Building a Cloud-Native SQL Database

CockroachDB: Scalable, Survivable, Consistent, SQL

Presented by Alex Robinson / Member of the Technical Staff

Cockroach LABS

Why a cloud-native SQL database?

What is “Cloud-Native”?

- Horizontally scalable
- Built to handle failures
    - No single point of failure
- Survivable (self-healing)
- Minimal operator overhead (automatable)
- Decoupled from underlying platform

Cloud-Native Databases

- Sounds a lot like many NoSQL DBs, right?
    - Replication, scaling out, tolerating failures
- But what are they lacking?

Strong Consistency and Transactions

- Reasoning about eventual consistency and multi-step operations is hard
    - Wastes developer time
    - Causes very subtle bugs
- Avoid stale reads or data loss on failover
- Enable true SQL support
- “Make data easy”

Why aren't they in NoSQL databases?

- Fundamentally it was a matter of prioritization
- Coordination is very difficult
  - Especially when time is involved
  - Building a distributed database is hard enough as it is
- Consistency is often at odds with performance and scalability

Database Limitations

Existing database solutions place an undue burden on application developers:

- Scale (sql)
- Fault tolerance (sql)
- Limited transactions (nosql)
- Limited indexes (nosql)
- Consistency issues (nosql)

CockroachDB

CockroachDB

- Scalable
- Survivable
- Strongly Consistent
- SQL

And...

- Open Source

How does CockroachDB do it?

How does CockroachDB do it?

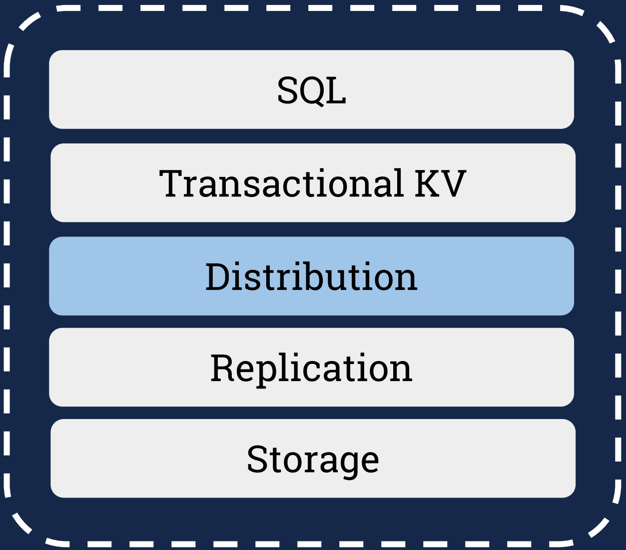
1. Data distribution and replication
2. Consensus protocol (Raft)
3. Distributed transaction model

Architecture (high-level)

Abstraction stack:

- SQL
- Transactional KV
- Distribution
- Replication
- Storage

Data Distribution



Data Distribution

Two key questions:

- At what granularity is data distributed?
- How do I locate a particular piece of data?

The primary options:

Hashing or Order-Preserving

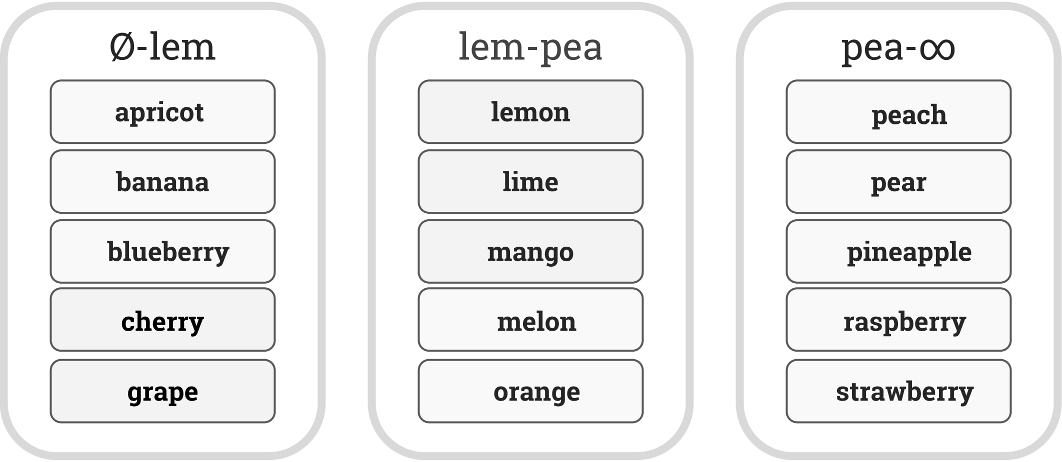
Data Distribution: Order-Preserving

The alternative to hashing is an order-preserving data distribution:

- Pro: efficient scans
- Con: requires additional indexing

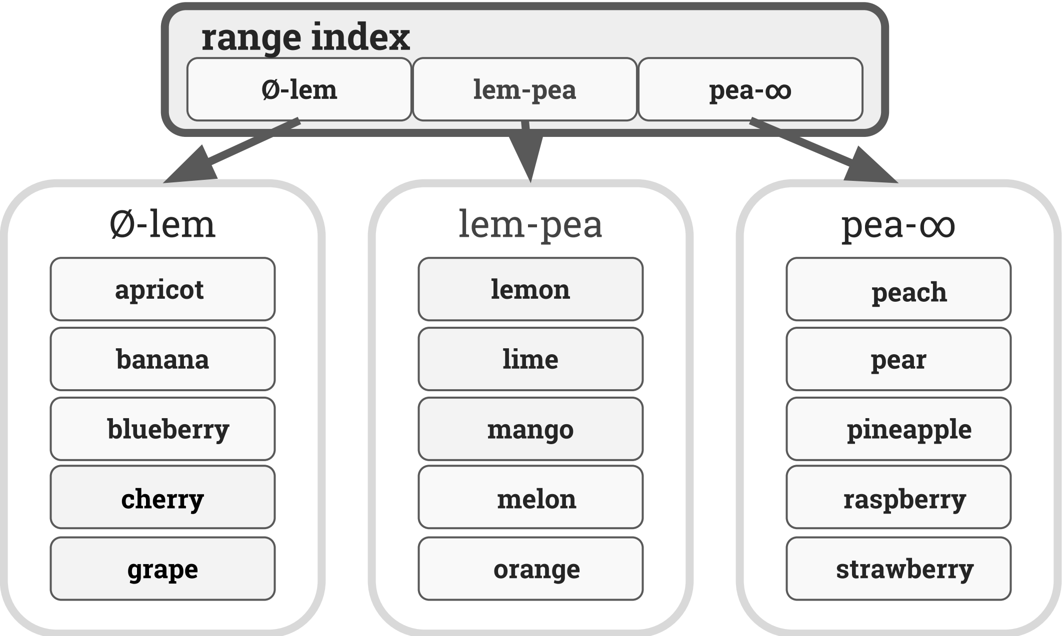
Data Distribution: Order-Preserving

Each range contains a contiguous segment of the key space



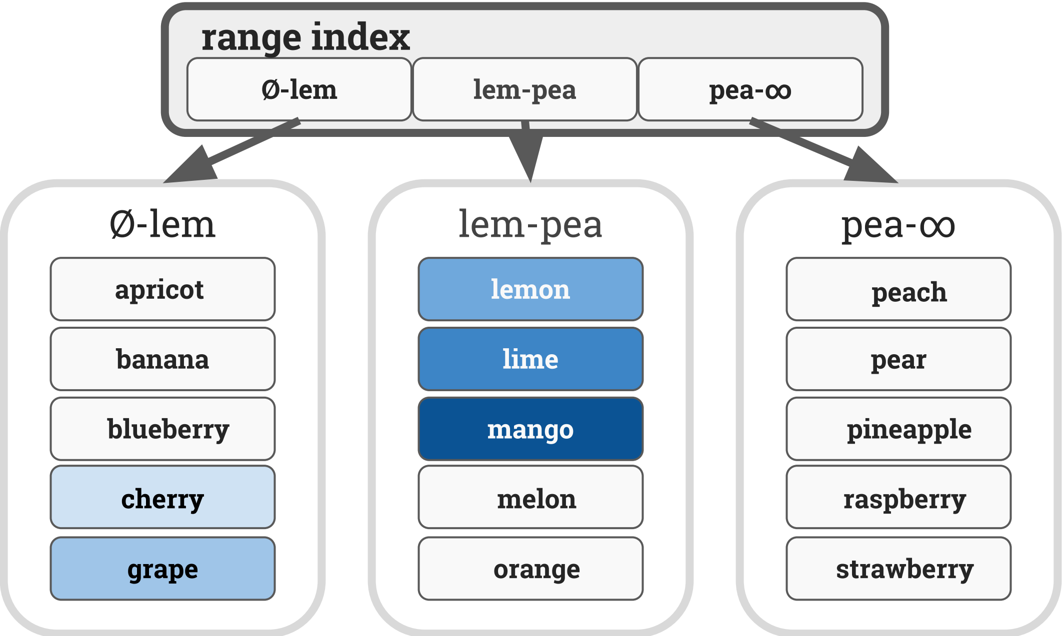
Data Distribution: Order-Preserving

We need an indexing structure to locate a range



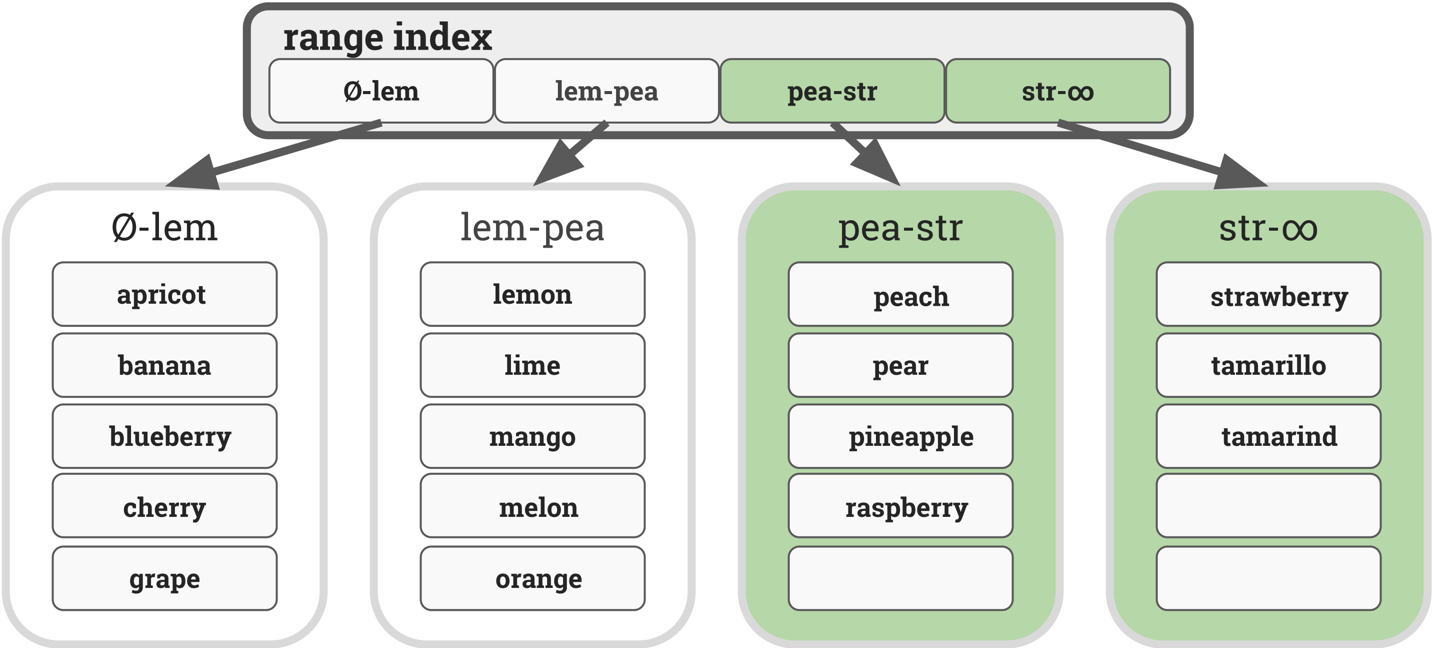
Data Distribution: Order-Preserving

Scans (fruits >= "cherry" AND <= "mango") are efficient

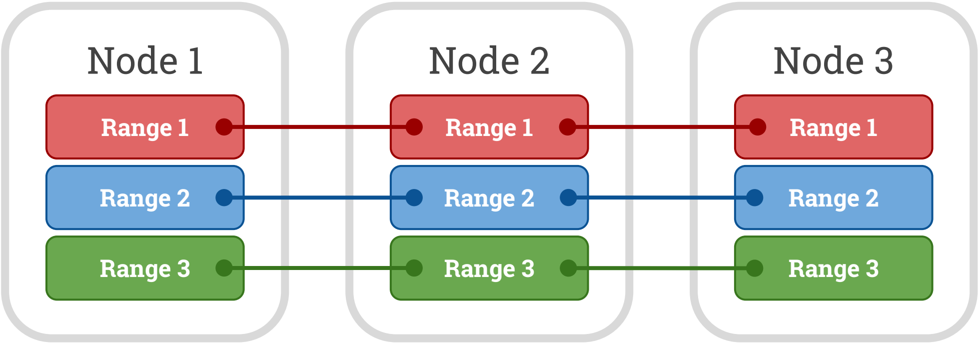


Data Distribution: Order-Preserving

Split when a range is too large

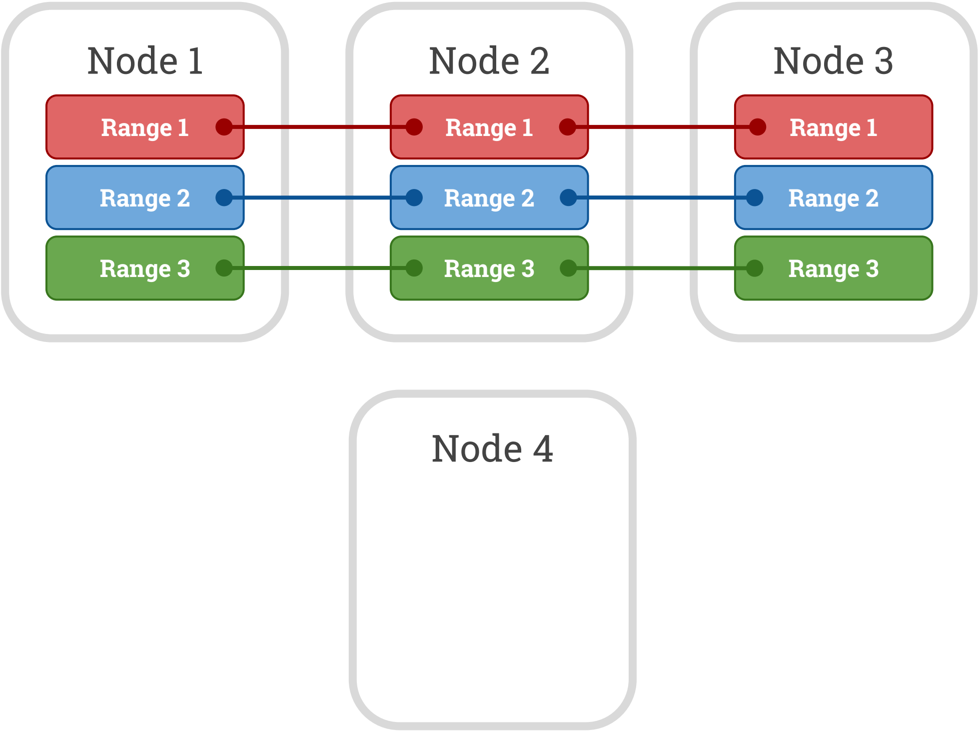


Data Distribution: Placement



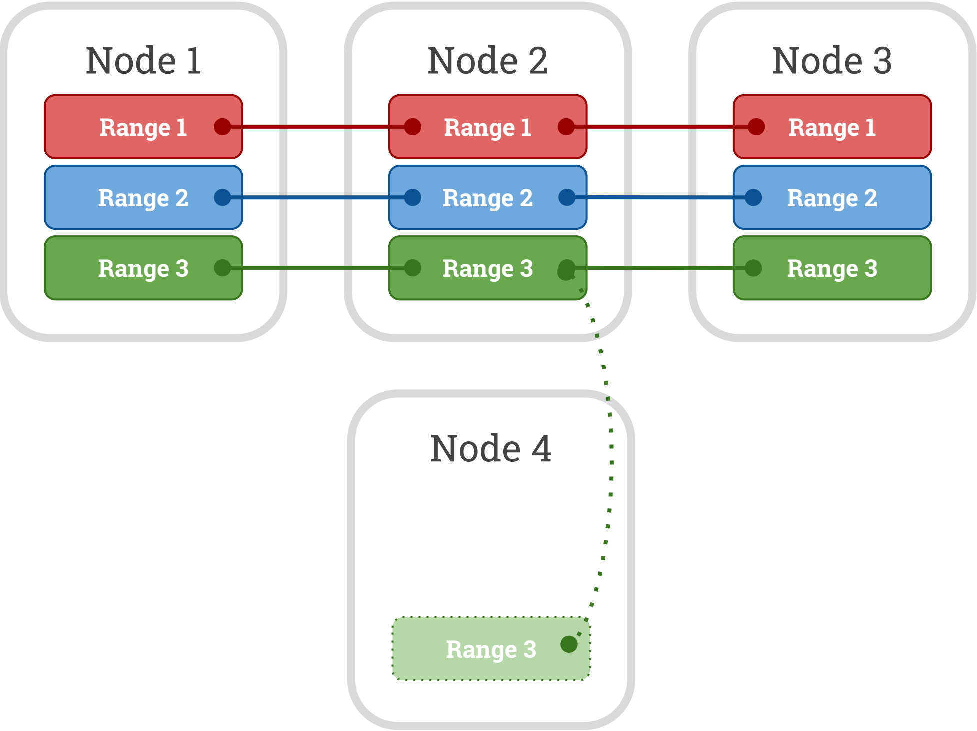
Each range is replicated to three or more nodes

Data Distribution: Rebalancing



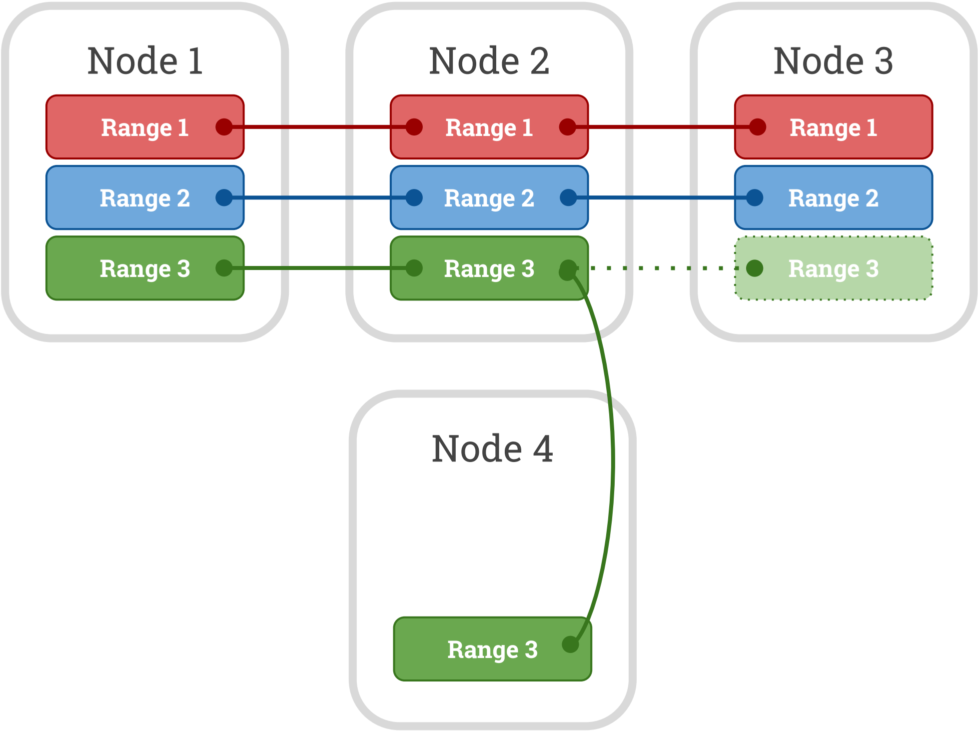
Adding a new (empty) node

Data Distribution: Rebalancing



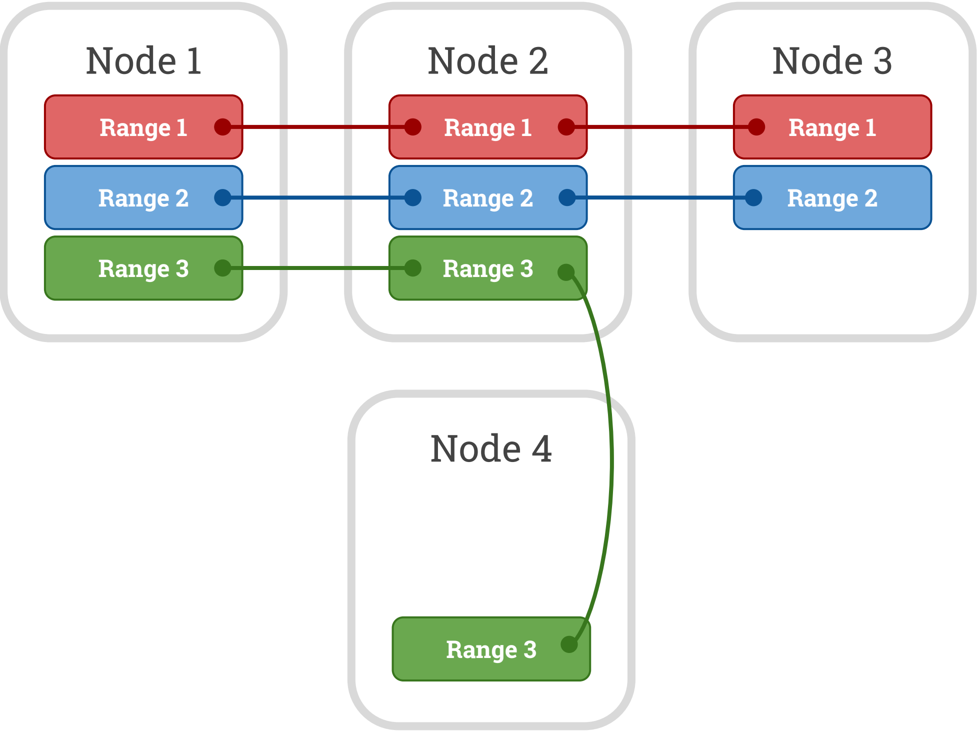
A new replica is allocated, data is copied.

Data Distribution: Rebalancing



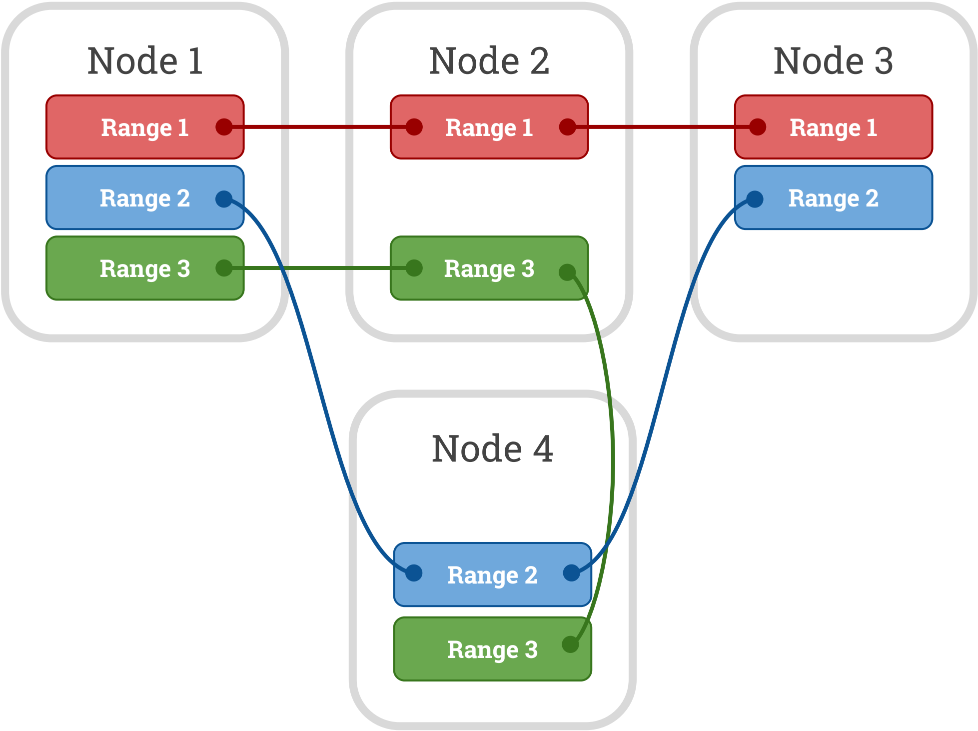
The new replica is made live, replacing another.

Data Distribution: Rebalancing



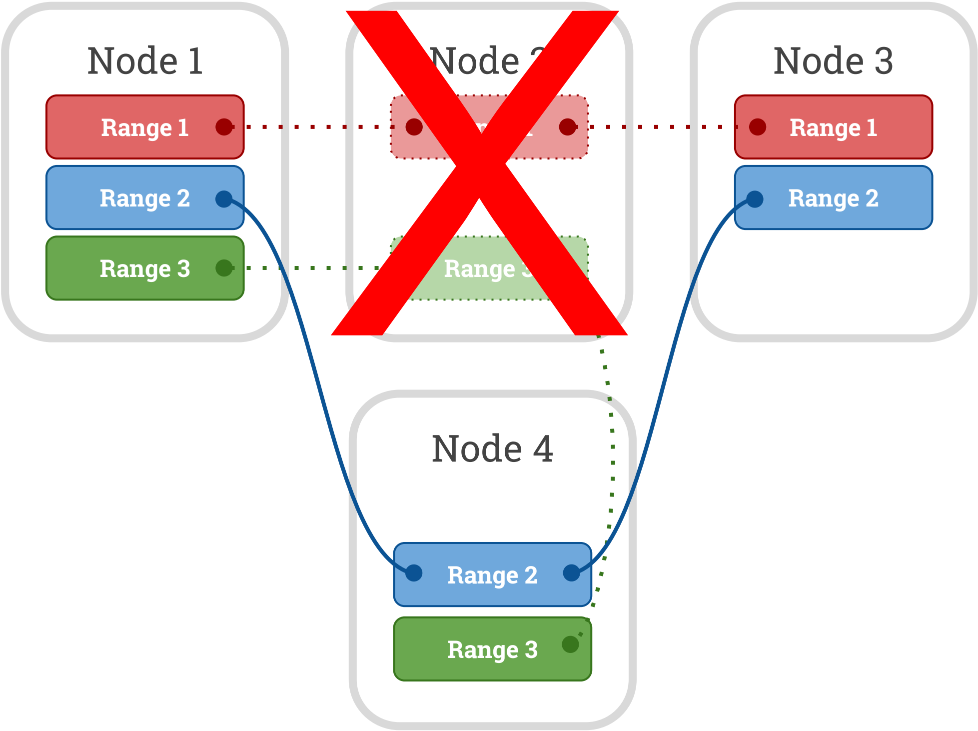
The old (inactive) replica is deleted.

Data Distribution: Rebalancing



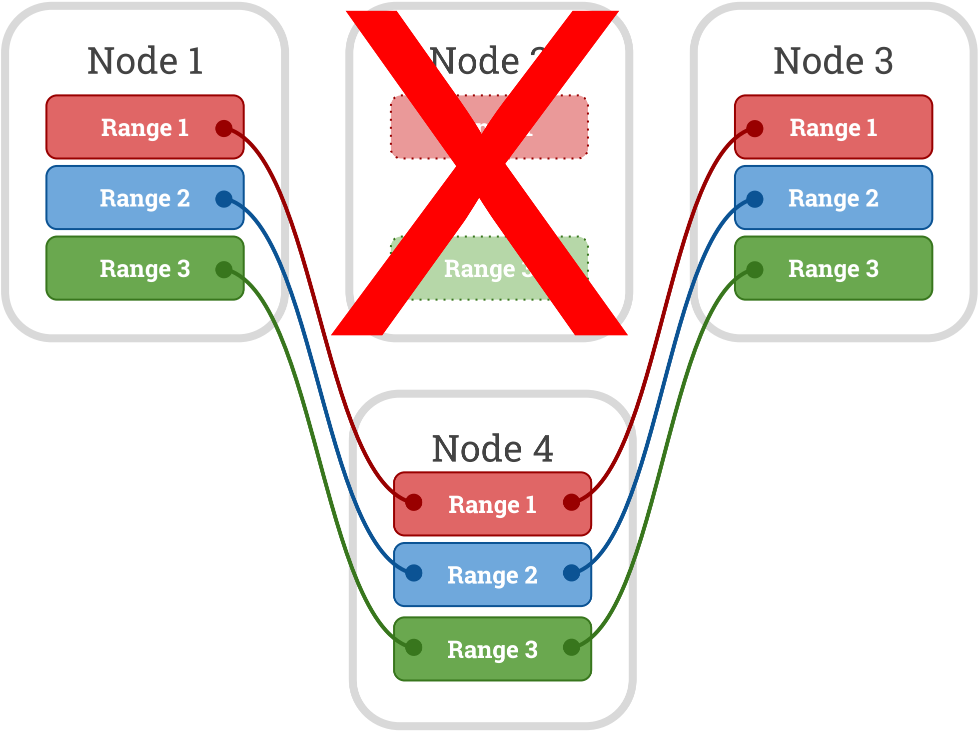
Process continues until nodes are balanced.

Data Distribution: Recovery



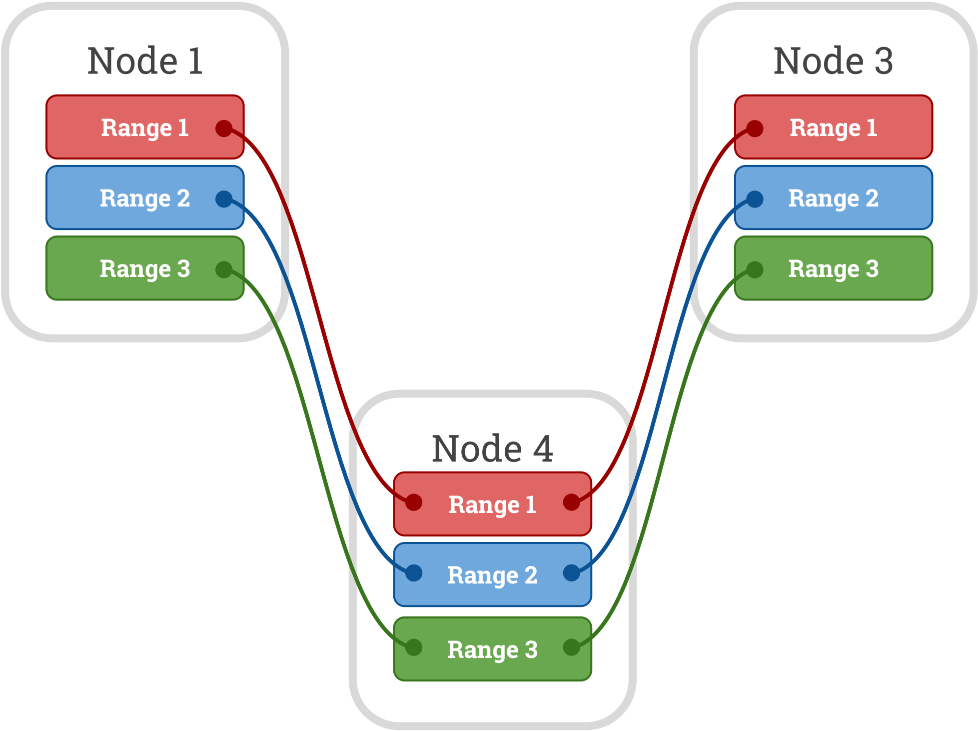
Losing a node causes recovery of its replicas.

Data Distribution: Recovery



A new replica gets created on an existing node.

Data Distribution: Recovery

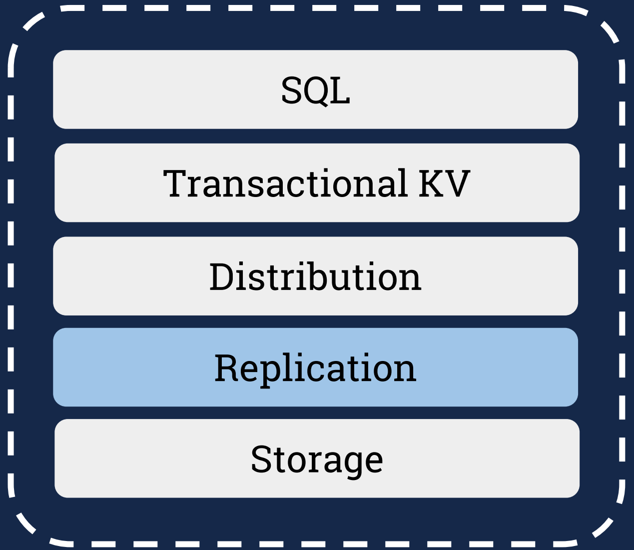


Once at full replication, the old replicas are forgotten.

Data Distribution

- Ranges are ~64 MB of data
  - Small enough to be moved/split quickly
  - Large enough to amortize indexing overhead
- This is fairly standard
  - CockroachDB/Bigtable/HBase/Spanner

Consensus



Achieving Consensus

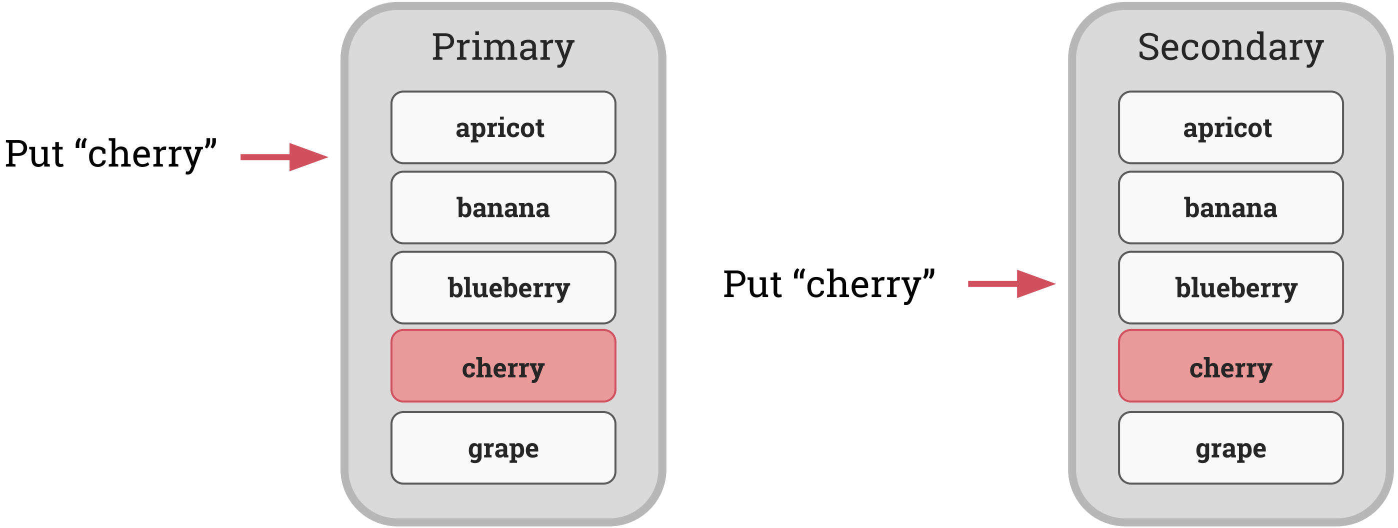
Production DBs must survive machine failure

Again, two main alternatives:

- Primary/secondary replication
- Consensus

Primary/Secondary Replication

Replicas contain identical copies of data: Voila!



Primary/Secondary Replication

- Asynchronous => stale reads
- Synchronous => lower availability, higher latency
- Primary to secondary failover requires care
  - Third-party needs to arbitrate primary

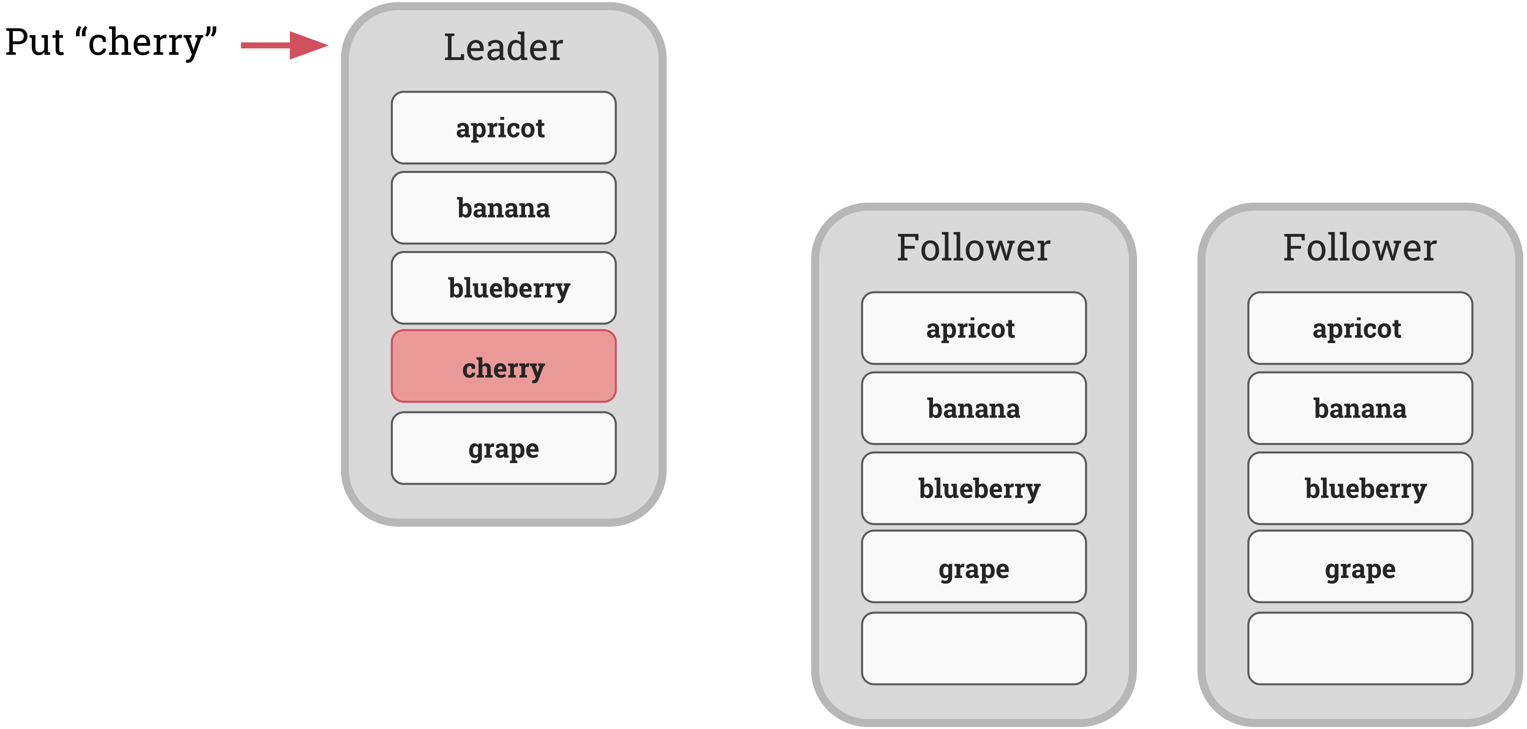
Consensus Replication

Replicate to N (often N=3) nodes

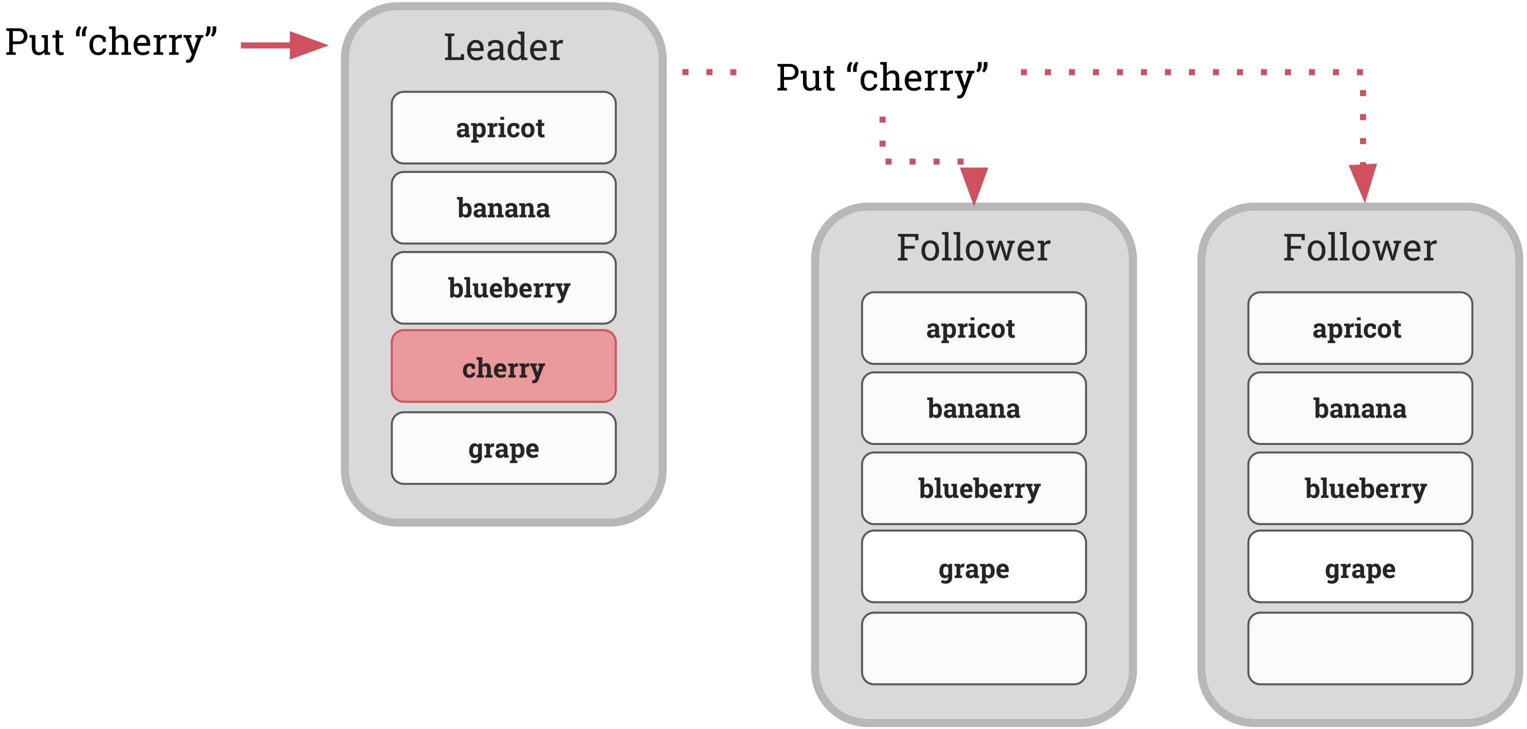
- Commit happens when a quorum have written the data

CockroachDB/Etcd/Spanner/Aurora/ZooKeeper/...

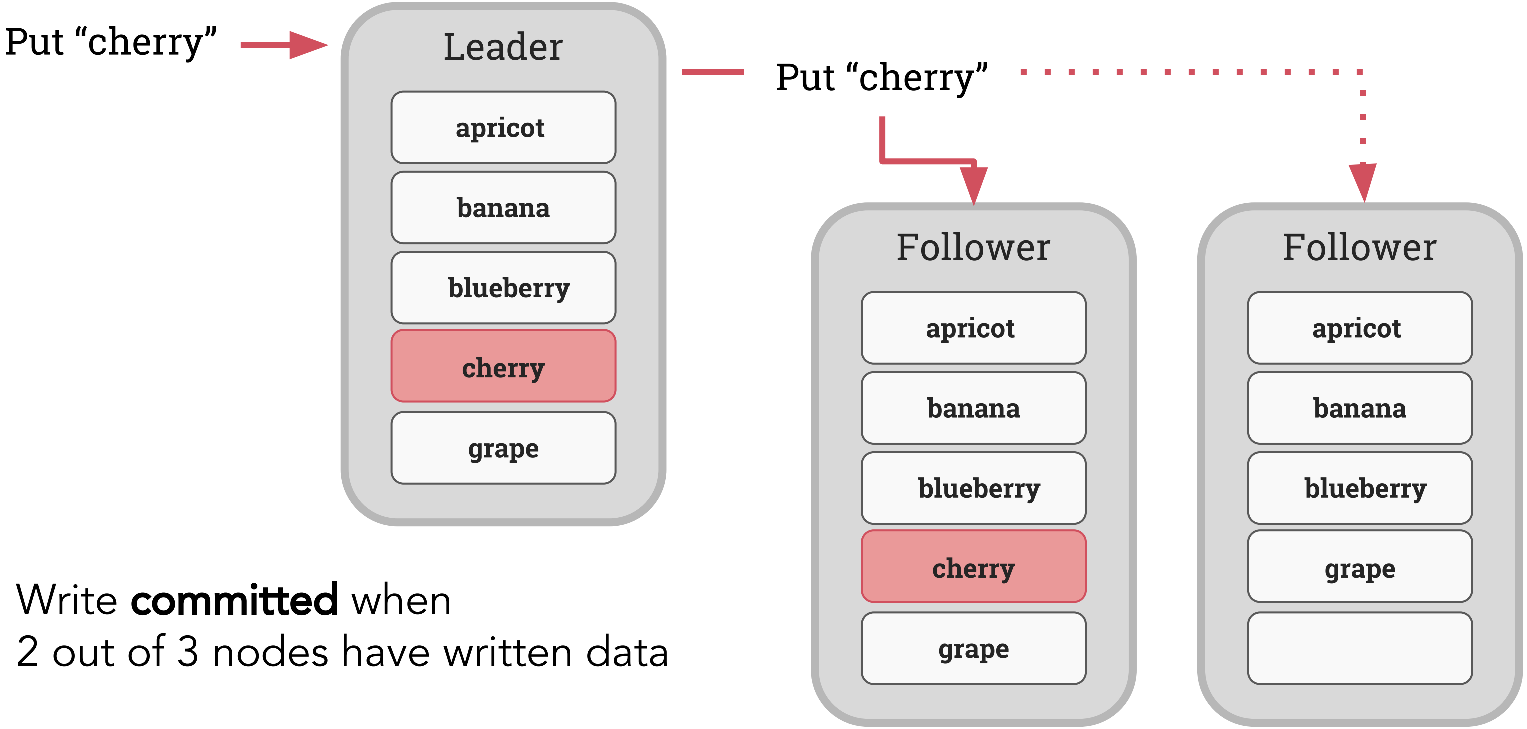
Consensus Replication



Consensus Replication

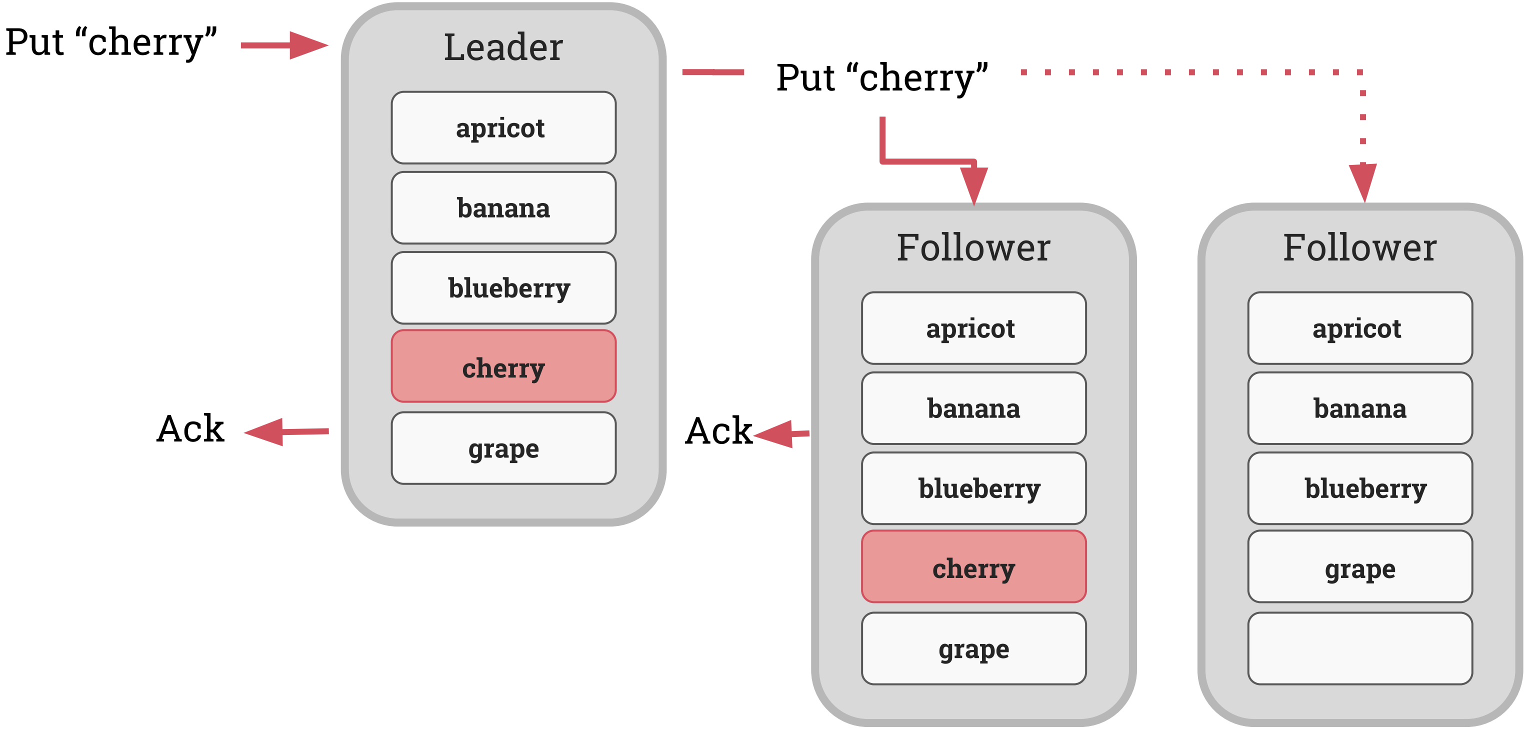


Consensus Replication



Write **committed** when
2 out of 3 nodes have written data

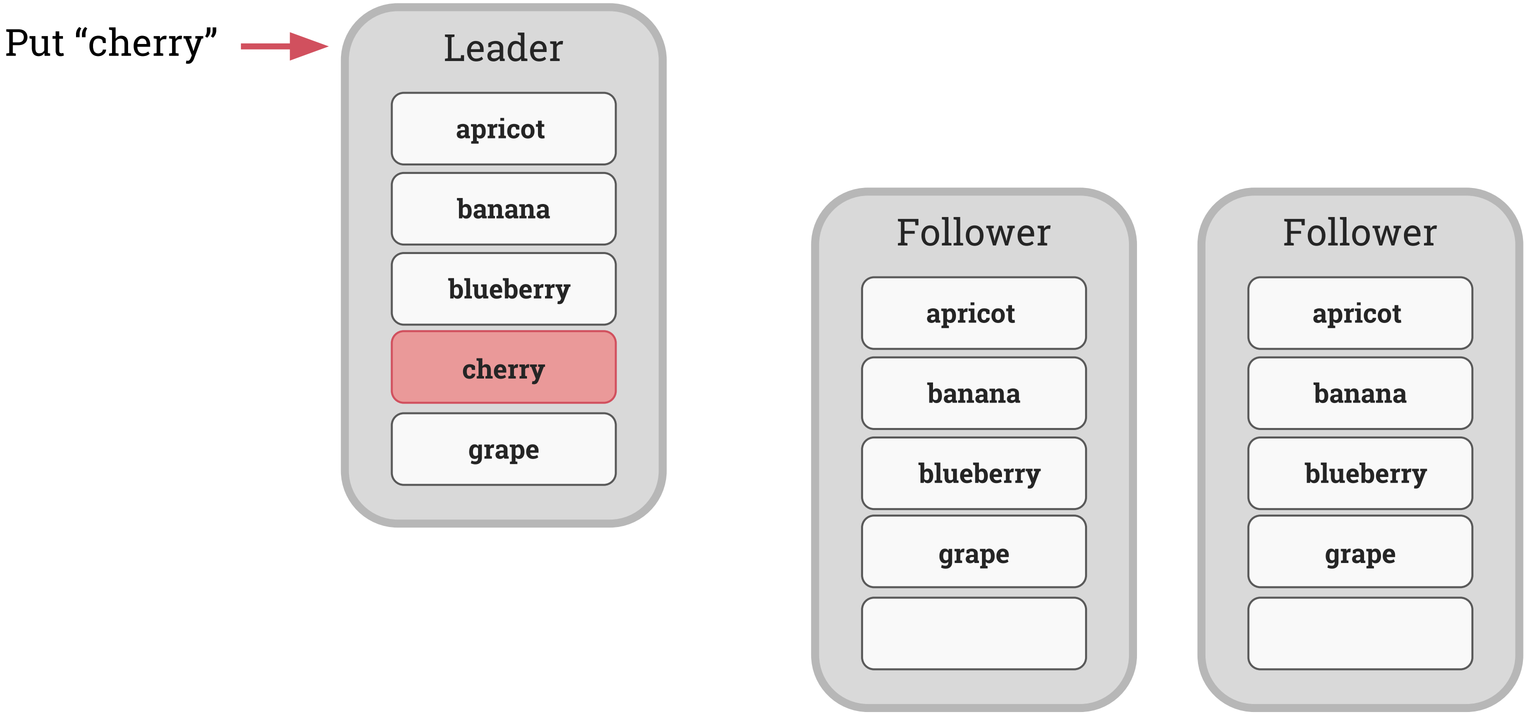
Consensus Replication



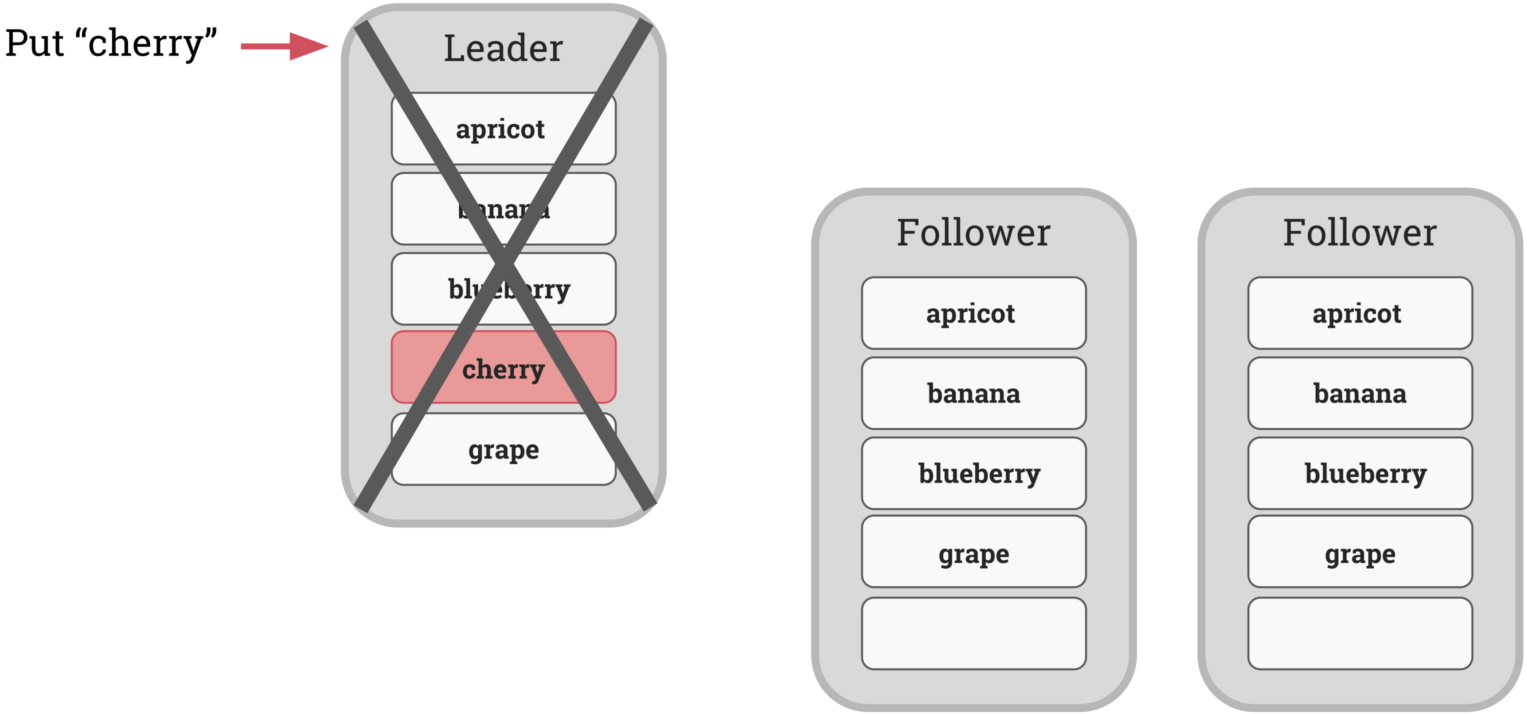
Consensus Replication

What happens during failover?

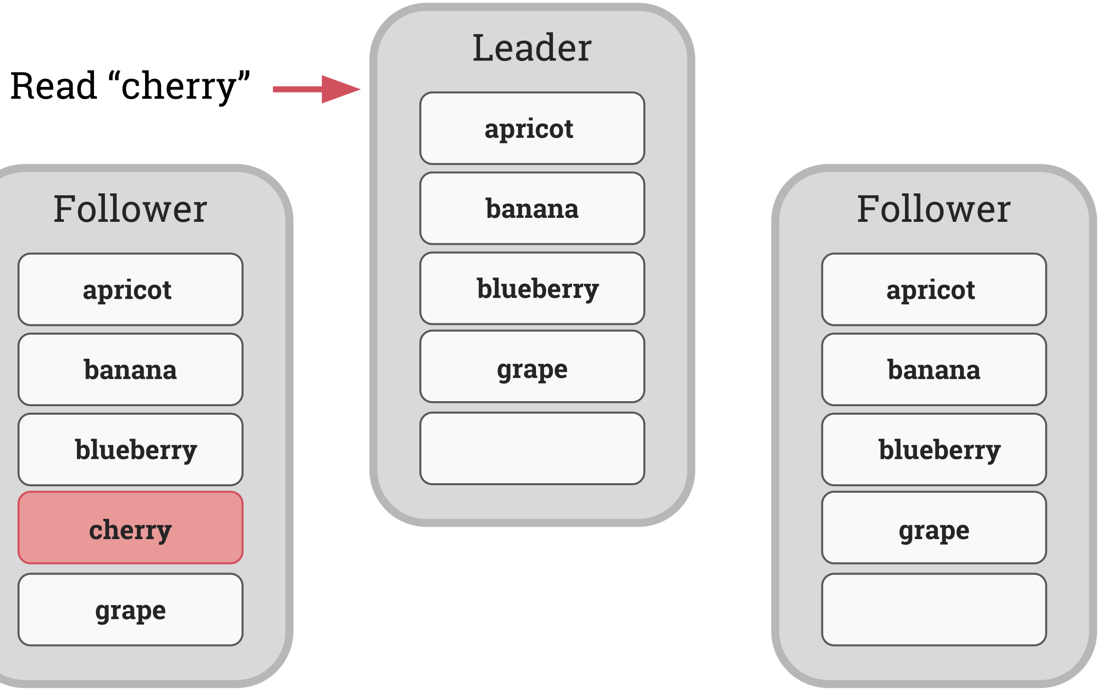
Consensus Replication: Failover



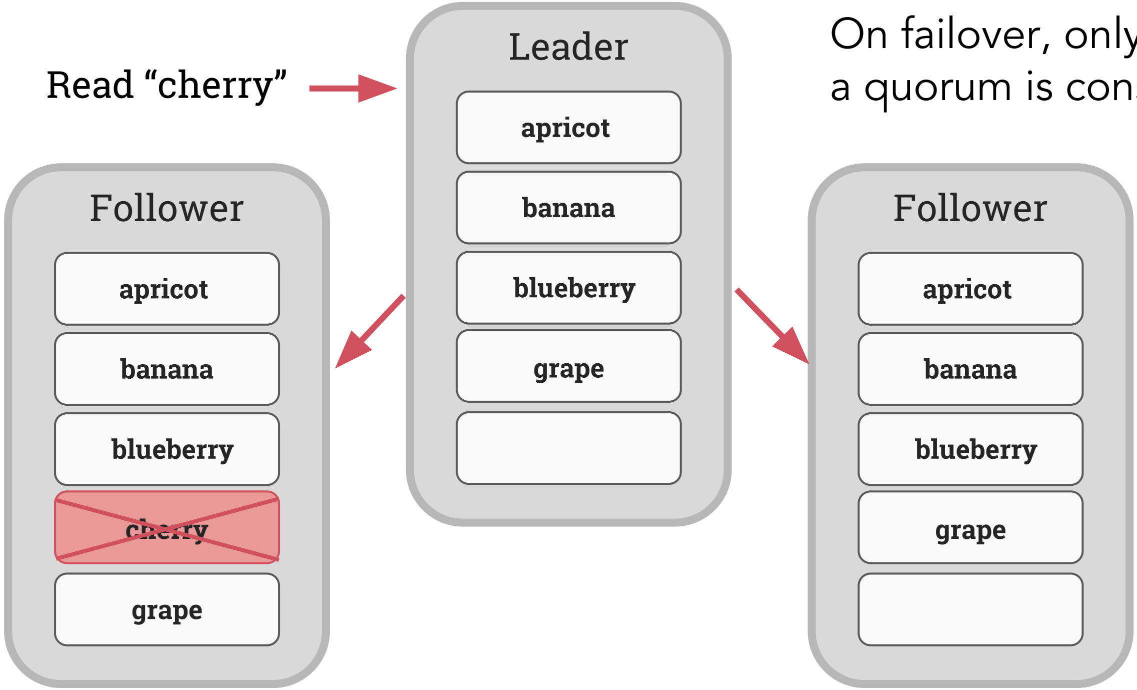
Consensus Replication: Failover



Consensus Replication: Failover

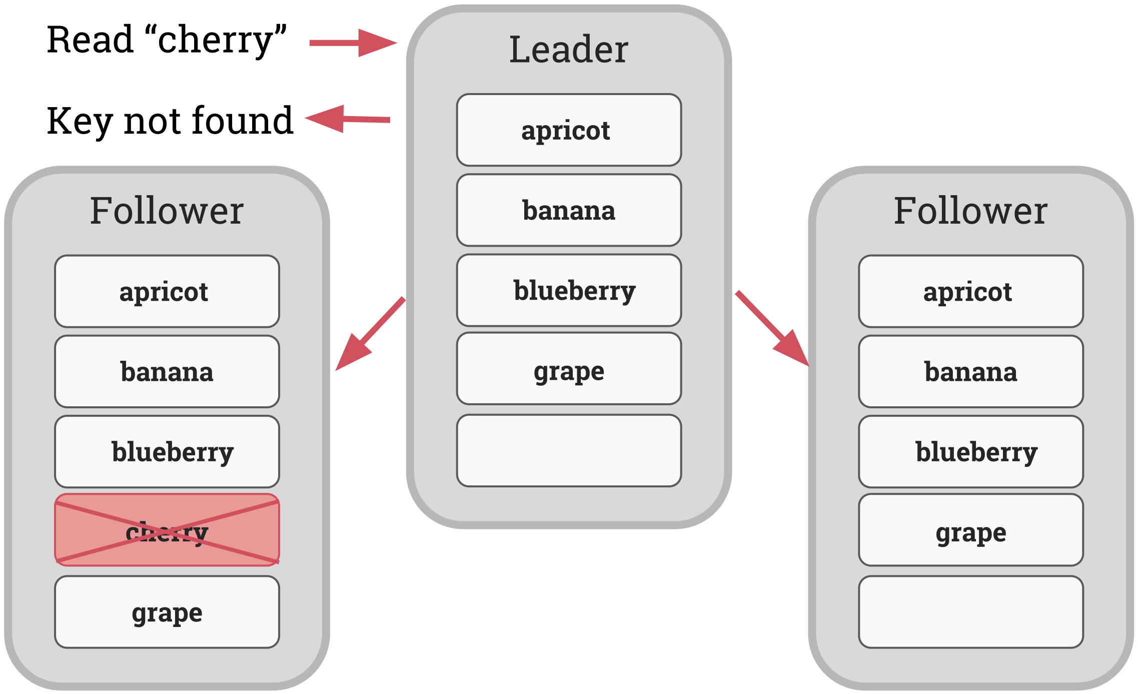


Consensus Replication: Failover



On failover, only data written to a quorum is considered present

Consensus Replication: Failover



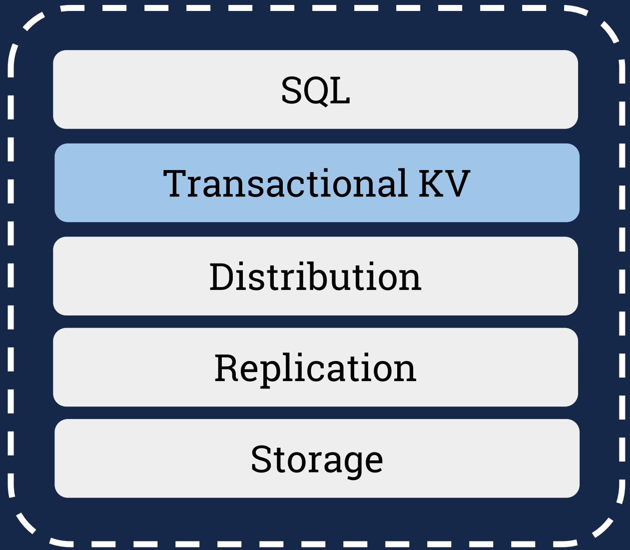
Consensus Replication

- **Raft** is our consensus protocol of choice.
- Run one consensus group per range of data
- Practical complications: member change, range splits, upgrades, scaling number of ranges

Consensus Replication

- Consensus provides “atomic” replication
    - But only for each range
- What about operations that hit multiple ranges?

Distributed Transactions



Distributed Transactions

- CockroachDB supports traditional ACID semantics
    - "All-or-nothing"
    - Defaults to the Serializable isolation level for true isolation
- Can't just copy single-node databases
    - Can't rely on write of a single disk block for atomicity
    - Distributed locking is quite expensive for most workloads

Distributed Transactions (CockroachDB)

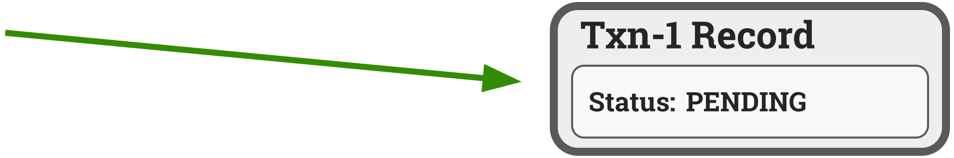
Need lower-level primitive to bootstrap atomic "commit" of transaction:

- Write to a range (i.e. a Raft consensus group)
- A transaction has an associated transaction record keyed by the transaction ID
- All writes during transaction are tagged with the transaction ID
- A transaction is atomically committed or aborted by updating the transaction record via a Raft write

Detect conflicts as they arise rather than locking

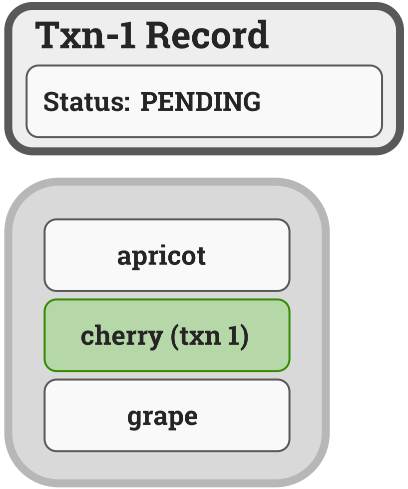
Distributed Transactions (CockroachDB)

1. Begin Txn 1



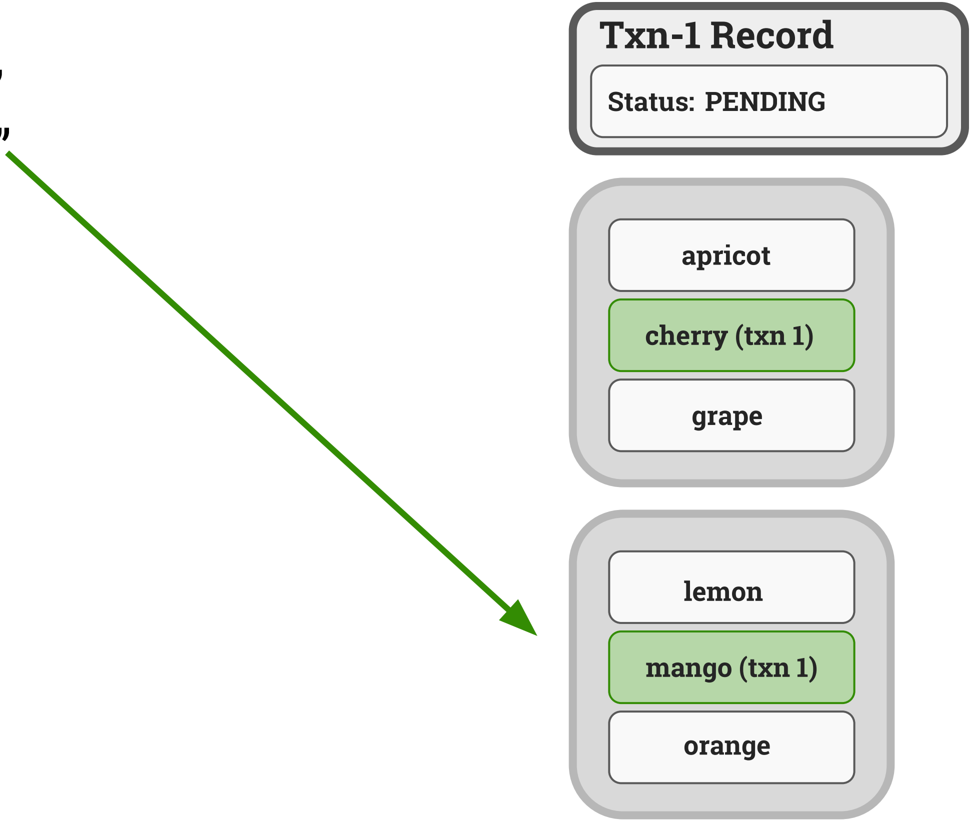
Distributed Transactions (CockroachDB)

1. Begin Txn 1
2. Put “cherry”



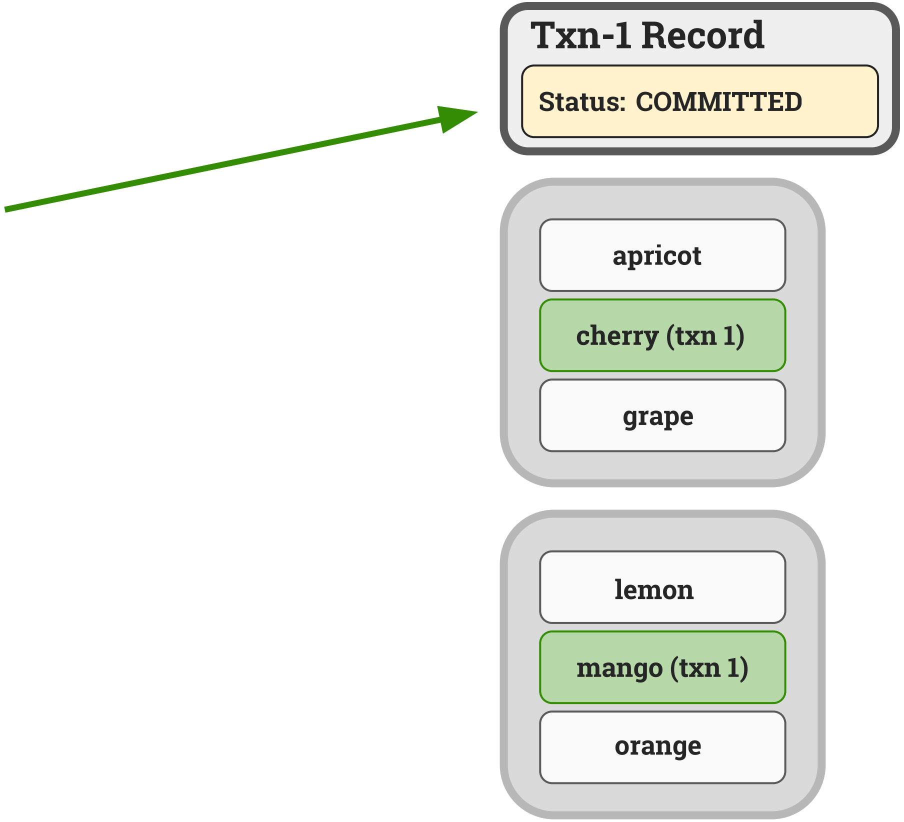
Distributed Transactions (CockroachDB)

1. Begin Txn 1
2. Put “cherry”
3. Put “mango”



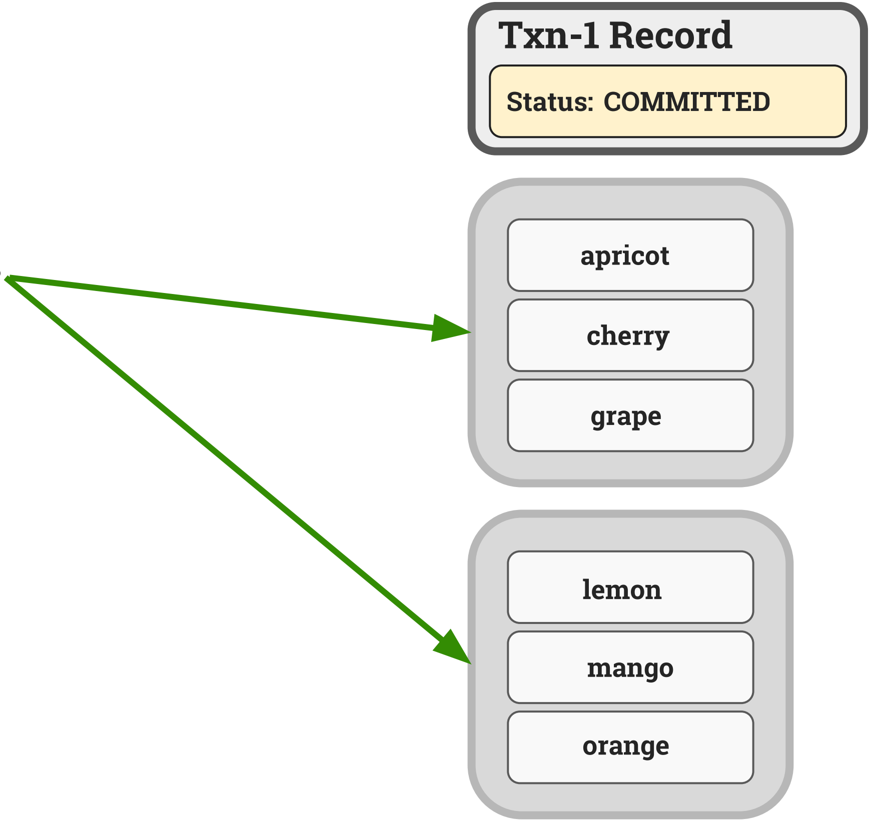
Distributed Transactions (CockroachDB)

1. Begin Txn 1
2. Put "cherry"
3. Put "mango"
4. Commit Txn 1



Distributed Transactions (CockroachDB)

1. Begin Txn 1
2. Put “cherry”
3. Put “mango”
4. Commit Txn 1
5. Clean up intents

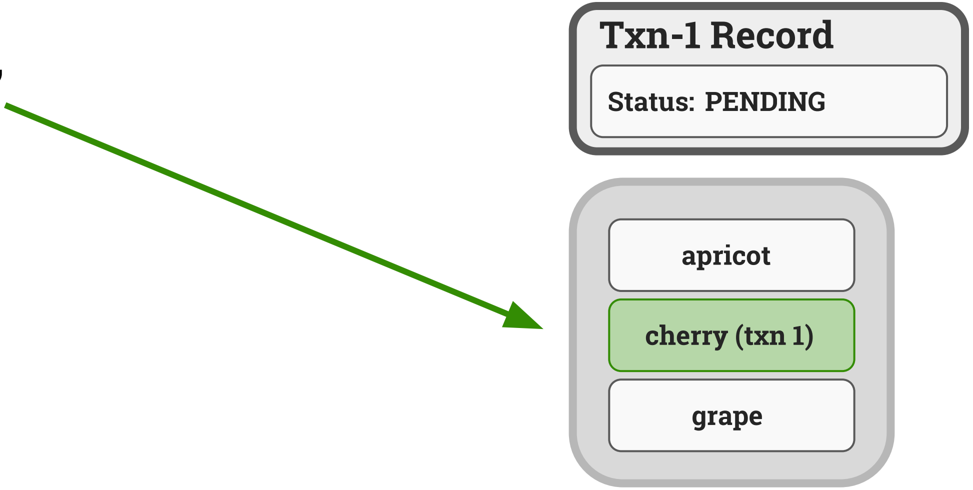


Distributed Transactions

- That's the happy case
- What about conflicting transactions?

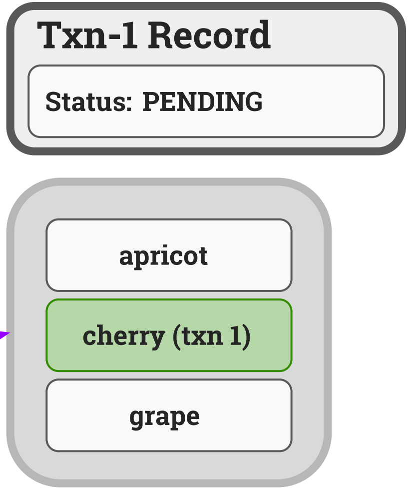
Distributed Transactions (read conflict)

1. Begin Txn 1
2. Put “cherry”



Distributed Transactions (read conflict)

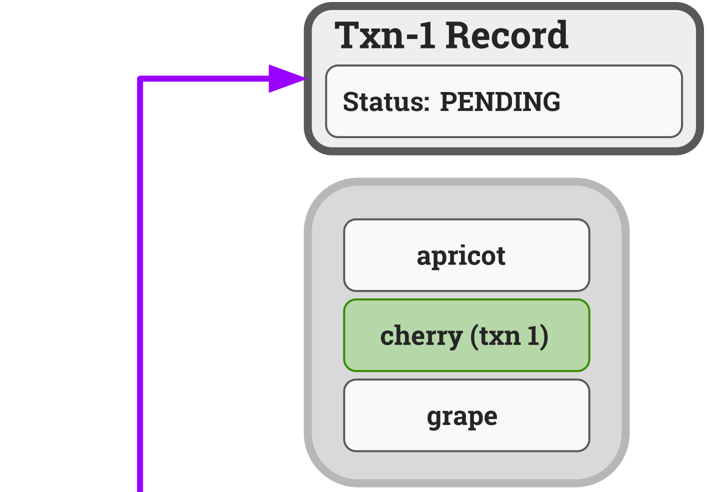
1. Begin Txn 1
2. Put “cherry”



1. Begin Txn 2
2. Get “cherry”

Distributed Transactions (read conflict)

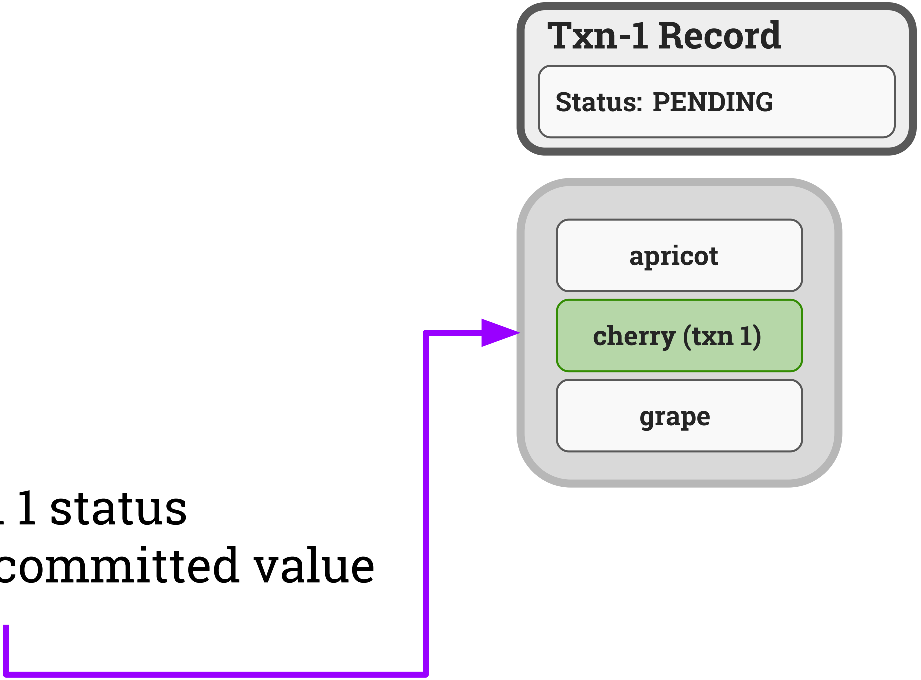
1. Begin Txn 1
2. Put “cherry”



1. Begin Txn 2
2. Get “cherry”
    - Check txn 1 status

Distributed Transactions (read conflict)

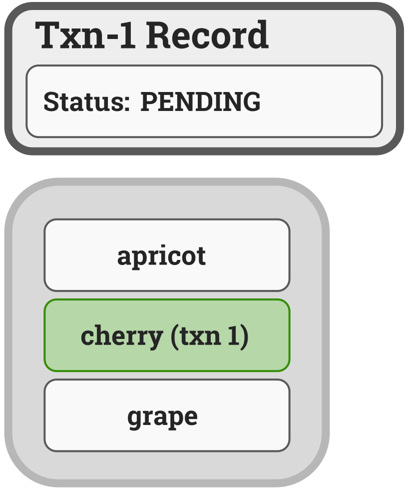
1. Begin Txn 1
2. Put “cherry”



1. Begin Txn 2
2. Get “cherry”
   - Check txn 1 status
   - Ignore uncommitted value

Distributed Transactions (read conflict)

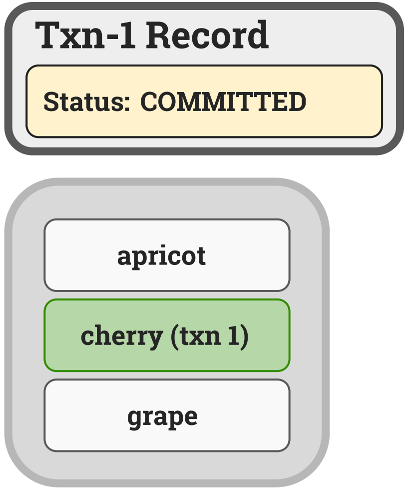
1. Begin Txn 1
2. Put “cherry”



1. Begin Txn 2
2. Get “cherry”
    - Check txn 1 status
    - Ignore uncommitted value
3. Commit

Distributed Transactions (read conflict)

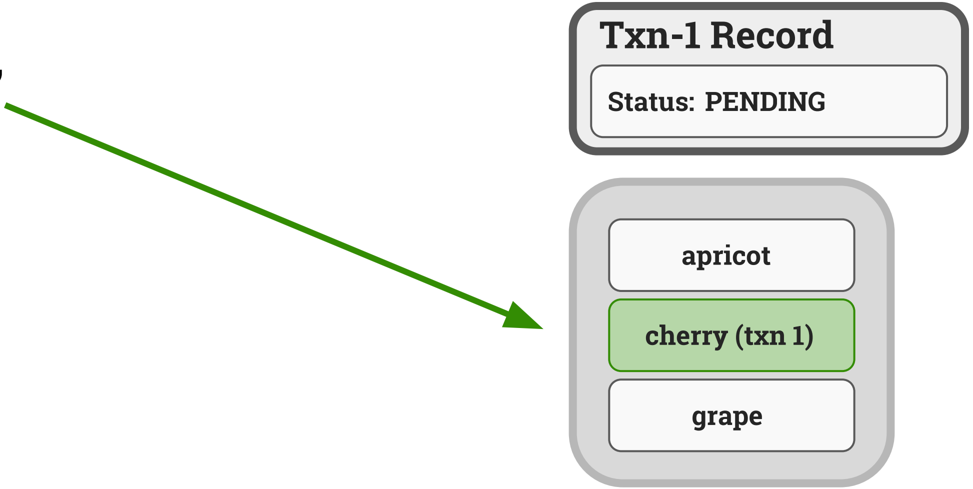
1. Begin Txn 1
2. Put “cherry”
3. Commit (potentially at later timestamp)



1. Begin Txn 2
2. Get “cherry”
    - Check txn 1 status
    - Ignore uncommitted value
3. Commit

Distributed Transactions (write conflict)

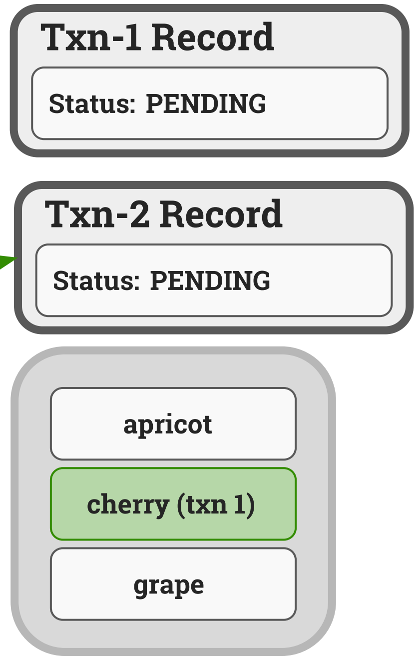
1. Begin Txn 1
2. Put “cherry”



Distributed Transactions (write conflict)

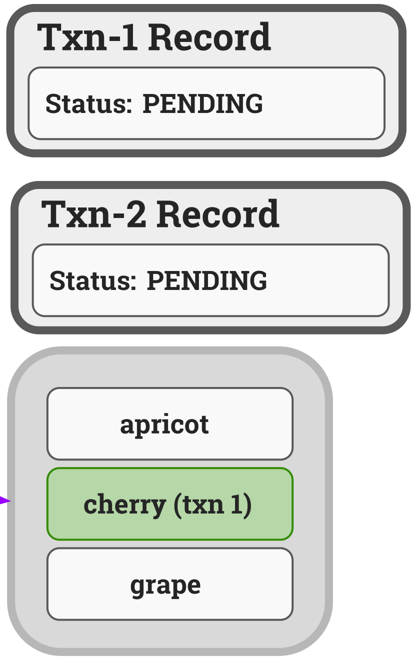
1. Begin Txn 1
2. Put “cherry”

1. Begin Txn 2



Distributed Transactions (write conflict)

1. Begin Txn 1
2. Put "cherry"

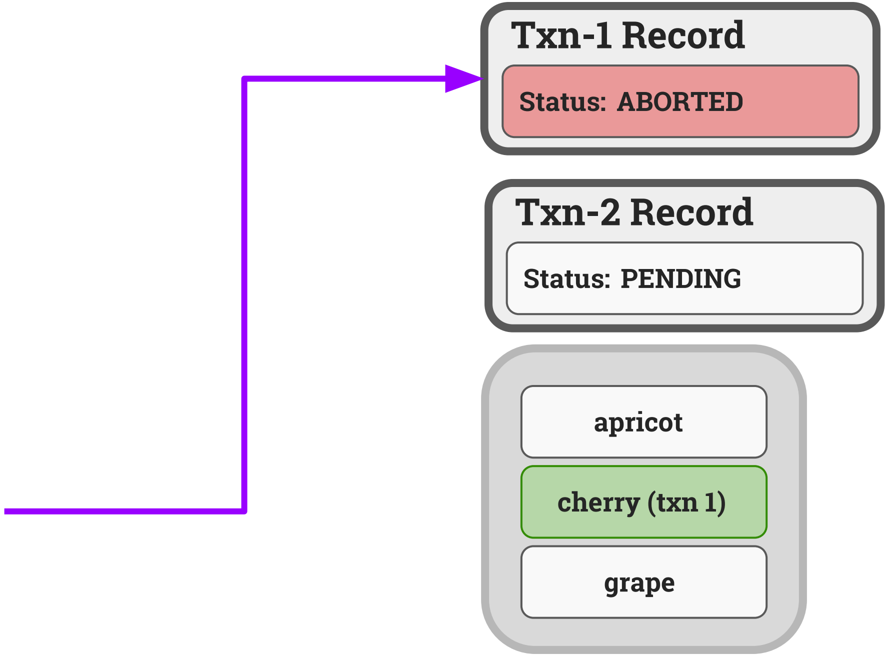


1. Begin Txn 2
2. Put "cherry"

Distributed Transactions (write conflict)

1. Begin Txn 1
2. Put “cherry”

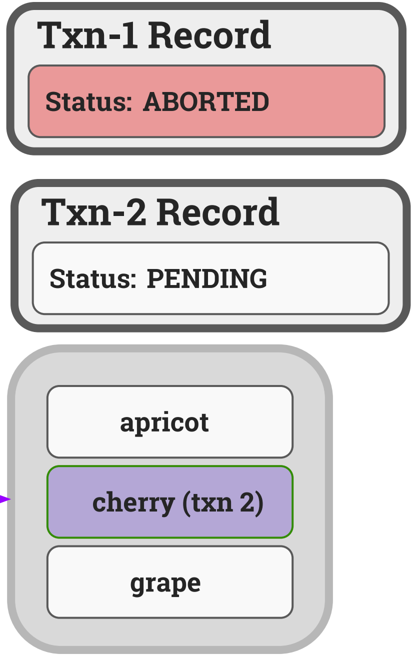
1. Begin Txn 2
2. Put “cherry”
    - Push Txn-1



Distributed Transactions (write conflict)

1. Begin Txn 1
2. Put “cherry”

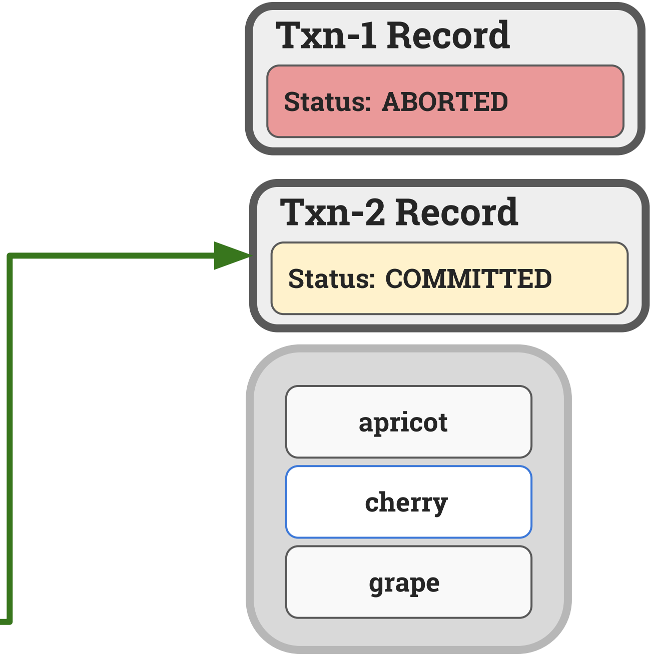
1. Begin Txn 2
2. Put “cherry”
    - Push Txn-1
    - Update intent



Distributed Transactions (write conflict)

1. Begin Txn 1
2. Put “cherry”

1. Begin Txn 2
2. Put “cherry”
   - Push Txn 1
   - Update intent
3. Commit Txn 2



Distributed Transactions (CockroachDB)

- Transaction atomicity is bootstrapped on top of Raft atomicity
- Isolation, MVCC, other conflicts: mostly ignored in this description
  - More details on the Cockroach Labs blog[1]

[1] https://www.cockroachlabs.com/blog/serializable-lockless-distributed-isolation-cockroachdb/

Summary

- Building a SQL database for cloud environments
  - Distribute data to support SQL workloads and fault-tolerance
  - Replicate with Raft as foundation of atomicity
  - Distributed transactions built on top

Deploying CockroachDB

Deploying CockroachDB

- Single Binary
  - UI
  - Client
  - Tools
- Symmetric nodes
- Auto-balancing
- Self-healing

Deploying CockroachDB

- As normal processes:

```
machine-1 $ cockroach start
machine-2 $ cockroach start --join=machine-1:26257
machine-3 $ cockroach start --join=machine-1:26257
```

- In Docker:

```
$ docker run --rm cockroachdb/cockroach:beta-20170323 start
```

- On Kubernetes

```
$ kubectl create -f https://tiny.cc/cockroachdb-statefulset
    OR
$ helm install stable/cockroachdb
```

Thank You

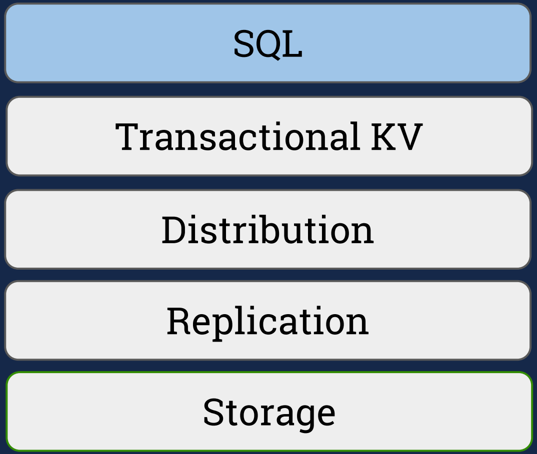
CockroachLabs.com
github.com/cockroachdb/cockroach

alex@cockroachlabs.com / github.com/a-robinson

Cockroach LABS

Extra slides

SQL Logical Data Storage



CockroachDB: KV Primitives

```
Get(key)

Put(key, value)

ConditionalPut(key, value, expValue)

Scan(startKey, endKey)

Del(key)
```

CockroachDB: Row Storage

- All tables have a primary key
- One key/value pair per column
  - Column families for efficiency
- Key anatomy:

```
/<table>/<index>/<key>/<column>
```

CockroachDB: Example Table

```
CREATE TABLE test (
    id      INTEGER PRIMARY KEY,
    name    VARCHAR,
    price   FLOAT,
);
```

CockroachDB: Key Anatomy

```
INSERT INTO test VALUES (1, “ball”, 2.22);
```

Key: /<table>/<index>/<key>/<column>	Value
/test/primary/**1**/name	“ball”
/test/primary/**1**/price	2.22

CockroachDB: Key Anatomy

```
INSERT INTO test VALUES (1, “ball”, 2.22);
INSERT INTO test VALUES (2, “glove”, 3.33);
```

Key: /<table>/<index>/<key>/<column>	**Value**
/test/primary/**1**/name	“ball”
/test/primary/**1**/price	2.22
/test/primary/**2**/name	“glove”
/test/primary/**2**/price	3.33

CockroachDB: Logical Data Storage

- Keys and values are byte strings
- Unique indexes
- Non-unique indexes

CockroachDB: Unique Indexes

```
CREATE UNIQUE INDEX bar ON test (name);
INSERT INTO test VALUES (1, “ball”, 2.22);
```

Key: /<table>/<index>/<key>	**Value**
/test/bar/”ball”	1

CockroachDB: Unique Indexes

```
CREATE UNIQUE INDEX bar ON test (name);
INSERT INTO test VALUES (1, "ball", 2.22);
INSERT INTO test VALUES (2, "glove", 3.33);
```

Key: /<table>/<index>/<key>	**Value**
/test/bar/"ball"	**1**
/test/bar/"glove"	**2**

CockroachDB: Unique Indexes

```
CREATE UNIQUE INDEX bar ON test (name);
INSERT INTO test VALUES (1, “ball”, 2.22);
INSERT INTO test VALUES (2, “glove”, 3.33);
INSERT INTO test VALUES (3, “glove”, 4.44);
```

Key: /<table>/<index>/<key>	**Value**
/test/bar/”ball”	1
/test/bar/”glove”	2
~~/test/bar/”glove”~~	~~3~~

CockroachDB: Non-Unique Indexes

```
CREATE INDEX foo ON test (name);
INSERT INTO test VALUES (1, “ball”, 2.22);
```

Key: /<table>/<index>/<key>/<pkey>	Value
/test/foo/”ball”/ **1**	∅

CockroachDB: Non-Unique Indexes

```
CREATE INDEX foo ON test (name);
INSERT INTO test VALUES (1, “ball”, 2.22);
INSERT INTO test VALUES (2, “glove”, 3.33);
```

Key: /<table>/<index>/<key>/<pkey>	**Value**
/test/foo/”ball”/ **1**	∅
/test/foo/”glove”/ **2**	∅

CockroachDB: Non-Unique Indexes

```
CREATE INDEX foo ON test (name);
INSERT INTO test VALUES (1, "ball", 2.22);
INSERT INTO test VALUES (2, "glove", 3.33);
INSERT INTO test VALUES (3, "glove", 4.44);
```

Key: /<table>/<index>/<key>/<pkey>	**Value**
/test/foo/"ball"/**1**	∅
/test/foo/"glove"/**2**	∅
/test/foo/"glove"/**3**	∅