facebook

facebook

Intro to NoSQL Database Algorithms

Nicolas Spiegelberg
Software Engineer, Facebook

QCon Hangzhou,
October 28th, 2012

Agenda

1	Deconstructing a Database
2	Coordination Algorithms
3	Persistence Options
4	Storage Engine
5	Parting Thoughts

What’s in a database?

In the beginning… MySQL

- Query Layer for Data Normalization
- Fits on Single Machine
- Read-dominated
- Stability using Custom Hardware

Shard Manager :: MySQL :: InnoDB :: EXT

Now… NoSQL

New Use Cases: Internet & Data Analytics

- Data **DE**Normalization
- Fits on ~~a Single~~ **100/1000+** Machine(s)
- ~~Read~~ **Write** Dominated
- Stability using Custom ~~Hardware~~ **Software**

Thin Client :: HBase :: HDFS

New Problems…

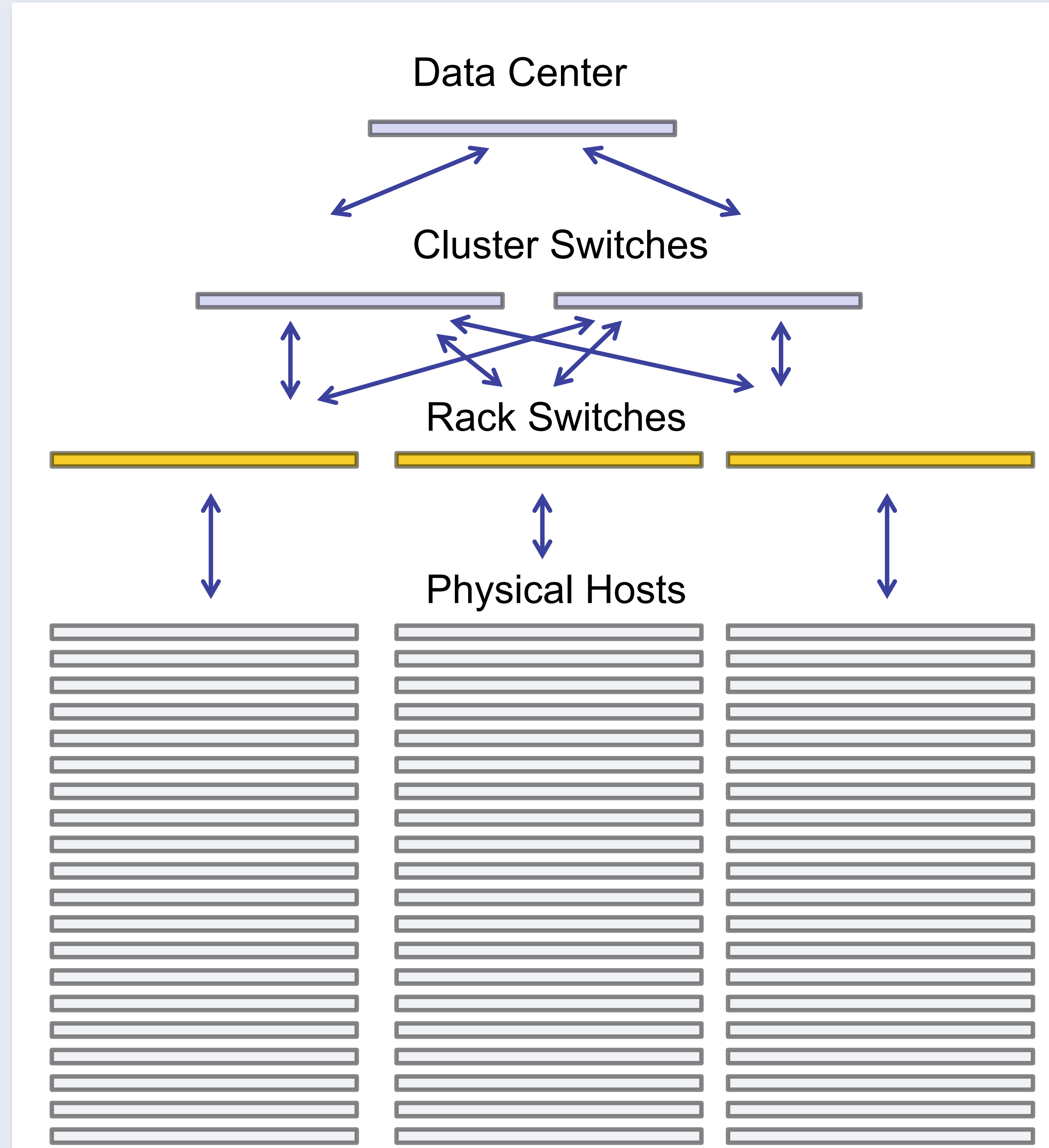
1. Data Denormalization => ~~SQL~~ *NO! (well… kinda)*
2. Fits on 1000+ Machines => Coordination Algorithms
3. R/W Flexibility => Storage Engine
4. Stability via Software => Persistence Options

Coordination Algorithms

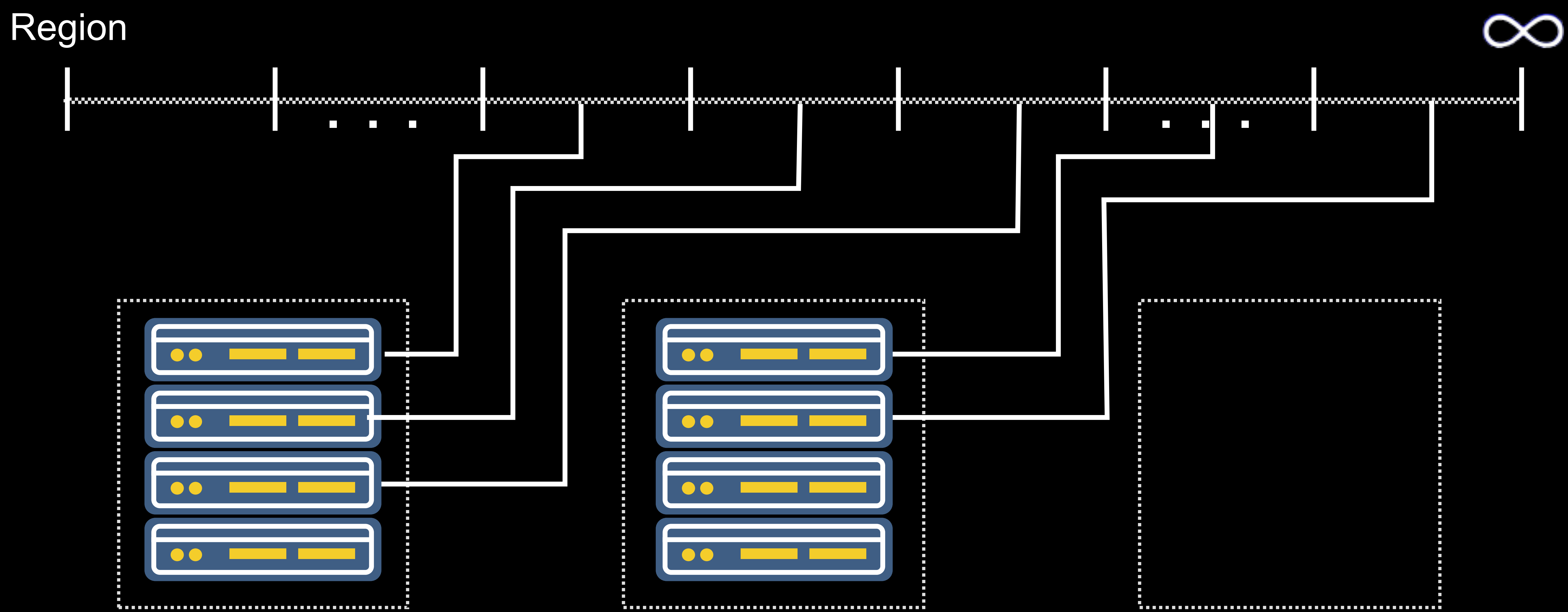
Network Topology

Typical layout

- Physical hosts
- Rack switch
- Cluster switch
- Data center



Sharding: Horizontal Scalability



Sharding Creation

1. Do not manually handle splits
   - Also in original Cassandra

2. Pre-split table on startup
   - Shards = servers^2 / rack

3. Default to MD5 Prefixing
   - row => md5(row) + row
   - harder to cross-row scan

Shard Assignment

Master/Slave

Maintain a shard -> server[] map

+ Placement Control

+ Easy to reason with bugs

+ Locality on Splits

- Separate process, more code

Distributed Hash Table

Hash ring map based on servers

+ Simpler

+ Decentralized

- Large rebalance on split/death

- No control

Shard Assignment (cont)

HBase

- Started out with randomly assigned map
- Tried a couple complicated algorithms: Munkres
- Switched to a Controlled DHT
  - h[0] = hash(shard_name)   pos[0] = h[0] % servers
  - h[1] = hash(h[0])   pos[1] = h[1] % servers

Failure Management

Server Death

- Log Splitting
  - Send 1 log to every server on the rack
- IO Fencing
  - In Paxos, achieved by Quorum Requirement
  - In M/S, achieved by independent failure domains (HDFS/ZK)
- Client Side Multiplexing

Persistence Options

Shard Replication

Where should it be handled?

- Kernel-level : MySQL
- File-level : HDFS/HBase
- Database-level : Cassandra
- Datacenter-level : Spanner

What do you mean by replicated?

- in-memory
- fsync

Shard Replication (cont)

How consistent?

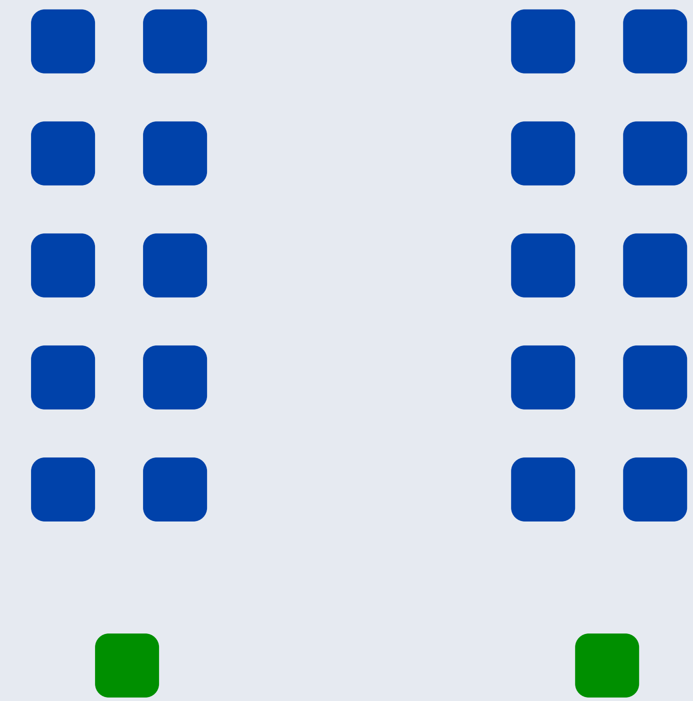
- Strict: Pipeline
- Quorum: Paxos
- Loose: R + W < N

How many copies?

- 2x (MySQL)
- 3x (HBase)
- 2.2x (HDFS Raid)

HDFS Raid

- Stripe Every N files by Parity
- Requires N files of similar [start,end]



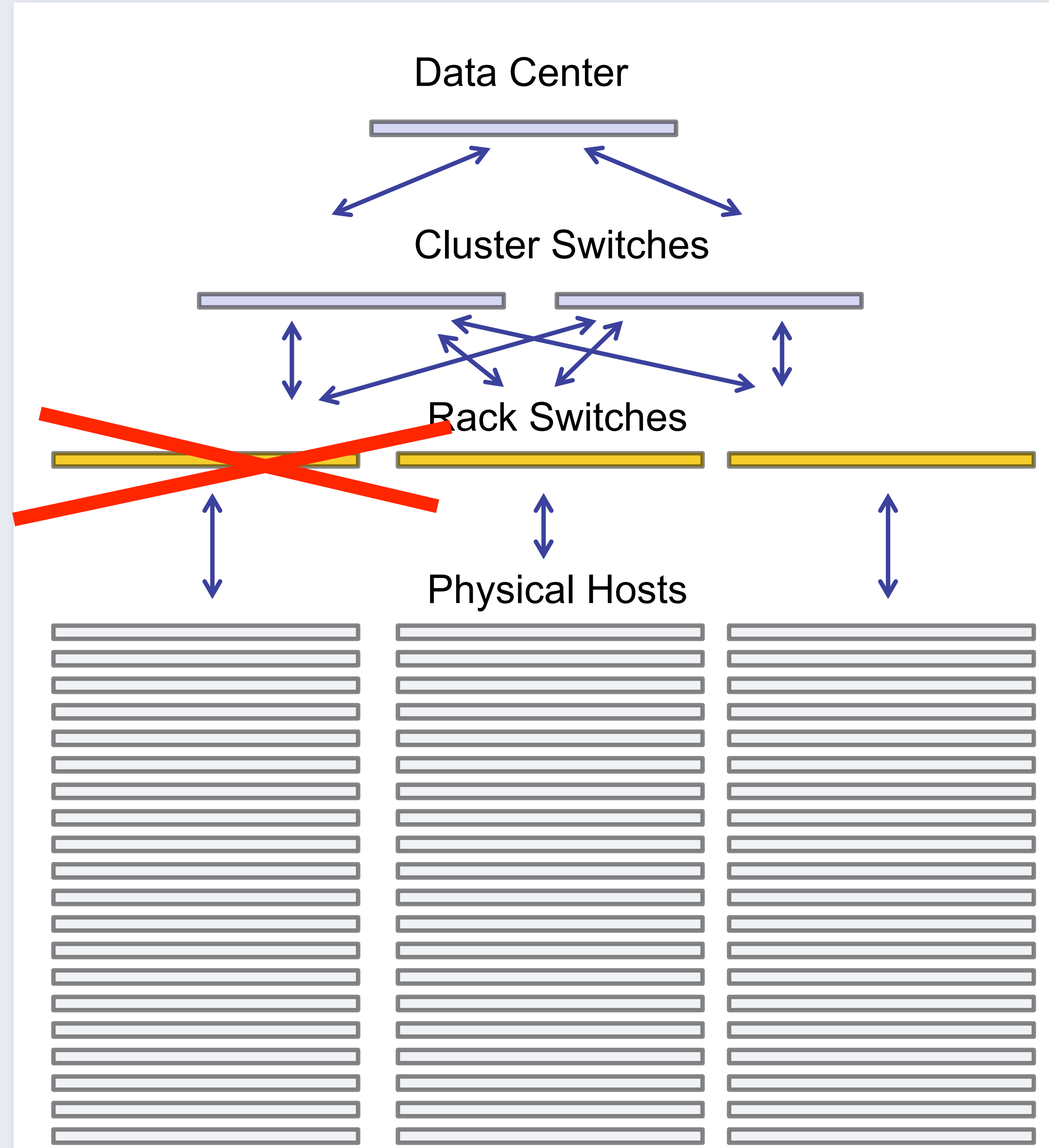
Theory: use oldest files in LSMT

- meet this criteria
- in most cases, are the largest
- should achieve this in active state

Rack Switch Failure

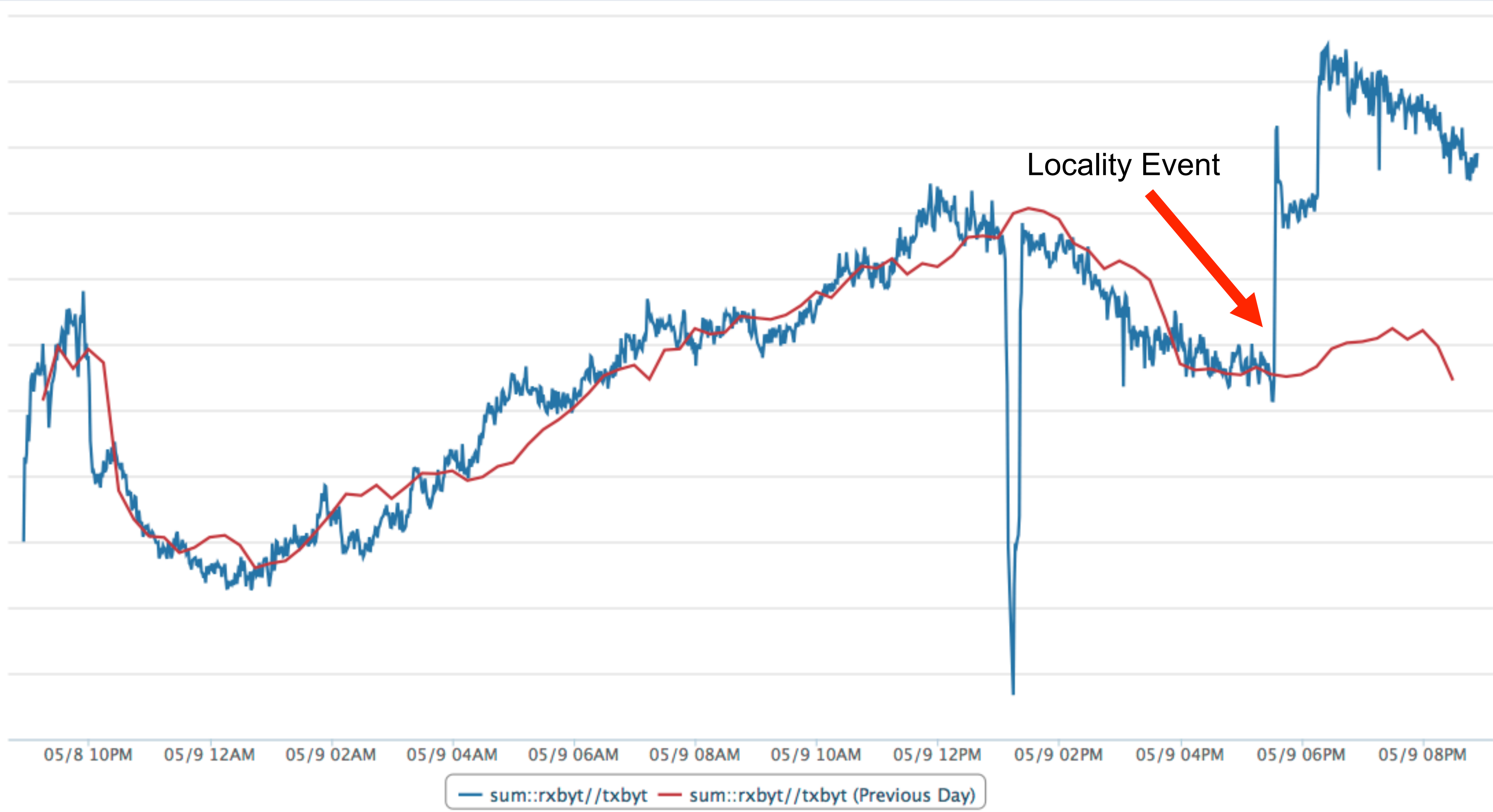
The Problem

- Sometimes rack switches die
- Master reassigns new regions
  - Where?
  - Why?



Locality

Effect on network traffic



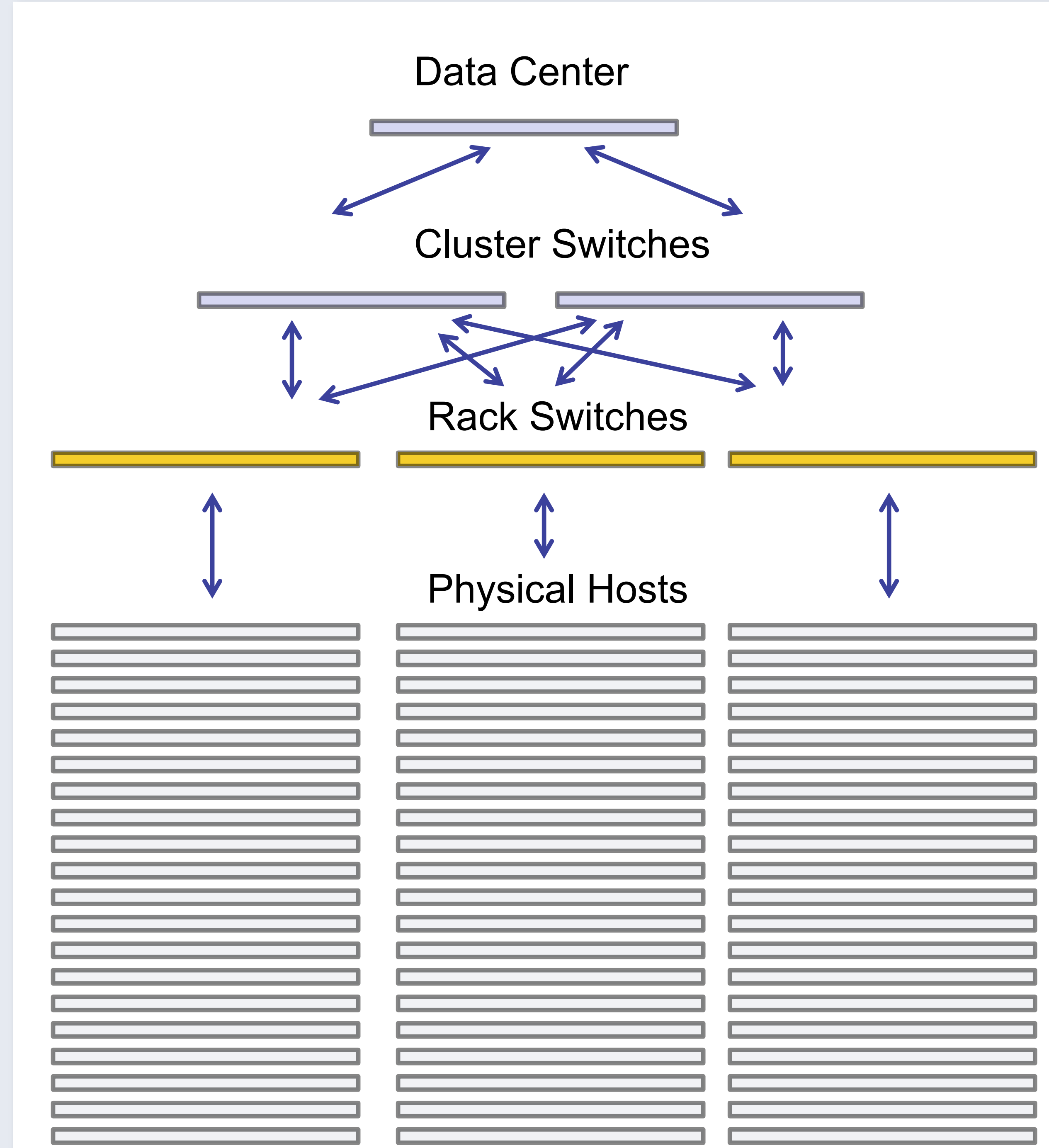
Block Placement

Problem:

- Making 3 copies of the data
- Placing them in the cluster

Original Algorithm:

- Local + rand() + rand()
- Meant for MR, so each file could be scattered across every node



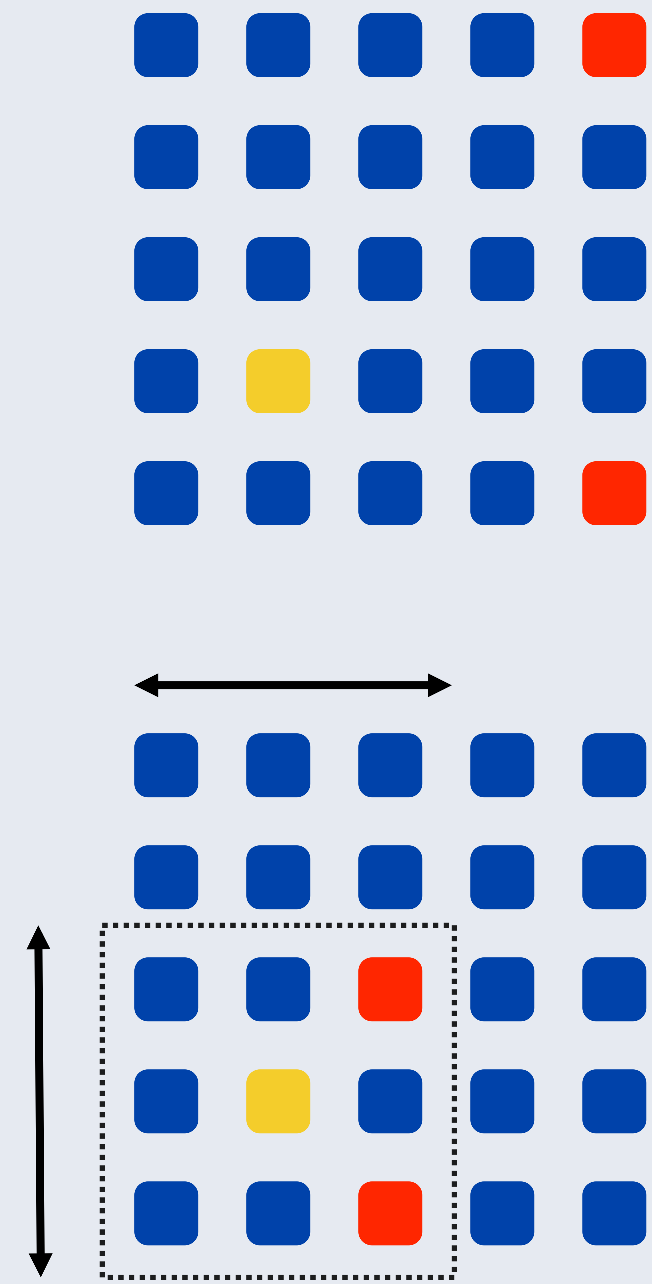
Grid Placement

Grid Placement window

- local + 2 other rack

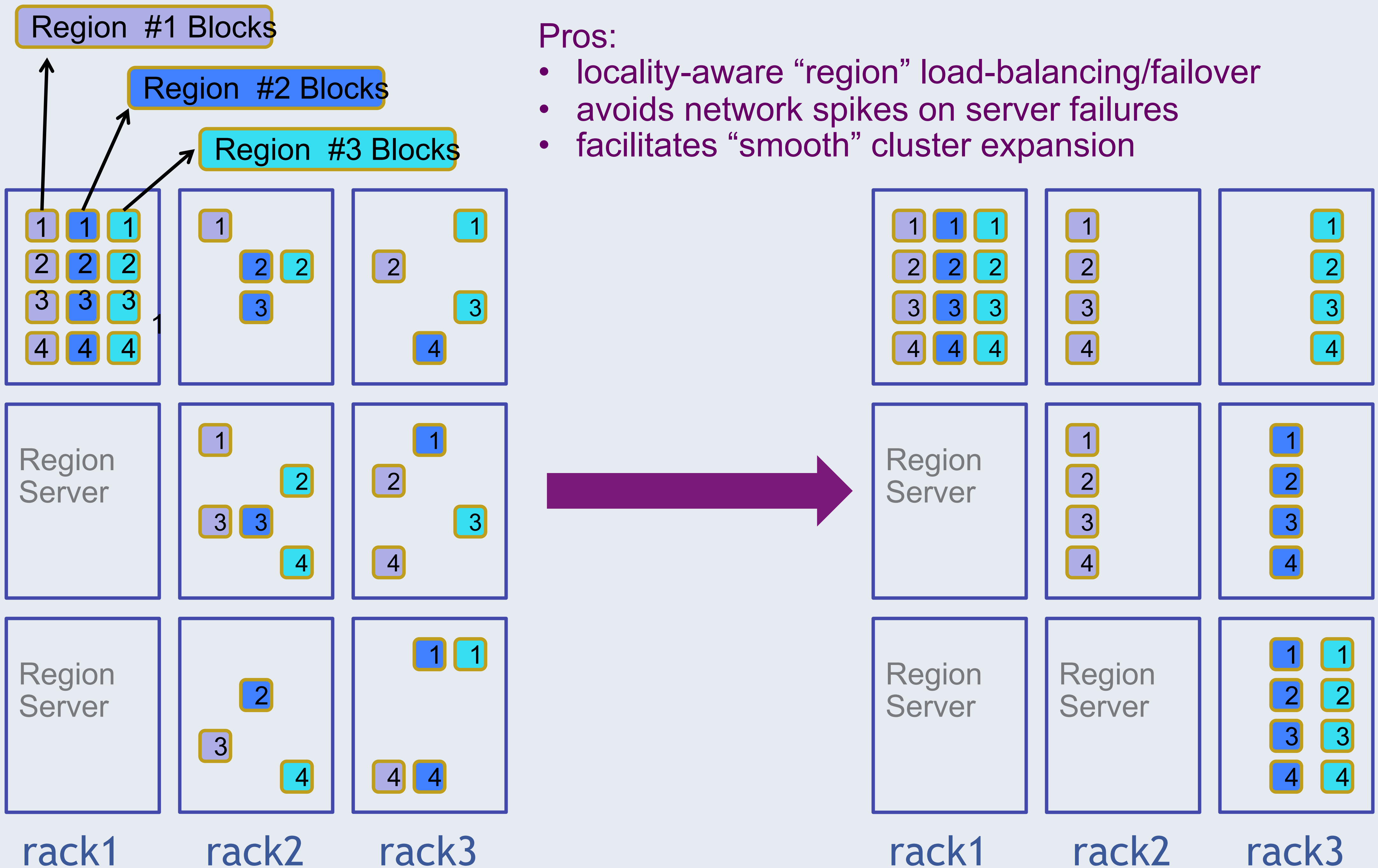
Stats:

- p = 0.0001
- Default: 6.46171e-08
- New (x=1, y=2): 1.88544e-09
- 30X improvement!



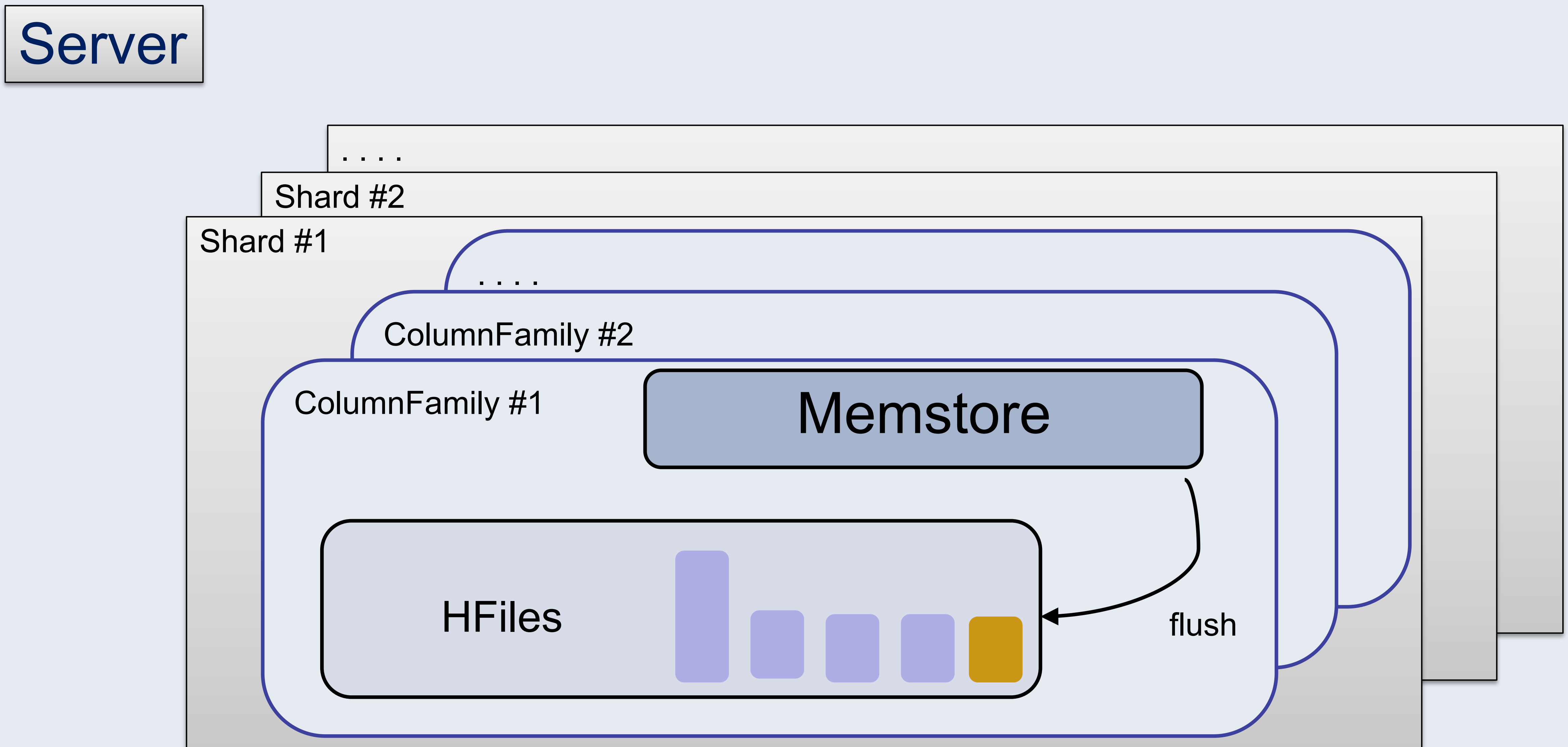
Per-Region Hash Ring Placement

Pros:
- locality-aware “region” load-balancing/failover
- avoids network spikes on server failures
- facilitates “smooth” cluster expansion



Storage Engine

Log Structured Merge Tree



Data in HFile is sorted; has block index for efficient retrieval

About LSMT

Write Algorithms are relatively-trivial

- Write new, immutable file
- Avoid stalls

Read Algorithms are varied

- Block Index
- Bloom Filter
- Time Range Filters
- Compaction

Block Index

Purpose

- Data stored in “Blocks”, which is ~ optimal disk read
- Shard contents within a file, based on block
- Avoids unnecessary seeks around the block

Bloom Filter

About

- Cheap point query
- Make a Hash of every Row or Row+Col (32 bits/entry)
- Set bits instead of using full Hash (~8 bits/entry)
  - This makes false positives possible, but probabilistically bound
  - Need to use a hash ring to manage probability

for i in [0,n]: array[Hash[i] % bloom.size()] = 1

Bloom Filter

Optimizations

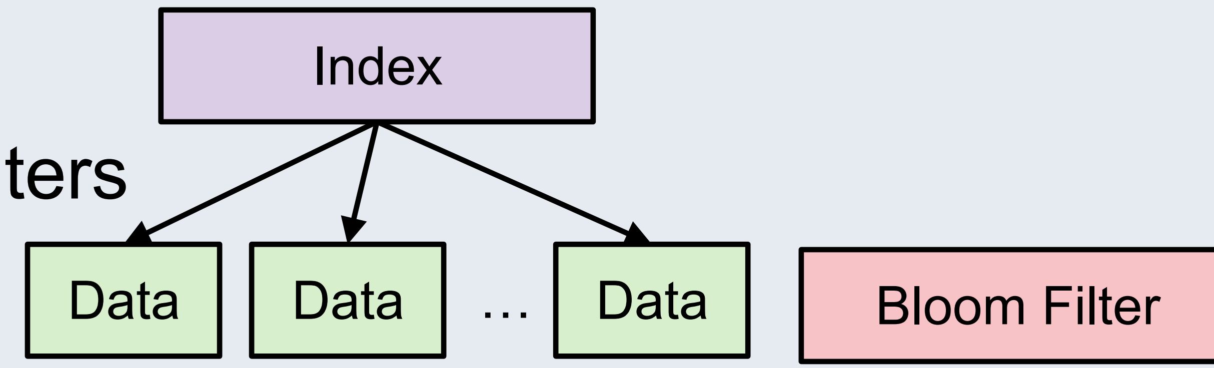
- Combinatorial Hashing
  - Hashing (Murmur, Jenkins) is a big CPU expense
  - Instead of N different Hashes: Hash[0] + N * Hash[1]
- Folding
  - If we oversize our bloom array, we can shrink it if size % 2 = 0

    (Both N % 100 == X && N % 100 == X + 50 map to the same new location)
- Sharding
  - Treat blooms like block index & have multiple per file

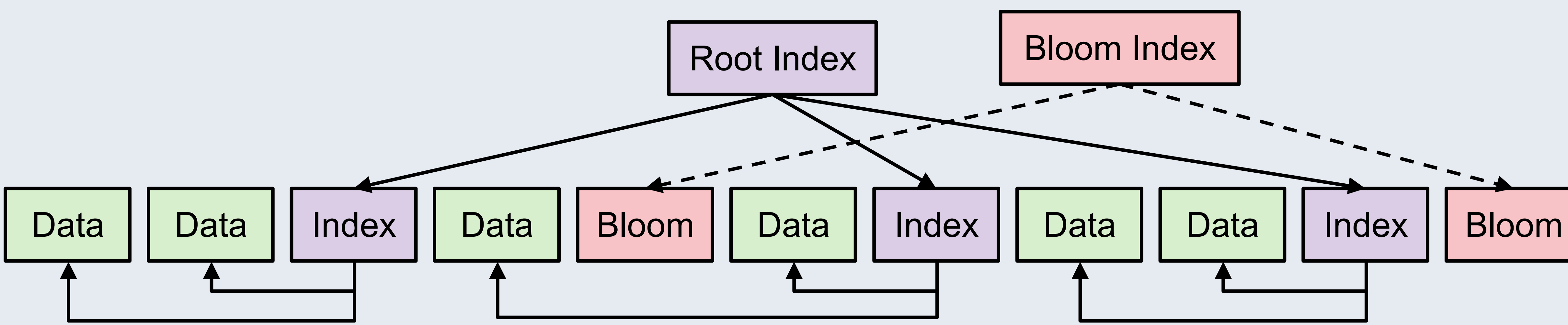
Optimizing HBase File Format

Block Index and Bloom Filter Shards are Stored Inline

- **HFile v1**
  - Arbitrarily large indexes, Bloom filters
  - Bloom filter loaded on 1st access

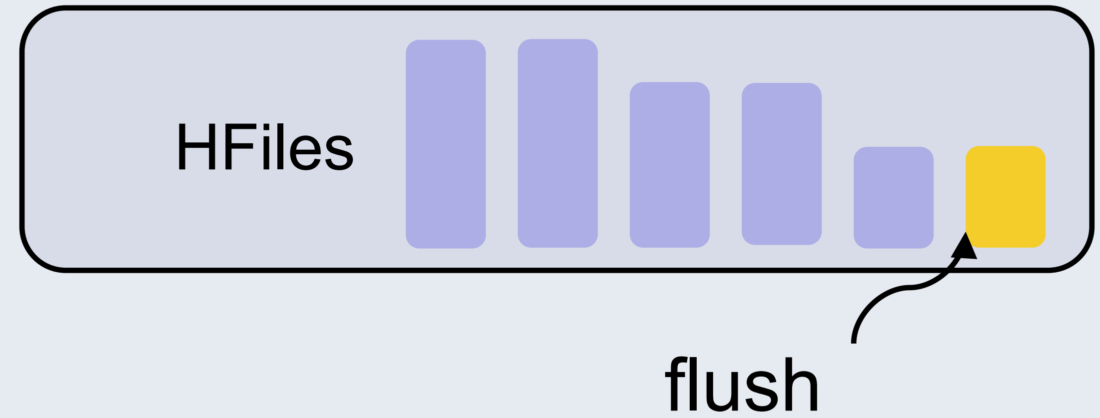


- **HFile v2** (in production since Fall 2011)

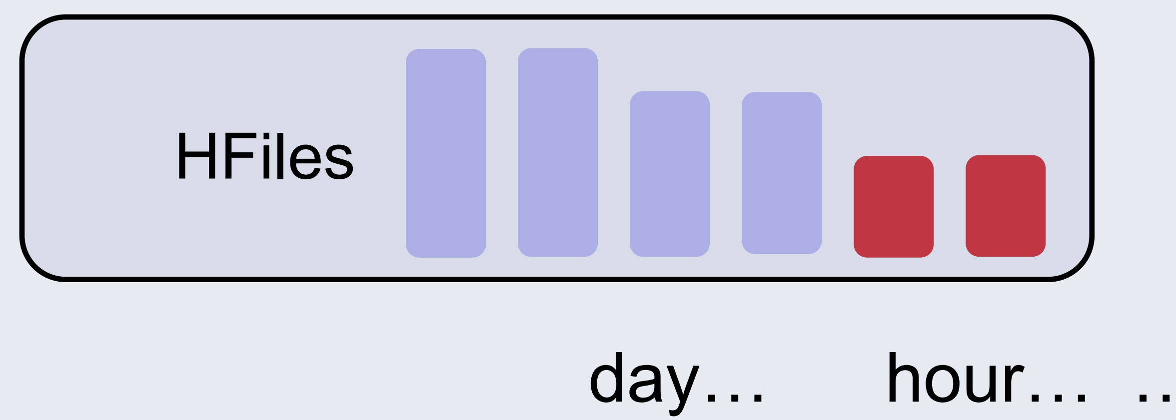


Time Range Filters

- Log-structured Merge Tree



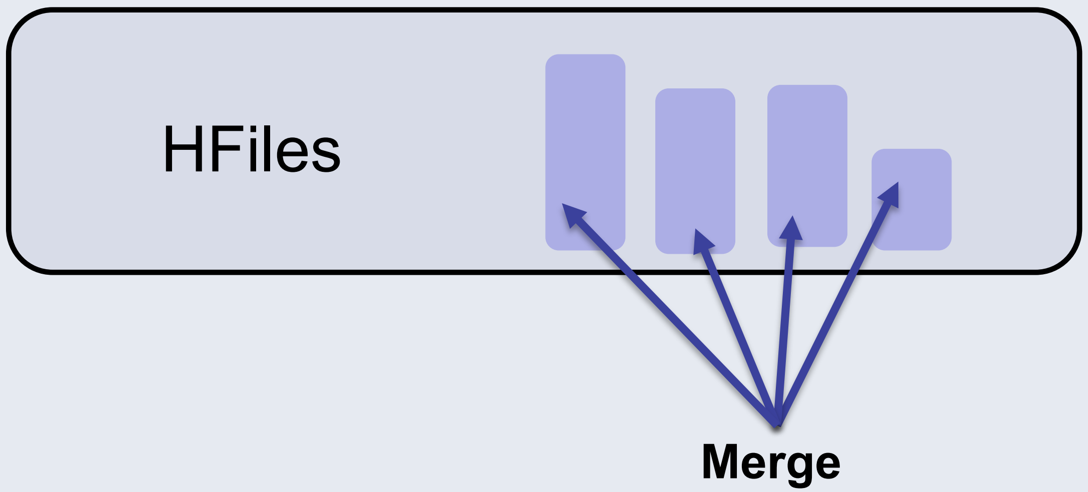
  - Time-ordered Data Storage!
    - Time-series data optimized
    - Write-biased query optimized
    - Short circuit on Mutations



Compactions: Intro

Critical for Read Performance

- Merge N files
- Reduces read IO when earlier filters don't help enough
- The most complicated part of an LSMT
  - What & when to select



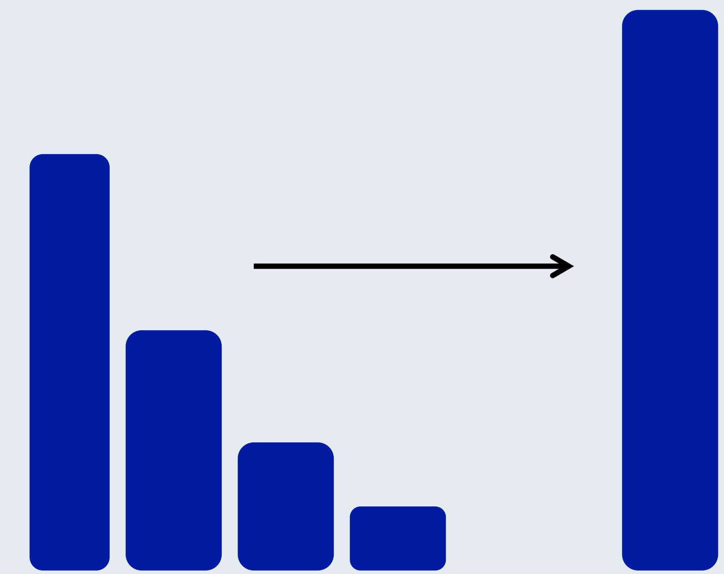
Sigma Compaction

Default algorithm in HBase 0.90

#1. File selection based on summation of sizes. Σ

size[i] < (size[0] + size[1] + …size[i-1]) * C

#2. Compact only if at least N eligible files found.



+ trivial implementation

+ minimal overwrites

- non-deterministic latency

- files have variable lifetime

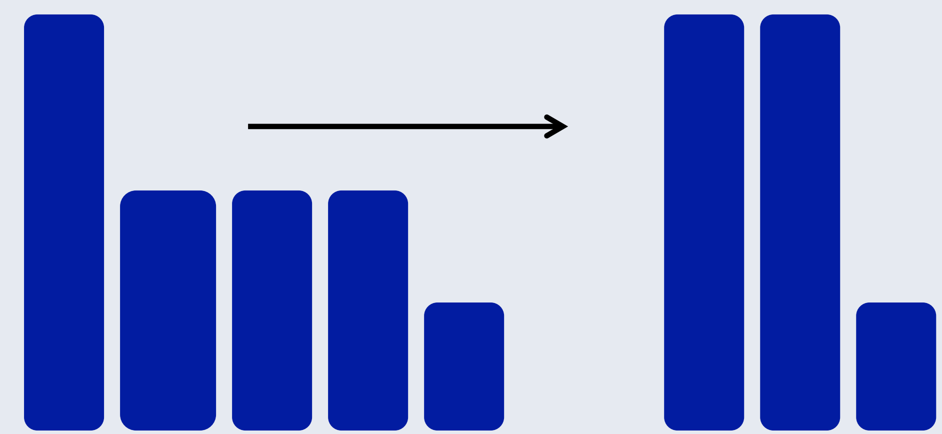
- no incremental compaction benefit

Tiered Compaction

Default algorithm in BigTable/HBase

#1. File selection based on size relative to a pivot:

size[i] * C >= size[p] <= size[k] / C :: i < p < k



#2. Compact only if at least N eligible files found.

(groups files into “tiers”)

+ trivial implementation
+ more deterministic behavior
+ medium size files are warm

- more files seeks necessary
- not good for read-heavy workload
- no incremental compaction benefit

Leveled Compaction

Default algorithm in LevelDB

#1. Bucket into tiers of magnitude difference (~10x)

#2. Shard the compaction across files (not just block index)

#3. Only the shard that goes over a certain size

+ optimized for read-heavy use
+ faster compaction turnaround
+ easy to cache-on-compact

- complicated algorithm
- heavy rewrites on write-dominated use
- time range filters less effective

Parting Thoughts

Material Covered

1. **Coordination Algorithms**
    1. Sharding Selection & Placement
    2. Server Recovery
2. **Persistence Options**
    1. Replication Options
    2. Block Placement
3. **Storage Engine**
    1. Filters: Block Indices, Bloom Filters, & Time Range
    2. Compactions

Material “*I wished I could cover*”

1. **Coordination Algorithms**
   1. Paxos in-depth
   2. Read-repair
2. **Persistence Options**
   1. Compression: Delta-encoding, Columnar Storage, LZO-GZ tiers
   2. Backup/Replication
3. **Storage Engine**
   1. Delete Blooms
   2. Lazy Seek

Thought about Databases

- The underlying concepts are simple
- You keep coming back to the same handful of metrics
- The fun part: you must continually look at them in a different light
  - This is what takes Databases so long to build
  - It's also why NoSQL DBs are still young

A mature database has 1000+ features, you can only add 1 at a time…
CHOOSE WISELY

facebook