Building a Cloud-Native SQL Database

CockroachDB: Scalable, Survivable, Consistent, SQL

presented by Alex Robinson / Member of the Technical Staff

What is “Cloud-Native”?

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

What isn't "Cloud-Native"?

- Can't scale out (monolithic)
- Not designed to recover from failures
  - Single point of failure
  - Can't spread across availability zones
  - Manual recovery/failover
- Tied to a particular platform

Cloud-native databases

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

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)

Why should you care about consistency or transactions?

Strong Consistency

- Reasoning about eventual consistency is hard
  - Wastes developer time
  - Causes bugs
- Avoid stale reads or data loss on failover
- Enables true SQL support

Transactions

“We believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions”

-Authors of Google’s Spanner paper

Why is this hard?

Why is getting the best of both worlds so hard?

- Fundamentally, coordination is very difficult
  - Especially when time is involved
  - Building a database is hard enough as it is
- It’s tough to pair consistency with performance and scalability

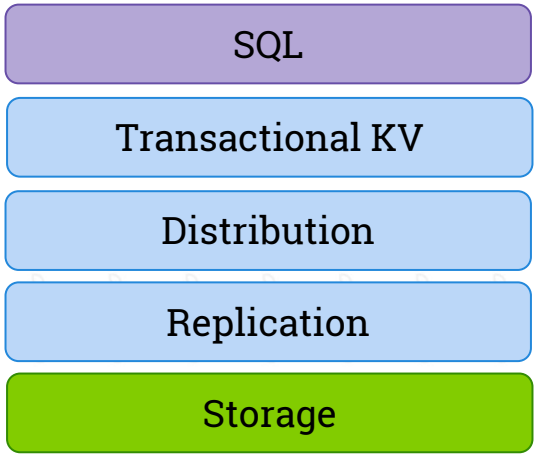
How does CockroachDB do it?

How does CockroachDB do it?

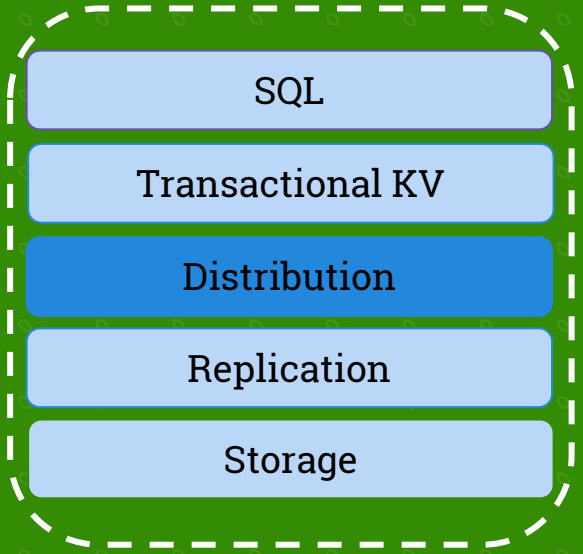
1. Data distribution and replication
2. Consensus protocol (Raft)
3. Distributed transaction model

Architecture (high-level)

Abstraction stack:



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

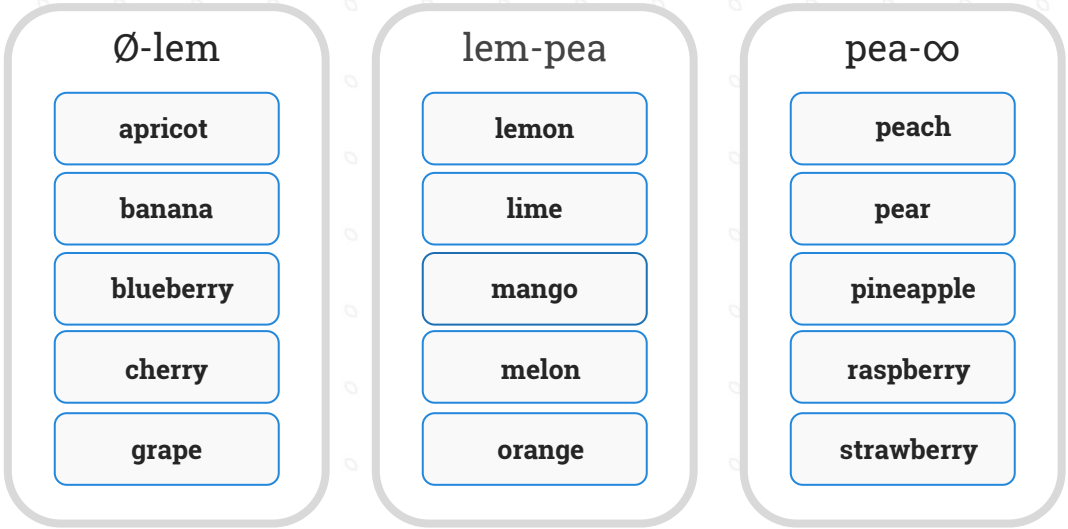
Data Distribution: Order-Preserving

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

- Pro: efficient scans
- Con: requires additional indexing

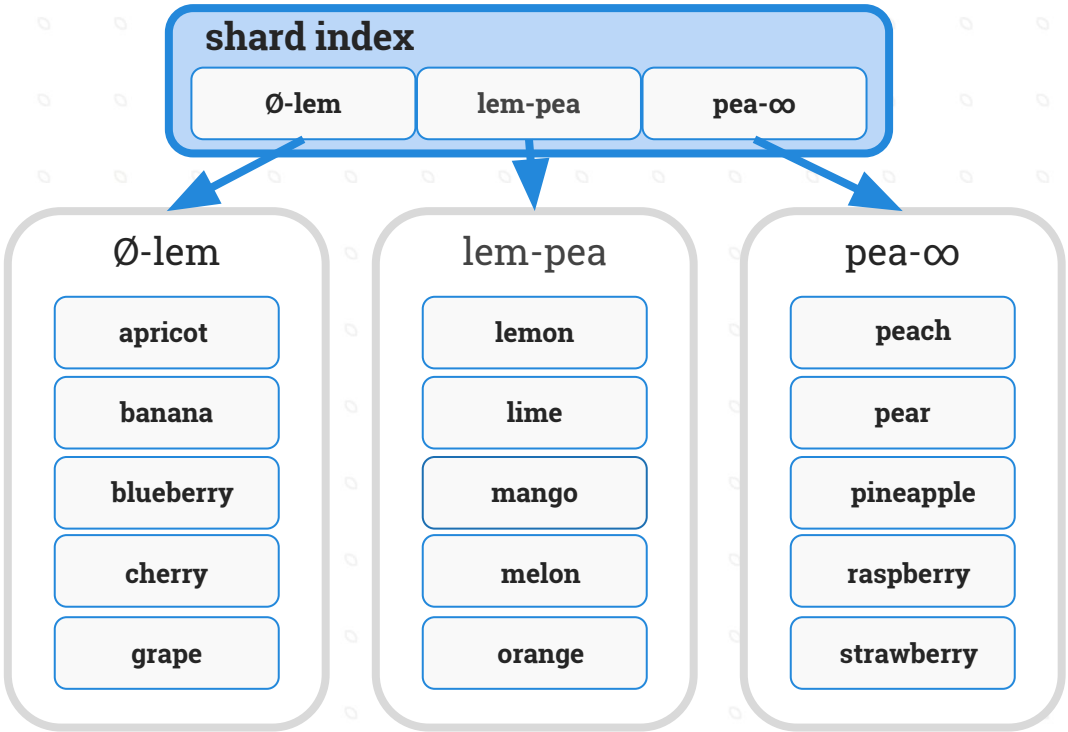
Data Distribution: Order-Preserving

Each shard contains a contiguous segment of the key space



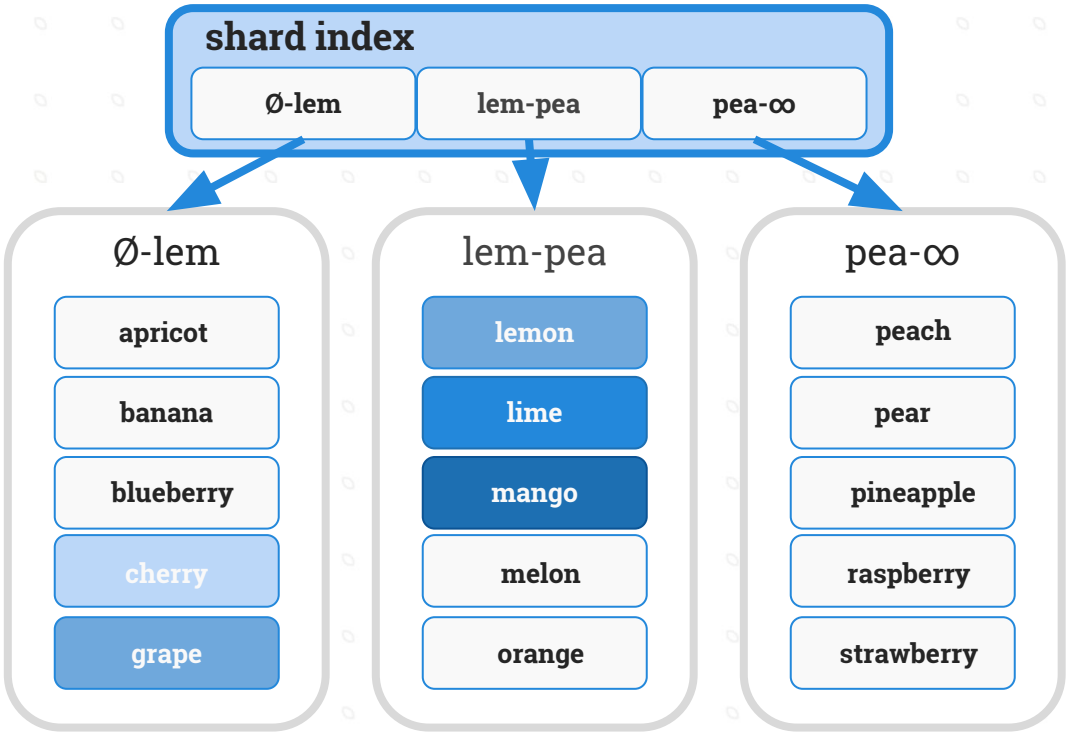
Data Distribution: Order-Preserving

We need an indexing structure to locate a range



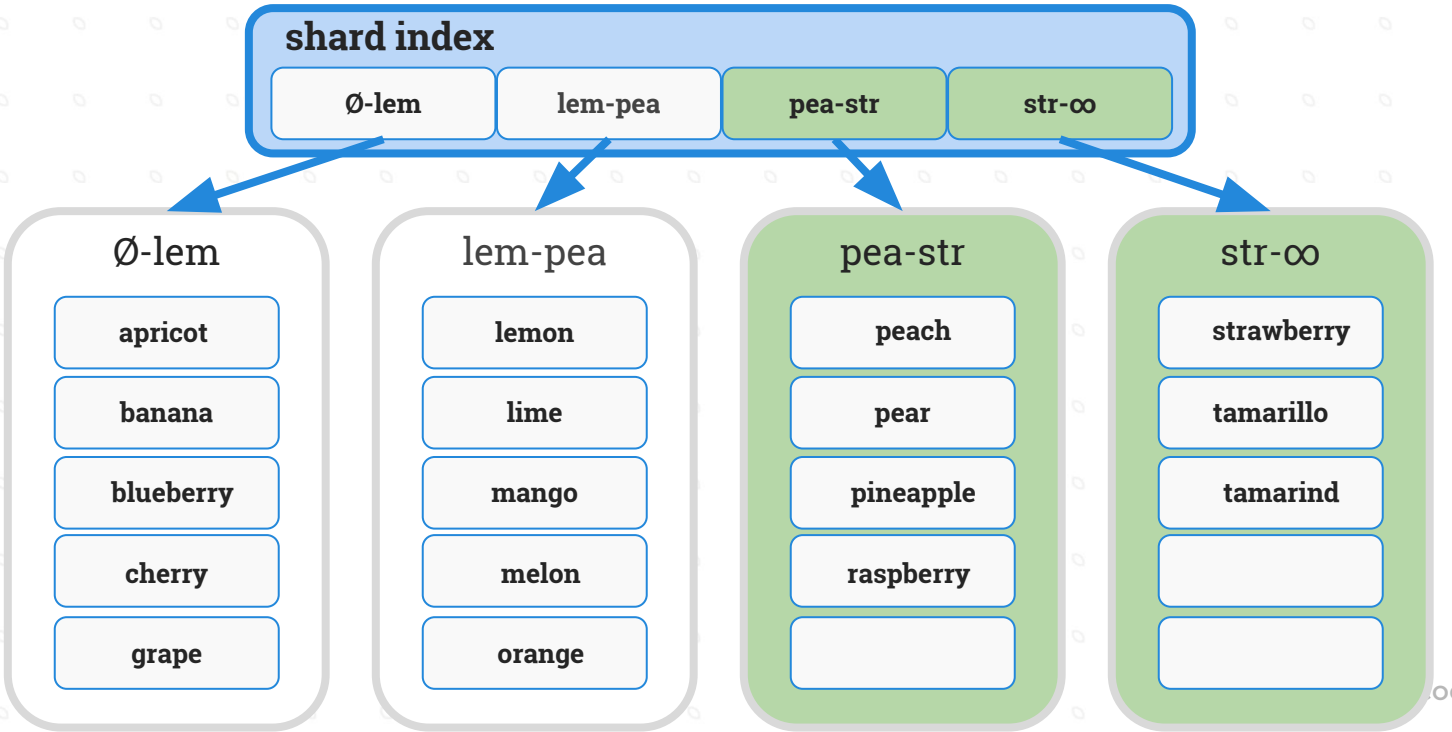
Data Distribution: Order-Preserving

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

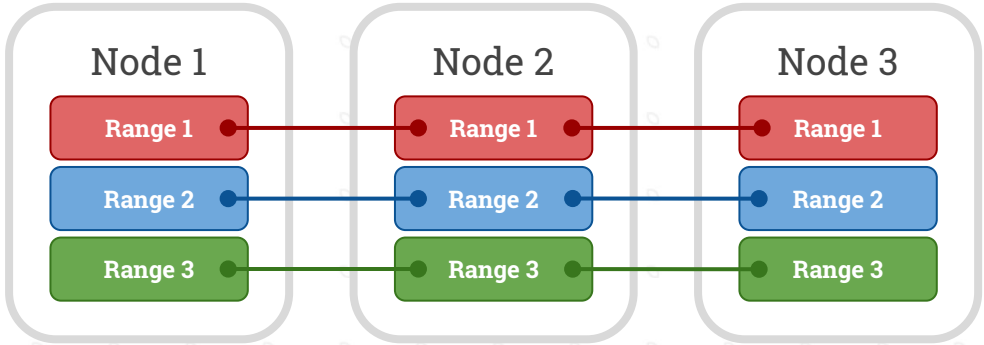


Data Distribution: Order-Preserving

Split when a shard is too large

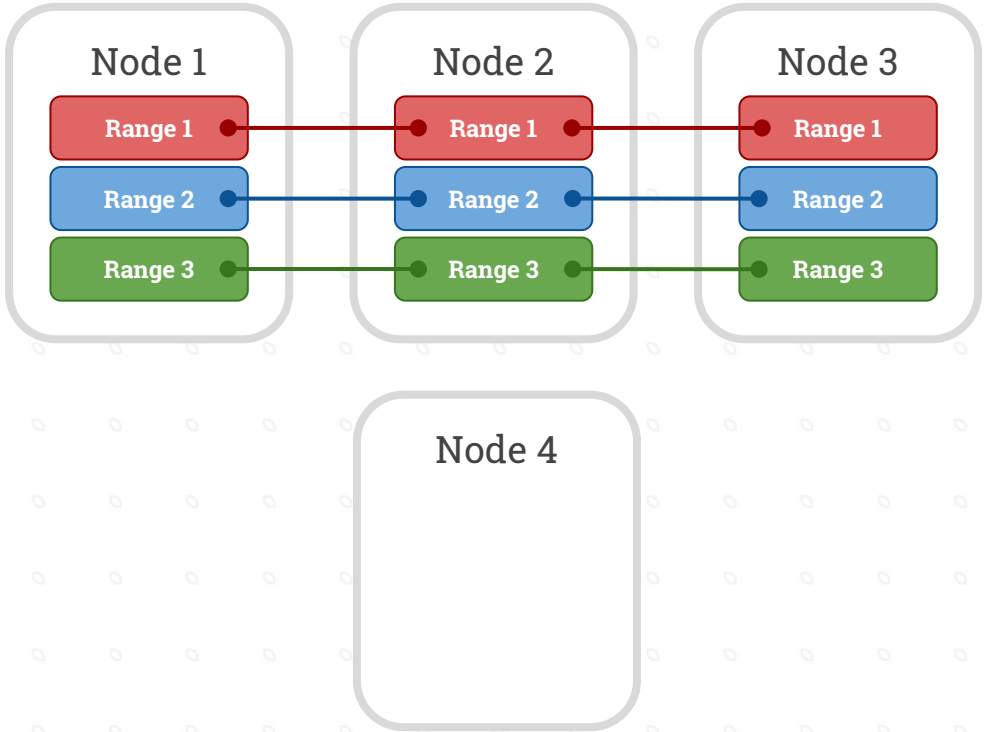


Data Distribution: Placement



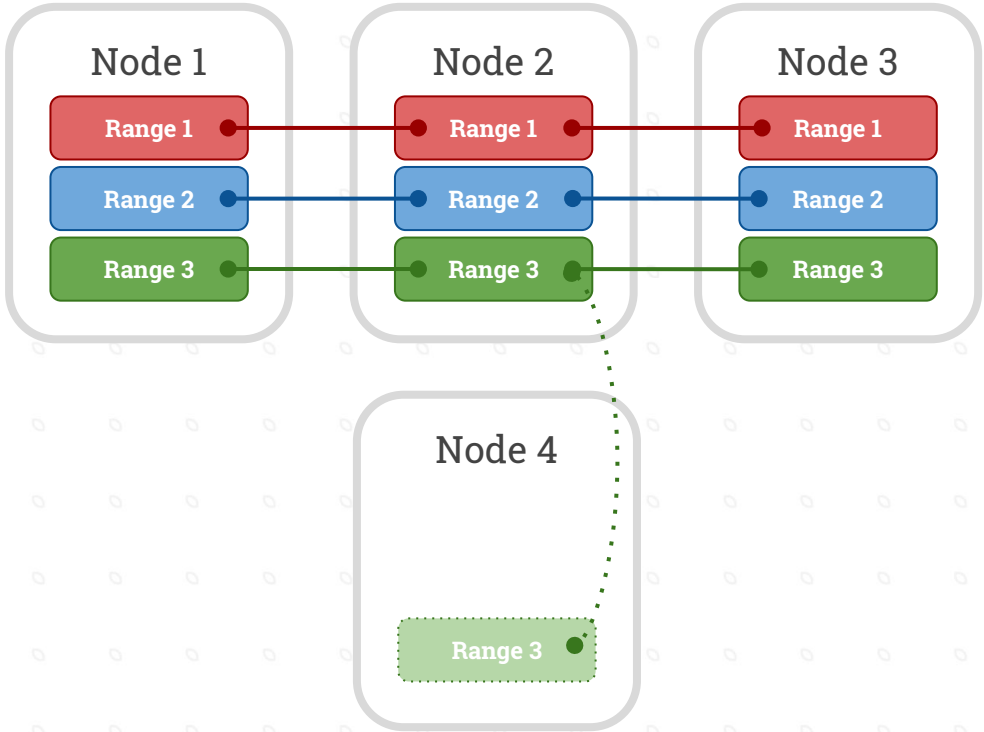
Each range is replicated to three or more nodes

Data Distribution: Rebalancing



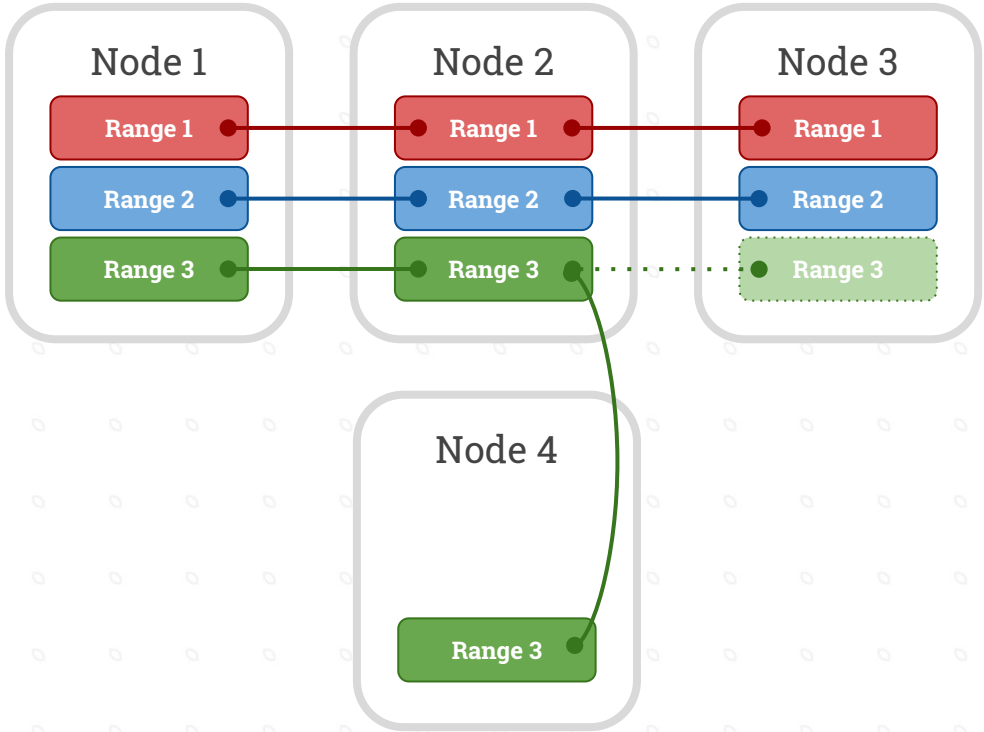
Adding a new (empty) node

Data Distribution: Rebalancing



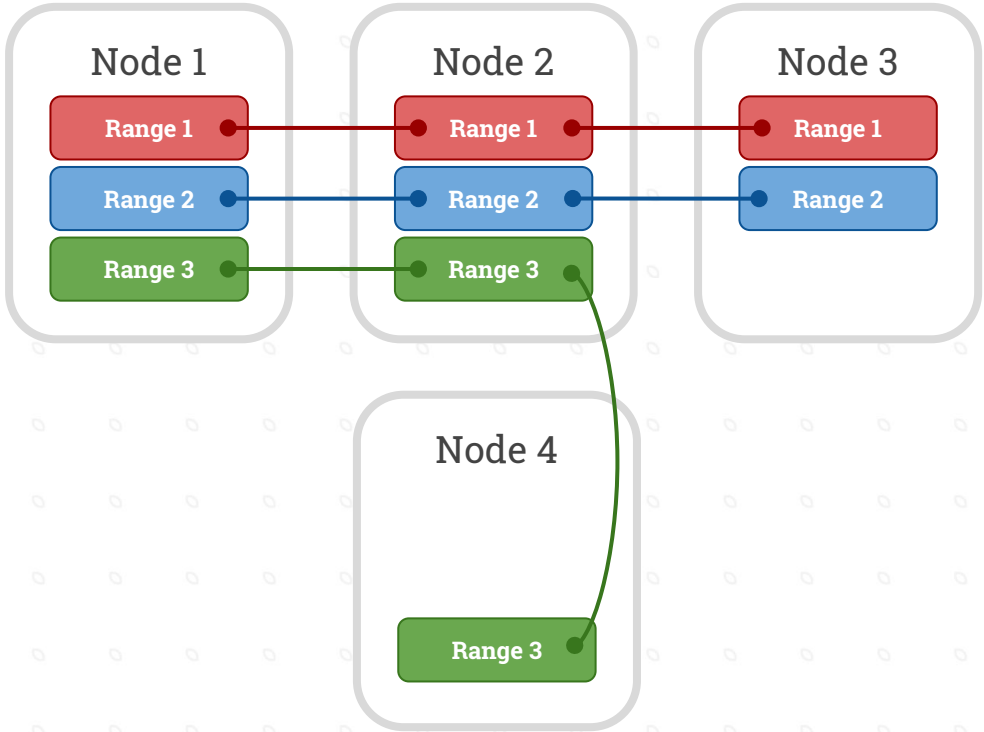
A new replica is allocated, data is copied.

Data Distribution: Rebalancing



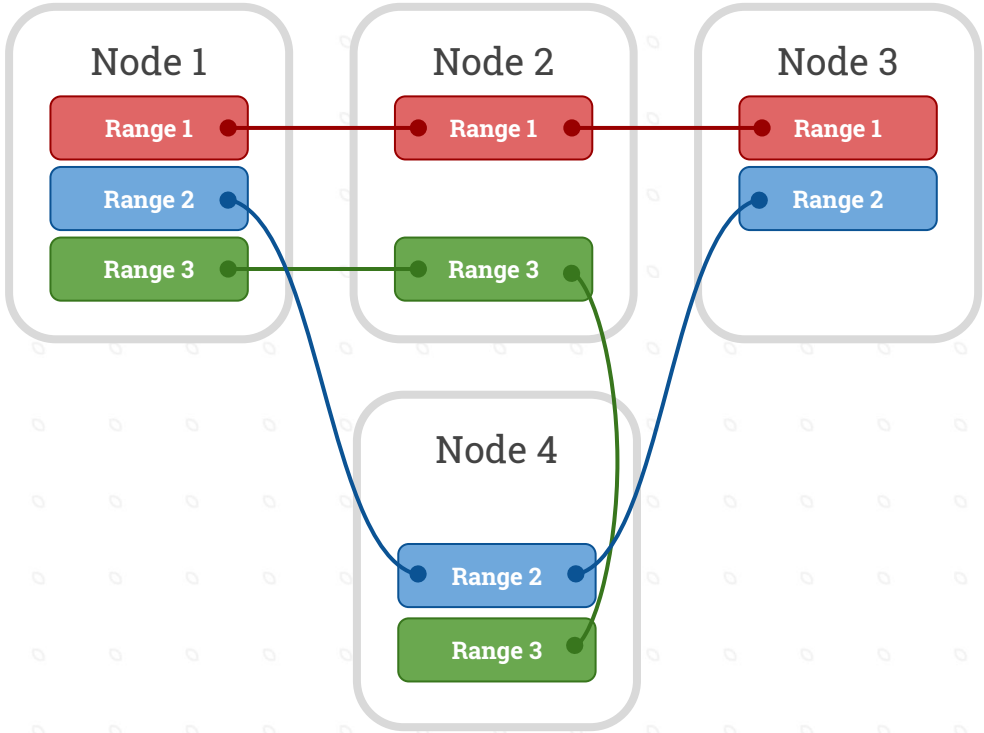
The new replica is made live, replacing another.

Data Distribution: Rebalancing



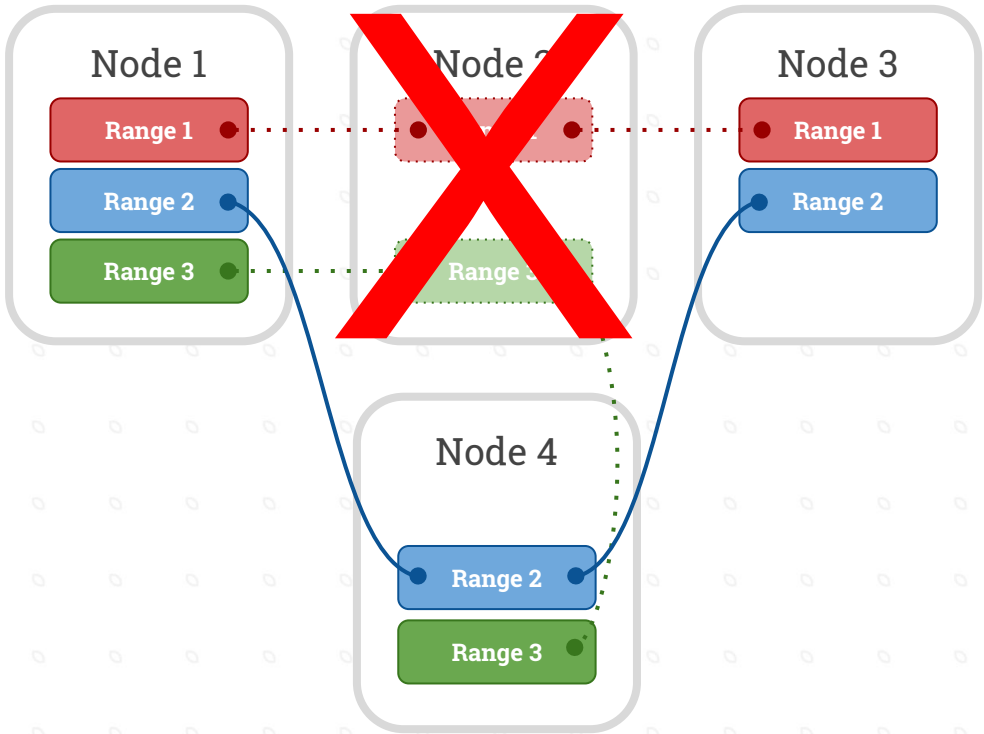
The old (inactive) replica is deleted.

Data Distribution: Rebalancing



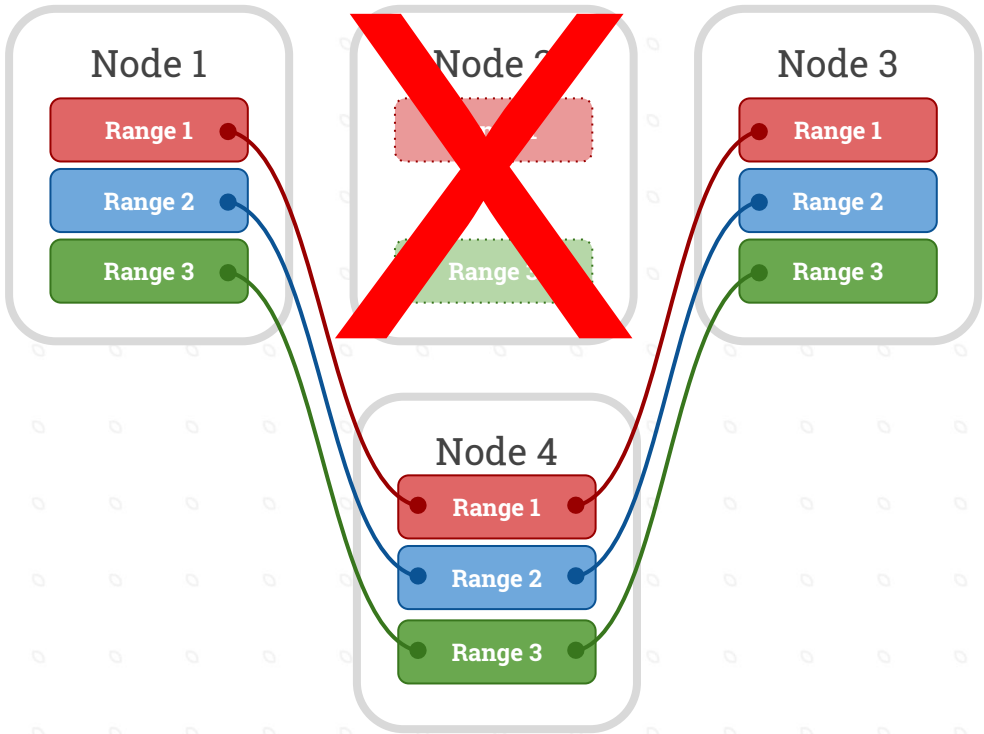
Process continues until nodes are balanced.

Data Distribution: Recovery



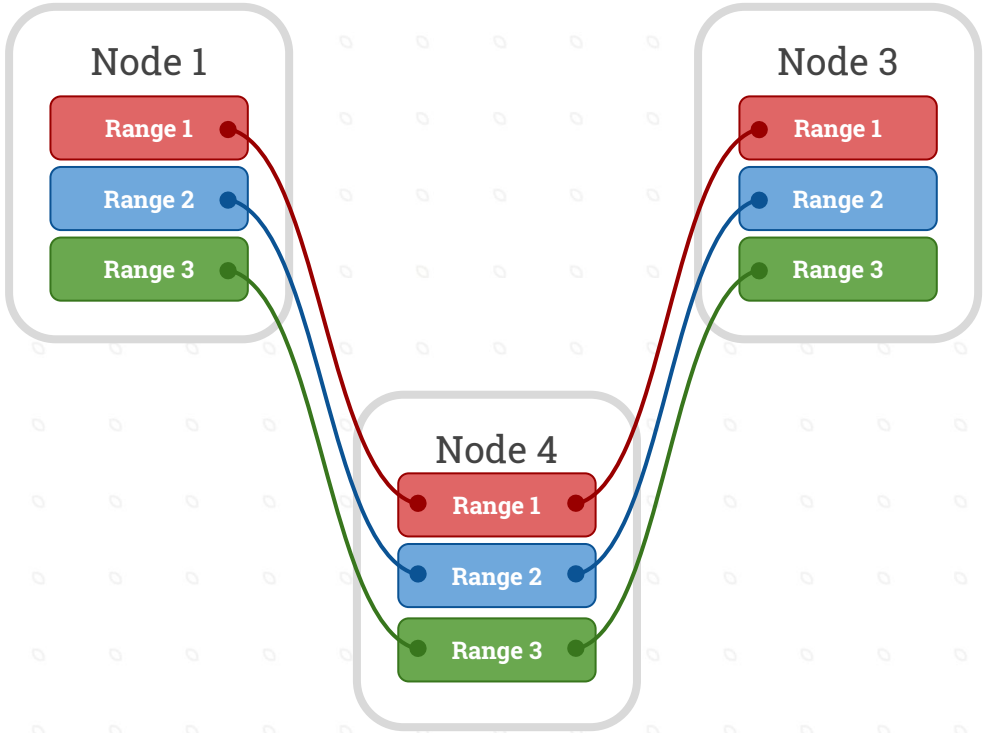
Losing a node causes recovery of its replicas.

Data Distribution: Recovery



A new replica gets created on an existing node.

Data Distribution: Recovery

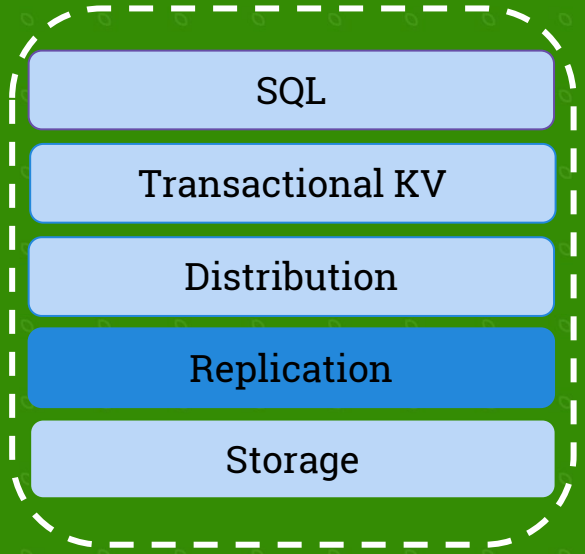


Once at full replication, the old replicas are forgotten.

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

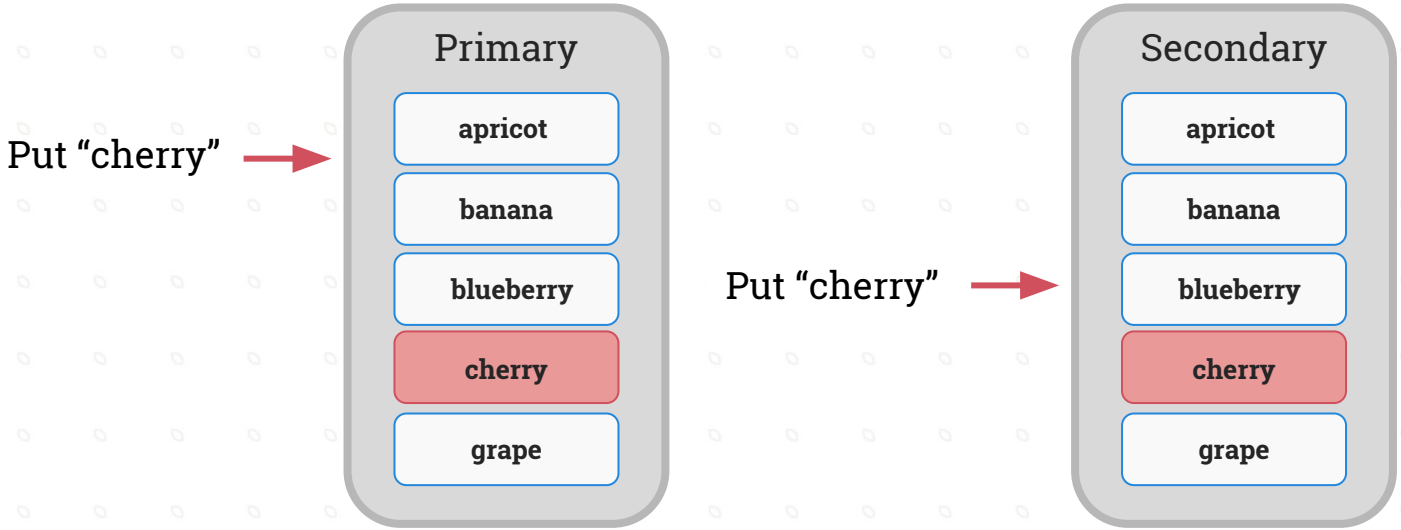
Production DBs must survive machine failure

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 => low availability
- Primary to secondary failover requires care
  - Third-party needs to arbitrate primary

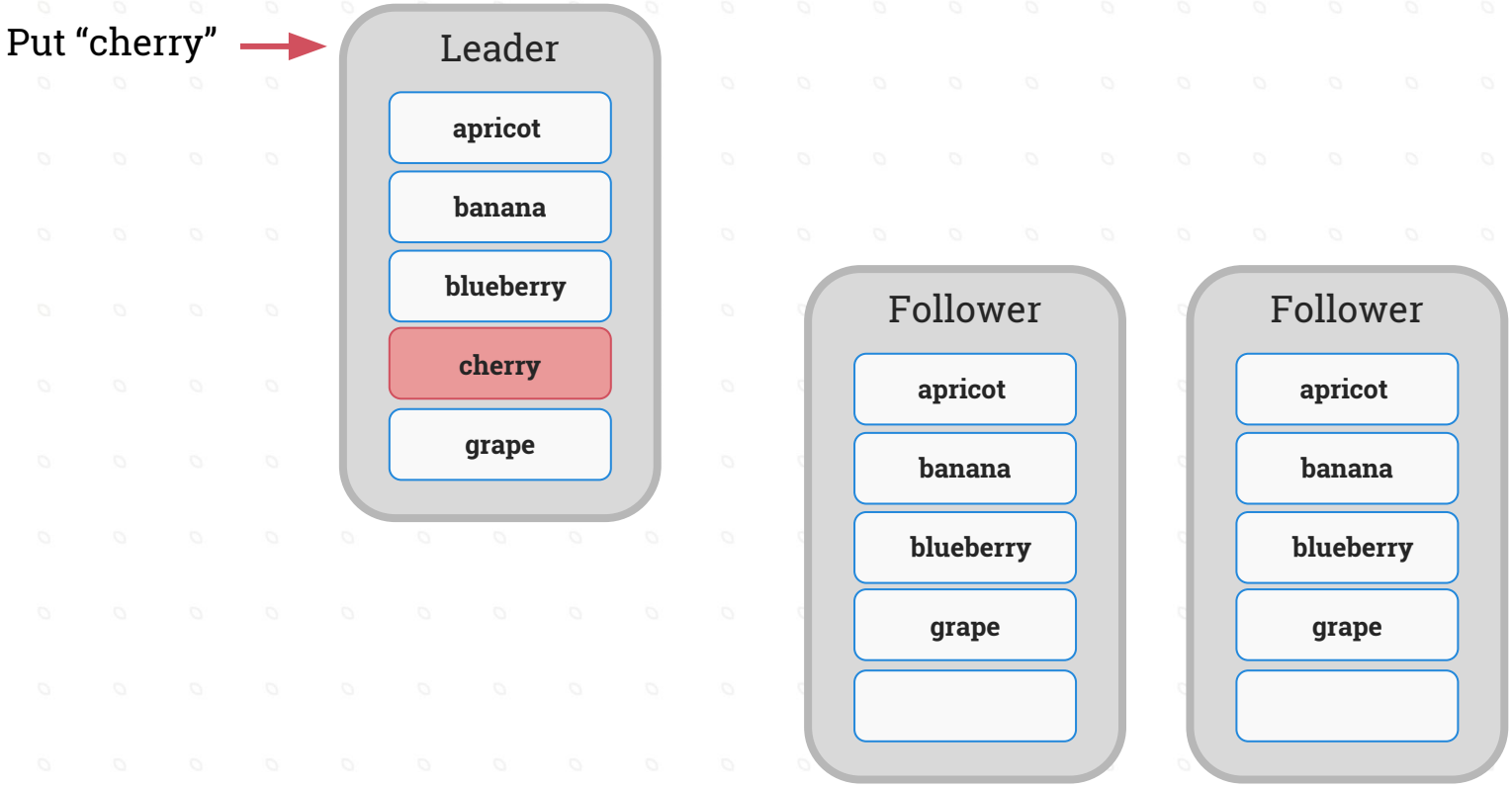
Consensus Replication

Replicate to N (often N=3) nodes

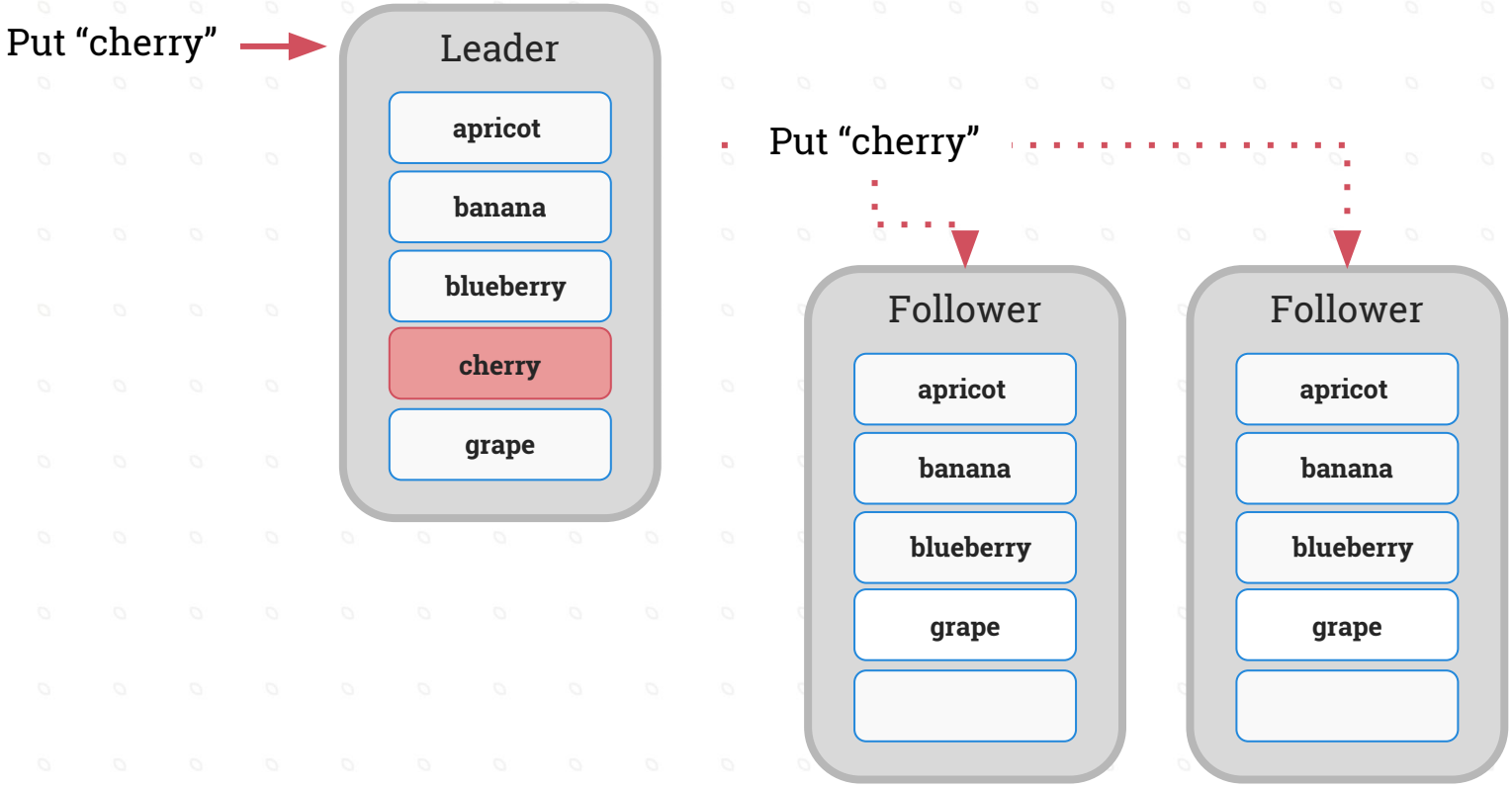
- Commit happens when a quorum have written the data

CockroachDB/Etcd/Spanner/Aurora/...

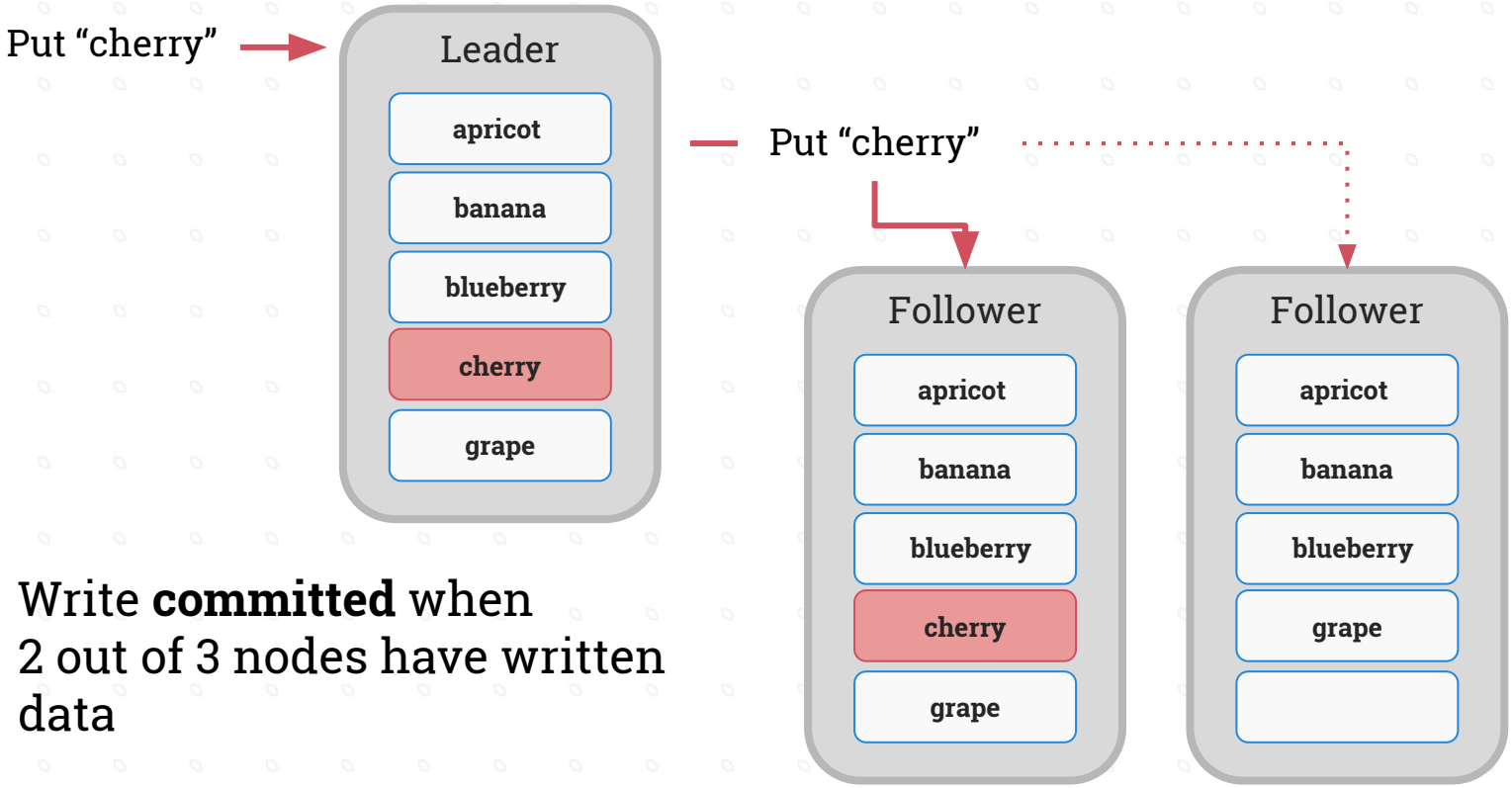
Consensus Replication



Consensus Replication

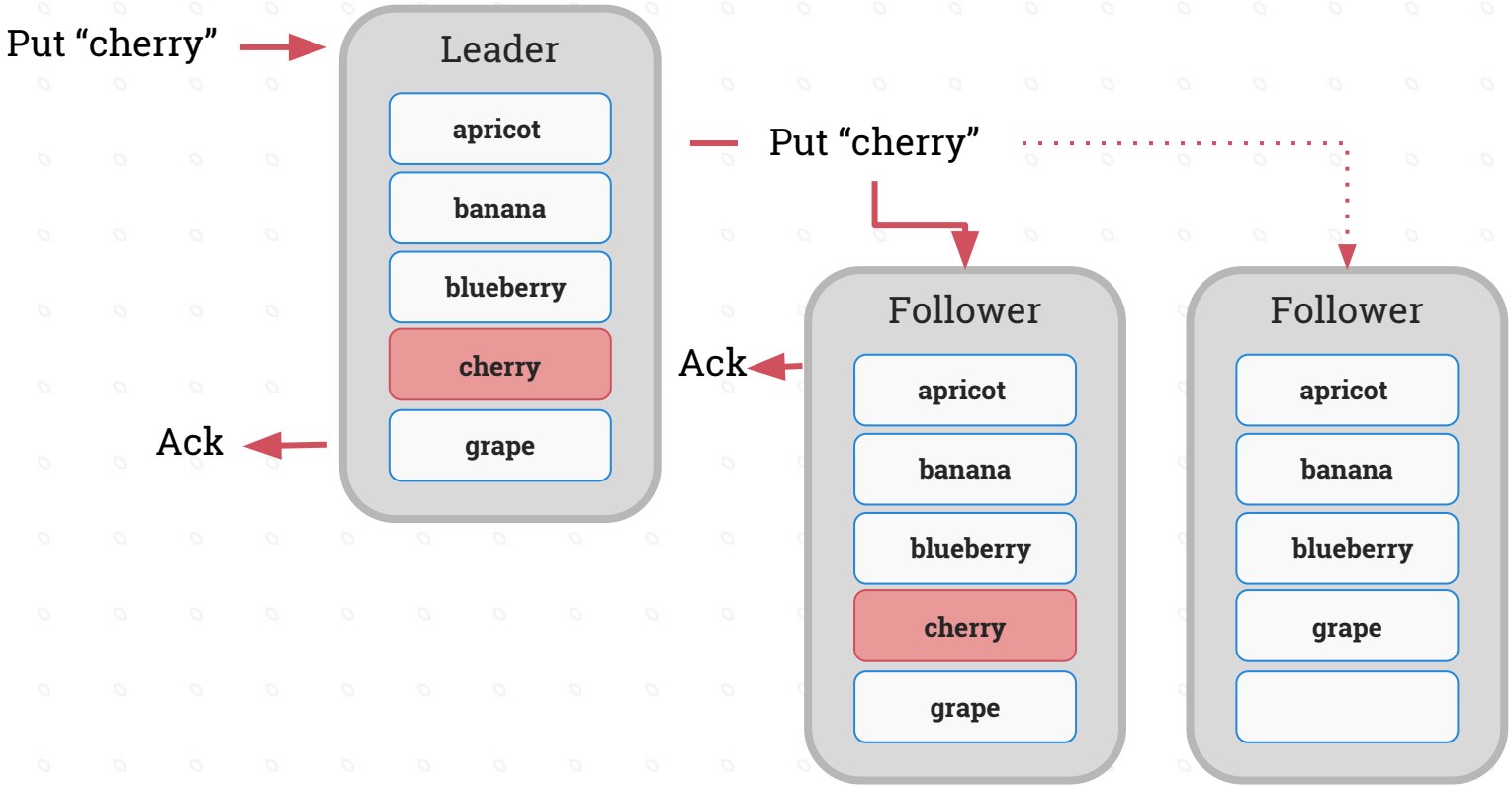


Consensus Replication



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

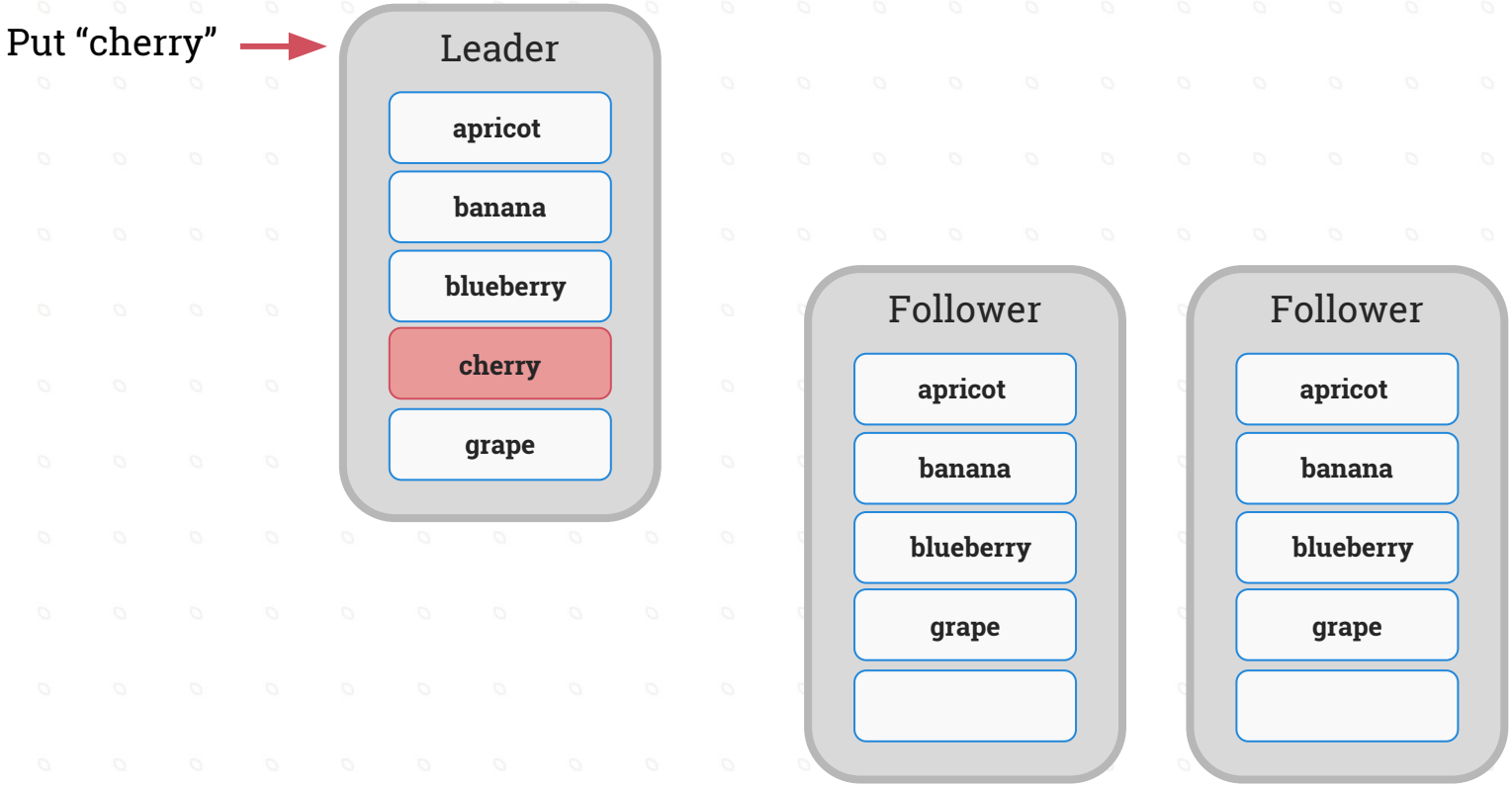
Consensus Replication



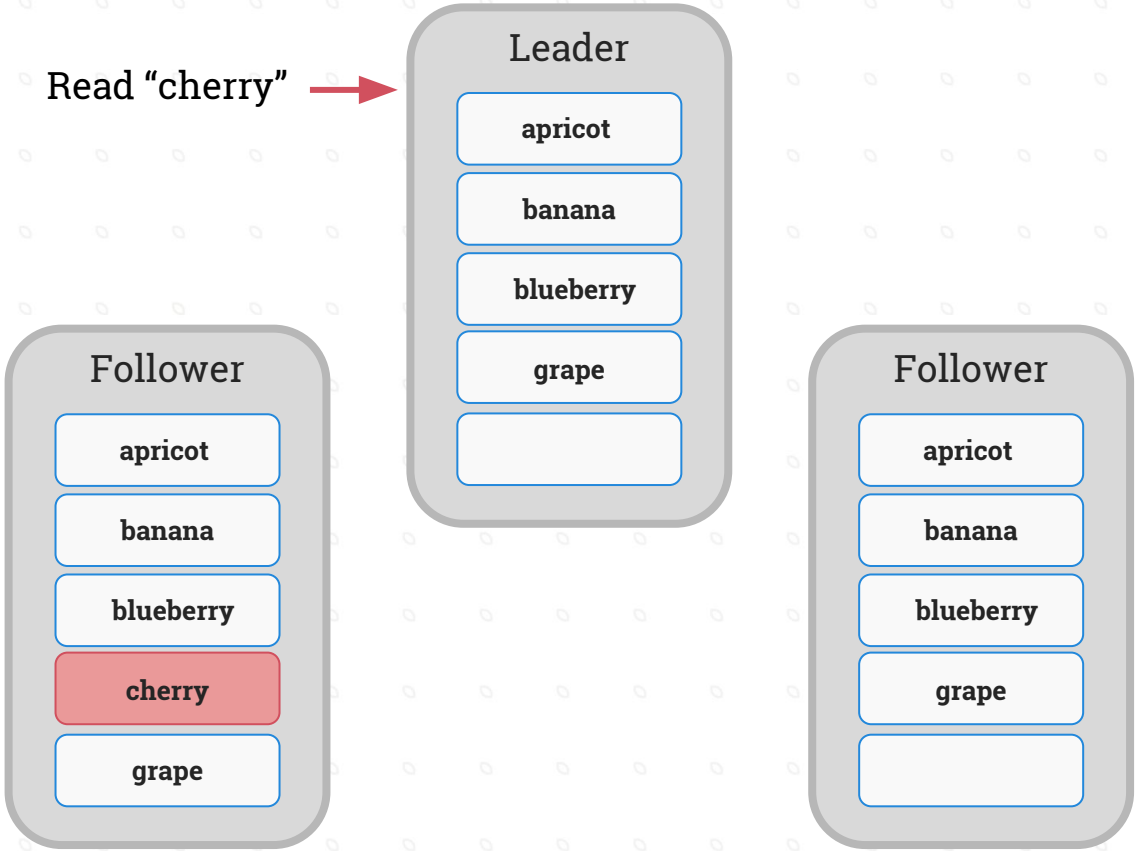
Consensus Replication

What happens during failover?

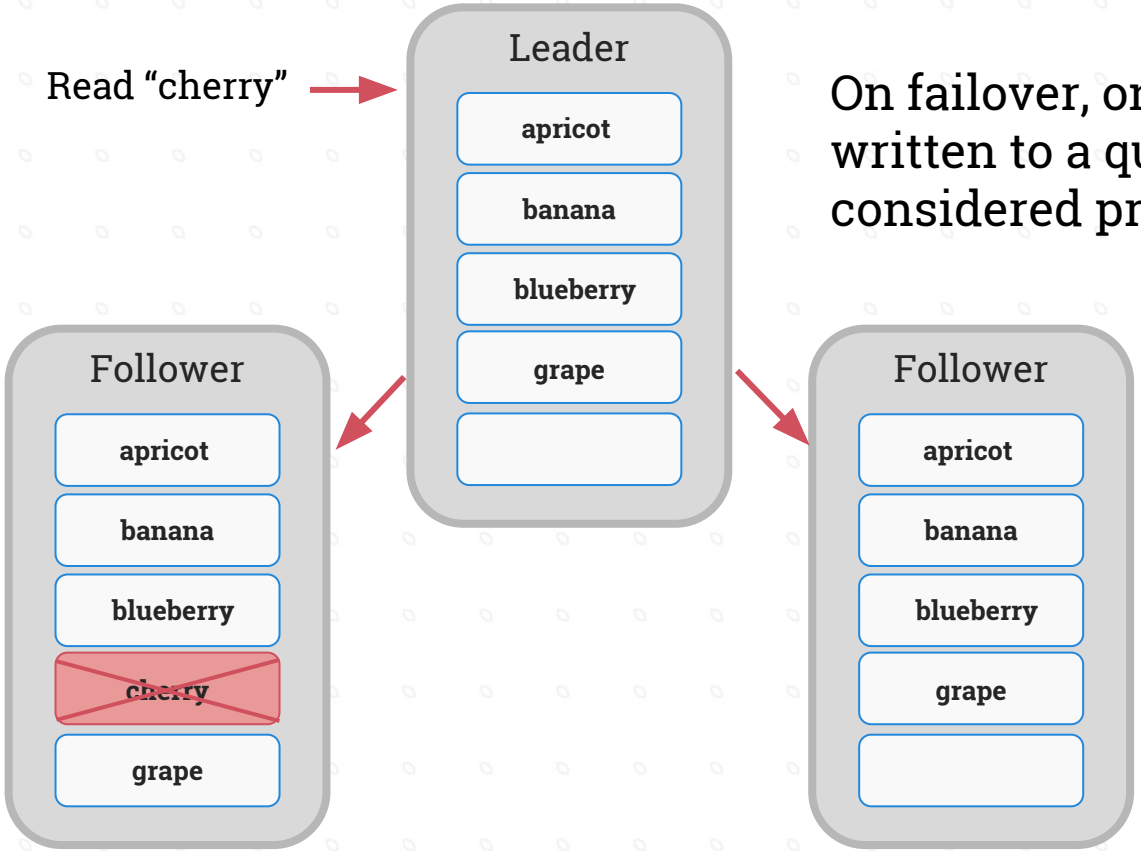
Consensus Replication: Failover



Consensus Replication: Failover

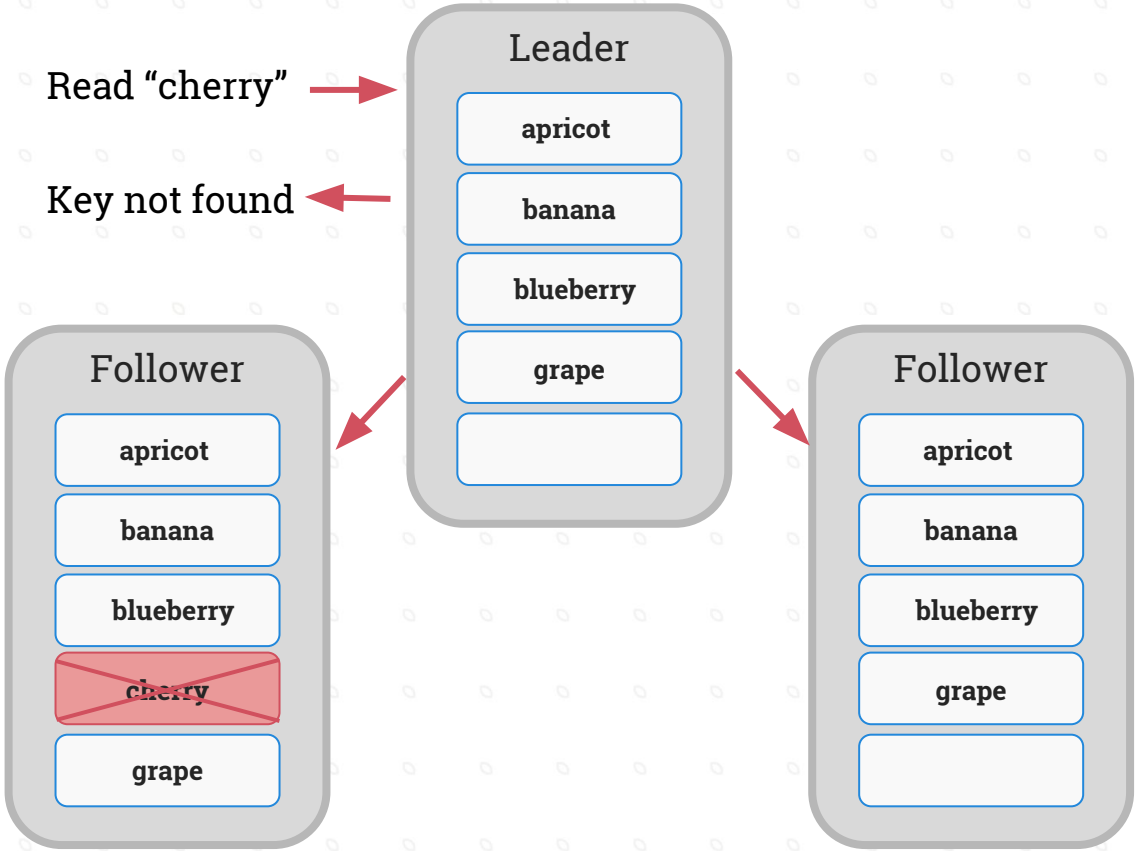


Consensus Replication: Failover



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

Consensus Replication: Failover



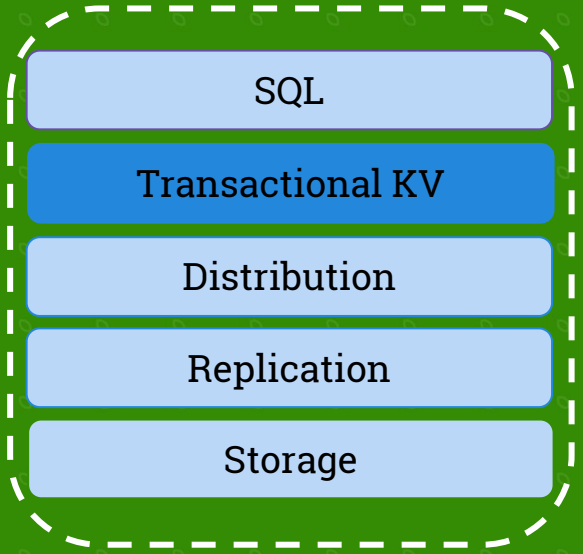
Consensus Replication

- **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



Transactions

- CockroachDB supports traditional ACID semantics
  - “All-or-nothing”
  - Defaults to the Serializable isolation level

Let’s first look at a basic transaction implementation in a traditional single-node DB

Single-node DB Transactions

Atomicity and durability are achieved by bootstrapping off a lower-level atomic/durable primitive: log writes

- Log entry written prior to mutations being applied to the database, tagged with transaction ID
- “Commit” log entry marks the transaction as committed

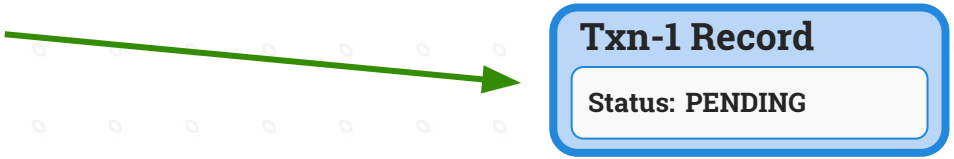
Distributed Transactions (CockroachDB)

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
- A transaction is atomically committed or aborted by updating the transaction record via a Raft write

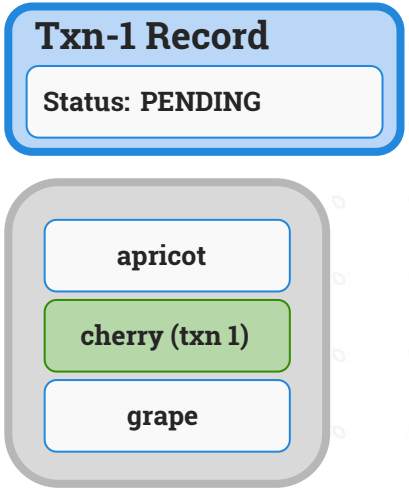
Distributed Transactions (CockroachDB)

1. Begin Txn 1



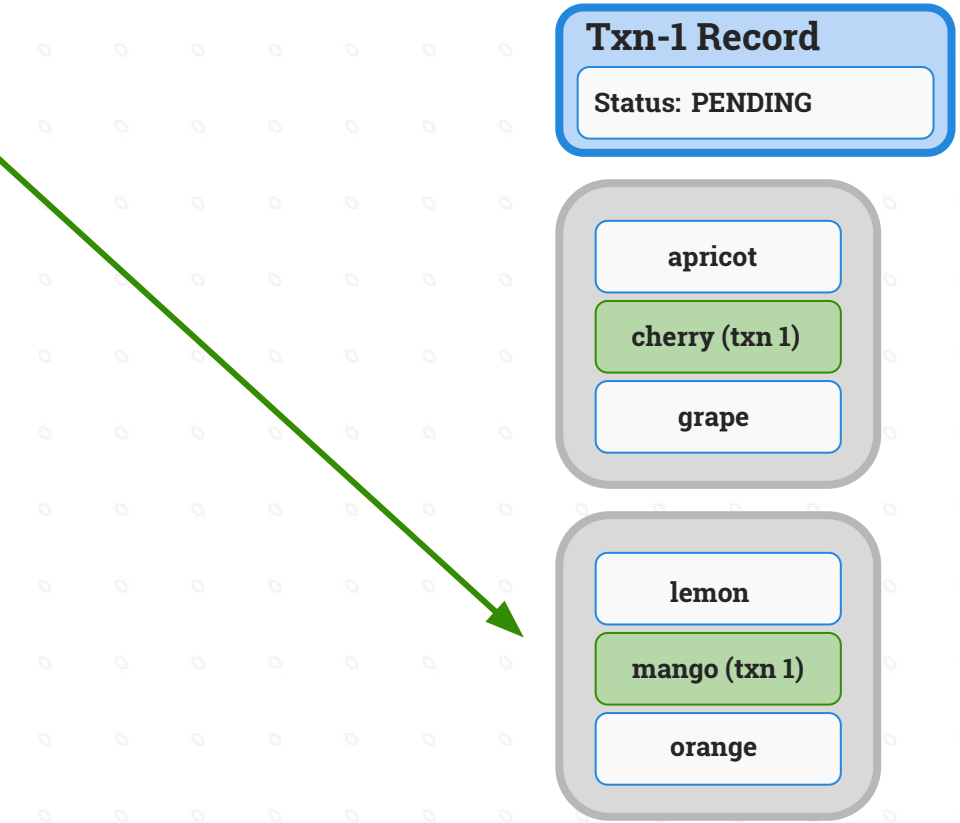
Distributed Transactions (CockroachDB)

1. Begin Txn 1
2. Put “cherry”



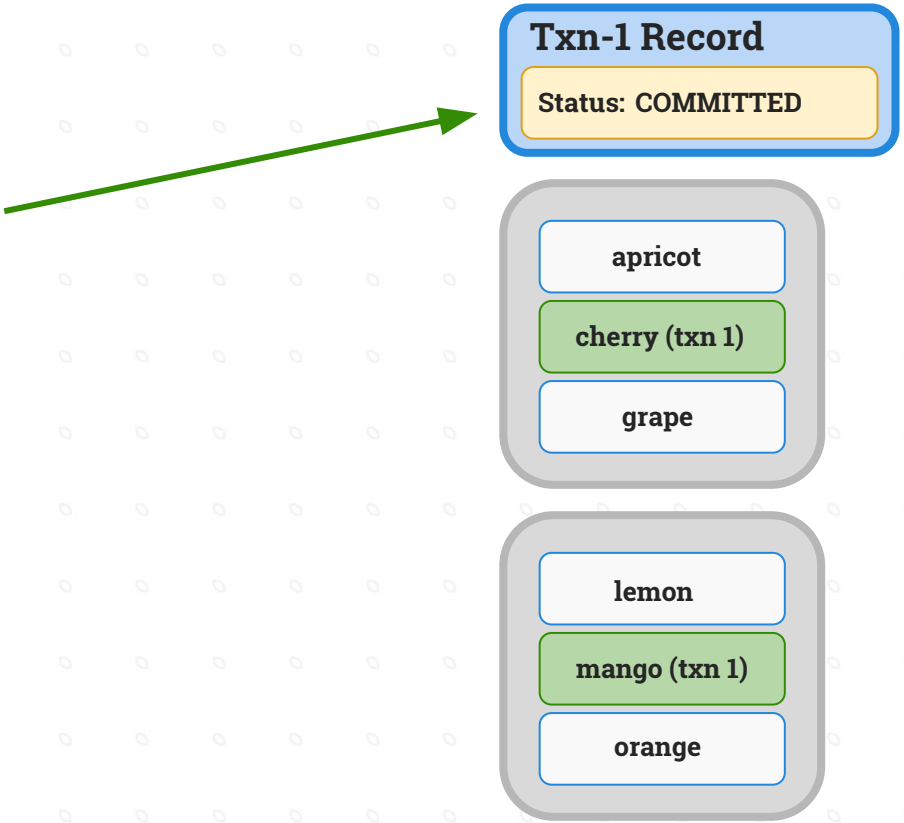
Distributed Transactions (CockroachDB)

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



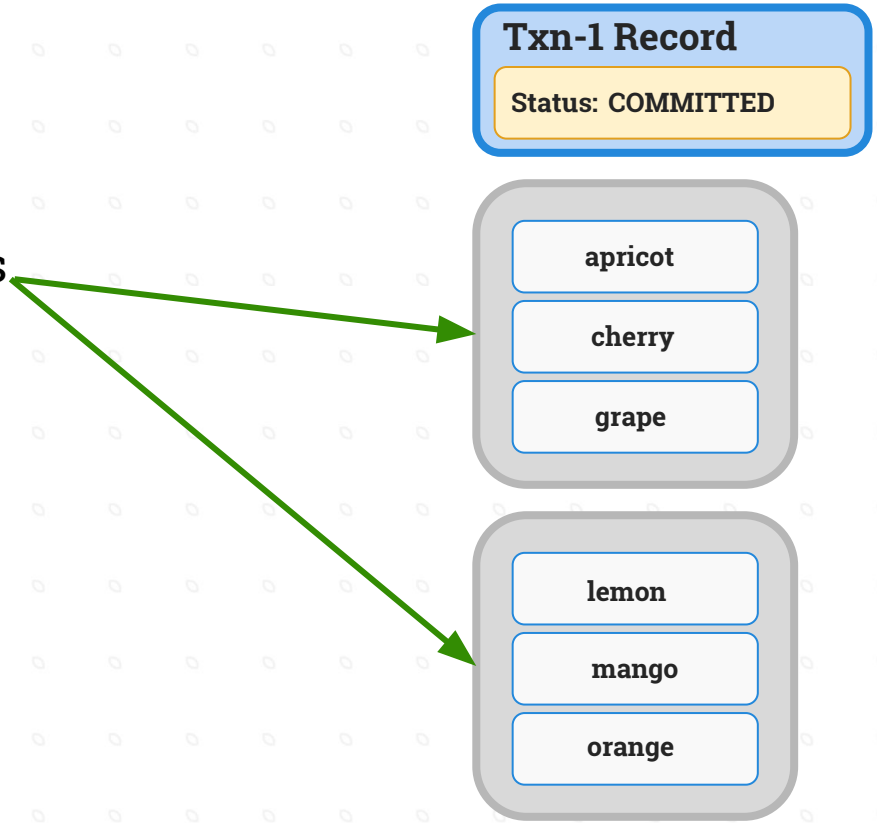
Distributed Transactions (CockroachDB)

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

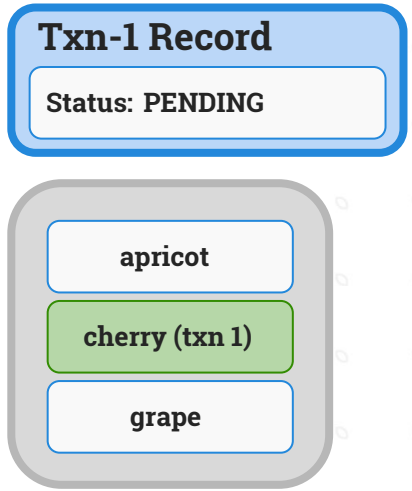


Distributed Transactions

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

Distributed Transactions (read conflict)

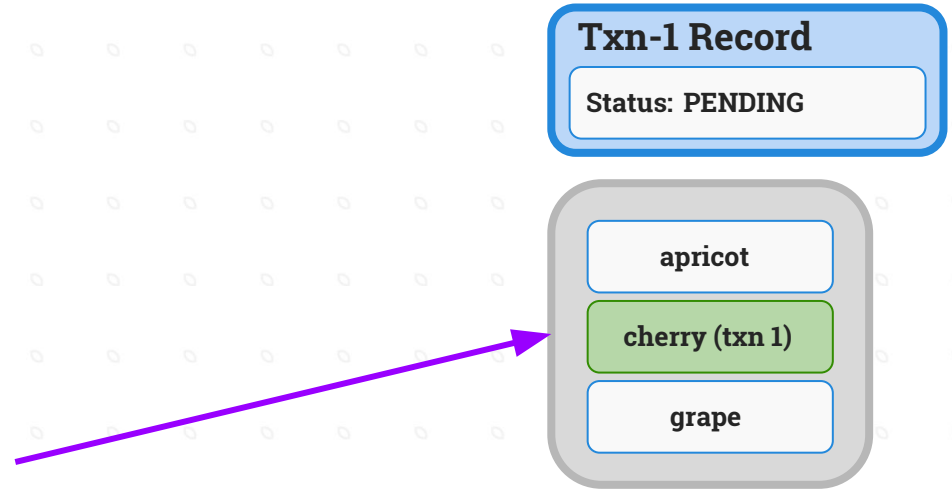
1. Begin Txn 1
2. Put “cherry”



Distributed Transactions (read conflict)

1. Begin Txn 1
2. Put “cherry”

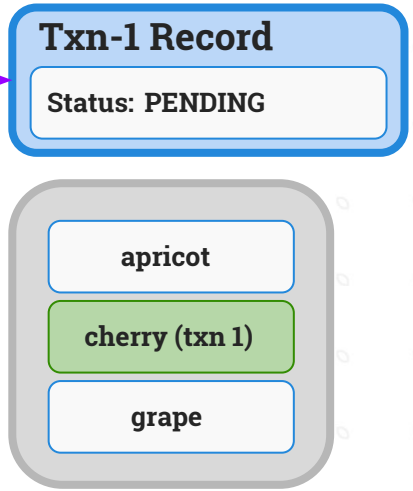
1. Begin Txn 2
2. Get “cherry”



Distributed Transactions (read conflict)

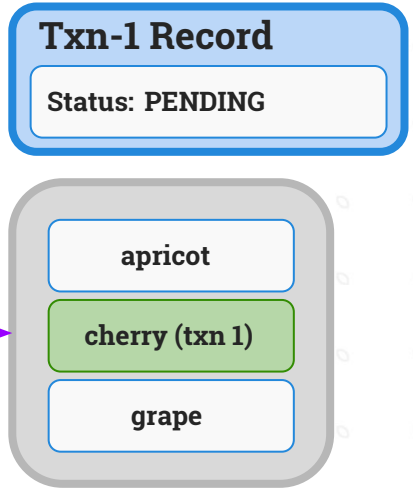
1. Begin Txn 1
2. Put “cherry”

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



Distributed Transactions (read conflict)

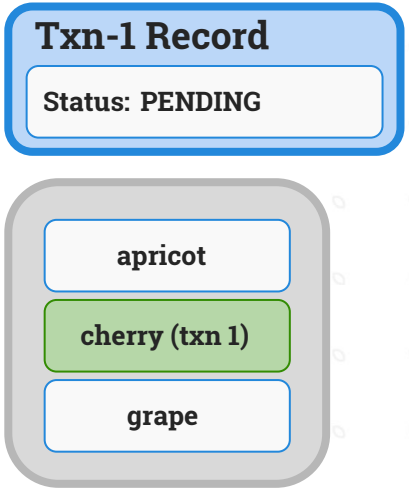
1. Begin Txn 1
2. Put “cherry”



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”

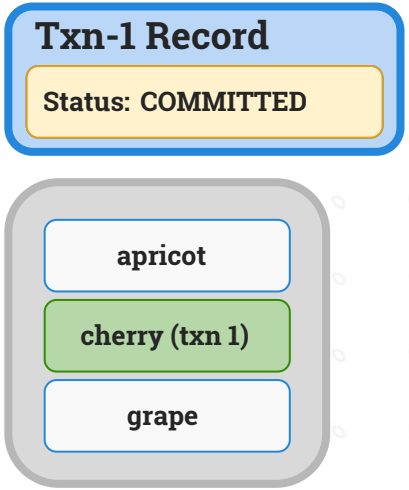


1. Begin Txn 2
2. Get “cherry”
   - Check txn 1 status
   - Ignore uncommitted value
3. Commit

Distributed Transactions (read conflict)

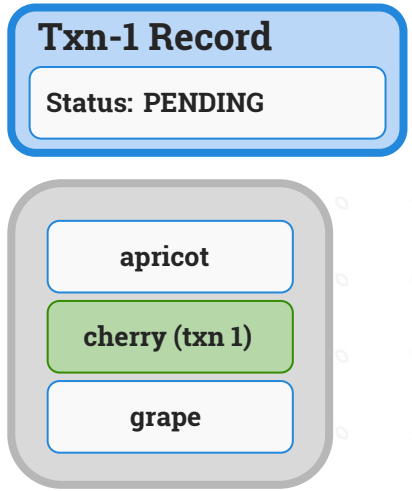
1. Begin Txn 1
2. Put “cherry”
3. Commit (potentially at later timestamp)

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



Distributed Transactions (write conflict)

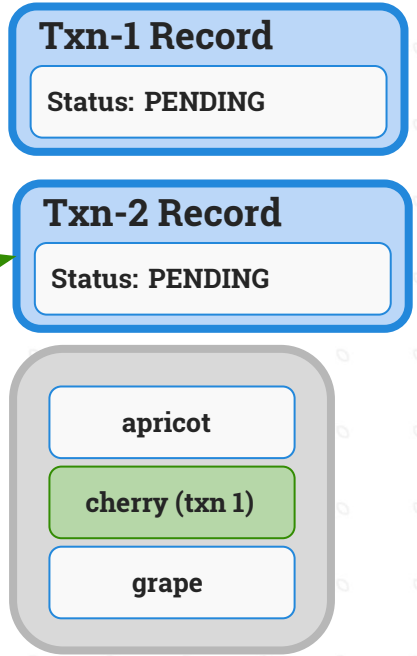
1. Begin Txn 1
2. Put “cherry”



Distributed Transactions (write conflict)

1. Begin Txn 1
2. Put “cherry”

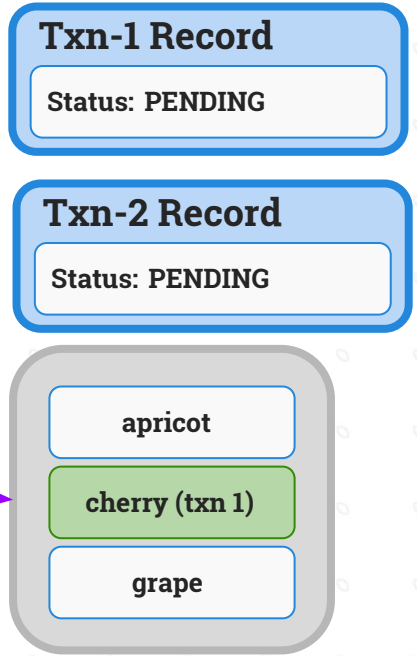
1. Begin Txn 2



Distributed Transactions (write conflict)

1. Begin Txn 1
2. Put “cherry”

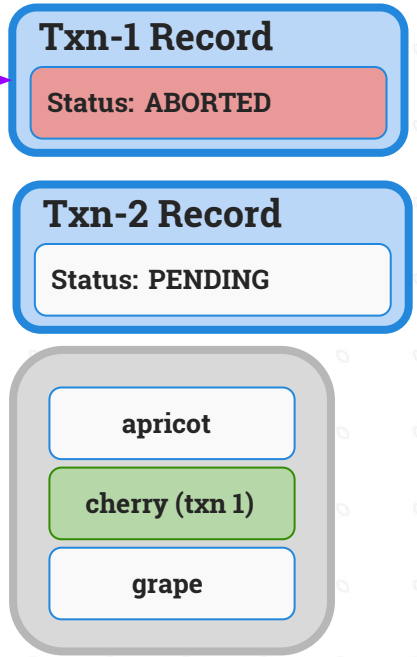
1. Begin Txn 2
2. Put “cherry”



Distributed Transactions (write conflict)

1. Begin Txn 1
2. Put “cherry”

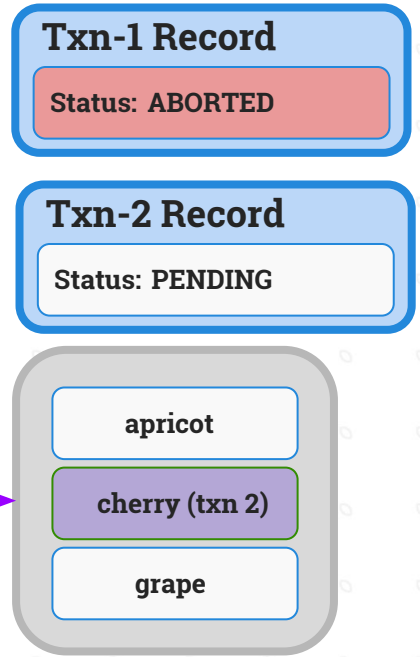
1. Begin Txn 2
2. Put “cherry”
   - Push Txn-1



Distributed Transactions (write conflict)

1. Begin Txn 1
2. Put “cherry”

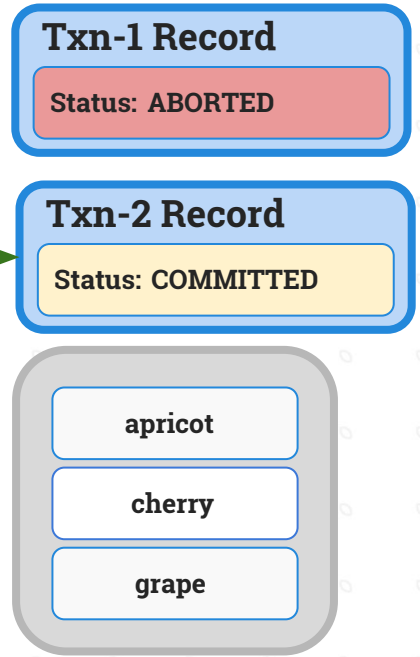
1. Begin Txn 2
2. Put “cherry”
   - Push Txn-1
   - Update intent



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: 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

Thank You

CockroachLabs.com

github.com/cockroachdb/cockroach

alex@cockroachlabs.com

github.com/a-robinson