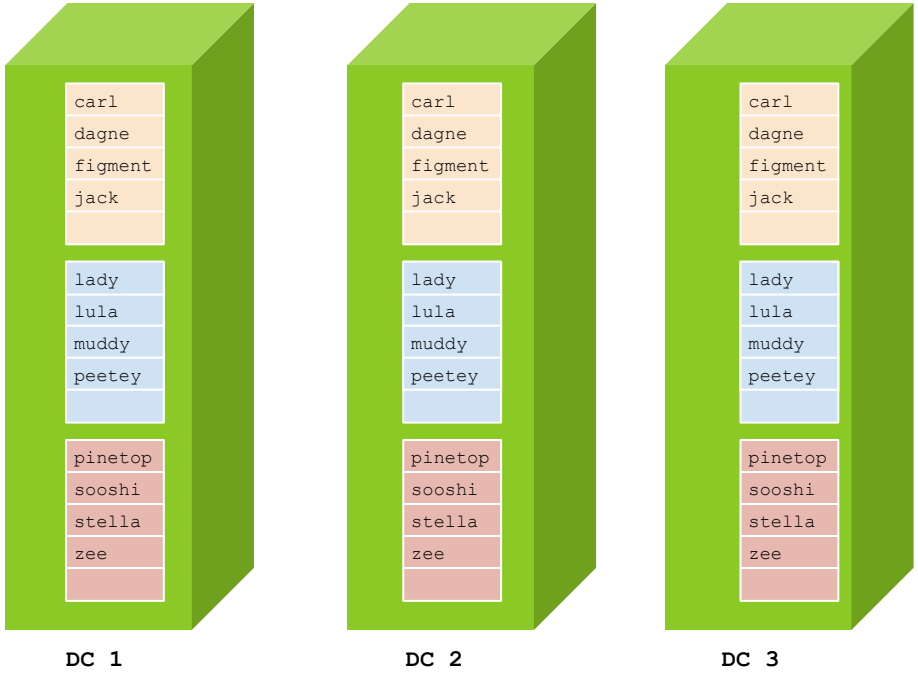
1. INTRODUCTION

The rapid increase in the amount of data that is handled by web services as well as the globally-distributed client base of those web services has driven many web service providers towards adopting cently, however, major web service providers have moved towards building storage systems that provide unrestricted ACID transactional guarantees. Google's Spanner [8] is a prominent example of such new trend. Spanner uses Two-Phase Commit and Two-Phase Locking to provide atomicity and isolation, running on top of a Paxos-replicated log to provide fault-tolerant synchronous replication across datacenters. The same architecture is also used in Scatter [11], a distributed hashtable datastore that provides ACID transactional guarantees for sharded, globally replicated data, through a key-value interface. Such layered architecture, in which the protocols that guarantee transactional atomicity and isolation are separated from the protocol that guarantees fault-tolerant replication, has many advantages from an engineering perspective, such as modularity, and clarity of semantics.

We show in this paper that it is possible to provide ACID transactional guarantees for cross-datacenter databases with a smaller number of cross-datacenter roundtrips, compared to a system that uses log replication, such as Spanner, by using a more efficient architecutre. Instead of running Two-Phase Commit and Two-Phase Locking on top of Paxos to replicate the transactional log, we run Paxos on top of Two-Phase Commit and Two-Phase Locking to replicate the commit operation itself. That is, we execute the Two-Phase commit multiple time, once per datacenter, with each datacenter executing Two-Phase Commit and Two-Phase Locking internally, and we use Paxos to reach a consensus among datacenters as to whether the transaction should eventually commit. We refer to this approach as Replicated Commit, in contrast to the replicated log approach.

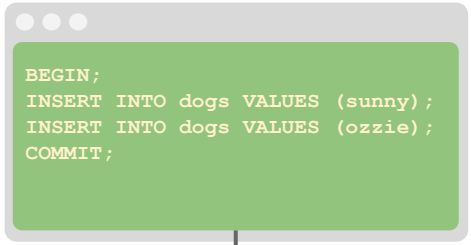
Replicated Commit has the advantage of replacing several inter-datacenter communication trips with intra-datacenter communi-

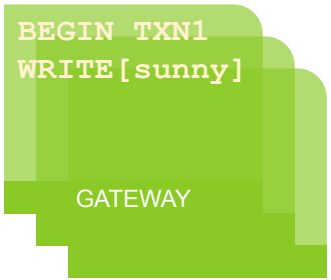
Replicated Commit

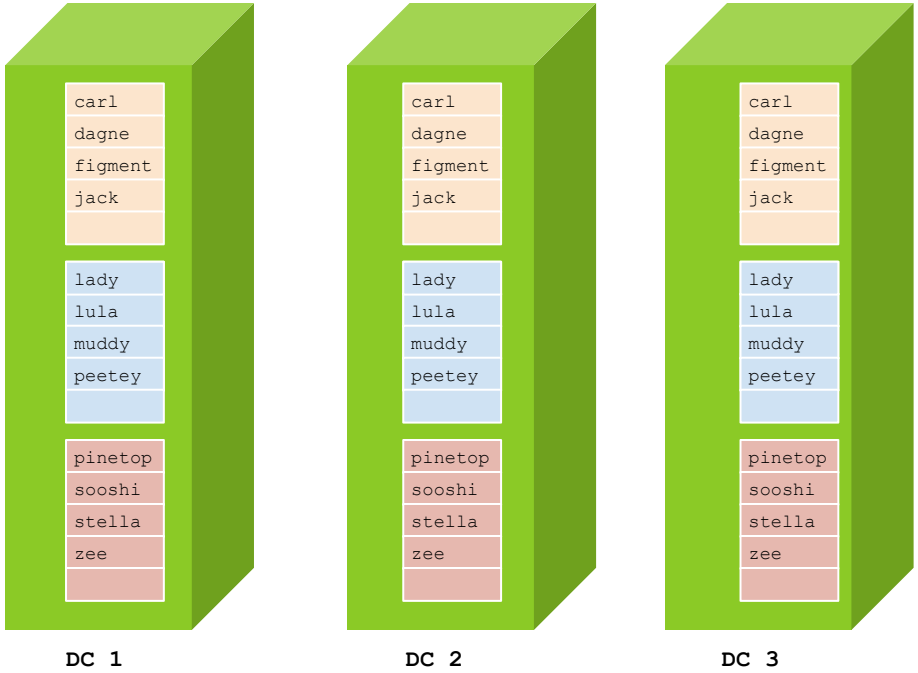


t = 0

Replicated Commit

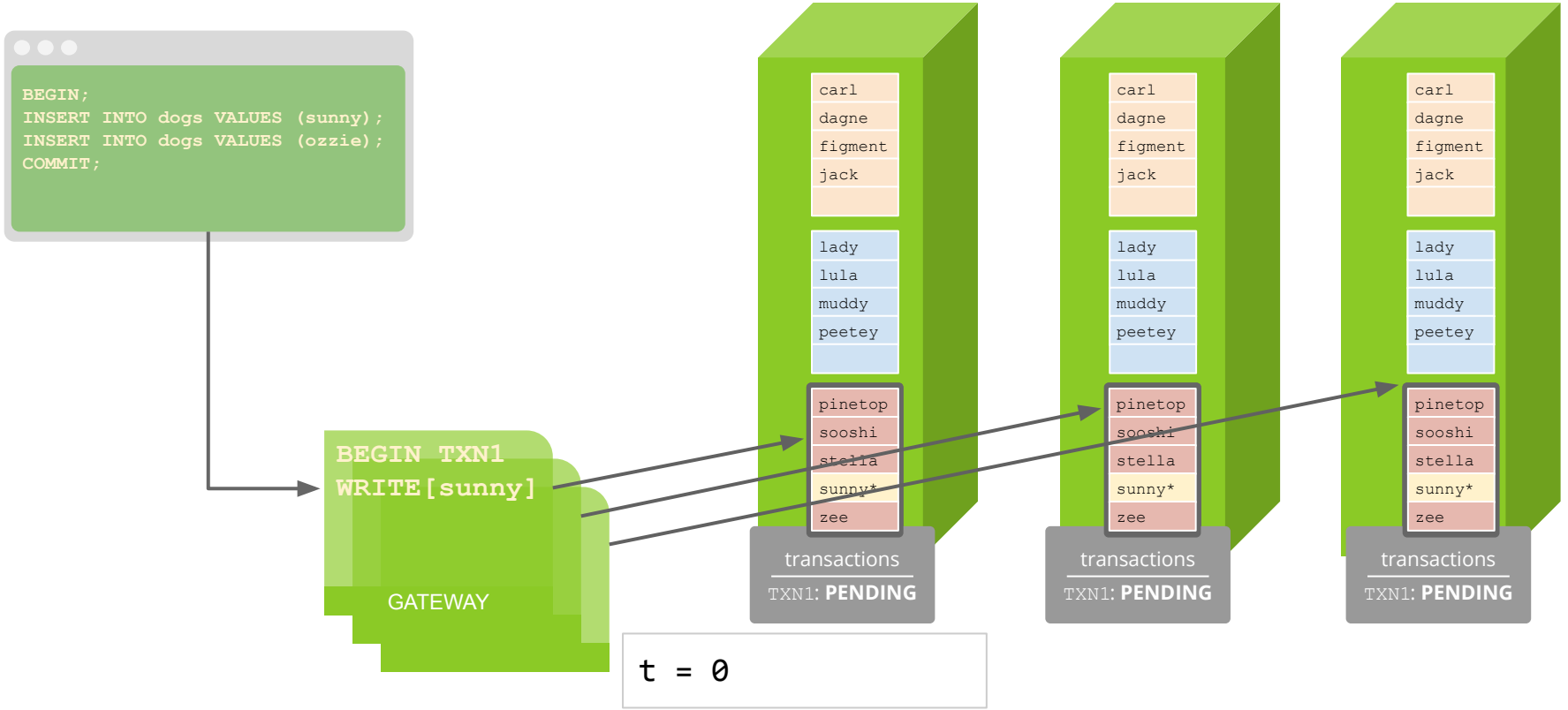




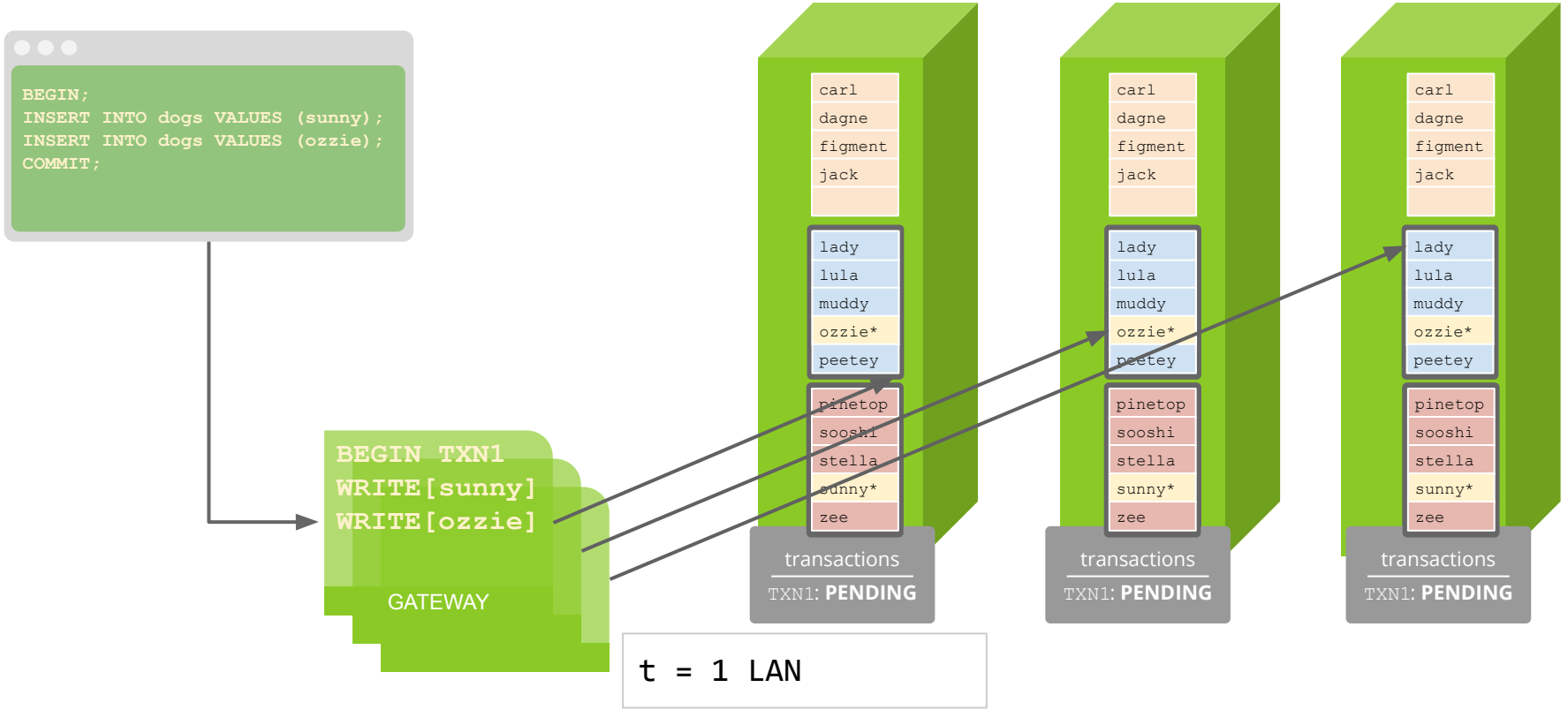


t = 0

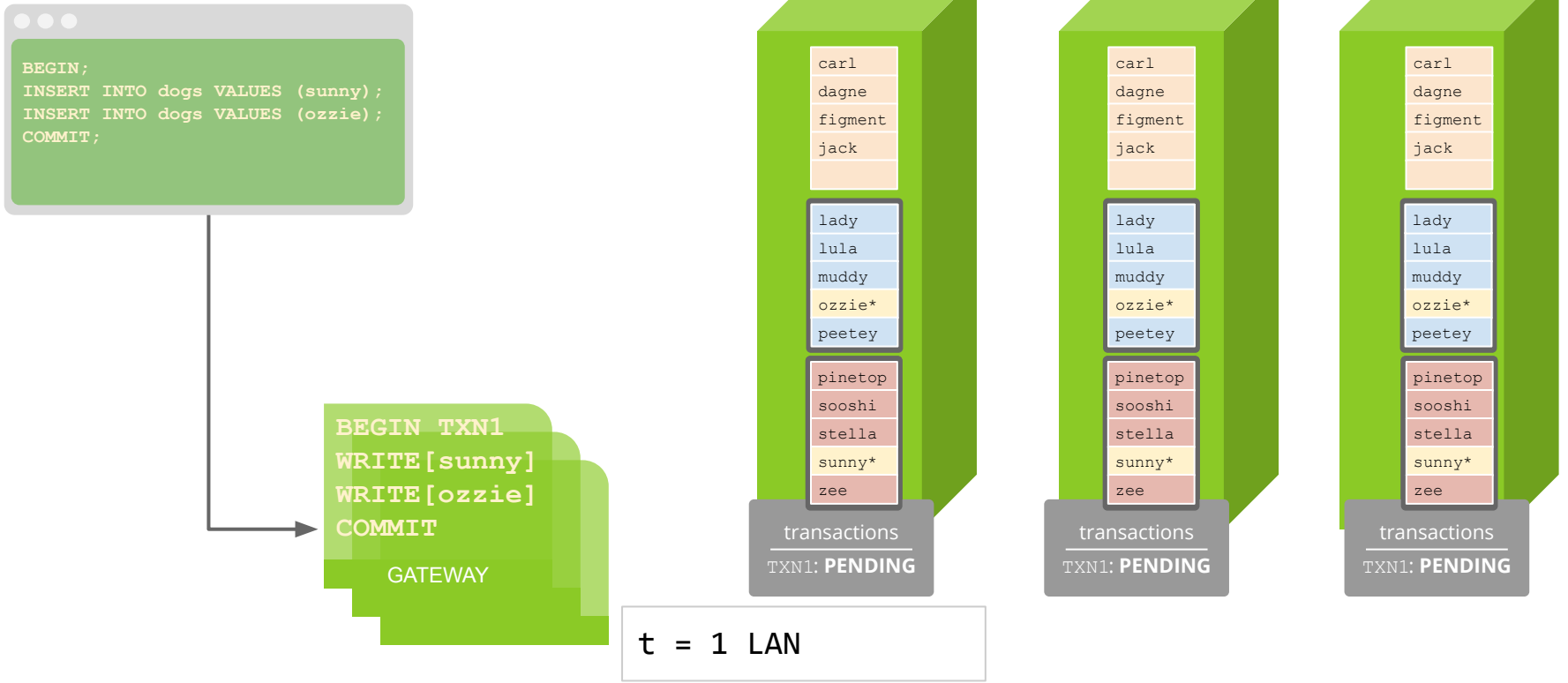
Replicated Commit



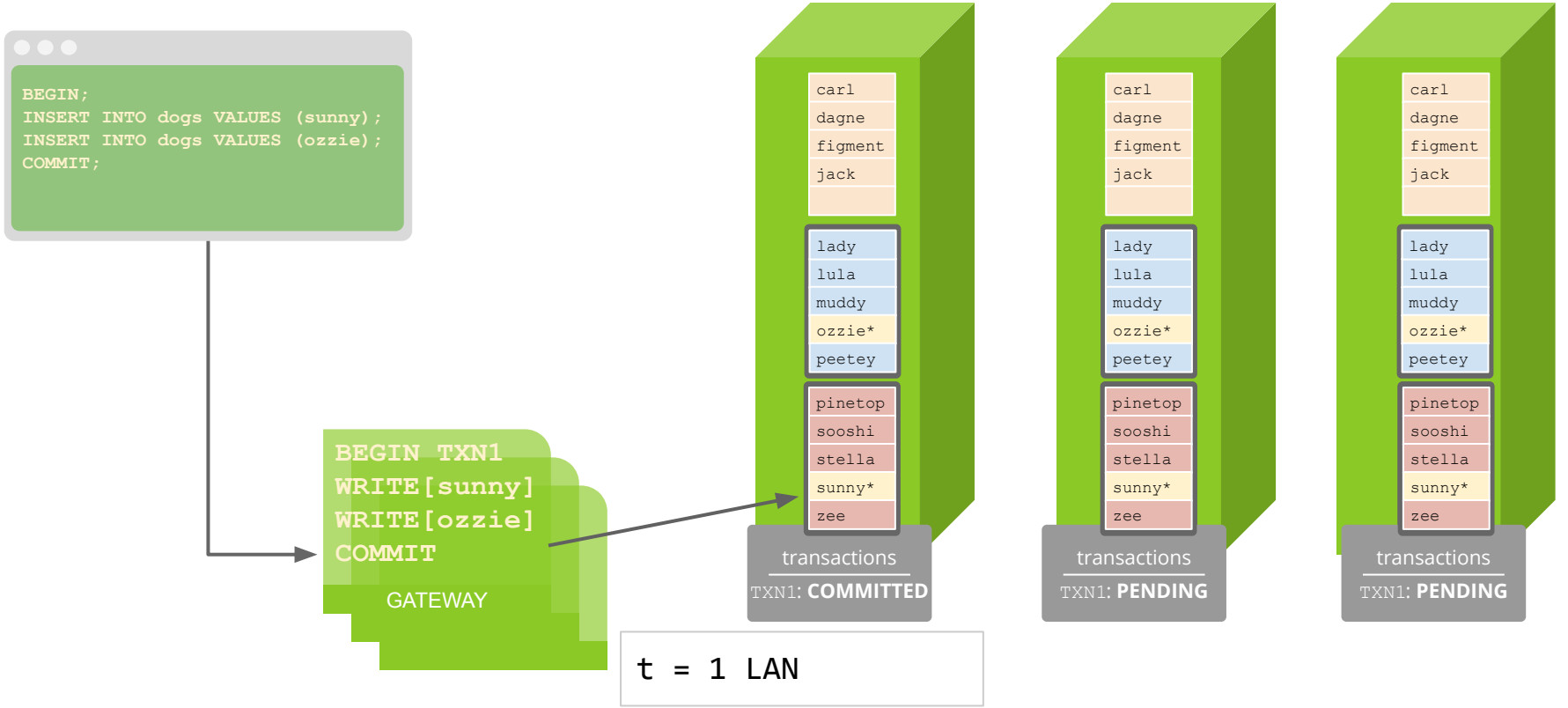
Replicated Commit



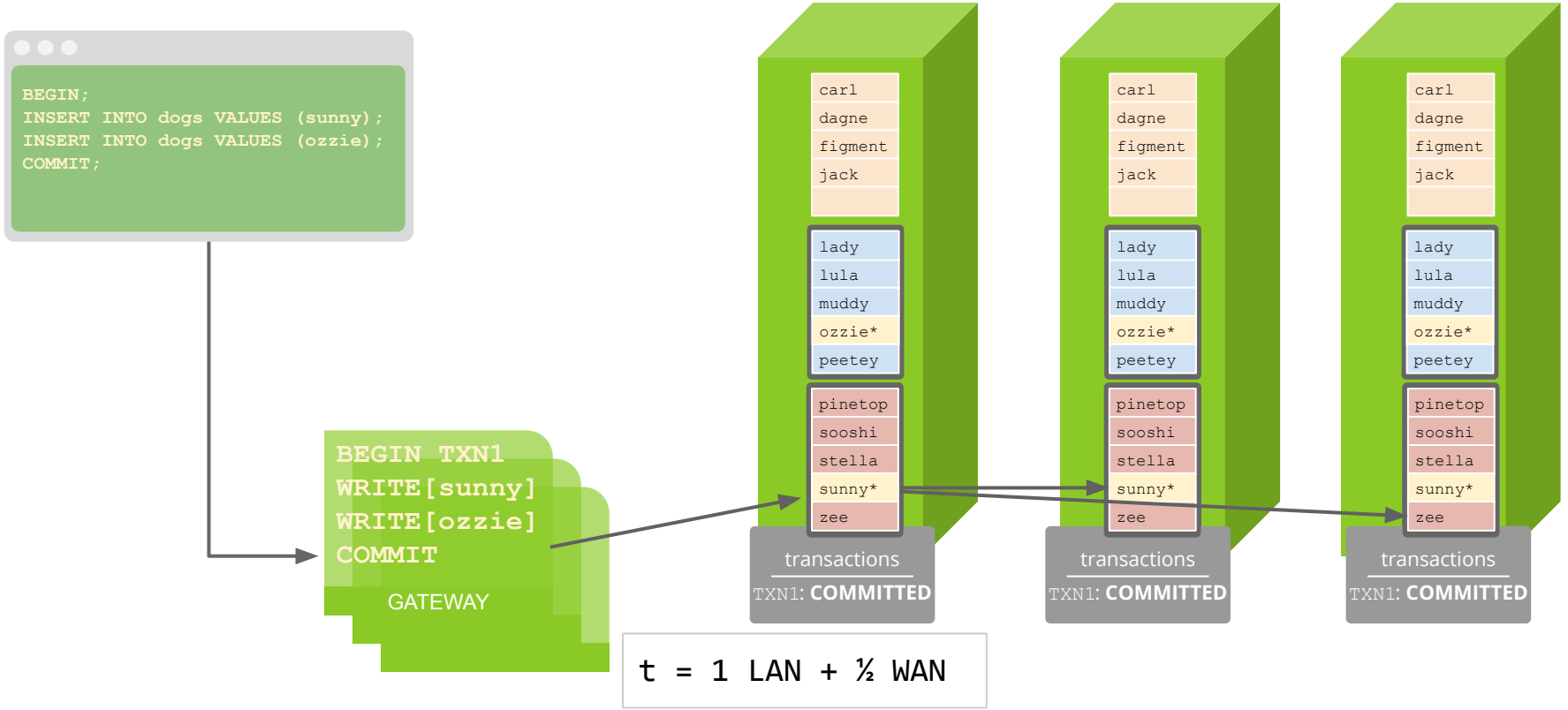
Replicated Commit



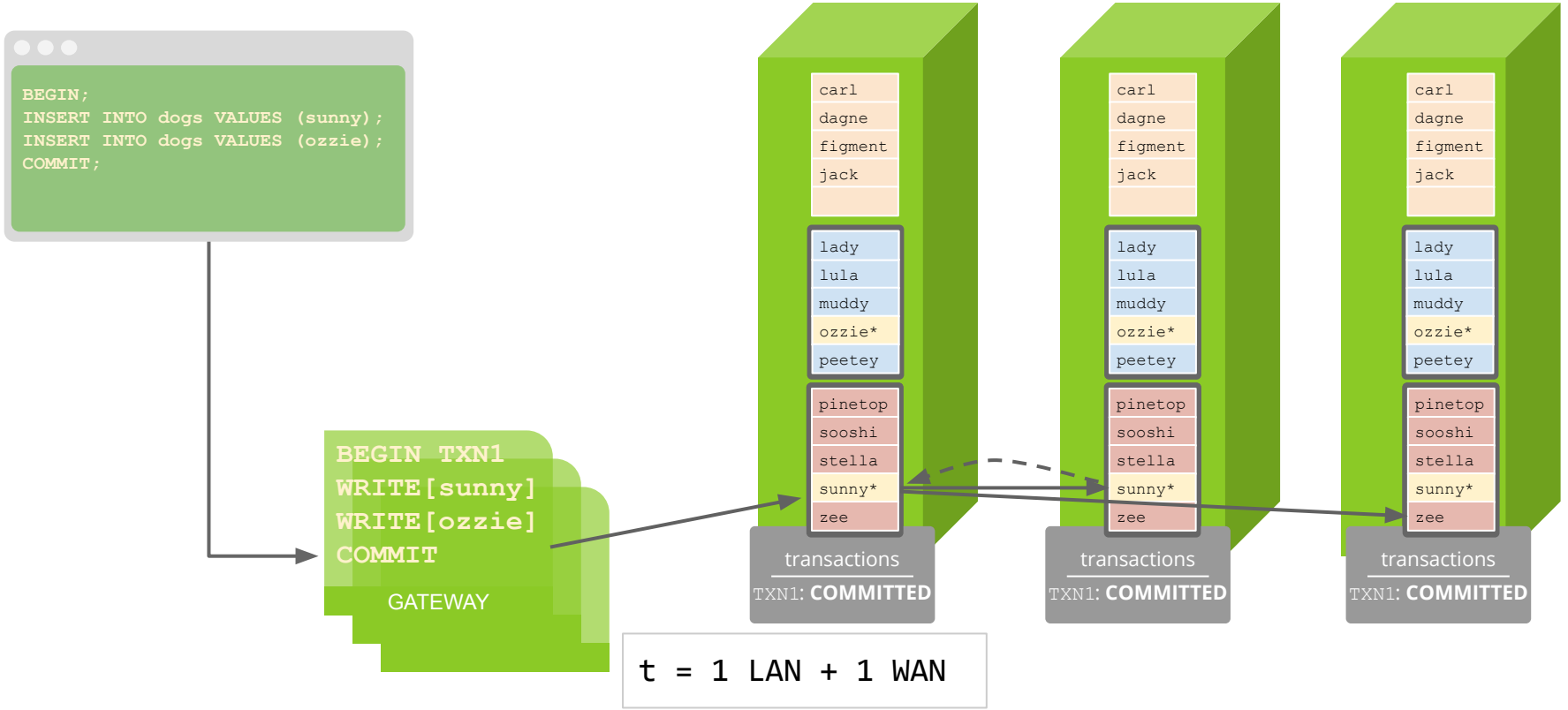
Replicated Commit



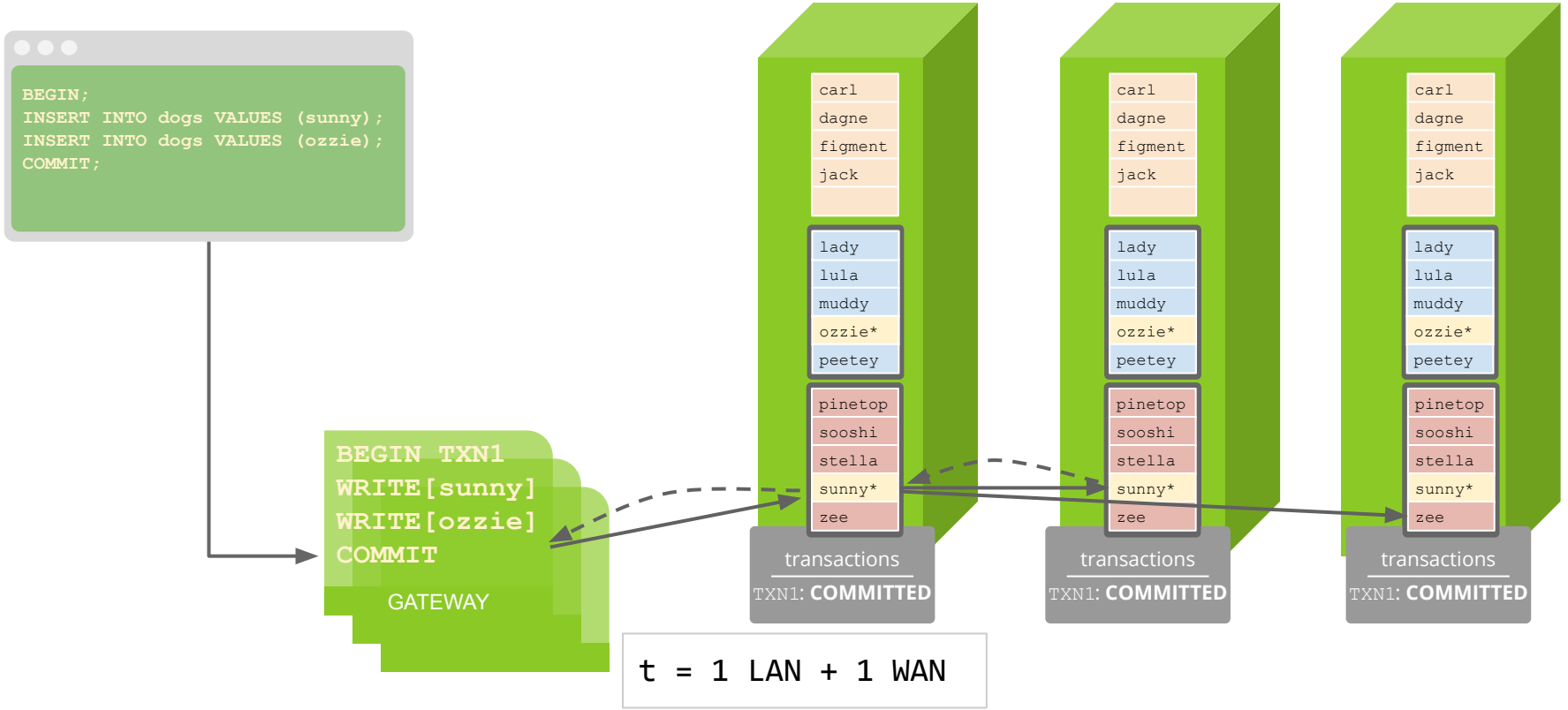
Replicated Commit



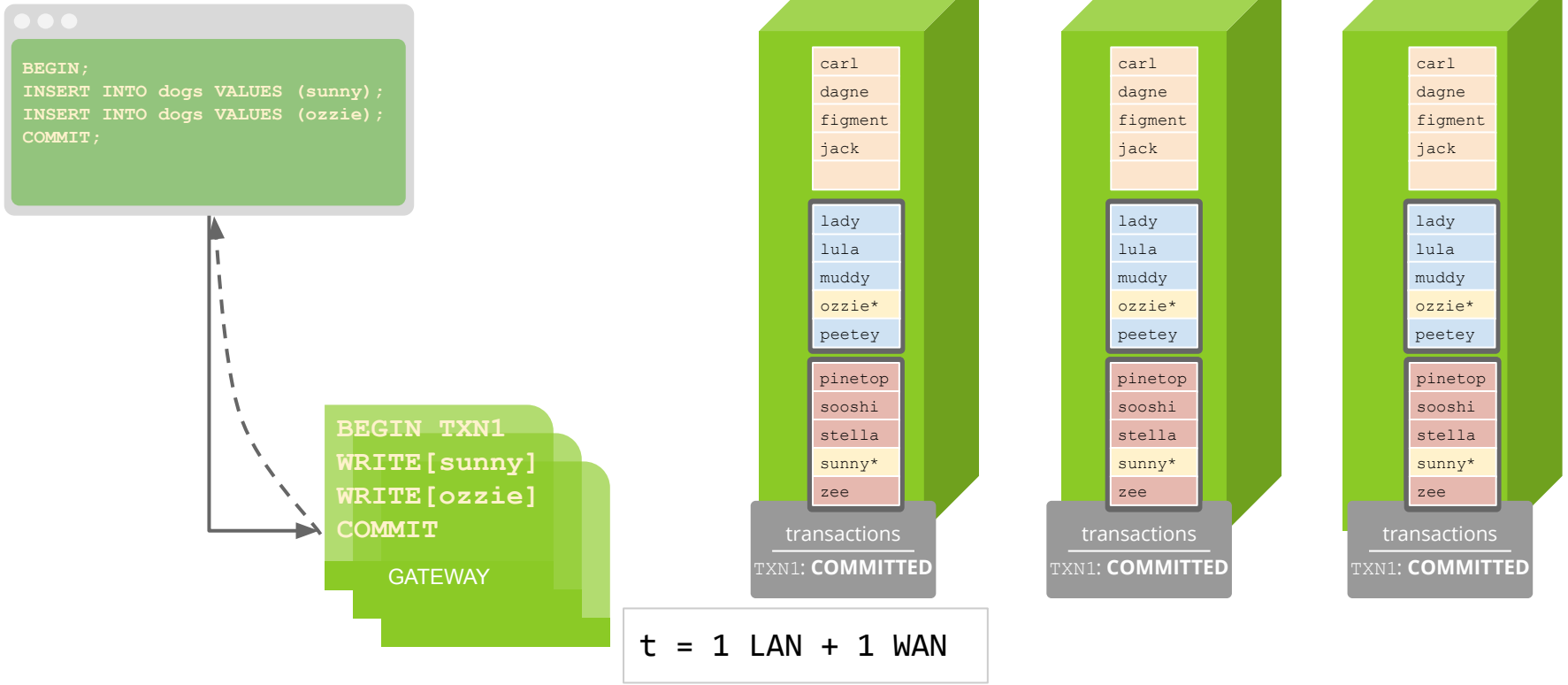
Replicated Commit



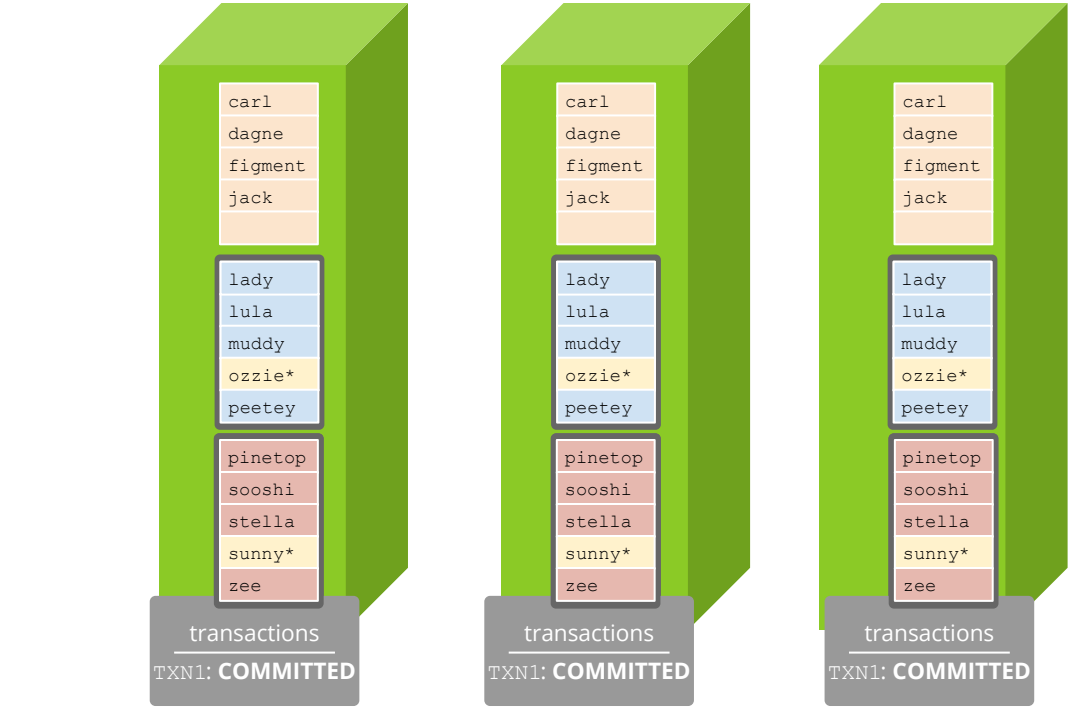
Replicated Commit



Replicated Commit

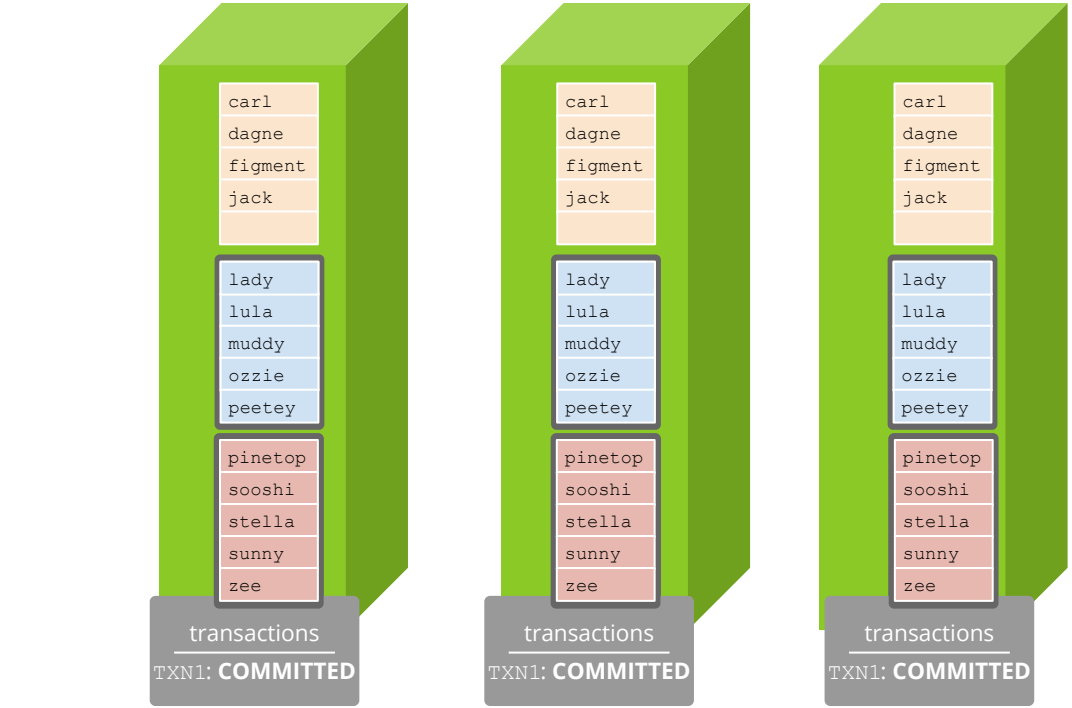


Replicated Commit



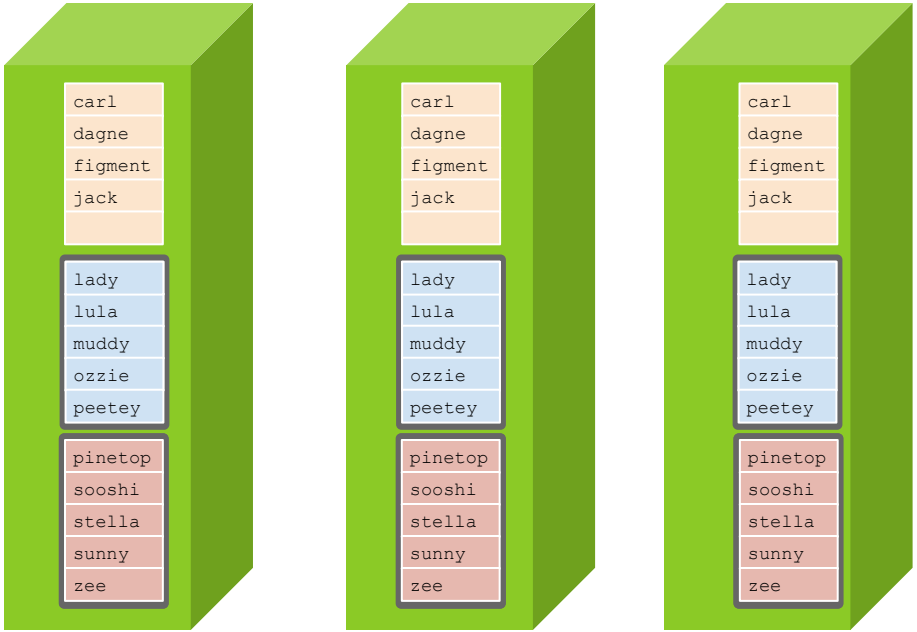

```
t = 1 LAN + 1 WAN
```

Replicated Commit



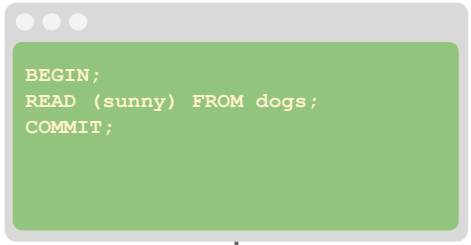
`t = 1 LAN + 1 WAN`

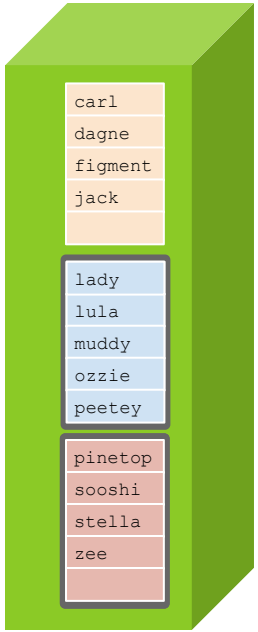
Replicated Commit

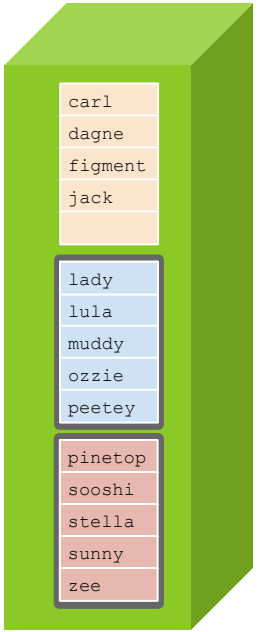


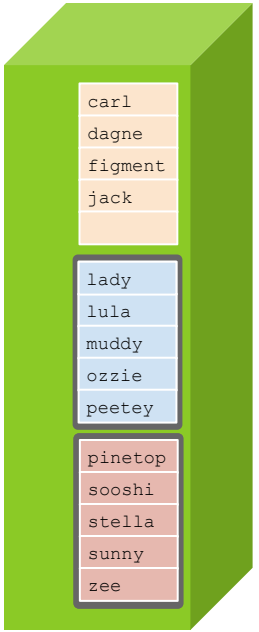
t = 1 LAN + 1 WAN

Replicated Commit (reads)



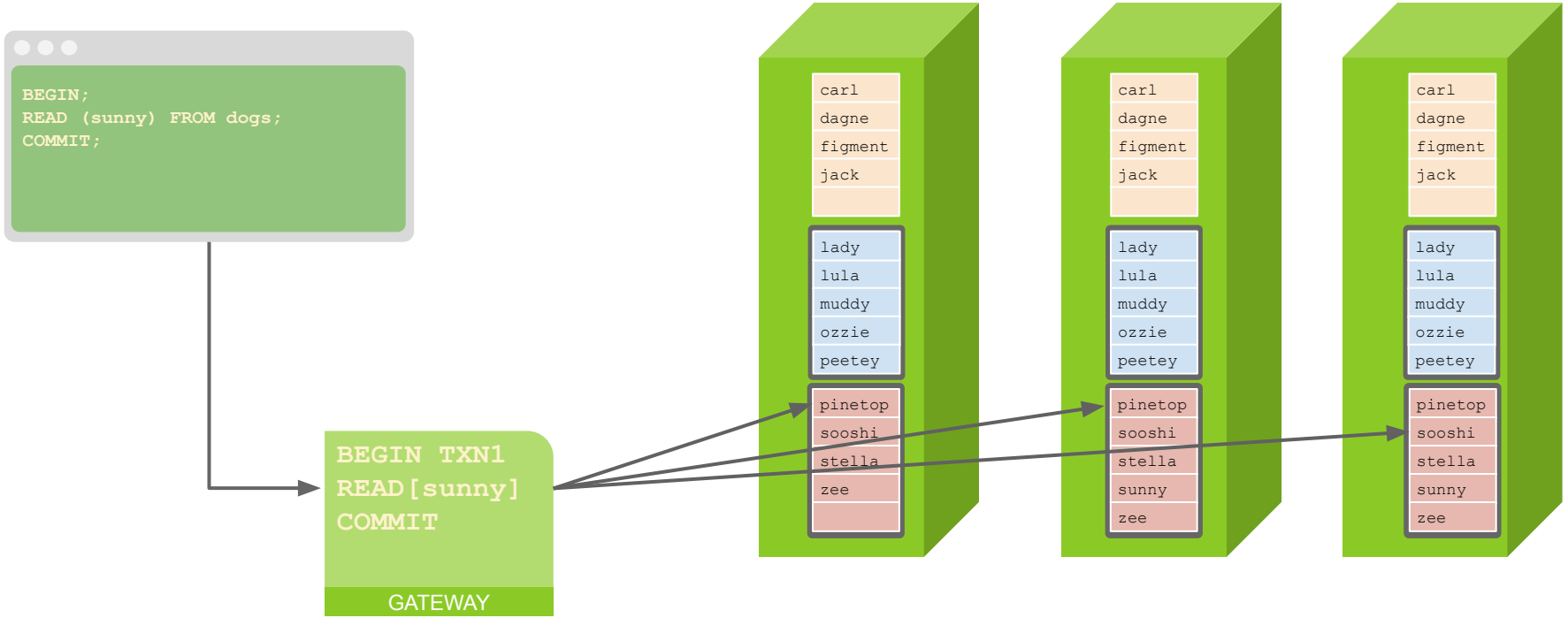






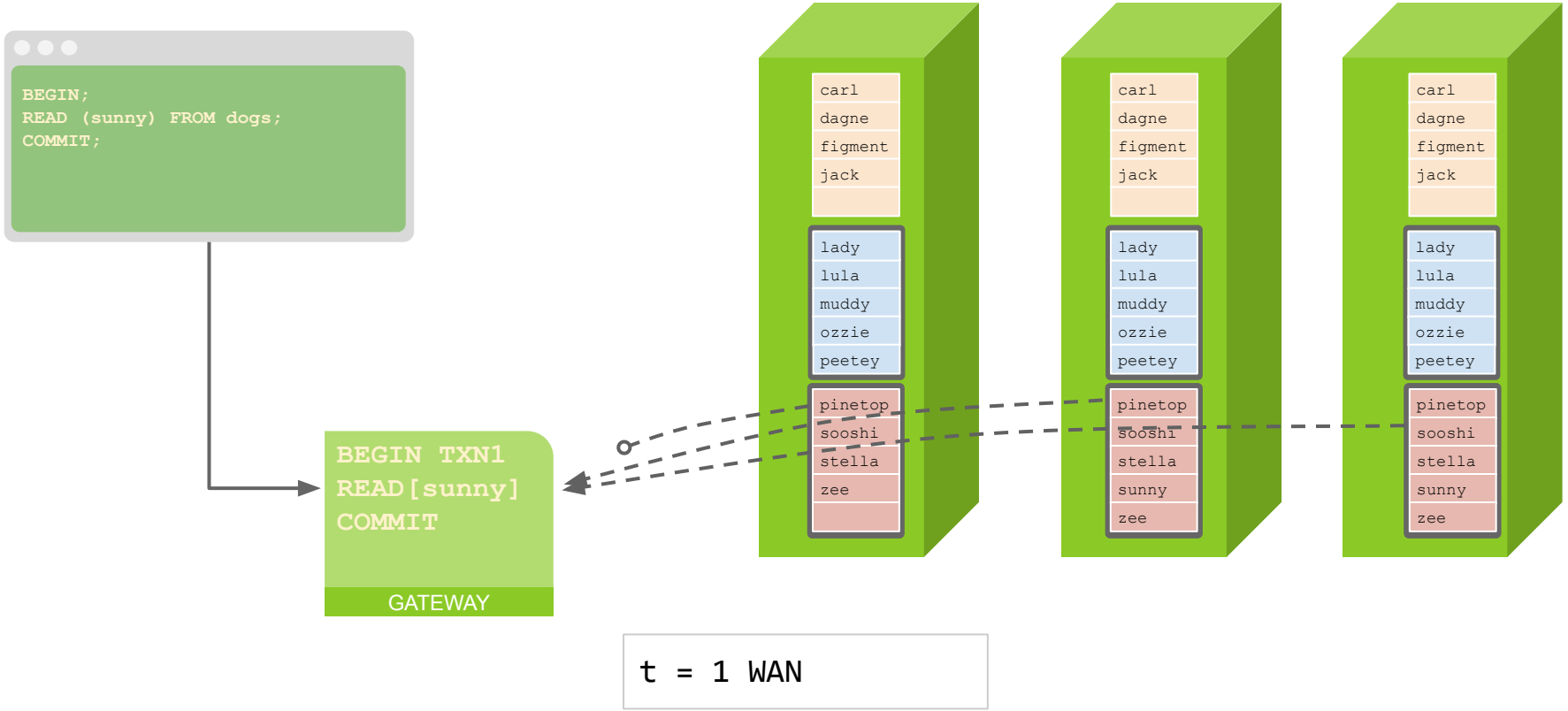
t = 0

Replicated Commit (reads)



t = ½ WAN

Replicated Commit (reads)



Agenda

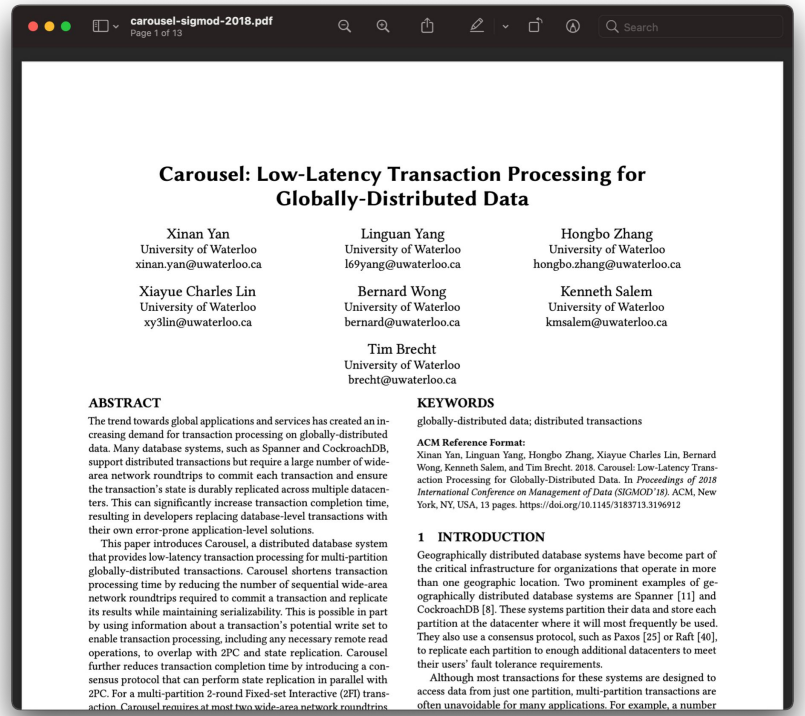
1. Foundations
2. Transactions
3. **Implementations**

I. Spanner/Pipelined Transactions
II. Parallel Commits
III. Replicated Commit
IV. Carousel
V. MDCC
VI. SLOG/OceanVista
VII. TAPIR

Carousel

From Waterloo, 2018. Limits transaction model to 2FI (2-fixed set interactive): a round of reads followed by round of writes, with all keys declared in advance. Write values can depend on reads, but keys written to/read from cannot.

Uses Fast Paxos to replicate to all replicas directly, avoiding the leader hop. Retries on conflict, given it's optimistic.



carousel-sigmod-2018.pdf
Page 1 of 13

Carousel: Low-Latency Transaction Processing for Globally-Distributed Data

Xinan Yan
University of Waterloo
xinan.yan@uwaterloo.ca

Linguan Yang
University of Waterloo
l69yang@uwaterloo.ca

Hongbo Zhang
University of Waterloo
hongbo.zhang@uwaterloo.ca

Xiayue Charles Lin
University of Waterloo
xy3lin@uwaterloo.ca

Bernard Wong
University of Waterloo
bernard@uwaterloo.ca

Kenneth Salem
University of Waterloo
kmsalem@uwaterloo.ca

Tim Brecht
University of Waterloo
brecht@uwaterloo.ca

ABSTRACT

The trend towards global applications and services has created an increasing demand for transaction processing on globally-distributed data. Many database systems, such as Spanner and CockroachDB, support distributed transactions but require a large number of wide-area network roundtrips to commit each transaction and ensure the transaction's state is durably replicated across multiple datacenters. This can significantly increase transaction completion time, resulting in developers replacing database-level transactions with their own error-prone application-level solutions.

This paper introduces Carousel, a distributed database system that provides low-latency transaction processing for multi-partition globally-distributed transactions. Carousel shortens transaction processing time by reducing the number of sequential wide-area network roundtrips required to commit a transaction and replicate its results while maintaining serializability. This is possible in part by using information about a transaction's potential write set to enable transaction processing, including any necessary remote read operations, to overlap with 2PC and state replication. Carousel further reduces transaction completion time by introducing a consensus protocol that can perform state replication in parallel with 2PC. For a multi-partition 2-round Fixed-set Interactive (2FI) transaction, Carousel requires at most two wide-area network roundtrips

KEYWORDS

globally-distributed data; distributed transactions

ACM Reference Format:

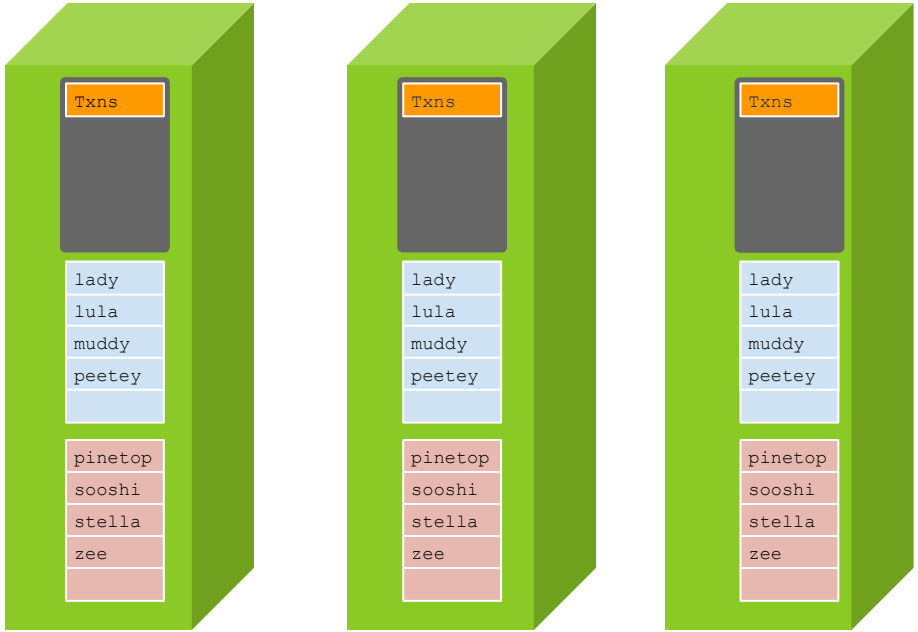
Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. 2018. Carousel: Low-Latency Transaction Processing for Globally-Distributed Data. In *Proceedings of 2018 International Conference on Management of Data (SIGMOD'18).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3183713.3196912

1 INTRODUCTION

Geographically distributed database systems have become part of the critical infrastructure for organizations that operate in more than one geographic location. Two prominent examples of geographically distributed database systems are Spanner [11] and CockroachDB [8]. These systems partition their data and store each partition at the datacenter where it will most frequently be used. They also use a consensus protocol, such as Paxos [25] or Raft [40], to replicate each partition to enough additional datacenters to meet their users' fault tolerance requirements.

Although most transactions for these systems are designed to access data from just one partition, multi-partition transactions are often unavoidable for many applications. For example, a number

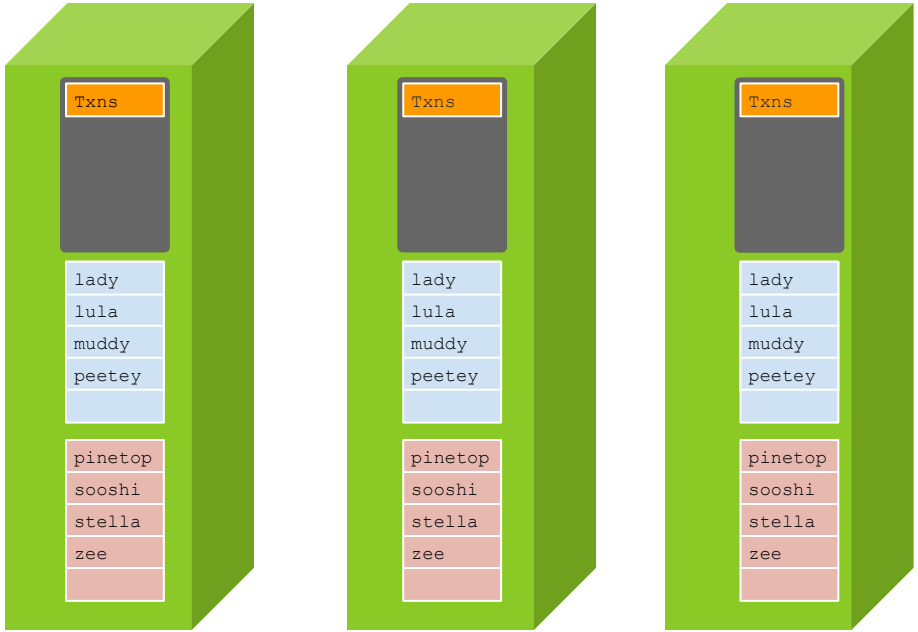
Carousel


Txns
lady
lula
muddy
peetey
pinetop
sooshi
stella
zee
Txns
lady
lula
muddy
peetey
pinetop
sooshi
stella
zee
Txns
lady
lula
muddy
peetey
pinetop
sooshi
stella
zee

t = 0

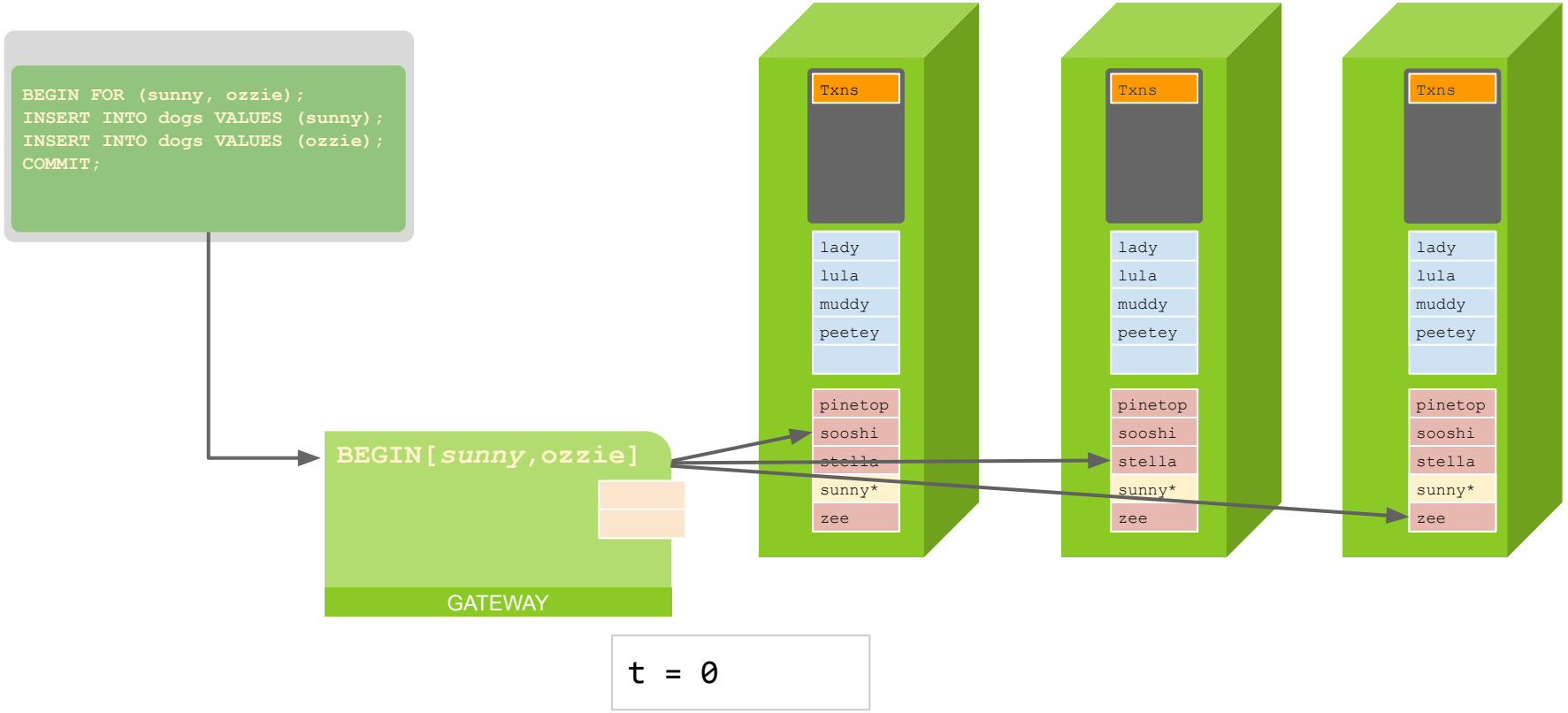
Carousel

```
BEGIN FOR (sunny, ozzie);
INSERT INTO dogs VALUES (sunny);
INSERT INTO dogs VALUES (ozzie);
COMMIT;
```

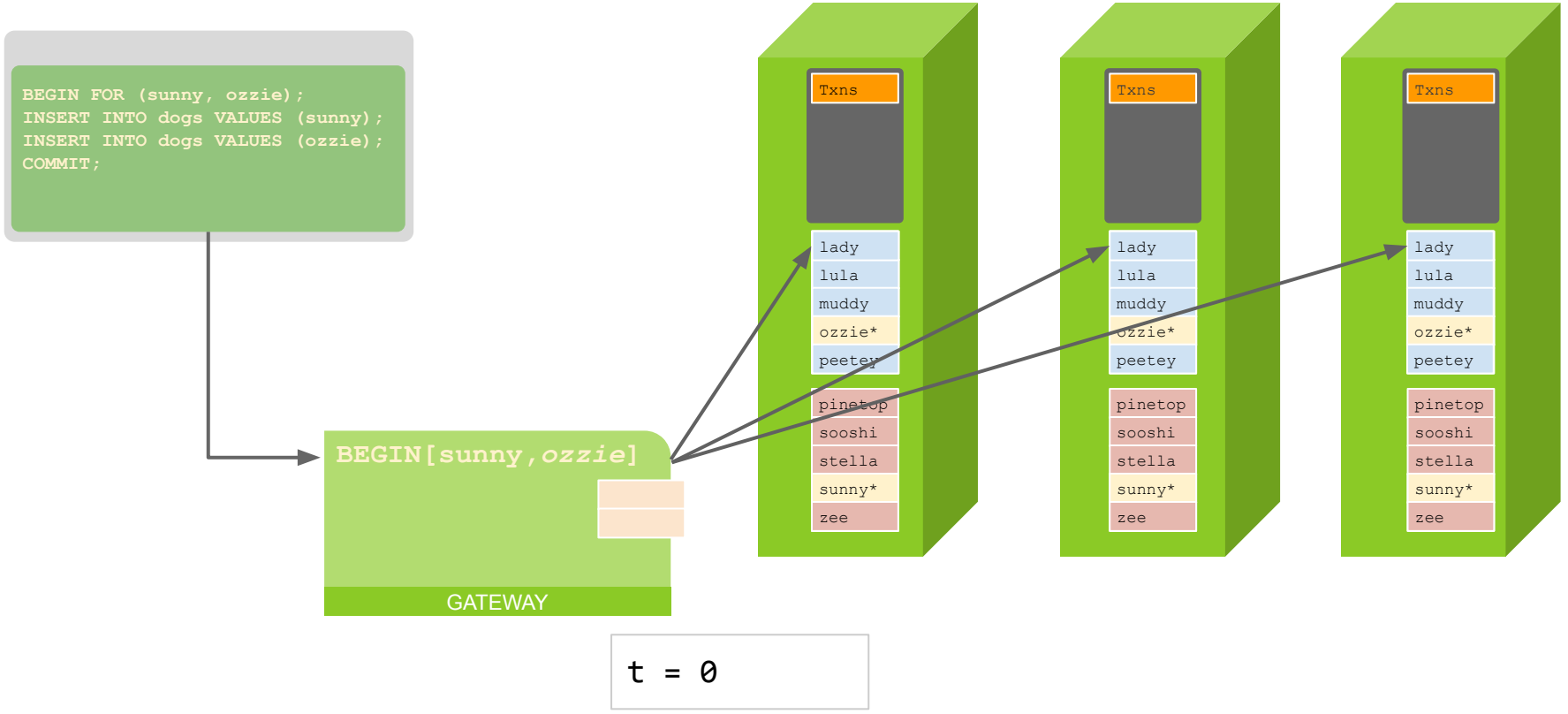


t = 0

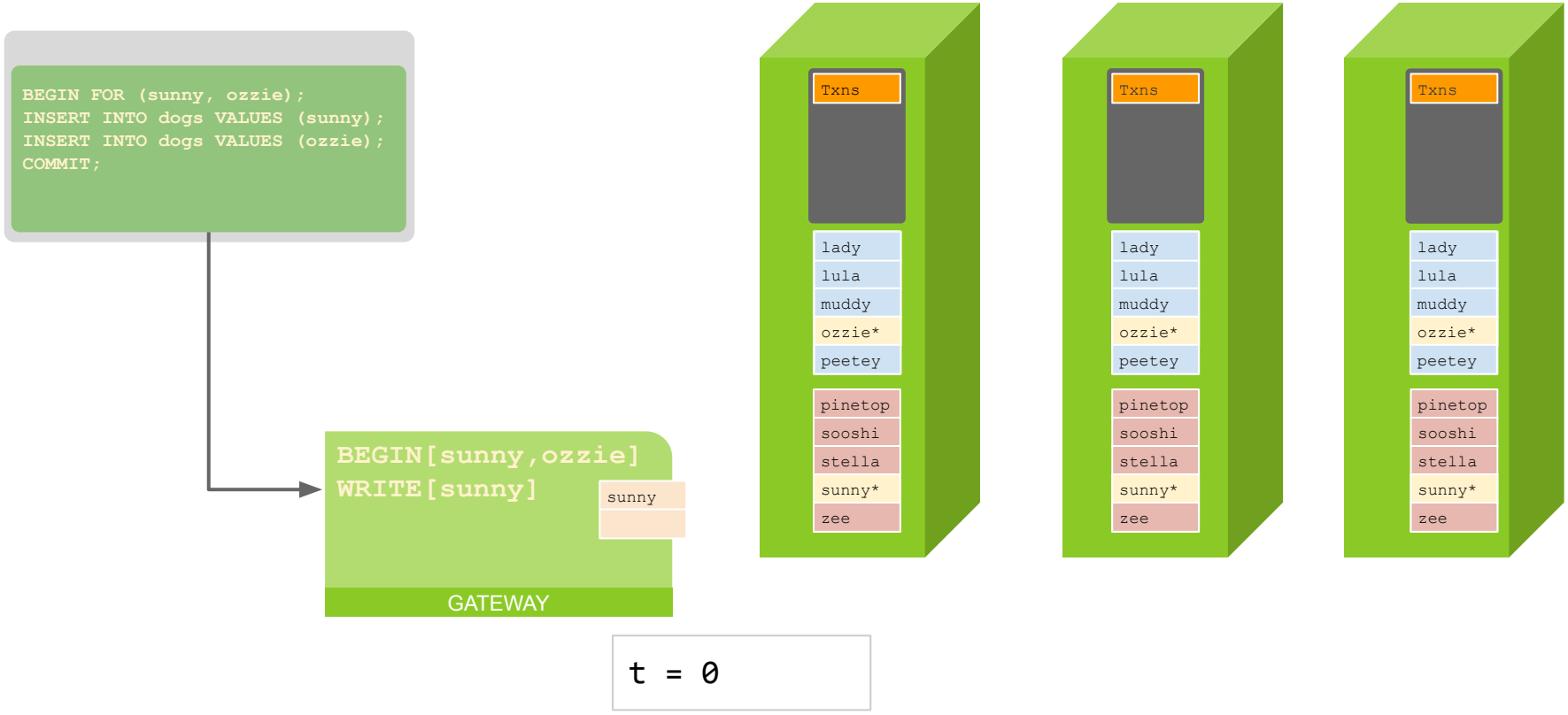
Carousel


BEGIN FOR (sunny, ozzie);
INSERT INTO dogs VALUES (sunny);
INSERT INTO dogs VALUES (ozzie);
COMMIT;
BEGIN[sunny,ozzie]
GATEWAY
Txns
lady
lula
muddy
peetey
pinetop
sooshi
stella
sunny*
zee
Txns
lady
lula
muddy
peetey
pinetop
sooshi
stella
sunny*
zee
Txns
lady
lula
muddy
peetey
pinetop
sooshi
stella
sunny*
zee
t = 0

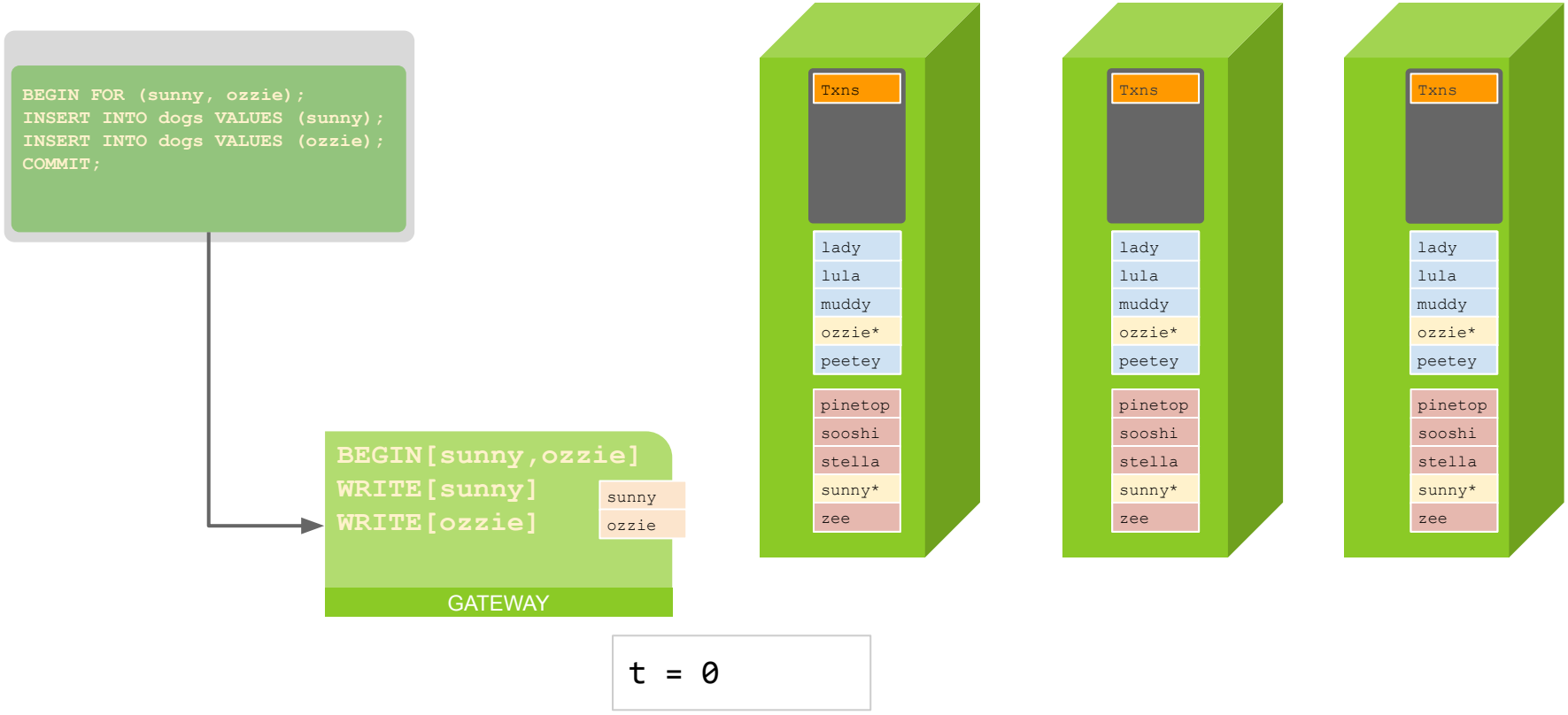
Carousel


BEGIN FOR (sunny, ozzie);
INSERT INTO dogs VALUES (sunny);
INSERT INTO dogs VALUES (ozzie);
COMMIT;
BEGIN[sunny,ozzie]
GATEWAY
Txns
lady
lula
muddy
ozzie*
peetey
pinetop
sooshi
stella
sunny*
zee
t = 0

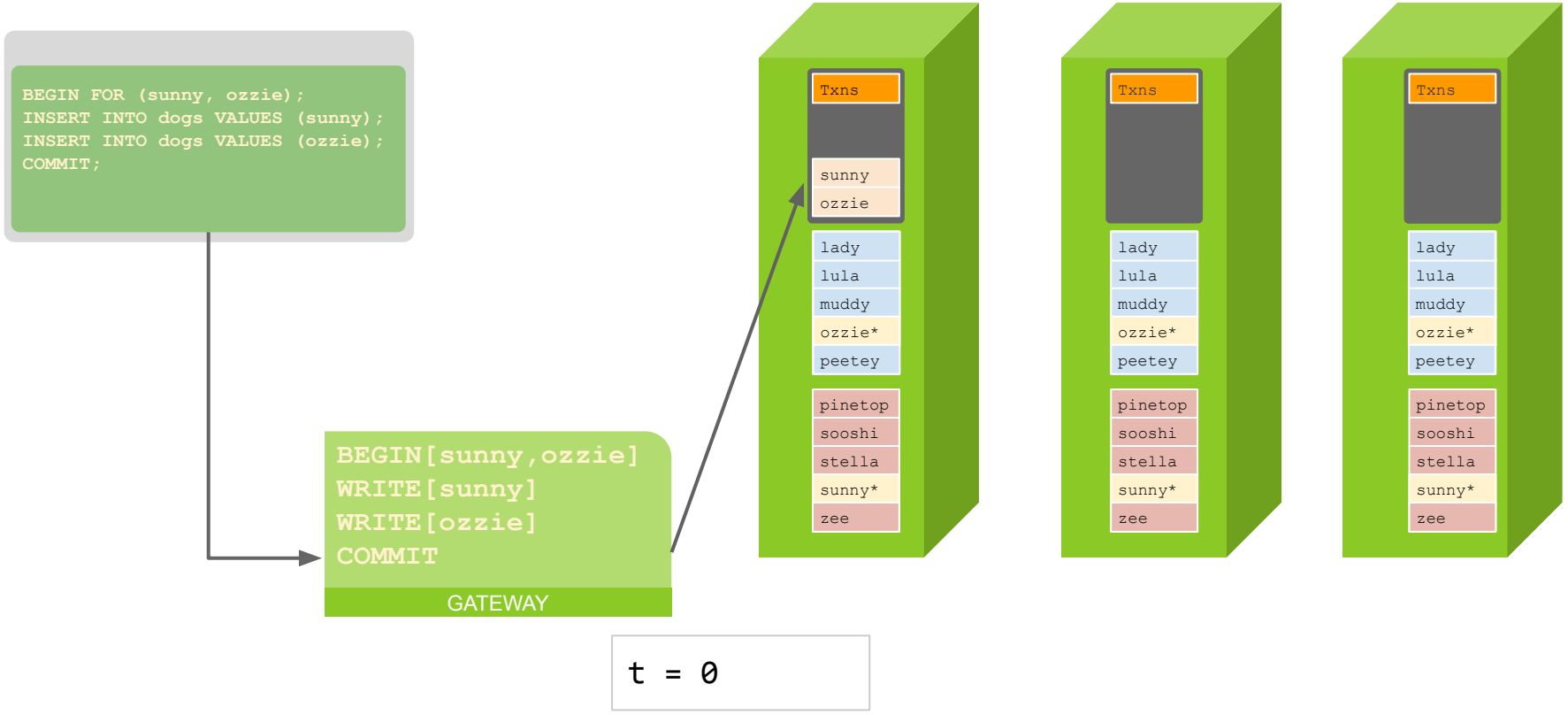
Carousel


BEGIN FOR (sunny, ozzie);
INSERT INTO dogs VALUES (sunny);
INSERT INTO dogs VALUES (ozzie);
COMMIT;
BEGIN[sunny,ozzie]
WRITE[sunny]
sunny
GATEWAY
Txns
lady
lula
muddy
ozzie*
peetey
pinetop
sooshi
stella
sunny*
zee
Txns
lady
lula
muddy
ozzie*
peetey
pinetop
sooshi
stella
sunny*
zee
Txns
lady
lula
muddy
ozzie*
peetey
pinetop
sooshi
stella
sunny*
zee
t = 0

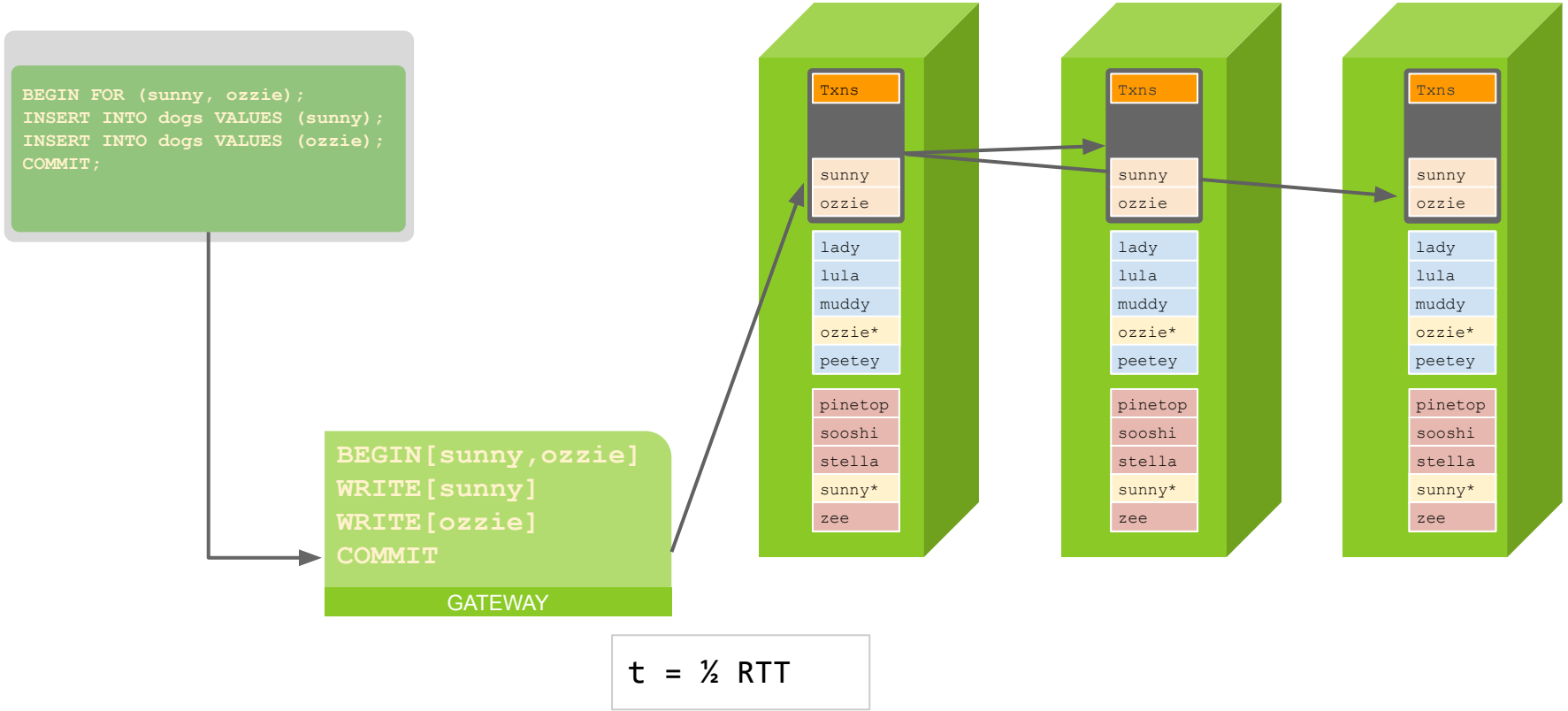
Carousel


BEGIN FOR (sunny, ozzie);
INSERT INTO dogs VALUES (sunny);
INSERT INTO dogs VALUES (ozzie);
COMMIT;
BEGIN[sunny,ozzie]
WRITE[sunny]
WRITE[ozzie]
sunny
ozzie
GATEWAY
Txns
lady
lula
muddy
ozzie*
peetey
pinetop
sooshi
stella
sunny*
zee
Txns
lady
lula
muddy
ozzie*
peetey
pinetop
sooshi
stella
sunny*
zee
Txns
lady
lula
muddy
ozzie*
peetey
pinetop
sooshi
stella
sunny*
zee
t = 0

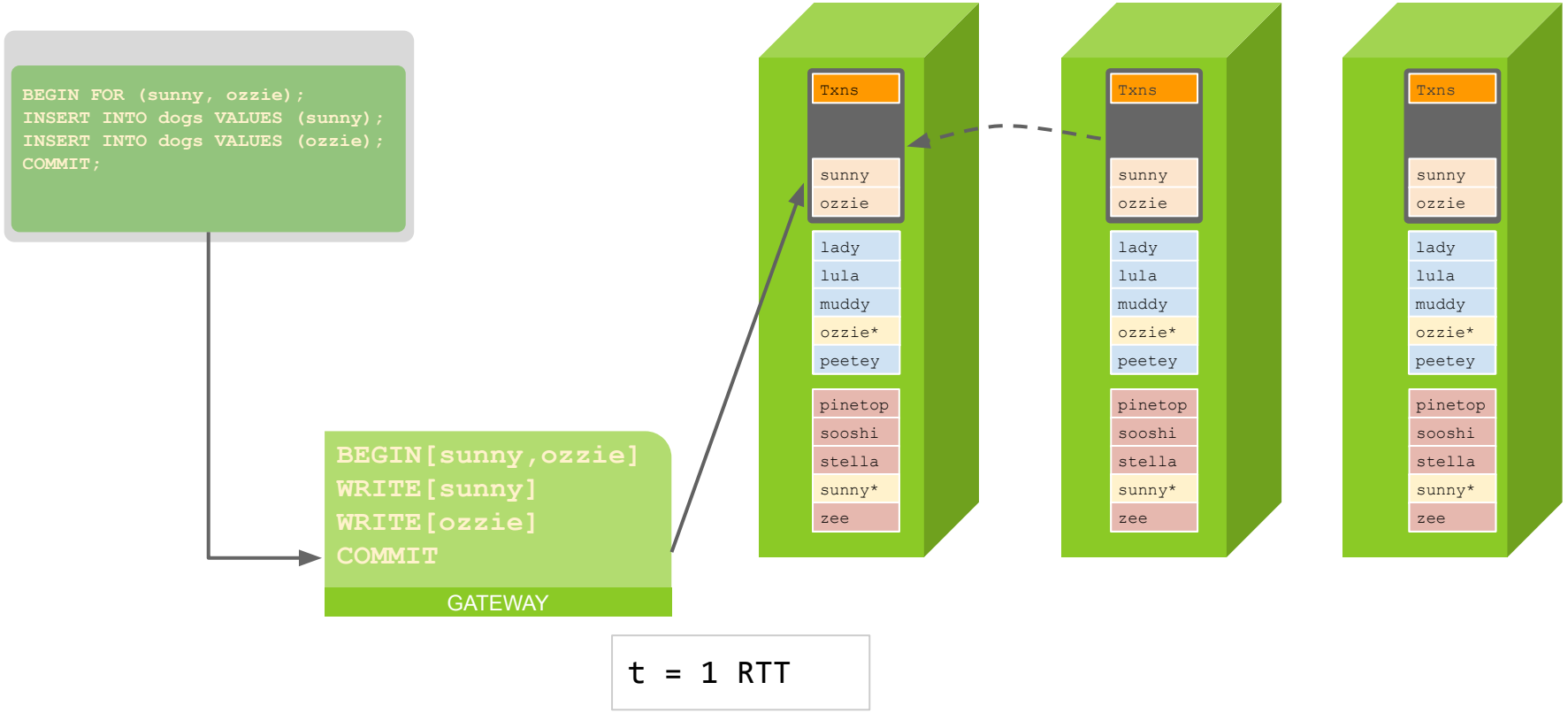
Carousel


BEGIN FOR (sunny, ozzie);
INSERT INTO dogs VALUES (sunny);
INSERT INTO dogs VALUES (ozzie);
COMMIT;
BEGIN[sunny,ozzie]
WRITE[sunny]
WRITE[ozzie]
COMMIT
GATEWAY
Txns
sunny
ozzie
lady
lula
muddy
ozzie*
peetey
pinetop
sooshi
stella
sunny*
zee
Txns
lady
lula
muddy
ozzie*
peetey
pinetop
sooshi
stella
sunny*
zee
Txns
lady
lula
muddy
ozzie*
peetey
pinetop
sooshi
stella
sunny*
zee
t = 0

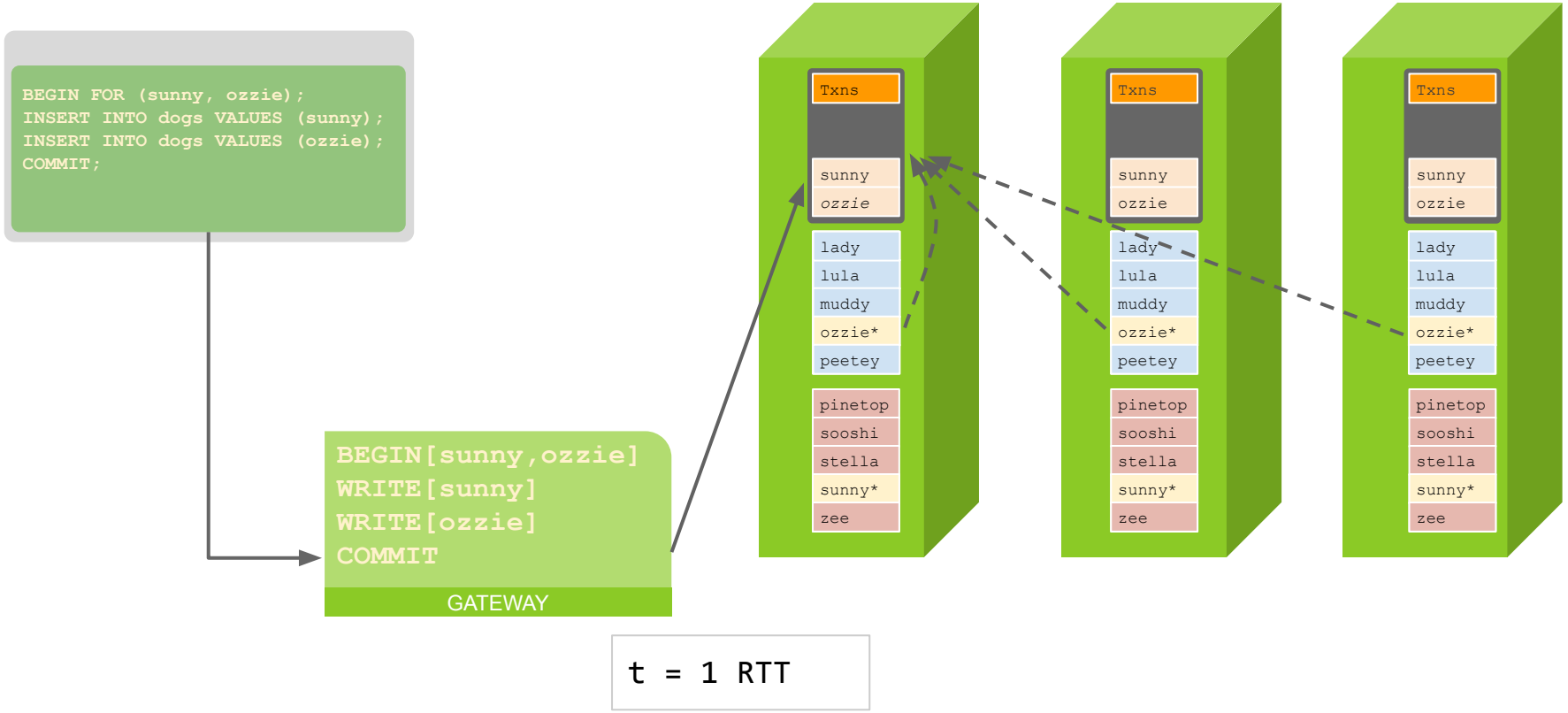
Carousel


BEGIN FOR (sunny, ozzie);
INSERT INTO dogs VALUES (sunny);
INSERT INTO dogs VALUES (ozzie);
COMMIT;
BEGIN[sunny,ozzie]
WRITE[sunny]
WRITE[ozzie]
COMMIT
GATEWAY
Txns
sunny
ozzie
lady
lula
muddy
ozzie*
peetey
pinetop
sooshi
stella
sunny*
zee
Txns
sunny
ozzie
lady
lula
muddy
ozzie*
peetey
pinetop
sooshi
stella
sunny*
zee
Txns
sunny
ozzie
lady
lula
muddy
ozzie*
peetey
pinetop
sooshi
stella
sunny*
zee
t = ½ RTT

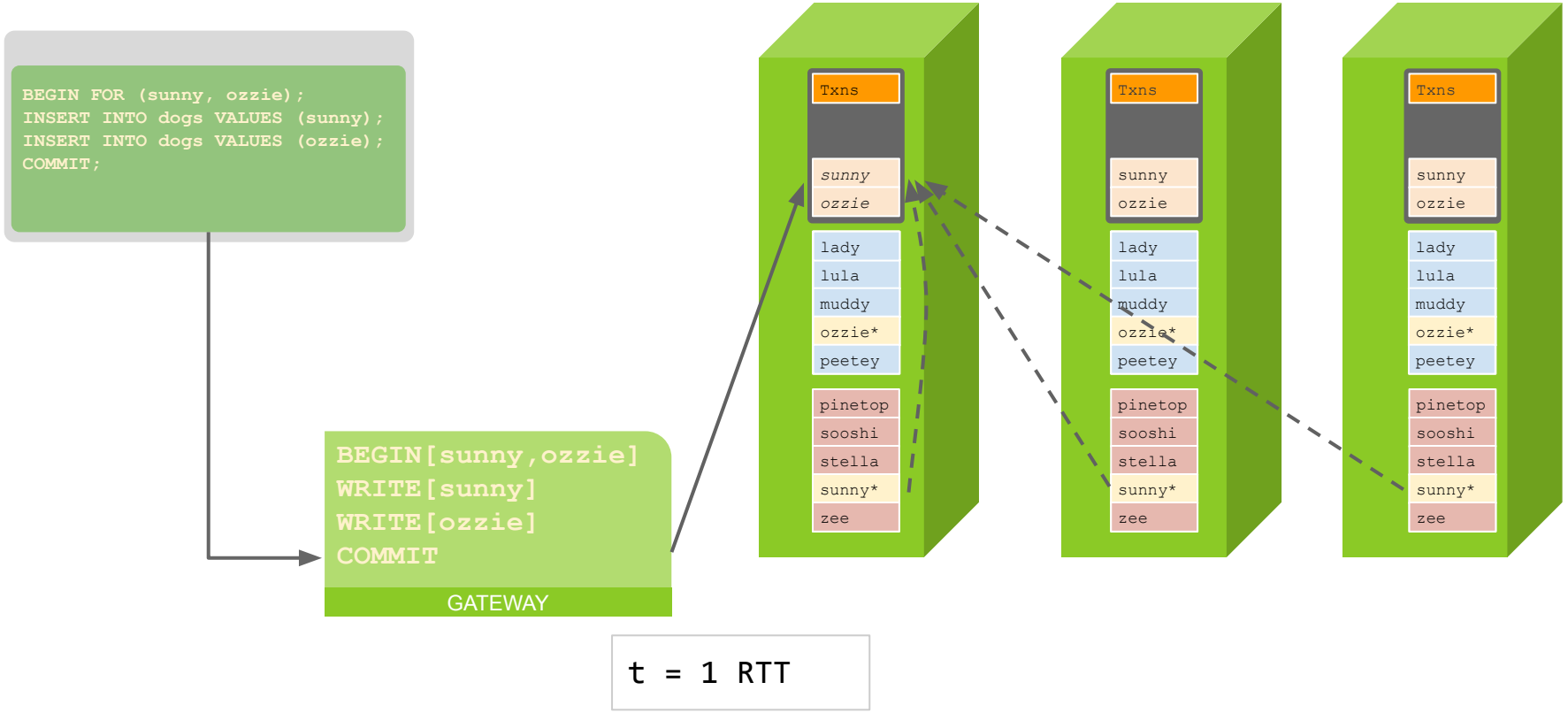
Carousel


BEGIN FOR (sunny, ozzie);
INSERT INTO dogs VALUES (sunny);
INSERT INTO dogs VALUES (ozzie);
COMMIT;
BEGIN[sunny,ozzie]
WRITE[sunny]
WRITE[ozzie]
COMMIT
GATEWAY
Txns
sunny
ozzie
lady
lula
muddy
ozzie*
peetey
pinetop
sooshi
stella
sunny*
zee
Txns
sunny
ozzie
lady
lula
muddy
ozzie*
peetey
pinetop
sooshi
stella
sunny*
zee
Txns
sunny
ozzie
lady
lula
muddy
ozzie*
peetey
pinetop
sooshi
stella
sunny*
zee
t = 1 RTT

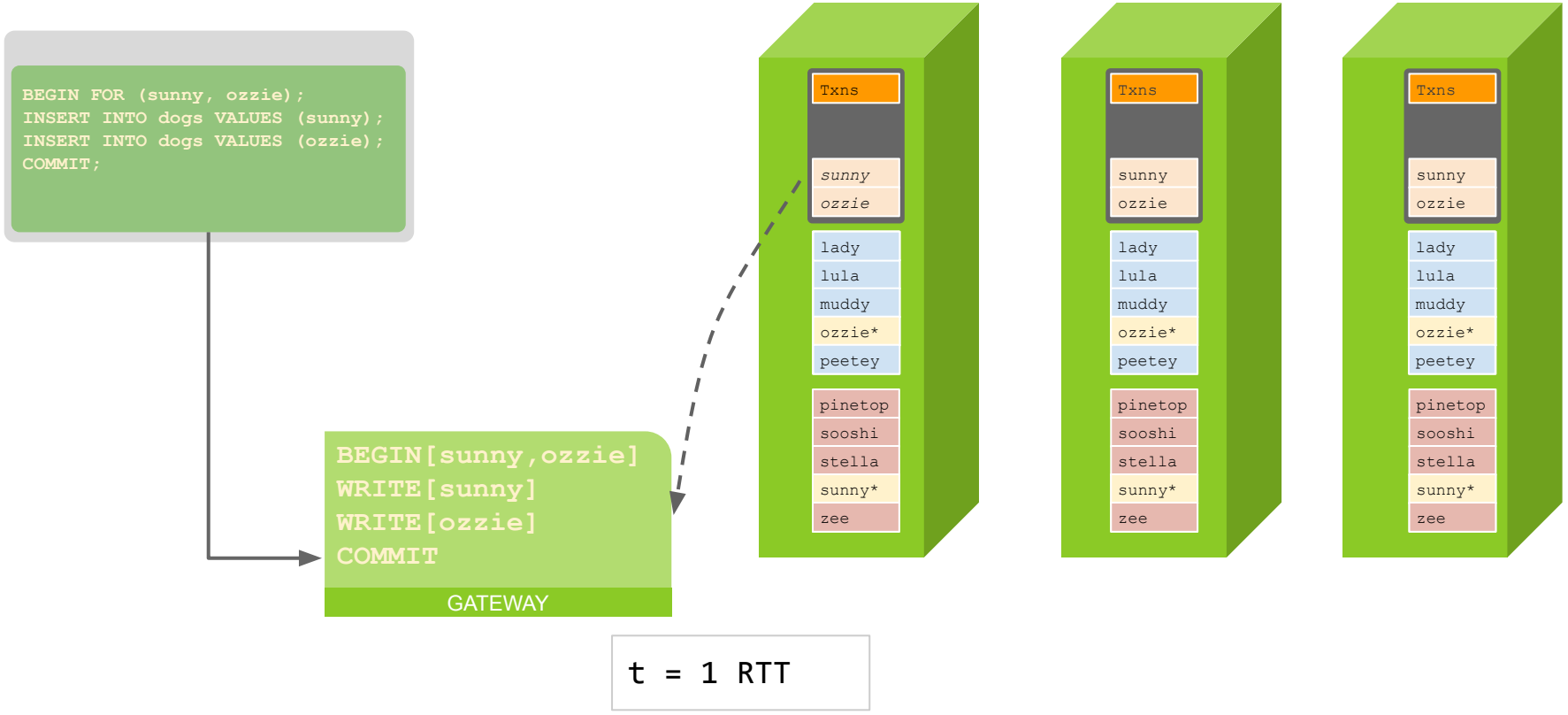
Carousel


BEGIN FOR (sunny, ozzie);
INSERT INTO dogs VALUES (sunny);
INSERT INTO dogs VALUES (ozzie);
COMMIT;
BEGIN[sunny,ozzie]
WRITE[sunny]
WRITE[ozzie]
COMMIT
GATEWAY
Txns
sunny
ozzie
lady
lula
muddy
ozzie*
peetey
pinetop
sooshi
stella
sunny*
zee
Txns
sunny
ozzie
lady
lula
muddy
ozzie*
peetey
pinetop
sooshi
stella
sunny*
zee
Txns
sunny
ozzie
lady
lula
muddy
ozzie*
peetey
pinetop
sooshi
stella
sunny*
zee
t = 1 RTT

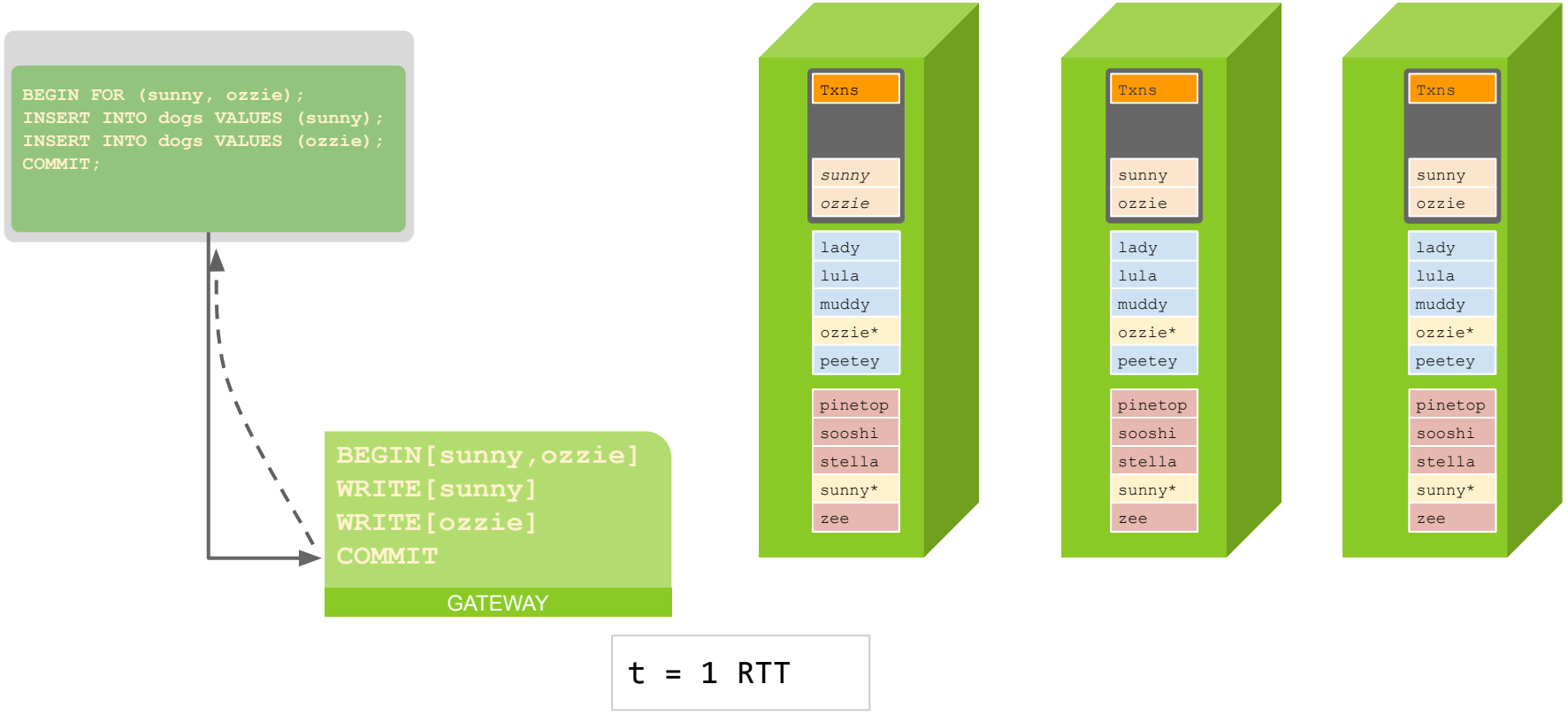
Carousel


BEGIN FOR (sunny, ozzie);
INSERT INTO dogs VALUES (sunny);
INSERT INTO dogs VALUES (ozzie);
COMMIT;
BEGIN[sunny,ozzie]
WRITE[sunny]
WRITE[ozzie]
COMMIT
GATEWAY
Txns
sunny
ozzie
lady
lula
muddy
ozzie*
peetey
pinetop
sooshi
stella
sunny*
zee
t = 1 RTT

Carousel

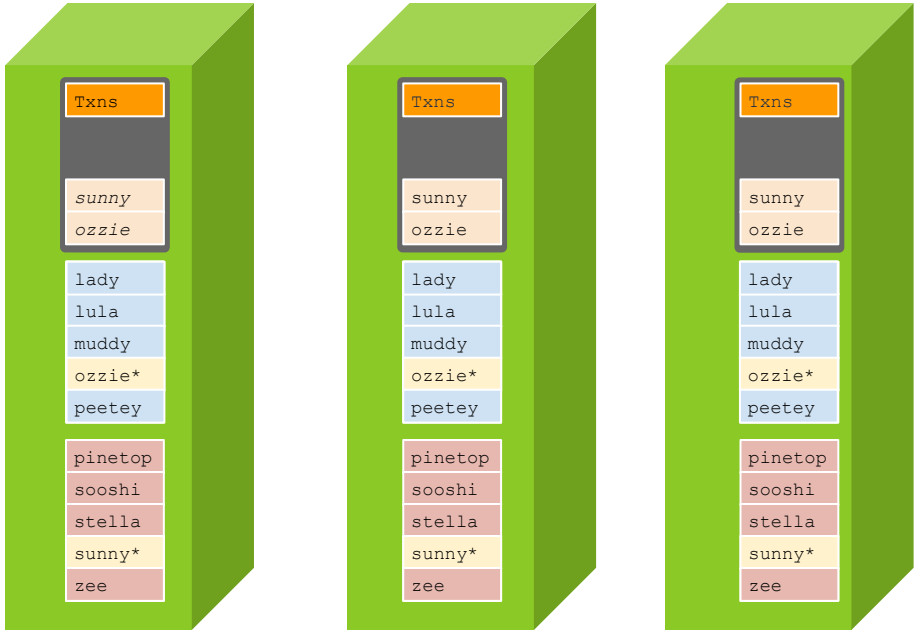

BEGIN FOR (sunny, ozzie);
INSERT INTO dogs VALUES (sunny);
INSERT INTO dogs VALUES (ozzie);
COMMIT;
BEGIN[sunny,ozzie]
WRITE[sunny]
WRITE[ozzie]
COMMIT
GATEWAY
Txns
sunny
ozzie
lady
lula
muddy
ozzie*
peetey
pinetop
sooshi
stella
sunny*
zee
Txns
sunny
ozzie
lady
lula
muddy
ozzie*
peetey
pinetop
sooshi
stella
sunny*
zee
Txns
sunny
ozzie
lady
lula
muddy
ozzie*
peetey
pinetop
sooshi
stella
sunny*
zee
t = 1 RTT

Carousel



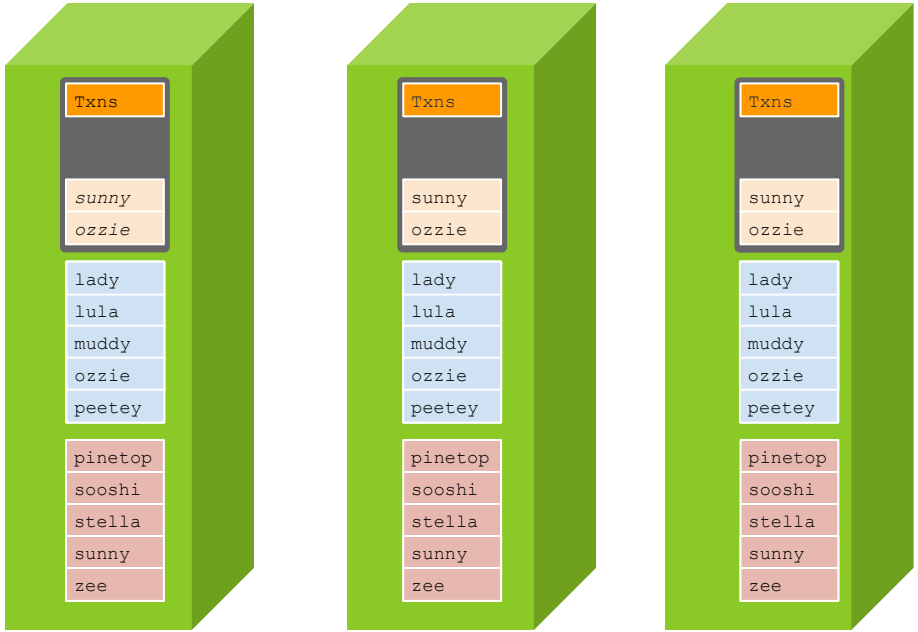
t = 1 RTT

Carousel



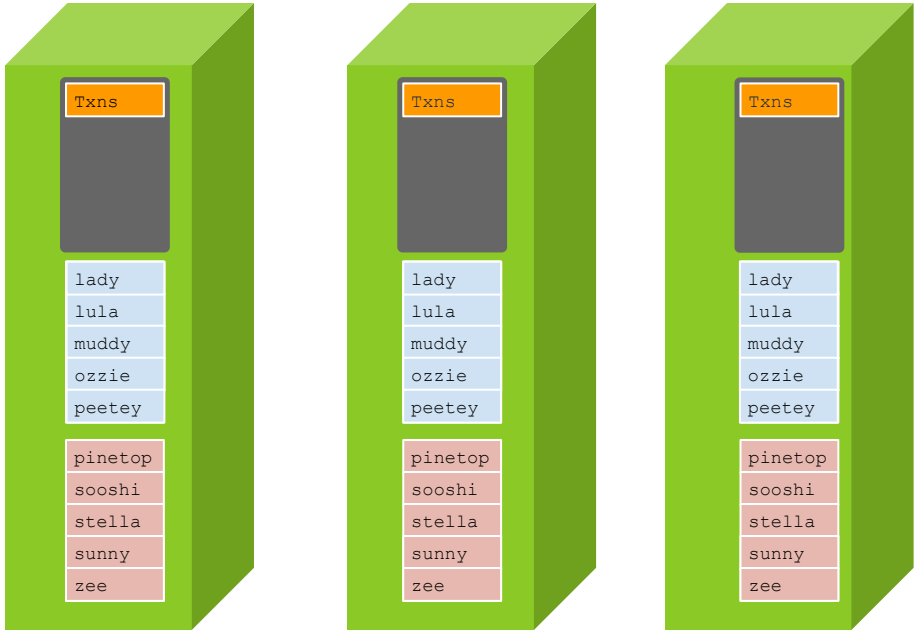
t = 1 RTT

Carousel



t = 1 RTT

Carousel



t = 1 RTT

Agenda

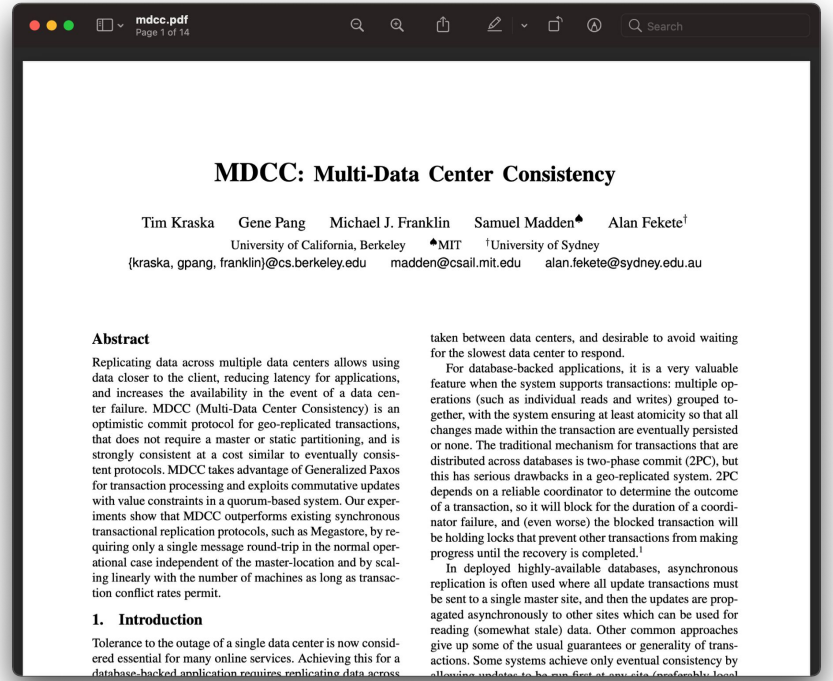
1. Foundations
2. Transactions
3. **Implementations**

I. Spanner/Pipelined Transactions
II. Parallel Commits
III. Replicated Commit
IV. Carousel
V. MDCC
VI. SLOG/OceanVista
VII. TAPIR

MDCC

2013, from Berkeley. The closest one to Parallel Commits. Replicates “options” instead of update values directly, options record the full set of keys written to in the txn. Client can be notified of commit once all options are replicated.

Uses fast paxos to avoid leader hop, generalized paxos to reason about commutative operations. Supports only upto read-committed isolation.



mdcc.pdf
Page 1 of 14

Search

MDCC: Multi-Data Center Consistency

Tim Kraska Gene Pang Michael J. Franklin Samuel Madden[♠] Alan Fekete[†]
University of California, Berkeley [♠]MIT [†]University of Sydney
{kraska, gpang, franklin}@cs.berkeley.edu madden@csail.mit.edu alan.fekete@sydney.edu.au

Abstract

Replicating data across multiple data centers allows using data closer to the client, reducing latency for applications, and increases the availability in the event of a data center failure. MDCC (Multi-Data Center Consistency) is an optimistic commit protocol for geo-replicated transactions, that does not require a master or static partitioning, and is strongly consistent at a cost similar to eventually consistent protocols. MDCC takes advantage of Generalized Paxos for transaction processing and exploits commutative updates with value constraints in a quorum-based system. Our experiments show that MDCC outperforms existing synchronous transactional replication protocols, such as Megastore, by requiring only a single message round-trip in the normal operational case independent of the master-location and by scaling linearly with the number of machines as long as transaction conflict rates permit.

1. Introduction

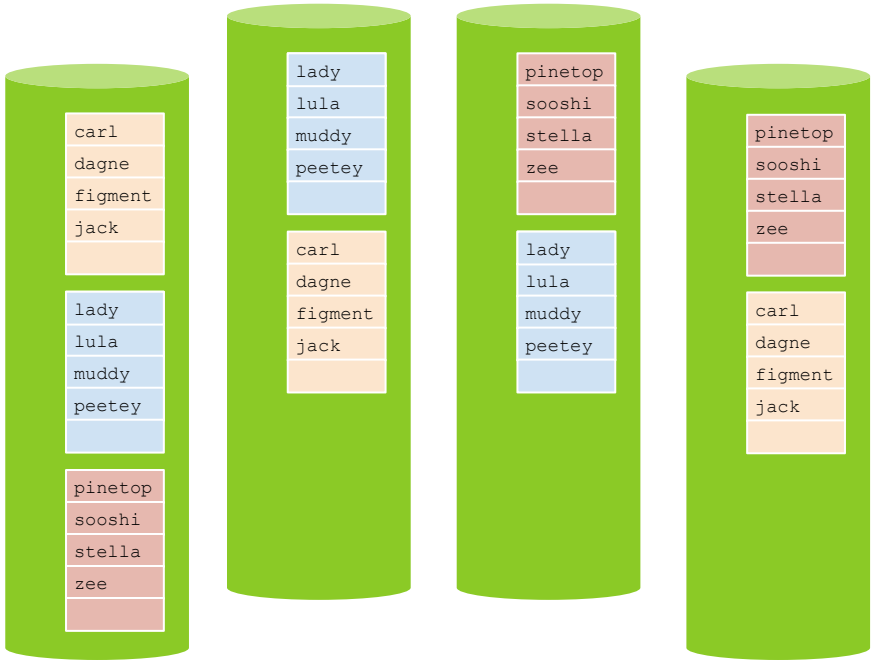
Tolerance to the outage of a single data center is now considered essential for many online services. Achieving this for a database-backed application requires replicating data across

taken between data centers, and desirable to avoid waiting for the slowest data center to respond.

For database-backed applications, it is a very valuable feature when the system supports transactions: multiple operations (such as individual reads and writes) grouped together, with the system ensuring at least atomicity so that all changes made within the transaction are eventually persisted or none. The traditional mechanism for transactions that are distributed across databases is two-phase commit (2PC), but this has serious drawbacks in a geo-replicated system. 2PC depends on a reliable coordinator to determine the outcome of a transaction, so it will block for the duration of a coordinator failure, and (even worse) the blocked transaction will be holding locks that prevent other transactions from making progress until the recovery is completed.[1]

In deployed highly-available databases, asynchronous replication is often used where all update transactions must be sent to a single master site, and then the updates are propagated asynchronously to other sites which can be used for reading (somewhat stale) data. Other common approaches give up some of the usual guarantees or generality of transactions. Some systems achieve only eventual consistency by

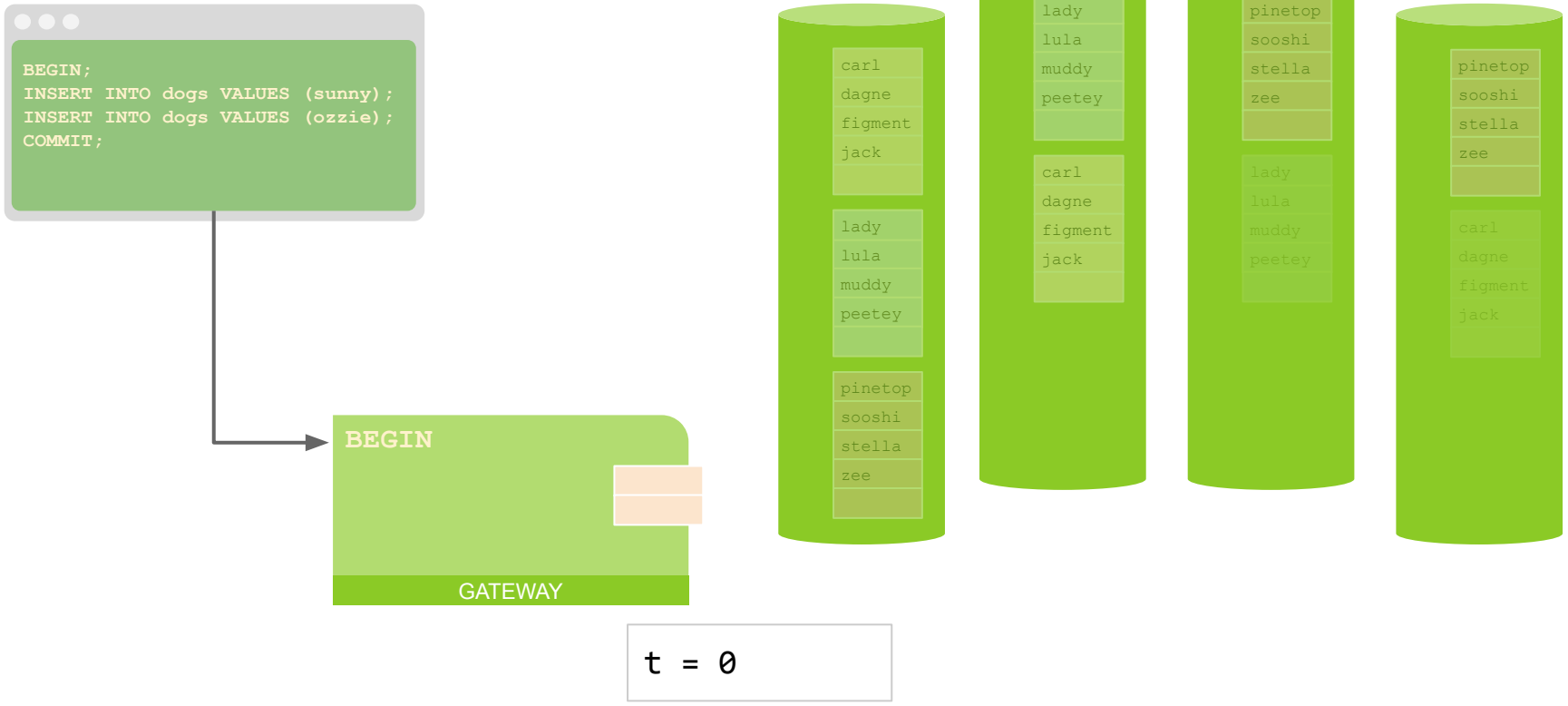
MDCC



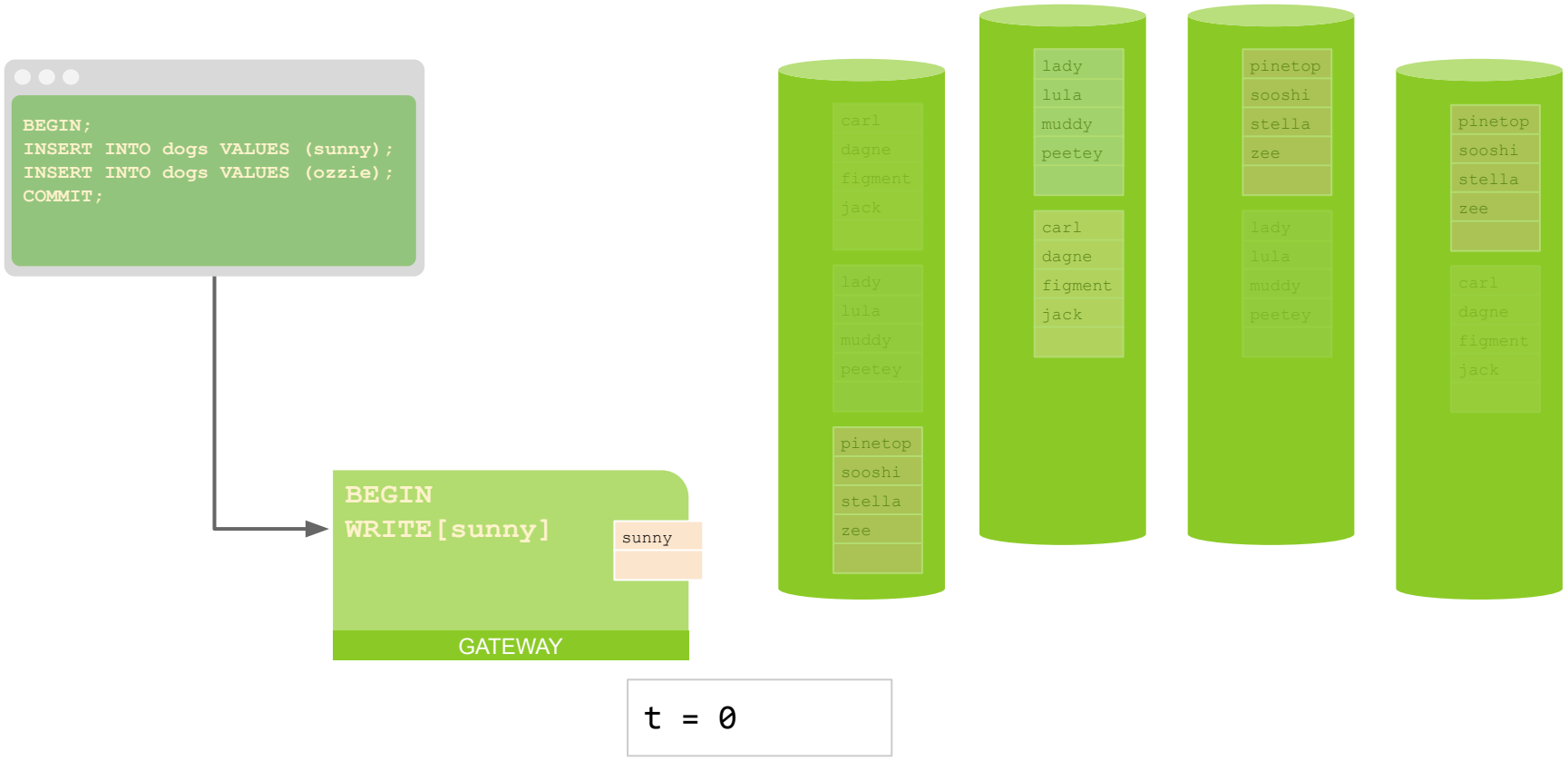
t = 0

NB: with fast paxos we need at least 4 replicas, not shown here

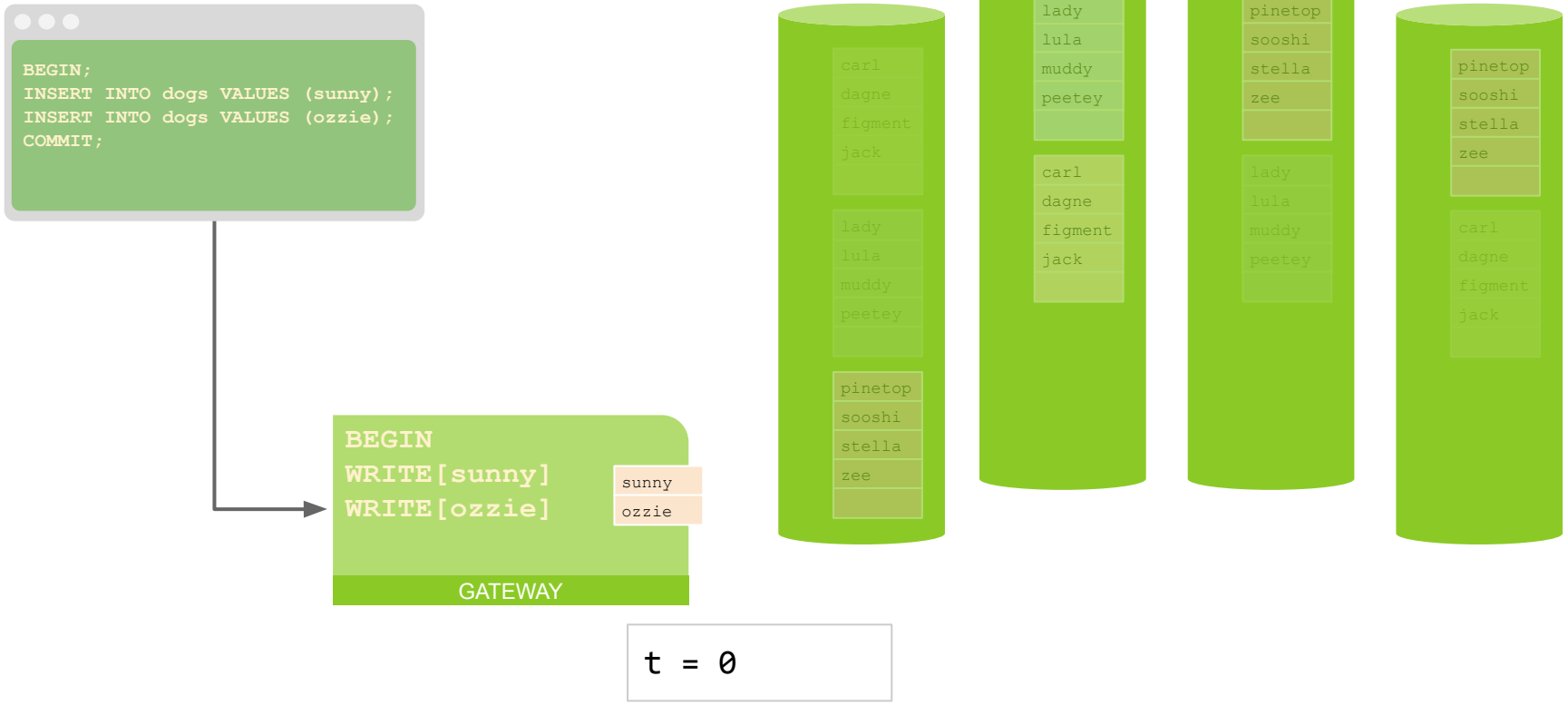
MDCC


BEGIN;
INSERT INTO dogs VALUES (sunny);
INSERT INTO dogs VALUES (ozzie);
COMMIT;
BEGIN
GATEWAY
carl
dagne
figment
jack
lady
lula
muddy
peetey
pinetop
sooshi
stella
zee
t = 0

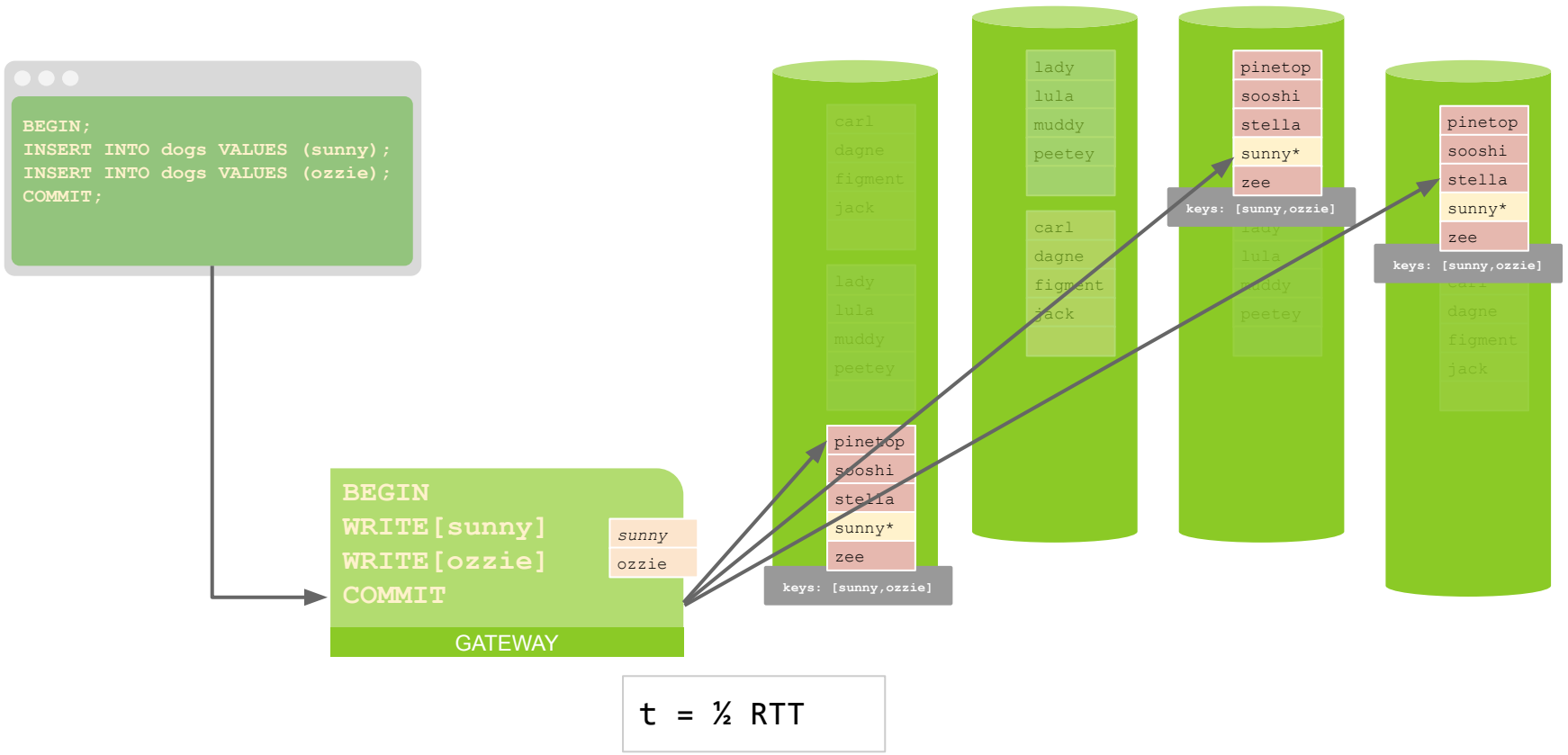
MDCC


BEGIN;
INSERT INTO dogs VALUES (sunny);
INSERT INTO dogs VALUES (ozzie);
COMMIT;
BEGIN
WRITE[sunny]
sunny
GATEWAY
carl
dagne
figment
jack
lady
lula
muddy
peetey
pinetop
sooshi
stella
zee
t = 0

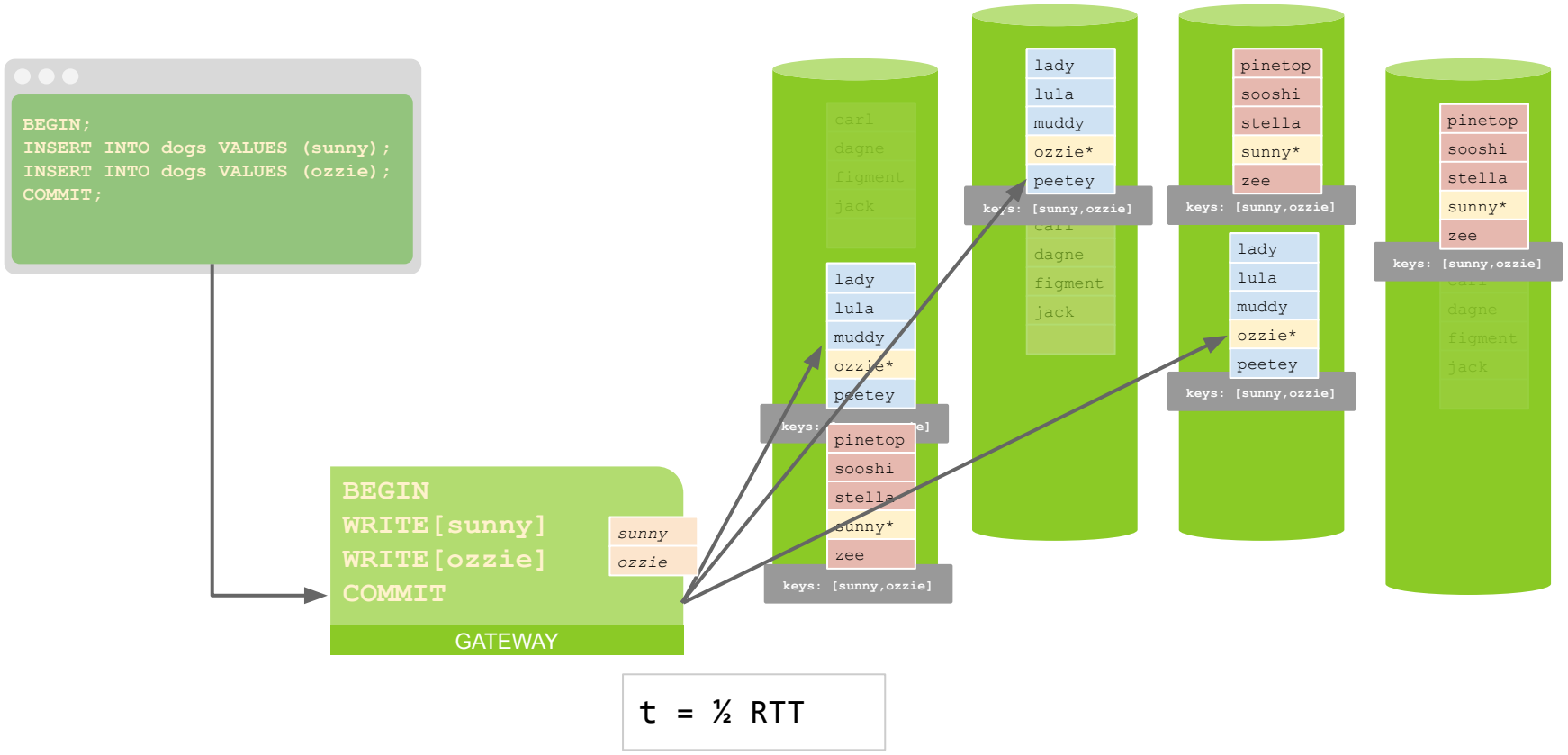
MDCC


BEGIN;
INSERT INTO dogs VALUES (sunny);
INSERT INTO dogs VALUES (ozzie);
COMMIT;
BEGIN
WRITE[sunny]
WRITE[ozzie]
sunny
ozzie
GATEWAY
carl
dagne
figment
jack
lady
lula
muddy
peetey
pinetop
sooshi
stella
zee
t = 0

MDCC

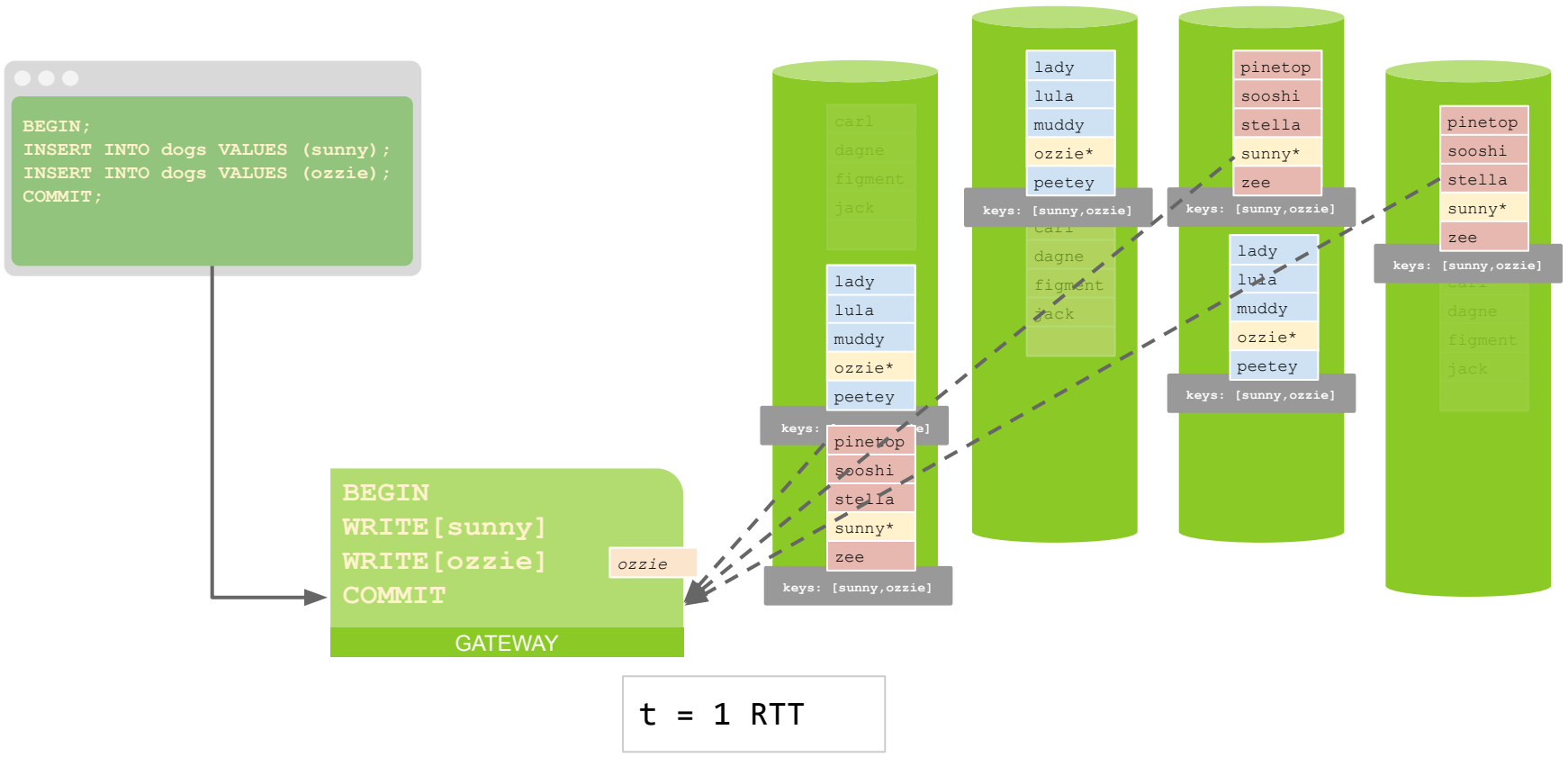

BEGIN;
INSERT INTO dogs VALUES (sunny);
INSERT INTO dogs VALUES (ozzie);
COMMIT;
BEGIN
WRITE[sunny]
WRITE[ozzie]
COMMIT
sunny
ozzie
GATEWAY
carl
dagne
figment
jack
lady
lula
muddy
peetey
pinetop
sooshi
stella
sunny*
zee
keys: [sunny,ozzie]
t = ½ RTT

MDCC

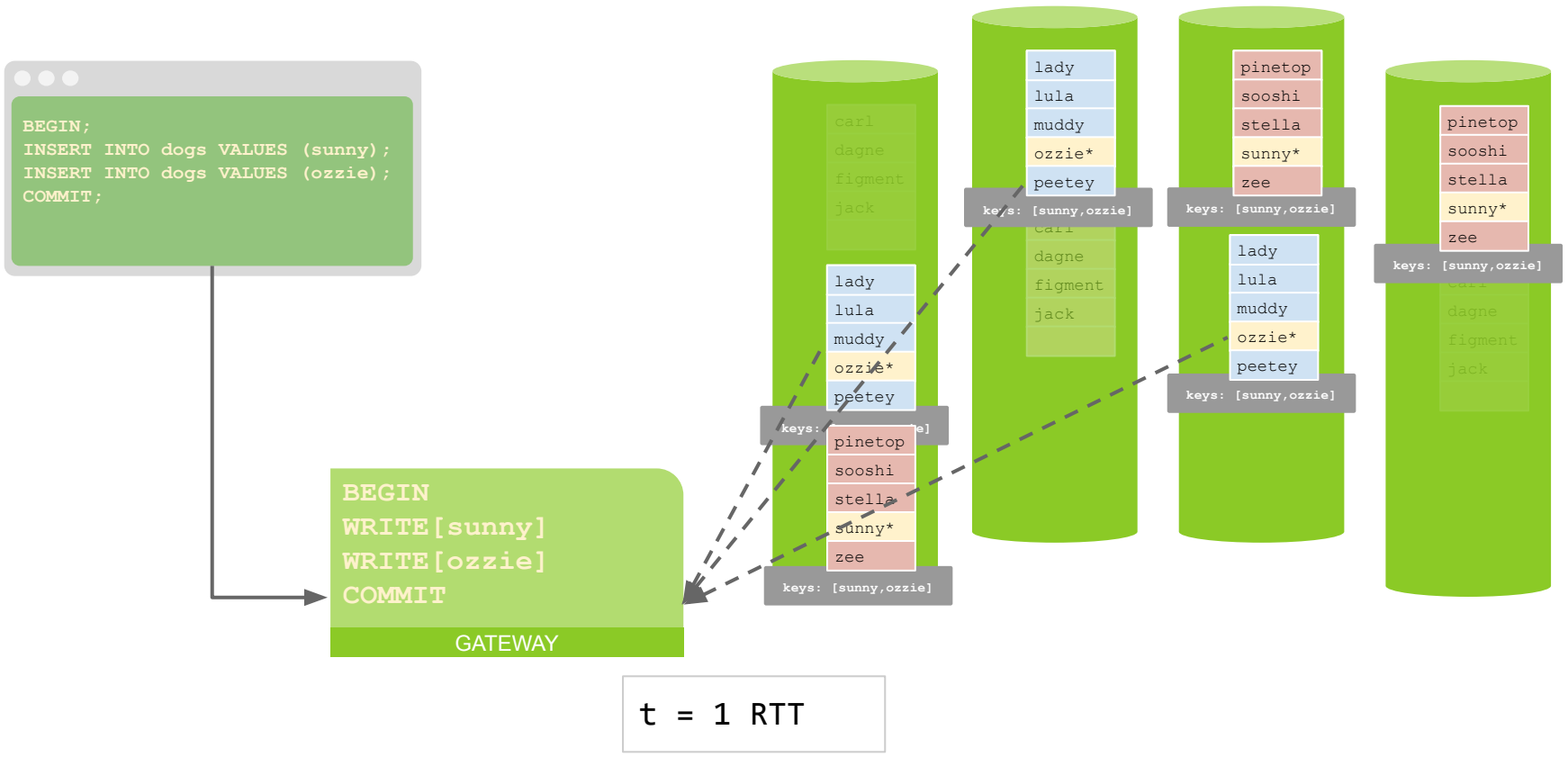


$t = \frac{1}{2} RTT$

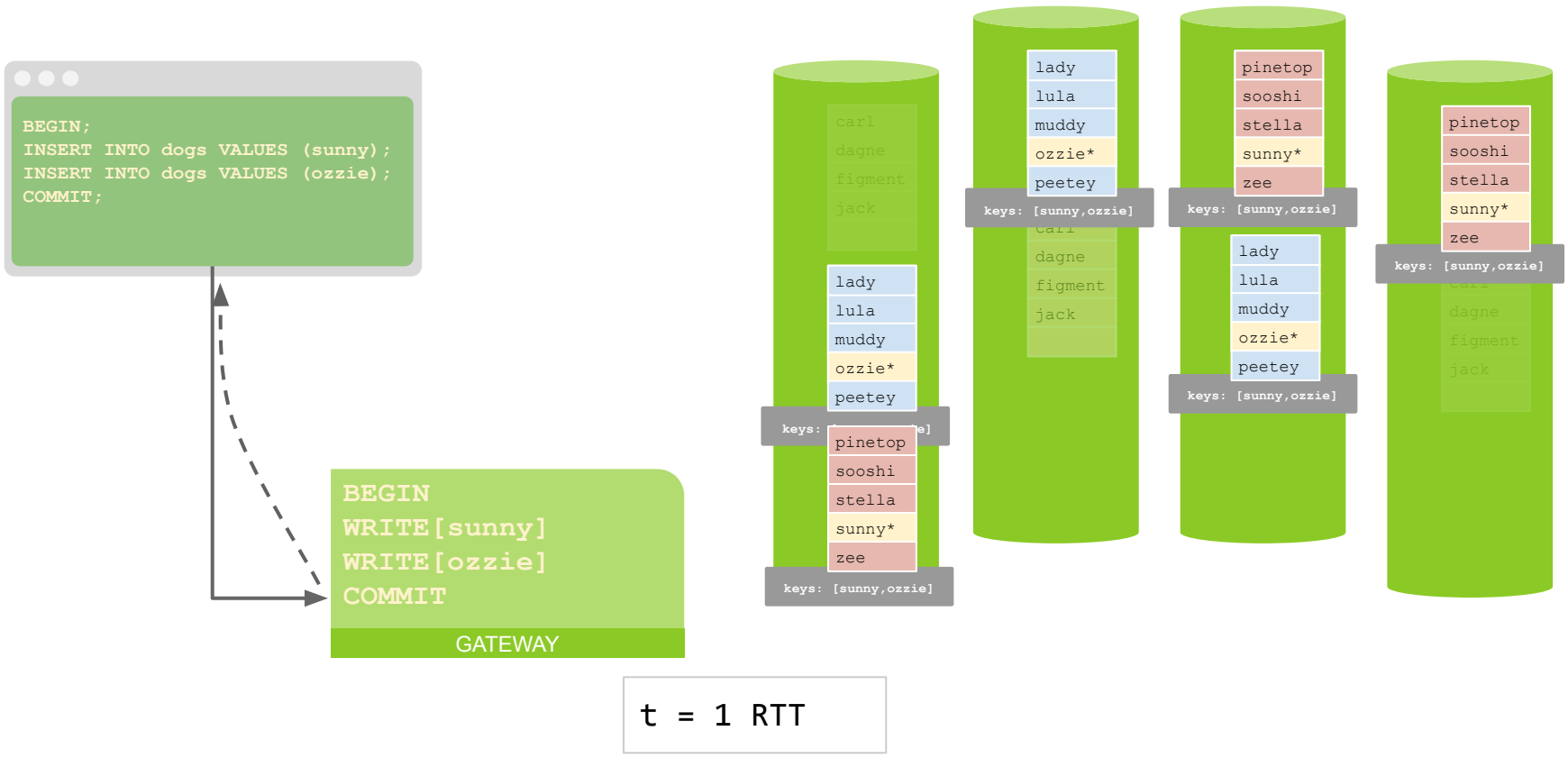
MDCC


BEGIN;
INSERT INTO dogs VALUES (sunny);
INSERT INTO dogs VALUES (ozzie);
COMMIT;
BEGIN
WRITE[sunny]
WRITE[ozzie]
COMMIT
ozzie
GATEWAY
lady
lula
muddy
ozzie*
peetey
pinetop
sooshi
stella
sunny*
zee
keys: [sunny,ozzie]
t = 1 RTT

MDCC

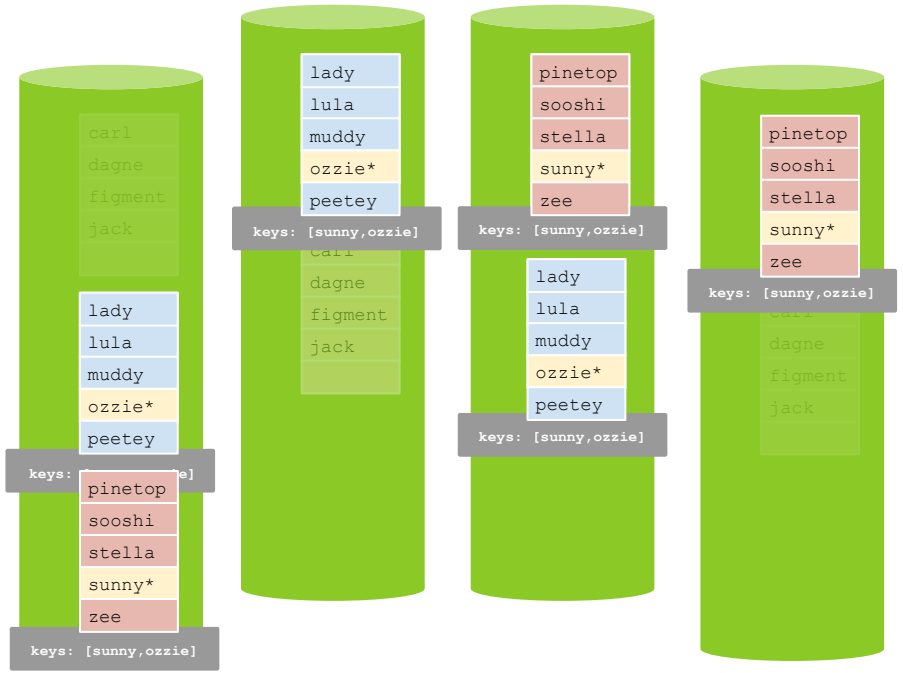

BEGIN;
INSERT INTO dogs VALUES (sunny);
INSERT INTO dogs VALUES (ozzie);
COMMIT;
BEGIN
WRITE[sunny]
WRITE[ozzie]
COMMIT
GATEWAY
carl
dagne
figment
jack
lady
lula
muddy
ozzie*
peetey
pinetop
sooshi
stella
sunny*
zee
keys: [sunny,ozzie]
t = 1 RTT

MDCC


BEGIN;
INSERT INTO dogs VALUES (sunny);
INSERT INTO dogs VALUES (ozzie);
COMMIT;
BEGIN
WRITE[sunny]
WRITE[ozzie]
COMMIT
GATEWAY
carl
dagne
figment
jack
lady
lula
muddy
ozzie*
peetey
pinetop
sooshi
stella
sunny*
zee
keys: [sunny,ozzie]

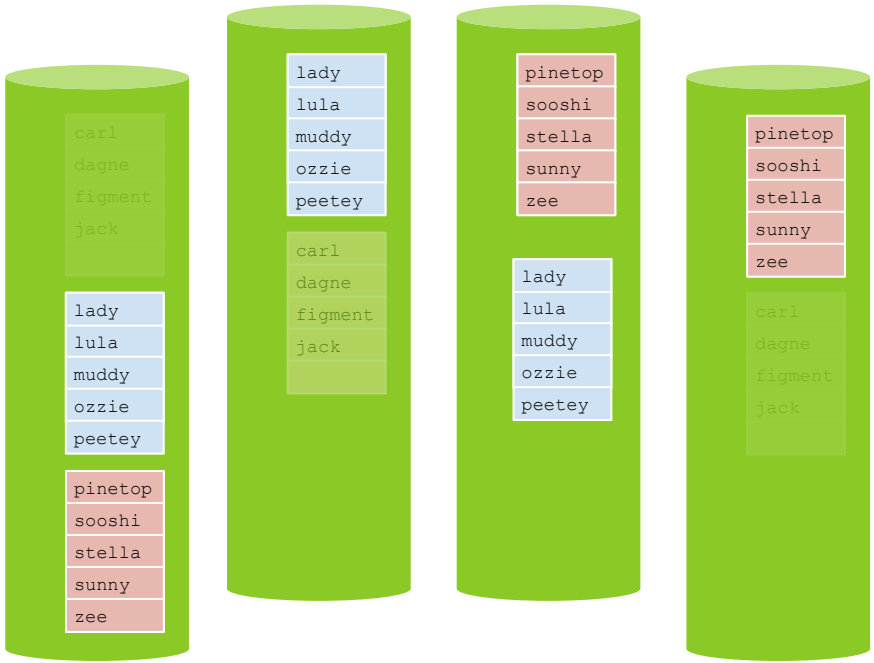
t = 1 RTT

MDCC


carl
dagne
figment
jack
lady
lula
muddy
ozzie*
peetey
keys: [sunny,ozzie]
pinetop
sooshi
stella
sunny*
zee
keys: [sunny,ozzie]
lady
lula
muddy
ozzie*
peetey
keys: [sunny,ozzie]
pinetop
sooshi
stella
sunny*
zee
keys: [sunny,ozzie]

t = 1 RTT

MDCC


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

t = 1 RTT

Agenda

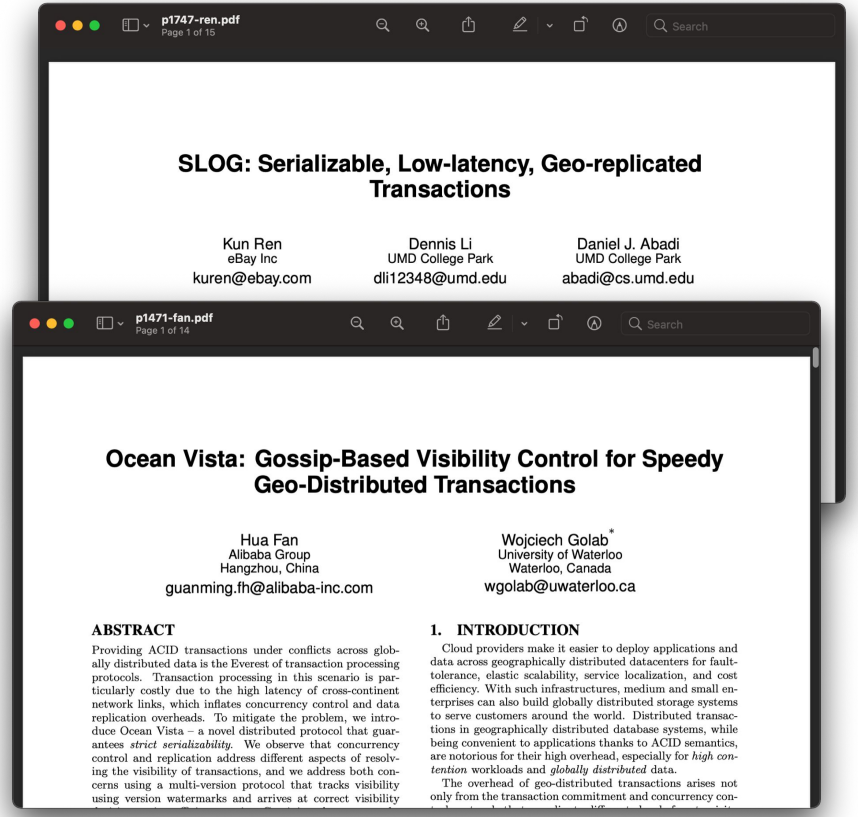
1. Foundations
2. Transactions
3. **Implementations**

I. Spanner/Pipelined Transactions
II. Parallel Commits
III. Replicated Commit
IV. Carousel
V. MDCC
VI. SLOG/OceanVista
VII. TAPIR

SLOG/OceanVista

2019, from University of Maryland. Fully deterministic transactions, with read and write sets pre-declared. Lets it avoid 2PC; replicates the txn code itself.

Ditto for OceanVista. 2019, from University of Waterloo. Though SLOG provides local latencies for single-region transactions.



p1747-ren.pdf
Page 1 of 15

SLOG: Serializable, Low-latency, Geo-replicated Transactions

Kun Ren
eBay Inc
kuren@ebay.com

Dennis Li
UMD College Park
dli12348@umd.edu

Daniel J. Abadi
UMD College Park
abadi@cs.umd.edu

p1471-fan.pdf
Page 1 of 14

Ocean Vista: Gossip-Based Visibility Control for Speedy Geo-Distributed Transactions

Hua Fan
Alibaba Group
Hangzhou, China
guanming.fh@alibaba-inc.com

Wojciech Golab*
University of Waterloo
Waterloo, Canada
wgolab@uwaterloo.ca

ABSTRACT

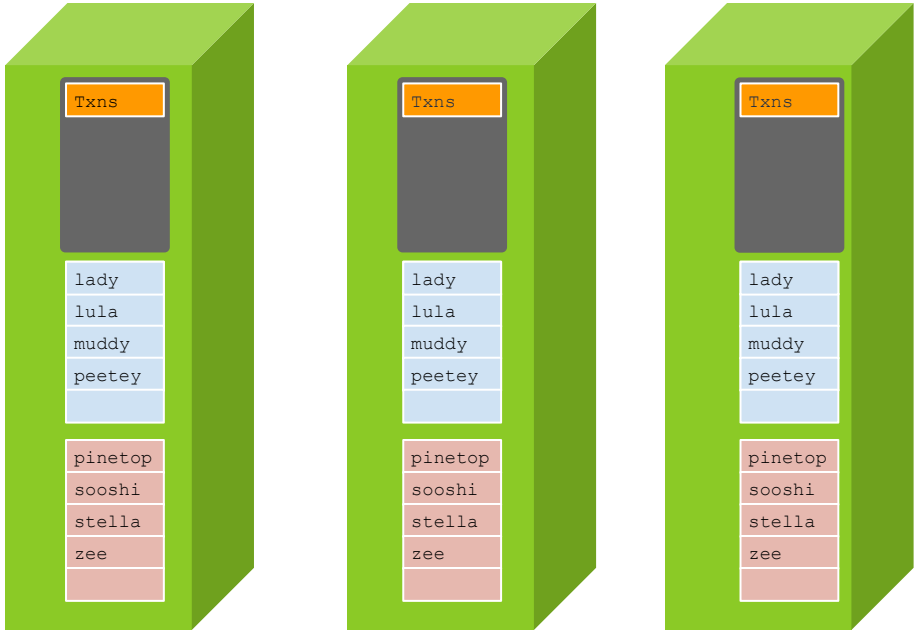
Providing ACID transactions under conflicts across globally distributed data is the Everest of transaction processing protocols. Transaction processing in this scenario is particularly costly due to the high latency of cross-continent network links, which inflates concurrency control and data replication overheads. To mitigate the problem, we introduce Ocean Vista – a novel distributed protocol that guarantees *strict serializability*. We observe that concurrency control and replication address different aspects of resolving the visibility of transactions, and we address both concerns using a multi-version protocol that tracks visibility using version watermarks and arrives at correct visibility

1. INTRODUCTION

Cloud providers make it easier to deploy applications and data across geographically distributed datacenters for fault-tolerance, elastic scalability, service localization, and cost efficiency. With such infrastructures, medium and small enterprises can also build globally distributed storage systems to serve customers around the world. Distributed transactions in geographically distributed database systems, while being convenient to applications thanks to ACID semantics, are notorious for their high overhead, especially for *high contention* workloads and *globally distributed* data.

The overhead of geo-distributed transactions arises not only from the transaction commitment and concurrency con-

OceanVista

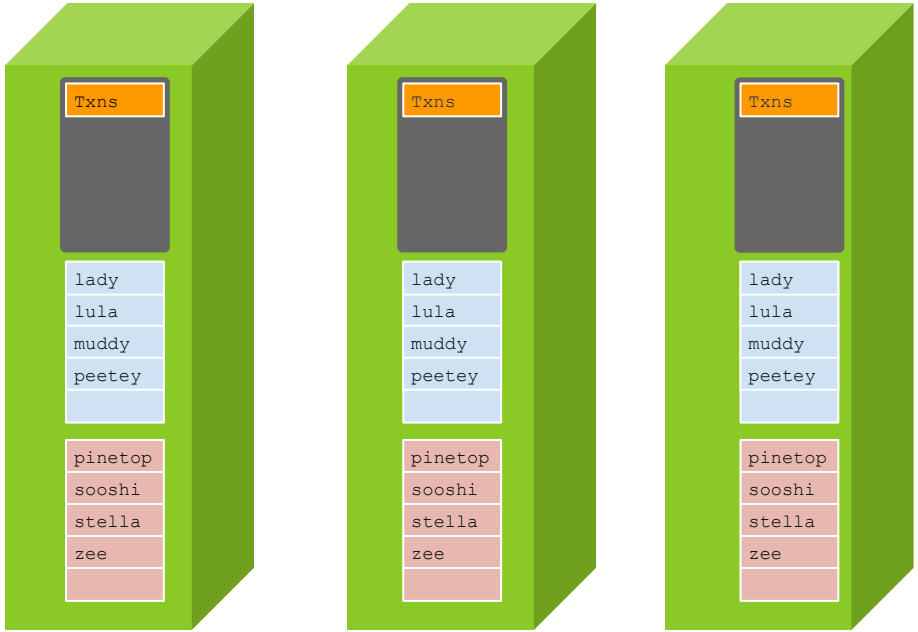


t = 0

OceanVista

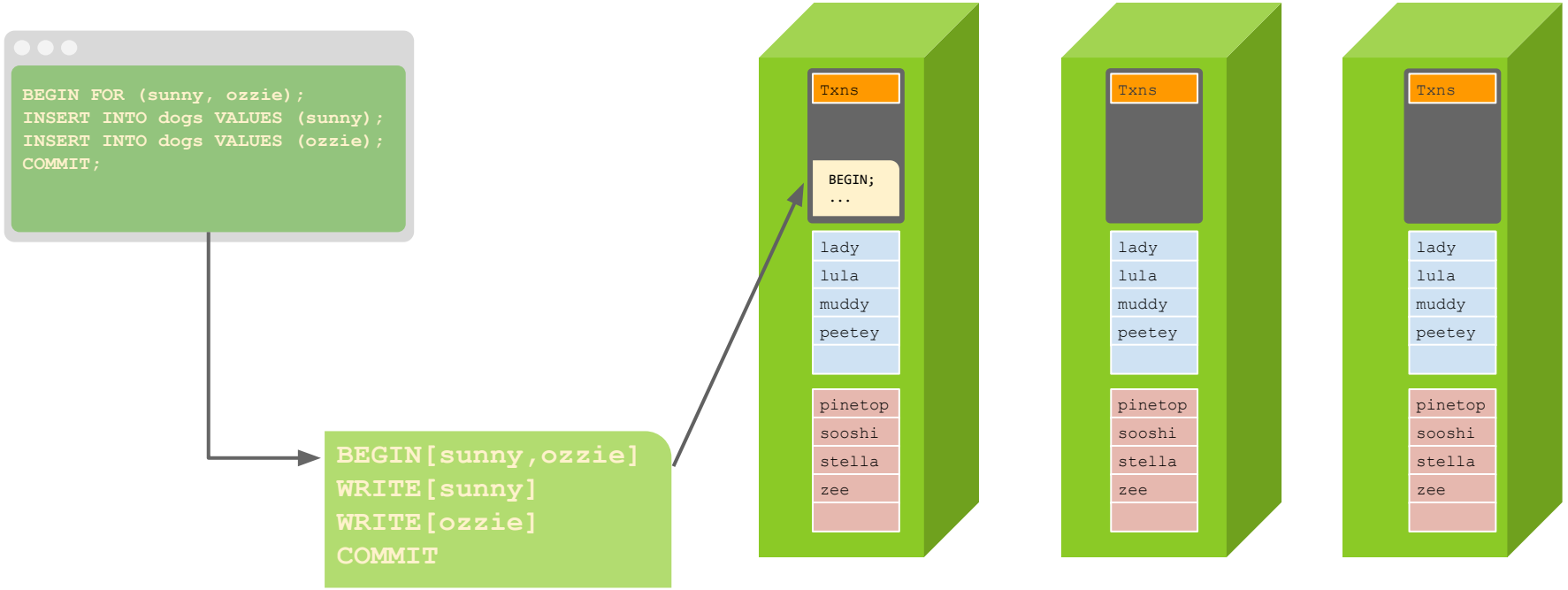
```
BEGIN FOR (sunny, ozzie);
INSERT INTO dogs VALUES (sunny);
INSERT INTO dogs VALUES (ozzie);
COMMIT;
```

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



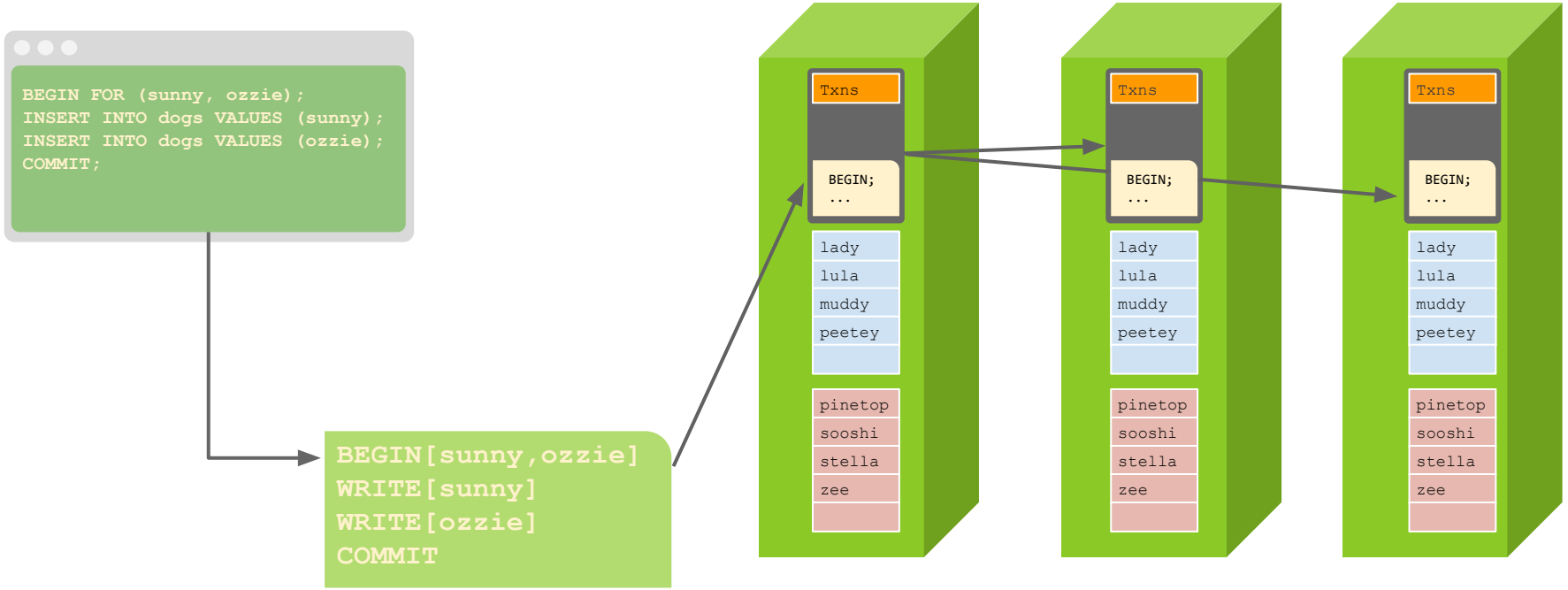
t = 0

OceanVista



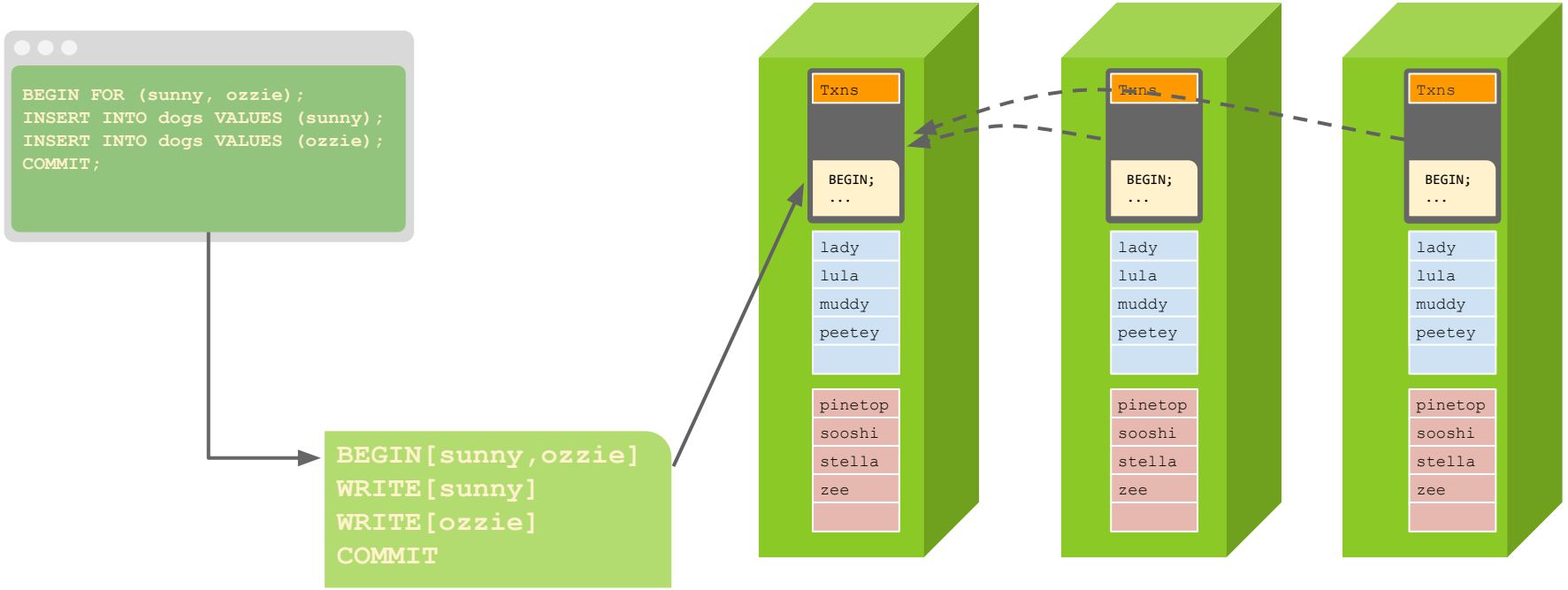
t = 0

OceanVista



t = ½ WAN

OceanVista

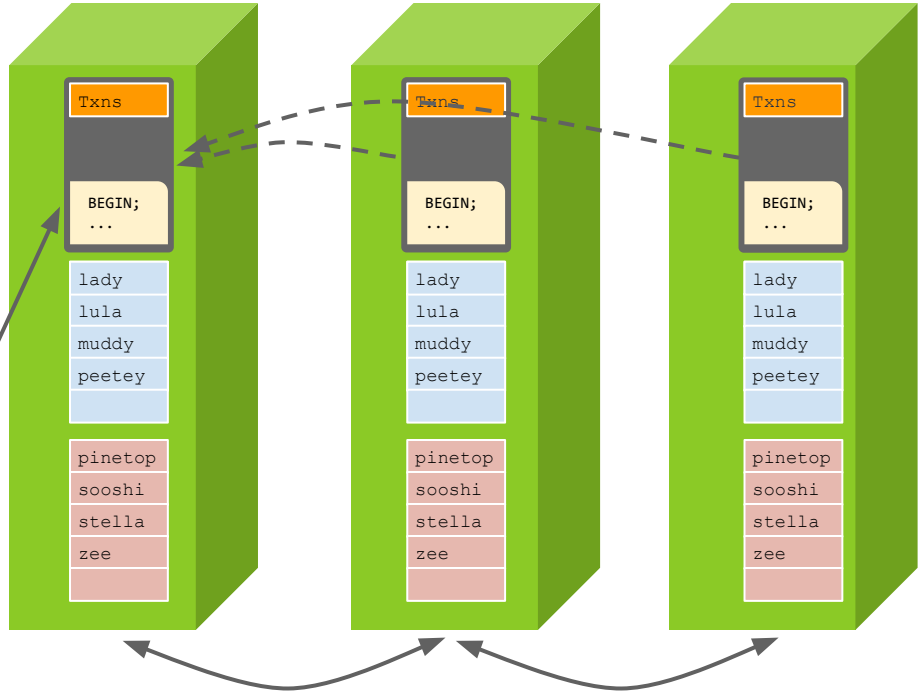


t = 1 WAN

OceanVista

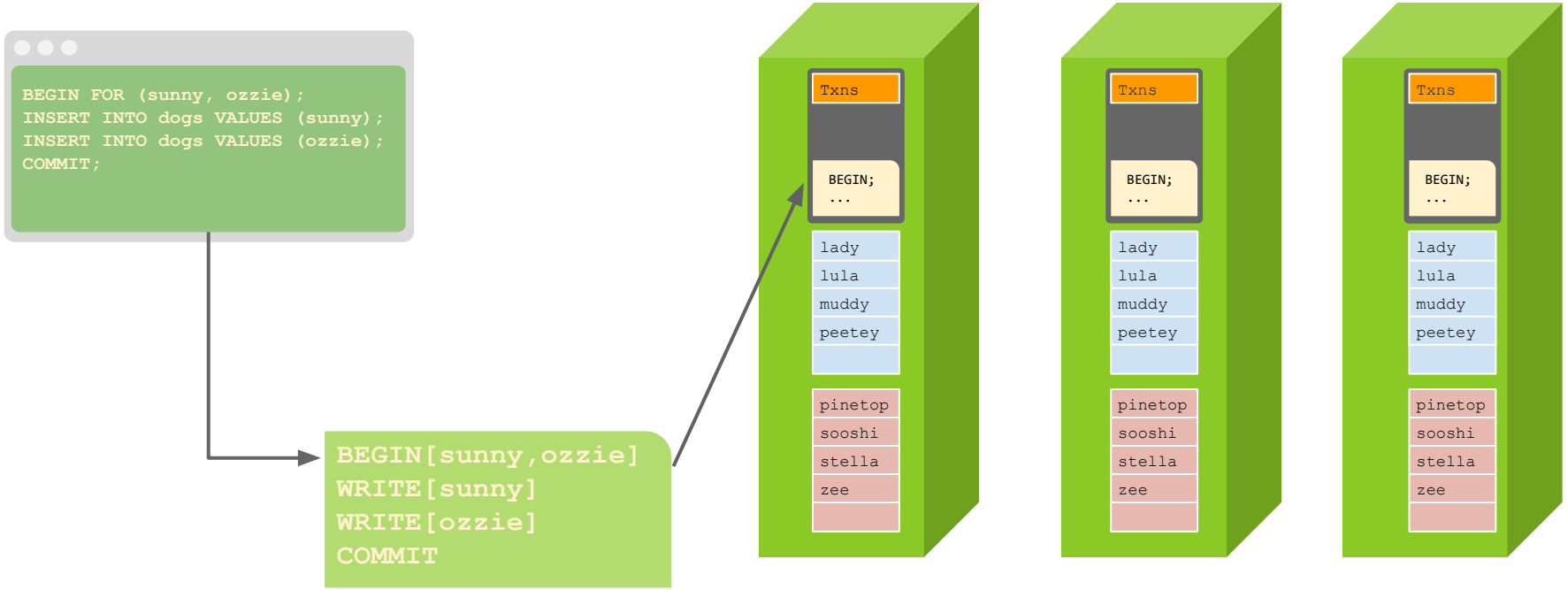
```
BEGIN FOR (sunny, ozzie);
INSERT INTO dogs VALUES (sunny);
INSERT INTO dogs VALUES (ozzie);
COMMIT;
```

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



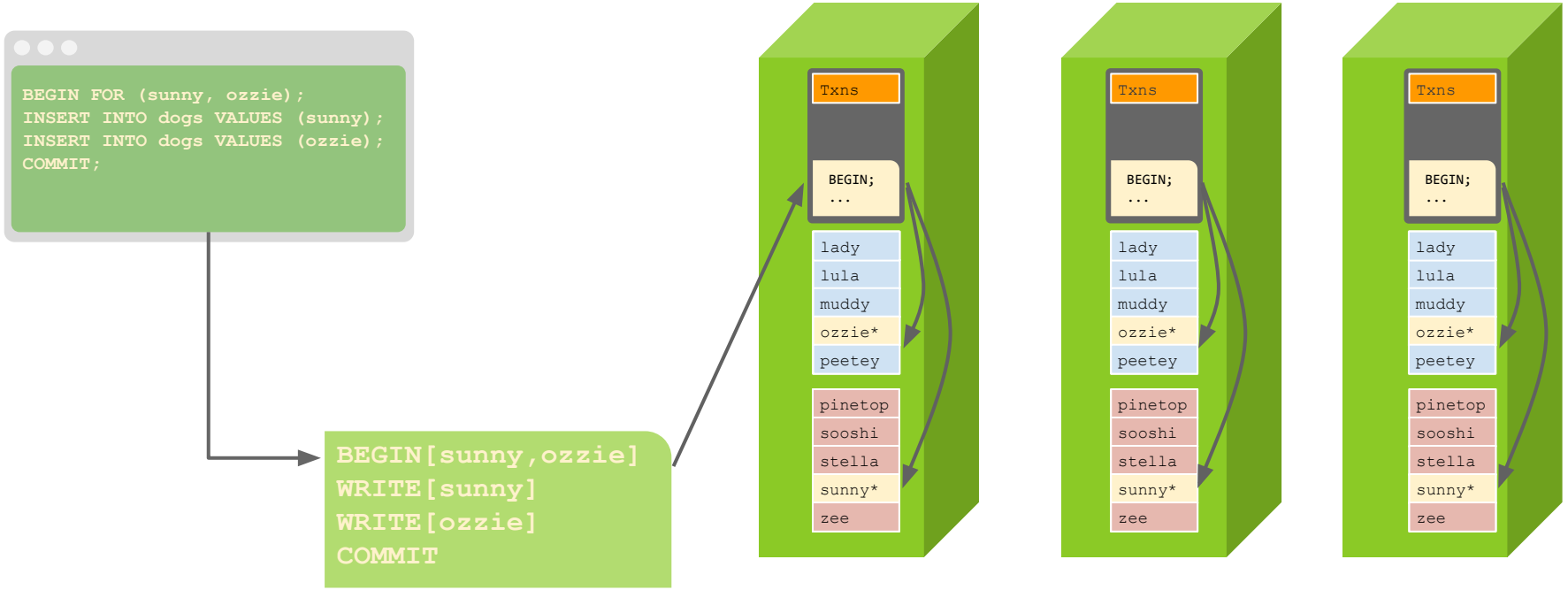
t = 1 WAN

OceanVista



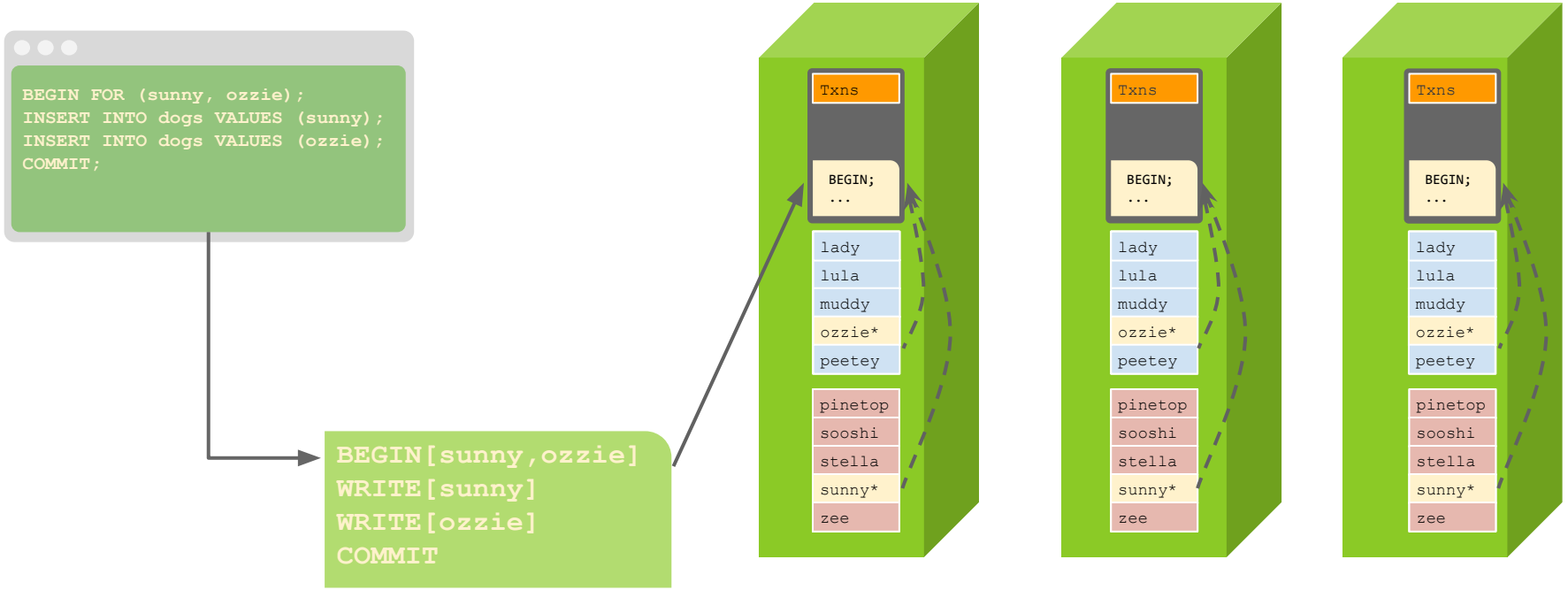
t = 1 WAN

OceanVista



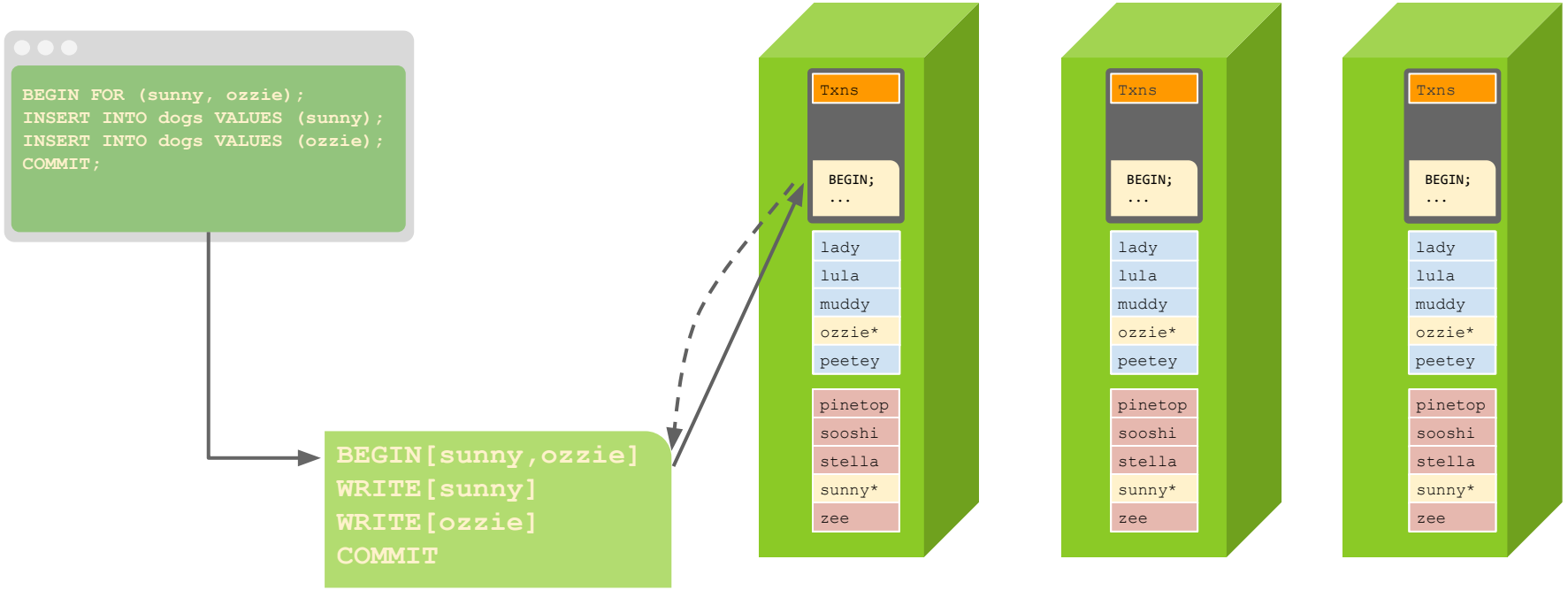
t = 1 WAN + ½ LAN

OceanVista



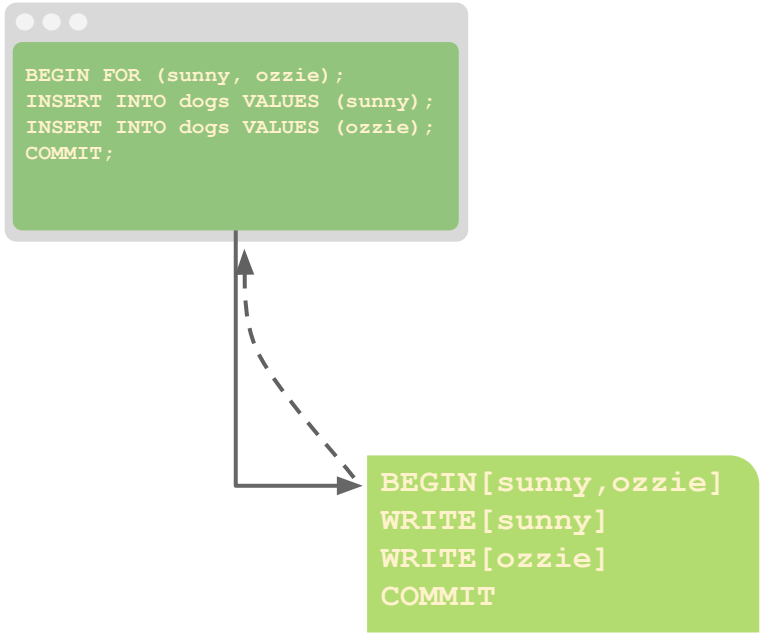
t = 1 WAN + 1 LAN

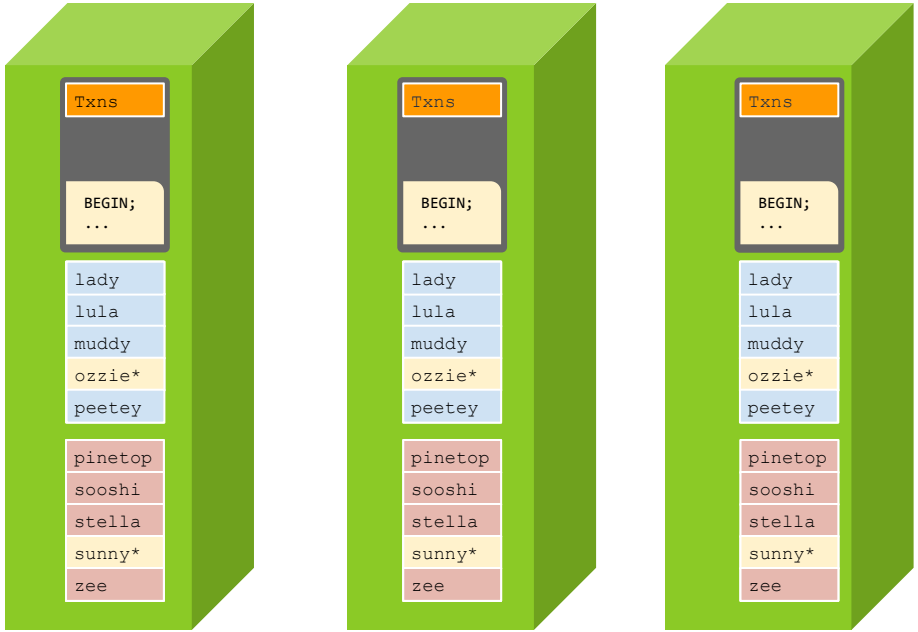
OceanVista



t = 1 WAN + 1 LAN

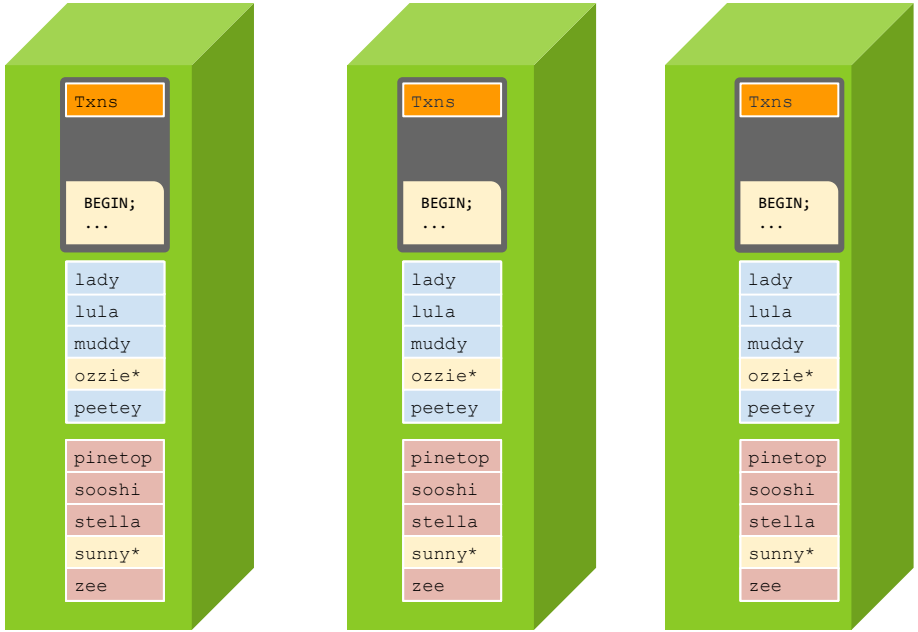
OceanVista





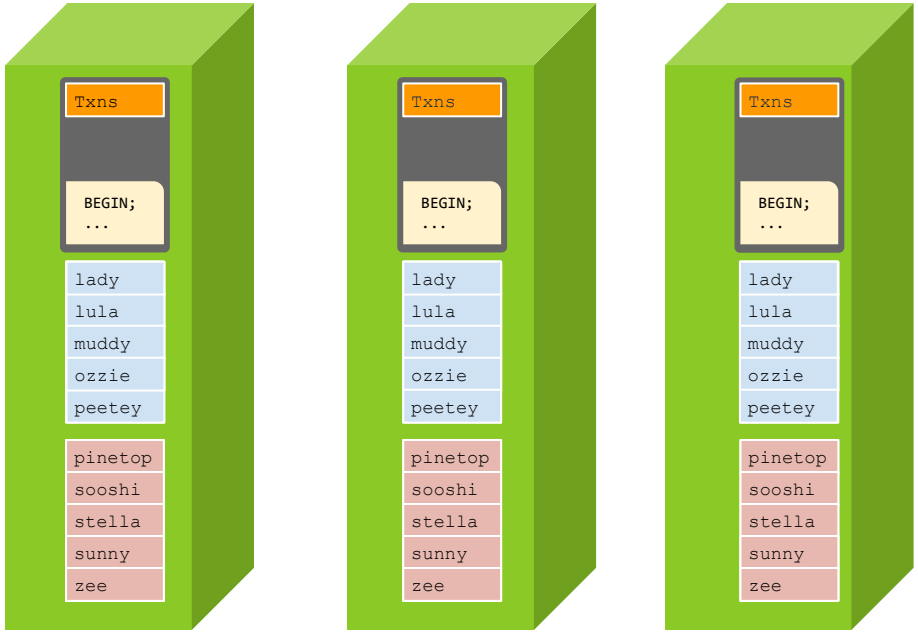
t = 1 WAN + 1 LAN

OceanVista



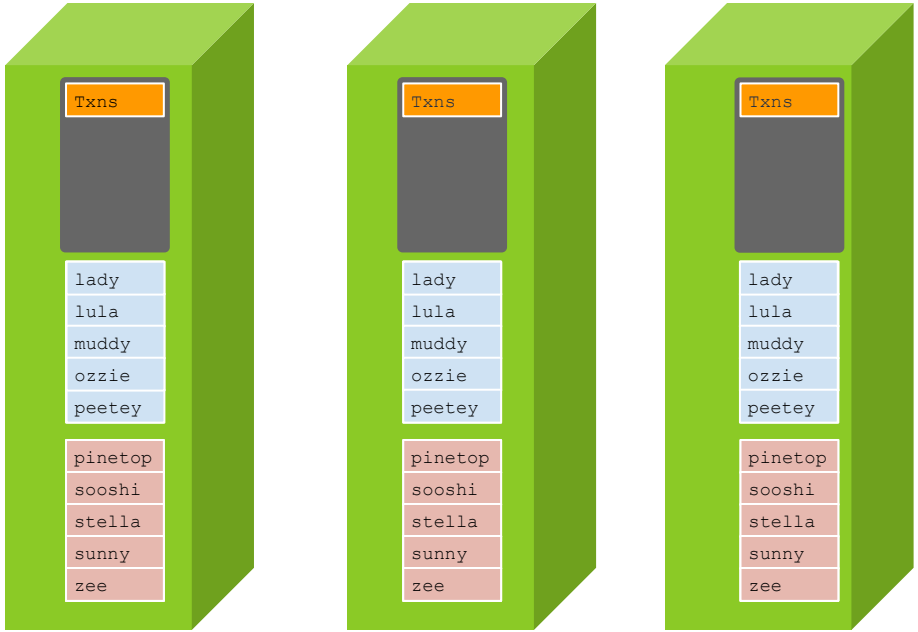

```
t = 1 WAN + 1 LAN
```

OceanVista



t = 1 WAN + 1 LAN

OceanVista



t = 1 WAN + 1 LAN

Agenda

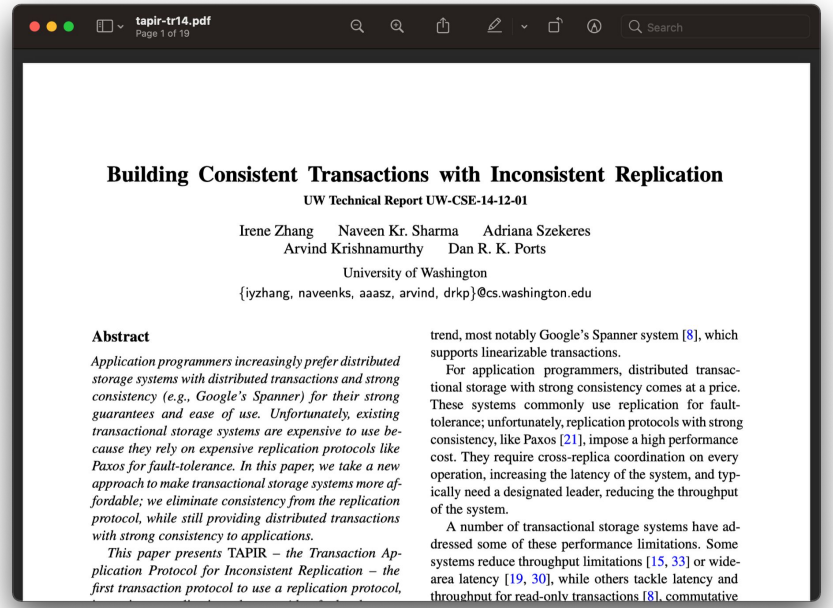
1. Foundations
2. Transactions
3. **Implementations**

I. Spanner/Pipelined Transactions
II. Parallel Commits
III. Replicated Commit
IV. Carousel
V. MDCC
VI. SLOG/OceanVista
VII. TAPIR

TAPIR

2014, from University of Washington. Observes that 2PC and Consensus-based replication both order operations (each contributing 1RTT). Proposes unordered replication, and builds transaction protocol on top of it.

Primitive: replicas all contain a set of operations, but in no particular order. Two sets are guaranteed to overlap.



tapir-tr14.pdf
Page 1 of 19

Building Consistent Transactions with Inconsistent Replication

UW Technical Report UW-CSE-14-12-01

Irene Zhang Naveen Kr. Sharma Adriana Szekeres
Arvind Krishnamurthy Dan R. K. Ports

University of Washington

{iyzhang, naveenks, aaasz, arvind, drkp}@cs.washington.edu

Abstract

Application programmers increasingly prefer distributed storage systems with distributed transactions and strong consistency (e.g., Google's Spanner) for their strong guarantees and ease of use. Unfortunately, existing transactional storage systems are expensive to use because they rely on expensive replication protocols like Paxos for fault-tolerance. In this paper, we take a new approach to make transactional storage systems more affordable; we eliminate consistency from the replication protocol, while still providing distributed transactions with strong consistency to applications.

This paper presents TAPIR *– the Transaction Application Protocol for Inconsistent Replication – the first transaction protocol to use a replication protocol,*

trend, most notably Google's Spanner system [8], which supports linearizable transactions.

For application programmers, distributed transactional storage with strong consistency comes at a price. These systems commonly use replication for fault-tolerance; unfortunately, replication protocols with strong consistency, like Paxos [21], impose a high performance cost. They require cross-replica coordination on every operation, increasing the latency of the system, and typically need a designated leader, reducing the throughput of the system.

A number of transactional storage systems have addressed some of these performance limitations. Some systems reduce throughput limitations [15, 33] or wide-area latency [19, 30], while others tackle latency and throughput for read-only transactions [8], commutative

Agenda

1. Foundations
2. Transactions
3. **Implementations**

I.	Spanner/Pipelined Transactions	**2 WAN RTT**
II.	Parallel Commits	**1 WAN RTT**
III.	Replicated Commit	**1 WAN + LAN RTT**
IV.	Carousel	**1 WAN RTT**
V.	MDCC	**1 WAN RTT**
VI.	SLOG/OceanVista	**1 WAN + LAN RTT**
VII.	TAPIR	**1 WAN RTT**

Questions?

irfansharif.io
@irfansharifm