Cockroach Labs

What can go wrong with data... will!

\\or ACID transactions & database consistency

Today...

The Session today is

BEGINNER to INTERMEDIATE

Jim Walker
Principal Product Evangelist
@jaymce

Stages of an (OLTP) database transaction

1. Begin the transaction
2. Execute the transaction
3. Commit or rollback the transaction

What could possibly go wrong?

1. Begin the transaction
2. Execute the transaction
3. Commit or rollback the transaction

- Only half your transaction is successful
- Two transactions try to happen at the same time
- One transaction has to wait for another
- A rollback cause another rollback

ACID

Cockroach Labs Confidential - Internal Use ONLY

Atomicity

Consistency

Isolation

Durability

1983

Principles of Transaction-Oriented Database Recovery

THEO HAERDER

Fachbereich Informatik, University of Kaiserslautern, West Germany

ANDREAS REUTER[1]

IBM Research Laboratory, San Jose, California 95193

In this paper, a terminological framework is provided for describing different transaction-oriented recovery schemes for database systems in a conceptual rather than an implementation-dependent way. By introducing the terms materialized database, propagation strategy, and checkpoint, we obtain a means for classifying arbitrary implementations from a unified viewpoint. This is complemented by a classification scheme for logging techniques, which are precisely defined by using the other terms. It is shown that these criteria are related to all relevant questions such as speed and scope of recovery and amount of redundant information required. The primary purpose of this paper, however, is to establish an adequate and precise terminology for a topic in which the confusion of concepts and implementational aspects still imposes a lot of problems.

Categories and Subject Descriptors: D.4.5 [**Operating Systems**]: Reliability—*fault tolerance*; H.1.0 [**Models and Principles**]: General; H.2.2 [**Database Management**]: Physical Design—*recovery and restart*; H.2.4 [**Database Management**]: Systems—*transaction processing*; H.2.7 [**Database Management**]: Database Administration—*logging and recovery*

General Terms: Databases, Fault Tolerance, Transactions

INTRODUCTION

Database technology has seen tremendous progress during the past ten years. Concepts and facilities that evolved in the single-user batch environments of the early days have given rise to efficient multiuser database systems with user-friendly interfaces, distributed data management, etc. From a scientific viewpoint, database systems today are established as a mature discipline with well-approved methods and technology. The methods and technology of such a discipline should be well represented in the literature by systematic surveys of the field. There are, in fact, a number of recent publications that attempt to summarize what is known about different aspects of database management [e.g., Astrahan et al. 1981; Stonebraker 1980; Gray et al. 1981; Kohler 1981; Bernstein and Goodman 1981; Codd 1982]. These papers fall into two categories: (1) descriptions of innovative prototype systems and (2) thorough analyses of special problems and their solutions, based on a clear methodological and terminological framework. We are con-

[1] Permanent address: Fachbereich Informatik, University of Kaiserslautern, West Germany.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1983 ACM 0360-0300/83/1200-0287 $00.75

Computing Surveys, Vol. 15, No. 4, December 1983

A set of properties of database transactions to guarantee validity in event of errors/power failures

Theo Haerder
Andreas Reuter
(Jim Gray)

Atomicity

a transaction completes in total.. Could be simple or complex, it is an atomic unit and will complete fully

Consistency

Isolation

Durability

Atomicity

a transaction completes in total.. Could be simple or complex, it is an atomic unit and will complete fully

Consistency

any transaction will not violate the integrity of the database. Each will leave it in a valid state

Isolation

Durability

Atomicity

a transaction completes in total.. Could be simple or complex, it is an atomic unit and will complete fully

Consistency

any transaction will not violate the integrity of the database. Each will leave it in a valid state

Isolation

ensures that any concurrent transactions will leave the database in the same state as if they were executed sequentially.

Durability

Atomicity

a transaction completes in total.. Could be simple or complex, it is an atomic unit and will complete fully

Consistency

any transaction will not violate the integrity of the database. Each will leave it in a valid state

Isolation

ensures that any concurrent transactions will leave the database in the same state as if they were executed sequentially.

Durability

ensure that once committed, the transaction remains committed no matter the circumstance or failure, etc.

Atomicity

a transaction completes in total.. Could be simple or complex, it is an atomic unit and will complete fully

Consistency

any transaction will not violate the integrity of the database. Each will leave it in a valid state

Isolation

ensures that any concurrent transactions will leave the database in the same state as if they were executed sequentially.

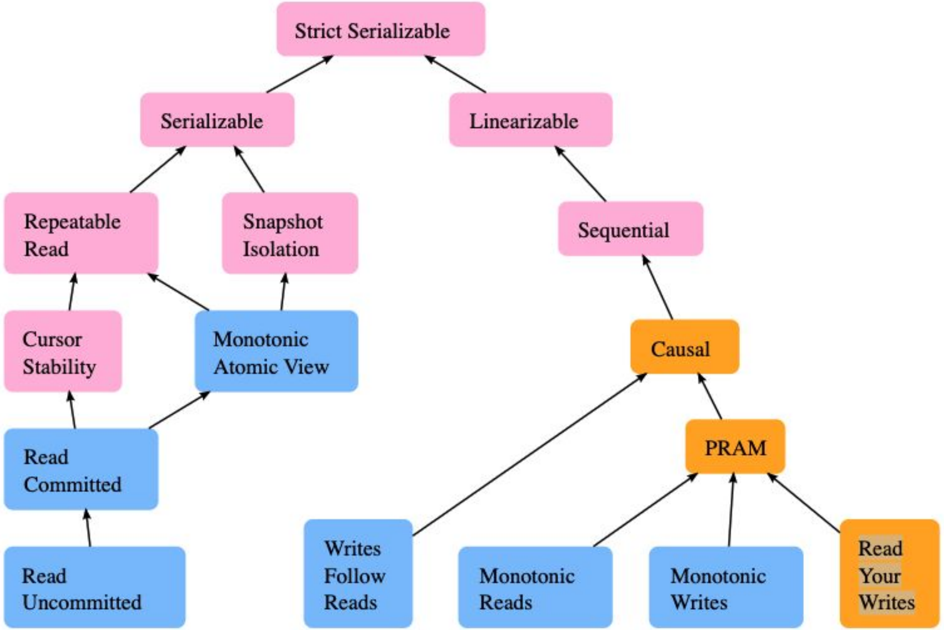
Durability

ensure that once committed, the transaction remains committed no matter the circumstance or failure, etc.

Database isolation levels

The isolation level of a database is a setting.

There are default levels set for each database as well



https://jepsen.io/consistency

Weak isolation levels result in common issues

Dirty Read

a second transaction reads uncommitted data

Non Repeatable Read

two transactions return different answers when reading the same row twice.

Write Skew

two transactions overlap and one reads data that another is writing both can succeed.

Phantom Read

row matches the search criteria but is not initially seen.

Default isolation levels for a few databases

Default Isolation	Isolation Level	P0 Dirty Write	P1 Dirty Read	P4C Cursor Lost Update	P4 Lost Updates	P2 Fuzzy Read	P3 Phantom Read	A5a Read Skew	A5b Write Skew
MongoDB	Read Uncommitted	Not Possible	Possible	Possible	Possible	Possible	Possible	Possible	Possible
Oracle PostgreSQL SQL Server	Read Committed	Not Possible	Not Possible	Possible	Possible	Possible	Possible	Possible	Possible
	Cursor Stability	Not Possible	Not Possible	Not Possible	Sometimes Possible	Sometimes Possible	Sometimes Possible	Possible	Possible
MySQL	Repeatable Read	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Possible	Not Possible	Possible
	Snapshot	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Sometimes Possible	Not Possible	Possible
CockroachDB	Serializable	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible

Transactions - Update account balance

Two simultaneous transactions:

- The account starts at $0
- Read the account balance
- Update the balance to reflect a deposit
  - Txn 1 is depositing $100
  - Txn 2 is depositing $50
- We expect it to contain $150 after the two transactions

Transactions: read_committed

```
begin;
select bal from accounts where id = 1;
  bal
-------
    0

update accounts set bal = 100 where id = 1;
UPDATE 1

select bal from accounts where id = 1;
  bal
-------
  100


commit;
```

```
begin;
select bal from accounts where id = 1;
  bal
-------
    0

update accounts set bal = 50 where id = 1;
<blocks>
UPDATE 1

select bal from accounts where id = 1;
  bal
-------
   50

commit;
```

```
select bal from accounts where id = 1;
  bal
-------
   50
```

Transactions: serializable

```
begin;
select bal from accounts where id = 1;
  bal
-------
    0

update accounts set bal = 100 where id = 1;
UPDATE 1

select bal from accounts where id = 1;
  bal
-------
  100



commit;
```

```
begin;
select bal from accounts where id = 1;
  bal
-------
    0

update accounts set bal = 50 where id = 1;
<blocks>
ERROR:  could not serialize access due to concurrent
update
```

```
select bal from accounts where id = 1;
  bal
-------
   100
```

A difficult anomaly: Write Skew

When write skew happens, a transaction reads something, makes a decision based on the value it saw, and writes the decision to the database. However, by the time the write is made, the premise of the decision is no longer true.

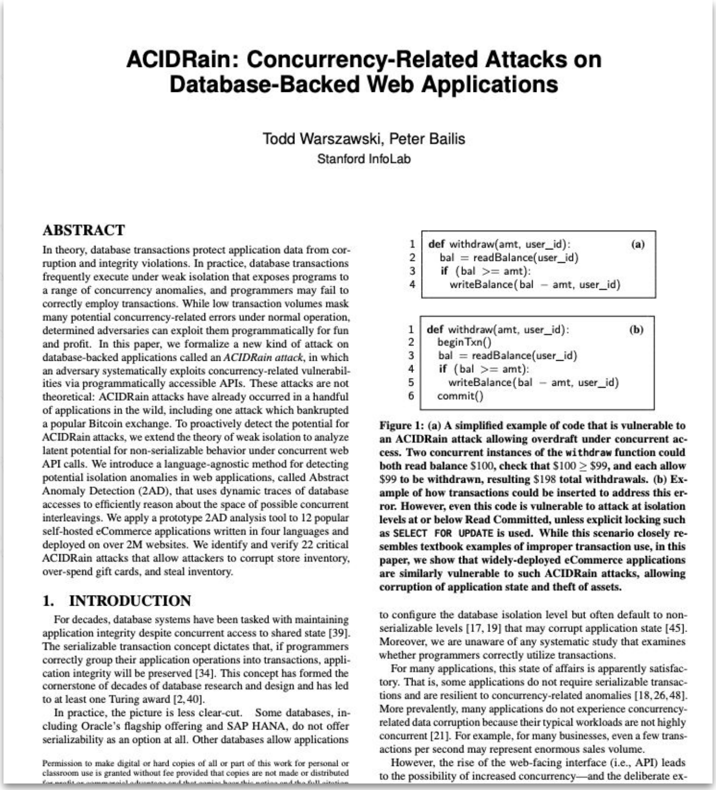
timeline

Client 1	Client 2
`Begin;` Employee 1 requests to vacation and app confirms employee 2 is on call	
	`Begin;` Employee 2 requests leave and app confirms Employee 1 is on call
Application allows Employee 1 to take leave	
	Application allows Employee 2 to take leave
`Commit;`	
	`Commit;`

Hackers exploit isolation issues

Stanford researchers found that not only do weak isolation levels result in bugs, they pose security risks.

Hackers can take advantage of weak isolation levels to put items into shopping carts AFTER checkout, for example.



ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications

Todd Warszawski, Peter Bailis
Stanford InfoLab

ABSTRACT

In theory, database transactions protect application data from corruption and integrity violations. In practice, database transactions frequently execute under weak isolation that exposes programs to a range of concurrency anomalies, and programmers may fail to correctly employ transactions. While low transaction volumes mask many potential concurrency-related errors under normal operation, determined adversaries can exploit them programmatically for fun and profit. In this paper, we formalize a new kind of attack on database-backed applications called an *ACIDRain attack*, in which an adversary systematically exploits concurrency-related vulnerabilities via programmatically accessible APIs. These attacks are not theoretical: ACIDRain attacks have already occurred in a handful of applications in the wild, including one attack which bankrupted a popular Bitcoin exchange. To proactively detect the potential for ACIDRain attacks, we extend the theory of weak isolation to analyze latent potential for non-serializable behavior under concurrent web API calls. We introduce a language-agnostic method for detecting potential isolation anomalies in web applications, called Abstract Anomaly Detection (2AD), that uses dynamic traces of database accesses to efficiently reason about the space of possible concurrent interleavings. We apply a prototype 2AD analysis tool to 12 popular self-hosted eCommerce applications written in four languages and deployed on over 2M websites. We identify and verify 22 critical ACIDRain attacks that allow attackers to corrupt store inventory, over-spend gift cards, and steal inventory.

```
def withdraw(amt, user_id):                       (a)
  bal = readBalance(user_id)
  if (bal >= amt):
    writeBalance(bal − amt, user_id)
```

```
def withdraw(amt, user_id):                       (b)
  beginTxn()
  bal = readBalance(user_id)
  if (bal >= amt):
    writeBalance(bal − amt, user_id)
  commit()
```

Figure 1: (a) A simplified example of code that is vulnerable to an ACIDRain attack allowing overdraft under concurrent access. Two concurrent instances of the `withdraw` function could both read balance \$100, check that \$100 ≥ \$99, and each allow \$99 to be withdrawn, resulting \$198 total withdrawals. (b) Example of how transactions could be inserted to address this error. However, even this code is vulnerable to attack at isolation levels at or below Read Committed, unless explicit locking such as `SELECT FOR UPDATE` is used. While this scenario closely resembles textbook examples of improper transaction use, in this paper, we show that widely-deployed eCommerce applications are similarly vulnerable to such ACIDRain attacks, allowing corruption of application state and theft of assets.

1. INTRODUCTION

For decades, database systems have been tasked with maintaining application integrity despite concurrent access to shared state [39]. The serializable transaction concept dictates that, if programmers correctly group their application operations into transactions, application integrity will be preserved [34]. This concept has formed the cornerstone of decades of database research and design and has led to at least one Turing award [2,40].

In practice, the picture is less clear-cut. Some databases, including Oracle's flagship offering and SAP HANA, do not offer serializability as an option at all. Other databases allow applications to configure the database isolation level but often default to non-serializable levels [17, 19] that may corrupt application state [45]. Moreover, we are unaware of any systematic study that examines whether programmers correctly utilize transactions.

For many applications, this state of affairs is apparently satisfactory. That is, some applications do not require serializable transactions and are resilient to concurrency-related anomalies [18,26,48]. More prevalently, many applications do not experience concurrency-related data corruption because their typical workloads are not highly concurrent [21]. For example, for many businesses, even a few transactions per second may represent enormous sales volume.

However, the rise of the web-facing interface (i.e., API) leads to the possibility of increased concurrency—and the deliberate ex-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed

Solving the problem in a distributed environment

Raft

Distributed Consensus Algorithm
provides “atomic writes” and consistent reads

Raft group

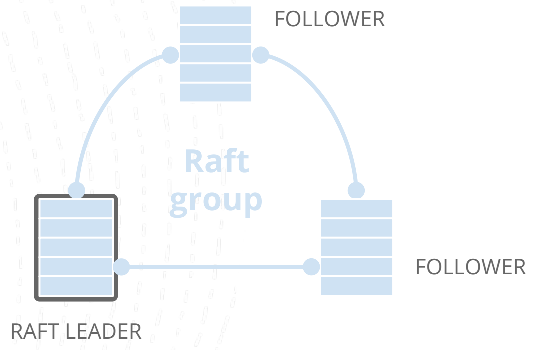
- A odd number of replicas
- Raft is a chatty protocol - gossip
- Coalesced heartbeats

Raft

Distributed Consensus Algorithm
provides “atomic writes” and consistent reads

Raft Leader

- Leader is elected
- Coordinates all writes, proposes commands to followers
- Only leader allowed to serve authoritative up-to-date

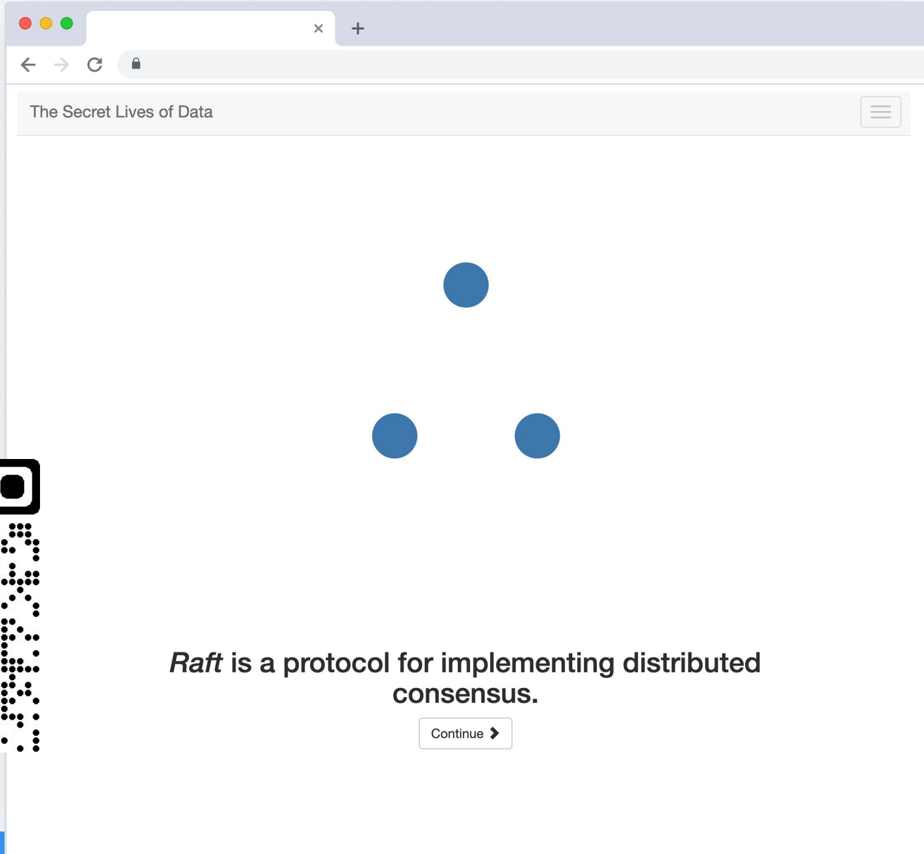


Raft

Distributed Consensus Algorithm

provides “atomic writes” and consistent reads

thesecretlivesofdata.com



MVCC - Multiversion concurrency control

Ensures consistent data is always present and that there is no “overlap” for transactions

Think of this as the “I” in ACID transactions - Isolation

MVCC - VERY Basic flow

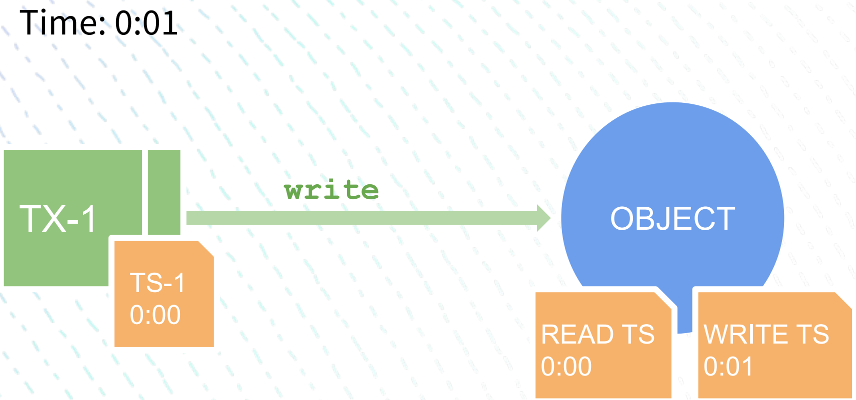
TX	TS	OBJECT
something we want to do to the data	a timestamp for transaction	a row of data, a record

MVCC - VERY Basic flow

Time: 0:00

For a simple transaction
At time zero, we simply have created a transaction request of an object - we want to write or update perhaps

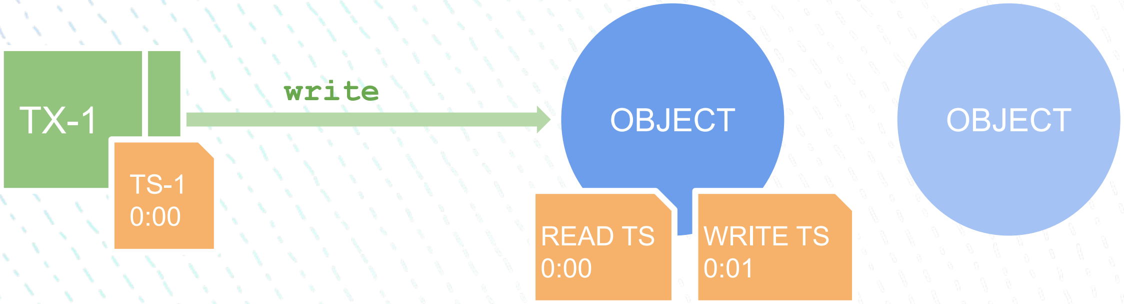
MVCC – VERY Basic flow



We write at 0:01
The object gets this write request at 0:01 and starts to deal with it

MVCC – VERY Basic flow

Time: 0:02

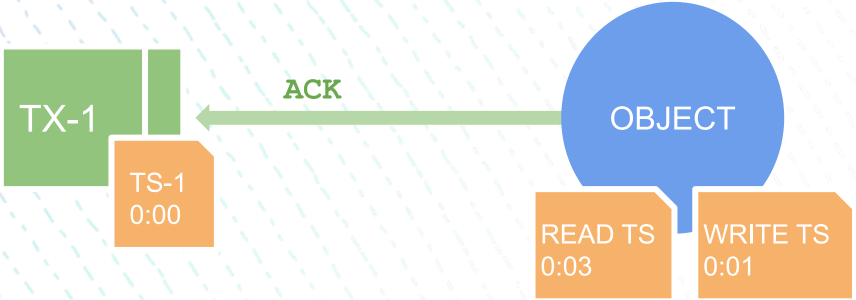


Temp Object
As part of the algorithm, a temp object is created so that we can do work to it without having a half completed state.

At this point, the write TS is incremented, but the rad remains as the most current “good” state.

MVCC – VERY Basic flow

Time: 0:03



Success

The temp object has done whatever it needed to do to commit the change and we now sync the read timestamp with the write timestamp so we have a “good” state of the object

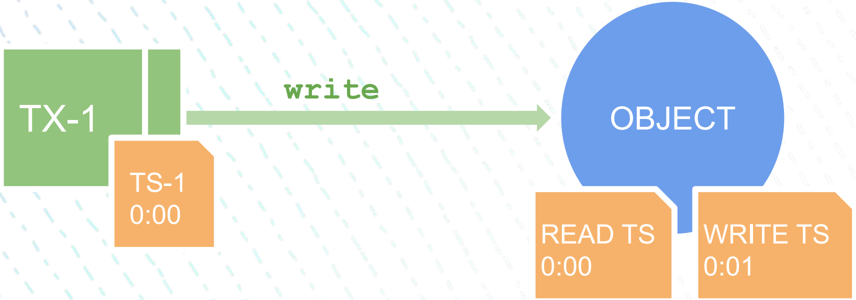
MVCC - Conflict

Time: 0:00

Let's try again, but with conflict
We start at 0:00 creating the transaction as a request

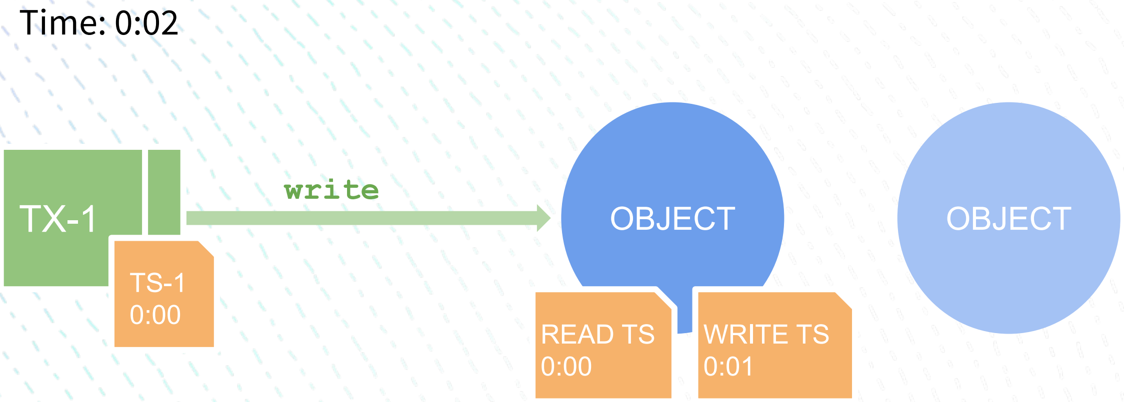
MVCC - Conflict

Time: 0:01



We write at 0:01
The object gets this write request at 0:01 and starts to deal with it

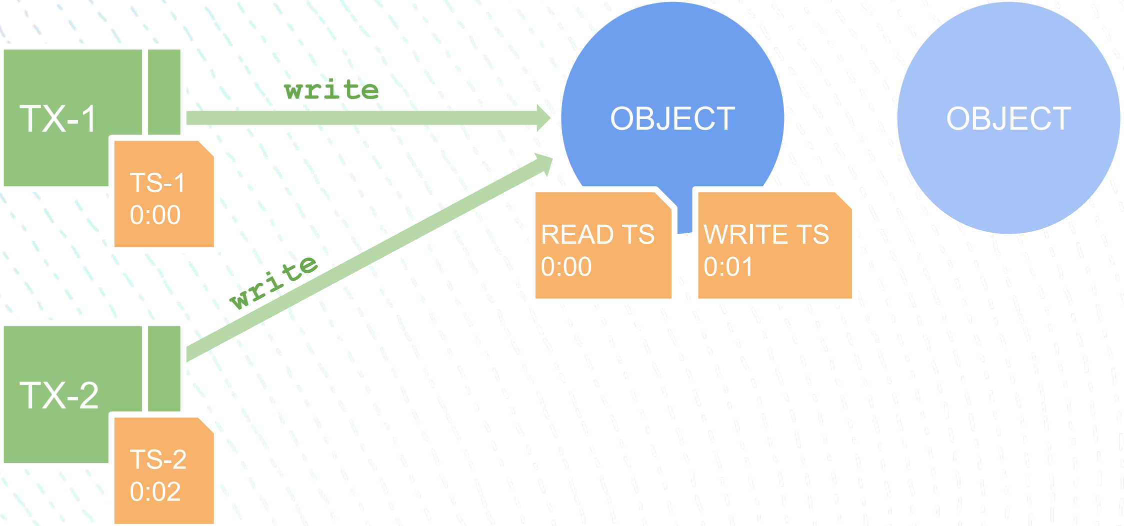
MVCC - Conflict



Temp Object
Just as before a temp object is created so we can work in it and timestamps are updated...

MVCC – Conflict

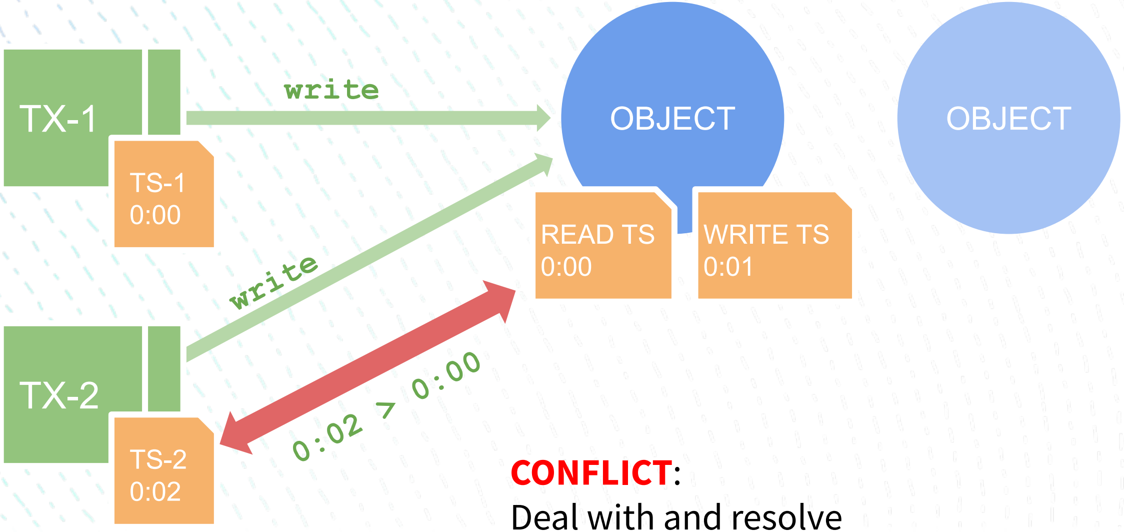
Time: 0:03



However...
While we are working in the temp object another transaction comes in on the same object.

MVCC - Conflict

Time: 0:03



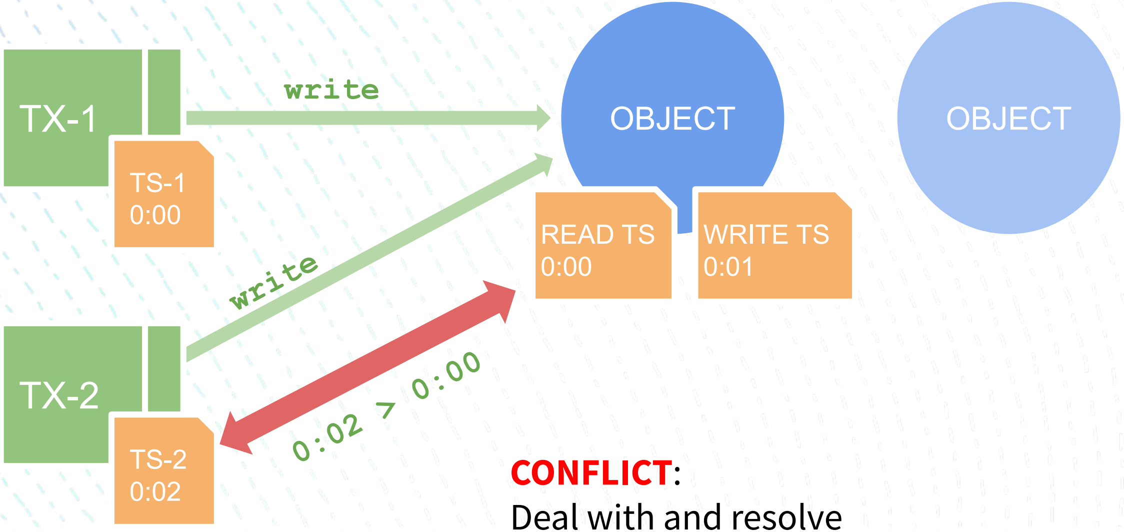
CONFLICT:
Deal with and resolve
Transaction retry?

However...
While we are working in the temp object another transaction comes in on the same object.

We compare timestamps and see that our timestamp food transaction two is greater than current timestamps on the object, so we have conflict

MVCC – Conflict

Time: 0:03



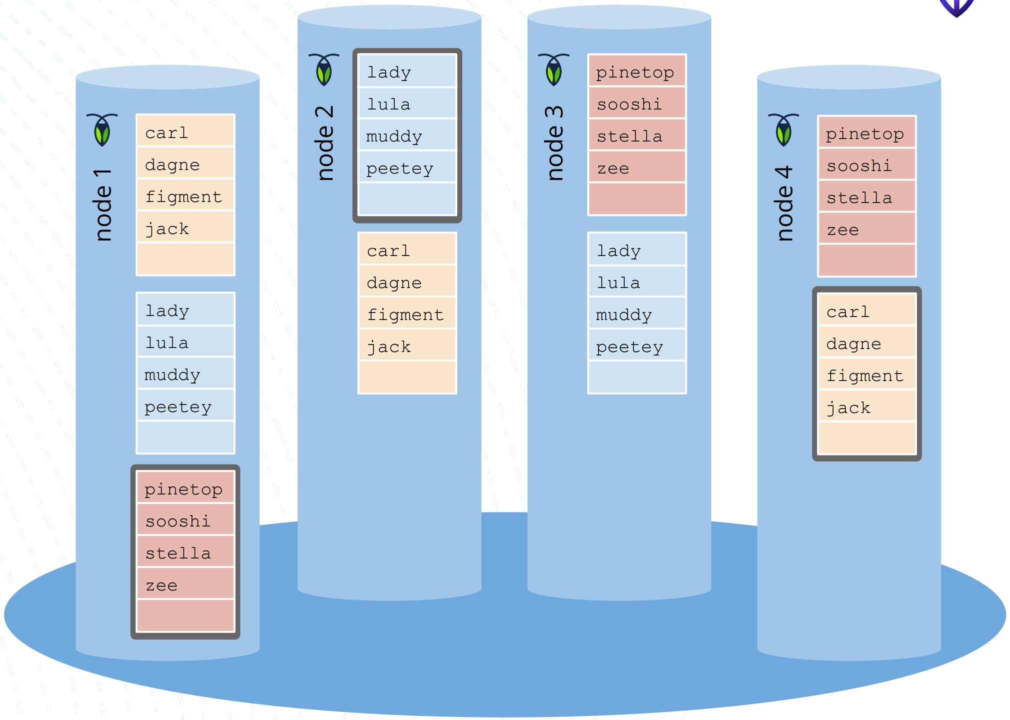
CONFLICT:
Deal with and resolve
Transaction retry?

Long Story Short...
Like standing in line at the store, you cannot complete your checkout transaction until those in front of you have completed theirs.

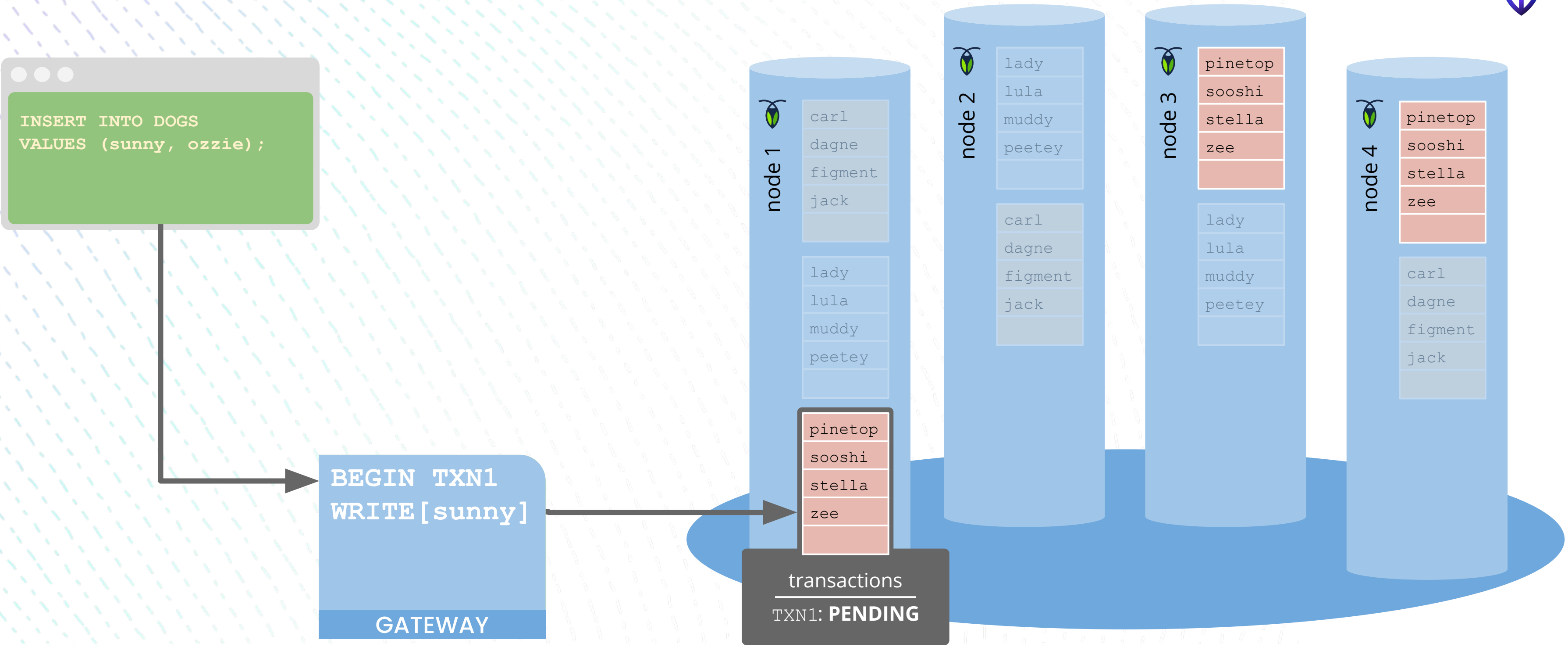
Distributed execution

Distributed Transactions

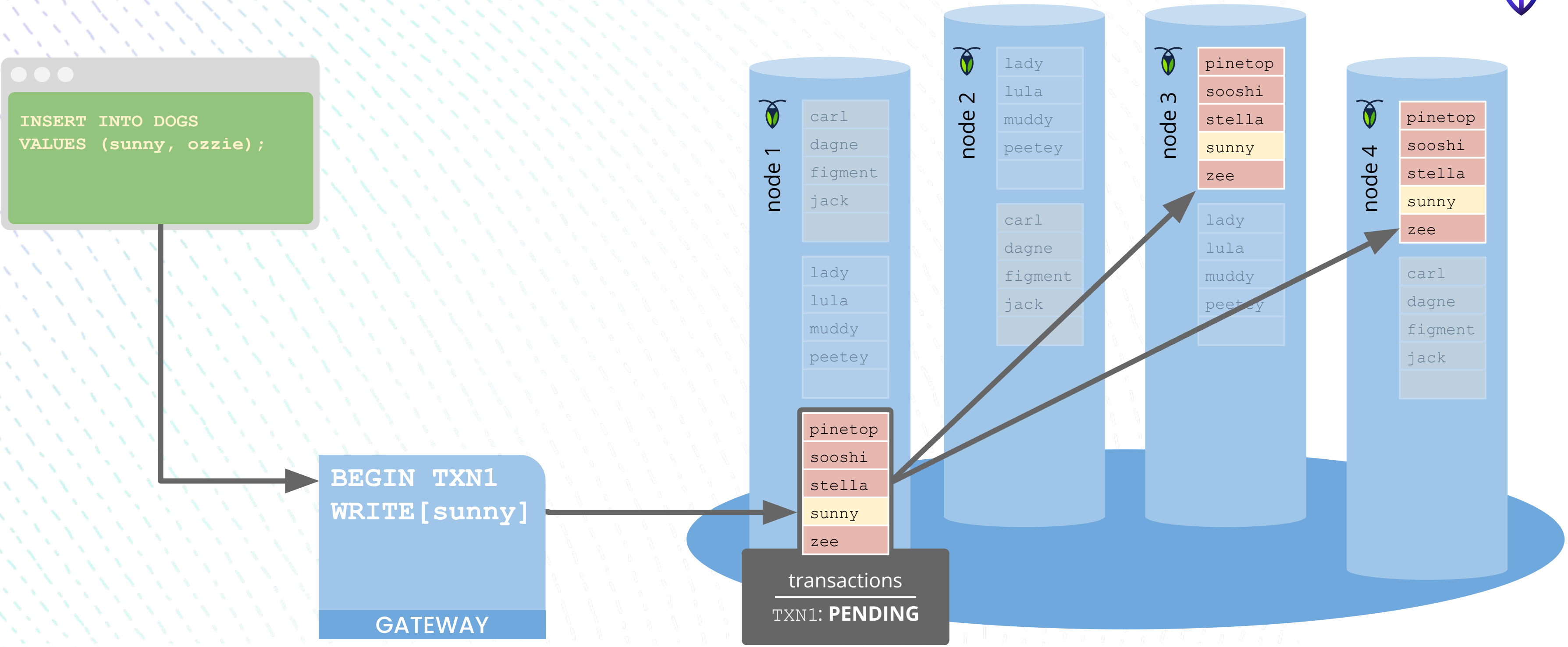
```
INSERT INTO DOGS
VALUES (sunny, ozzie);
```



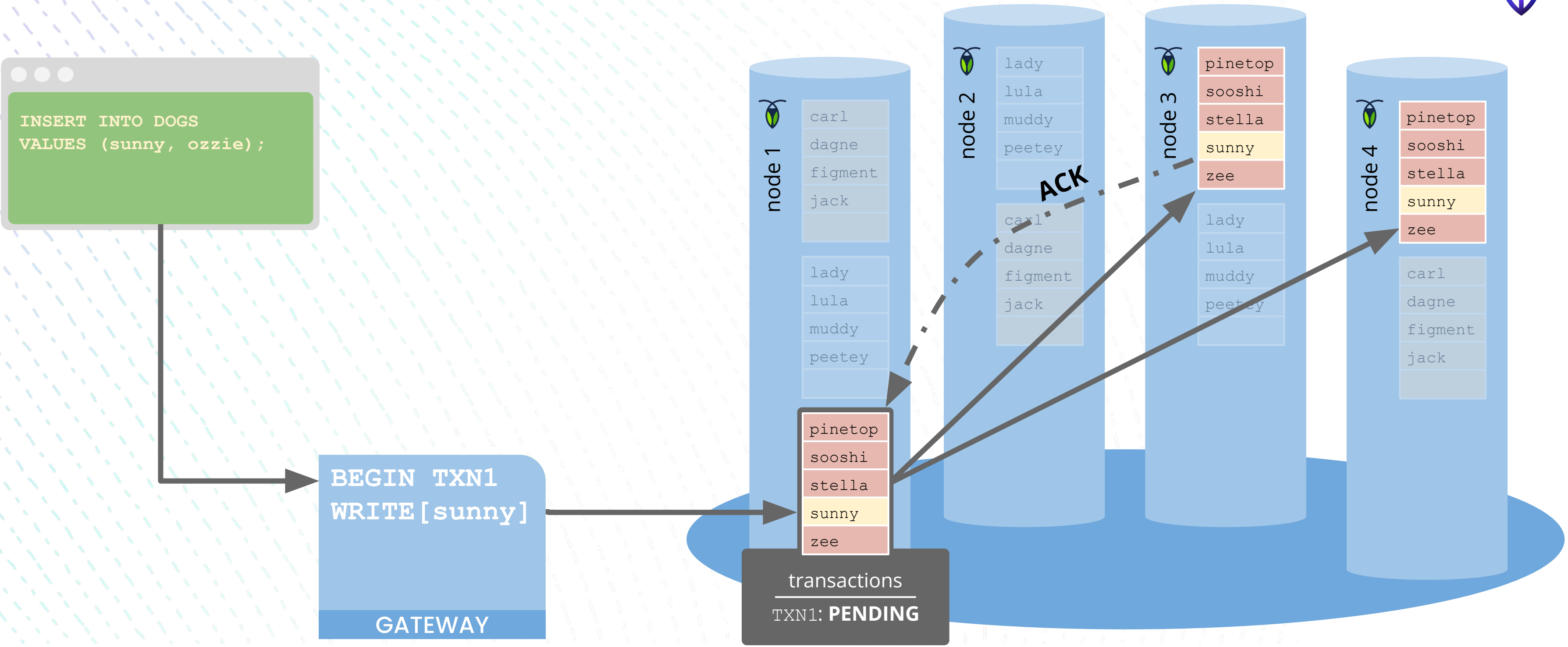
Distributed Transactions



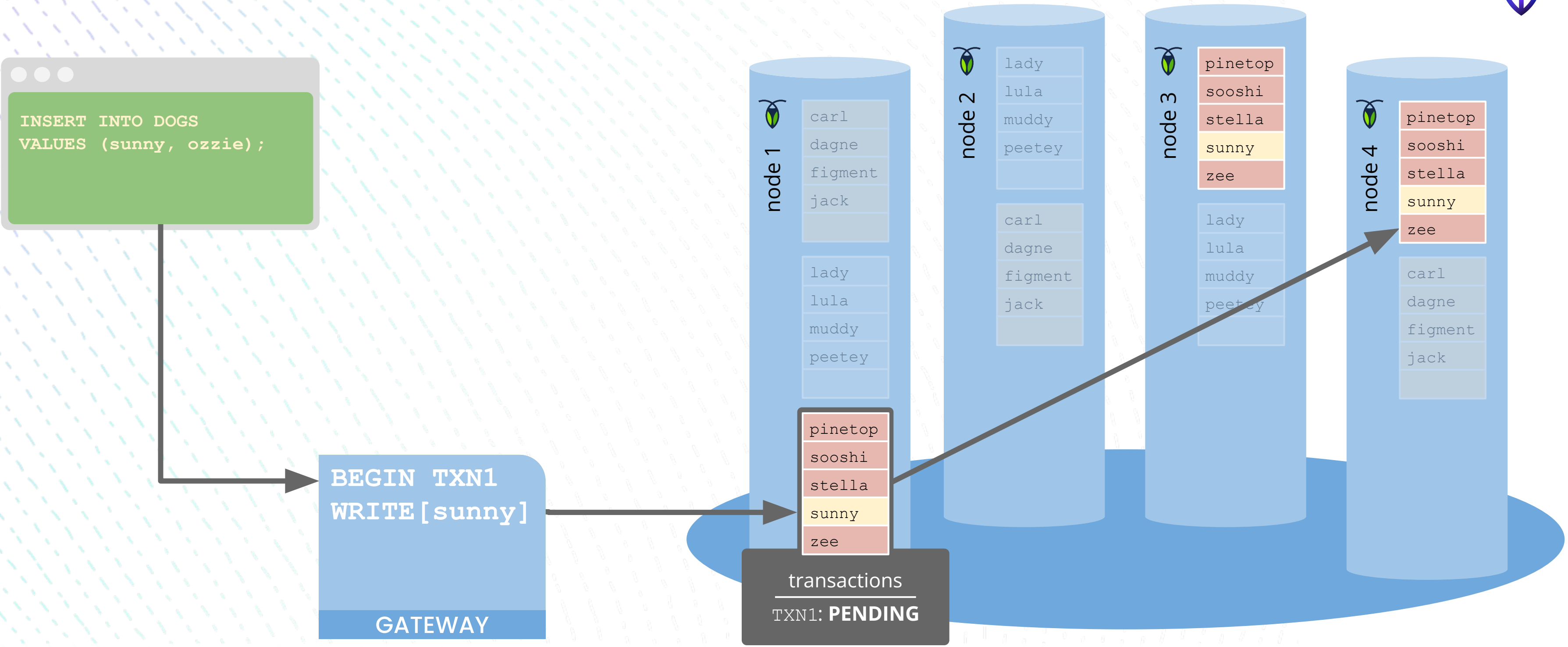
Distributed Transactions



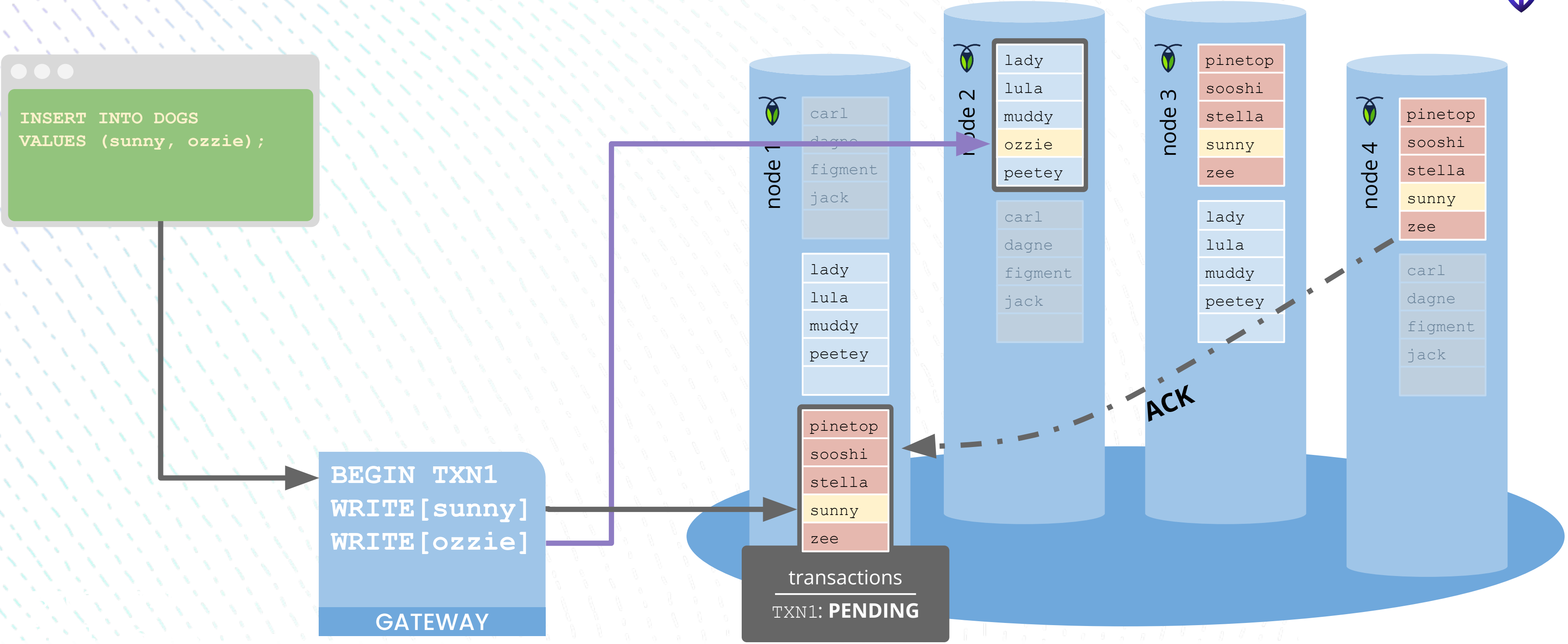
Distributed Transactions



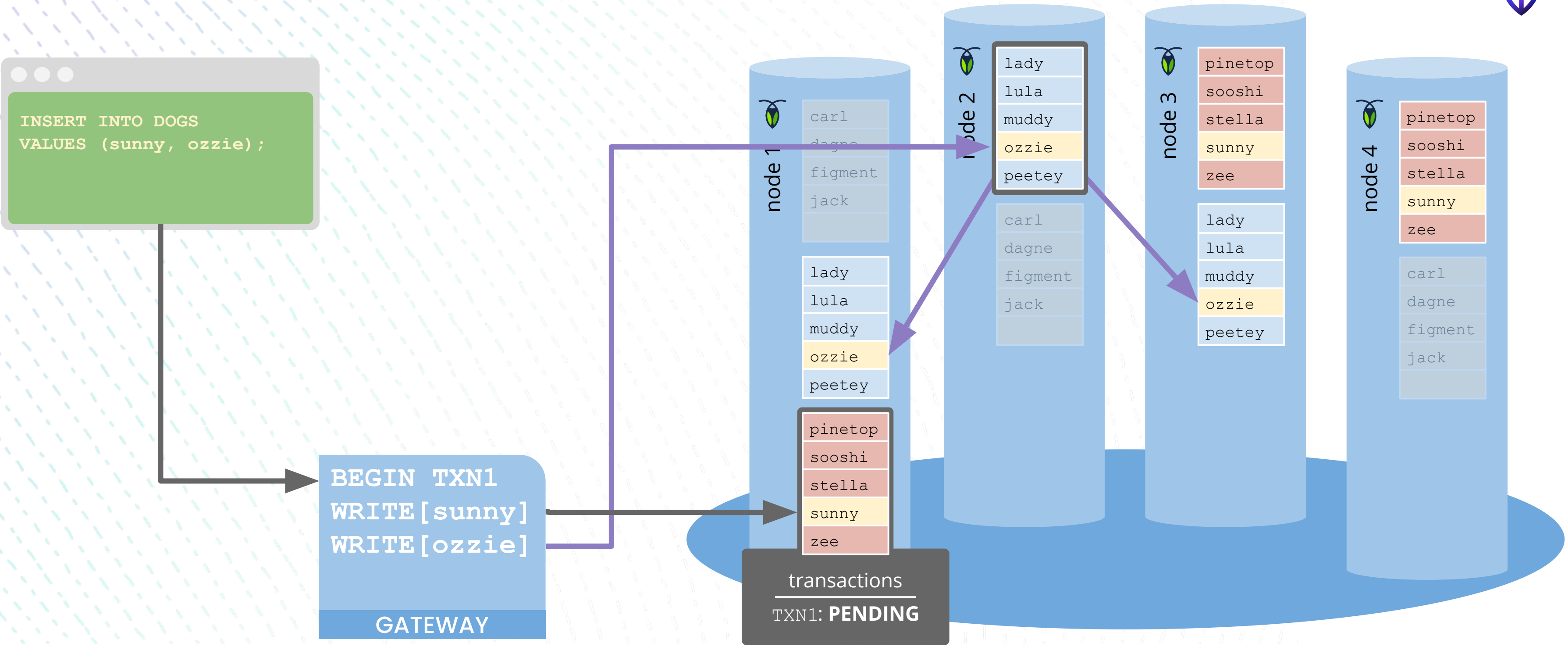
Distributed Transactions



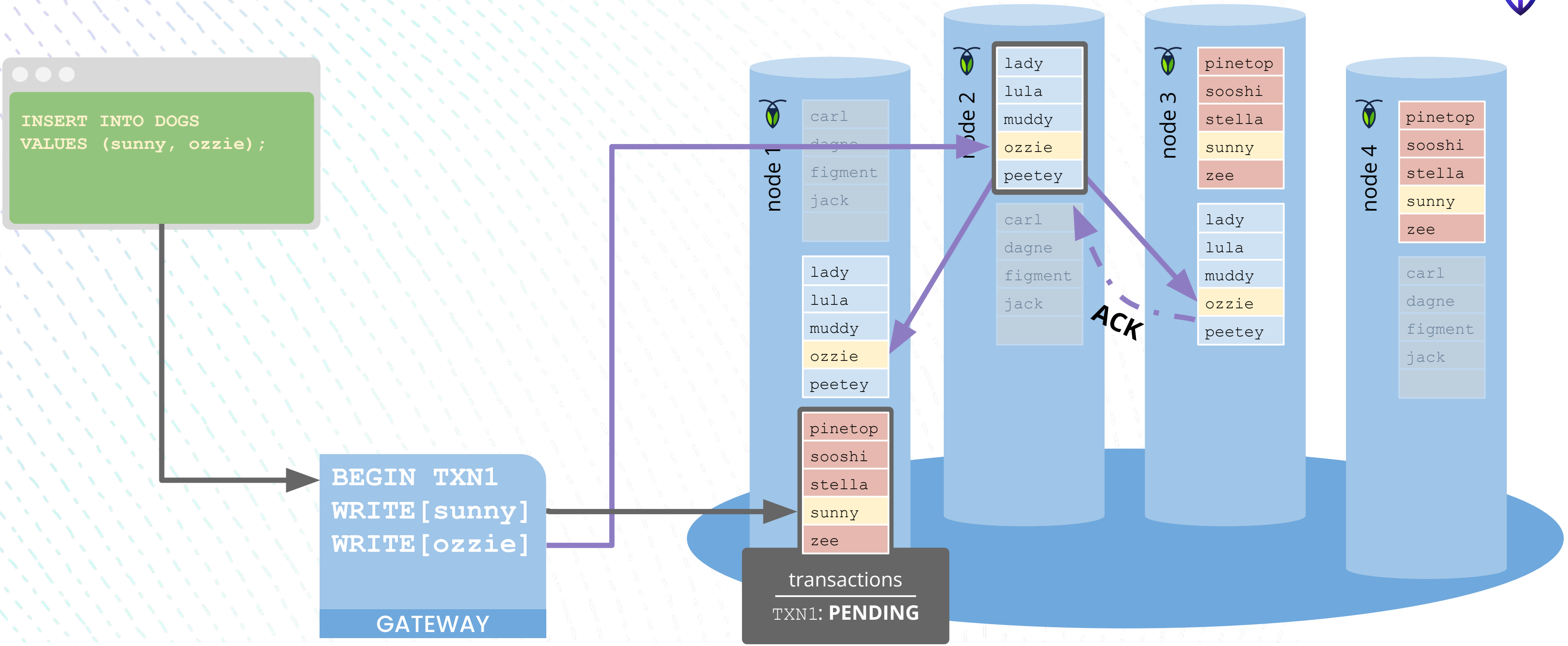
Distributed Transactions



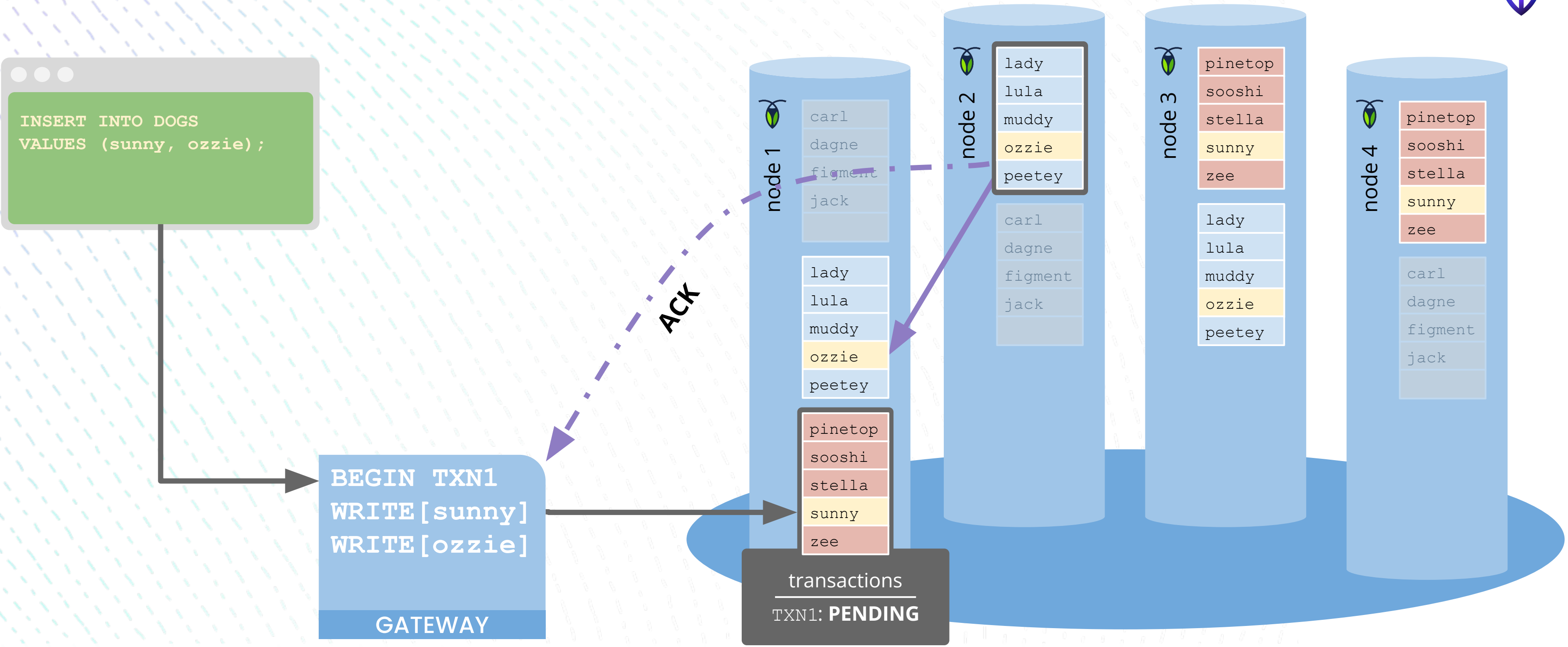
Distributed Transactions



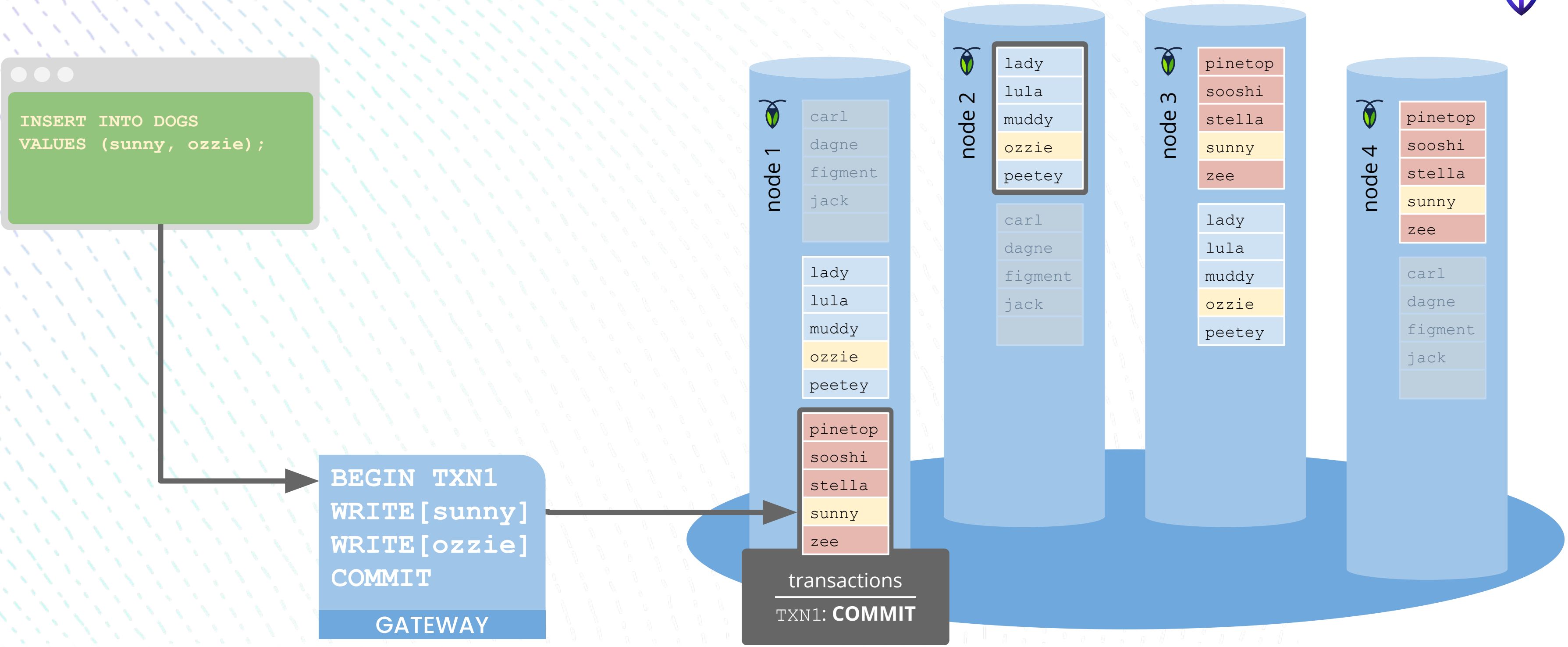
Distributed Transactions



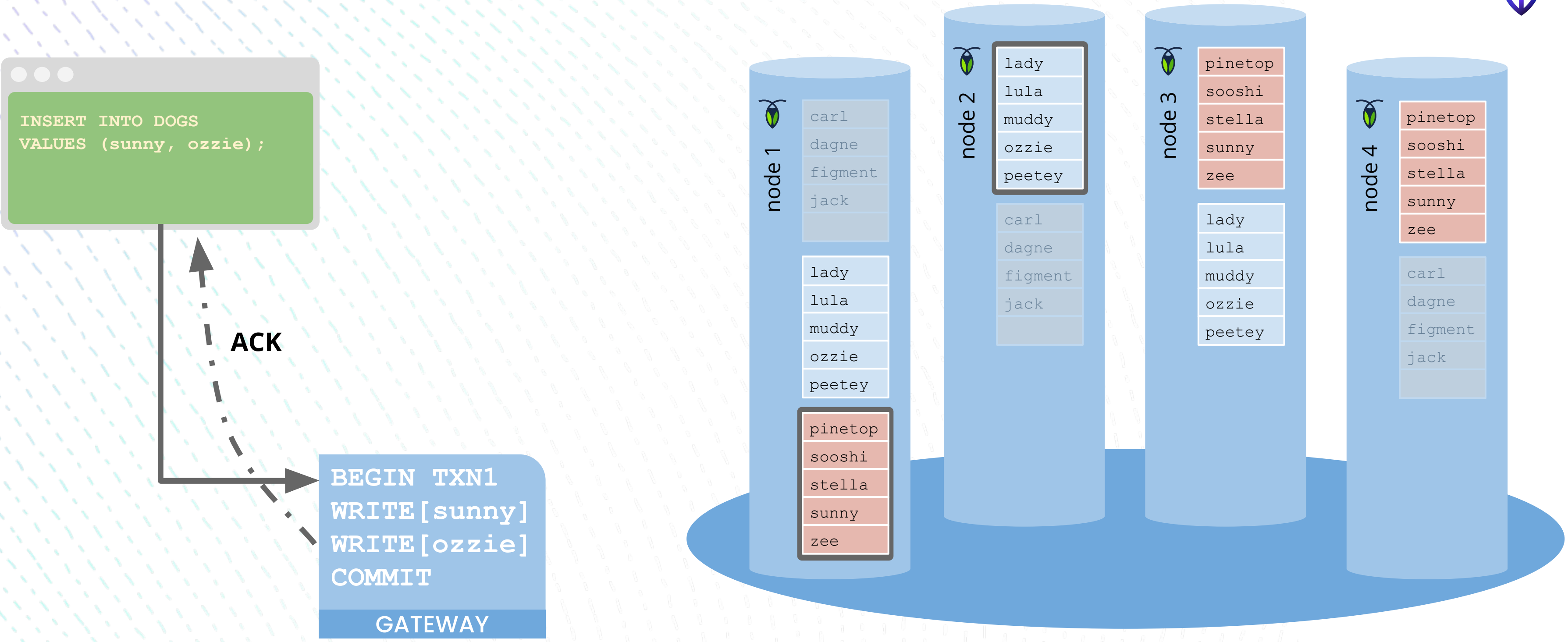
Distributed Transactions



Distributed Transactions



Distributed Transactions



Create a CockroachDB instance now…

Dedicated

A full featured, single tenant instance. Deploy instantly on AWS or GCP in a single region or across multiple regions with 99.99% guaranteed uptime.

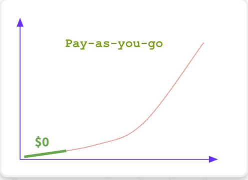
Serverless (beta)

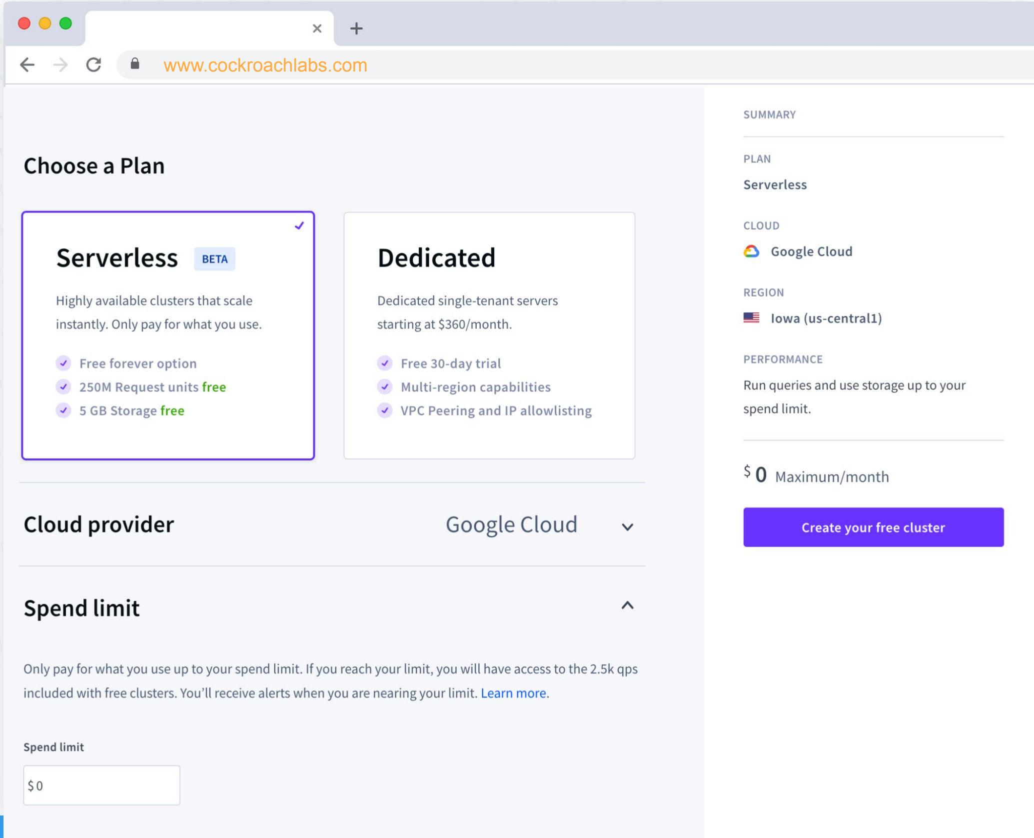
A single region instance with a generous free tier and with a capped pay for usage beyond free limits

No credit card required!

Free, every month up to:

- 5GB Storage
- 250M Request Units





Let our SRE team provision and manage your database.

THANK YOU!

Transactions: CockroachDB

```
begin;
select bal from accounts where id = 1;
  bal
-------
    0

update accounts set bal = 100 where id = 1;
UPDATE 1

select bal from accounts where id = 1;
  bal
-------
  100


commit;
```

```
select bal from accounts where id = 1;
  bal
-------
  150
```

```
begin;
select bal from accounts where id = 1;
<blocks>

  bal
-------
  100

update accounts set bal = 150 where id = 1;
UPDATE 1

select bal from accounts where id = 1;
  bal
-------
  150

commit;
```

CockroachDB vs Postgres Capability Matrix

	MySQL/ PostgreSQL	CockroachDB
Database Horizontal Scale Increase capacity of the database by adding more instances/nodes	Manual Sharding	Node based, Automated for both reads and writes
Database Load Balancing (internal) Locate data across multiple instances/nodes based on optimization criteria for balancing load	Manual - not part of database	Detailed options to optimize storage, compute and latency
Failover Provide access to backup data upon failure	Manual - Active Passive	Fully automated for both reads and writes
Automated Repair and RPO Repair the database after failure and the time it takes for the db to come back online	Manual Repair RPO ~1-60 mins	Automated Repair RPO <10 sec
Distributed Reads Reliably read data in any instance/node of the database	Manual - Asynchronous	Yes

Capability Matrix

	MySQL/ PostgreSQL	CockroachDB
Distributed Transactions Allow for acid writes across multiple instances/nodes	No	Yes
Database Isolation Levels Transaction isolation levels allowed for writes in the database	Single Region Consistent Default: Snapshot Highest: Serializable	Guaranteed Consistent Default: Serializable Highest: Serializable
Potential data issues (default) Possible data inconsistency issues at default isolation level	Phantom Reads, Non-repeatable reads, Wite skew	None
SQL Compliance with standard SQL	Yes	Yes - wire compatible with PostgreSQL
Database Schema Change Modify database schema across all tables	Yes	Online, Active, Dynamic

Capability Matrix

	MySQL/ PostgreSQL	CockroachDB
Cost Based Optimization Optimize execution of queries based on transaction analytics	Yes	Yes
Data Geo-partitoning Tie data to an instance/node to comply with regulations or optimize access latency	No	Yes, Row level
Upgrade Method Upgrade the database software	Offline	Online, Rolling
Multi-region Deploy a single database across multiple regions	Yes - Manual	Yes for both reads and writes
Multi-cloud Deploy a single database across multiple cloud providers or on-prem	No	Yes

Dirty Read Non repeatable Read Phantom

Atomicity

a transaction completes in total.. Could be simple or complex, it is an atomic unit and will complete fully

Consistency

any transaction will not violate the integrity of the database. Each will leave it in a valid state

Isolation

ensures that any concurrent transactions will leave the database in the same state as if they were executed sequentially.

Durability

committed no matter the circumstance of failure, etc.

Atomic

a transaction … e simple or complex, it is an atomic …

Isolation level

Read Uncommitted

Read Committed

Repeatable Read

$\boxed{\text{Serializable}}$

Consis…

any transacti… ty of the database. Each will leav…

Isolation

ensures that any concurrent transactions will leave the database in the same state as if they were executed sequentially.

Durability

committed no matter the circumstance of failure, etc.

Isolation level	Dirty Read	Nonrepeatable Read	Phantom
Read Uncommitted	✓	✓	✓
Read Committed	✗	✓	✓
Repeatable Read	✗	✗	✓
Serializable	✗	✗	✗

Isolation

ensures that any concurrent transactions will leave the database in the same state as if they were executed sequentially.