

# **Highly Available RDF Data at Enterprise Scale**

## **The Bigdata<sup>®</sup> High-Availability Architecture**

# Agenda

- Background
- Goals
- bigdata<sup>®</sup> Scale-Out Overview
- Current HA Architecture – Shared Nothing
- Future HA Architecture – Shared Disk

# What is “big data?”

- Big data is a new way of thinking about and processing massive data.
  - Petabyte scale
  - Commodity hardware
  - Distributed processing

# The origins of “big data”

- Google published several inspiring papers that have captured a huge mindshare.
  - GDFS, Map/Reduce, bigtable.
- Competition has emerged among “cloud as service” providers:
  - E3, S3, GAE, BlueCloud, Cloudera, etc.
- An increasing number of open source efforts provide cloud computing frameworks:
  - Hadoop, Bigdata, CouchDB, Hypertable, mg4j, Cassandra....

# Who has “big data?”

- USG
- Finance
- Biomedical & Pharmaceutical
- Large corporations
- Major web players
- High energy physics

<http://dataspora.com/blog/tipping-points-and-big-data/>

<http://www.wired.com/wired/archive/14.10/cloudware.html>

<http://radar.oreilly.com/2009/04/linkedin-chief-scientist-on-analytics-and-bigdata.html>

<http://www.nature.com/nature/journal/v455/n7209/full/455001a.html>

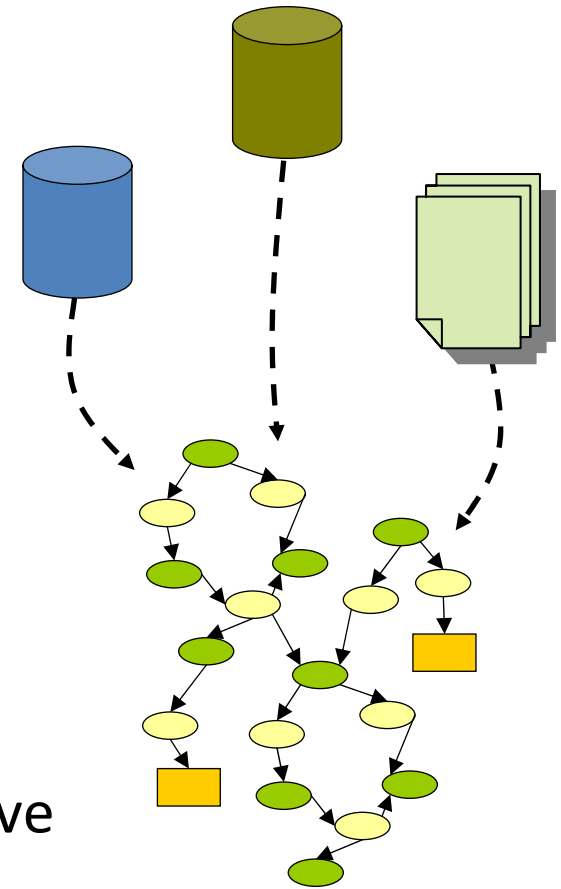
<http://queue.acm.org/detail.cfm?id=1563874>

# Technologies that go “big”

- Distributed file systems
  - GFS, S3, HDFS
- Map / reduce
  - Lowers the bar for distributed computing
  - Good for data locality in *inputs*
    - E.g., documents in, hash-partitioned full text index out.
- Sparse row stores
  - High read / write concurrency using atomic row operations
  - Basic data model is
    - { primary key, column name, timestamp } : { value }

# The killer big data application

- Clouds + “Open” Data = Big Data Integration
- Critical advantages
  - *Fast* integration cycle
  - Open standards
  - Integrates heterogeneous data, linked data, structured data.
  - Opportunistic exploitation of data, including data which can not be integrated quickly enough today to derive its business value.



# Goals

- Bring “Big Data” to the Semantic Web
- Remove any realistic scaling limits
- Provide service failover and guaranteed data availability
- Retain historical state of data
  - Infinite history if necessary
  - Eliminate need for backups
- Decouple compute power from disk
  - Allow arbitrary compute concurrency

# Removing Scaling Limits

## Dynamic Key-Range Partitioning

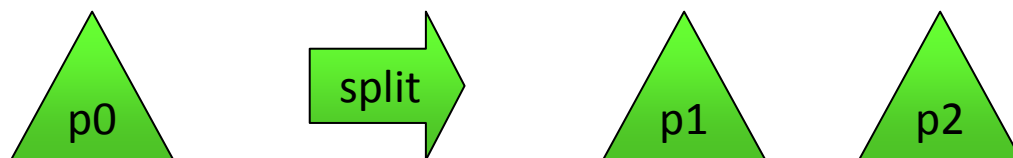
# Data Partitioning Overview

- Data partitioning the only game in town for massive scale
- Need to choose a partitioning method
- Hash partitioning easier to implement, but re-balancing and re-partitioning is a headache
- Dynamic key-range partitioning eases incremental growth and re-balancing over time

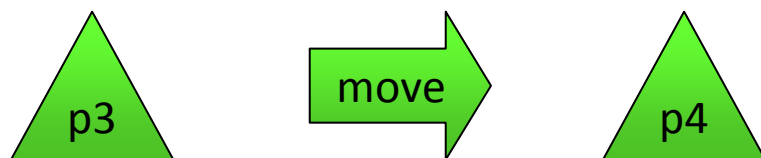
# Dynamic Key-Range Partitioning

- bigdata<sup>®</sup> is all about dynamically key-range partitioned indices
- Indices broken down into shards at runtime on-demand rather than statically partitioned up-front
- As shards grow they are automatically split into smaller shards
- Index shards managed by data services running on a cluster of commodity hardware
- Shards move between machines based on relative load
- Clients locate data via a “shard locator service” and then interact with data services directly

# Dynamic Key Range Partitioning



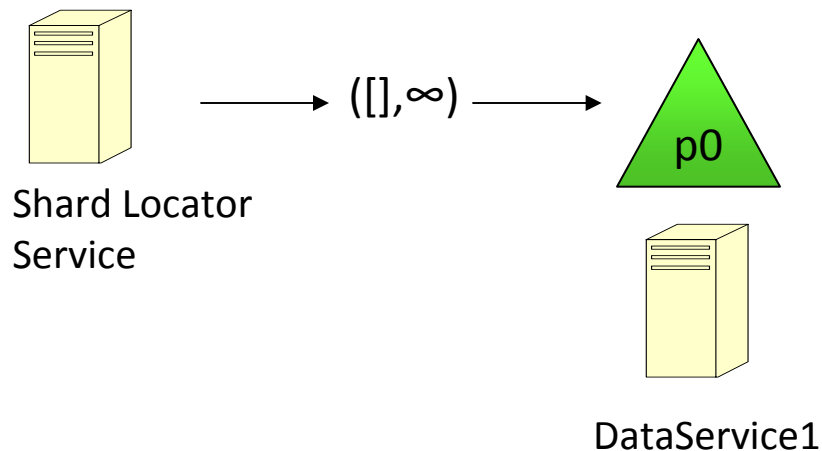
Splits break down the indices dynamically as the data scale increases.



Moves redistribute the data onto existing or new nodes in the cluster.

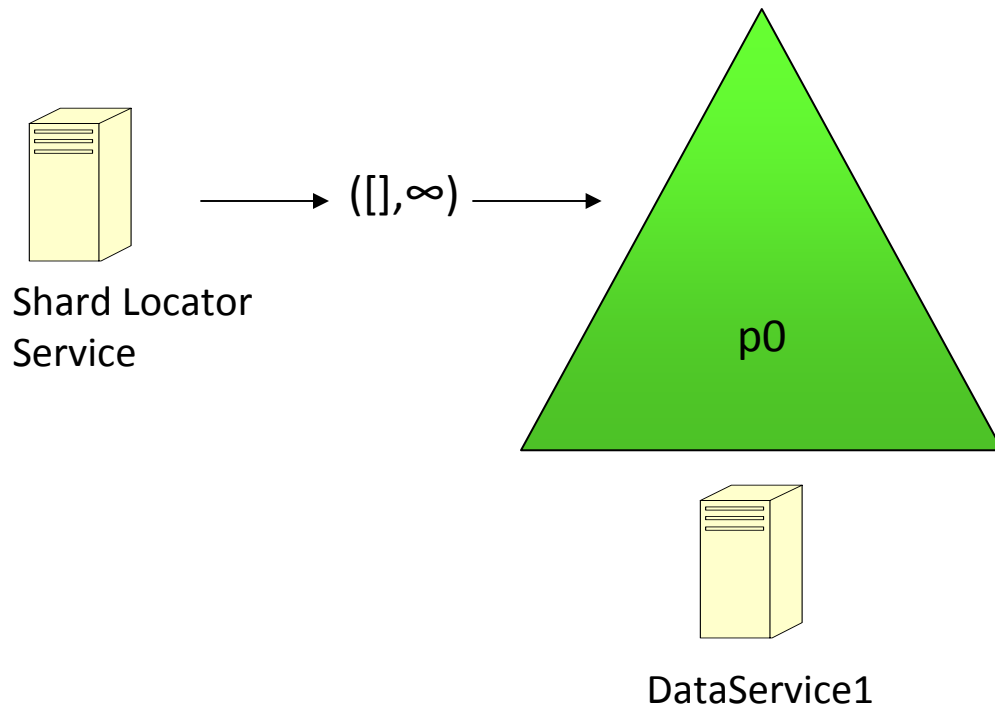
# Dynamic Key Range Partitioning

Initial conditions place the first shard on an arbitrary data service representing the entire index key range.



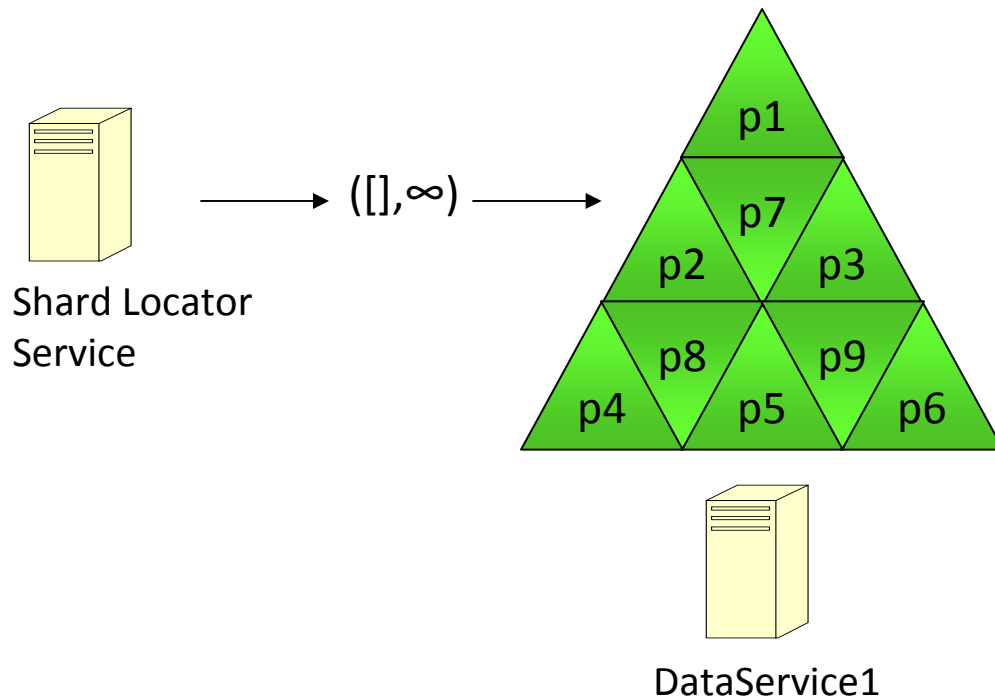
# Dynamic Key Range Partitioning

Writes cause the shard to grow. Eventually its size on disk exceeds a preconfigured threshold.

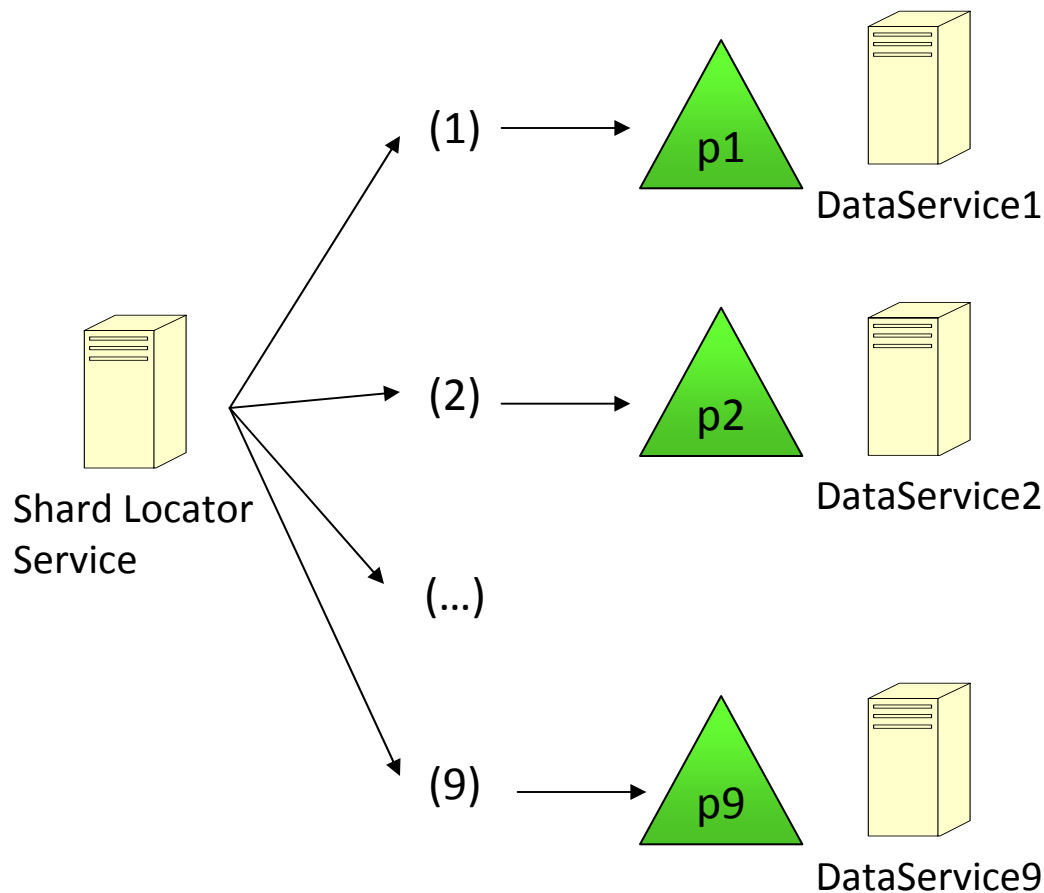


# Dynamic Key Range Partitioning

Instead of a simple two-way split, the initial shard is “scatter-split” so that all data services can start managing data.



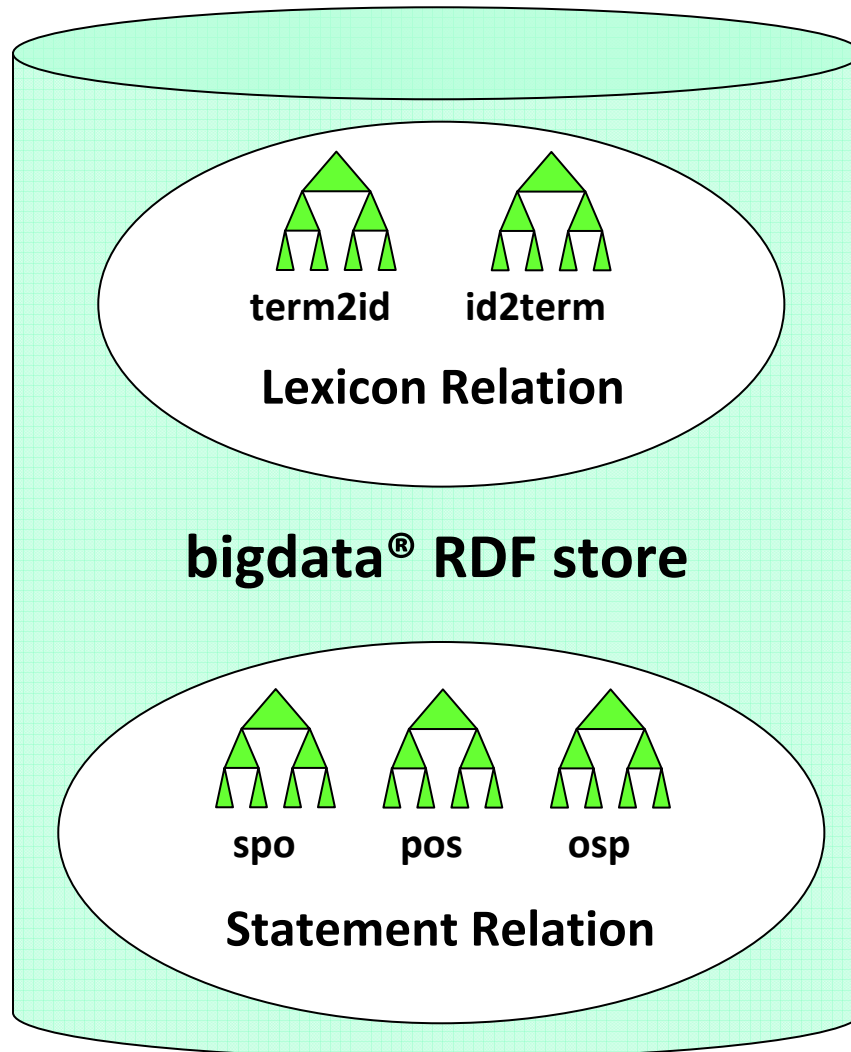
# Dynamic Key Range Partitioning




The newly created shards are then dispersed around the cluster.

Subsequent splits are two-way and moves occur based on relative server load (decided by Load Balancer Service).

# bigdata<sup>®</sup> Architecture Overview



- bigdata<sup>®</sup> is all about dynamically key-range sharded indices: 
- bigdata<sup>®</sup> RDF store is modeled using these indices
- Scale-out indices gives us scale-out RDF

# Covering Indices

- Two relations: lexicon and statements
- Two lexicon indices
  - term2id index { term : id }
  - id2term index { id : term }
- Three (triples) or six (quads) statement indices
  - SPO, OSP, POS
- Relations and indices are namespaced to correspond to a single triple or quad store in the bigdata<sup>®</sup> federation

# Covering Indices

Lexicon
term : Value
termId : long



Statement
s : long
p : long
o : long
c : long
type : StatementType



The triples and provenance modes use three statement indices.



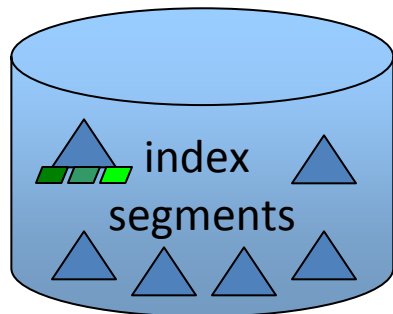
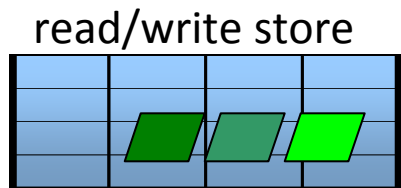
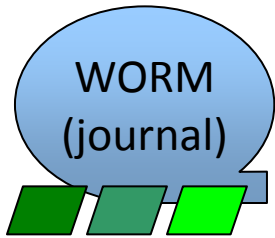
The quad store uses six indices whose keys are based on permutations of (s,p,o,c).

«enumeration» StatementType
Axiom
Inferred
Explicit

# Service Failover and Guaranteed Data Availability

## Shared-Nothing HA Architecture

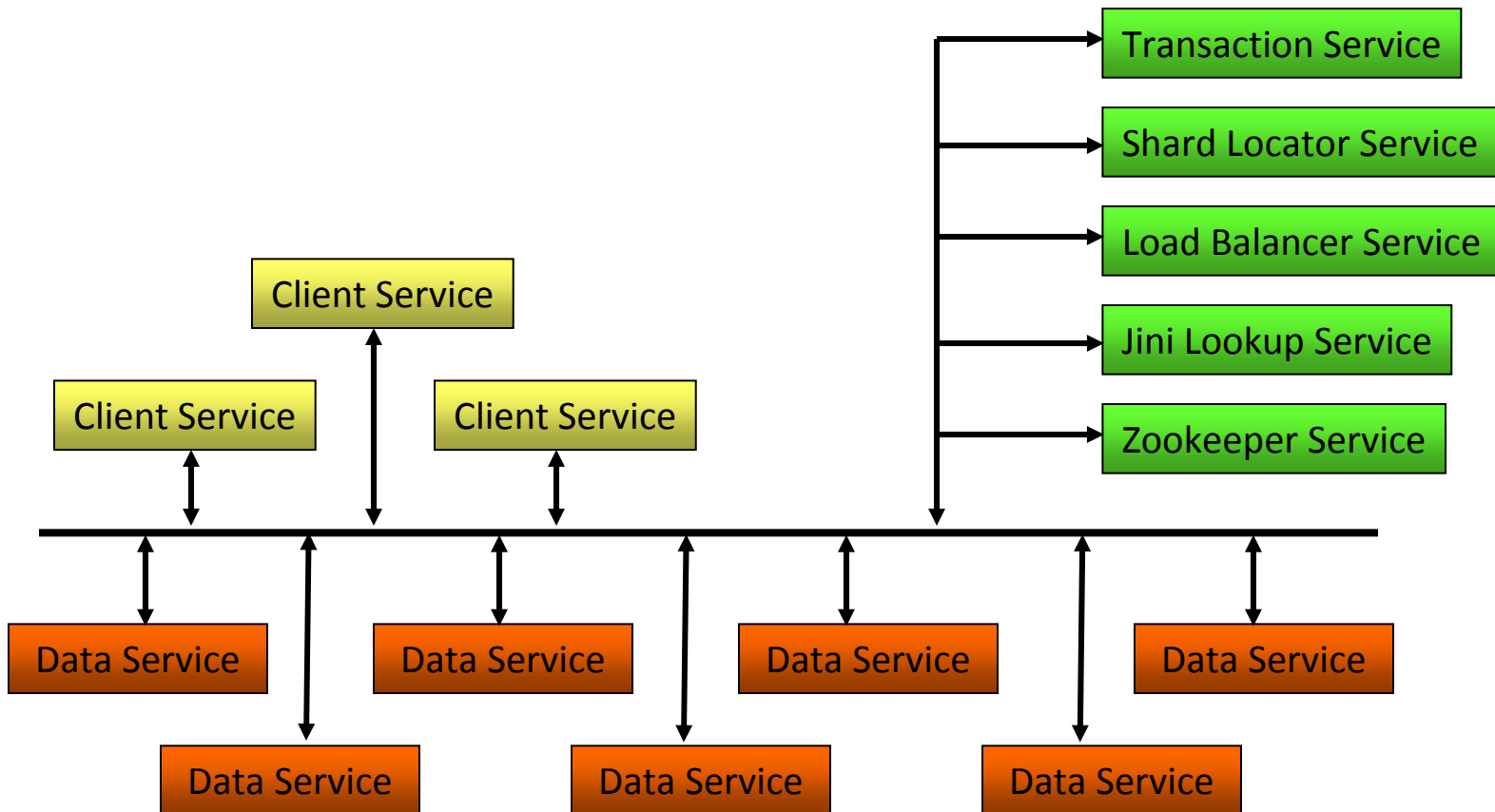
# Persistence Store Mechanisms



- Write Once, Read Many (WORM) store
  - append-only, log structured store
  - used by the data services used to absorb writes
  - aka the “journal”
- Read/Write (RW) store
  - reuses allocation slots on the backing file
  - used by services that need persistence without sharding
- Index segment (SEG) store
  - read-optimized B+Tree files
  - generated by bulk index build operation on a data service

# bigdata<sup>®</sup> Services

a bigdata<sup>®</sup> instance is a federation of services



# Shared Nothing HA Strategy

- Shared nothing means all local resources
  - CPU, memory, disk
- Eliminate all single points of failure
- Build redundancy into the system at a low level
- Quorum model for all persistent services
- Build on existing distributed systems infrastructure components (Jini, Zookeeper)

# Service Failover

- Basic design pattern is write replication across failover chain
- bigdata<sup>®</sup> services grouped into logical instances
- Each logical instance realized by a failover chain of  $k$  physical instances
- Failover chain established in Zookeeper using master election protocol

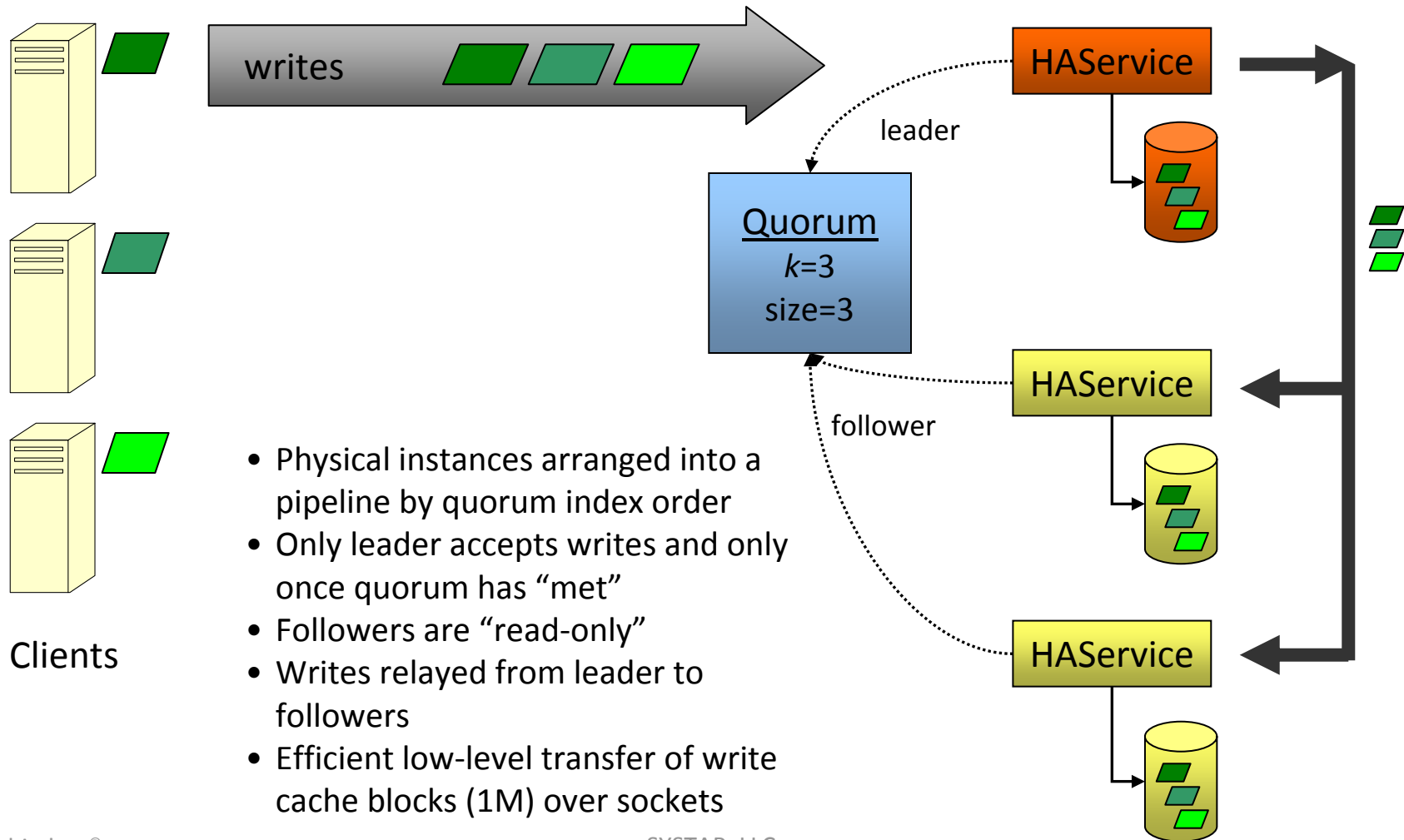
# Quorum Model

- Highly available service has replication factor  $k$ , where  $k$  is an odd integer  $\geq 3$
- Quorum meets when  $(k + 1) / 2$  nodes agree on persistent state (2 of 3, 3 of 5, etc.)
- Agreement formed around *lastCommitTime* in current root block for logical service
  - Concise summary of persistent state of service
- Highly available met quorum has one leader and one or more followers

# Quorum Model

- Quorum model ensures updates cannot be lost following a restart
- If one node of three fails, quorum is still met and accepting writes
- On cluster restart two nodes will agree on persistent state
- Quorum will reform around those two nodes
- Third node will re-synchronize persistent state and join quorum

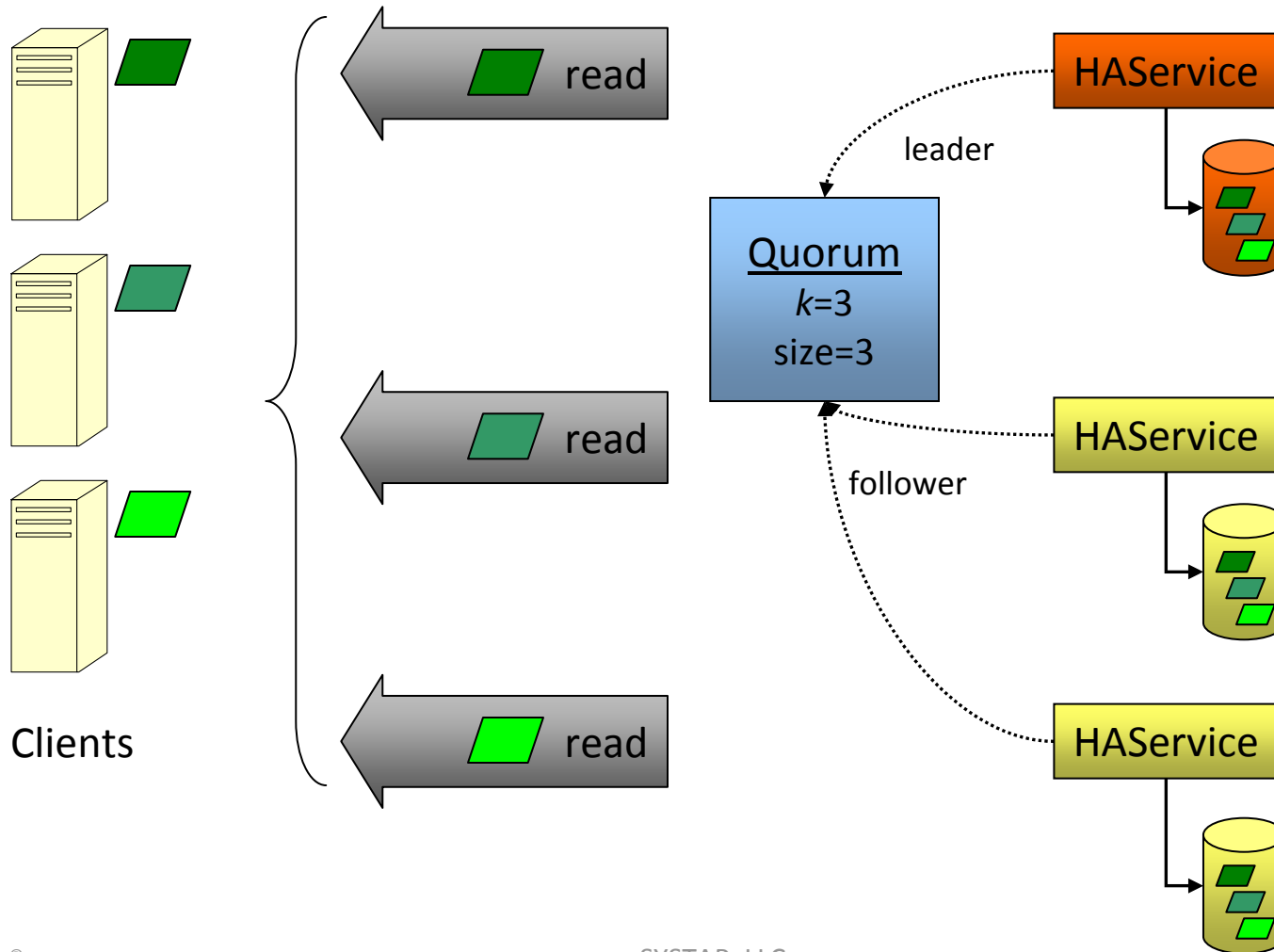
# Write Pipeline



- Physical instances arranged into a pipeline by quorum index order
- Only leader accepts writes and only once quorum has “met”
- Followers are “read-only”
- Writes relayed from leader to followers
- Efficient low-level transfer of write cache blocks (1M) over sockets

Clients

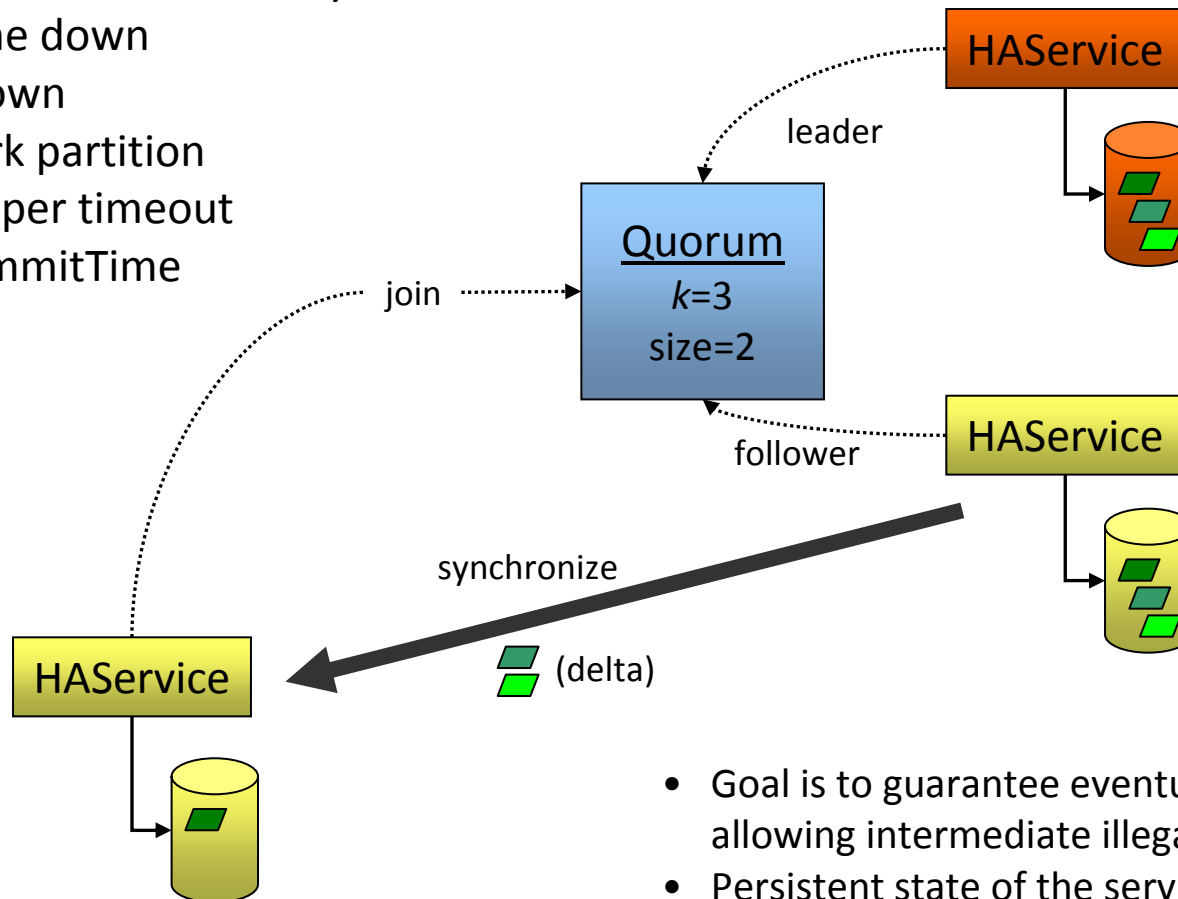
# Write Pipeline



# Synchronization

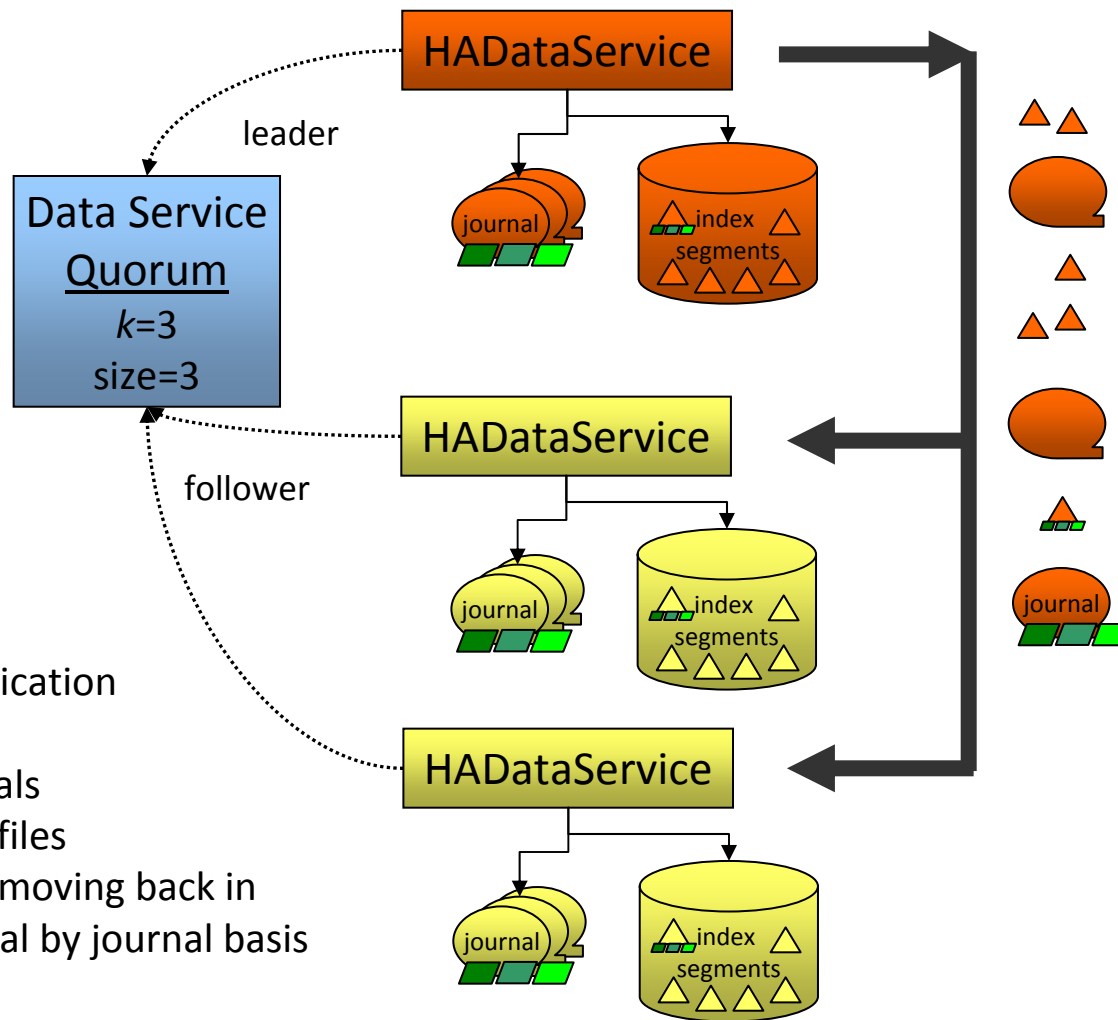
Service can fail for a variety of reasons:

- machine down
- JVM down
- network partition
- zookeeper timeout
- lastCommitTime



- Goal is to guarantee eventual consistency without allowing intermediate illegal states.
- Persistent state of the service must remain self-consistent

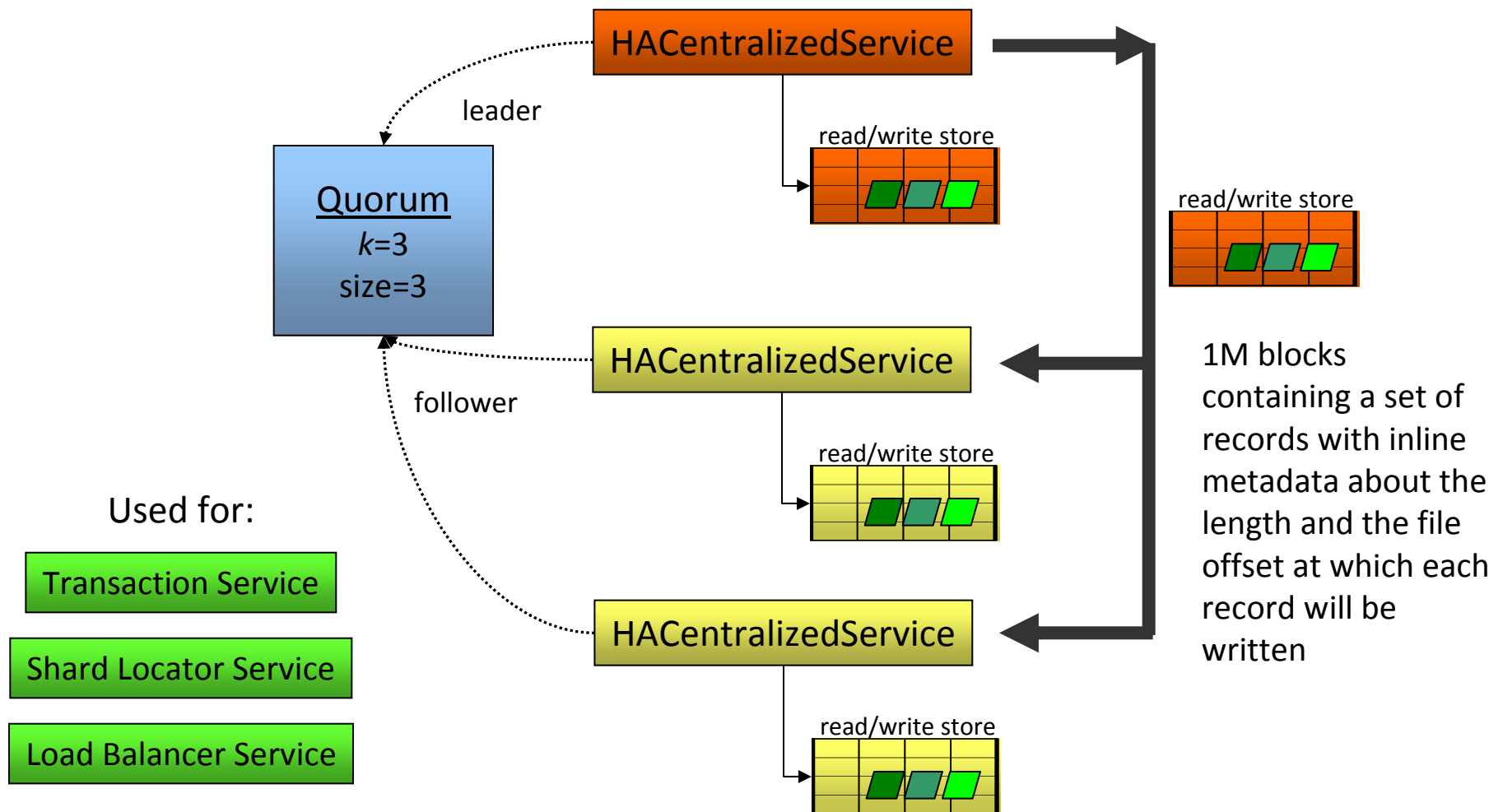
# Data Services



## Data Service Replication

- live journal
- historical journals
- index segment files
- files replicated moving back in time on a journal by journal basis

# Centralized Persistent Services

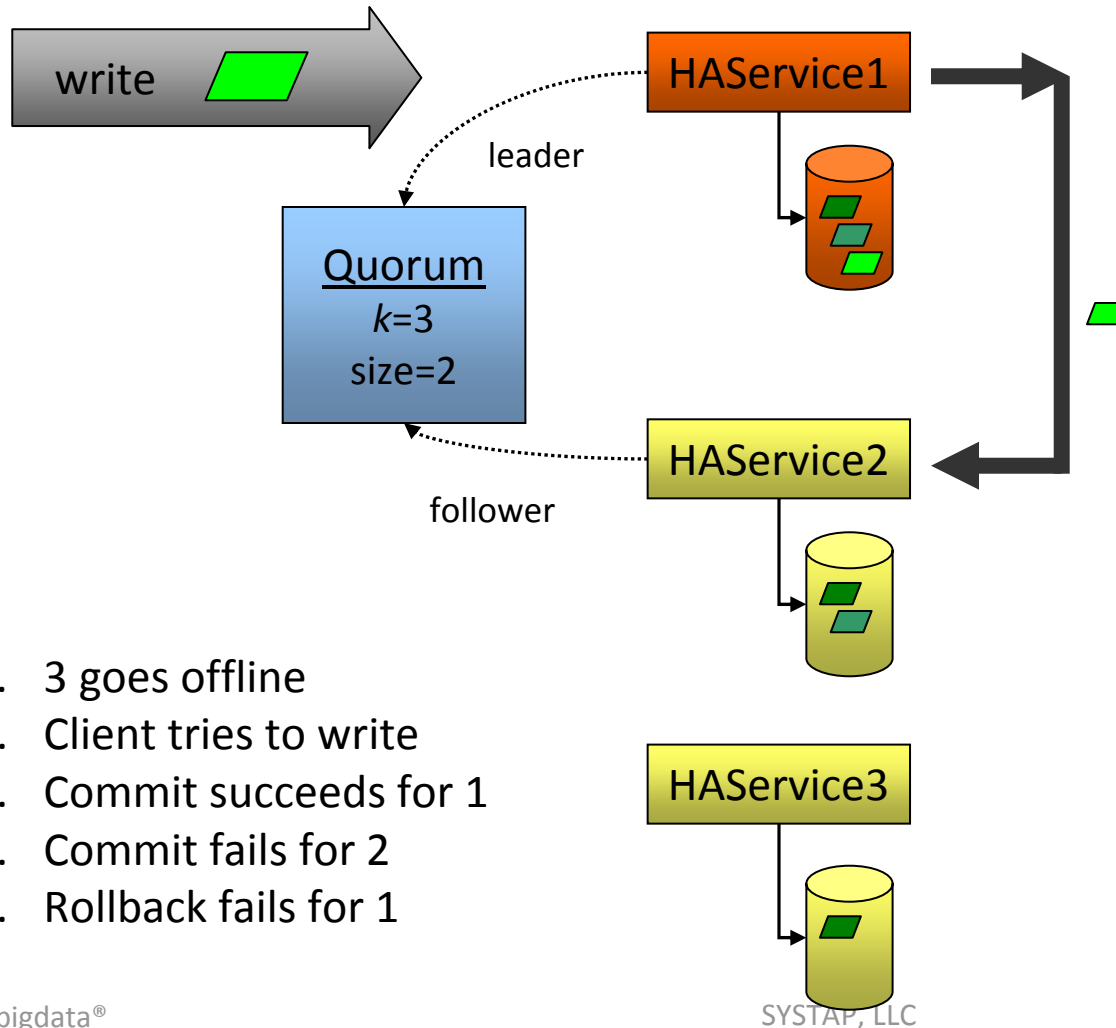


# Failsafe Mechanisms

Failure can occur in a nearly infinite number of ways

- Record level checksums (detect bad reads)
- Block level checksums (detect errors replicating writes)
- Failover reads (handle bad reads gracefully by reading on another node in quorum)
- Quorums (guarantee data consistency)
- Service failover (handle service death)
- Hot spares (handle long-lived node death)

# Quorum Can Not Meet



1. 3 goes offline
2. Client tries to write
3. Commit succeeds for 1
4. Commit fails for 2
5. Rollback fails for 1

- Client never notified of successful commit
- No agreement on persistent state
- Requires operator intervention
- Operator should force 1 and 3 to synch to 2

# Hot Spares

- HA cluster should be provisioned with hot spares
  - Location aware pre-allocation policy (e.g. one per rack)
- Hot spares recruited as necessary
  - Replace failed or failing nodes
  - Temporarily increase replication count
- Requires re-synchronization of all data
  - Heavy operation

# Robust Messaging

- Provide robust operations in the face of transient and long-lived failover events
- Clients must respond appropriately to quorum state change exceptions:

1. Trap exceptions
2. Inspect zookeeper to determine new leader (writes) or node with greatest shard affinity (reads)
3. Re-issue operations

- Encapsulate robust messaging using smart proxy pattern

# Point In Time Recovery

- bigdata<sup>®</sup> retains historical state
- Historical retention period configured via transaction service
- Clients can read from historical commit points using read-only transactions
- History window allows point in time rollback

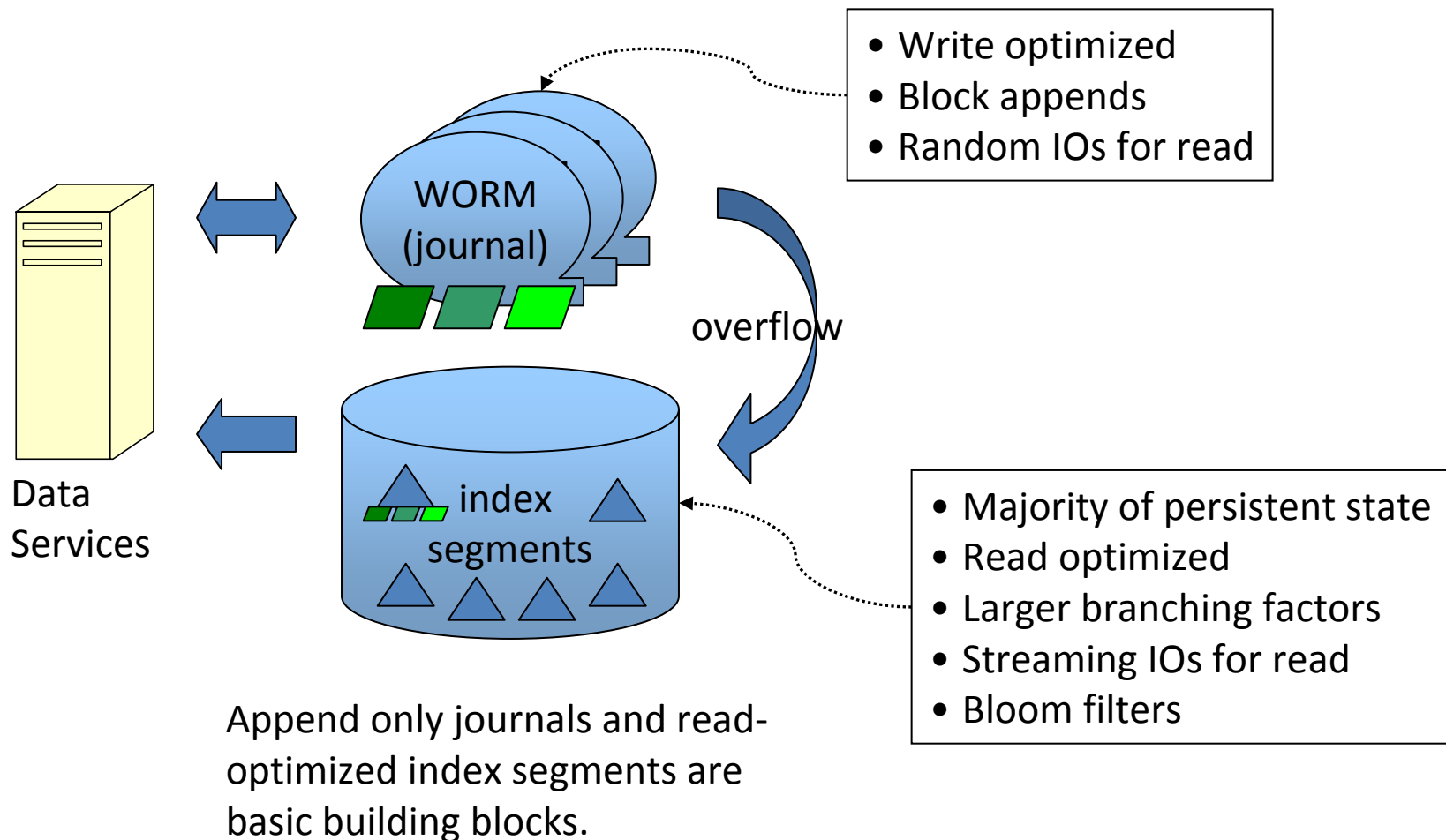
# Immortal Data and Arbitrary Compute Concurrency

Shared Disk HA Architecture

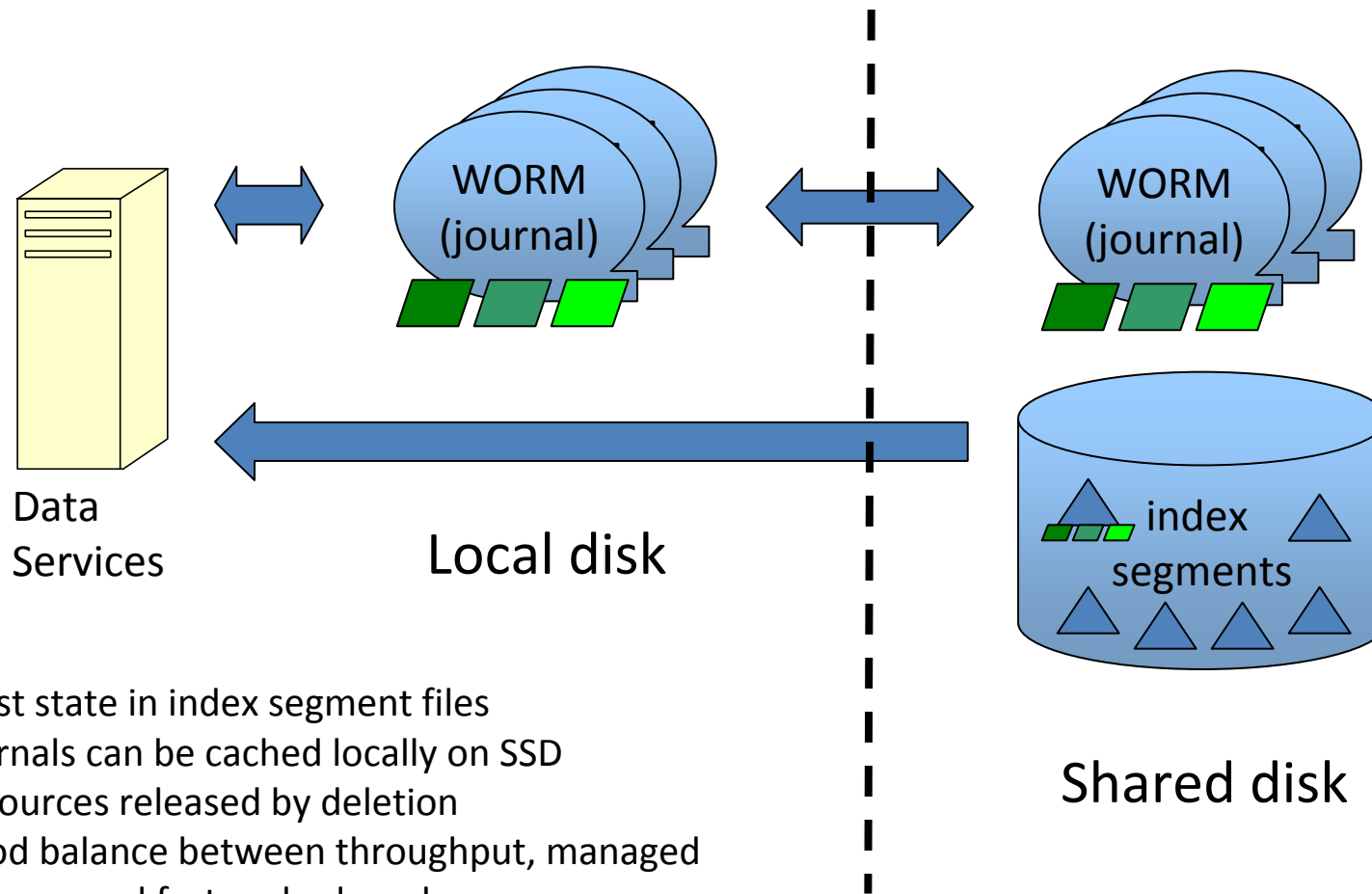
# Shared Disk HA Strategy

- Future direction, currently in design stage
- Persistent state centralized to inherently robust managed storage
  - Parallel file system, SAN, NAS
- Allow retention of infinite historical state
- Pools of compute resources can be dynamically allocated to specific expensive operations
  - Inference
  - Long running queries

# Data Service Persistent State



# Data Service Persistent State



- Most state in index segment files
- Journals can be cached locally on SSD
- Resources released by deletion
- Good balance between throughput, managed storage, and fast cached reads

# Benefits of Shared Disk

- History window no longer limited by local disk capacity
  - Disk resources can grow independently of compute resources
- Load balancing no longer requires shard moves
  - Shed/take pattern
- Finer grained control over resource allocation
  - No longer at the mercy of only using the nodes housing the shards you happen to need

# The Big Picture

- Multi-tenant highly-scalable highly-available RDF database
- Suitable for:
  - Systems of record
  - Web-scale services
  - Lateral integration of services
  - Online and data center workloads
  - Very high throughput