GENESIS

Michal Szymaniak
mips@infradepot.com

v2020-08-20

Genesis

- Zero-dependency storage platform
  - Replication
    - Goal: reliability and availability
    - How: Raft over RocksDBs
  - Sharding
    - Goal: capacity and throughput
    - How: BigTable-style range splitting
  - Change notification
    - Goal: consistency and integration
    - How: Zookeeper-style watches

Genesis

- Google-inspired storage layer
  - Critical problems solved once
    - Less code – Fewer bugs – Higher reliability
    - Re-usable service – Easier maintenance
  - Higher layers simpler and focused
    - Less complexity – Faster development
    - Less specialized expertise – Easier staffing
  - Lower layers abstract and disaggregated
    - Same API for physical / virtual / cloud storage – Natural data mobility
    - Decoupled resource lifetimes – Smooth HW / DC / Cloud operations

Genesis Use Cases

- Infrastructure metadata store
    - Distributed filesystem, package management, ...
- Massively shardable NoSQL
    - User metadata, event processing, ...
- Alternative implementation of popular APIs
    - DynamoDB, BigTable, ...
- Geo-distributed data storage
    - Multi-AZ, multi-region, multi-cloud, …
- Vehicle for physical data migrations
    - Zero-downtime moves between clouds, regions, datacenters, ...

Key-Value Store API

```
// Keys: N-tuples of binary strings, with key[0] determining the shard

// Read: point reads or range scans, optionally with key filtering
ReadResult = { Key, Value, Stat }
Status Read(ReadOptions, Store, Key, ReadResult)
Iterator NewIterator(ReadOptions, Store)

// Commit (aka MultiOp): atomic multi-key batch of operations
Op = CheckExists | CheckNotFound | CheckValue | CheckVersion |
     SetCounter | IncCounter | DecCounter |
     Write | WriteWithCounter | Delete | DeleteRange
Mutation = [ Op1, Op2, Op3, .. ]
Status Commit(CommitOptions, Store, Mutation)

// Watch: individual keys or key “patterns” across entire store
Status WatchKey(WatchOptions, Store, Key, WatchCallback)
Status WatchStore(WatchOptions, Store, KeyFilter, WatchCallback)
```

Hello World

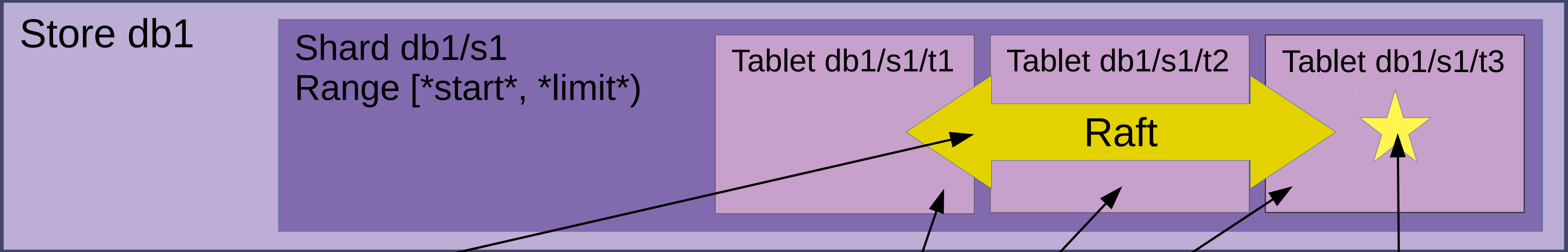
```
// Connect to Genesis
GrpcNetworkEnv env;
Client client(&env, FLAGS_genesis_bootstrap_servers);

// Open the store
StoreHandle store;
CHECK_OK(client.OpenStore(“db1”, &store));

// Write something
Mutation mutation;
const Key key(“user:alice”, “email”);
mutation.Write(key, “alice@foo.com”);
CHECK_OK(client.Commit(CommitOptions(), store, mutation));

// Read it back
std::string email;
CHECK_OK(client.Read(ReadOptions(), store, key, &email));
```

Logical View

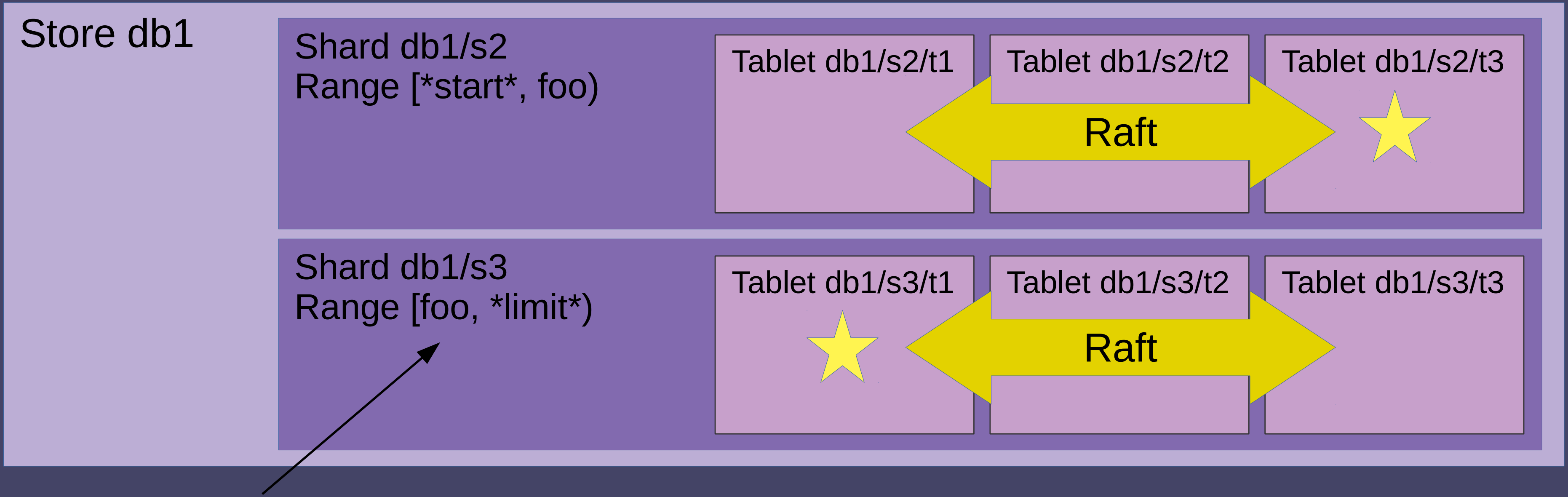


Raft
– Consensus protocol
– Simplified Paxos
– Replicates mutation log
– Needs only majority to work

Tablet == Shard Replica
– Voter or Observer
– Added / removed at will
(“shard re-configuration”)

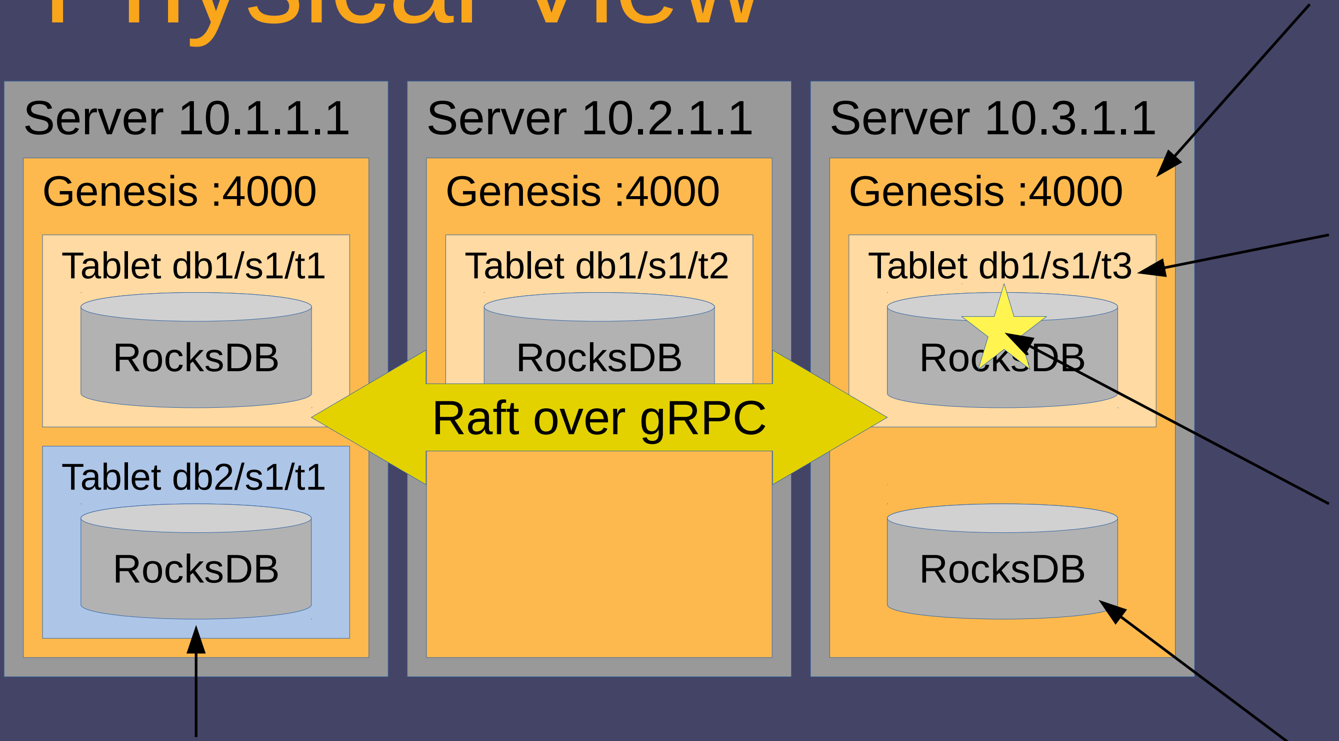
Raft Leader
– Elected among voters
– Replication driver
– Consistent state
– Automatic failover

Logical View: Multiple Shards



- – Old shard s1 split at key “foo”
- – New shards own adjacent key ranges
- – Separate tablets, Raft state, leaders
- – Sharding hidden from clients

Physical View



Tablet Server
– Owner of disk space
– Platform for tablets
– RPC dispatcher

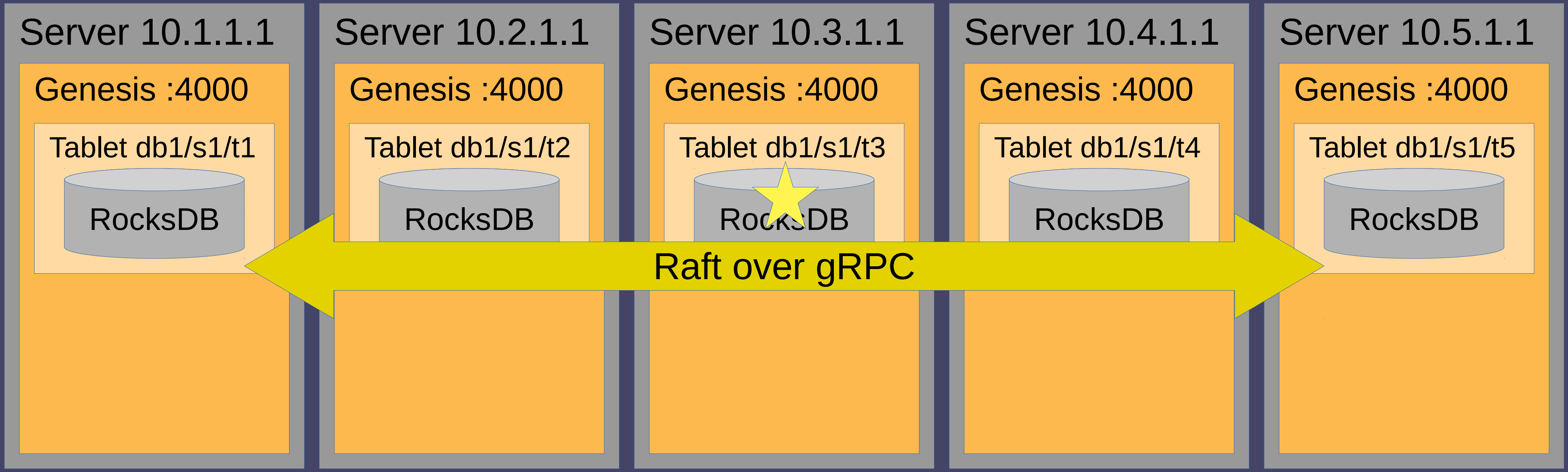
Tablet
– Owner of RocksDB
– Raft code and buffers
– RPC destination

Raft Leader
– Hot tablet handling all writes
– Commit calls + replication
– Leader flip == traffic switch

Tablet Storage
– “Unloaded” == inaccessible
– Tablet either being created, moving, or pending deletion

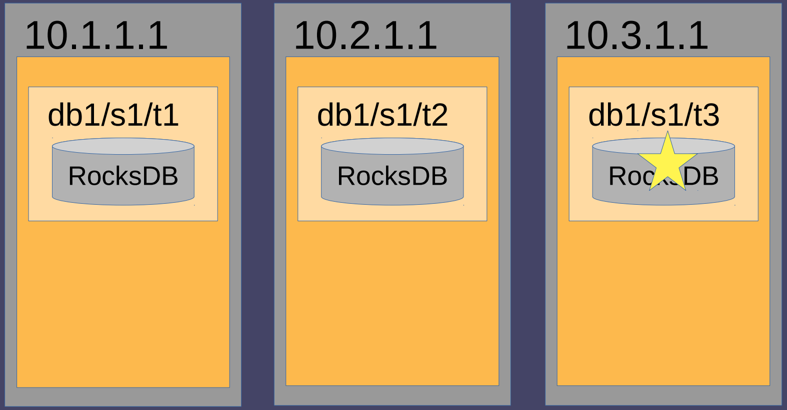
Colocated Tablet
– Many tablets per server
– Same or different store

Physical View: More Tablets



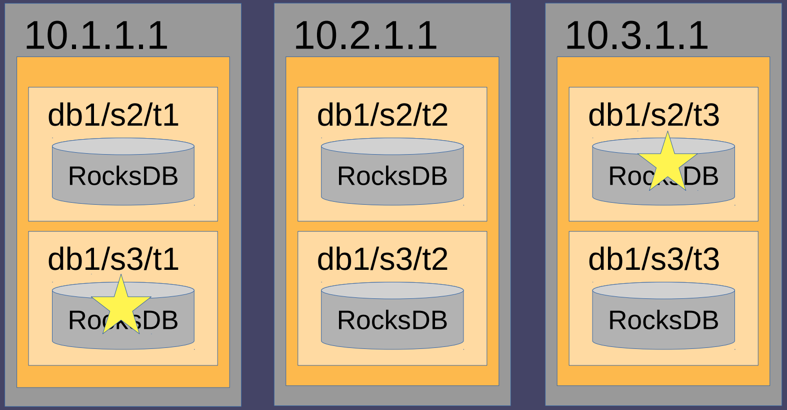
– More tablets == higher availability + higher read throughput
– Any odd number of voters, plus any number of observers
– Configured separately per shard

Physical View: More Shards



– More shards == more capacity + more throughput

Physical View: More Shards

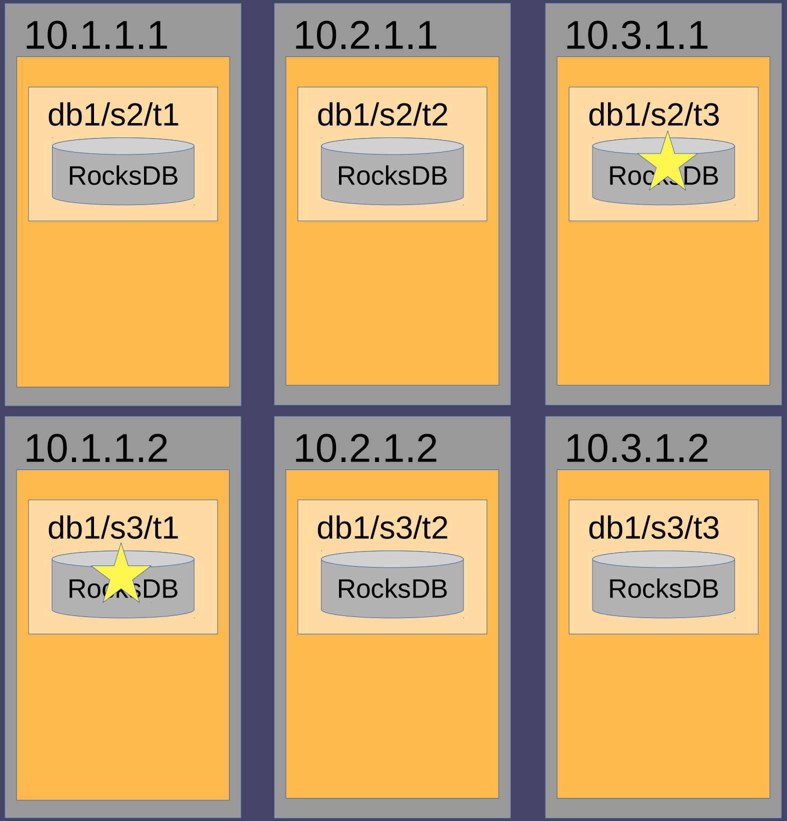


– More shards == more capacity + more throughput

– Shard split:

1) split each tablet into N >= 2 new ones

Physical View: More Shards



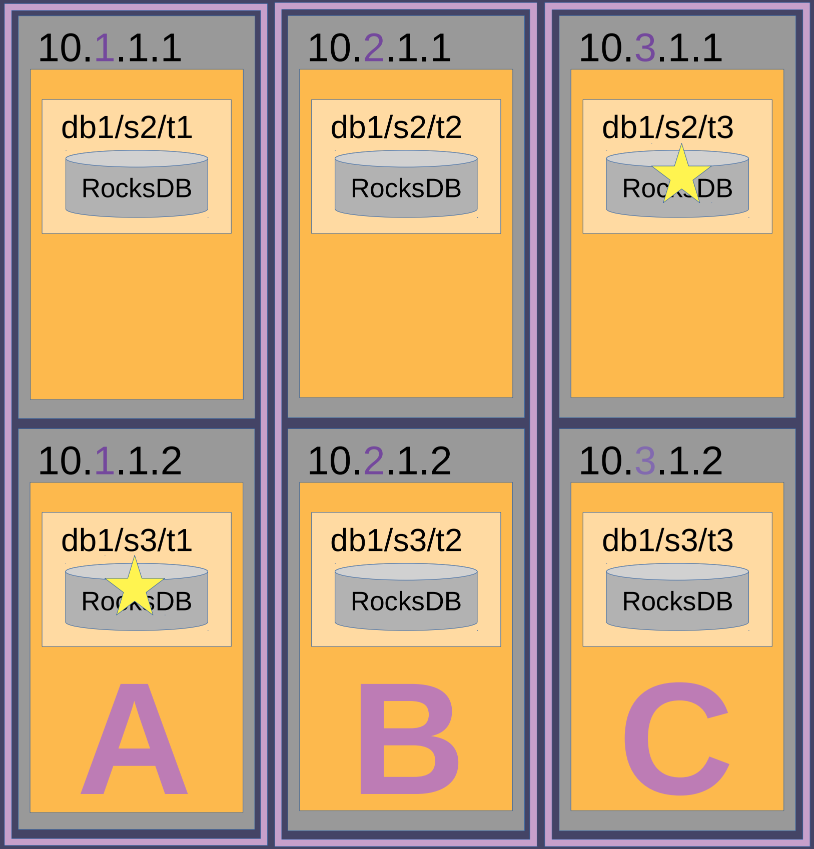
– More shards == more capacity + more throughput

– Shard split:

1) split each tablet into N >= 2 new ones

2) keep 1 new tablet in place, move other(s)

Physical View: More Shards



– More shards == more capacity + more throughput

– Shard split:

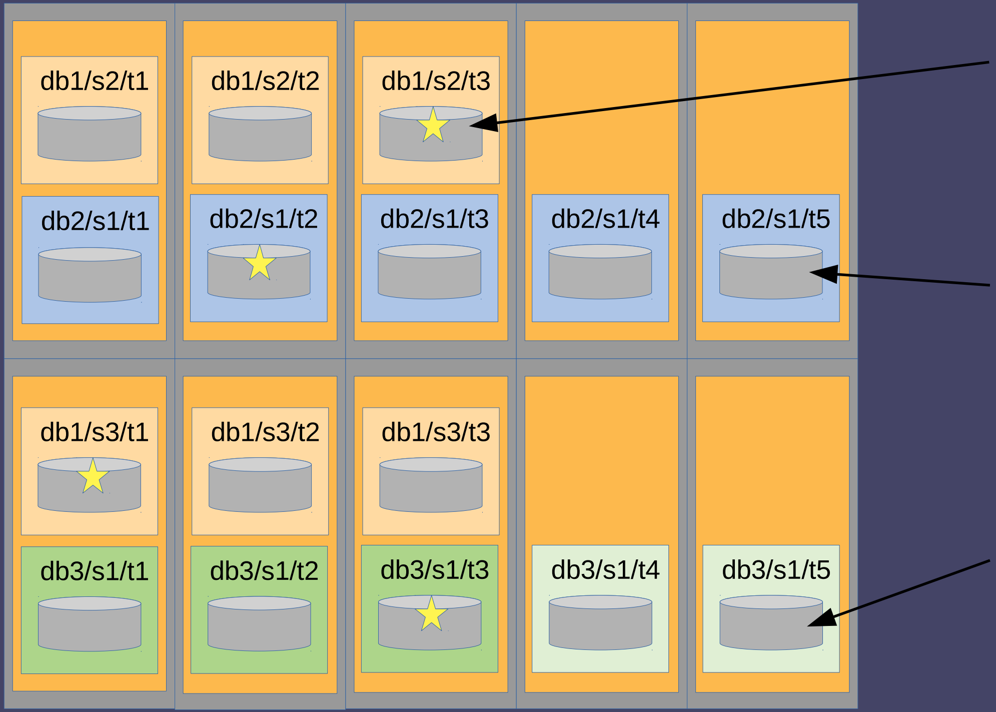
1) split each tablet into N >= 2 new ones

2) keep 1 new tablet in place, move other(s)

– New shards replicated across same server pools (typically, different availability zones)

– Tablets move within each server pool

Physical View: More Stores

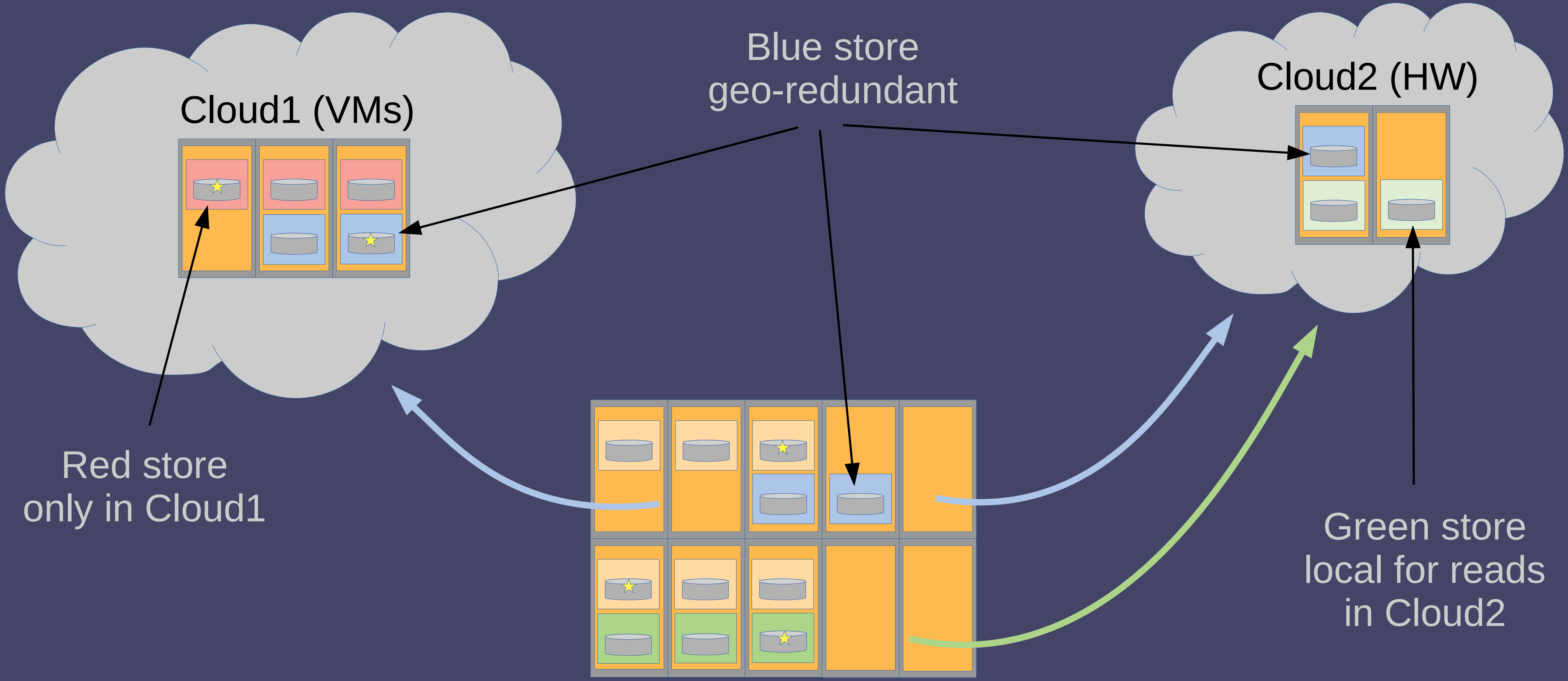


Orange store “db1”:
– 2 shards
– sharded for capacity and throughput

Blue store “db2”:
– 1 shard, 5 tablets
– higher availability than with 3 tablets

Green store “db3”:
– 1 shard, 5 tablets, including 2 observers
– higher read throughput than with 3 tablets

Physical View: More Locations



Cloud1 (VMs)

Blue store
geo-redundant

Cloud2 (HW)

Red store
only in Cloud1

Green store
local for reads
in Cloud2

Maintenance

- Store
  - Create / Delete
  - Backup / Restore
  - Update – change store configuration (ACLs, rate limits, ...)
- Shard
  - Split / Merge / AddObserver / RemoveObserver
  - Update – promote observers to voters or vice versa
  - SetLeader – force leader election for traffic rebalancing
- Tablet
  - Move – move tablets between servers for disk space rebalancing
  - Checkpoint – clone tablet storage for backup or offline processing
  - Recover – re-create lost tablet from another, in emergency

Performance

- Read Latency
    - 128b / 1kb / 16kb / 128kb / 1MB
    - 170 / 174 / 195 / 217 / 594 usec
    - Mostly leader RPC time
- Write Latency
    - 128b / 1kb / 16kb / 128kb / 1MB
    - 882 / 894 / 1148 / 2477 / 15429 usec
    - Mostly time of replication to quorum
- Throughput
    - Scales linearly with shard count
    - Reads per shard: up to 600,000 rps / 50 Gbps
    - Writes per shard: up to 30,000 rps / storage saturation