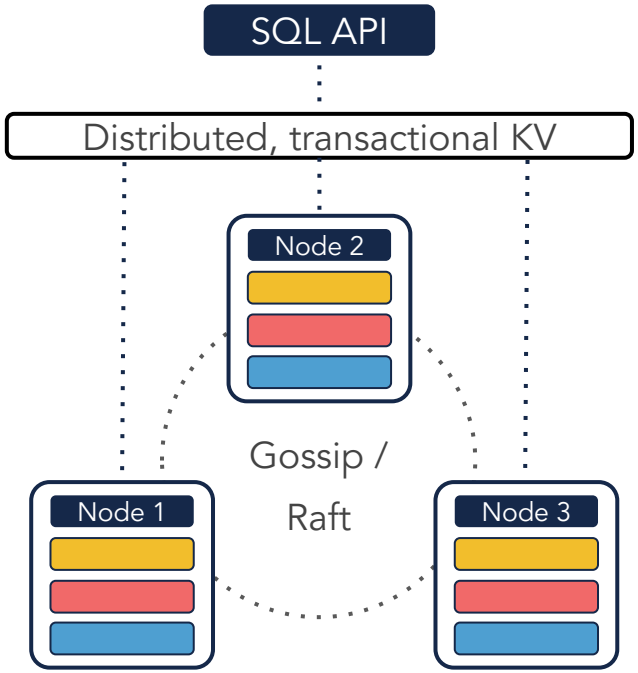
CockroachDB Architecture

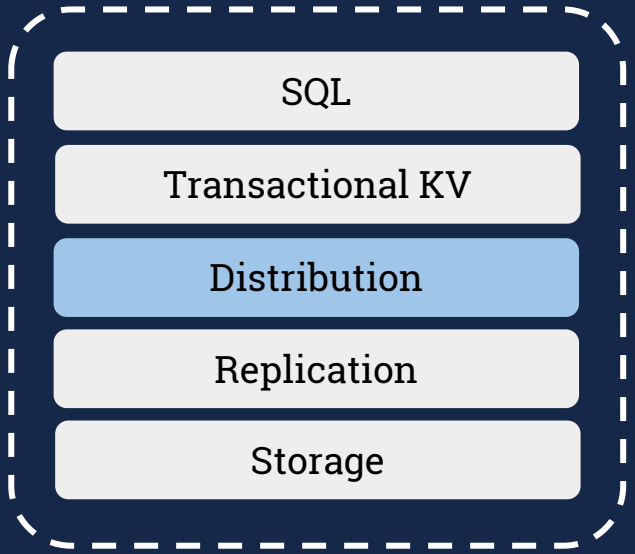
We built a database so you don't have to.

Architecture

- Topics
  - Data distribution
  - Consensus replication protocol (Raft)
  - Distributed transactions
  - Deployment, repair, rebalance



Data Distribution

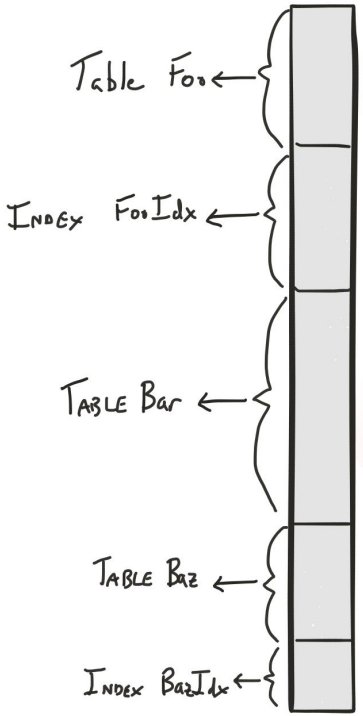


Data Distribution Mechanism

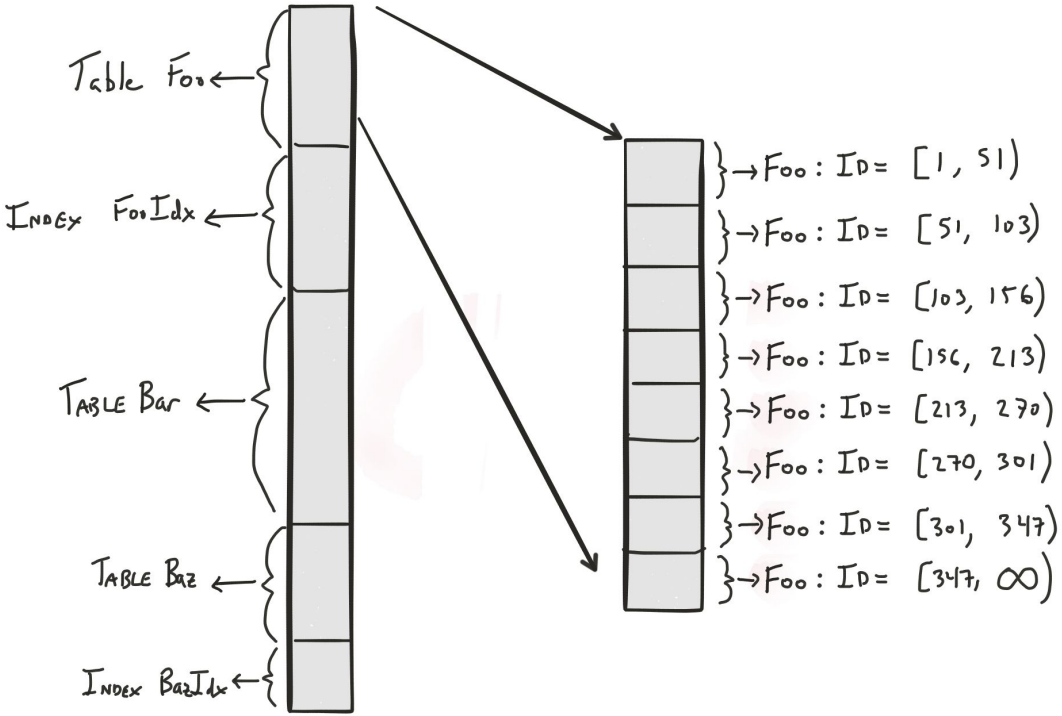
Order-preserving:

- Pro: efficient scans over total ordering of data
- Con: requires additional indexing

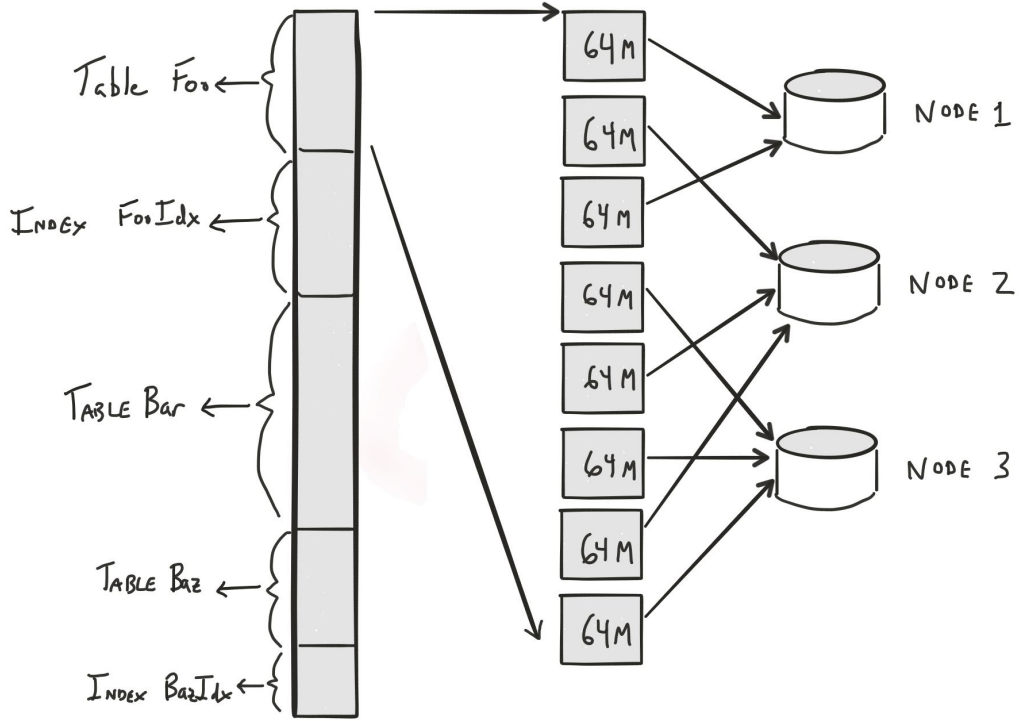
Keyspace as Monolithic, Sorted Map



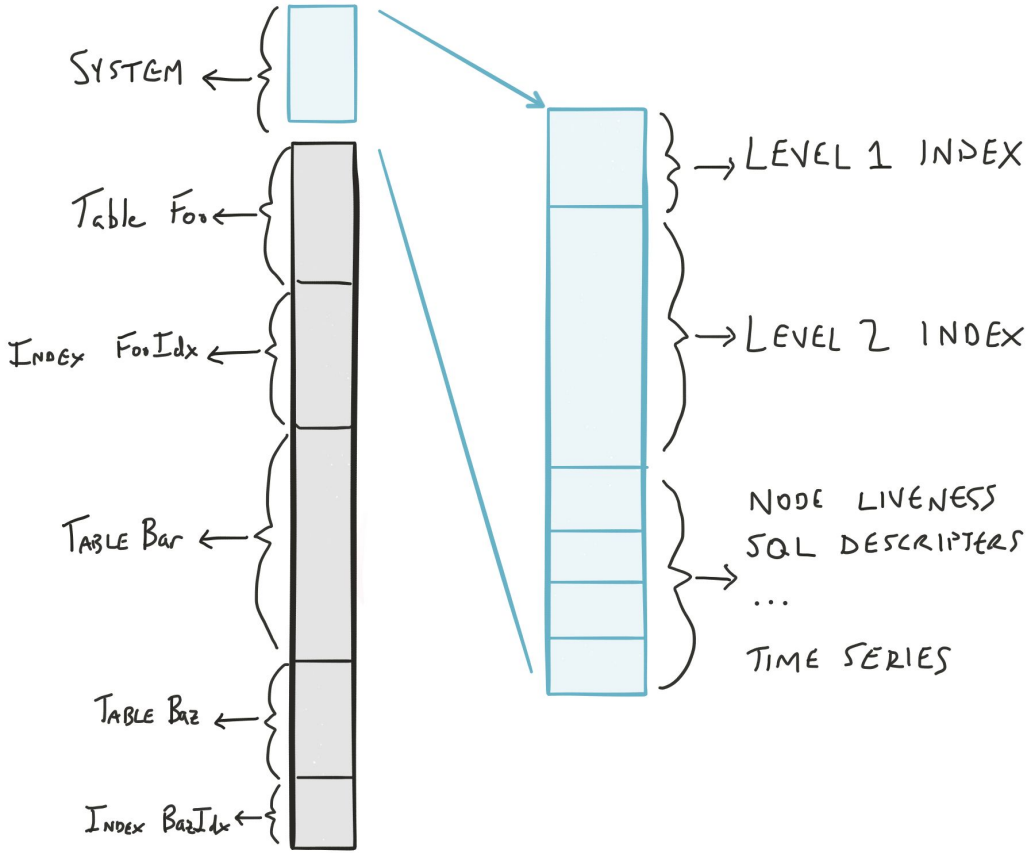
Ranges Form the Keyspace



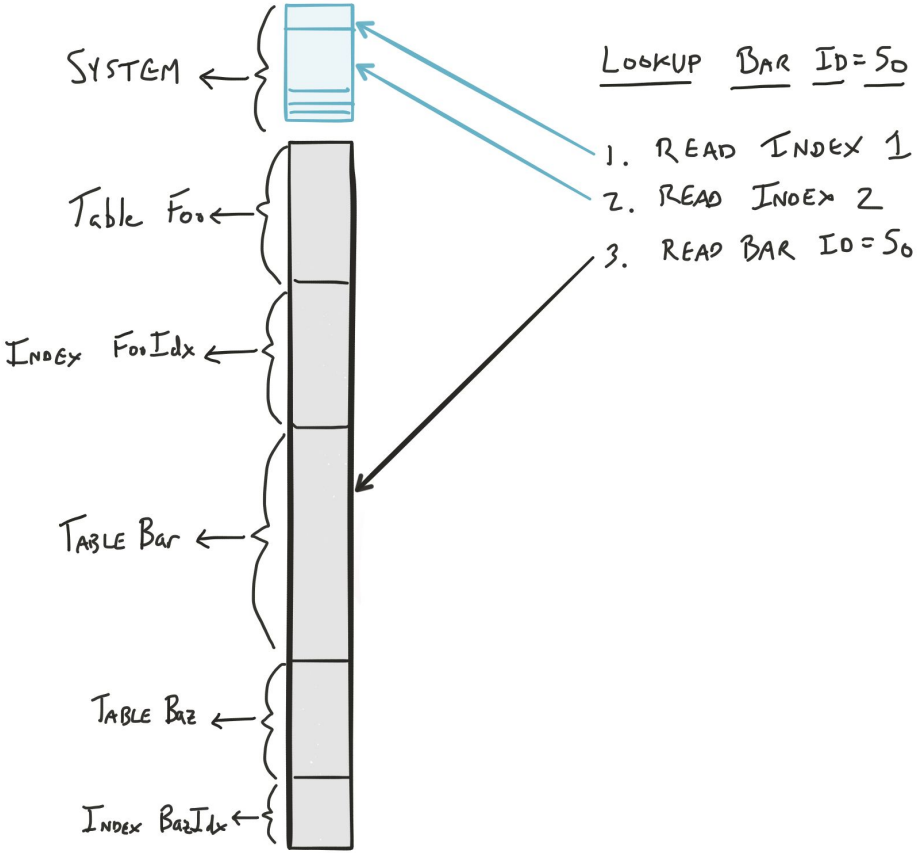
Ranges Map to Nodes



The System Keyspace



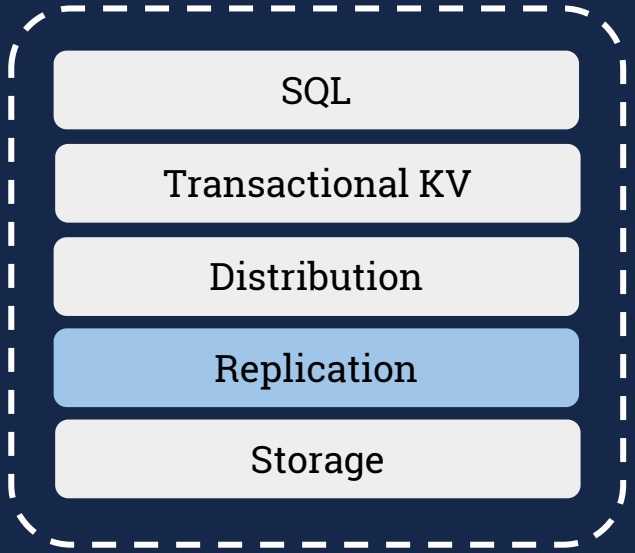
Data Lookup


SYSTEM
Table Foo
INDEX FooIdx
TABLE Bar
TABLE Baz
INDEX BazIdx
LOOKUP BAR ID=50
1. READ INDEX 1
2. READ INDEX 2
3. READ BAR ID=50

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



Consensus Replication

Replicate to N (where N >= 3) nodes

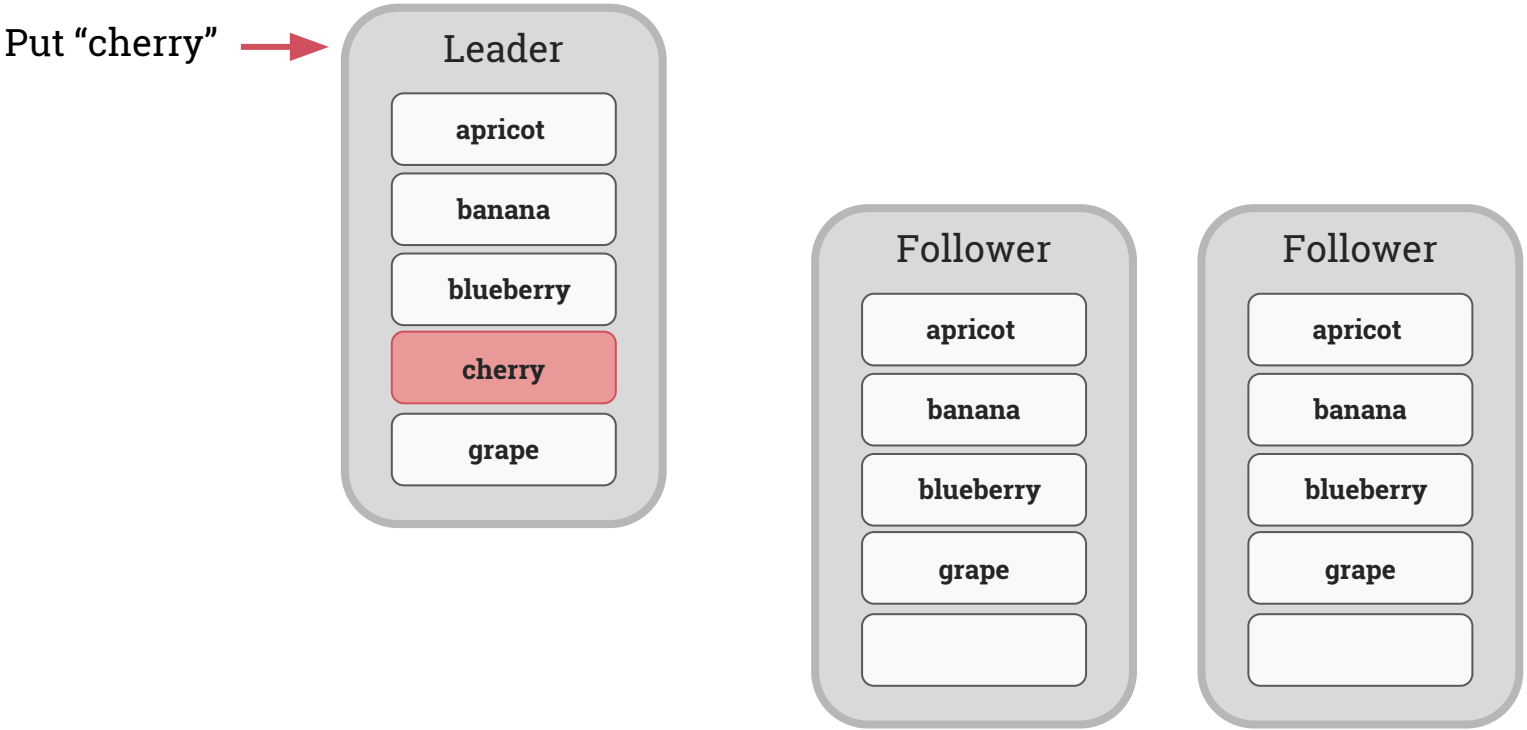
- Commit happens when a quorum have written the data

CockroachDB/Etcd/Spanner/Aurora/Zookeeper/...

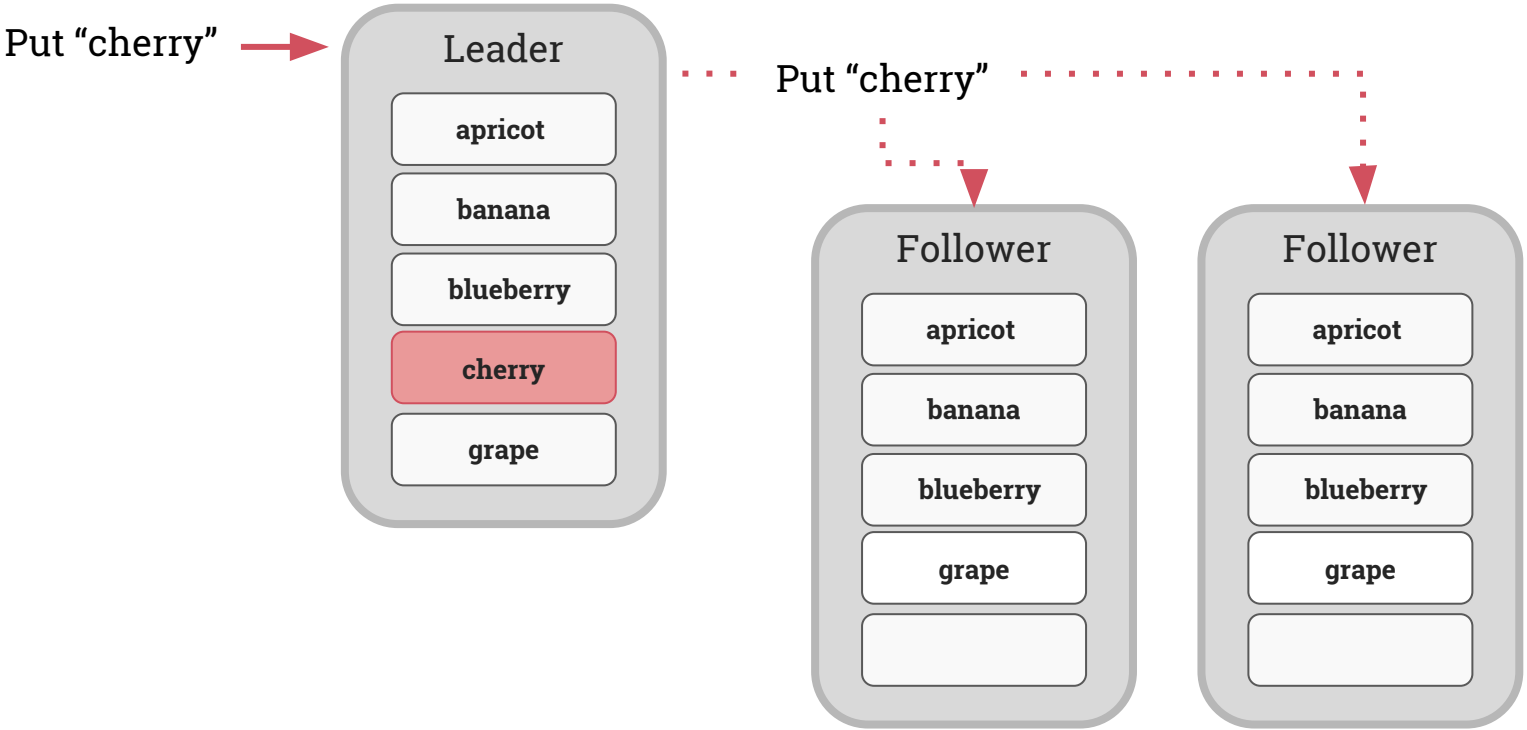
Consensus Replication

- **Raft** is our consensus protocol of choice.
- Run one consensus group per range of data
- Practical complications: membership, splits, etc.

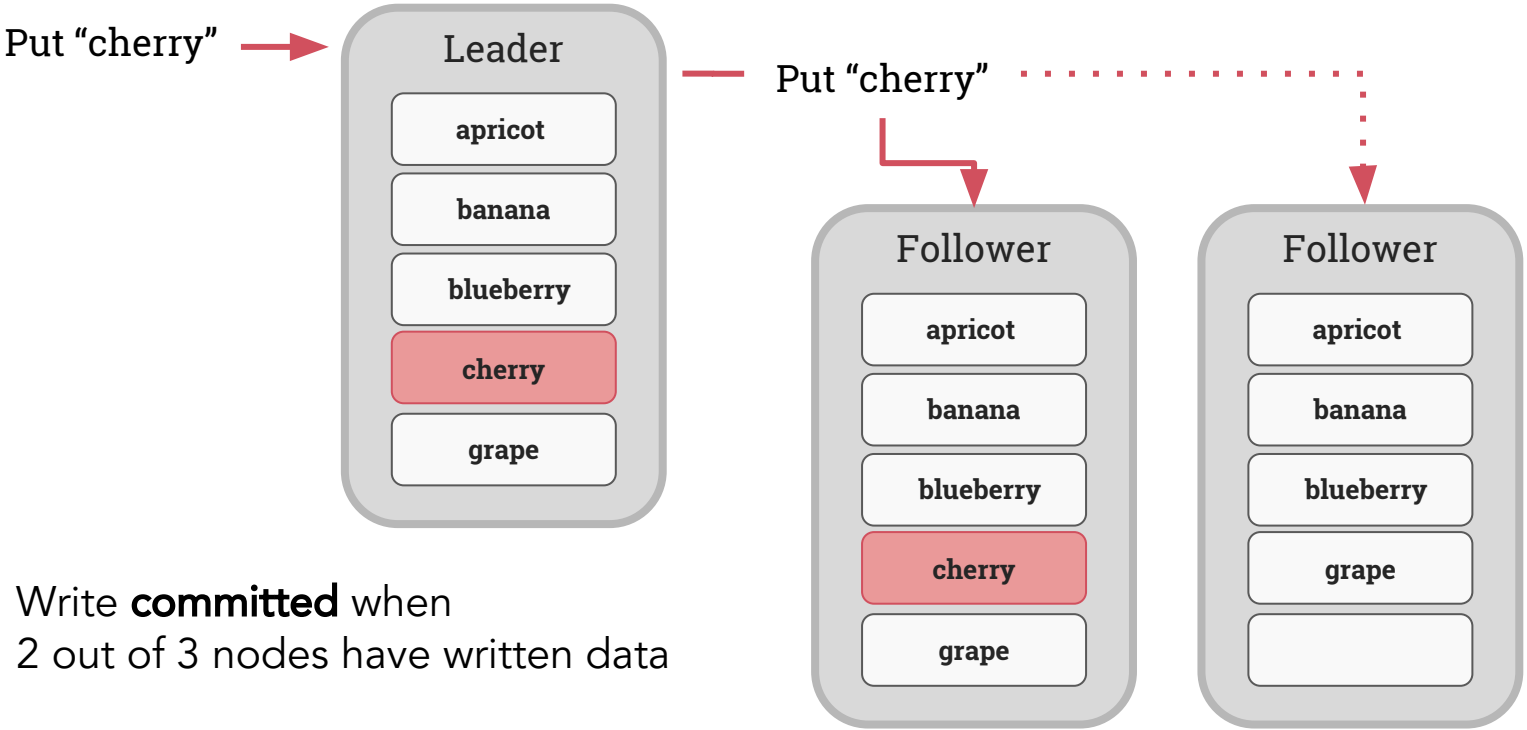
Consensus Replication



Consensus Replication

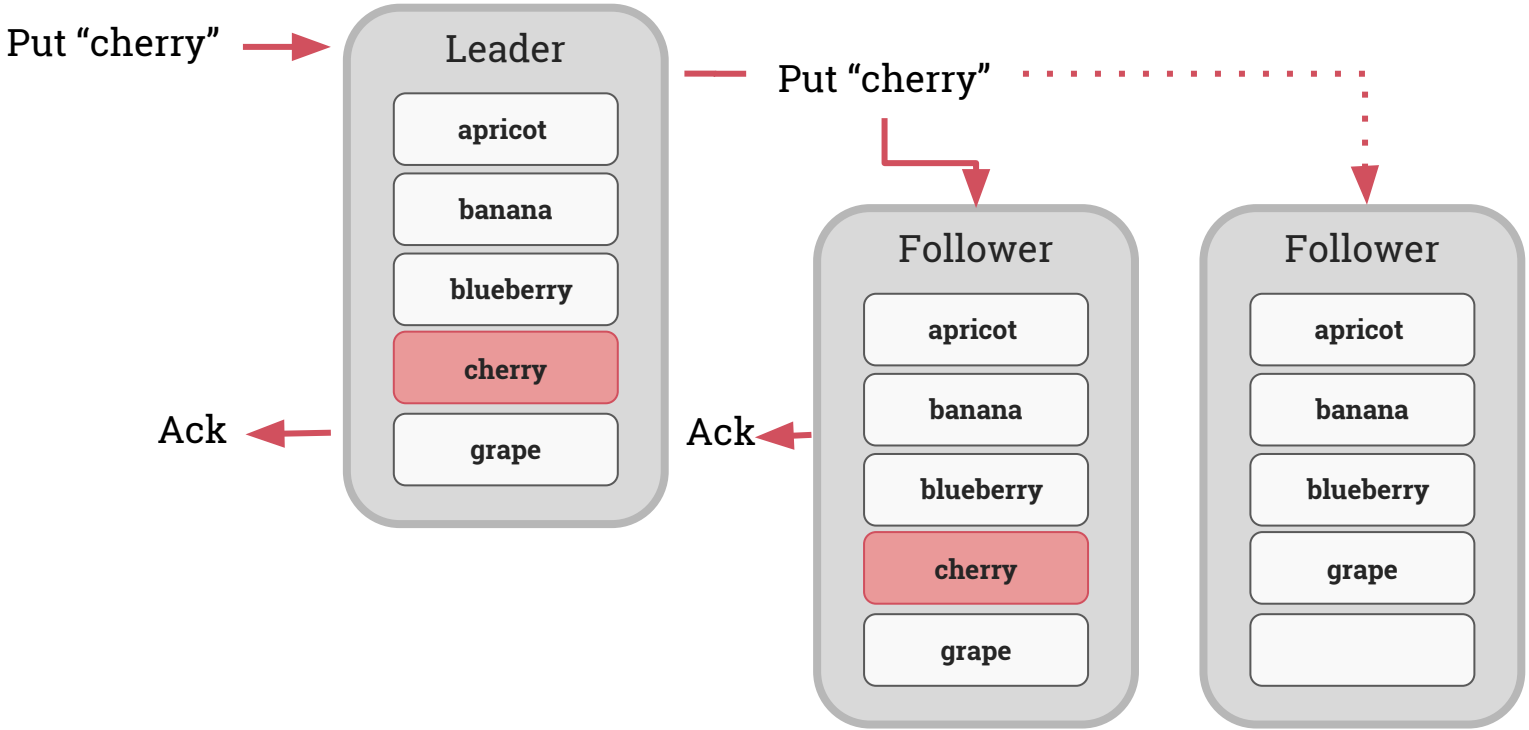


Consensus Replication

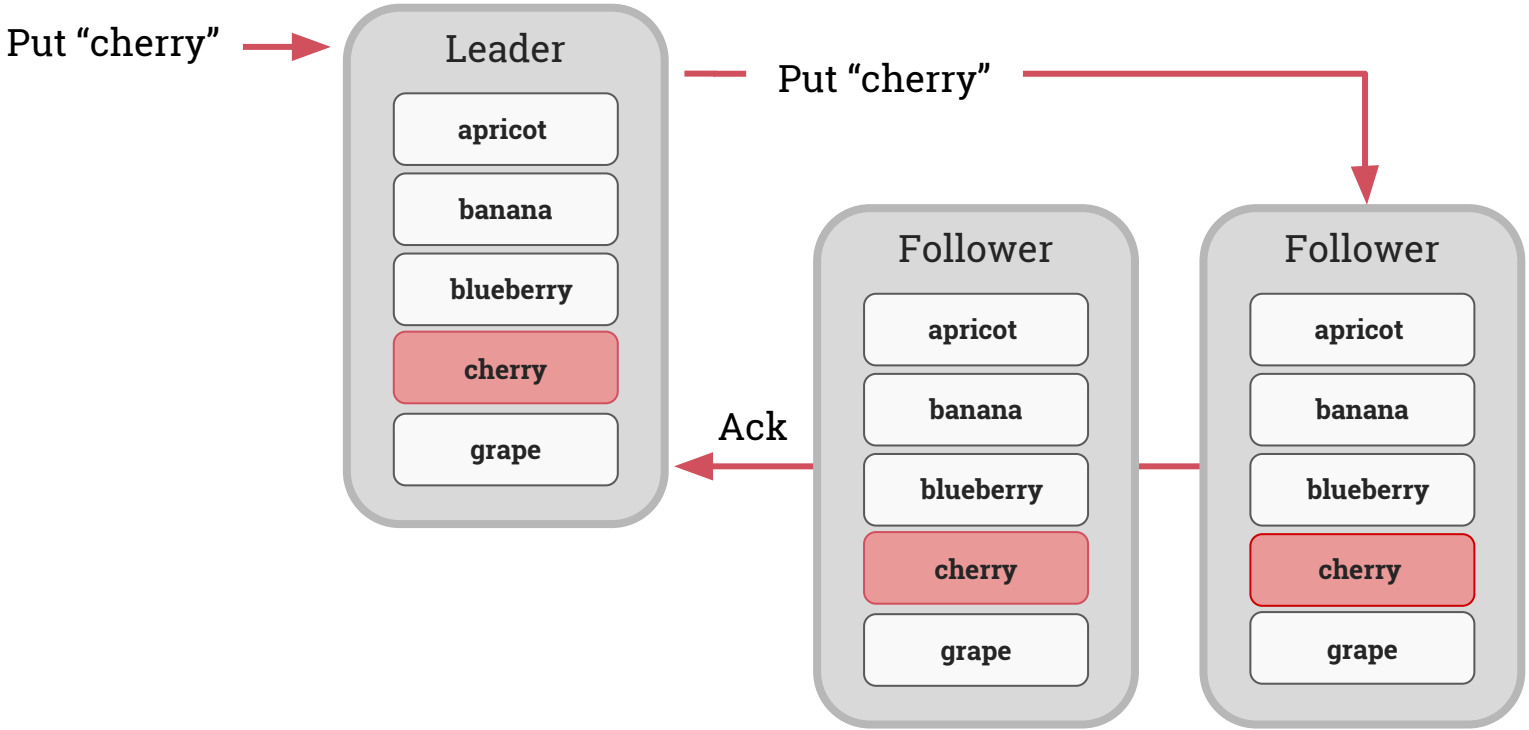


Write **committed** when
2 out of 3 nodes have written data

Consensus Replication



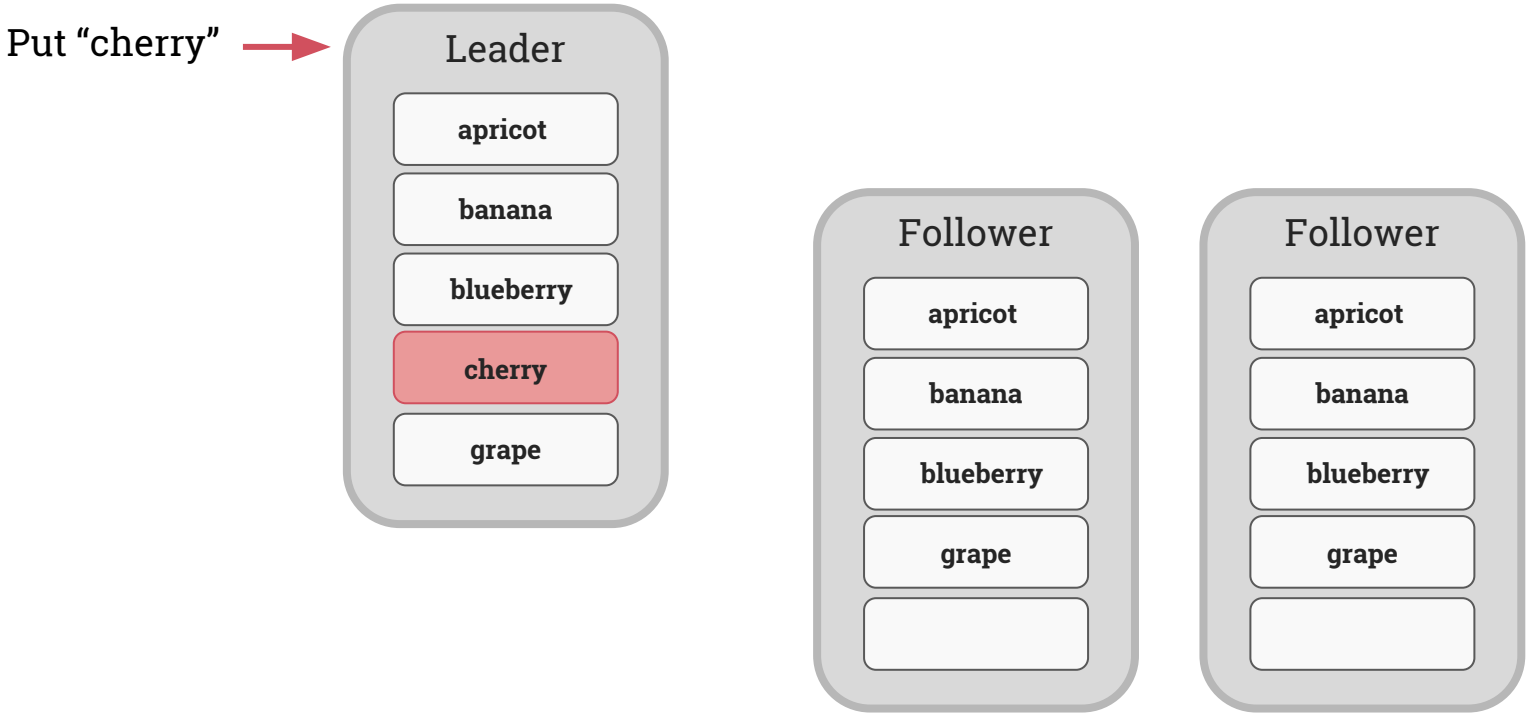
Consensus Replication



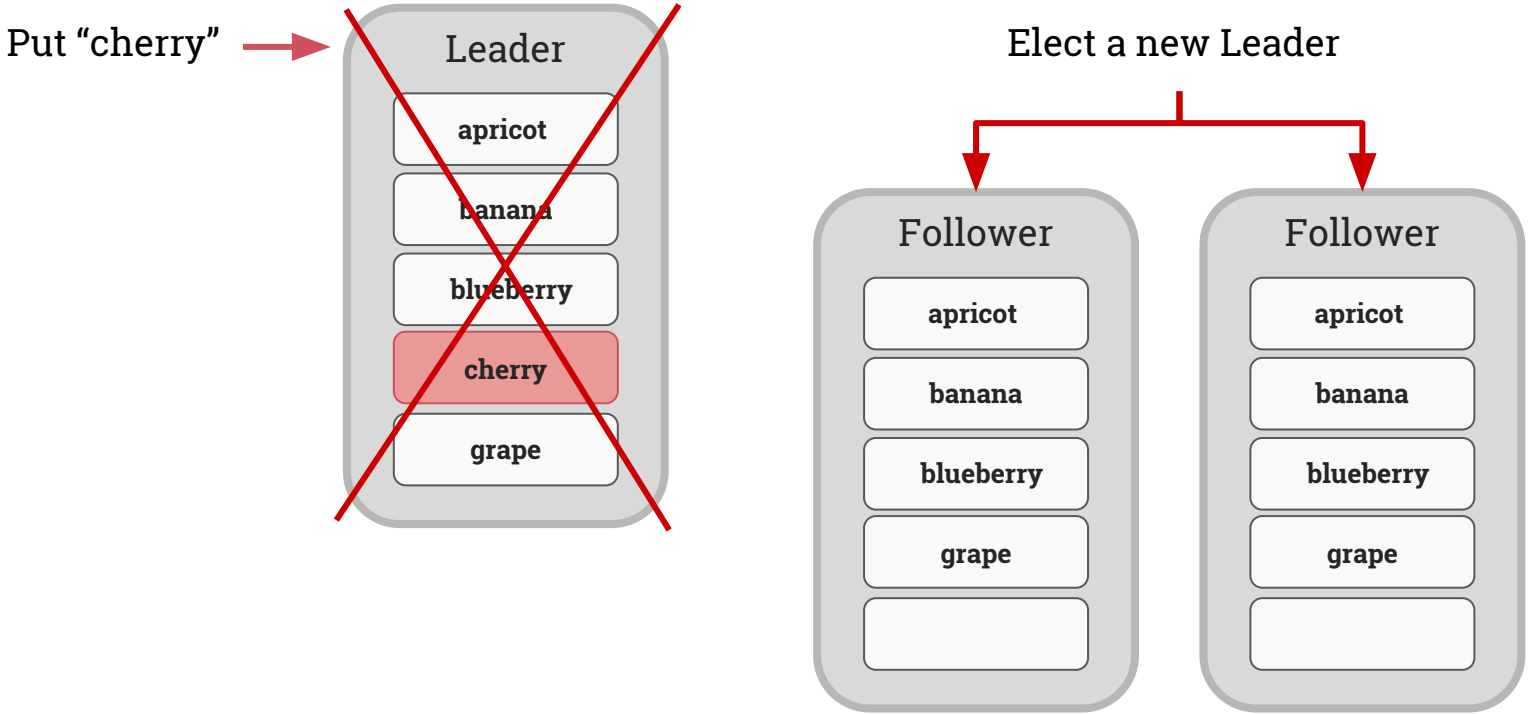
Consensus Replication

What happens during failover?

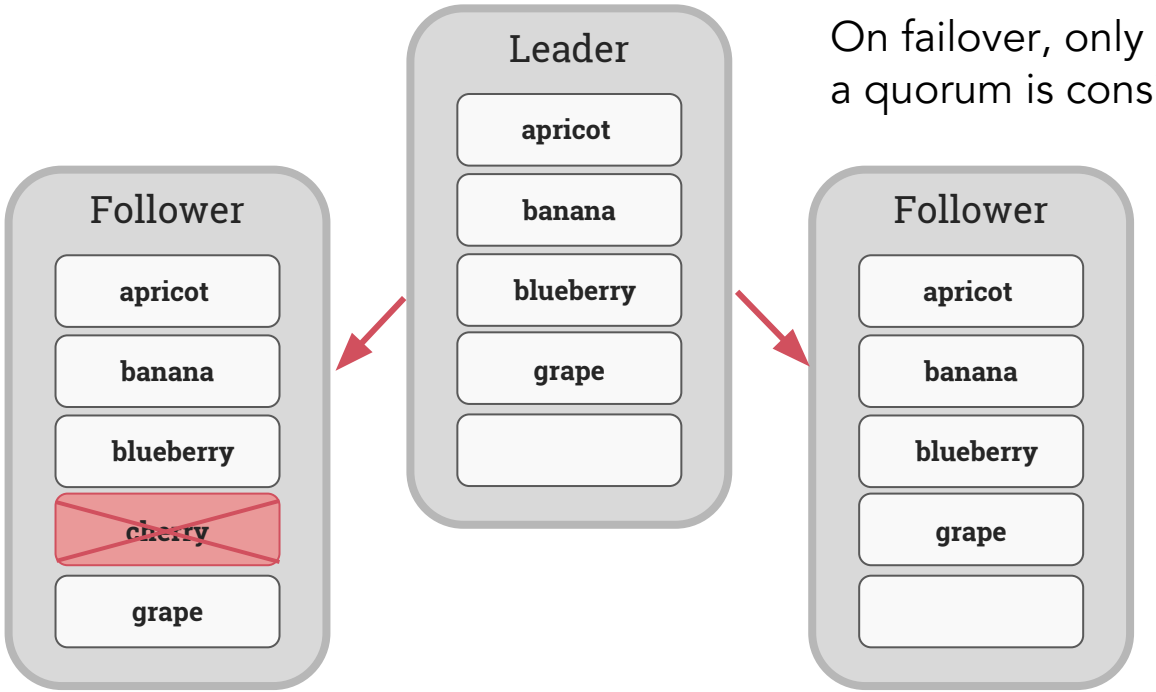
Consensus Replication: Failover



Consensus Replication: Failover

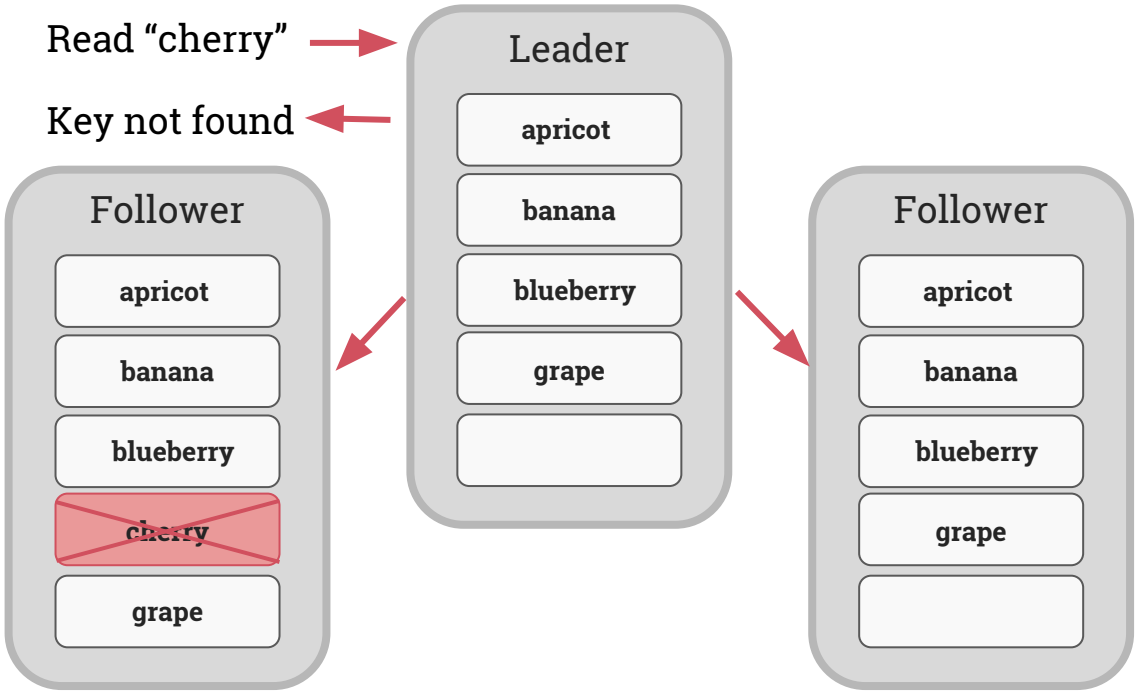


Consensus Replication: Failover



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

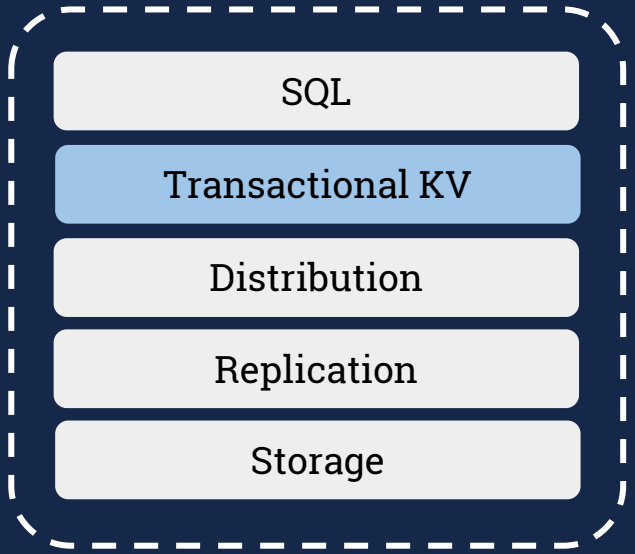
Consensus Replication: Failover



Consensus Replication

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

Distributed Transactions



Transactions

- Supports ACID semantics
  - “All-or-nothing”
  - Serializable [default] & Snapshot isolation levels

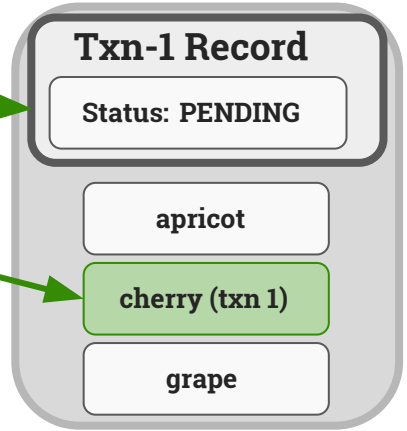
Distributed Transactions (CockroachDB)

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

- Transaction record
  - Keyed by a transaction UUID
  - Located on first written range (i.e. a Raft consensus group)
- Atomic commit or abort via Raft write to txn record
- One phase commit fast path

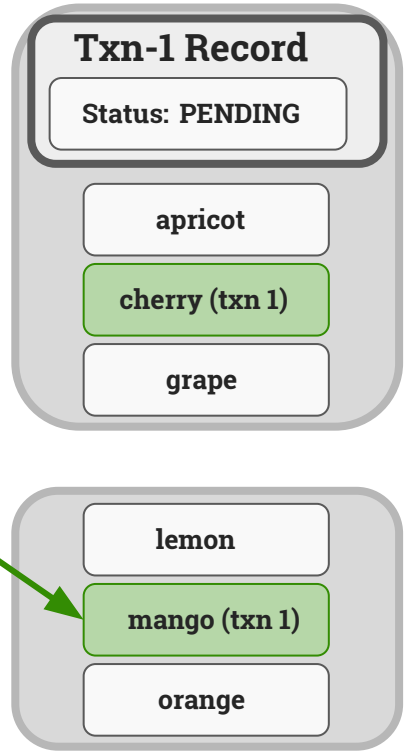
Distributed Transactions (CockroachDB)

1. Begin Txn 1
   Put “cherry”



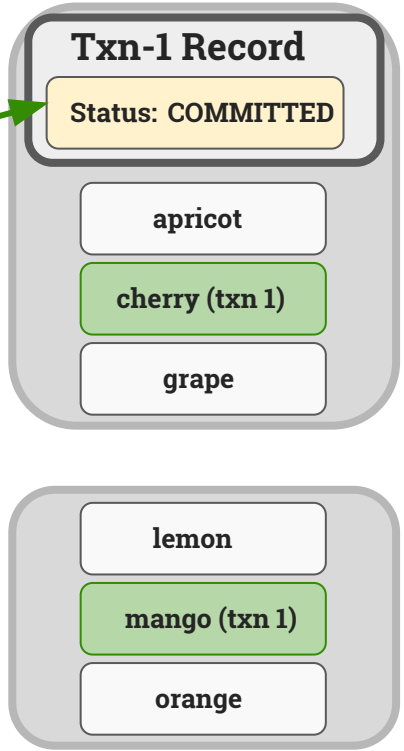
Distributed Transactions (CockroachDB)

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



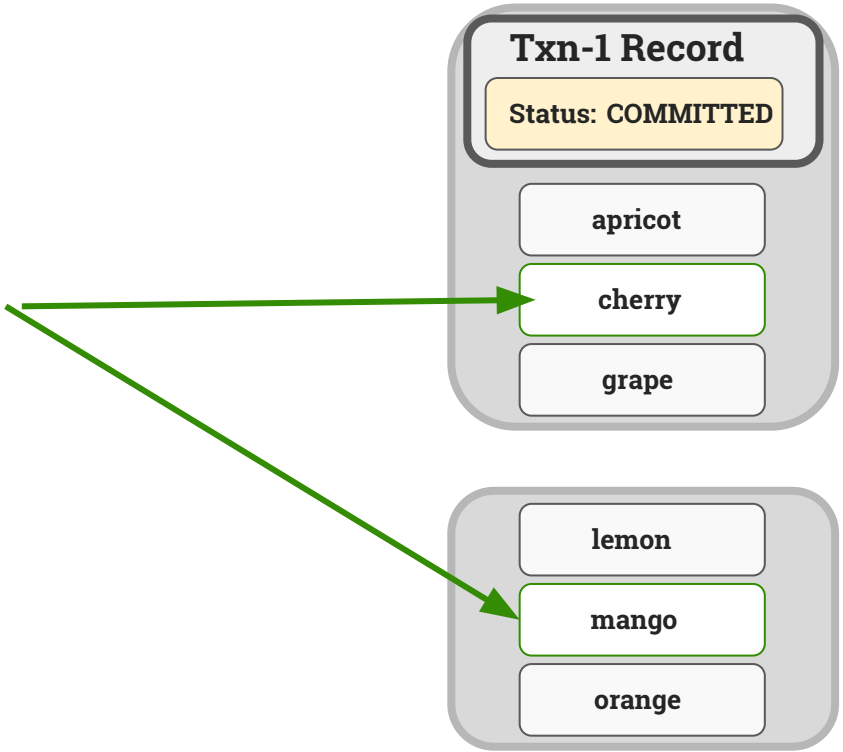
Distributed Transactions (CockroachDB)

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



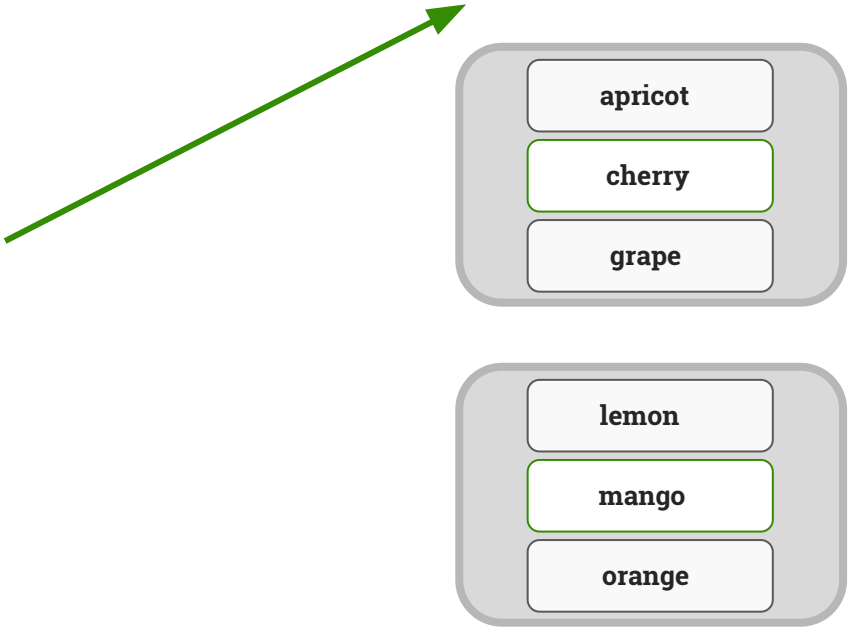
Distributed Transactions (CockroachDB)

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



Distributed Transactions (CockroachDB)

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

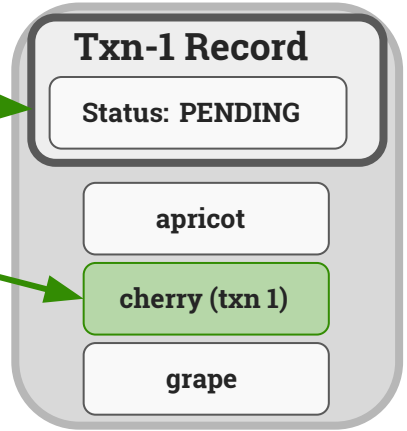


Distributed Transactions

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

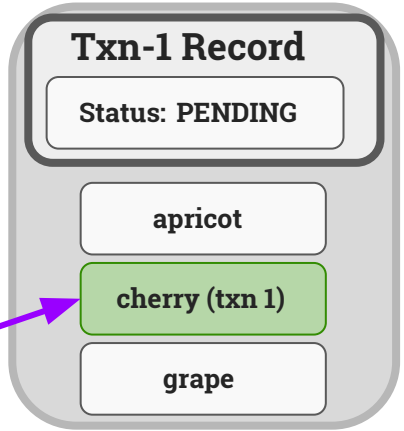
Distributed Transactions (read conflict)

1. Begin Txn 1
   Put “cherry”



Distributed Transactions (read conflict)

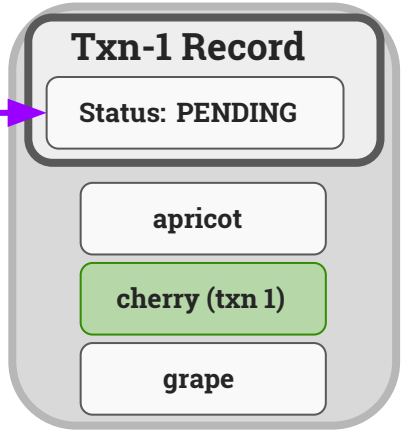
1. Begin Txn 1
   Put “cherry”



1. Begin Txn 2
2. Get “cherry”

Distributed Transactions (read conflict)

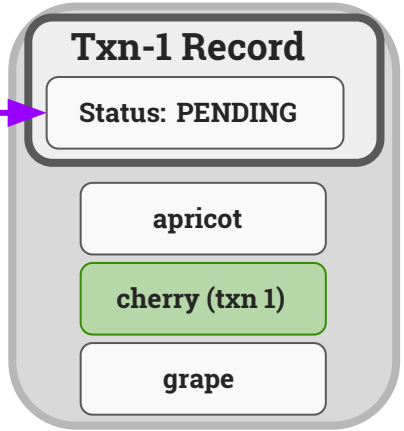
1. Begin Txn 1
   Put “cherry”



1. Begin Txn 2
2. Get “cherry”
   - Check Txn-1 record

Distributed Transactions (read conflict)

1. Begin Txn 1
   Put “cherry”



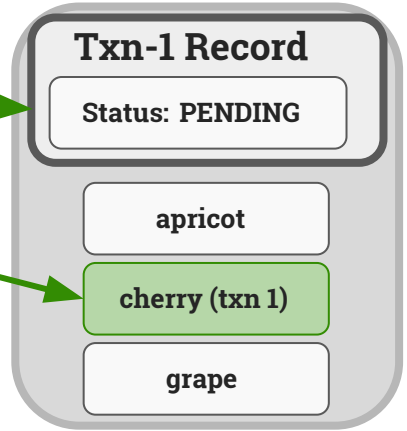
1. Begin Txn 2
2. Get “cherry”
   - Check Txn-1 record
   - Wait while PENDING

Distributed Transactions

- What about write / write conflicts?

Distributed Transactions (write conflict)

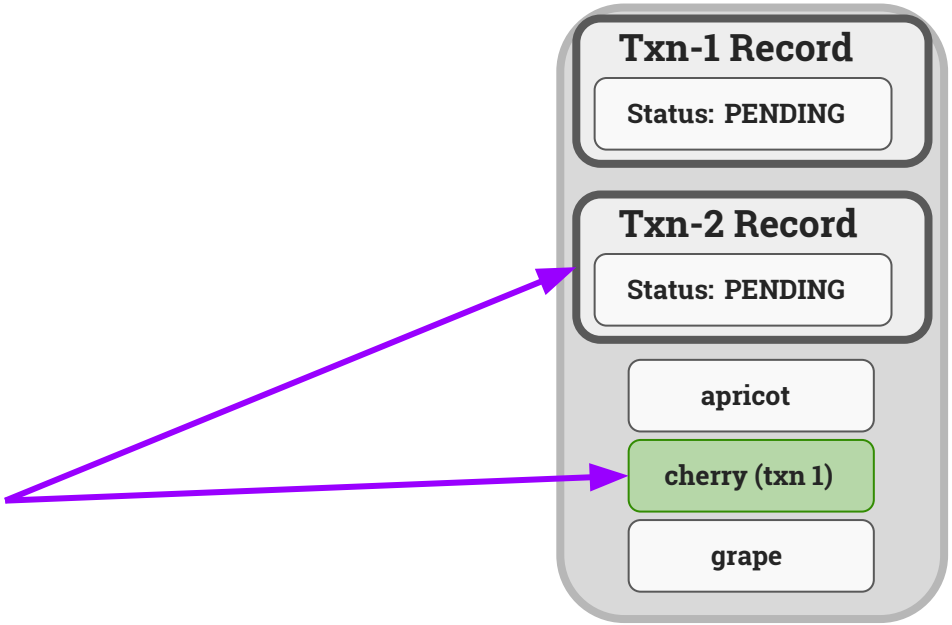
1. Begin Txn 1
   Put “cherry”



Distributed Transactions (write conflict)

1. Begin Txn 1
   Put “cherry”

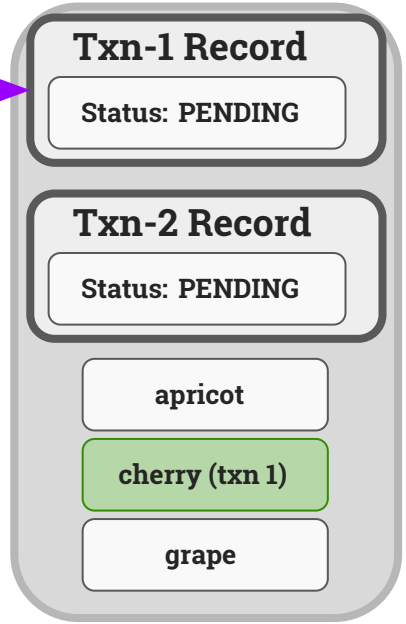
1. Begin Txn 2
2. Put “cherry”



Distributed Transactions (write conflict)

1. Begin Txn 1
   Put “cherry”

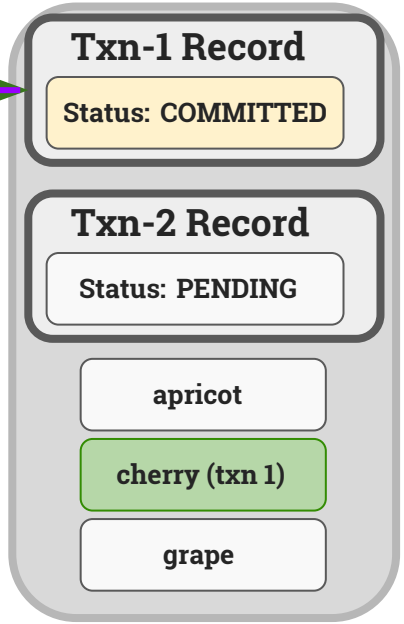
1. Begin Txn 2
2. Put “cherry”
   - Check Txn 1 record



Distributed Transactions (write conflict)

1. Begin Txn 1
   Put “cherry”
2. Commit Txn 1

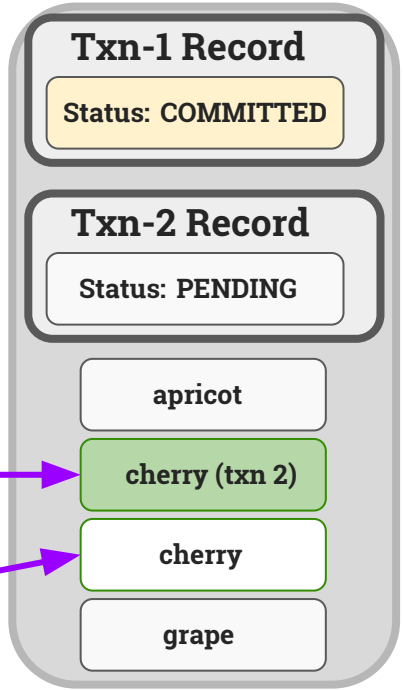
1. Begin Txn 2
2. Put “cherry”
   - Check Txn 1 record



Distributed Transactions (write conflict)

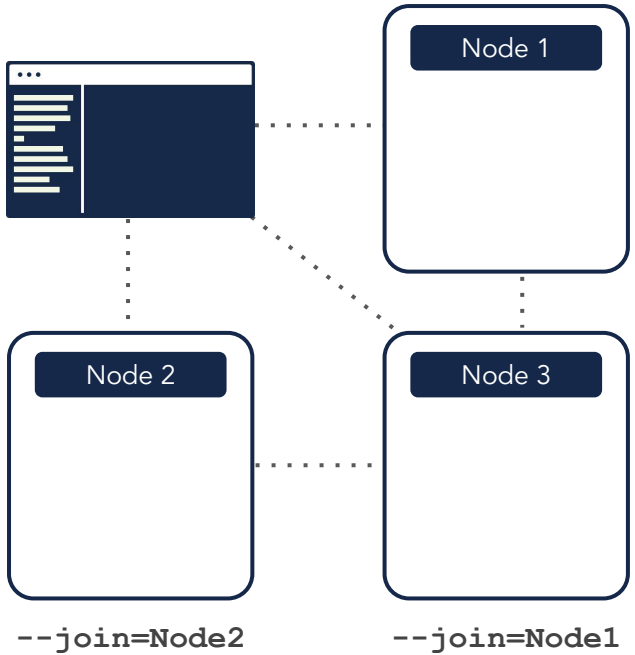
1. Begin Txn 1
   Put “cherry”
2. Commit Txn 1

1. Begin Txn 2
2. Put “cherry”
   - Check Txn 1 record
   - Resolve cherry (txn 1)
   - Put “cherry”



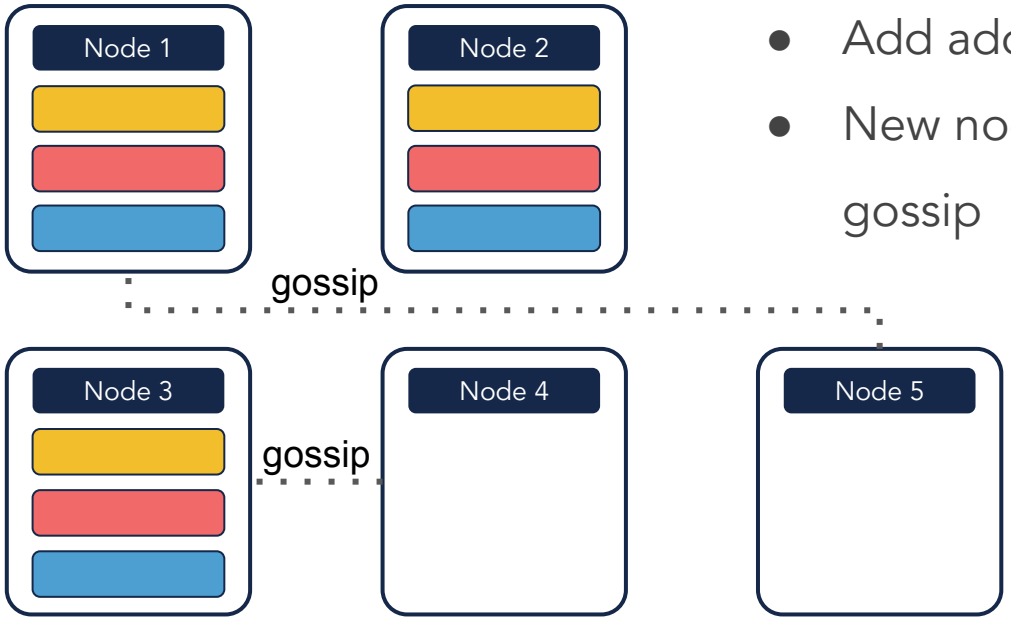
Deployment

Simple Deployment with Symmetric Nodes



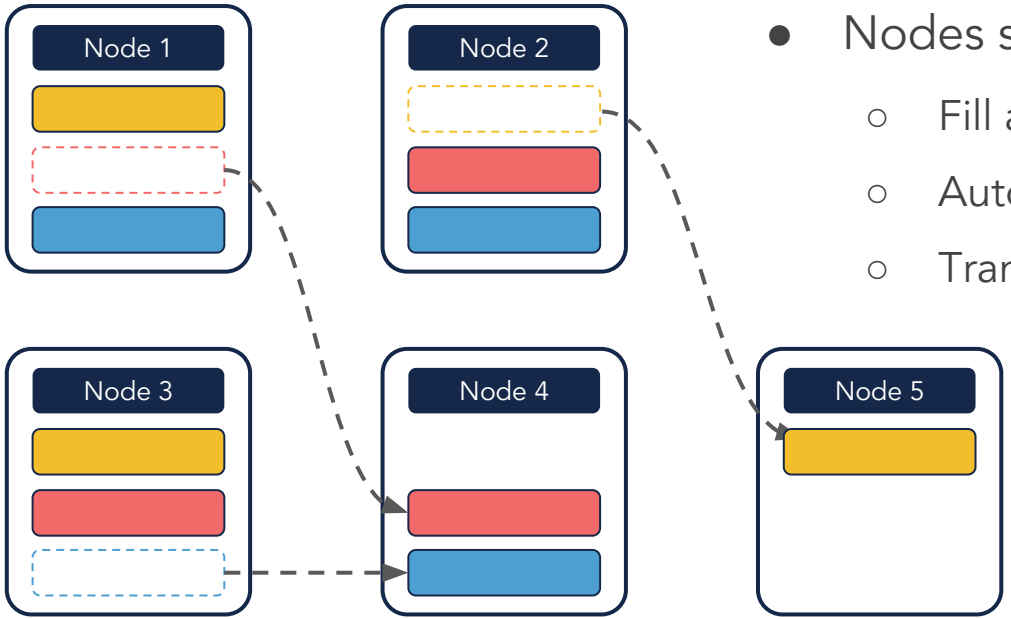
- Download single binary
- Install with a single command
- Every node is a client gateway
- Applications see one logical DB

Scale Out



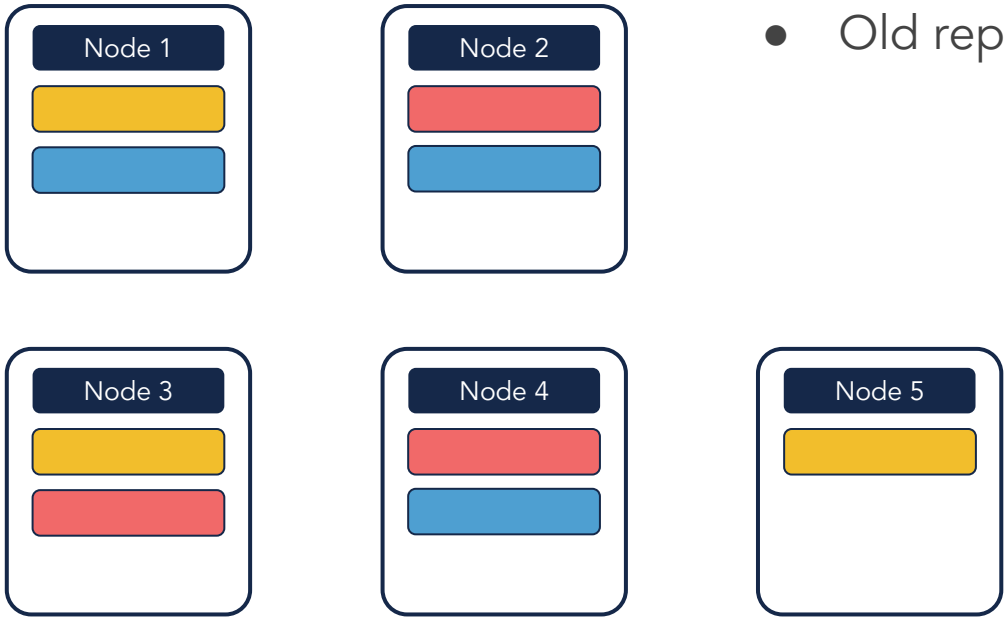
- Add additional nodes to scale
- New nodes connect / communicate via gossip

Scale Out



- Nodes self-organize via rebalancing
  - Fill available space from new nodes
  - Automated failover and recovery
  - Transparent to applications

Scale Out



- Old replicas are garbage collected