The Hows and Whys of a Distributed SQL Database

Alex Robinson / Member of the Technical Staff
alex@cockroachlabs.com / @alexwritescode

Agenda

- Why a distributed SQL database?
- How does a distributed SQL database work?
  - Data distribution
  - Data replication
  - Transactions

Why a distributed SQL database?

A brief history of databases

1960s: The first databases

- Records stored with pointers to each other in a hierarchy or network
- Queries had to process one tuple at a time, traversing the structure
- No independence between logical and physical schemas
  - To optimize for different queries, you had to dump and reload all your data
- Required a lot of programmer effort to use

1970s: SQL / RDBMS

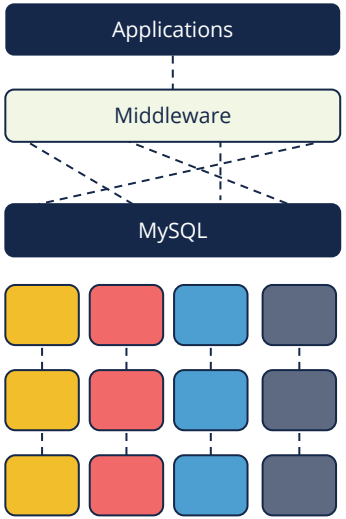
- Relational model was designed in contrast to prior hierarchical DBs
  - Expose simple data structures and high-level language
  - Queries are independent of the underlying physical storage
- Very developer friendly
- Designed to run on a single machine

1980s-1990s: Continued growth of SQL / RDBMS

- SQL databases matured, took over the industry
- Object-oriented databases tried, but didn't gain much traction
  - Didn't support complex queries well
  - No standard API across systems

Early 2000s: Custom Sharding

- Rapidly growing scale necessitated new solutions
- Companies added custom middleware to connect single-node DBs
- No cross-shard transactions, operational nightmares



2004+: NoSQL

- Further growth and operational pain necessitated new systems
- Big change in priorities - scale and availability above all else
- Sacrificed a lot to get there
  - No relational model (custom APIs instead of SQL)
  - No (or very limited) transactions and indexes
  - Manual joins

2010s: “NewSQL” / Distributed SQL

Why another type of database?

Database Limitations

Existing database solutions place an undue burden on application developers and operators:

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

Strong Consistency and 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"

-Google's Spanner paper

2010s: “NewSQL” / Distributed SQL

- Distributed databases that provide full SQL semantics
- Attempt to combine the best of both worlds
  - Fully-featured SQL
  - Horizontally scalable and highly available
- Entirely new systems, tough to tack onto existing databases

2010s: “NewSQL” / Distributed SQL

- Burst onto the scene with Google’s Spanner system (published in 2012)
- CockroachDB started open source development in 2014

So we know why, but how?

And what's different from SQL and NoSQL databases?

What we're going to cover

1. How data is distributed across machines
2. How remote copies of data are replicated and kept in sync
3. How transactions work

Plus how all of this differs from conventional SQL and NoSQL systems

Disclaimer

Data Distribution

Data Distribution in SQL

- Option 1: All data on one server
  - With optional secondary replicas/backups that also store all the data
- Option 2: Manually shard data across separate database instances
  - All data for each shard is still just on one server (plus any secondary replicas)
  - All data distribution choices are being made by the human DB admin

Data Distribution in NoSQL / NewSQL

- Core assumption: entire dataset won't fit on one machine
- Given that, there are two key questions to answer:
  - How do you divide it up?
  - How do you locate any particular piece of data?
- Two primary approaches: Hashing or Order-Preserving

Option one: Hashing

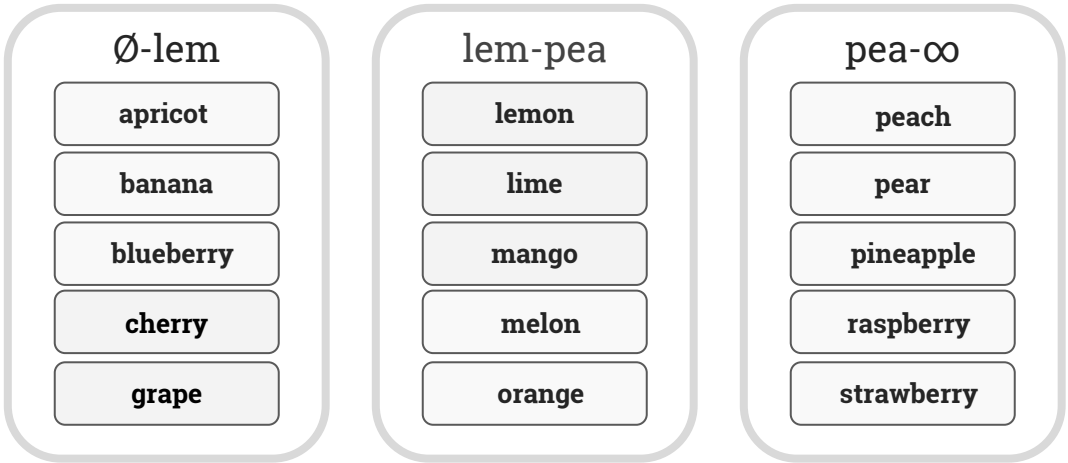
- Pick one or more hash functions, divide up data by hashing each key
- Deterministically map hashed keys to servers
- Pros: easy to locate data by key
- Con: awkward, inefficient range scans

Option two: Order-Preserving

- Divide sorted key space up into ranges of approximately equal size
- Distribute the resulting key ranges across the servers
- Pros: efficient scans, easy splitting
- Con: requires additional indexing

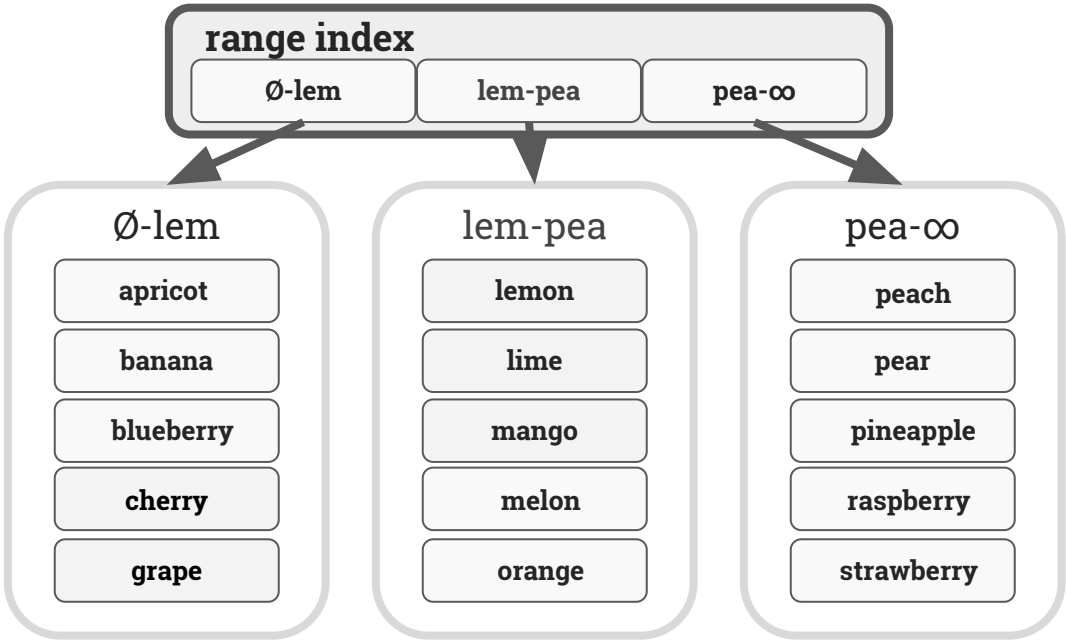
Option two: Order-Preserving

Each shard (“range”) contains a contiguous segment of the key space



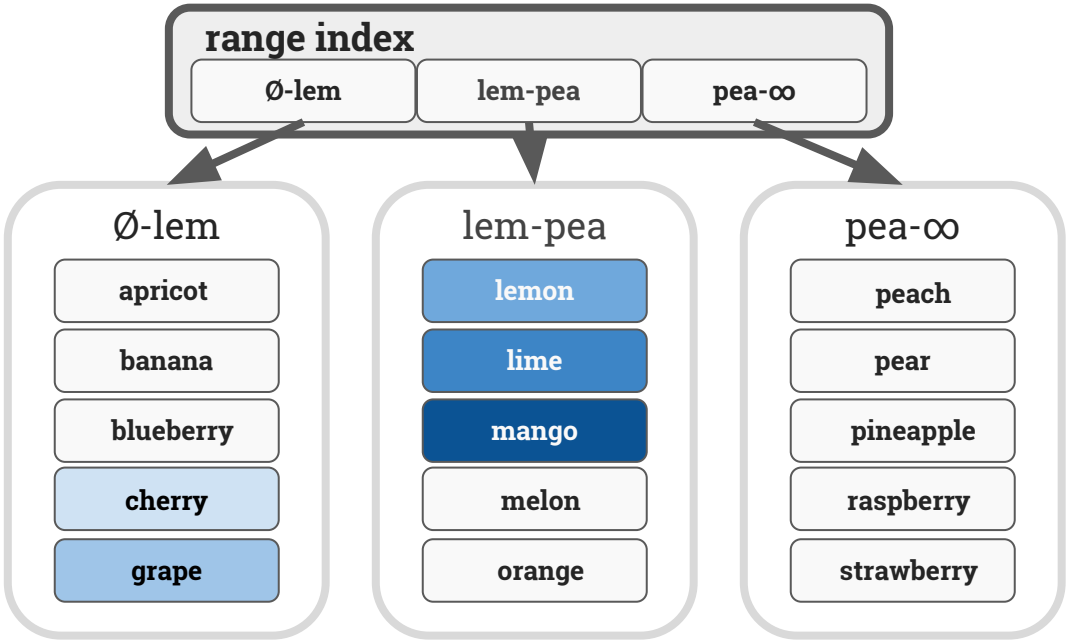
Option two: Order-Preserving

We need an indexing structure to locate a range

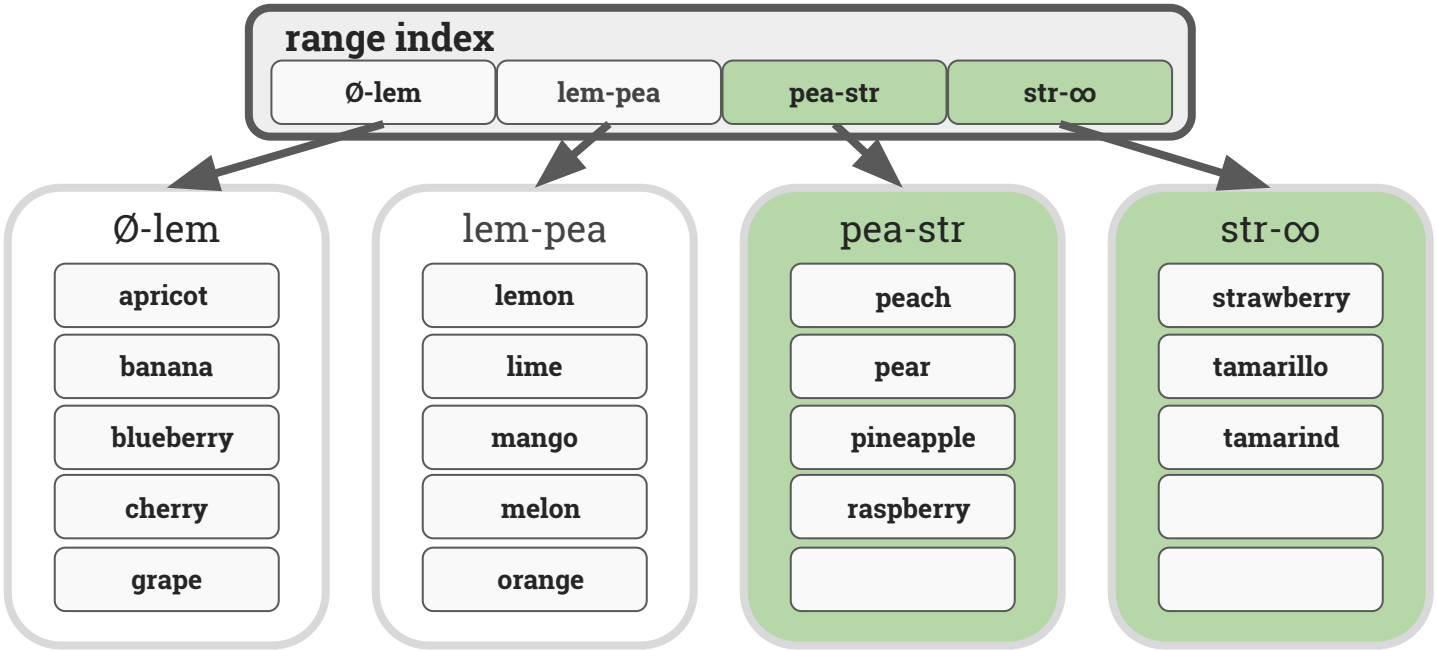


Option two: Order-Preserving

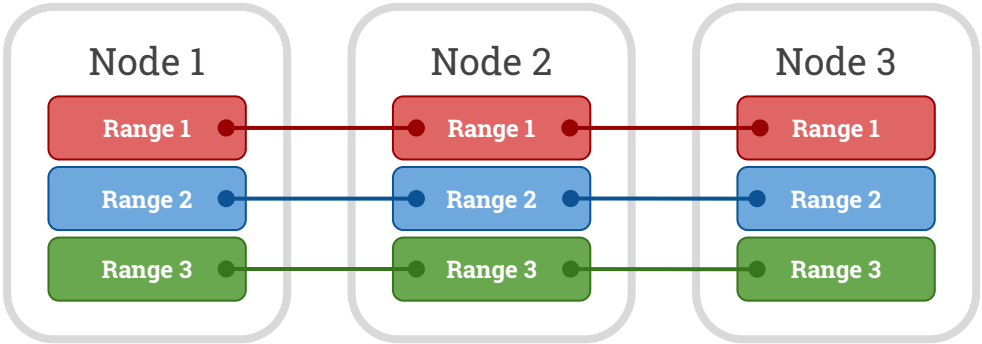
Scans (fruits >= “cherry” AND <= “mango”) are efficient



Option two: Order-Preserving

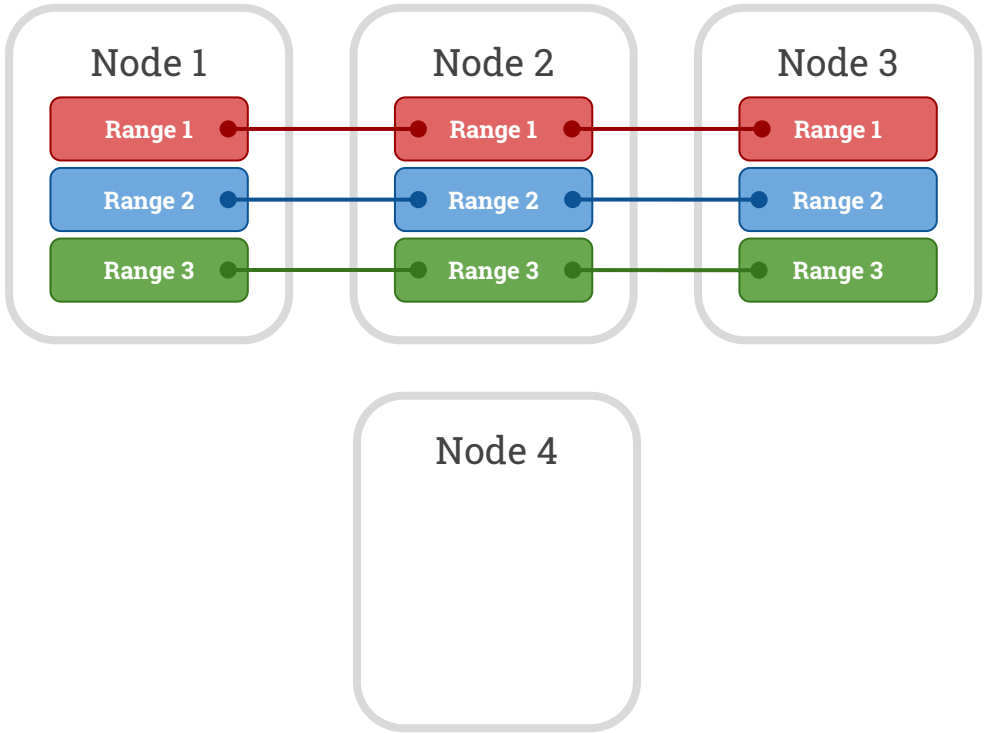


Data Distribution: Placement



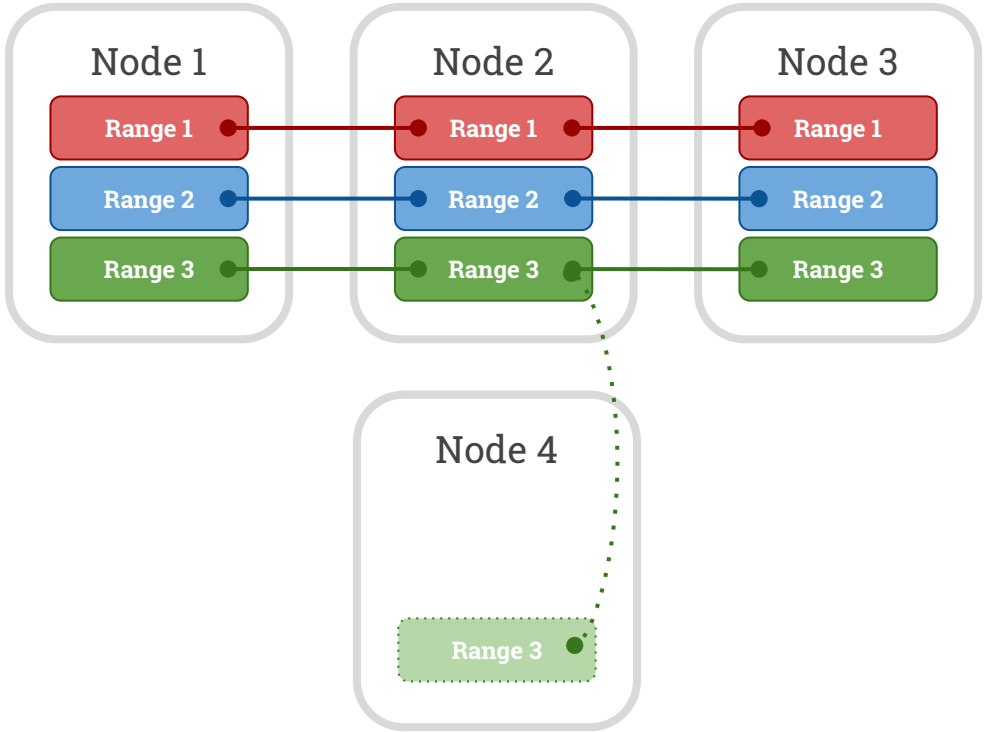
Each range is replicated to three or more nodes

Data Distribution: Rebalancing



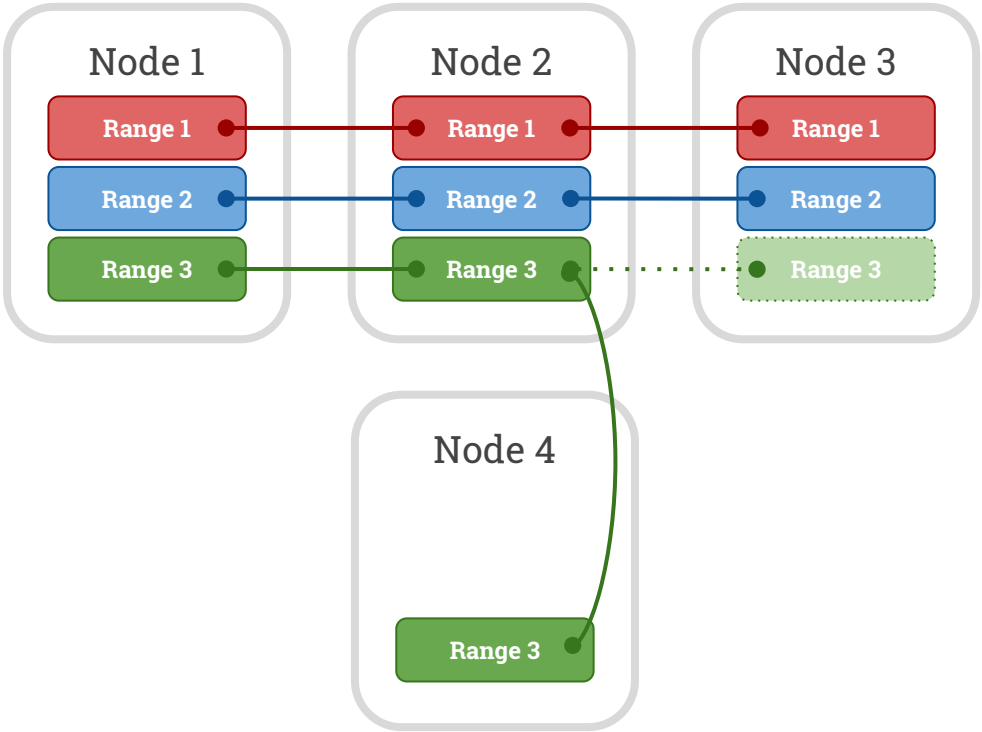
Adding a new (empty) node

Data Distribution: Rebalancing



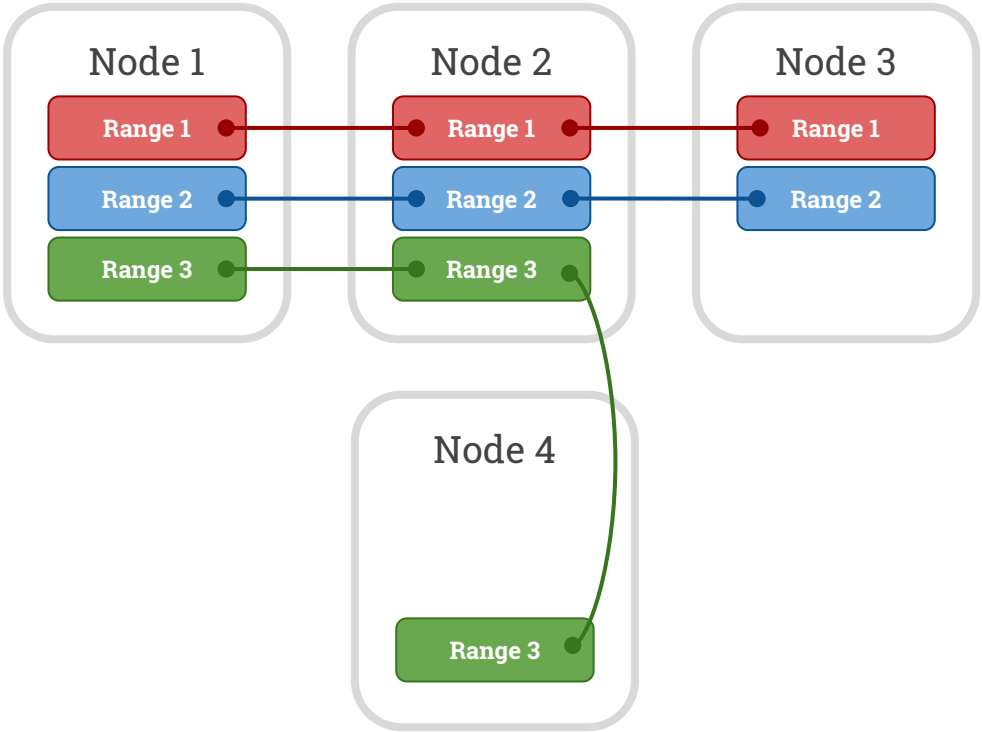
A new replica is allocated, data is copied.

Data Distribution: Rebalancing



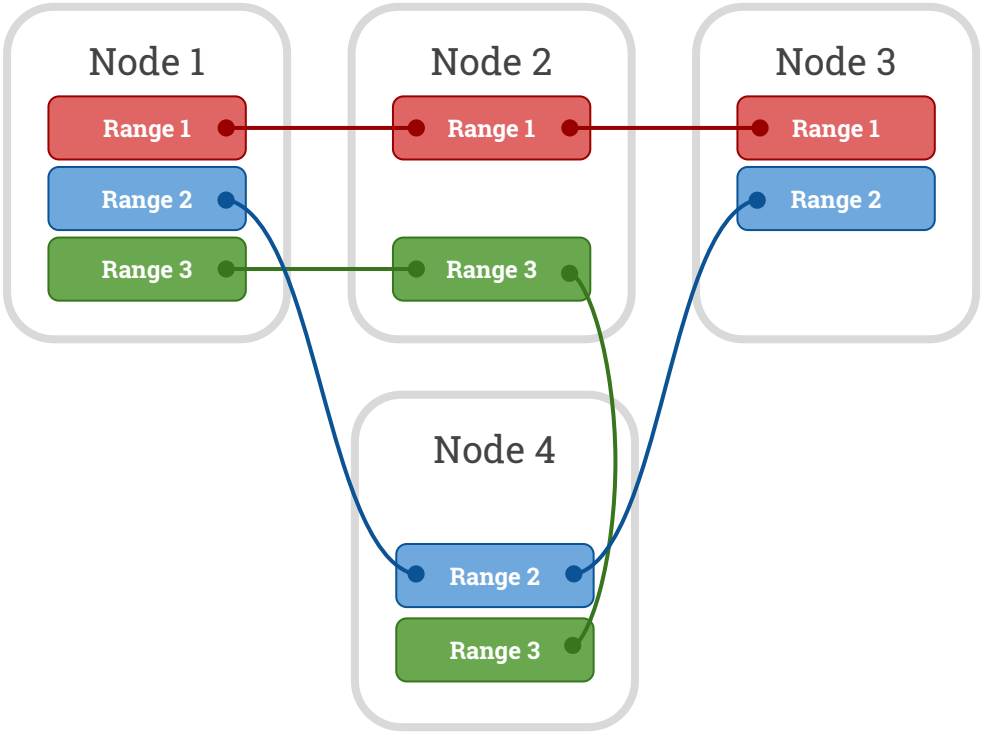
The new replica is made live, replacing another.

Data Distribution: Rebalancing



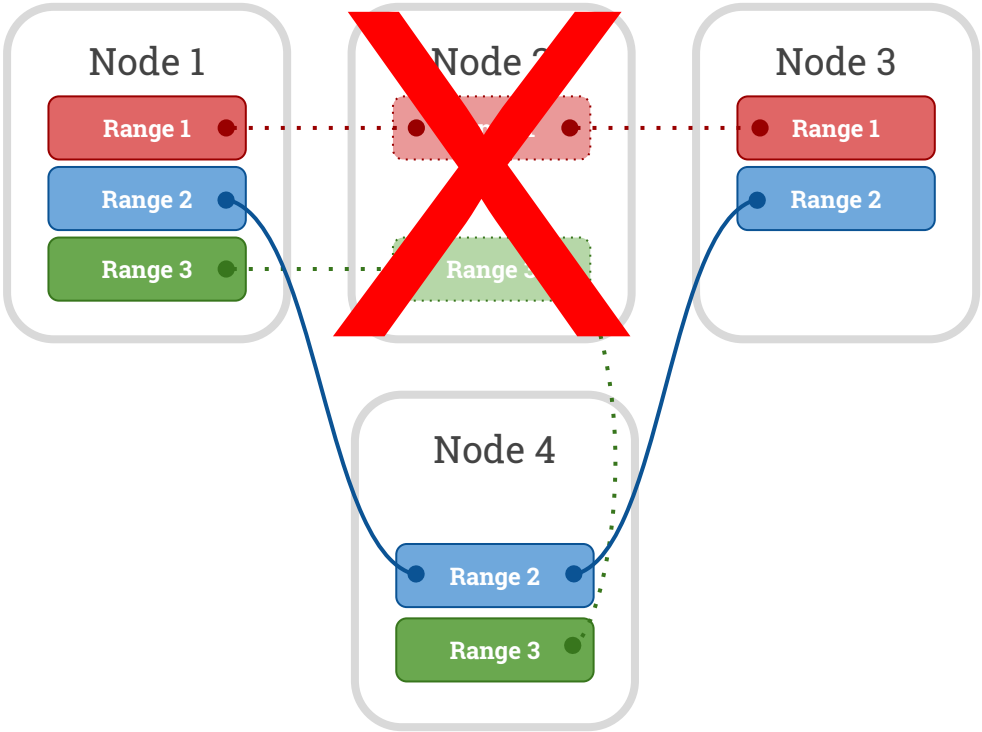
The old (inactive) replica is deleted.

Data Distribution: Rebalancing



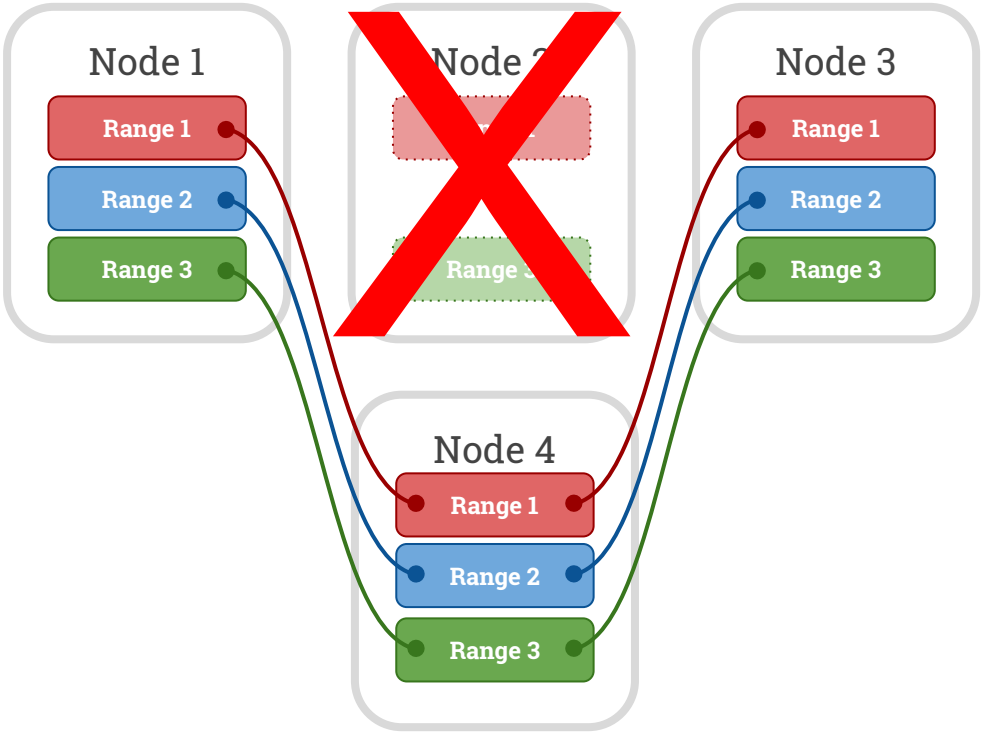
Process continues until nodes are balanced.

Data Distribution: Recovery



Losing a node causes recovery of its replicas.

Data Distribution: Recovery



A new replica gets created on an existing node to replace the lost replica.

Data Replication

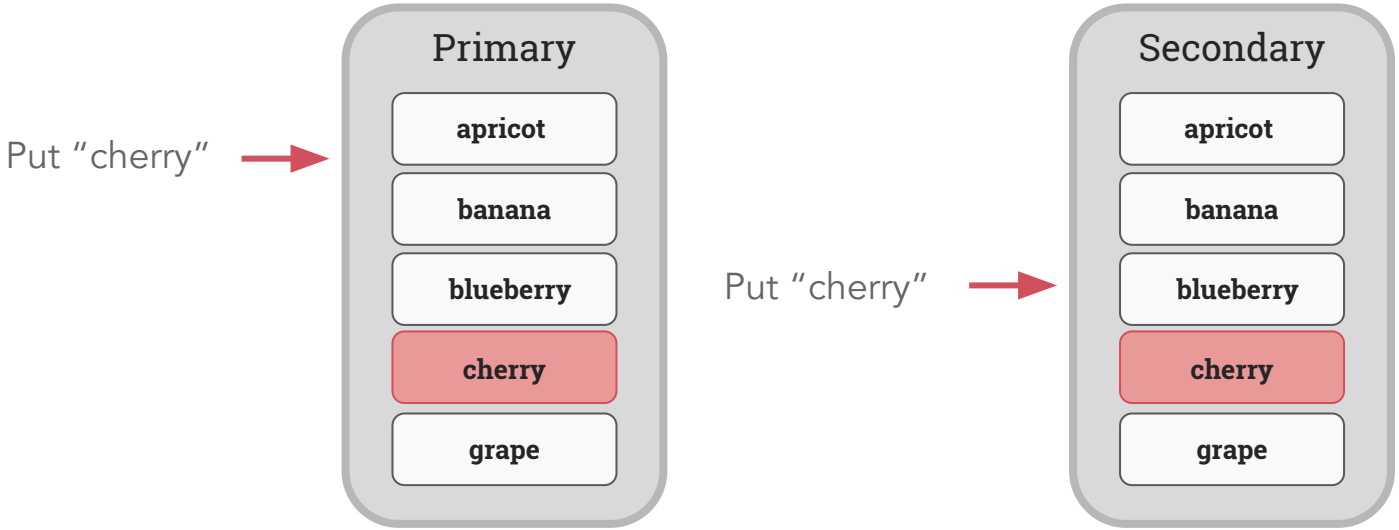
Keeping Remote Copies in Sync

Keeping Copies in Sync in SQL Databases

- Option one: cold backups
  - No real expectation of backups being fully up-to-date
- Option two: primary-secondary replication
  - All writes (and all reads that care about consistency) go to the “Primary” instance
  - All writes get pushed from the Primary to any Secondary instances

Primary/Secondary Replication

In theory, replicas contain identical copies of data: Voila!



Primary/Secondary Replication

- In practice, you have to choose
  - Asynchronous replication
    - Secondaries lag behind primary
    - Failover is likely to lose recent writes
  - Synchronous replication
    - All writes get delayed waiting for secondary acknowledgment
    - What do you do when your secondary fails?
- Failover is really hard to get right
  - How do clients know which instance is the primary at any given time?

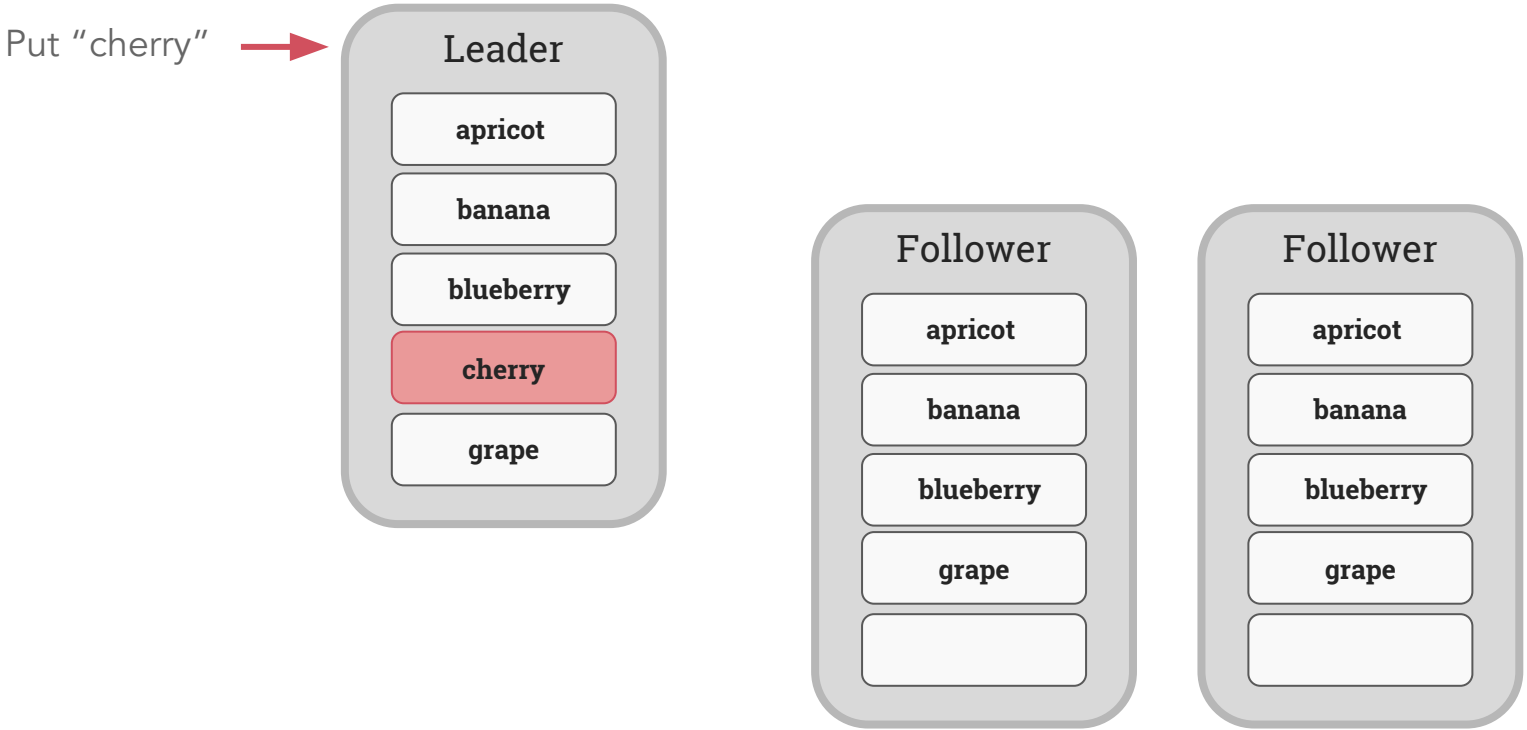
Keeping Copies in Sync in NoSQL Databases

- Most NoSQL systems are eventually consistent
- Very wide space of solutions, many different possible behaviors
  - Last write wins
  - Last write wins with vector clocks
  - Leader elections
  - Quorum reads and writes
  - Conflict-free replicated data types (CRDTs)

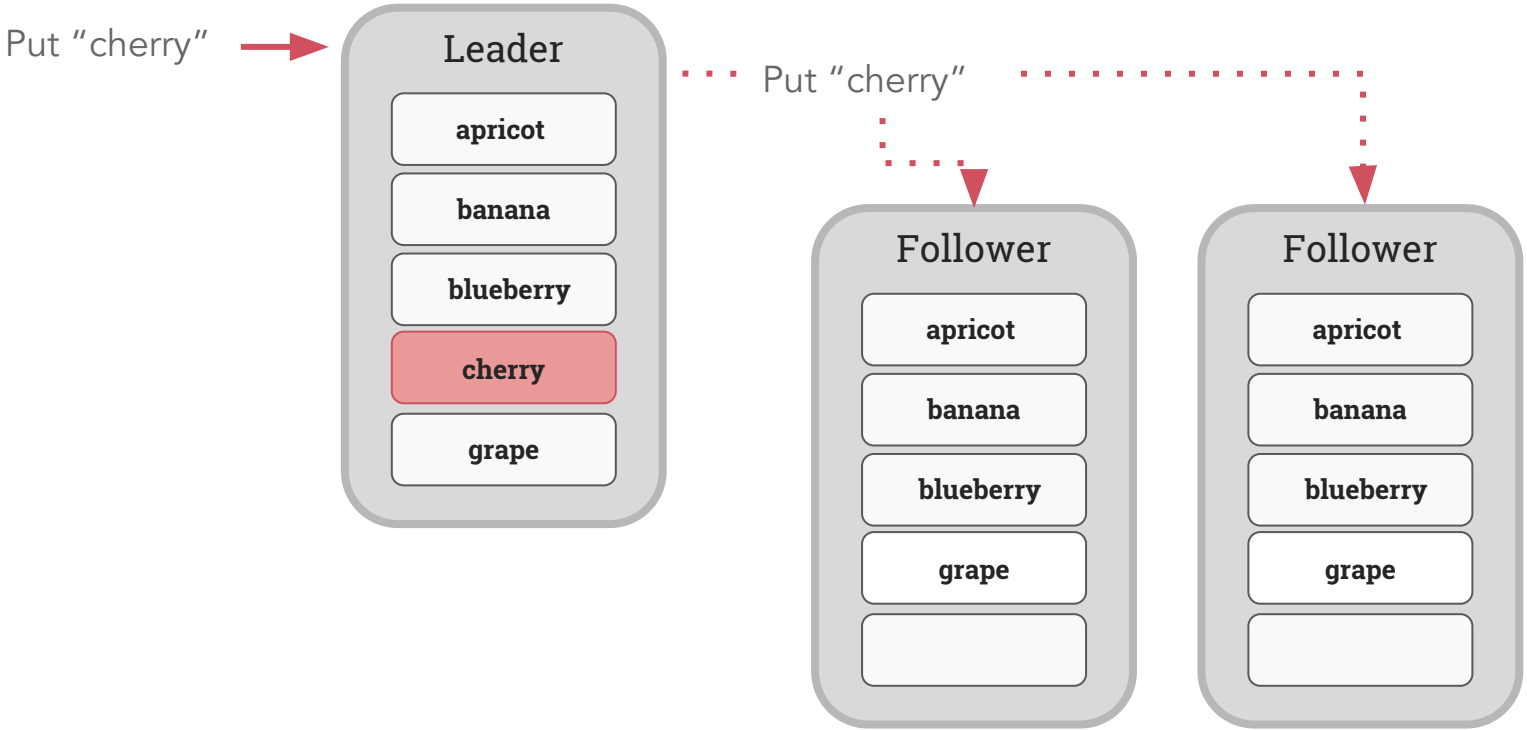
Keeping Copies in Sync in NewSQL Databases

- Use a distributed consensus protocol from the CS literature
  - Available protocols: Paxos, Raft, Zab, Viewstamped Replication, …
- At a high level:
  - Replicate data to N (often N=3 or N=5) nodes
  - Commit happens when a quorum have written the data

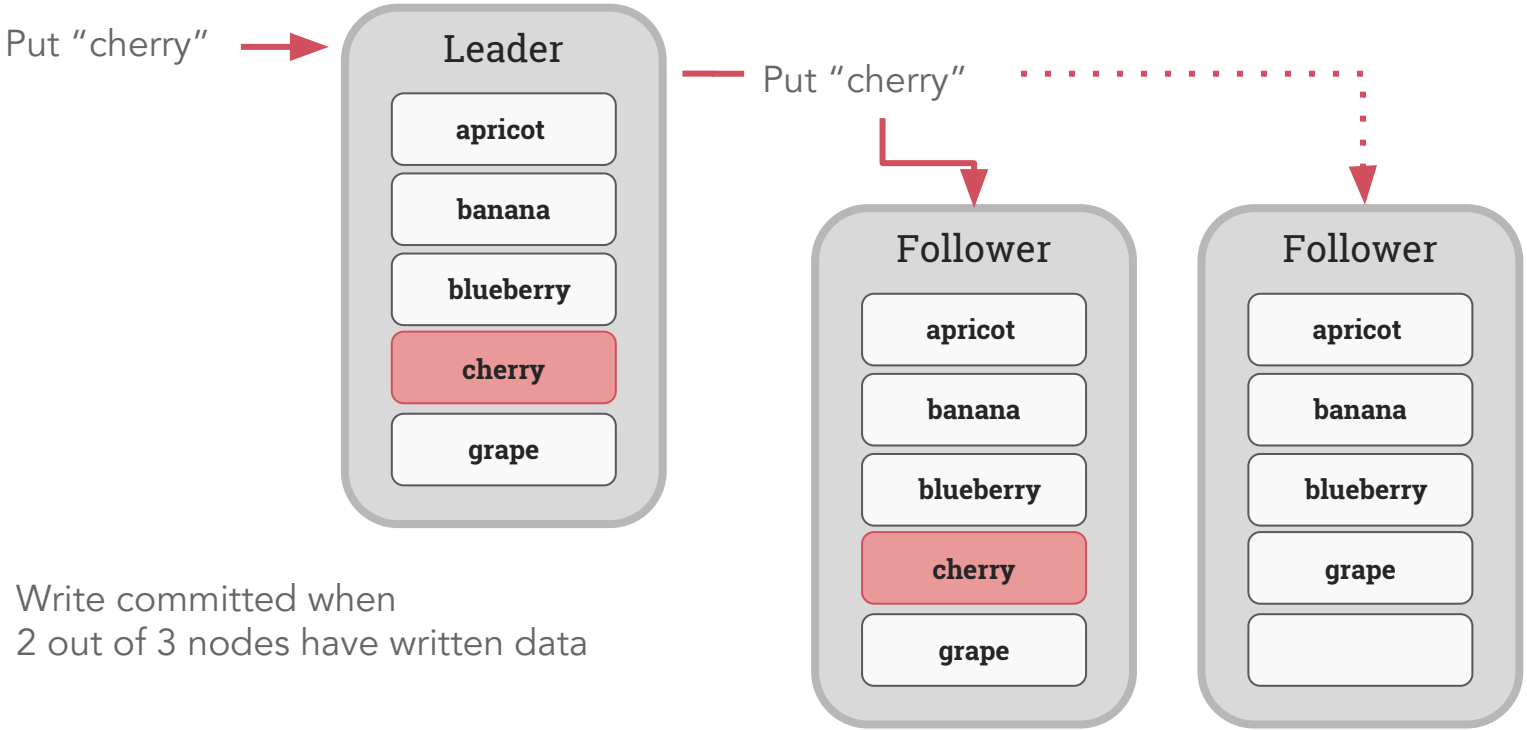
Distributed Consensus: Raft made simple



Distributed Consensus: Raft made simple

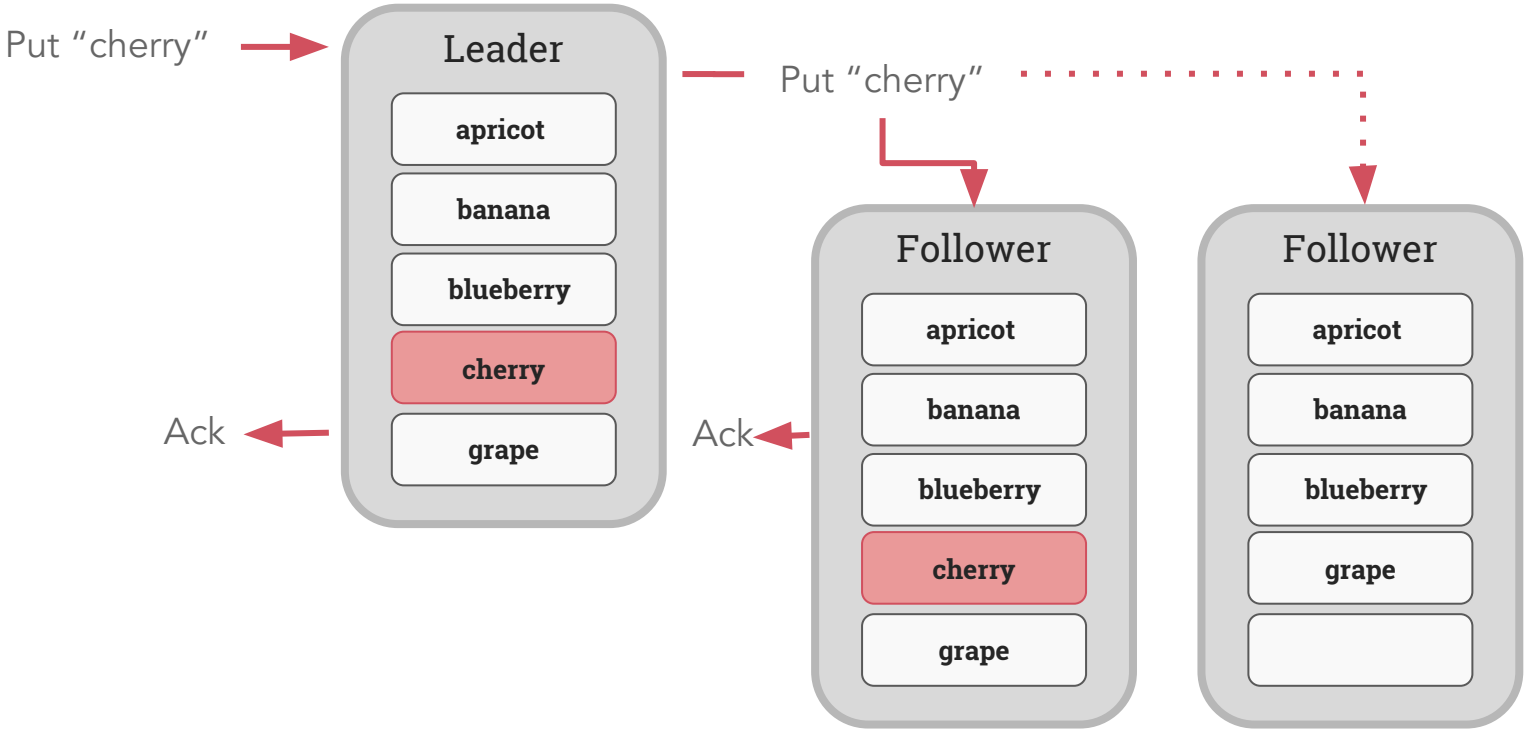


Distributed Consensus: Raft made simple



Write committed when
2 out of 3 nodes have written data

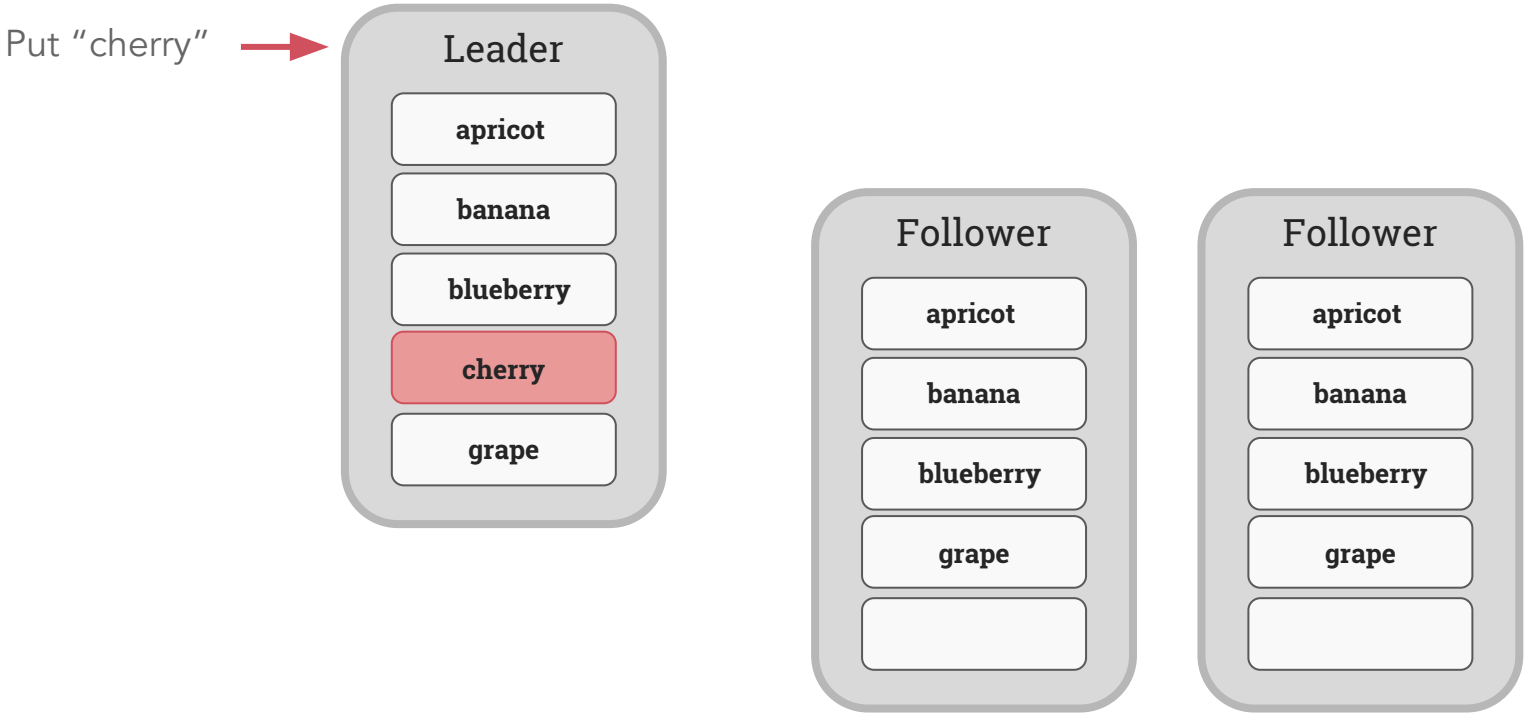
Distributed Consensus: Raft made simple



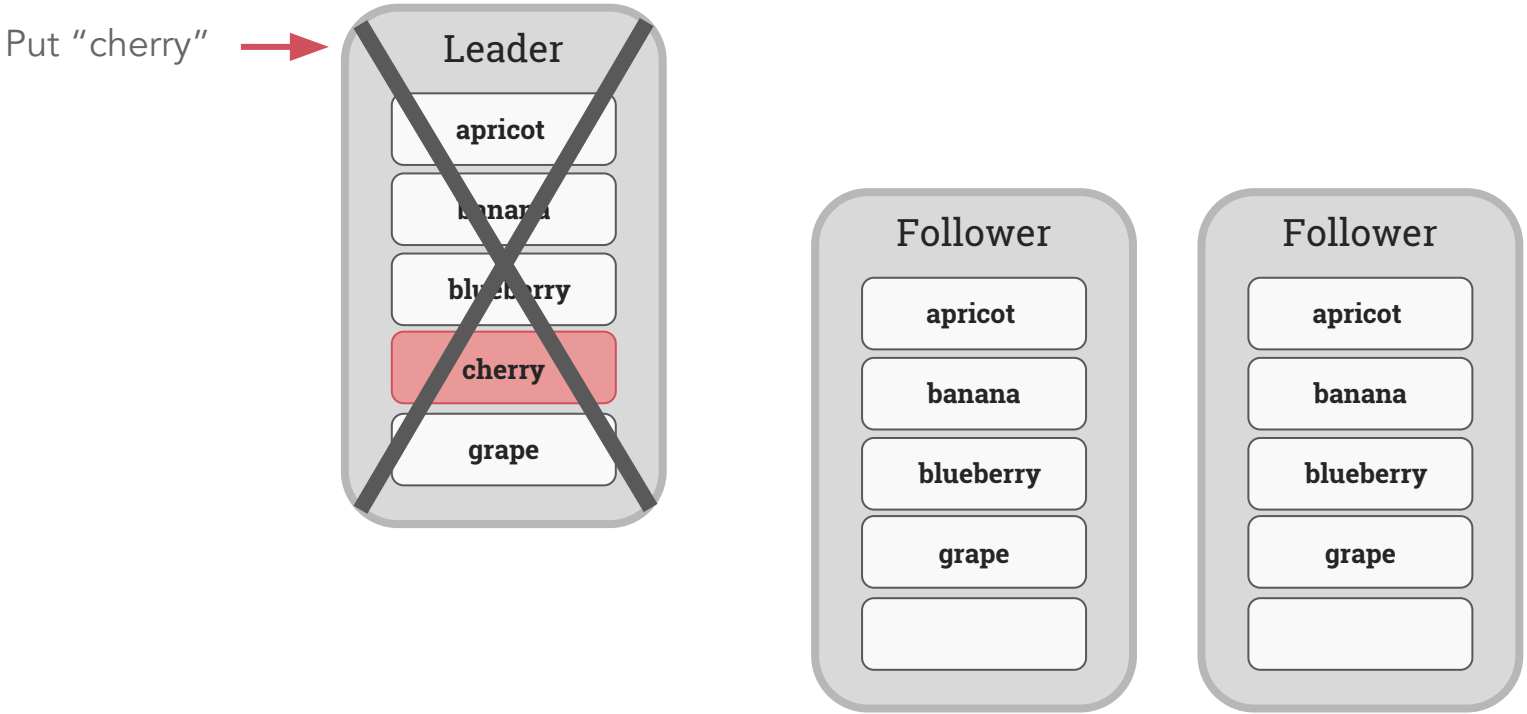
Distributed Consensus: Raft made simple

What happens during failover?

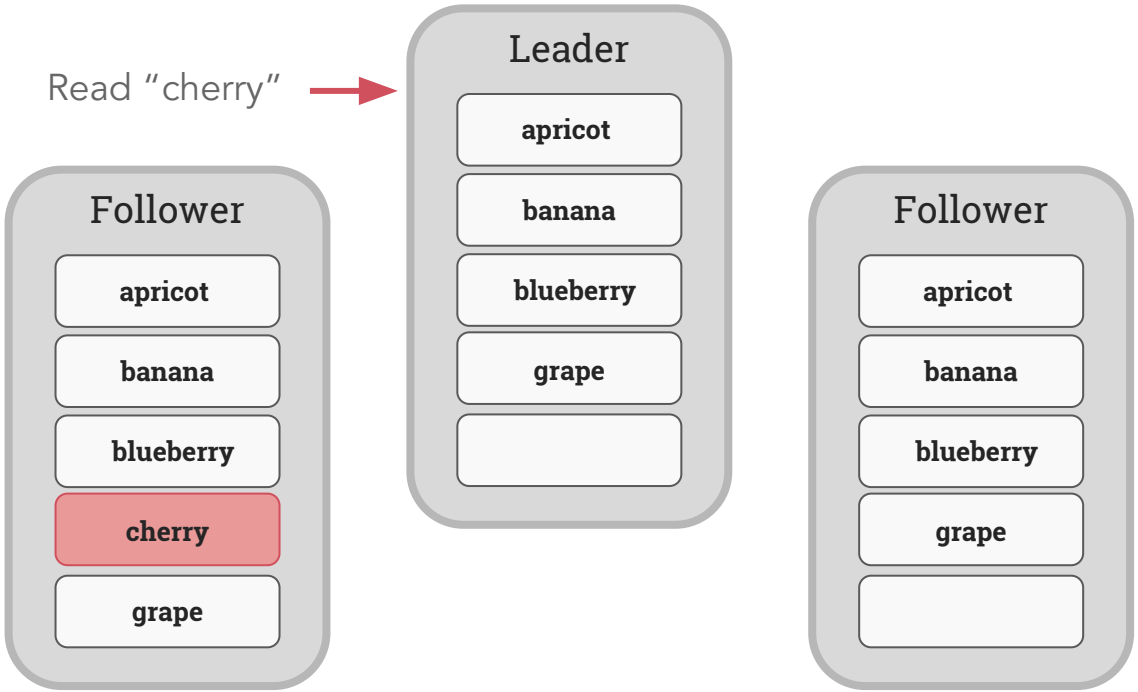
Distributed Consensus: Raft made simple



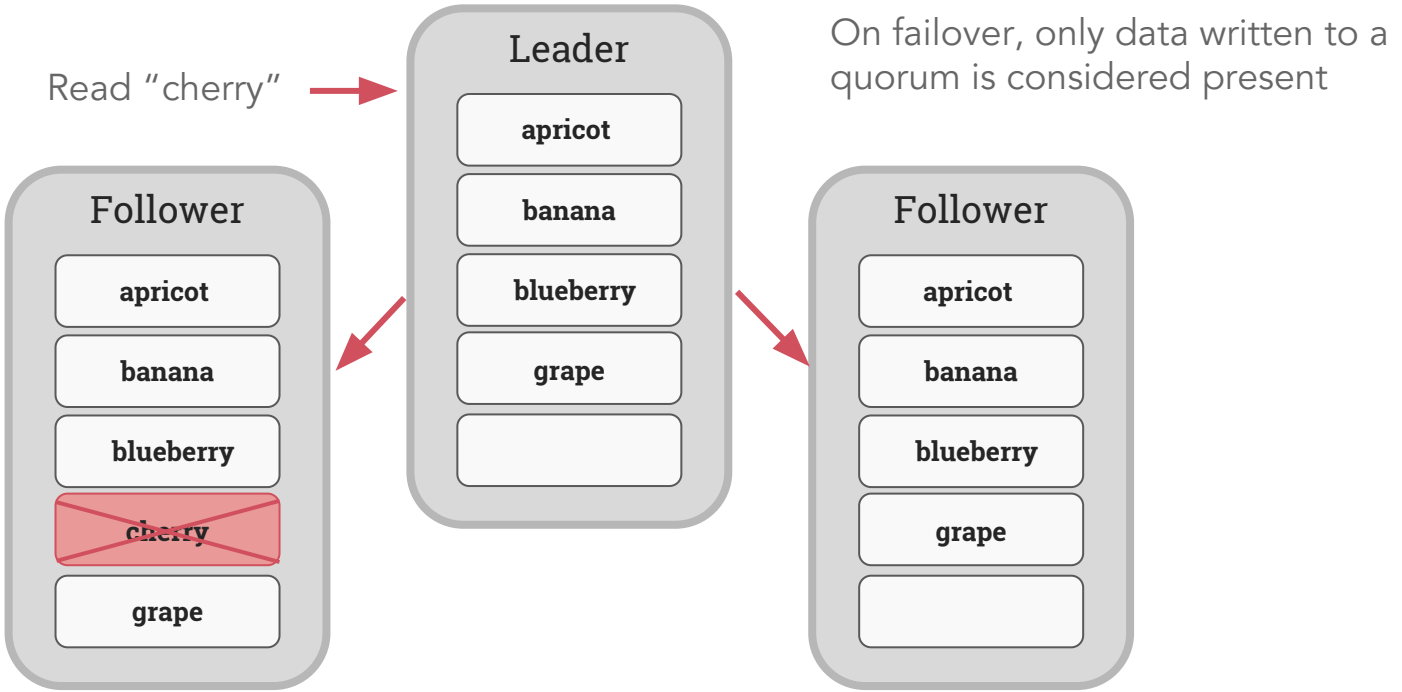
Distributed Consensus: Raft made simple



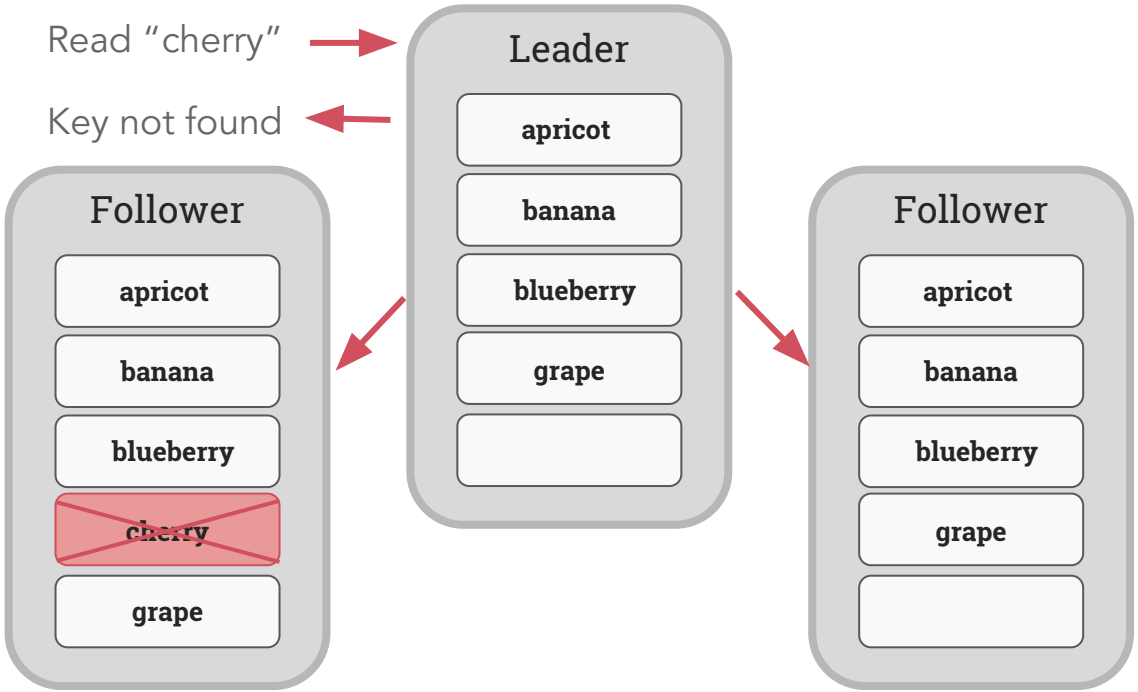
Distributed Consensus: Raft made simple



Distributed Consensus: Raft made simple



Distributed Consensus: Raft made simple



Consensus in NewSQL Databases

- **Run** one consensus group per range of data
- Many practical complications: member change, range splits, upgrades, scaling number of ranges

Transactions

How can they be made to work in a distributed system?

What is an ACID Transaction?

- Atomic - entire transaction either fully commits or fully aborts
- Consistent - moves from one valid DB state to another
- Isolated - concurrent transactions don’t interfere with each other
    - “Serializable” is needed for true isolation
- Durable - committed transactions won’t disappear

Transactions in SQL Databases

- Atomicity is bootstrapped off a lower-level atomic primitive: log writes
  - All mutations applied as part of a transaction get tagged with a transaction ID
  - “Commit” log entry marks the transaction as committed, making mutations visible
- Isolation is typically provided in one of two ways
  - Read/Write Locks
  - Multi-Version Concurrency Control (MVCC)

Transactions in SQL Databases: Read/Write Locks

- Pretty simple at a high level:
  - When you read a row, take a read lock on it
  - When you write a row, take a write lock on it
  - When you commit, release all locks
- Requires deadlock detection

Transactions in SQL Databases: MVCC

- Store a timestamp with each row
- When updating a row, don’t overwrite the old value
- When reading, return the most recent value from before the read
    - Allows access to historical data
    - Allows long-running read-only transactions to not block new writes
- Need to detect conflicts and abort conflicting transactions

Transactions in NoSQL Databases

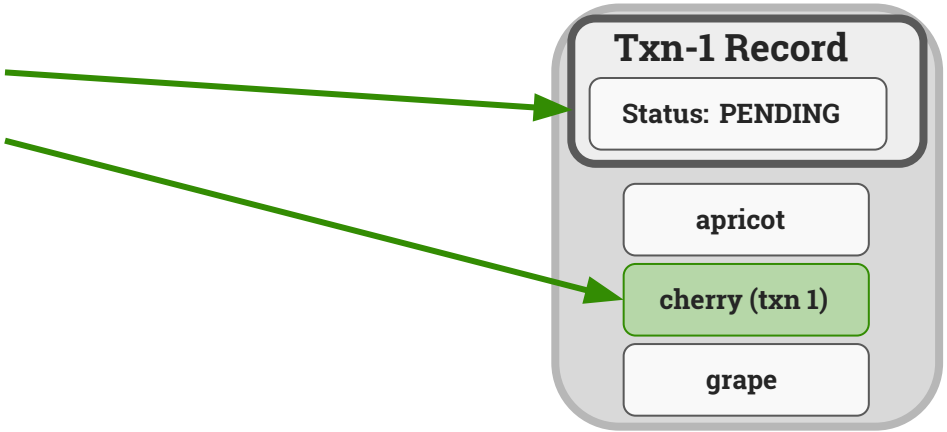
- Many NoSQL systems don't offer transactions at all
- This is one of the biggest tradeoffs that was made to improve scalability
- Those that do typically limit transactions to a single key
  - Single-key transactions don't require coordinating across shards
  - Can be implemented as just supporting a compare-and-swap operation

Transactions in NewSQL Databases

- Support traditional ACID semantics
- Atomicity is bootstrapped off distributed consensus
  - A “Commit” write to consensus system marks the transaction as committed
- Isolation can be handled similarly to single-node SQL databases
  - But the added latency and distributed state makes things tougher

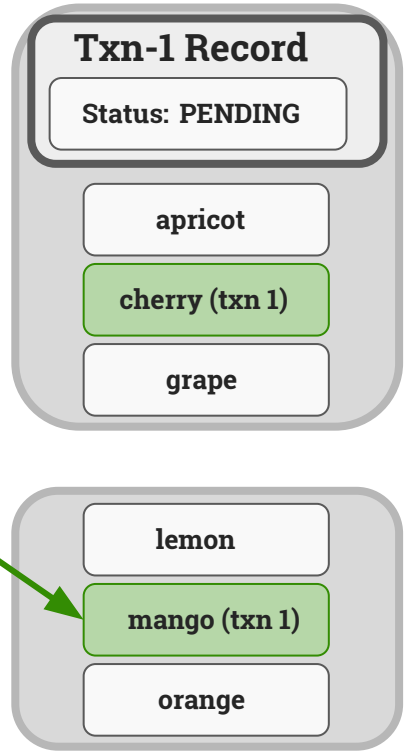
Distributed Transactions (CockroachDB)

1. Begin Txn 1
2. Put “cherry”



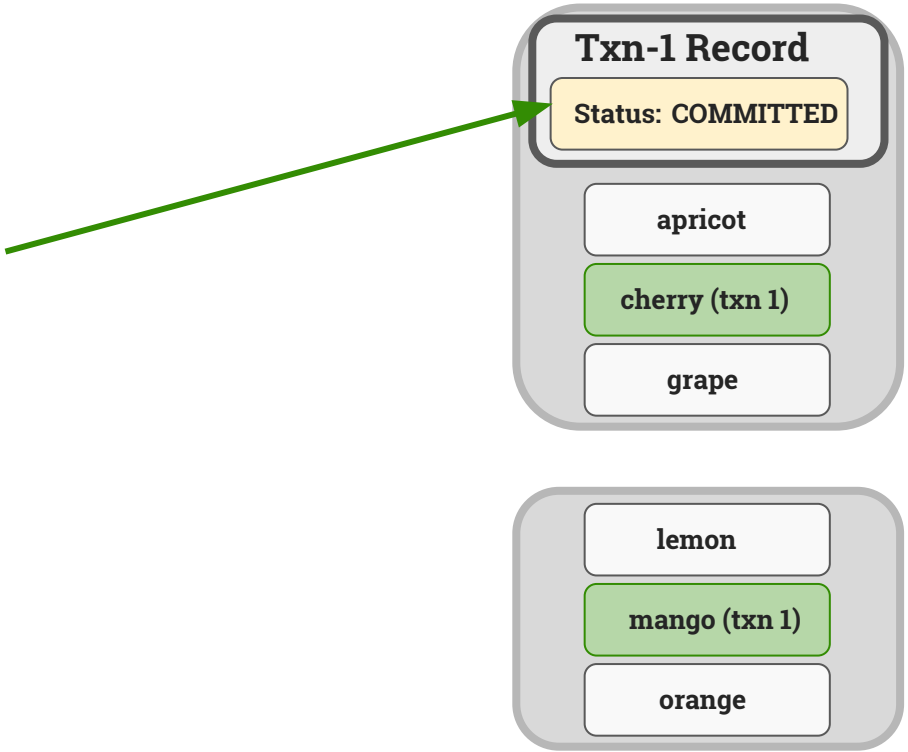
Distributed Transactions (CockroachDB)

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



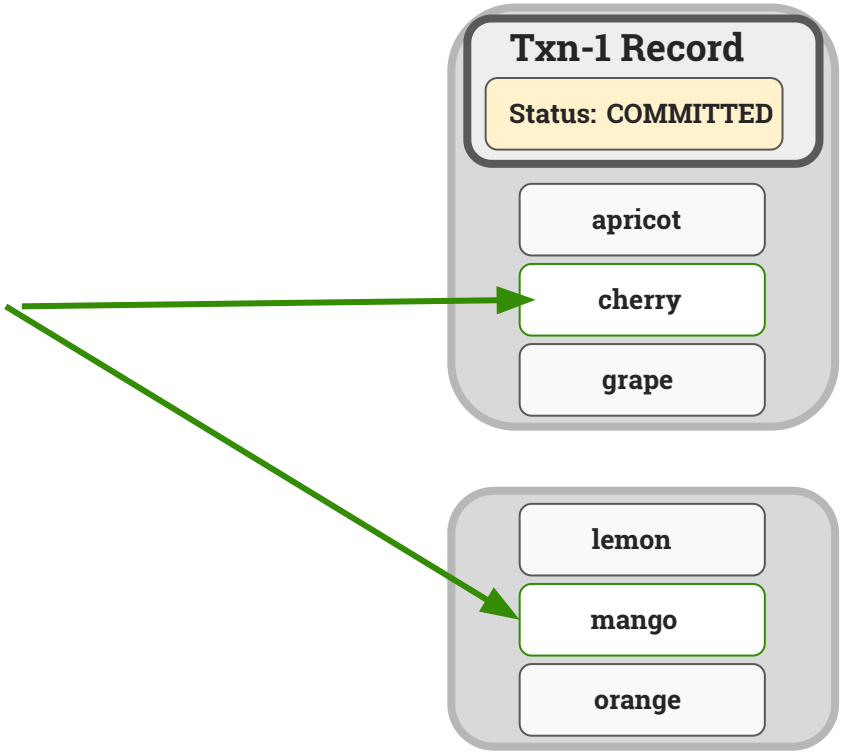
Distributed Transactions (CockroachDB)

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



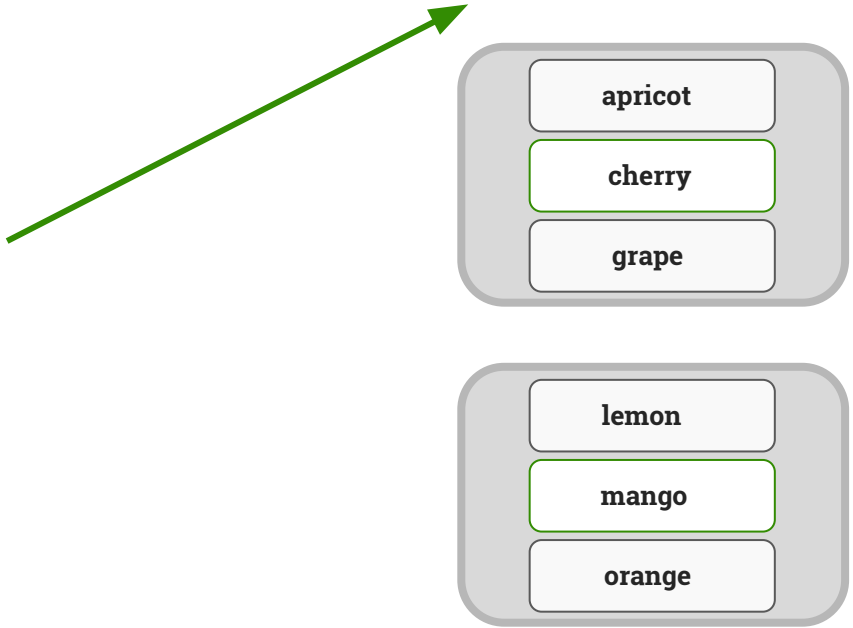
Distributed Transactions (CockroachDB)

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



Distributed Transactions (CockroachDB)

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

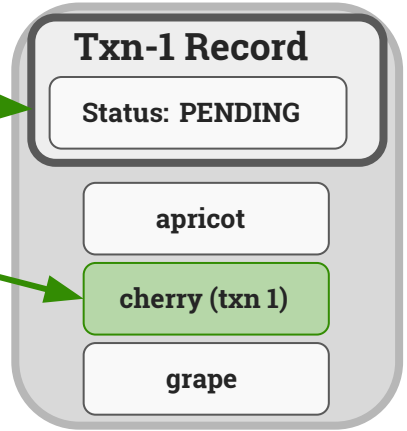


Distributed Transactions

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

Distributed Transactions (write conflict)

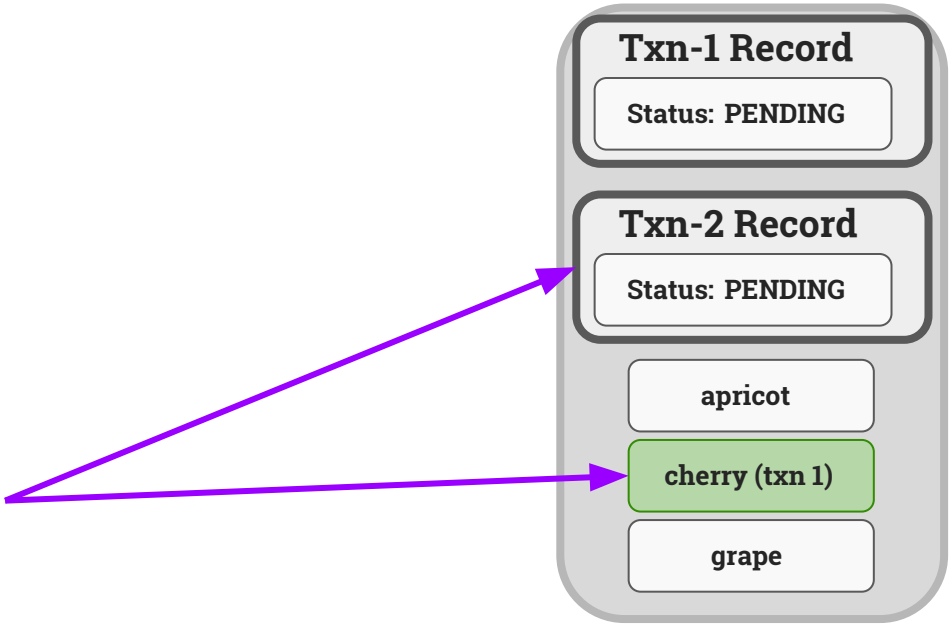
1. Begin Txn 1
2. Put “cherry”



Distributed Transactions (write conflict)

1. Begin Txn 1
2. Put “cherry”

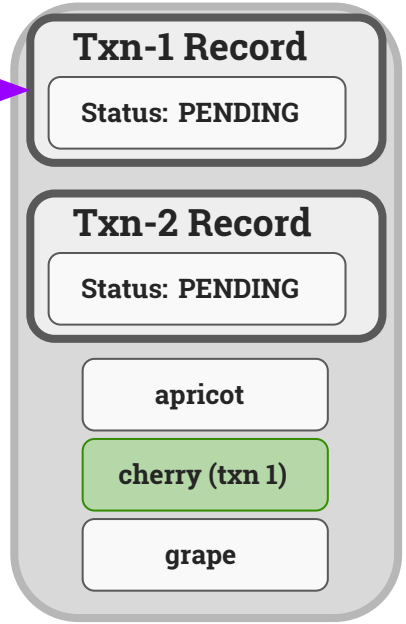
1. Begin Txn 2
2. Put “cherry”



Distributed Transactions (write conflict)

1. Begin Txn 1
2. Put “cherry”

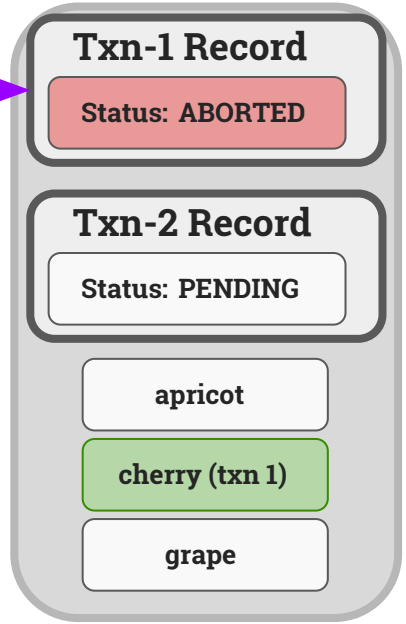
1. Begin Txn 2
2. Put “cherry”
   a. Check Txn 1 record



Distributed Transactions (write conflict)

1. Begin Txn 1
2. Put “cherry”

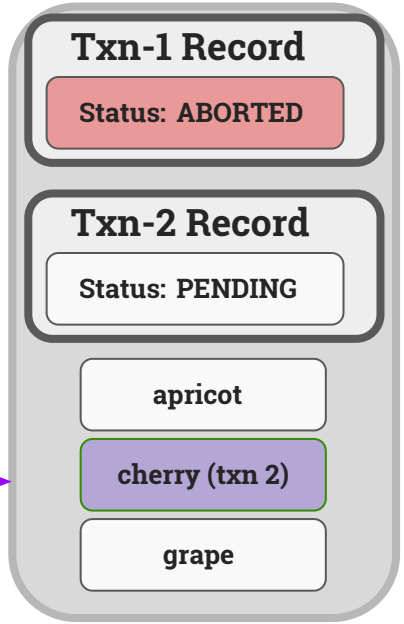
1. Begin Txn 2
2. Put “cherry”
   a. Check Txn 1 record
   b. Push Txn 1



Distributed Transactions (write conflict)

1. Begin Txn 1
2. Put “cherry”

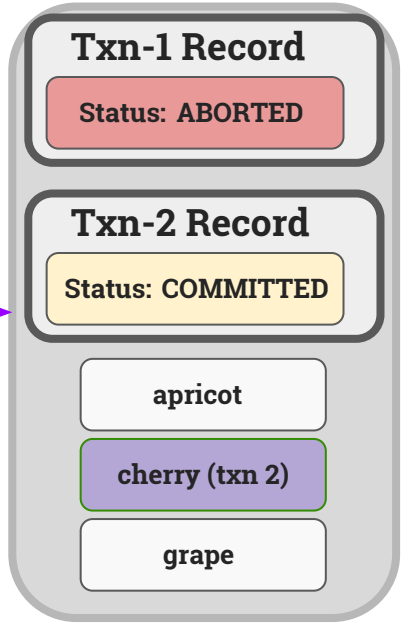
1. Begin Txn 2
2. Put “cherry”
   a. Check Txn 1 record
   b. Push Txn 1
   c. Update intent



Distributed Transactions (write conflict)

1. Begin Txn 1
2. Put “cherry”

1. Begin Txn 2
2. Put “cherry”
   a. Check Txn 1 record
   b. Push Txn 1
   c. Update intent
3. Commit Txn 2



Distributed Transactions

- Transaction atomicity is bootstrapped on top of Raft atomicity
- This description omitted MVCC and other types of conflicts
  - More details on the Cockroach Labs blog[1] or Spanner paper[2]

[1] https://www.cockroachlabs.com/blog/serializable-lockless-distributed-isolation-cockroachdb/
[2] https://research.google.com/archive/spanner.html

Summary

Summary

- Databases are best understood in the context in which they were built
- How does a NewSQL database work in comparison to previous DBs?
  - Distribute data to support large data sets and fault-tolerance
  - Replicate consistently with consensus protocols like Raft
  - Distributed transactions using consensus as the foundation

Thank You!

CockroachLabs.com/blog
github.com/cockroachdb/cockroach

alex@cockroachlabs.com / @alexwritescode